Hubert Baumeister
Michele Marchesi
Mike Holcombe (Eds.)

LNCS 3556

Extreme Programming and Agile Processes in Software Engineering

6th International Conference, XP 2005
Sheffield, UK, June 2005
Proceedings

Lecture Notes in Computer Science 3556

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
New York University, NY, USA

Doug Tygar
University of California, Berkeley, CA, USA

Moshe Y. Vardi
Rice University, Houston, TX, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Hubert Baumeister Michele Marchesi
Mike Holcombe (Eds.)

Extreme Programming and Agile Processes in Software Engineering

6th International Conference, XP 2005
Sheffield, UK, June 18-23, 2005
Proceedings

Springer

Volume Editors

Hubert Baumeister
Ludwig-Maximilians-Universität München
Institut für Informatik
Oettingenstr. 67, 80538 München, Germany
E-mail: baumeist@informatik.uni-muenchen.de

Michele Marchesi
University of Cagliari
DIEE, Department of Electrical and Electronic Engineering
Piazza d'Armi, 09123 Cagliari, Italy
E-mail: michele@diee.unica.it

Mike Holcombe
University of Sheffield, Department of Computer Science
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK
E-mail: m.holcombe@dcs.shef.ac.uk

Library of Congress Control Number: 2005927234

CR Subject Classification (1998): D.2, D.1, D.3, K.6.3, K.6, K.4.3, F.3

ISSN 0302-9743
ISBN-10 3-540-26277-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-26277-0 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11499053 06/3142 5 4 3 2 1 0

Preface

Extreme Programming has come a long way since its first use in the C3 project almost 10 years ago. Agile methods have found their way into the mainstream, and at the end of last year we saw the second edition of Kent Beck's book on Extreme Programming, containing a major refactoring of XP.

This year, the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering took place June 18–23 in Sheffield. As in the years before, XP 2005 provided a unique forum for industry and academic professionals to discuss their needs and ideas on Extreme Programming and agile methodologies. These proceedings reflect the activities during the conference which ranged from presentation of research papers, invited talks, posters and demonstrations, panels and activity sessions, to tutorials and workshops. Included are also papers from the Ph.D. and Master's Symposium which provided a forum for young researchers to present their results and to get feedback.

As varied as the activities were the topics of the conference which covered the presentation of new and improved practices, empirical studies, experience reports and case studies, and last but not least the social aspects of agile methods.

The papers and the activities went through a rigorous reviewing process. Each paper was reviewed by at least three Program Committee members and was discussed carefully among the Program Committee. Of 62 papers submitted, only 22 were accepted as full papers.

We would like to sincerely thank the several chairs and the members of the Program Committee for their thorough reviews and dedicated involvement in shaping the contents of the conference. We would also like to thank the authors, the workshop and activity leaders, the tutorial speakers, the panelists, those who served on the various committees, our sponsors, those who offered their experience of running previous XP conferences, the staff of Sheffield University and, last but not least, everyone who attended.

April 2005

Hubert Baumeister
Michele Marchesi
Mike Holcombe

Organization

XP 2005 was organized by Sheffield University.

Executive Committee

Conference Chair	Michele Marchesi (Italy)
Local Chair	Mike Holcombe (UK)
Program Chair	Hubert Baumeister (Germany)
Tutorials Co-chairs	Geoffrey Bache (Sweden)
	Emily Bache (Sweden)
Workshops Chair	Vera Peeters (Belgium)
Panel Co-chairs	David Hussman (USA)
	David Putman (UK)
Ph.D. Symposium Chair	Sandro Pinna (Italy)
Posters Chair	Barbara Russo (Italy)
Sponsorship Chair	Steven Fraser (USA)
Communications Chair	Erik Lundh (Sweden)
Social Activities Chair	Nicolai Josuttis (Germany)

Program Committee

Alberto Sillitti (Italy)
Ann Anderson (USA)
Barbara Russo (Italy)
Bernhard Rumpe (Germany)
Charlie Poole (USA)
Chet Hendrickson (USA)
Daniel Karlström (Sweden)
David Hussman (USA)
Diana Larsen (USA)
Dierk König (Switzerland)
Don Wells (USA)
Erik Lundh (Sweden)
Francesco Cirillo (Italy)
Frank Westphal (Germany)
Giancarlo Succi (Italy)
Helen Sharp (UK)
Jim Highsmith (USA)
Joe Bergin (USA)
John Favaro (Italy)
Joshua Kerievsky (USA)
Jutta Eckstein (Germany)
Laurent Bossavit (France)
Laurie Williams (USA)
Linda Rising (USA)
Marco Abis (Italy)
Martin Lippert (Germany)
Mary Poppendieck (USA)
Michael Hill (USA)
Nicolai Josuttis (Germany)
Paul Grünbacher (Austria)
Pekka Abrahamsson (Finland)
Rachel Davis (UK)
Rick Mugridge (New Zealand)
Ron Jeffries (USA)
Sandro Pinna (Italy)
Scott W. Ambler (USA)
Sian Hopes (UK)
Steve Freeman (UK)
Steven Fraser (USA)
Till Schümmer (Germany)
Tim Mackinnon (UK)
Vera Peeters (Belgium)
Ward Cunningham (USA)
Yael Dubinsky (Israel)

Referees

Michael Barth
Phil McMinn
Tom Poppendieck
Greg Utas

Table of Contents

Experience Reports

Lean Software Management Case Study: Timberline Inc. 1
Peter Middleton, Amy Flaxel, and Ammon Cookson

XP South of the Equator: An eXPerience Implementing XP in Brazil 10
Alexandre Freire da Silva, Fábio Kon, and Cicero Torteli

Introducing Extreme Programming into a Software Project at the Israeli Air Force 19
Yael Dubinsky, Orit Hazzan, and Arie Keren

The Agile Journey – Adopting XP in a Large Financial Services Organization 28
Jeff Nielsen and Dave McMunn

New Insights

From User Stories to Code in One Day? 38
Michał Śmiałek

Evaluate XP Effectiveness Using Simulation Modeling 48
Alessandra Cau, Giulio Concas, Marco Melis, and Ivana Turnu

Agile Security Using an Incremental Security Architecture 57
Howard Chivers, Richard F. Paige, and Xiaocheng Ge

Quantifying Requirements Risk 66
Fred Tingey

Social Issues

Social Perspective of Software Development Methods: The Case of the Prisoner Dilemma and Extreme Programming 74
Orit Hazzan and Yael Dubinsky

A Framework for Understanding the Factors Influencing Pair Programming Success 82
Mustafa Ally, Fiona Darroch, and Mark Toleman

Empirical Study on the Productivity of the Pair Programming 92
Gerardo Canfora, Aniello Cimitile, and Corrado Aaron Visaggio

The Social Side of Technical Practices 100
Hugh Robinson and Helen Sharp

Testing

A Survey of Test Notations and Tools for Customer Testing . . . 109
Adam Geras, James Miller, Michael Smith, and James Love

Testing with Guarantees and the Failure of Regression Testing in eXtreme Programming . . . 118
Anthony J.H. Simons

Examining Usage Patterns of the FIT Acceptance Testing Framework . . . 127
Kris Read, Grigori Melnik, and Frank Maurer

Agile Test Composition . . . 137
Rick Mugridge and Ward Cunningham

Tools

E-TDD – Embedded Test Driven Development a Tool for Hardware-Software Co-design Projects . . . 145
Michael Smith, Andrew Kwan, Alan Martin, and James Miller

Multi-criteria Detection of Bad Smells in Code with UTA Method . . . 154
Bartosz Walter and Błażej Pietrzak

An Eclipse Plugin to Support Agile Reuse . . . 162
Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick

Case Studies

An Approach for Assessing Suitability of Agile Solutions: A Case Study . . . 171
Minna Pikkarainen and Ulla Passoja

XP Expanded: Distributed Extreme Programming . . . 180
Keith Braithwaite and Tim Joyce

A Case Study on Naked Objects in Agile Software Development . . . 189
Heikki Keränen and Pekka Abrahamsson

Invited Talks

Extreme Programming for Critical Systems? . . . 198
Ian Sommerville

That Elusive Business Value: Some Lessons from the Top . . . 199
John Favaro

Agility – Coming of Age . . . 200
Jutta Eckstein

Another Notch . . . 201
Kent Beck

Posters and Demonstrations

A Process Improvement Framework for XP Based SMEs 202
Muthu Ramachandran

Standardization and Improvement of Processes and Practices Using XP, FDD and RUP in the Systems Information Area of a Mexican Steel Manufacturing Company 206
Luis Carlos Aceves Gutiérrez, Enrique Sebastián Canseco Castro, and Mauricio Ruanova Hurtado

Multithreading and Web Applications: Further Adventures in Acceptance Testing 210
Johan Andersson, Geoff Bache, and Claes Verdoes

Using State Diagrams to Generate Unit Tests for Object-Oriented Systems 214
Florentin Ipate and Mike Holcombe

The Positive Affect of the XP Methodology 218
Sharifah Lailee Syed-Abdullah, John Karn, Mike Holcombe, Tony Cowling, and Marian Gheorge

Adjusting to XP: Observational Studies of Inexperienced Developers 222
John Karn, Tony Cowling, Sharifah Lailee Syed-Abdullah, and Mike Holcombe

An Agile and Extensible Code Generation Framework 226
Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

UC Workbench – A Tool for Writing Use Cases and Generating Mockups 230
Jerzy Nawrocki and Łukasz Olek

Desperately Seeking Metaphor 235
Ben Aveling

Agile Testing of Location Based Services 239
Jiang Yu, Andrew Tappenden, Adam Geras, Michael Smith, and James Miller

Source Code Repositories and Agile Methods 243
Alberto Sillitti and Giancarlo Succi

Writing Coherent User Stories with Tool Support 247
Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak

BPUF: Big Picture Up Front 251
Frank Keenan and David Bustard

Agile Development Environment for Programming and Testing (ADEPT) – Eclipse Makes Project Management eXtreme 255
Mike Holcombe and Bhavnidhi Kalra

Tailoring Agile Methodologies to the Southern African Environment 259
Ernest Mnkandla, Barry Dwolatzky, and Sifiso Mlotshwa

Panels and Activities

XP/Agile Education and Training 263
Angela Martin, Steven Fraser, Rachel Davies, Mike Holcombe, Rick Mugridge, Duncan Pierce, Tom Poppendieck, and Giancarlo Succi

Off-Shore Agile Software Development 267
Steven Fraser, Angela Martin, Mack Adams, Carl Chilley, David Hussman, Mary Poppendieck, and Mark Striebeck

The Music of Agile Software Development 273
Karl Scotland

The XP Game 274
Pascal Van Cauwenberghe, Olivier Lafontan, Ivan Moore, and Vera Peeters

Leadership in Extreme Programming 276
Kent Beck, Fred Tingey, John Nolan, and Steve Freeman

Tutorials

Agile Project Management 277
Ken Schwaber

Expressing Business Rules 278
Rick Mugridge

Introduction to Lean Software Development – Practical Approaches for Applying Lean Principles to Software Development 280
Mary Poppendieck and Tom Poppendieck

The Courage to Communicate: Collaborative Team Skills for XP/Agile Teams 281
Diana Larsen

Test-Driven User Interfaces 285
Charlie Poole

The XP Geography: Mapping Your Next Step, a Guide to Planning Your Journey 287
Kent Beck

Workshops

Lightning Writing Workshop Exchange Ideas on Improving Writing Skills 288
Laurent Bossavit and Emmanuel Gaillot

The Coder's Dojo – A Different Way to Teach and Learn Programming ... 290
Laurent Bossavit and Emmanuel Gaillot

Informative Workspace "Ways to Make a Workspace that Gives Your Team Useful FeedBack" 292
Rachel Davies and Tim Bacon

Exploring Best Practice for XP Acceptance Testing 294
Geoff Bache, Rick Mugridge, and Brian Swan

Hands-on Domain-Driven Acceptance Testing 296
Geoff Bache, Rick Mugridge, and Brian Swan

How to Sell the Idea of XP to Managers, Customers and Peers 299
Jan-Erik Sandberg and Lars Arne Skår

Agile Contracts – How to Develop Contracts that Support Agile Software Development 302
Mary Poppendieck and Tom Poppendieck

When Teamwork Isn't Working 303
Tim Bacon and Dave Hoover

The Origin of Value: Determining the Business Value of Software Features 305
David L. Putman and David Hussman

The Drawing Carousel: A Pair Programming Experience 308
Vera Peeters and Peter Schrier

Agile Development with Domain Specific Languages: Scaling Up Agile – Is Domain-Specific Modeling the Key? 311
Alan Cameron Wills and Steven Kelly

Ph.D. and Master's Symposium

A Thinking Framework for the Adaptation of Iterative Incremental Development Methodologies 315
Ernest Mnkandla

Exploring XP's Efficacy in a Distributed Software Development Team 317
Alessandra Cau

Agile Methods for Embedded Systems ... 319
Dirk Wilking

Tool Support for the Effective Distribution of Agile Practice ... 321
Paul Adams and Cornelia Boldyreff

The Software Hut – A Student Experience of eXtreme Programming with Real Commercial Clients ... 323
Bhavnidhi Kalra, Chris Thomson, and Mike Holcombe

Eclipse Platform Integration of Jester – The JUnit Test Tester ... 325
Simon Lever

Extreme Programming: The Genesys Experience ... 327
Susheel Varma and Mike Holcombe

Shared Code Repository: A Narrative ... 329
Susheel Varma and Mike Holcombe

Author Index ... 331

Lean Software Management Case Study: Timberline Inc.

Peter Middleton[1], Amy Flaxel[2], and Ammon Cookson[3]

[1] School of Computer Science, Queen's University Belfast, BT7 1NN, UK
p.middleton@qub.ac.uk
[2] Timberline Inc., 15195 NW Greenbrier Parkway, Beaverton, OR 97006 USA
amy.flaxel@timberline.com
[3] 670 SE 31st Ct. Hillsboro, OR 97123, USA
ammonc@lean360.com

Abstract. This paper is believed to be the first documented account of a full adoption of lean by a software company. Lean techniques were devised by Toyota and other manufacturers over the last 50 years. The techniques are termed lean because they require less resource to produce more product and exceptional quality. Lean ideas have also been successful in service industries and product development. Applying lean to software has been advocated for over 10 years. Timberline, Inc started their lean initiative in Spring 2001 and this paper records their journey, results and lessons learned up to Fall 2003. This case study demonstrates that lean thinking can work successfully for software developers. It also indicates that the extensive lean literature is a valuable source of new ideas for software engineering.

1 Introduction

There are two motivations for this work. Firstly, the comprehensive lean literature offers 50 years of experience on how to resolve quality and productivity issues. This case study seeks to show this experience can be usefully applied to software engineering and therefore offer a new way forward. This is believed to be the first documented instance that lean software development this has been carried out on this scale.

Secondly, lean software is significant because many large corporations are using lean techniques in their manufacturing operations. They are now finding that software is a substantial part of their products but it is not managed using lean principles. Software can be a great source of risk and delay therefore they wish to transfer their successful lean experience to their software developers. It this could be achieved, as this study suggests, it would be of great benefit to these corporations.

2 Literature Review

That lean ideas should work with software has been advocated by several writers. Initial experiments have generally supported this. The specific lean techniques and how they were adapted for use with software are described later. This section provides an overview of the lean literature and how it relates to software.

Henry Ford [1] invented mass production and eliminated the craft car producers. His method which relied on a simple product line with long production runs was studied by the Japanese in the late 1940s. Ford's approach did not suit Japan which at that time lacked capital and did not have a market to buy large numbers of identical cars [2]. Toyota [3, 4] gradually refined Ford's ideas so they required less capital, were

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 1–9, 2005.
© Springer-Verlag Berlin Heidelberg 2005

more flexible and produced higher quality products. Lean thinking is now established as the dominant mode for world class manufacturing [5].

The successful application of lean to services and administrative work is described in detail by George [6]. The possible use of lean in software development had been mentioned and to some extent explored by several authors [7-9]. The most extensive work on lean software, although lacking in evidence, is by Poppendieck [10]. This case study is a record of the first extensive application of lean concepts to software development, and therefore seeks to move this subject forward.

3 Background

Timberline is an American software company with all 450 staff based in Oregon. It employed 160 software developers that used C++ and object oriented techniques at the time of the lean implementation. The other people were in areas such as: telephone support, technical writing, administration, sales and marketing. The company was founded in the early 1970's and it is a market leader in software for the construction industry. During the period of the study Timberline was NASDAQ listed. In September 2003 the company was sold for approximately US$100 million to Best Software Inc.

Their software development problems manifested themselves in several ways. There was little insight into how projects were progressing until near the end. This meant there was no predictability of output and they could suddenly find a project needed an extra 60 calendar days to finish. The time from 'code complete' to 'ready to ship' was unpredictable, due to the time taken to 'stabilize' the completed code. Therefore functions such as marketing and training could not prepare, so often their material was not ready when needed. This damaged the product when launched. Often the wrong mix of people and unnecessary people were assigned to projects because the exact nature of what was to be done was unclear. Quality Assurance were testing the wrong things because did not understand what the customer was trying to do with the product. Most importantly, features that customers needed were missed yet money was poured into creating features that customers did not want. This was clear from customers' calls after products were launched.

Although the company had over 20,000 customers worldwide, development costs were too high at 27% of revenue. The use of non-standard processes and tacit knowledge was not proving scalable to handle the growth experienced. When the processes in use were analysed some 900 discrete steps, 600 handoffs and 275 review meetings were identified. Many of the 900 steps did occur in more than one process and were therefore counted more than once. Therefore the raw numbers overstate the complexity of the organization in Spring 2001. Nevertheless the teams charting the processes were routinely astonished at their convoluted and over elaborate nature, which hindered flow. Waste in many forms including rework, inventory, transportation and searching were commonplace.

4 Study Procedures

Peter Middleton is an academic who has been researching software quality for 15 years. He followed up a reference to lean software in a presentation posted on the

Internet that Timberline had made to their local software association. Permission was sought and granted to visit the company in September 2003 for a week to write up their experiences in detail. Follow up meetings and clarification took place in Spring 2004. Timberline offered unrestricted access to staff and material. Extensive group and individual interviews took place with staff from all areas of the company. The consultants were also interviewed to understand their perspective. Amy Flaxel and Ammon Cookson were Timberline staff who worked extensively on the lean software initiative and contributed greatly to this case study.

The research task was to collect as much raw material from Timberline as possible in the form of interview notes, metrics, diagrams and reviewed documents. This evidence was then written up and drafts circulated to various participants to ensure accuracy.

5 Lean Concepts

Team members applied the lean principles and techniques to their software processes. These are principles not prescriptions so each lean implementation will be different depending on the context and constraints of the organisation. All processes were redesigned at the same time. The lean principles support each other so there is no particular sequence or hierarchy for their adoption. The principles and techniques used were:

5.1 Continuous-Flow Processing

In many software companies it takes months or years to get a product out of the door. This represents a substantial amount of money tied up in work in progress, e.g. untested code or requirement documents written but not acted on. If the entire software development process is analysed there are queues and bottlenecks delaying product from being released to customers. Often requirements are put into the system in large batches, which is disruptive and hides errors. The solution is to handle the work in small batches so with a small 'inventory' of requirements, design, code and test are started earlier and therefore mistakes in requirements will be caught sooner.

5.2 Customer Defined Value

Much more effort was allocated to the requirements definition stage. All members of the cross-functional team went to visit customers to see how they worked. If possible they would do elements of the customer's work to experience it directly. Key people in the customer's staff would be identified so they could be interviewed in more depth and later shown early versions of the product. The team would then brainstorm to create a feature list and then survey customers to prioritise the features. This would be repeated to ensure the final set of requirements did not miss anything needed yet also contained attractive 'bonus' features. Care was taken to omit any features that would have a negative impact on customers.

The work of Noritaki Kano who developed a model of the relationship between customer satisfaction and quality was particularly useful. He identified 3 types of customer needs: basic, expected and exciting. Basic needs are assumed from the

product or service and will not be mentioned by the customer. Expected needs are those the customer already experiences but would like them faster, better or cheaper.

The exciting requirements are difficult to discover. They are beyond the customers' expectations. Their absence doesn't dissatisfy; their presence excites. These needs, which are most tied to adding value, are unspoken and thus invisible to both the customer and the producer. Further they change over time, technology and market segment. This stronger customer orientation and the Kano techniques helped Timberline to improve their delivery of products that reflected all 3 types of customer needs.

5.3 Design Structure Matrix (DSM) and Flow

With the Kano analysis the voice of the customer is connected to data. This enables the scope / features trade off decisions to be based on facts not political power. Each requirement can then be broken down into the 2-5 man day chunks of work needed to meet it. Once this was done the skills needed to complete the work became clear. This approach resembles Function Point Analysis as a way of estimating work from breaking down requirements.

But by using the lean principle of flow this could be taken one valuable step further. It was apparent that the work was not flowing smoothly between people with different skills, because some skill areas were overworked while other areas were waiting for work. Therefore what was required was load balancing to ensure for example, that sufficient Quality Assurance staff were allocated to a project and were not overworked, so becoming a bottleneck.

5.4 Common Tempo or 'Takt' Time

Setting a common tempo or 'takt' time was the heartbeat of this lean operation. The objective was to pace work according to customer demand. A key problem was that too much work was being pushed into the software development system. While intuitively appealing it caused the developers to 'thrash'. This meant that they switched tasks frequently so incurring many wasteful 'set up' times. The work would also accumulate in queues where no value was added and delivery would be erratic. This would cause further problems as other project schedules were impacted by the delays.

The solution was to break down projects into units, termed 'kits' of between 2 to 5 staff-days work. Tasks of unknown difficulty would also be allocated 'kits', so breaking them up into manageable chunks. This enabled a track to be kept of progress. The total number of staff-days available for each skill type is divided into the total amount of days work to be done. If there was simply too much work allocated then it could be cut back or extra resource identified. When this was done the team was expected to meet their target date.

The generic takt time calculation is:

Takt time = (Net working days available / no of units required)

Therefore to determine the production rate, x number of kits should be produced in x number of days. It was therefore possible to use takt time to gauge the current production as an indicator of whether or not a project was on target for delivery.

A chart was posted in each team's workspace and used to indicate product delivery status. A green sticker indicated progress at 90-100% of takt, a yellow sticker 80-90%

of takt and a red sticker 70-80% of takt. This chart showing performance to takt time was very popular and used with enthusiasm.

5.5 Linked Processes

Processes are linked by placing their component parts near one another. Originally in the company different functions were located separately. This meant that even setting up simple meetings could take 2 weeks. Analysis showed that during the course of its life a typical project moved over 20 miles within the office buildings. This travelling was pure waste, as it did not add value to the customer. This departmental structure was therefore complemented with the formation of cross-functional teams. This meant that meetings could be held quickly when needed to resolve any issues. A short briefing 'stand up' meeting was held each morning, which was invaluable. Co-location made a considerable difference to productivity. Being such a well-established corporation with many long service employees moving workplaces caused considerable agitation but this did settle down.

5.6 Standardised Procedures

Standardised procedures (e.g. file storage, names, locations, work space) enabled people to be moved around easily. A key idea is to be able to transfer people between projects as they are needed. This raises productivity and speeds product delivery. This was being hindered by idiosyncratic work practices. This was nothing to do with creativity but was the arbitrary result of history. The presence of 'spaghetti' legacy software can also be a serious constraint on the implementation of new software processes. But once the existing processes were looked at from the perspective of consistency, many simple changes to improve could be made.

5.7 Eliminate Rework

Rework severely disrupts current work schedules and contributes to delays and low productivity. It is therefore necessary to eliminate rework by tracking the causes. The number of 'bounce backs' was tracked and has now been halved. Once the root cause of a defect had been found it was then permanently resolved. Rework was also eliminated by investing more time understanding the customer need and context. This also allowed the scope of a project to be cut where it did the least harm, if it was starting to run late. By analysis of the different software components to be built, ones that had high dependencies were identified as having the highest risk. These would be started early with particular care and more thorough testing. This also reduced rework.

5.8 Balancing Loads

Balancing loads eliminates necessary delays. This follows from the Takt time concept. What was happening was that a shortage of one skill would cause delays and hold up other people. A classic bottleneck was that quality assurance staffs were in short supply so testing was not completed quickly enough. By looking at the skills needed per unit of work the work and skills could be rearranged to eliminate the bot-

tlenecks. This was done by an assessment of how many days people with a specific skill were available, after allowing for holidays, training and other commitments. The work to be carried out is captured in the units or 'kits' it was broken down into. By comparing work to be done with the resources available, it then becomes clear if there are any constraints. Often it indicated that some people are overloaded with work, while others have little to do. Multi-skilling staff and reorganising workload can achieve more balance. This raises productivity and improves predictability of delivery. It also creates a less stressful work environment.

5.9 Posting Results

The results to be posted would ideally be each team's hourly productivity rate but daily or weekly is also fine. This is vital. A key advantage is to create a learning organisation. To learn requires constant feedback as to where the problems and errors are. A basic test is that a complete stranger can walk into the work area and see the status of work. This was an initial concern for the staff but quickly became accepted. The reason for this was that it indicated project progress and allowed sensible discussions about when work would be complete and how much capacity was available. This data enabled a team to become self-managing so allowing supervisors to focus on eliminating sources of variation. For example, by monitoring cycle times it was seen on one occasion that manual tests were taking far too long. On investigation it was found the person assigned did not want to do manual testing and was taking every opportunity to be distracted elsewhere. This person was therefore redeployed to writing automated test scripts that they were keen to do.

5.10 Data Driven Decisions

By collecting the data needed to make decisions the teams could be largely self-managing which reduced supervision costs. It greatly reduced the number and duration of meetings. For example, a customer survey showed that a 'cool' new feature that the developers were convinced would be of great benefit to customers was dropped after it was given a low priority in a customer survey. Importantly this decision could be taken quickly and with out politics or rancour as the data behind it was impartial.

5.11 Minimise Inventory

Rather than having large requirements documents, batches clogging up the development process, only a small amount of work was allowed into the system at any one time. This was done by breaking major parts of the product into 'stories' made up of 3-5 'features', which in turn were made up of 3-5 units of work. Each unit of work would be for 2-5 days and have multiple work types, e.g. coding, QA, marketing, within it. Teams could only work on a maximum of 2 features or feature level integration at any one time. This also stopped teams 'cherry picking' features they wanted to develop at expense of the whole product.

The most frequent inventory problem was the build up of untested code because QA was busy. Putting inventory limits in place means that a discipline can only work

so many kits ahead of the discipline who will be working next. For example, Engineering would be only allowed to work 3 kits ahead. Minimising inventory is very important as untested code or large volumes of requirements inevitably contain defects which need to be discovered and the problems fed back into the process. This enables self managed teams to learn and steadily improve the quality of the software produced.

6 Lessons Learned

After over 2 years of effort from Spring 2001 to September 2003 the following can be reported. It was clear that the lean techniques did transfer into a software development organization. To establish if lean software development provided significant advantages six areas were looked at.

6.1 Initial Process Analysis

The complexity of the software development process in May 2001 is illustrated by the following findings. One process involved 498 steps, in another process only 1.4% of the steps served to add value to the product and a third process travelled 28 miles as it weaved it's way through the company's offices. All these processes have now been substantially improved due to lean thinking. This indicates using lean techniques to diagram and refine processes is effective.

6.2 Informal Estimate of Productivity Gains

The informal estimate by senior staff was that there had been about a 25% productivity gain over the two years. This was based on carrying out more work with fewer staff. As there was no baseline measurement before the work started it is not possible to provide a more specific figure.

6.3 Staff Survey

This was carried out anonymously in September 2003, to provide insight into what the software developers, QA engineers, technical writers and business systems analysts, thought after experiencing over two years of lean software development. In the survey with a high response level, 55% of the respondents either strongly agreed or agreed that lean ideas do apply to software development, while another 24% were neutral. When asked whether or not lean software had made things worse, 82% indicated that lean software had not made things worse, or were neutral. Another question was asked regarding whether or not employees thought that lean was just a passing fad. To this question, only 10% of respondents felt that lean was a fad.

6.4 Quality Metrics

For each new feature about 20-25 customers drawn from a representative sample were interviewed and many more were surveyed. This raised the quality of products, as no

rework was required after release. Smooth progress to the next version could now take place, which was a significant benefit. Schedule slippage fell to at most 4 weeks where previously it had been months or years. There was a 65%–80% decrease in time taken to fix defects during the development cycle. Defects needed to be repaired a second time, due to a faulty first fix, dropped by 50%.

6.5 Process Data

A key test of whether a process is under control is if a person unfamiliar with a project can establish its status in a few minutes. To do this there is a need for simple visual indicators. The takt time to actual chart shows the number of units of work completed over time, compared to a target. It easily allows the viewer to see where the project is relative to its time-related goals, as well as the overall trend.

6.6 Customer Satisfaction

When the first product produced under the lean process was released at a trade show, the customer response was overwhelmingly positive. This was due to the continual focus on customer needs combined with the frequent, iterative development cycles. This approach delivered the functionality and ease of use the customers needed.

7 Conclusion

The evidence from Timberline's experience over the last two years is that lean thinking does transfer across to software development. As software quality is impacted upon by all sections of the company, it is essential it is a company wide effort, rather than just confined to the software developers. The place to start is with the senior management team with some training and a hands-on simulation. A software quality initiative only within software development will be severely limited in the benefits it can hope to achieve. The way forward is to take the eleven lean ideas described and adapt them to your culture and circumstances. Each lean implementation will be different. The short feed back loops accelerate organizational learning and enable the continuous reduction in process and product variation. This provides the productivity and quality gains that software people often need.

Lean software development enormously facilitates short, frequent iterations, each with a minimum useful number of features delivered to the internal or external customers. When these frequent iterations are coupled with capturing the voice of the customer more effectively, the results can be excellent. The conventional approach of just gathering customer requirements is taken to a new level of customer intimacy and understanding. The extra effort to really connect with the customer pays off in less rework to correct skimped analysis and greater customer satisfaction.

For manufacturers whose products have high software content, lean software development does show a way forward. They can leverage their lean expertise into their software organisations. It gives a common company wide language with which to address quality and productivity issues. The lean focus on customer needs helps to integrate the software people more closely with the rest of their corporation.

References

1. Henry Ford, *Today and Tomorrow,* Productivity Press, New York, 1926, reprinted and updated 2003
2. Richard J. Schonberger, *Japanese Manufacturing Techniques,* The Free Press, New York, 1982
3. Shigeo Shingo, *A Study of the Toyota Production System,* Productivity Press, Portland, OR, 1989
4. Taiichi Ohno, *Toyota Production System: Beyond Large-Scale Production,* Productivity Press, Cambridge MA, 1988
5. James P. Womack and Daniel T. Jones, *Lean Thinking,* Touchstone Books, London, 1997
6. Michael L. George, *Lean Six Sigma for Service,* McGraw-Hill, New York, 2003
7. Jeffrey K. Liker, *Becoming Lean,* Productivity Press, Portland, OR, 1998
8. Earll Murman et al, *Lean Enterprise Value: Insights from MIT's Lean Aerospace Initiative,* Palgrave, New York, 2002
9. Barry Boehm and Richard Turner, *Balancing Agility and Discipline,* Addison-Wesley, Boston MA, 2004
10. Mary and Tom Poppendieck, *Lean Software Development: An Agile Toolkit,* Addison-Wesley, 2003

XP South of the Equator: An eXPerience Implementing XP in Brazil

Alexandre Freire da Silva[1], Fábio Kon[1], and Cicero Torteli[2]

[1] Department of Computer Science of the University of São Paulo
{ale,kon}@ime.usp.br
http://www.ime.usp.br/~xp
[2] Paggo Ltda.
torteli@paggo.com.br
http://www.paggo.com.br

Abstract. Many have reported successful experiences using XP, but we have not yet seen many experiences adapting agile methodologies in developing countries such as Brazil. In a developing economy, embracing change is extremely necessary. This paper relates our experience successfully introducing XP in a start-up company in Brazil. We will cover our adaptations of XP practices and how cultural and economical aspects of the Brazilian society affected our adoption of the methodology. We will discuss how we managed to effectively coach a team that had little or no previous skill of the technologies and practices adopted. We will also cover some new practices that we introduced mid-project and some practices we believe emerged mostly because of Brazilian culture. The lessons we learned may be applicable in other developing countries.

1 Introduction

There are many reports of successful experiences introducing XP, both in research and industrial contexts, throughout the northern hemisphere. There have been numerous accounts of success in the USA, Finland, Sweden, England, Spain, Italy and Japan[1–7]. Closer to our reality, there is a report of introducing only one of XP practices in developing areas in China [8]. However, there is little recorded evidence of successful implementations of XP in the Southern Hemisphere and in developing economies such as Brazil, specially adopting all of XP original practices[9].

Learning to adapt to change is specially important in a developing economy such as ours. Businesses come and go rapidly, and fluctuations in the economy have even caused our currency's name and value to change twice in a single decade. Good developers are hard to find, and many enterprises survive through constant recycling of interns. Low salaries (ranging from USD 100 to USD 2000 per month) reflect an untrained work force, composed mostly of interns making a little more than USD 200 a month, and a culture of constant people turnover. Tools and frameworks have scarce documentation in Portuguese, which lead to many weak developers in the market.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 10–18, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Also, there are some cultural aspects of a tropical country that have impact on software development industries. According to Seŕgio Buarque de Holanda's *Cordial Man* theory[17], brazilians react from their hearts, being passionate in all aspects of life, developing a need to establish friendly contacts, create intimacy, and shorten distances. Brazilians reject last names, referring to everyone by their nicknames. We reject formalities, even in the workplace. We are incapable of following a hierarchy, of obeying too rigid a discipline. This has positive impacts, brazilians tend to be open-minded, creative, friendly, and collaborative. Teams tend to get along well and work together having fun. As a multi-cultural and mixed society we tend to welcome change and get along very well in the workplace. Disadvantages also exist, compared to most cultures from northern hemispheres, we tend not to be punctual and constantly miss deadlines. Some mention fear that XP is heavily based on north-american culture and therefore would not work on a very different culture such as ours. Kent Beck guesses that the biggest disadvantage for XP in Brazil is exactly the lack of commitment to deadlines (even when they might be exceeded because the team is having fun) [10]. Our experience shows that this is not the case.

Developing high quality software, on time and on budget is a must if one plans to survive in this context. As such, the first author was invited to help introduce XP in a start-up enterprise, Paggo, trying to get into the credit card business. From the beginning, many challenges were present; we believed that the two most difficult were going to be the heterogeneous aspect of the team, composed of developers with different skills, from interns with little or no experience programming to seniors accustomed with their own way of programming, and the fact that our coach could not be present full-time because of the limited budget. We had high hopes since adopting XP was a suggestion from a team member and everyone in the team accepted the challenge with no knowledge of the difficult times ahead.

We have successfully trained our team in all of XP practices and consider the project to be a success. This paper will briefly outline the 6 months in which we trained our team in the practices and in most technologies they would need to use, describing changes encountered along the way and how we coped with them. We will then consider the adaptations we performed for XP practices and lessons learned in the experience. We will list some other valuable techniques implemented during the project and some special practices we believe are the result of the cultural and social aspects of Brazil. We will then conclude with some remarks that might be of value to similar attempts in developing countries.

2 Project Evolution

Paggo is a start-up venture in the credit card business. It attempted to go into a very competitive market and its bets were in a new business model based on new technologies and implementing an agile method so the enterprise could have functioning software quickly, to secure more investments by reducing time-to-market. Our main objective was to have an XP proficient team ready to be independent from the coach within 6 months.

The software to be developed was cutting-edge, using technologies such as J2ME and J2EE and free and open source frameworks such as *VRaptor*, *Hibernate* and *JBoss*. The project had many aspects, from a credit transaction handler with high performance requirements, to mobile technology to be embedded in cellular phones, and a dynamic Web site where customers could sign up for credit cards and check their monthly balance.

The development team was really heterogeneous, skills ranged from interns with almost no programming or OO knowledge to senior developers with years of experience, we believed this would be a real obstacle to installing XP. How to get everyone on board and at the same time address individual difficulties? Even though every member was willing to work hard on implementing XP, there were clear tendencies from some developers to be CowboyCoders [11] and many did not yet have the skills necessary to do XP. In our favor one of the founders of the company played an excellent in-house customer. A part-time consultant was hired to mentor the less skilled in the team in topics ranging from Java programming, OO, and the frameworks and technologies to be used in the project and also coach the team in XP. In the beginning of the project another part-time consultant was hired to help with the new technologies. By the end, two more developers were hired as well, adapting quickly to our XP environment and writing production code within one week of beginning work, contributing with very relevant code already in the second week. This was due to the team being comfortable with XP by the time they were hired and the fact that one of them took an undergraduate course in XP [14].

We decided to implement all of XP practices as proposed by Beck[9] at once, knowing that some would take more time to reach a mature and acceptable level. We managed to go through 12 releases, using mostly two week iterations. We produced four applications, successfully implementing 269 stories out of an original 340, of which 42 were later discarded or deemed unnecessary by the customer. From a technical point of view, we delivered 90% of wanted functionality, fully tested and free of bugs. From a business perspective, the project was such a success that the company was sold for a good value and restructured to focus on software development with the same XP team.

During the first two months we fully explored all of XP practices but tread lightly into practices that demanded more knowledge such as test-first design and refactoring. In the next 4 months we trained the team in some OO patterns and in the open-source frameworks used. As the team became more comfortable with patterns and advanced OO techniques so did our testing and refactoring practice evolve. After attending a local XP conference, the coach introduced some new practices, most importantly the retrospective technique suggested by Linda Rising[12] and analyzed in detail in [2]. We decided to use a slightly modified version of the KJ method [13] using colored post-its grouped in positive and negative findings by the development team. The introduction of this new practice also had some unexpected results as discussed in Section 4.1. At the end of the 5th month, the company had to cut expenses because it had not yet secured a new investment. By this time the coach was satisfied with how our

XP practices were being followed and it was decided that he would leave the team. He then proceeded to help ensure that the team would be able to keep on going without him as detailed in Section 4.4. Recently, an investment has been secured and the company now plans to double its development team, we plan to document this new effort in a future paper.

3 Adaptations to XP Practices

Customer Always Present. We were really lucky to have an inside customer who wrote stories and was very much in favor of XP and enthusiastic about the agile practices. He wrote acceptance tests and executed all of them after each release. The customer was also available for our daily stand-up meetings (actually running some of them when the coach could not be present) and re-prioritized stories as time went by. This was very productive, as our team was learning to estimate development effort, some estimates were really blown but, in the end of a iteration, only stories that were not really important for the customer were left out. In our experience a committed customer is essential, especially if the team is composed of less skilled interns and can still let bugs escape tests and badly estimate some stories.

Coding Standards. Coding standards were easy to implement, due to the fact that most of the team was learning Java at the time. Standards were discussed in meetings, mostly suggested by senior members of the team, and put on a poster on the wall called "Team Arrangements". It was straight forward to teach and impose the standards through pair-programming. We believe calling the standards *arrangements*, and being flexible about their adoption made them easier to absorb by less experienced team members.

Continuous Integration. We had problems with continuous integration due to the fact that most were learning how to use tools for version control. There were a couple of times were code was actually lost during complicated merges. As the team became more comfortable with these notions they suggested we adopt *Cruise Control*, which we did to many benefits. Through our retrospective meetings, we identified problems with this practice and took concrete actions that helped us improve, such as having quick stand-up meetings when difficult merges were about to happen.

Metaphor. We had no trouble to implement metaphor. This is mostly due to our customer being available to give daily business explanations to the team and, during planning games, agreeing on common metaphors. The fact that team members were also helping each other learn OO concepts and frameworks helped. Eg., as the team would learn about a particular pattern, we could easily incorporate this abstraction in our metaphor.

Test Driven Development. In the early months of the project it was difficult to write good tests that covered our demo application completely. Most developers did not know how to write automated tests and we were dealing with relatively

hard technology to test (eg., J2ME applications or serial device communication). Part of the team did not have enough OO know-how for us to use techniques such as *MockObjects*, so in the beginning we only had the customers manual acceptance tests for feedback.Our coach decided to pair with developers whenever he could to teach testing techniques. We had difficulty with the less skilled developers, especially the interns lacking OO knowledge, but a lot of resistance was also encountered from the senior developer, who could not see benefits in having automated tests for his code. After a couple of iterations and some failed releases the team understood how important it was to have a full test suite, covering all production code. What happened then was a truly "test-infected" scenario, developers suddenly saw tests as an excellent tool and strived to excel in this practice. We kept daily metrics for the number of tests created and they started growing exponentially. It helped that the new developer, with previous XP experience, was quick to develop intimacy with the team, and felt courageous enough to rewrite all tests for a J2EE project when the customer saw the need to code new features for it. At the end of the sixth month period, the team was looking into technologies to automate the customer acceptance tests, this was again an initiative of their own. We learned that teaching testing can be difficult, especially with heterogeneous teams like ours, but having test metrics helped everyone to be conscious about the problem.

Refactoring. Refactoring was also one of the hardest techniques to teach. In the beginning, we did some minor refactorings to get the team to understand their value, mostly cleaning up class and method names. During the project we introduced agile modeling techniques[15] that were useful for us to discover areas of our applications that could go through more extensive refactorings. We held design meetings and used the white board to draw UML diagrams and decided, as a team, where we should refactor. The senior developers were eager to refactor but we found that the interns and junior developers did not want to refactor as much, for they had not yet had time to grasp some more complex OO concepts. It was helpful to have a tool such as Eclipse that would automate refactorings. It made them easier to learn and gave the team more courage to execute them.

Small Releases. The project had 12 releases, most taking 2 weeks. If the customer was not satisfied with the acceptance tests we had special 1-week "bug-fix" releases . This was specially true in the beginning of the project when we did not have enough tests and the developers were learning the technologies. We developed an automated deployment system, composed of a development server, a homologation server and a production server. After a release was tagged, it would be automatically updated on the homologation server, which kept a recent copy of the production server's database. The acceptance tests were run in this server and, if the client was satisfied, the release would be manually deployed on the production server.

Planning Game. We had good planning games, the customer had interest in commenting on previous releases and did not hesitate to change his mind. We

divided a work day into 2 individual working hours and 3 pair-programming sessions, estimating stories in terms of these sessions. If a story was estimated in less than 1/4 of a session or more than 6 sessions it would be rewritten. The client prioritized and grouped stories. As we were developing a couple of applications simultaneously, we wrote stories for all of them, developers liked being able to move from one project to another. As most of our releases had a 2-week duration, we built a special calendar on the wall, where 10 days were represented. After the planning game, we would place stories along the days for the two weeks, starting with the highest customer priority, and fitting next stories according to estimates of stories already on the board and our developer resources. It was also used daily when we would review what stories we had left, assign them to pairs and eventually re-manage other stories. We found this to be a very efficient way to assign stories and keep track of progress. Latter we used this board for our retrospective technique as described in Section 4.1. Feedback from our retrospectives lead us to introduced some "studying stories" where developers could take a few sessions to dedicate themselves to studying new technologies as described in Section 4.3.

Sustainable Pace. This was a hard practice to follow, mostly due to economic reasons. In Brazil, people are willing to work extra hours (without payment) and this was not any different in our team, we counted with a couple of extra hours per developers weekly. The fact that interns and trainees were not present full-time encouraged this, as they were eager to put in extra, unpayed, hours.

Pair Programming. In a economy were developer turnover is high, our customer did not want any production code created individually, so he instated pair programing as a rule. Pair programming was very valuable to teach developers testing and refactoring techniques, and our coach wished he had more time to be able to pair even more with the team. Less skilled developers also benefited from a hidden pair, *Eclipse*, it helped them to learn the language with it's rapid feedback about syntax mistakes and compilation problems. We encountered resistance from the most senior developer, accustomed to working alone, he had a passionate reaction to being forced to pair and others avoided pairing with him. We also found that, although it was good to pair more experienced coders with beginners for mentoring, sometimes it was more productive to let the less experienced pair program on tasks that seniors found repetitive and boring.The biggest advantage we found with pair programming was when hiring new developers, when they pair-programmed we were quick to identify if they would adapt to the company's structure and philosophy.

Simple Design. Simple Design was not trivial, but, as we were also teaching developers how to design, we did accomplish a satisfactory simple design. Having modeling meetings, as proposed by the agile modeling community, made it easier to teach and discuss simple design, proposing refactorings upon the design that had evolved so far.

Collective Code Ownership. As we had a set of coding standards that was working, it was easy to implement collective code ownership. We found that the senior developers were more comfortable with this practice, especially when they were refactoring code produced by interns.

4 Other Practices

4.1 Retrospectives

We found retrospectives to be really valuable and greatly improved our communication. We used the same story board from our planning game to pin red or blue post-its on the days we encountered nice or bad things to say about our practices, at the begging of each week we would collect the post-its from the previous one and have a retrospective meeting to discuss them.

Discussing our process and techniques helped developers to identify problem areas and suggest solutions. In the beginning, we held weekly retrospectives and came up with really good suggestions to fix problems. After some time, however, the need for these meetings was lessened because we were good at fixing problems, this has been pointed out by Cockburn [16].

Due to the proximity developed because of pair programming and the increase in communication needs, the retrospective technique as it was done at Paggo started to be used for personal differences. At some point in time the team even took a cold shoulder approach to some of the developers. They did not want to pair program with some specific members anymore. The rest of the company realized that something was going on. In the meantime, a real paper war developed on the board, with red notes flying in all directions, even posted by people in the company outside of the development team. Our retrospective technique had turned into an enormous gossip board, as brazilians, reacting according to our hearts had shown it's downside. The result was the invention of a practice we call "dirty laundry meeting"

4.2 Dirty Laundry Meeting

After seeing that things were going astray with the team, the customer decided to hold a meeting in which everyone was supposed to resolve their conflicts. This meeting was called "dirty laundry meeting" because it was a chance for everyone to say what was on their mind about others and walk away with a clean slate.

Team members, as expected by their brazilian culture, had grown closer, making our work relationship almost a family one. This made this meeting very emotional and intense, a couple of people even cried. It was a strange experience we believe happens more often in countries like Brazil, derived from our social and cultural inclinations. In this meeting we found a place to put our personal differences in check and wash away everything that was bothering us. It resolved most issues but was a very extreme practice and we do not advise that it should happen frequently. Sometimes it is necessary, producing nice results, if people

are willing to be frank and share their feelings. We believe that certain personal differences that affect productivity can stay hidden for long periods of time in most corporations, but will surface very fast with XP. These will have to be resolved or will affect production, and dirty laundry meetings are an interesting solution.

4.3 Specialists and Study Time

Given the heterogeneous nature of our team it was clear that some people had a lot to learn that others could teach. We came up with the concept of specialists, not in the sense that they would do all stories related to their field of expertise, they were people that the team could count on, knowledgeable about latest advances on their field and capable of solving hard problems encountered in stories related to their areas. The need for specialists arose from our retrospective meetings. Developers said that they were more motivated to work on things they liked and they would like time to learn more and research. So we instituted some special "research stories". The specialists could take these stories and have a break from pair programming in a couple of study sessions when they would research technologies of interest and program spikes.

The specialists brought some fresh air into the team and reduced the burden of everyone having to study all new technologies. They did not have special rights to stories in their areas. In fact they were discouraged from taking these stories at all. They were available to pair program when someone had trouble in their areas of research and also conducted seminars to teach the rest of the team what they were learning.

4.4 Coach of the Week

Approaching the end of the sixth month the company no longer needed the presence of the external mentor to play the role of coach any more. Most developers were comfortable with the process and had mastered the technologies and techniques used. As such the coach started to plan his leave, the team had to be able to do XP on their own. The coach started a practice where the team would elect a developer to play the role of the coach for a week. After a couple of weeks most of the team had been in the role of coach (with the mentor's supervision) and were ready to walk on their own.

5 Conclusions

The chaotic economy and culture of Brazil have impacts on implementing XP. We have successfully used all of XP practices, adopted most of them and even came up with some unique practices of our own. XP helped us adapt quickly to the constant changes in the economic reality of a developing country. Even though our team was very heterogeneous and had many lesser skilled developers, we managed to help them evolve and fit in to the team. By promoting everyone's

participation, XP can help all to successfully learn practices and technologies due to an open, motivating and friendly environment. In a market were teams have to grow quickly to be competitive, companies can suffer from hiring the wrong people. XP helped us welcome newcomers, and find out quickly if they were going to fit in. We believe XP is harder to implement when the team is heterogeneous as ours, but it is possible to do with patience and brazilian passion. When constantly refining one's practices through retrospectives, politics and personal conflicts can not go unnoticed for long, this allows a company to take quick measures to maintain productivity. We believe other developing countries could benefit from our experience.

References

1. A. Fuqua and J. Hammer, "Embracing Change: An XP Experience Report", XP 2003, Lecture Notes in Computer Science, vol. 2675, pp. 298-306. Springer, 2003.
2. O. Salo, K. Kolehmainem, P. Kyllönem, J. Löthman, S. Salmijärvi, and P. Abrahamsson, "Self-Adaptability of Agile Software Processes: A Case on Post-iteration Workshops", XP 2004, Lecture Notes in Computer Science, vol. 3092, pp. 184-193. Springer, 2004.
3. H. Svensson, "A Study on Introducing XP to a Software Development Company", XP 2003, Lecture Notes in Computer Science, vol. 2675, pp. 433-434. Springer, 2003.
4. T. Mackinnon, "XP - Call in the Social Workers", XP 2003, Lecture Notes in Computer Science, vol. 2675, pp. 288-297. Springer, 2003.
5. T. Bozheva, "Practical Aspects of XP Practices", XP 2003, Lecture Notes in Computer Science, vol. 2675, pp. 360-362. Springer, 2003.
6. W. Ambu and F. Gianneschi, "Extreme Programming at Work", XP 2003, Lecture Notes in Computer Science, vol. 2675, pp. 347-350. Springer, 2003.
7. Y. Kuranuki and K. Hiranabe, "XP "Anti-Practices" : anti-patterns for XP practices", presented at The Agile Development Conference, Salt Lake City, Utah, 2004.
8. K. Lui and K. Chan, "Test Driven Development and Software Process Improvement in China", XP 2004, Lecture Notes in Computer Science, vol. 3092, pp. 219-222. Springer, 2004.
9. K. Beck, *Extreme Programming Explained, Embrace Change*. Addison Wesley, 2000.
10. K. Beck in *Extreme Programming Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade* by V. Teles. Novatec, 2004.
11. "Cowboy Coder" online at `http://c2.com/cgi/wiki?CowboyCoder`
12. L. Rising and E. Derby, "Singing the Songs of Project Retrospectives: Patterns and Retrospectives", Cutter IT Journal, pp. 27-33, September 2003.
13. R. Scupin, "The KJ Method: A Technique for Analyzing Data Derived from Japanese Ethnology", Human Organization, vol. 56, pp. 65-72, 1996.
14. F. Kon, A. Goldman, P. Silva, and J. Yoder, "Being Extreme in the Classroom: Experiences Teaching XP", Journal of the Brazilian Computer Society, 2004.
15. S.W. Ambler, *Agile Modeling*. John Wiley & Sons, 2002.
16. A. Cockburn, *Agile Software Development*. Addison Wesley, 2002.
17. S.B. de Holanda, *Raízes do Brasil*. Companhia das Letras, 1995.

Introducing Extreme Programming into a Software Project at the Israeli Air Force

Yael Dubinsky[1], Orit Hazzan[2], and Arie Keren[3]

[1] Department of Computer Science,
Technion – Israel Institute of Technology, Haifa 32000, Israel
yael@cs.technion.ac.il
[2] Department of Education in Technology & Science
Technion – Israel Institute of Technology, Haifa 32000, Israel
oritha@techunix.technion.ac.il
[3] MAMDAS – Software Development Unit, Air Force, IDF, Israel

Abstract. Introducing Extreme Programming (XP) to an industrial software development team usually involves technical and professional aspects as well as social and organizational ones. The introducing of a new software development method in general and XP in particular into a software project team that operates in the army emphasizes and extends these issues. In this paper we present a process through which XP has been introduced into a 60-members software development project at the Israeli Air Force. Focus is placed on an XP workshop conducted with ten officers who worked at different teams of the project. Specifically, we present the principles according to which we facilitated the workshop, the workshop agenda and data regarding the way the participants perceive some of the XP practices. Recently, the first XP team in this project has started to work the XP way.

1 Introduction

Awareness to agile software development methods increases in the last few years. More companies seek for innovations and solutions that may answer their special needs, and find the agile methods in general ([3],[5]) and Extreme Programming (XP) ([1]) in particular as a possible answer. Experience gathered in the community indicates that the introduction of XP into an organization goes along with conceptual and organizational changes that are integrated part of the process.

In this paper we present the process of introducing XP into a software project at the Israeli Air Force. In Section 2 we describe the project setting and the preliminary phase that resulted in an XP workshop. In Section 3 we focus on the XP workshop activities. In Section 4 we bring data and its analysis regarding the way the workshop participants conceived some of the XP practices. We summarize in Section 5.

2 The Air Force Software Project

The software project we deal with is being developed by MAMDAS which is a software development unit in the Israeli Air Force. The project is being developed by a team of 60 skilled developers and testers organized in a hierarchical scheme of small groups. The software is intended to be used by thousands of users.

The third author, who is in charge of the system engineering group of this project, was requested to lead a change in the current development process to implement a new process that would enable rapid response to customers' requests and requirement

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 19–27, 2005.
© Springer-Verlag Berlin Heidelberg 2005

changes, and would obtain feedback with respect to released features. This sub-project is named as Short-Cycles and its main goal is to change the method of work in the project itself. The duration of Short-Cycles was set to one year in which a new methodology has to be suggested and an initial team should start to implement. It was clear that an organizational and conceptual change should take place. Since the team was relatively large, such a change could not be performed over night, but rather in a gradual, stage-based process which was to be planned accordingly.

Here are three characteristics of the software project with which Short-Cycles had to deal. First, an explicit role for each project employee could not be defined nor its performance enforced; thus, it was not clear who was in charge of what. Second, work habits are so well rooted in the software development unit that even if a change were to be implemented forcefully, the accepted development process might well eventually "return through the back door". Third, communication channels between teammates and customer and among teammates were cumbersome and could not be forced; for example, in some cases no users' feedback was received and in other cases when feedback was finally received it often turned to be irrelevant.

These characteristics further explain the need for a process that would gradually guide the transition and assimilation of the new software development method. This process was also to take into account the individual interests of different people, the objections each sub-team was expected to raise, and the harmony and synergy between the different changes that were to take place in each specific defined process.

It is important to note that the Air Force leadership supported the Short-Cycles phase. Furthermore, the leadership specifically declared that, while a reduction in quantity might be accepted, quality and fitness to customers' needs were not to be compromised. It was realized that resistance may be raised by the mid-level officers.

In preparation for this transition, a Short-Cycles Group of 10 volunteers was established. Its aim was to formulate and lead the process that would end with the entire team working according to the new desired development process. The Short-Cycles Group was composed of members representing all levels of the 60-people team. The group met every two weeks.

One of the decisions of the Short-Cycles Group was to learn about the agile approach and specifically the XP method. Part of this decision was to conduct an XP workshop with project members who will be able later to evaluate and decide upon the sequel of Short-Cycles.

3 The XP Workshop

In this section we describe the XP workshop that was conducted as part of Short-Cycles. The workshop was facilitated by the two first authors and its orientation was based on our belief that Extreme Programming (XP) is best understood when a team experiences and works according to the XP values and practices in real-life software development situations.

In the 2-day workshop we aim at introducing XP while enabling the participants to experience, as far as possible, the actual construction of a software system. The idea is to show the participants just how much can be achieved in two days if the plan and time management are conducted properly, all while paying proper attention to the customer's needs. In order to achieve this goal within the time limits of a 2-day work-

shop, the participants complete during the workshop the development of the first iteration of a software system that they choose to develop. In addition, reflections are conducted, and a discussion is held on the suitability of XP to the organization.

Following are the principles that were applied in the 2-day workshop.

Principle 1: Experience the different XP practices as much as possible. Since we believe that real experience is needed in order to capture the main ideas of XP, hands-on experience by the participants is integrated, as much as possible.

Principle 2: Elicit reflection on experience. The importance of introducing reflective processes into software development processes in general and into the XP development environment in particular, has been already discussed ([4]). During the workshop, we aim to elicit such processes in several ways, such as asking the participants to recall and reflect on situations taken from their own experience in software development, and presenting questions in which the participants are asked to compare different situations in software development.

Principle 3: Relate to participants' feelings towards the presented software development environment. The adoption of XP requires a conceptual change with respect to what a software development process is. Consequently, it is well known that XP raises emotions that, in some cases, are in fact objections. When the audience expresses emotional statements against XP, we take advantage of this opportunity and encourage participants to express their feelings towards the subject, open it to discussion, and explain how XP addresses the issue just raised.

Principle 4: Use narratives. As an alternative to a technical explanation of the XP practices, we have adopted the storytelling approach, which has become more and more popular in management process[1]. Following this spirit, we present the XP practices (in various levels of detail) using a story.

The story describes two days in the life of a software developer working in an XP environment: one business day and one development day. The appropriate XP practices are used to describe these two kinds of days. In addition to the story itself, we invite the audience to envision themselves in the described environment and to continuously compare it with their current software development environment.

Principle 5: Stick to timetable unless an interesting topic comes up. During the workshop we try to adhere to the timetable exactly as planned (see Tables 1, 2). The idea underlying this guideline is to illustrate how much can be achieved in a relatively short period of time. In addition, on such a tight time schedule, without too many long breaks, participants feel that their time is valued. Naturally, there are cases in which we decide not to follow this principle. This happens in cases in which we feel that an interesting topic has come up and by focusing on it for a while we may improve the workshop in a way that was not pre-planned.

Principle 6: Our preferred group size is 6-12 participants. The upper limit (12) is obvious. We require the lower limit (6) because we believe that this number of participants affords a feeling of how XP works for a team. In addition, we require that all participants commit to attend the full two days. If a participant is forced to leave the workshop for some unexpected reason, we take the opportunity to discuss the analogy of such a case to real life situations.

---

[1] See at http://www.creatingthe21stcentury.org/

Principle 7: Sustainable pace (8 hours every day). Similar to the XP practice of Sustainable Pace, we make sure that the development performed during the workshop is conducted at a reasonable pace. The fact that the first iteration is developed within two such days serves to reinforce the participants' feelings about the potential contribution of XP to software development processes.

Principle 8: Workshop site. We request that the host company equips the room in which the workshop is to take place with a large table for the planning game, computers, flipcharts or a whiteboard, and a well-stocked coffee corner.

Tables 1 and 2 present schedules for a 2-day XP workshop. In order to illustrate how the days look in practice, we added hour notations to the workshop plan. The schedule presented here is a general schedule of the workshop. The details of each activity are beyond the scope of this paper.

As can be observed from Table 1, the main role in the first day is that of the customer, and the message that we try to convey in this day is that the customer is an integral part of the development environment.

Table 1. First day of the 2-day XP Workshop

<table>
<tr><th>10:00-10:30</th><th>30 minutes</th><th>Introduction</th></tr>
<tr><td colspan="3">The 2-day story is presented briefly and references are made to what the participants can expect to experience during the next two days.</td></tr>
<tr><th>10:30 – 11:00</th><th>30 minutes</th><th>Selection of a topic</th></tr>
<tr><td colspan="3">We let the participants decide on the software they are going to develop in the workshop. This is done by asking them to suggest ideas and than taking a vote on the preferred topic.</td></tr>
<tr><th>11:00 – 13:00</th><th>2 hours</th><th>Planning Game & Role Scheme</th></tr>
<tr><td colspan="3">▪ Planning game. All stages of the Planning Game are performed: telling of customer stories, allocation of releases and iterations, simple design of the first iteration, breakdown into development tasks, allocation to developers, time estimation, and load balance.
▪ Role scheme. We build a role scheme together with the team, launch the created scheme, and refer to special roles during the activities.</td></tr>
<tr><th>13:00 – 13:30</th><th>30 minutes</th><th>Using Metaphors</th></tr>
<tr><td colspan="3">We present the essence of metaphors and delve into the details of using them during the development process in general and during the planning game in particular. We also use metaphors when discussions regarding problems arose.</td></tr>
<tr><th>13:30 – 14:00</th><th>30 minutes</th><th>Lunch break</th></tr>
<tr><th>14:00 – 15:00</th><th>1 hour</th><th>Planning Game (Cont.)</th></tr>
<tr><td colspan="3">The part of task allocation, time estimations and load balancing is performed.</td></tr>
<tr><th>15:00 – 17:15</th><th>2 hours and 15 minutes</th><th>Start of Development</th></tr>
<tr><td colspan="3">XP practices (technical and organizational) are presented during development.</td></tr>
<tr><th>17:15–18:00</th><th>45 minutes</th><th>Reflection & Summary</th></tr>
<tr><td colspan="3">▪ We end the day with reflection and retrospection on the first day.
▪ In preparation for the second day of the workshop, participants receive thinking homework.</td></tr>
</table>

Table 2 presents the schedule for the second day of the XP workshop, together with an explanation of the main activities conducted during each time slot. All roles are active in the second day so to bring the first iteration towards completion.

Table 2. Second day of a 2-day XP Workshop

10:00 – 10:15 15 minutes Stand up meeting
Stand-up meeting.
10:15 – 10:45 30 minutes TDD
Presenting test-driven development (TDD).
10:45 – 13:30 2 hours and 45 minutes Software Development
▪ Software development. ▪ XP practices (technical and organizational) are presented during development. ▪ Roles scheme is performed.
13:30 – 14:00 30 minutes Lunch break
14:00 – 16:00 2 hours Software Development
▪ Stand up meeting. ▪ Completion of development of iteration 1.
16:00 – 16:30 30 minutes Presentation
▪ Presentation of the first iteration. ▪ Roles summaries. ▪ Feedback activity.
16:30 – 18:00 1.5 hours Reflection & Summary
▪ Reflection activities. ▪ **Implementation of XP in the participants' organization.** This last part of the workshop is dedicated to a discussion in which the participants present their thoughts on the implementation of XP in their organization in general, and in their team in particular. ▪ Summary.

4 Data and Analysis

This section presents an analysis of data that was gathered during the workshop mainly by using anonymous written reflections filled in by the participants.

4.1 The Planning Game and the Roles Scheme

The first feedback was received from the participants after the first planning game had been conducted. This was the first time they participated in such an activity, yet they were not familiar with most of the XP practices. Participants were asked to reflect on the contribution of the planning game to their understanding of the project and to describe the advantages as well as the disadvantages of this practice. Nine participants (out of ten) have reported that the planning game helped them in better understanding the project's essence and its requirements, among them the officer who played the role of the customer. Only one participant reported that the planning game did not contribute at all.

Following is a sample of participants' expressions when describing the *advantages* of their first planning game: “Sharing information among teammates. Everyone knows what happens. Understanding of requirements by all teammates, acquaintance with all factors involved.”; “…enables distribution to sub tasks.”; “The process was quick and enabled making many decisions and tasks in short time.”; “The customer was available and understand our constraints.”; “We see everything in front of our eyes.”; “Collaboration and better acquaintance with the people we work with.”

Following are participants' expressions when describing the *disadvantages* of their first planning game: “More discussions, which were not necessarily directed to solve problems.”; “Many people involved, lack of focus.”; “Tendency to diversion. The customer is in pressure and doesn't remember everything.”; “We didn't check duplications of subjects.”; “We designed top-down and didn't verify using bottom-up.”; “Sometimes, over talking causes lack of interest and concentration.”; “Someone should manage the planning game so it won't be scattered.”; “I didn't figure out how this will work. The planning seems very optimistic.”

At this stage, before actually starting to develop the XP way and learning more about the XP practices, the above reflections can be considered premature. However, they do indicate the level of involvement of the participants and how they grasp most of the planning game benefits while still being suspicious and judgmental.

During the same reflection the participants were asked to describe their personal role in the team and how the planning game can help them perform their role. The roles scheme we use in such workshops, described in details in [2], consists on the XP roles and adds roles that help in the management of a software project. In practice, the scheme implies that all teammates are developers while everyone has in addition a special own role. Table 3 presents participants' feedbacks according to their role.

4.2 The Test-First Practice

Participants filled in a written reflection after their first experience with the test-first practice. Eight participants (out of nine) were in favor of the practice saying it is good, effective and ensures quality. One had no specific position. Four participants (out of nine) reported positive feelings like pleasure, innovation, and comfort, when experienced test-first. Four participants reported negative feelings like frustration, uncertainty, and time overhead. One participant didn't indicate any special feeling.

Participants were asked for the most significant *disadvantage* of the test-first practice. Five participants (out of ten) reported that the most significant disadvantage is that test-first is time consuming; three participants claimed it can lead to the introducing of later changes in the tests; one participant claimed it lacks a global perspective; one claimed it is annoying.

Participants were asked for the most significant *advantage* of the test-first practice. Three participants (out of ten) reported that the most significant advantage is ‘thinking before coding’. Nine participants (out of ten) claimed test-first ensures the existence of tests and that it enables knowing what the minimal code that should be written is.

We have observed two main phenomena when examining the written reflections of the workshop participants and analyzing our observations. The first one is the participants' reference to test-first as a thinking activity in general and a ‘thinking before

Table 3. Planning game reflections according to roles

Personal Role	How can the planning game help you with performing your role?
coach	" – Tasks distribution. – Accountability of all teammates. – Statements regarding times. – Work according to a method. – Coordination of expectations."
tracker	"Getting acquainted with all tasks and time estimations to be used for control."
customer	"Receiving a real situation regarding my requirements – what is possible and when."
in charge of acceptance tests	"The planning game will help me realize the size and scope of the tests."
in charge of continuous integration	"Keeping communication with the customer. Presenting results to the customer, sharing development constraints with the customer…"
in charge of design	"Enables figure exactly the structure of the system and influence it since all tasks can be easily observed."
in charge of infrastructure	"The planning game can help me plan which work and development environment should arranged for the project. Which tools will be used…"
in charge of unit tests	"Getting acquainted with all tasks and understanding functionality before implementation."
in charge of presentations and documentation	"Understanding of the processes in the project."
in charge of code control	"Can help in thinking of possible ways to code…"

coding' activity in particular. This observation indicates that coding is not conceived by the workshop participants as a developer-computer simple interaction, similar to people interaction, but rather, that it requires thinking before performing even when we know what we want to say, i.e. start developing right away. This phenomenon can be explained by the common way developers refer to the coding. Specifically, developers usually start coding in an intuitive way, and therefore see test-first as a practice that enforces them to think prior to the coding.

The second observation is the contradictions and conflicts that the practice of test-first brings with. One participant claims that test-first ensures fewer bugs and consequently leads to shorter integration times. Still, this participant indicates as a disadvantage of test-first that it leads to time overhead. Another participant claims that since we think before coding, we know exactly what we are going to code. At the same time this participant indicates uncertainty as a feeling he feels while practicing the test-first approach. A third participant claims that test-first cuts off the continuity of coding while, at the same time acknowledges the convenience in using the practice. This phenomenon can be explained by using another contradiction inherent in the test-first itself. Developers are used to code and then test while test-first enforces them to perform these activities in the reversed order. Accordingly, experiencing test-first for the first time can cause mixed feelings and contradicting opinions.

4.3 The Pair Programming Practice

Participants filled in a reflection after they had developed in pairs addressing the following open question" "Did you enjoy pair programming?". Eight (out of nine) reported that they enjoy pair programming. One reported "so-so". We asked which trait of your pair is the most appreciated by you. Two (out of eight) appreciate professionalism; two appreciate knowledge and acquaintance with the development environment; and the other four answers were: ability to learn, pedantry, doesn't get into panic and doesn't make pressure, agility.

4.4 Final Workshop Feedback

The final workshop feedback was received using a closed questionnaire in which the participants were asked to mark their agreement level with seven statements in a 1-5 scale (1 - weak agreement, 3 - reasonable agreement and 5 - strong agreement). Table 4 summarizes this feedback.

Table 4. Final workshop feedback of nine participants

Statement	1 slightly agree	2	3 reasonable agree	4	5 greatly agree
The workshop answers my expectations			1	4	4
XP suits me			2	6	1
XP suits my team	3	1	4	1	
XP suits my project	1	3	5		
XP suits my organization	2	3	3	1	
I'll act to assimilate XP in my team	1		5	2	1
If there will be a trial to assimilate XP in my project, I expect resistance	1		3	2	3

Table 4 shows that although participants think that XP suits them and some of them will act to assimilate it in their team, they still feel that XP suits less to their team, project and organization, and expect objections if XP is assimilated. This feedback can be explained by the fact that, indeed, in order to assimilate XP in the project, organizational and conceptual changes should occur.

5 Summary

This paper presents the introducing of XP into a software project at the Israeli Air Force. After conducting the workshop, a methodological specific process to implement XP into the unit was designed.

Recently, as a result of this process, one XP team has started to work according to XP on a new adjacent project which leans on the same infrastructure. Measures were defined and used. A new research has been established in order to assess the XP assimilation process in general and the team performance and progress in particular.

References

1. Beck, K., Extreme Programming Explained: Embrace Change, Addison-Wesley (2000).
2. Dubinsky Y. and Hazzan O., Roles in agile software development teams, 5th Int. Conf. on eXtreme Programming and Agile Processes in Software Engineering (2004) 157-165.
3. Highsmith, J., Agile Software developments Ecosystems, Addison-Wesley (2002).
4. Hazzan, O. and Tomayko, J., The reflective practitioner perspective in eXtreme Programming, Proceedings of the XP Agile Universe (2003) 51-61.
5. Reifer, D. J., How to get the most out of Extreme Programming/Agile Methods., Proceedings of the 2nd XP Universe and the first Agile Universe Conference, (2002) 185-196.

The Agile Journey
Adopting XP in a Large Financial Services Organization

Jeff Nielsen and Dave McMunn

Digital Focus, 13655 Dulles Technology Drive
Herndon, VA 20171, USA
{jeff.nielsen,dave.mcmunn}@digitalfocus.com
http://www.digitalfocus.com

Abstract. On January 14, 2004, two vice presidents met with a group of directors, project managers, and developers, and indicated their desire to embrace agile software development as "the way forward" in their organization. This was not the beginning of this company's adoption of XP and Agile Methodologies, but rather the culmination of almost two and a half years of learning, experimentation, prototyping, and promotion. Making change "stick" in any large organization is problematic, and dramatically changing the way a risk-averse, highly-regulated company develops software requires more than just a successful pilot and a couple of months of coaching. This experience report documents the "agile journey" undertaken by one such corporation between 2001 and 2004. They began by outsourcing a small effort to an XP-proficient consulting firm, and proceeded to use agile techniques on a series of increasingly-significant efforts, allowing sufficient time for the new approach to gain acceptance. In retrospect, all parties involved now believe that the slow, gradual approach to XP adoption – building on incremental successes project by project – was the key to its success.

1 Introduction

Introducing change in a large organization is difficult. While successful prototypes or pilot projects can demonstrate new techniques and ways of working, making any kind of change "stick" in a long-term way requires something more. This report details how one company made a lasting change in the way it develops software. Between 2001 and 2004 this organization transformed itself into one where business and IT collaborate closely to produce new releases of defect-free applications on a quarterly (or more frequent) basis.

The organization in question is part of a large financial services firm. This particular group handles all of the business related to loans for multifamily properties, and will be referred to hereafter simply as "Multifamily."

Multifamily has a dedicated IT staff (currently around 100) that is responsible for both new development and ongoing maintenance. Multifamily has been developing software applications to complement its operational efforts for almost two decades. For much of that time, it followed a traditional, phase-based approach to software development.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 28–37, 2005.
© Springer-Verlag Berlin Heidelberg 2005

But, as in many organizations following this approach, a good deal of tension had developed over the years between the IT and business personnel. IT was frustrated with what it saw as a never-ending succession of requirements changes from business people who could "not make up their minds." Business, for its part, felt the all-too-common frustrations with the pace and cost of development. They had a difficult time understanding why putting their new ideas into production always took longer and cost more than seemed reasonable. As the 21st century began, both business and IT realized that something needed to change.

2 The Agile Answer

Multifamily found its answer in the agile software development movement, which it learned about almost by chance. In the fourth quarter of 2001, the business needed to put a small system in place with as much speed, and as little operational risk, as possible. They engaged Digital Focus to meet that need. While Digital Focus's solution satisfied the demand for rapid, reliable development, it also had the effect of introducing Multifamily to the advantages of an agile approach. Digital Focus had been experimenting with Extreme Programming (XP) for the preceding two years. After initial success, it had made the conversion to XP-style development for all of its client work.

This outsourced project became the first step of a multi-year "agile journey" undertaken by Multifamily. Accepting an iterative, incremental approach at this organization would take time, and would require overcoming some specific challenges. One of these was a bias towards large, multi-year architectural efforts. Another was the sensitive, regulated nature of the environment in which they worked. Had agile-development advocates attempted to effect change all at once or on a large scale, the reception would have been strongly negative, with little chance of long-term success.

With a combination of innovation and patience, however, agile champions at Digital Focus worked with allies in Multifamily to facilitate the introduction of XP. The solution was to start with a modest effort, and then build on small, tangible successes. Through a series of increasingly-significant initiatives over a three-year timeframe, Digital Focus was able to demonstrate the effectiveness of agile software development and continue to reinforce a trust-based partnership with Multifamily. In the inaugural effort of 2001, Digital Focus led the charge while Multifamily observed the results; by 2004, developers at Multifamily had become full-fledged agile experts. Here's how it happened.

3 Acclimating to Agile Development – Project by Project

3.1 Deal Registration (Built October – December, 2001)

Initial Project Expectations. As mentioned above, Multifamily's first exposure to XP was through an outsourcing effort. They needed a tactical application that would provide more visibility into their lending pipeline, and they were

willing to outsource based on the promise of rapidly-developed new functionality delivered on time and in budget.

Digital Focus was contracted to build the application. Multifamily contributed a single developer and tester to the team (to provide some continuity for future deployment and maintenance). Because of the fixed-time, fixed-price contract, Multifamily was less concerned with the specific development process that would be followed. As long as the deliverables were met, Digital Focus was free to execute the project using its own XP-based methodology.

Introducing Agile Practices. Because the particularities of XP were initially downplayed by Digital Focus team members, Multifamily was exposed to agile practices only indirectly through their involvement in the development effort. The critical practices used by the team included co-location in a shared team room, decomposition of the application's functionality into user stories, test-driven development (TDD), continuous integration, incremental design, and ongoing functional testing. The single Multifamily developer was able to experience these practices first-hand and to see how an application could be developed and tested incrementally.

The business-side professionals were also introduced gradually to some of these ideas. A noticeable difference for them was the way that the requirements discussions continued throughout the project. The team's conscious decision to build the application incrementally and to concentrate first on those stories with the highest business value meant that the greatest amount of attention was given to the most significant features. Because of this, the user acceptance testing phase at the end of the project was greatly accelerated, with the application being deployed in just one week.

Result. While the focus of this initial effort was on building and deploying an application rather than introducing agile concepts, exposure to these was a natural side effect. The business began to see how being more involved throughout development could significantly shorten the traditional post-development activities. They also learned that by refining the detailed requirements just as the development of those requirements was about to occur, they could get better feedback and tune the application to more closely match their vision.

Although IT involvement with this project was minimal, they did get their first look at XP in action. While Multifamily IT as a whole was still unsure about the general applicability of agile methodologies, at least one of their developers could now speak to the benefits she had observed firsthand. This exposure to the ideas in a non-threatening way began to generate discussion and awareness.

Finally, this project laid an important foundation of trust between Digital Focus and Multifamily, which would be crucial in moving forward.

3.2 Pricing (Built September, 2002 – January, 2003)

Initial Project Expectations. Midway through 2002, Multifamily began development of an application to provide pricing information about potential deals.

The correct way to build this application was neither obvious nor without controversy, since the pricing process was governed by an extensive set of rules and relied heavily on human skill and judgment. Given the significance of the application and the number of players involved, initial discussions about how the system should behave produced more questions than answers.

Based on their past experience with Digital Focus, Multifamily business again asked for help in working through the requirements and driving the delivery of this particular system. Outsourcing the complete project, however, was not in the plan. Business and IT personnel agreed that Multifamily IT needed to play a significant role. The eventual contract specified that Digital Focus would take charge of the web-based user interface pieces while a Multifamily team would build the pricing "engine." Implicit in this agreement was an understanding that the two teams would work as closely together as possible.

Introducing Agile Practices. After some (persuasive) discussion, it was decided that both teams would co-locate in a development area at Digital Focus. This turned out to be a key decision. The two teams worked for the duration of the project in a large room, which not only enhanced communication, but also gave the Multifamily employees a "front-row seat" to observe the day-by-day execution of XP practices. It wasn't long before they began to appreciate the benefits of working in this fashion and started trying to adopt many of the practices themselves – most importantly story-based planning and iterative delivery. Training occurred naturally, as questions such as "What is a story?" and "How do I size a story?" could be answered and then put into practice right away.

The Multifamily team also worked to adopt the practices of pair programming, automated unit testing, and continuous integration. In terms of the latter, it quickly became apparent that having a single codebase, repository, and build process benefited everyone. The two teams together learned the practice of performing iteration retrospectives. Through the use of the SAMOLO (Same As, More Of, Less Of) technique, they fine-tuned their application of many of the practices – e.g., how often they should pair. More importantly, they learned how to continue improving the way that they worked together.

By the end of the project, the two teams were collaborating to a much larger extent than anyone had imagined would happen. Developers frequently crossed teams to pair on different stories, and producing a quality product became everyone's main focus. Everyone looked forward to alternate Fridays, when both teams celebrated their accomplishments together in an end-of-iteration ceremony (including the tongue-in-cheek "end-of-iteration song") and demonstration.

In interacting with the business, the teams tried to apply the lessons from the previous Deal Registration project. Getting active business participation throughout became a primary focus. For example, rather than deploying the code only once (at the end of the project), the team insisted that the Pricing application be deployed to the Multifamily testing environment after each iteration. The project manager made a personal visit to each of the customer's offices at least once per iteration, to walk them through the latest system features.

Finally, the teams began exposing the business to the concepts of stories and points. Within a fixed point budget, it was understood that the users could choose the exact stories to be included in the final release. And rather than the project manager determining the order of development, the users were encouraged to "drive" – selecting the specific stories to be tackled next. This alignment of authority with responsibility was especially critical for this application, where both requirements and priorities continued to change all through the project.

Result. By the end of the project, important progress had been made on several fronts in the XP adoption process. Multifamily IT had learned that, by using agile software development practices and techniques, they could develop an application in an environment where the requirements were constantly in flux. They learned that beginning development without having all of the requirements "locked down" was not only possible but often desirable. They saw firsthand the benefits of letting the users change their minds – leveraging the continual learning instead of discouraging it.

From the business point of view, the users better understood the importance of being involved throughout development. They also understood the value of moving forward in short cycles, even as requirements were still being defined. And they loved the ability to choose stories and control the release plan. Although there was not enough time to build everything that seemed desirable (as usual), they found that they could work together with IT to come up with creative solutions that both met the budget and satisfied the most important needs.

The successful implementation of Pricing also piqued the interest of IT management. The Multifamily project manager and director had seen the effectiveness of many of the XP practices. They were curious to explore how they might be able to incorporate some of these practices at Multifamily. Doing so would be the next step.

3.3 Waivers (Built April – August, 2003)

Initial Project Expectations. In the spring of 2003, Multifamily had made a commitment to provide a Waivers application in time for a major summer conference – then about four months away. Although a significant analysis and prototyping effort had occurred, both business and IT were apprehensive because two previous attempts to develop a similar application had failed.

Several factors combined to produce the ultimate project makeup. Budgets were tight, which precluded the option of outsourcing; and there was a growing desire to prove that agile development could work in-house at Multifamily. Nevertheless, several Digital Focus personnel had been involved in the analysis and had valuable domain knowledge about the users' needs.

The final decision was to implement the project at Multifamily's offices with the leadership of two Digital Focus coaches. The coaches would drive the project, but would be required to execute within Multifamily's environment, using Multifamily personnel, and integrating more closely with the rest of IT. Furthermore,

as an IT-managed project, the team would be required to demonstrate compliance with the approved corporate methodology.

Introducing Agile Practices. Running an XP-style project on site at Multifamily was challenging, from the simple necessity of finding space for the team to pushing for new uses of the development servers. Although an empty office was located to serve as a team room, it took more than four weeks to get permission to re-configure the furniture appropriately. Being able to build and test daily in multiple environments (contrary to the established usage model for the servers) required the team to exercise both creativity and diplomacy.

Working inside the Multifamily environment also necessitated changes in individual responsibilities. The database administrator, for example, had to get used to the idea of the table structure changing every two weeks. The testers needed to get comfortable with testing new stories daily, during the iteration, rather than waiting for the code to be "done."

Being on site, however, had huge advantages in terms of the team's interaction with the business users. It brought a whole new dynamic and level of visibility. Contact was no longer limited to pre-scheduled meetings. Business people could stop by the team room as often as they wanted, which they did with increasing frequency.

Each visit allowed examination of "the wall," a planning area in the team room which displayed all of the stories on index cards. On the left-hand side of a vertical line of masking tape were stories that the team had capacity to build before the release date. On the right-hand side were those which would not fit. As stories were added, removed, and split, the business people could easily work with the developers to update the release plan (based on the team's velocity) as often as desired.

Finally, being on site allowed in-person participation by the business users in the iteration kickoff and closeout meetings. (The previous projects had required the use of business proxies in these meetings.) The developers heard the requirements for each story in the users' own words and were able to ask questions interactively. The team found it very rewarding to be able to demonstrate the new functionality each iteration to the paying customers. The business users likewise appreciated getting to know the human side of the developers and being able to join in on the end-of-iteration ceremonies.

Result. The Waivers project proved so successful that the overseeing Director of IT became convinced of the benefits of adopting, in at least some form, the main practices of XP. Importantly, he became the XP management champion for IT. The project had proven that agile development could work (with some modifications) within the Multifamily environment and methodology constraints. The various IT specialist groups – testers, database administrators, and configuration managers – had shown that they could work together with the developers in an iterative fashion. Furthermore, all involved recognized the tremendous improvement in communication that occurred when the project team was physically close to the business and could interact with them daily.

The director also better appreciated the challenges he would face trying to bring agile practices into the mainstream at Multifamily. First, it would be challenging to create a shared workspace for each team, since the prevailing office layout at that time was not conducive to this. Second, it was obvious that some re-definition of traditional roles (especially for testers) would need to occur. Finally, having an XP team on site had created a significant amount of "buzz" throughout IT and not all of it was positive. The general perception continued to exist that an agile approach, while great for small projects, could not support the development of significant, real-world applications. The next step, then, was to identify a project that could address this broader concern.

3.4 HCD Front End (November, 2003 – April, 2004)

Initial Project Expectations. The project chosen was an ongoing effort that had been using a RUP-based methodology. Within that team, it was widely recognized that the current process was ineffective. The implementation had fostered separate camps of analysts, developers, and testers (with the accompanying sequence of handoffs); development was proceeding at a slow pace; and relationships among the various parties were extremely strained.

Other factors made this project an ideal platform with which to validate the director's proposed hybrid between Multifamily's existing practices and the agile approach. It was a large, mission-critical application with lots of legacy code and complex business rules. Any new development model would need to be able to work with this type of application, accommodate the skill structure and roles of existing team members, and be able to function under Multifamily leadership.

Digital Focus was asked to support the effort in a coaching role. The mandate was to help Multifamily tailor and refine XP practices to fit within their environment and with their teams. An additional objective was for the coaches to mentor Multifamily leads to be able to take the reins as soon as possible.

Introducing Agile Practices. Digital Focus began by conducting an assessment of the current state of the team. They proposed a tailored implementation plan that would address the highest pain points first. Multifamily management then augmented the existing team with additional developers who were already experienced in the agile approach (from the previous projects), placing them in key leadership positions.

It was not immediately clear how to find a way for this new (much larger) team to sit together. The expanded team included analysts from the Multifamily business side, and some "commuting compromise" was required to settle on a work site. In the end, a couple of walls even had to be knocked down to create team rooms that were large enough.

Introducing the specific agile practices was done iteration by iteration, with training sessions and one-on-one coaching provided by Digital Focus. The first priority was to get a continuous integration build running. Having an automated 10-minute build that could be pushed to the testing environment at will transformed a labor-intensive process into a trivial one. After this, the few existing

unit tests were resurrected, and a norm was established that no new or changed code could be checked in without test coverage. Then came TDD, simple design, coding standards, pair programming, etc. Concurrently, the coaches worked with the analysts and testers to teach them to think in terms of stories, to focus requirements work on the current and upcoming iteration, and to collaborate with the developers in developing a reasonable backlog of sized stories.

All of this met with understandable discomfort at first. It forced people to eliminate the unhealthy (if familiar) habits of interaction between analysts, developers and testers. Rather than following the handoff approach, people had to learn to work together to do "just-in-time requirements" and "just-in-time testing." Business started seeking IT's input on stories and sizing. IT, appropriately, learned how to better leverage verbal communication at the story level in order to refine requirements incrementally.

Early in the effort, the large group was divided into two sub-teams. The sub-teams shared a codebase and iteration schedule, but each had its own set of stories to implement. The sub-team structure was set up to encourage experimentation and accelerate new learning. For instance, teams learned through experience that they really did prefer face-to-face communication over email, even if it meant leaving their cubicles largely vacant. Also, the low-tech means of tracking iteration progress on a wall (with index cards) proved to be better than any computer-based project management software.

Together with the end users, the team discovered the benefits of converting the end-of-iteration meeting from a demo into a true user acceptance test. Now, instead of just watching, business folks were required to *use* the application at the end of each iteration. This greatly increased the quality of the feedback.

Result. As the XP-proficient team members settled into their newly-enlarged rooms, they proved that agile development could scale to multiple teams. It could work within the Multifamily environment and produce great results even on a large, complex application – starting from a legacy codebase. Even conformance to the corporate methodology guidelines could be accomplished in an agile way.

On a human level, team members themselves showed renewed enthusiasm for the development process. They forged much more effective relationships, and became more aware of how the contributions of each role were important in quickly producing useful, tested software. This new familiarity allowed them to expand their own skills, enhance their interactions, and develop a deeper level of friendship.

3.5 Continuing Progress (April 2004 – Present)

Multifamily has continued using its customized agile software development process, and this approach has begun to radiate out to other parts of the organization. While the introduction of specific agile practices has met with much success, the greater triumph has been the adoption of an agile *mindset* with increasing enthusiasm throughout Multifamily. Rather than merely trying to

create a new methodology, the emphasis has shifted to enhancing communication, feedback, and human interaction. The agile journey has changed the way that both business and IT think about software development.

An important effect of this effort has been a renewed focus on continuous process improvement. The practice of regular iteration retrospectives has continued, which encourages ongoing innovation and allows each team to optimize its execution. The need for external coaches has disappeared accordingly.

Just how well have the teams performed on their own? The "proof is in the pudding." In mid-2004, one of the Multifamily IT teams tackled a new implementation of the Applicant Experience Check application. Like Deal Registration three years before, the current technology didn't fully meet the business need and required numerous workarounds. In a period of three months the team of all Multifamily employees designed, delivered, and deployed a complete application that delighted the business users. No formal requirements phase (or documentation) was attempted or required, deployment took only one week after development was complete, and to date, not a single defect has been reported. Clearly, this is an IT team that embodies agile software development.

4 Conclusion

In little more than three years, Multifamily has witnessed its evolution into an organization that has embraced a nimble, dynamic, *agile* software development methodology. Theirs is a story of successful, lasting organizational change. They are proof that a large company can, with a measured introduction, dramatically enhance its approach to software development.

Multifamily's experience with adopting XP provides several lessons for other large organizations that wish to institutionalize agile software development.

- The slow, gradual approach was the key to success at Multifamily. Beginning with a cautious experiment and then proceeding to use agile techniques on a series of increasingly-significant efforts allowed sufficient time for new concepts to sink in and for new ways of working to gain acceptance.
- In addition to time, large companies need *proof* that an agile software process will work in their particular situation. They need this proof before they will begin to invest heavily in change.
- Working with business and management in addition to developers is imperative. This "three-pronged attack" is key, as is having a management champion within both IT and business.
- Both business and IT have their own sets of fears that need to be addressed. Multifamily's business leaders needed to learn that, ironically, they could be more successful at getting the systems they needed on time and on budget by *not* attempting to specify all the details of the requirements and schedule up front. IT, for its part, needed to be convinced that an iterative, incremental approach could produce high-quality systems and work in their environment.

It is interesting to note that Multifamily adopted XP in an incremental, evolutionary fashion. Each small step produced both the learning and the confidence

Table 1. Overview of Multifamily's agile journey

Project	Timeframe	Structure	Practice Introduction
Deal Registration	Oct. 2001 – Dec. 2001	Off-site at Digital Focus. Single DF team with one developer and one tester from Multifamily. Fixed-scope contract.	- Co-location in team room - Stories on cards - Project tracking (stories and tasks) on the wall - 2-week iterations - 100% unit testing coverage w/JUnit - Test-Driven Development (TDD) - Continuous integration - Simple design/refactoring
Pricing	Sep. 2002 – Jan. 2003	Off-site at Digital Focus. One DF team plus one complete Multifamily IT team (co-located). Fixed-scope contract for DF team.	- Pair programming - Coordinated teams - Coding standard & collective code ownership - Automated functional testing story-by-story - Release planning with story tradeoffs - Delivering code from each iteration to client - End users viewing system after each iteration - End-of-iteration demo, celebration, and song - Iteration retrospectives (SAMOLO)
Waivers	Apr. 2003 – Aug. 2003	On-site at Multifamily. Multifamily development team, testers, DBAs, etc., with two Digital Focus coaches in leadership roles.	- On-site team room - Flexible scope release planning each iteration - Direct customer participation in iteration kickoff - Customer involvement in end-of-iteration mtg. - Intensive pair programming - Daily business involvement - DBA integrated with team and schedule - Mapping of practices to regulatory requirements
HCD Front End	Nov. 2003 – Apr. 2004	On-site at Multifamily. Existing Multifamily project team split into two sub-teams. Digital Focus coaches in advisory roles.	- End-of-iteration user acceptance testing - Just-in-time requirements - Analysts and testers integrated with team - Scaling XP to multiple sub teams - Using Wiki (FitNesse) for functional tests - Working with legacy code

needed to be able to take the next one. Although it was an uphill battle, a slow, steady climb up that hill proved to be much more effective than any attempt to sprint to the top.

Perhaps the sentiments expressed by the senior leadership of Multifamily say it best. At the beginning of 2004, a pair of vice presidents met with a group of directors, project managers and developers, and indicated their desire to embrace agile software development as "the way forward," noting that they believed it was the "only way to make progress in [their] environment of constantly-changing priorities." On the heels of that insight, the director who had been the primary change advocate and who had witnessed the slow, steady acceptance of XP at Multifamily, added: "If anyone ever tries to make me go back to the old way of building software, I am quitting."

From User Stories to Code in One Day?

Michał Śmiałek

Warsaw University of Technology and Infovide S.A., Warsaw, Poland
smialek@iem.pw.edu.pl

Abstract. User stories in software engineering serve the purpose of discovering requirements and are used as units of system development. When applying stories in a project, two elements seem to be crucial: the ability to write coherent sequences of events and the ability to transform these sequences into code quickly and resourcefully. In this paper, these qualities are reflected in a notation that can be described as "stories with notions". This notation separates the story's sequence of events from the description of terms used in this sequence. Such a formal separation does not limit and rather enhances invention, at the same time rising the level of consistence, and facilitating translation into models of code. This translation maps domain notions into static code constructs (classes, interfaces) and also maps stories into dynamic sequences of messages. With such a mapping, programming becomes equivalent to skilled transformation of user stories, thus giving shorter development cycles.

1 Introduction

In Poland there is well known a novel written by Eliza Orzeszkowa, called "On the Niemen River". It is full of beautiful descriptions of the Polish-Lithuanian countryside around the Niemen river in the XIXth century. Of course, as for every normal novel, it also tells some story. The story is very suggestive and coherent, as all the characters move around this well-described piece of countryside, with rivers, lakes, roads, villages and mansions. The majority of Polish students remember this novel because of these rich descriptions of the nature.

At this moment one might ask what is the relation of Orzeszkowa's novel to software development. We shall try to argue that the relation is significant, especially in the area of requirements specification. Can we imagine a novel which has just the story and no description of the environment (people, places, landscape, and so on)? And when we quickly move to software engineering; can we imagine requirements specification that has just the story? An obvious answer is – no! Of course, when specifying the requirements for a software system we need to tell the developers a story which can be defined as an "account of incidents or events"[1] happening between a user (a role) and the system. However, this story has to be supported by descriptions of all the notions used therein.

Unfortunately, a majority of story writers in software engineering tend to mix stories with the descriptions of notions handled by their systems. These

[1] Merriam-Webster On-line Dictionary

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 38–47, 2005.
© Springer-Verlag Berlin Heidelberg 2005

notion definitions are buried somewhere inside the stories. What is worse – the same notions are often described inconsistently in different stories (see [1]). This inconsistency is more or less acceptable when we write a novel, while it is totally disastrous when specifying a software system.

The requirements specification in software engineering has a clear purpose: to produce code that satisfies the **real** needs of the client. Supposing that we can discover these needs through stories with separated notions, we are still left with an important question: how to make code out of them? It seems obvious that we somehow need to transform stories into sequences of instructions in code, but what about the "descriptions of the nature"? Can we define some process of transforming stories and notions into code? Can this process be automated?

In this paper we will try to answer the above questions. The paper proposes a notation for user stories [2] and notions that allows for easy translation into design level models and code. It also defines appropriate transformations. It is also argued that keeping the transformation mappings allows for more agility in treating the constantly changing user requirements.

2 Communicating the Users with the Developers

One of the fundamental practices of agile software development is close and constant cooperation of developers with the users. We can call such relations as "a cooperative game of invention and communication" [3]. However, we still have to bear in mind that clients and developers have their own backgrounds and their own points of view. A good way of communicating the users with the developers is to tell stories. Though, while most people like to listen to stories, only few are able in telling them well. Let's now ask a question in the context of software development: what would users and developers require from the stories? Gathering answers to this question would allow us to design suitable notation for the stories and determine the way they can be used in the development lifecycle.

Basically, the user wants to hear from the developer a story about "how the system shall work". The story should be communicated in a common language using simple sentences. The story should be a "real story" – with a starting event and with a "happy" or "sad" ending. These sentences should use well defined notions from the user's domain vocabulary. The notion definitions should be easily accessible when needed (not buried somewhere in other stories). The story should not use any special keywords or formal constructs. We should be able to group several stories that lead to a single goal from the user's point of view [4]. The users are usually not good in writing stories. They need someone that would listen to their stories and then write them down in some possibly standard way. Then, they can read the stories, judge them and suggest corrections.

The developers need stories to determine single units of the system's behavior to be developed. For them, it would be ideal if the stories were written as temporal sequences of interactions between roles and the developed system. Notions used in the stories should be easily transformable into design and code constructs (classes, interfaces, etc.) Stories should be formed of sentences that

allow for easy translation into sequences of code instructions. Moreover, when a story changes, it should be easy to trace the change into code. Developers that design systems and write code are often reluctant to write stories. However, they can easily verify quality of the stories from the design point of view. They are good at spotting inconsistencies, ambiguities and generally "holes" in the stories.

To write good stories we thus need good "story writers". We need to find them in our user or development team (and it seems not to be that easy). We also need to give them a "toolbox" which is necessary in order to establish a proper communication path. This toolbox should contain a notation for stories and tools to write them down. Developers would also be happy with a tool that could support translation of the stories into code.

3 Notation and Tools for Coherent User Stories

How to write a good story that would suit both the users and developers? Good stories, as they were written for centuries (see Orzeszkowa's novel), constitute a balance between describing the sequence of events and describing the environment. Good stories are also written in a coherent style.

When writing stories for a software system, we define a dialog between a user and the developed system (see [5] for an insight on such essential dialog). A story is a sequence of interactions between them. A good style here would be to describe these interactions with the simplest possible sentences, like:

- Student enters the semester.
- Teacher accepts the current marks.
- System assigns the student to the new semester.

These simplest sentences contain just the bare minimum for a full sentence: a subject, a verb, and one or two objects. What else do we need to tell a story? Well, of course we need some explanation of what do all the terms used in these sentences mean. Sometimes this may be quite obvious, but in many cases it is crucial to define them. For instance: the semester. One might think that it is simply a number between 1 and 10 denoting the level of studies in a five-year MSc program (like: I'm on the 5th semester). But maybe we are wrong, maybe it is the current academic half-year (like: the current semester is winter 2004/2005)?

Subject-verb-object (SVO) sentences are good at telling the sequence of events, but they are not appropriate for describing the environment of our story. So, what should we do? Should we allow for "SVO's with descriptions"? Experience shows that this is not a good idea. Writers are then tempted to write something like:

- Student enters the semester where the semester is a number between 1 and 10 denoting the level od studies.
  and somewhere else:
- Dean accepts the semester for the new student, where the semester is a number denoting the current student's status.

These two sentences are usually written in separate stories by separate writers or there is some period of time between writing them. The problem is – which

of these slightly different definitions is correct? Another problem is – how to find out that we in fact have two different definitions of the same notion? The solution is quite obvious: write only SVO sentences to describe the sequence of events and keep the notion descriptions in a separate vocabulary. This leads to significant improvement in coherence:

- Student enters the semester.
- Dean accepts the semester for the new student.
  where:
- Semester – number between 1 and 10 denoting the level of studies and the current student's status.

Having a notation for sentences we are now ready to tell full stories. These stories should form complete sequences of events. They should start with an initial interaction from a user and end with a final goal that gives the user a significant value. Requirements that contain such stories are perfectly suited for incremental development. They carry value for the users, and at the same time, they can be treated by the developers as atomic pieces of development. In every iteration we can now create increments based on stories like (see also Fig. 1, 2):

1. Student wants to select lectures.
2. System shows a list of possible lectures.
3. Student selects a lecture from the list of possible lectures.
4. System assigns the lecture to the student.

While writing the stories we constantly extend our vocabulary of notions. We describe all the sentence objects that might cause ambiguity when developing the stories. We also define relations between the notions. This gives us a static map of the "user's territory". We can write individual notions on index cards or we can draw simple class diagrams like the one shown on Figure 1 (see also [6]), consistent with Agile Modeling practices (like: Create Simple Content; [7]). These diagrams additionally extend and clarify our definition of semester:

- Semester – contains a number between 1 and 10 denoting the level of studies and the current student's status; has an associated **course** and a **list of possible lectures**.
  where:
- Course – contains several **semesters**; courses are taken by the **students**.
- List of possible lectures – a list of **lectures** associated with a given **semester**.

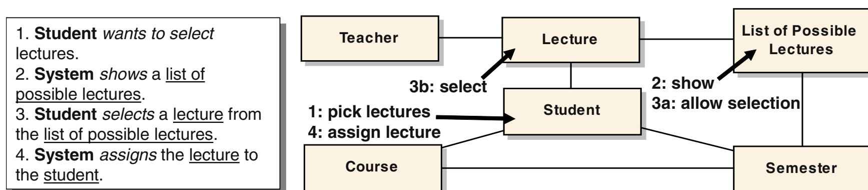


Fig. 1. Navigation through the domain for the "select lecture" story

User stories written with the above notation have two significant characteristics: they can be easily kept coherent and they can be easily transformed into design and code. Coherence of such stories lies in the fact that all of them are based on the same domain description. Notion definitions form a map of the territory that "glues together" functionality which sometimes seems totally independent. Stories only navigate through this static map giving it the necessary element of functionality, as illustrated on Figures 1 and 2. It verifies that all the notions used in our two stories are properly used. Note, that we have actually discovered that there are some inconsistencies in the stories. The story writer forgot that the system needs to determine the semester, before the student selects a lecture and that the dean should select the semester when adding a new lecture.

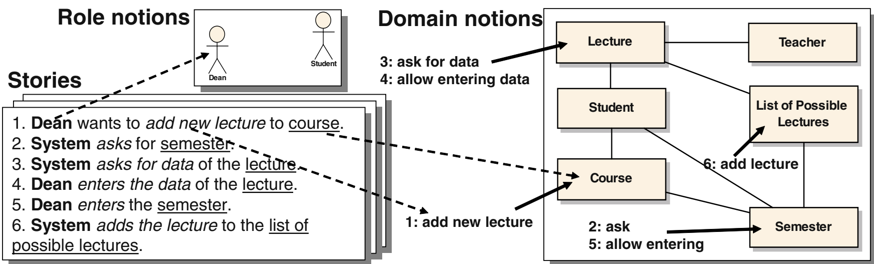


Fig. 2. Model of requirements based on "stories with notions"

Finding such inconsistencies is a difficult task having a typically sized system. We have hundreds of stories and tenths of notions to verify. It seems obvious that we need a tool to support our efforts. We can use the simplest possible "tool" – index cards with notions arranged on a wall. They can be manipulated easily and can be used to "play stories" as illustrated above. A step further would be to have this repository of notions and stories managed with an automated tool. This tool would allow us to organize sentences into stories, and hyperlink subjects, verbs and objects to appropriate elements of the vocabulary (see [8] and [9] for an example and a more detailed idea of such a tool). The stories and the vocabulary organized through hyperlinks, form in fact a model of requirements (Figure 2). We can call this model – "stories with notions". In the model, sentence subjects are linked to notions denoting roles for the system. Sentence objects have links with individual notions of the problem domain, and verbs denote operations on these notions. A model organized in a tool as described above has an important characteristic – it is ready to be transformed into a code model.

4 Getting from SVO User Stories to Code

Users write stories to tell developers what they need. Developers write stories to clarify that they have understood users correctly. Users and developers write

stories together to make sure that the final system will be correct (i.e. will satisfy the **real** needs of the user). This means, that we can in fact have various kinds of stories.

- initial user stories – simple statements from the users that reflect their wishes ("user wishes"),
- clarified user stories – more elaborated stories that contain more details about the functionality of the developed system,
- test stories – detailed stories with added test data, written for the purpose of acceptance testing.

All of these kinds of stories can be written using the SVO format. The initial stories can be written as just one to four SVO sentences. The clarified stories usually add to the initial stories more sentences (more details of the functionality) and add notions (details of the problem domain). These clarified stories can be "adorned" with test data to form test stories.

While the initial stories can jump-start elicitation of user needs, the more detailed stories are the starting point for all the development efforts. This leads us to defining a simple development cycle that uses SVO stories. A single iteration in such a lifecycle might look (in a simplified form) as follows (Fig. 3).

- The users write initial stories.
- The users meet with developers during a story writing session. Together they write clarified stories with notions.
- The developers translate the clarified stories into code. Coding is supported by class model derived from notions and sequence model derived from stories. During development, the stories are clarified with the users whenever necessary. Test stories are also written.
- Developers make sure that test stories are fulfilled and hand the system to the users.
- Users verify the system and possibly write corrected initial stories. They also write new initial stories. The lifecycle loop closes.

The simplest and possibly most efficient way to capture SVO stories and notions during story writing sessions are index cards. However, having a large set of stories and associated notions it seems worthwile to use an automated tool to support story clarification during development. Such a tool (mentioned in the previous section) can help organizing stories and keeping the overall model coherent.

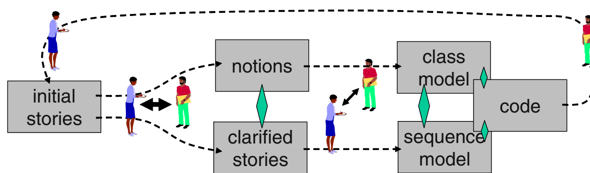


Fig. 3. Software lifecycle involving SVO stories and notions

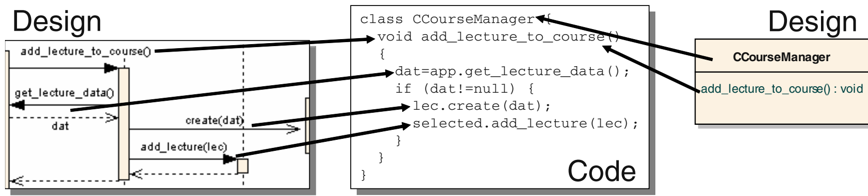


Fig. 4. Visual design models adding important abstraction level to code

As it can be noted on Figure 3, the overall development effort has been split into two groups of human activities. The first group involves translating the actual stories (treated as sequences of events) into sequences of instructions in code. The second group translates the domain notions into static code elements (classes). It is important to note that this translation can be done with the sole use of a typical programming environment only for the most trivial systems. Average systems are complex enough to necessitate some way of taming this complexity. In our approach, this taming is done through UML [10] class and sequence diagrams. These diagrams show the structure of code and its dynamics in a visual form (see Fig. 4). In most cases, diagrams hand-drawn on a white-board suffice. It can be also very beneficial to use an automated CASE tool (chosen from several on the market), integrated into our programming environment. This integration is essential, as only then it relieves us from the burden of actually synchronizing the pictures with code, and gives significant advantage over hand drawn pictures.

It has to be stressed that the use of UML CASE tools has to be done with great care. UML 2 has thirteen different types of diagrams. Extensive use of these diagrams might cause a severe "UML fever" (see [11] for an excellent survey of possible fevers). It seems from the current experience (several "clinical tests" were made) that applying the above lifecycle with class and sequence diagrams does not cause the UML fever. This is supposedly due to the fact that the diagrams in the lifecycle are drawn for a very specific purpose, which is to support structuring code in a clear manner. With CASE tool support, visual (graphical) documentation is created automatically while coding, similarly to using eg. JavaDoc. This makes creating additional heavy documentation completely unnecessary. At the same time it supplies the developers instantly with an additional level of abstraction that enhances comprehension of code.

The design model based on class and sequence diagrams has one more advantage. It can be linked directly with the requirements model based on SVO stories with notions. Keeping these links is important when handling changes in user needs. When SVO stories or notions change (and change the associated acceptance tests) we have direct visual pointers to places in code that need to be updated. These links can be kept simply by assigning appropriate index cards to appropriate hand drawn diagrams. SVO story cards are linked to sequence diagrams, and notion cards are linked to class diagrams (see. Fig. 5). With CASE

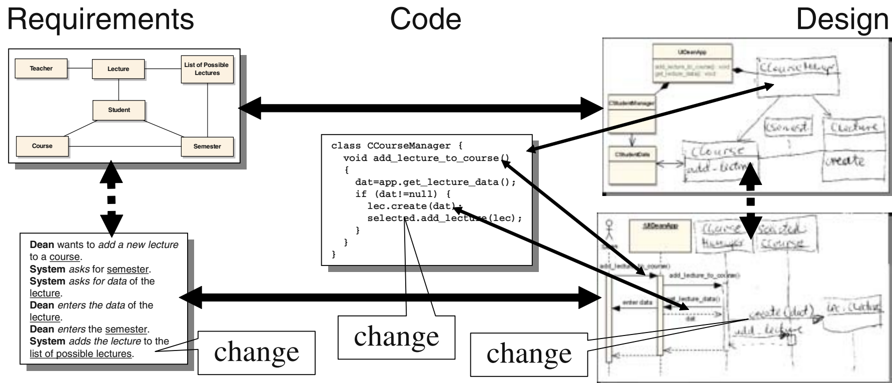


Fig. 5. Links between requirements and design pointing unambiguously to code

tool support, these links can be managed automatically thus supporting the human efforts. Keeping links between SVO stories and design diagrams can be treated as a version of Agile MDA [12]. It is a skilled (human-led) transformation between an inclusive model (user stories + notions) and the design model (class and sequence diagrams). The current approach can also be compared to Property-Driven Development [13], however, here the sequence diagrams are developed as part of the design model, not the requirements model (SVO stories are used instead).

It can be noted that the presented lifecycle uses practices already present in XP [14] and FDD [15]. SVO stories can be best compared to XP's user stories, with added SVO notation. The concept of structuring the initial user needs can be found in FDD, where features (and feature sets) have a very precise notation. The story writing session is closely related to the XP's planning game. Translating stories into design and then to code can be found as two major activities of FDD (design by feature, build by feature). Here, they are applied not to features as in FDD, but to SVO stories. Comparison with FDD can also show that actually the notion model is a simplified version of the "overall object model" (without attributes and operations). The two UML models used to design and document code are directly taken from FDD. UML models can also be applied with success in XP (see eg. [16]).

"Clinical tests" in a student lab project were made to verify that the method cures the UML fever. The students trained in UML were formed in groups of around 12 (around 7 groups in a year). These groups were assigned to develop a system during a one-semester lab. For three consecutive years, the lifecycle used during the lab changed from iterative, use case [4] driven with extensive UML models, through story-based with FDD [15] lifecycle, to SVO story-based. The SVO-based lifecycle seemed to be best suited for the less experienced developers like students. It resulted with higher interaction between the students and the "users" (i.e. the tutors) and much clearer code (see [16]). The final systems were more functional and more compatible with the real needs of the client.

5 Conclusions

Author's experience shows that catching the UML fever in software development is quite common. Most of the consulted development organizations were initially very eager to use automatic CASE tools that support UML notation and generate code automatically (well, almost automatically...). These tools looked like "silver bullets" for all their problems. They promised shorter (maybe one day long?) development cycles. Unfortunately, applying CASE tools in a documentation heavy environment resulted in a "look how much documentation we can now produce" syndrome. The organizations that wanted to change their development practices got bogged down in creating detailed analytical models supported by heavy architectural studies. This resulted in rejecting the tools as adding more work and returning to previous practices.

The approach presented in this paper seems to offer a cure for organizations caught by the UML fever. It offers a lightweight process, where UML diagrams are treated with great care. Tools are used only to support the actual development of working code by giving instant design level models. These models are very distinct from heavy documentation and give the advantage of having a higher level of abstraction (like XP's metaphors, and simple design diagrams). Moreover, these design models can be directly linked to the requirements models. Clear separation of structure from dynamics results in better communication among developers and between developers and users. This separation is done already on the requirements level and thus allows for clear distinction of activities that lead to implementing the domain structure (notions) from those that implement the system's dynamics (SVO stories). This distinction also supports invention and discovery. Many valuable notions can be discovered when SVO stories are used. It can be argued that supporting human activities with a clear path from constantly changing user needs to code, and with some simple transformation tools, could lead in the future to real reduction in development times (one day cycles?). However, this is still to come...

References

1. Breitman, K., Leite, J.: Managing user stories. In: International Workshop on Time-Constrained Requirements Engineering 2002 (TCRE'02), http://www.enel.ucalgary.ca/tcre02/ (2002)
2. Cohn, M.: User Stories Applied. Addison-Wesley (2004)
3. Cockburn, A.: Agile Software Development. Addison-Wesley (2002)
4. Cockburn, A.: Structuring use cases with goals. Journal of Object-Oriented Programming **5** (1997) 56–62
5. Constantine, L.L.: What do users want? Engineering usability into software. Windows Tech Journal (1995) revised in 2000, http://www.foruse.com/articles/whatusers.htm.
6. Ambler, S.W.: Agile data modeling. http://www.agiledata.org/essays/agileDataModeling.html (2005)
7. Ambler, S.W.: Agile Modeling (AM) practices v2. http://www.agilemodeling.com/practices.htm (2005)

8. Gryczon, P., Stańczuk, P.: Obiektowy system konstrukcji scenariuszy przypadków uzycia (Object-oriented use case scenario construction system). Master's thesis, Warsaw University of Technology (2002)
9. Śmiałek, M.: Profile suite for model transformations on the computation independent level. Lecture Notes on Computer Science **3297** (2005) 269–272
10. Fowler, M., Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman (2000)
11. Bell, A.E.: Death by UML fever. Queue **2** (2004) 72–80
12. Ambler, S.W.: A roadmap for Agile MDA. http://www.agilemodeling.com/ essays/agileMDA.htm (2005)
13. Baumeister, H., Knapp, A., Wirsing, M.: Property-driven development. In Cuellar, J.R., Liu, Z., eds.: Proc. 2nd IEEE Int. Conf. Software Engineering and Formal Methods (SEFM'04), IEEE Computer Society Press (2004)
14. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
15. Palmer, S.R., Felsing, J.M.: A Practical Guide to Feature-Driven Development. Prentice Hall PTR (2002)
16. Astels, D.: Refactoring with UML. In: XP 2002, The Third International Conference on eXtreme Programming. (2002) http://www.xp2003.org/xp2002/.

Evaluate XP Effectiveness Using Simulation Modeling*

Alessandra Cau, Giulio Concas, Marco Melis, and Ivana Turnu

DIEE, Universitá di Cagliari
{alessandra.cau,concas,marco.melis,ivana.turnu}@diee.unica.it
http://agile.diee.unica.it

Abstract. Effectively evaluating the capability of a software development methodology has always been very difficult, owing to the number and variability of factors to control. Evaluating XP is by no way different under this respect. In this paper we present a simulation approach to evaluate the applicability and effectiveness of XP process, and the effects of some of its individual practices. Such approaches using simulation are increasing popular because they are inexpensive and flexible. Of course, they need to be calibrated with real data and complemented with empirical research.
The XP process has been modelled and a simulation executive has been written, enabling to simulate XP software development activities. The model follows an object-oriented approach, and has been implemented in Smalltalk language, following XP process itself. It is able to vary the usage level of some XP practices and to simulate how all the project entities evolve consequently.

1 Introduction

The objective evaluation of a particular technology or methodology of software development has been studied for many years by empirical research. The main issue is that it is practically impossible to assert that a particular development methodology is better than another. In fact, the effectiveness of a particular development technique is influenced by many factors in continuous change.

This is even more obvious in the case of software development in which there is a strong influence of the human factor. For example, the experience of a particular development team can influence the result of a project. Experience is one of those factors that evolves continuously in time and makes impossible the objective evaluation of two development techniques. We would have to carry the same project in parallel under the same conditions, using the same persons and varying only the methodology.
It is practically impossible!

* This work was supported by MAPS (Agile Methodologies for Software Production) research project, contract/grant sponsor: FIRB research fund of MIUR, contract/grant number: RBNE01JRK8.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 48–56, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Some answers to this problem come from empirical research. However, it is not efficient or even possible to conduct empirical studies for a large number of context parameter variations. Also, empirical studies are extremely costly and cannot be performed with such a large degree of completeness.

One of the possible partial solutions found from researchers is to use simulation to estimate and verify the effectiveness of a particular development process. It is a partial solution because the results are always influenced by the process model itself, which is always a strong simplification of the real world. However, a well calibrated and validated simulator can provide a lot of information on the efficacy of a given methodology.

In recent years, a new lightweight methodology of software development has become very popular: Extreme Programming (XP) [1]. Academics and practitioners of the software engineering field need to assess the XP practices' efficacy with quantitative results. Works that report quantitative results are still very scarce and are based on empirical studies. Particularly significant is an experimental research conducted by Williams et al. [2] in which they conclude empirically that one of the most important XP practices – Pair Programming – increases the development cost by 15%, but it is repaid in shorter and less expensive testing, quality assurance and field support.

In this research, we have developed a simulation model in order to evaluate the applicability and efficiency of XP process, and the effects of some of its individual practices on project results.

The remainder of the paper is organized as follows: in section 2 is presented the state of the art of the simulation modeling of agile software processes, XP ones in particular. Section 3 describes the details of the model proposed. Section 4 illustrates how verification and validation of the simulator have been approached while the section 5 shows some results of our research.

2 Related Work

In recent years, agile methodologies for software development have become increasingly popular. Such methodologies must be tested in order to validate their effectiveness. The simulation represents a powerful tool in order to test new methodologies, avoiding or postponing experimentations in the real world [3]. In spite of the great diffusion of Extreme Programming (XP) in academic and industrial field, only recently the first attempts of XP processes simulation have appeared, all using the System Dynamics approach. Here we cite some significant contributions.

In [4] Cao proposed a system dynamic simulation model to investigate the applicability and effectiveness of agile methods and to examine the impact of agile practices on project performance in terms of quality, schedule, cost and customer satisfaction.

Misic et al. [5] investigated the possibility of using system dynamics to model, and subsequently simulate and analyze, the software development process of the XP software development process. In particular, they considered the effects of

four practices of this methodology: pair programming, refactoring, test-driven development, and small developmental iterations.

In [6] Kuppuswami et al. proposed a system dynamics simulation model of the XP development process to show the constant nature of the cost of change curve that is one of the most important claimed benefits of XP. They also described the steps to be followed to construct a cost of change curve using the simulation model.

Perhaps one of the most relevant works was made by Kuppuswami et al. [7]. They developed a system dynamics simulation model to analyze the effects of the XP practices on software development effort. The model developed was simulated for a typical XP project of the size of 50 User Stories and the effects of the individual practices were computed. The results indicated a reduction in software development cost by enhancing the usage levels of individual XP practices.

3 The Proposed Model

The most important goal of our work is to quantitatively evaluate the XP methodology effectiveness varying the usage level of its practices. Based on the considerations carried out in section 1, we have chosen the simulation modelling approach.

The model we are developing already implements a number of XP practices – Pair Programming, Test Driven Development, Small Releases, Planning Game, Code Ownership – and is able to vary the usage level of some of them, such as TDD and Pair Programming, and the size of Releases and Iterations. In this way, the user could vary the usage level of an XP practice to evaluate its effectiveness and the impact on the process in terms of quality, costs, time, etc, and see how the modelled entities evolve consequently.

3.1 Model Description

The model is characterized by several activities (Release Planning, Development Session, etc). The inputs to these activities are entities (User Stories, Integrated code, etc) that are modified and created by other instances of activities. The class diagram in figure 1 shows the relationships among the high-level entities of the XP process model. The activities are eventually composed of sub-activities such as the *user story estimation* activity. Each activity is executed by one or more actors of the process. The identified actors are the Team, made up of Developers, the Customer and the Manager. Each actor has some attributes, which vary in time, and can perform a number of actions. These actions can be performed in cooperation with other actors (two developers working in pair-programming) in order to carry out a particular activity (see section 3.2).

The time granularity of our model is that of a development session, which is typically of a couple of hours. The equations that regulate the variations on the model entities and the execution of each activity have been taken from existing

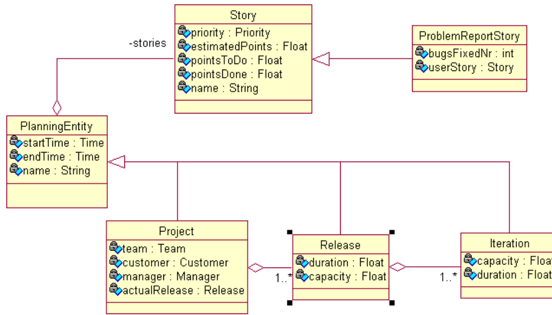


Fig. 1. Class diagram of some of the high-level model entities

models, empirical data and, where necessary, from authors assumptions. About the statistical distributions used in the model we have mostly chosen gaussian and log-normal functions.

3.2 Model Dynamics

The project starts with an initial number of User Storys (USs), which identify the main requirements of the project and represent a preliminary evaluation of the project's size. These USs are prioritized by the Customer and subsequently estimated by the Team using values taken from statistical distributions.

This estimate is affected by a stochastic error which is decreased by the overall experience of the Team on the project. When an US estimation exceeds a certain limit (a portion of the Iteration capacity) it will be splitted into two or more USs. The next phase consists of choosing the USs which will be implemented for the next Release and consequently assigned to a specific Iteration.

During an Iteration, design and development of the scheduled USs is performed. This activity produces the source code required to implement the functionality described by each US. The produced code is characterized by size (in terms of number of classes, methods and locs) and quality (number of defects).

The time actually spent to implement each US is affected by the estimation error and by the velocity of the developers who have worked on it[1]. In addition, it is influenced also by adoption levels of *Pair Programming* and *Test Driven Development (TDD)* practices.

In some cases not all the planned USs are completed within an Iteration. These USs are planned again and implemented in next Iterations. Moreover, the Customer can write new USs and possibly report problems that he has found after each release of the system.

After the end of each Release the Customer can report a number of problems he/she has found on the project released up to that moment. This number

[1] Developers are statistically different from each other in terms of initial skill and initial velocity. These attributes increases in accordance to the experience gained on the project

is related to the defect density of the system. These reports are planned by the Team as the other USs (ProblemReportStory (PR-US)) each of which has an associated US affected by the problem founded by the Customer. The implementation of each PR-US has the effect of reducing the number of defects of the related US.

For the sake of brevity, we only report an informal description of the Development Session activity.

Development Session. During a development session, characterized by a certain duration taken from a statistical distribution, a Developer develops the code relating to a particular User Story. In an XP project the development session would be normally performed by two developers working together at a single computer (Pair Programming). However, this practice is rarely adopted completely. For this reason, our model has an input parameter called *Pair Programming Adoption* that indicates the percentage of usage of this practice. This parameter gives the probability at which a Development Session will be performed by two developers instead of one.

A Development Session performed in pair programming is more efficiently than a "solo-programming" in terms of the time needed to implement a single User Story, defects injected (due to the continuous review made by the pair developer [2]), learning efficiency (the developer skill increases faster if one works together with another developer [8], [9]).

In particular we have made the assumption that the velocity of a pair of Developers is given by the average values of each developer velocity increased by 40%, as found experimentally in some empirical researches [2]). Moreover, in these studies it has been found that the defects injected during a pair-session is less than that of a "solo-programming". So, we model this behavior imposing that the maximum number of bugs injected during a Development Session activity is dictated by the best developer of the pair in terms of skill.

In an XP project Developers should write the code with the relating unit tests. Also in this case the model has a parameter called *Tdd Adoption* that accounts for the level of adoption of this practice. This parameter decreases the velocity of the development session and the number of the defects injected [10].

At the end of a Development Session activity, new code is produced and existing code is modified, introducing inevitably a certain number of defects. The level of these changes are affected by stochastic variables influenced by both Developers' attributes (experience and skill) and the usage levels of individual XP practices (*Testing* and *Pair Programming*).

4 Verification and Validation of the Model

One of the major problems in process simulation is the effective calibration and validation of the developed simulator. In order to reach this goal, data sets gathered in real projects are needed. However, these data are difficult to obtain for several reasons. The greater part of real projects are developed inside

privately-owned companies that, for obvious reasons, are generally reluctant to publish data regarding their inner development process.

We can cite two XP projects where tracking activity has been conducted systematically and whose data are available at a sufficient level of detail: *Repo Margining System* [11] and *Market Info* [12].

In order to calibrate the parameters of the simulation model, we have used some input variables coming from the *Repo Margining System* project [11], such as the number of developers, the release duration and so on. Also, we have used the project and process data gathered during the first iteration. We then simulated the evolution of the project starting from the second iteration.

With these input parameters a number of simulation runs have been performed. Then, we have iteratively calibrated the model parameters in order to better fit the final results of the real project. In table 1 the simulation outputs are compared with the ones taken from the *Repo Margining System* case study.

Table 1. Comparison between simulation results averaged on 200 runs and the Repo Margining System case study. Standard deviations are reported in parenthesiss. A story point corresponds to 30 minutes of work

Output variable	Simulation	Real Project
Total days of Development	60,2 (19,6)	60
Number of completed User Stories	28,6 (6,1)	29
Estimated Effort [Story points]	470,6 (145,6)	474
Actual Effort [Story points]	803,1 (254,0)	793
Number of Releases	2,4 (0,7)	2
Number of Iterations per Release	2,6 (0,3)	3
Developed Classes	243,8 (90,7)	251
Developed Methods	1066,3 (396,8)	1056
DSI	15234,2 (5669,3)	15543

A conceptual model validation has been done interviewing some individuals familiar with the XP process itself. The proposed approach was presented, and its various concepts – roles, activities and artifacts – were explained in detail. The collected feedback on our approach was positive.

Also, we performed an event validation process comparing the sequence of the events produced by the simulation with those of a real XP process.

As regards the verification of the correctness of the simulator, we implemented the system using pair-programming. Following this practice, a continuous review was made by the pair-developer diminishing, in this way, the probability of introducing errors during the implementation of the simulator. In addition, we covered all the functionalities implemented with unit and acceptance tests, enabling an automatic and continuous verification of the correctness of the system.

5 Results

Let's assume for the moment that TDD helps teams productively build loosely coupled, highly cohesive systems with low defect rate and low cost maintenance profiles.[. . .] How could such a thing happen? Part of the effect certainly comes from reducing defects. The sooner you find and fix a defect, the cheaper it is. [. . .] Do I have scientific proof? No. No studies have categorically demonstrated the difference between TDD and any of the many alternatives in quality, productivity, or fun. [. . .] Another advantage of TDD that may explain its effect is the way it shortens the feedback loop on design decisions. [Kent Beck [13]]

Starting from what Beck said, we have performed a number of simulations of our model, which has been calibrated on *Repo Margining System* (section 4), varying the usage level of the TDD practice. That project was performed using TDD at 100%. Our goal was to understand whether and how diminishing the adoption of this practice would have changed the outputs of the Repo Margining project, taken as a prototype of a small highly dynamic development project.

We will focus on output variables related to the effort spent on development and the final quality of the released project, in terms of total working days and defect density, with the same number of released functionalities (User Stories). We make the following working hypotheses:

Hypothesis A: The residual defect density of the project using the TDD practice is different from that obtained without TDD.
Hypothesis B: The total working days needed to complete the project using TDD is different from those without TDD.
Hypothesis C: The number of the released User Stories using TDD is equal from that without TDD.

Looking at the results reported in table 2 (2nd and 3rd columns) we can observe that there is a certain difference between the results obtained not-using or using TDD. In particular, using TDD at 100%, the completion date increases of 21%, the residual defect density decreases of 20%, while the number of released USs is quite the same. These figures are quite in agreement with those reported in [14] obtained from structured experiments conducted with professional programmers.

Table 2. Comparison between simulation results averaged on 200 runs obtained varying the usage level of Test Driven Development (TDD). Standard deviations are reported in parenthesis

Output variable	TDD = 0% (PP = 0%)	TDD = 0% (PP = 100%)	TDD = 100% (PP = 100%)
Total working days	45,4 (22,0)	49,7 (17,6)	60,1 (23,5)
Defecs/KDSI	29,1 (6,5)	25,2 (6,7)	20,1 (4,3)
Released Stories	28,5 (6,6)	27,9 (6,6)	28,5 (7,0)

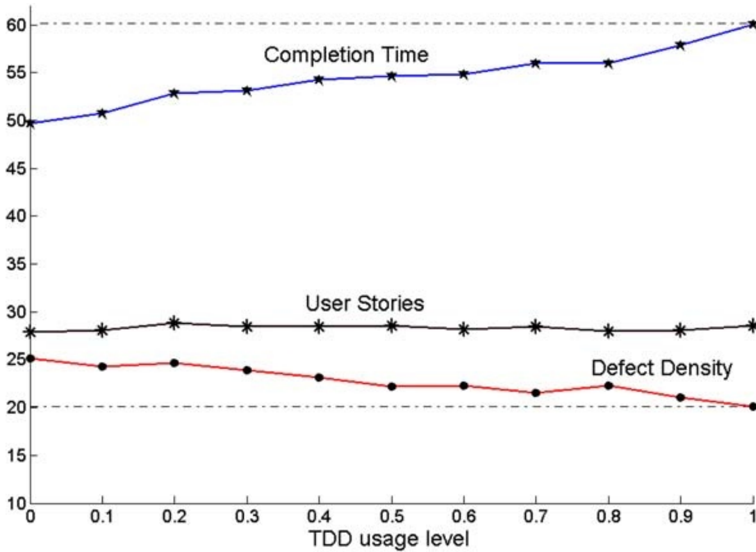


Fig. 2. Results obtained varying the TDD usage level. Average values, over 200 simulation runs, of Completion Time [days], User Stories and Defect Density [defects/KDSI]

Also, we have performed a two-sided t-test and we have found that there is statistical significance difference ($\alpha = 0,05$) between the two samples in terms of Total Days and Defect Density, while there is no difference for the number of released USs. Therefore, we can assert that the three starting hypotheses (A,B and C) have been confirmed by our experiment.

In figure 2 we have plotted the average of these output results gradually changing the usage level of TDD from 0 to 1. It can be easily seen how the total effort required increases with the TDD level while the defect density decreases.

Moreover, we have performed another experiment varying the usage level of Pair Programming. Looking at the first and third columns of table 2, it can be seen again a difference between the average values of working days, which increases of 32% using both practices at a maximum level. Again, the defect density decreases of 31%, with the same number of USs. These results have shown a statistical significance ($\alpha = 0,05$) after having performed a two-sided t-test.

6 Conclusions and Future Work

We have developed a simulation model of XP process in order to evaluate the effectiveness of this methodology. The input parameters are calibrated using a real XP project.

We have observed how the outputs of the simulated project vary with the usage level of TDD and Pair Programming. We have found that increasing the usage of such practices the defect density of the project significantly decreases.

On the other hand, the results have shown an increase on the number of days needed to implement the same functionalities.

Let us note that our model is not a complete representation of the intrinsic complexity of these practices and of the development process itself. We have based our model on what has been empirically found up to now, but many other issues have to be better investigated.

We are planning to improve the current simulator modeling other practices and activities of the software development process, with emphasis on XP. Another important stage of our research will be the validation of the model using other real projects and experiments.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
2. Cockburn, A., Williams, L.: The costs and benefits of pair programming. In: Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000), Cagliari, Sardinia, Italy (2000)
3. Kellner, M.I., Madachy, R.J., Raffo, D.M.: Software process simulation modeling: Why? What? How? The Journal of Systems and Software **46** (1999) 91–105
4. Cao, L.: A modeling dynamics of agile software development. In: Companion of 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press (2004) 46–47
5. Misic, V.B., Gevaert, H., Rennie, M.: Extreme dynamics: Modeling the extreme programming software development process. In: Proceedings of ProSim04 workshop on Software Process Simulation and Modeling. (2004) 237–242
6. Kuppuswami, S., Vivekanandan, K., Rodrigues, P.: A system dynamics simulation model to find the effects of xp on cost of change curve. In Marchesi, M., Succi, G., eds.: XP2003, Conference proceedings, Springer (2003) 54–62
7. Kuppuswami, S., Vivekanandan, K., Ramaswamy, P., Rodrigues, P.: The effects of individual xp practices on software development effort. SIGSOFT Softw. Eng. Notes **28** (2003) 6–6
8. Vivekanandan, K.: The Effects of Extreme Programming on Productivity, Cost of Change and Learning Efficiency. PhD thesis, Doctor of Philosophy in Computer Science and Engineering (2004)
9. Sanders, D.: Student perceptions of the suitability of extreme and pair programming. In: Proceedings of XP Universe Conference, Raleigh, NC (2001)
10. Turnu, I., Melis, M., Cau, A., Marchesi, M., Setzu, A.: Introducing TDD on a free-libre open source software project: a simulation experiment. In: Proceedings of Qute Swap workshop on QUantitative TEchniques for SoftWare Agile Processes. (2004)
11. KlondikeTeam: Tracking – A Working Experience. Published on: http://www.communications.xplabs.com/paper2001-2.html (1900)
12. Bossi, P.: Extreme programming applied: a case in the private banking domain. In: Proceedings of OOP2003. (2003)
13. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2003)
14. George, B., Williams, L.: An initial investigation of test driven development in industry. In: Proceedings of the 2003 ACM symposium on Applied computing, ACM Press (2003) 1135–1139

Agile Security Using an Incremental Security Architecture

Howard Chivers, Richard F. Paige, and Xiaocheng Ge

Department of Computer Science, University of York,
York, YO10 5DD, UK
Fax: +44 1904 432767
hrchivers@iee.org,{paige,xchge}@cs.york.ac.uk

Abstract. The effective provision of security in an agile development requires a new approach: traditional security practices are bound to equally traditional development methods. However, there are concerns that security is difficult to build incrementally, and can prove prohibitively expensive to refactor. This paper describes how to grow security, organically, within an agile project, by using an incremental security architecture which evolves with the code. The architecture provides an essential bridge between system-wide security properties and implementation mechanisms, a focus for understanding security in the project, and a trigger for security refactoring. The paper also describes criteria that allow implementers to recognize when refactoring is needed, and a concrete example that contrasts incremental and 'top-down' architectures.

1 Introduction

Security is an important part of system development; much of the web and all of e-commerce would not be successful without security engineering, and the more significant the application the more likely that security will be an important issue for customers. Agile development naturally matches stakeholders' needs for incremental delivery, and is therefore becoming the method of choice; however, little has been done to understand how security can be fully integrated in an incremental development.

Several researchers have contrasted Agile or XP developments with traditional security engineering processes [1, 2], and while these are generally favourable to XP they also highlight the gap between the documentation requirements of traditional engineering and the lightweight approach of agile methods. Unfortunately the gap is wider than merely aligning documentation practice: traditional security engineering is built on traditional design methods, which are qualitatively and quantitatively different [3]; for example, the documentation requirements of classical security engineering provide a firewall of independence between development and review, but XP has a completely different approach to quality assurance. What is needed is the development of new, agile, security practices.

The notion of 'good enough security' has been suggested by Beznosov [4] who reviews XP practices from the security perspective. Of course, the traditional security world practices 'good enough security', simply because security is not an absolute property or function: it must be cost-justified in terms of the system, its assets, stakeholders concerns, and the threat environment. Typically, risk based methods are used to judge the value of security in a system [5, 6] and determine what security features

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 57–65, 2005.
© Springer-Verlag Berlin Heidelberg 2005

are justified. Project Security has a number of facets, including how to establish what features are necessary, when they should be included in an iteration, and how to manage the development process. This paper deals with only the last problem: the development process.

It is generally claimed that iterative security is difficult: the system-wide nature of security properties mean that security may be prohibitively expensive to retrofit [7, 8] and refactoring is essentially a bottom-up process that may break system properties [2]; even Fowler comments that security may be hard to refactor (see Design Changes that are Difficult to Refactor at [9]). This is a bleak perspective for agile security, and highlights two critical questions:

- How can developers satisfy themselves that security mechanisms, which may be scattered through the system, provide useful system level security features?
- What factors have to be considered in an iteratively developing system, to ensure that new security features are not prohibitively expensive?

This paper proposes that these questions can be resolved by maintaining an iterative security architecture. Unlike conventional security architectures, an iterative architecture remains true to agile principles by including only the essential features needed for the current system iteration - it does not try to predict future requirements - but it does deal with both these questions directly: it provides a link between local functions and system properties, and it provides a basis for the ongoing review of the system from a security perspective.

The rest of this paper is divided into two main sections. Section 2 provides a criterion for architectural security: what constitutes an iterative architecture, what properties it should uphold, and how it fits into an agile development process. The second half of the paper gives a concrete example, drawn from the practical work which motivated this approach. This begins by describing a classical top-down security architecture; it then summarises two early iterations of an experimental system and their security. Given this background, the third iteration is an important test case. We outline possible security requirements for this iteration and show how they are compatible with the iterative approach, but that one of these would be infeasible had the classical architecture been implemented at the outset.

2 Iterative Security Architectures

The word *architecture* is often avoided in agile development, because of its traditional association with top-down design; however, it is not completely discounted as a concept. Kent Beck relates a system architecture to the idea of a metaphor, but stresses that it has to be the "*simplest thing that could possibly work*" [10]. Practical XP projects have also found that it is useful to record an architecture as the system develops: "*we lacked a clear overall vision … Eventually we decided to put up posters that showed sketches of the architecture" [11].*

Our concept of an iterative security architecture is exactly in conformance with these views:

> *An iterative security architecture is one that develops with the system, and includes only features that are necessary for the current iteration or delivery. The architecture is a working vision for developers, which encapsulates important security features of the existing design.*

We deliberately avoid the question of documentation for independent security assurance. Our aim is to build fit-for-purpose security within an agile development framework; experience will later allow us to speculate how we can best distinguish successful from unsuccessful security implementations.

However, we do need to make judgements about systems as they are developed. Functional developers refactor code because its structure is not fit for purpose: the existing code base inhibits the addition of new features, or is developing in a way that is structurally undesirable. Fowler describes *Bad Smells* [9], which allow developers to recognize situations that demand refactoring. If security is to be developed iteratively, then it is similarly necessary to have security criteria that allow the development team to distinguish good from bad practice.

2.1 Criteria for Good Security Architectures

Shore's work on Continuous Design [12] includes the addition of features that he describes as pervasive (i.e. not local): fine grain access control, transactional processing and internationalisation. The access security was added to a conventional project - it was tedious and extensive but not difficult; the other features were added in an XP project and required only limited changes to the code, because effective refactoring had already resulted in a target system with low functional duplication.

In Shore's work the features already present in the system included wrappers or interface functions (e.g. for HTML templates and persistence). The property that makes these structures effective is not just the avoidance of functional duplication, but reduced *coupling* between system functions: these design patterns reduce the number and type of function calls between modules. Low coupling is well established as a design principle, and we suggest that it is of critical importance to allow the subsequent introduction of security features.

This is consistent with security design principles that have been understood for some time; for example, Viega and McGraw [13] list ten principles, three of which are particularly relevant to security architectures:

- Be Reluctant To Trust.
- Execute all parts of the system with the least privilege possible.
- Keep it Simple.

To trust is to place reliance on something; any service constrained by a security control is relied on to support the security of the system, so the parts of the system that are security critical should be minimized. Privilege is the degree of access, or authority, that any process has; protection of the integrity of the system itself is the most significant privilege issue: a fundamental part of any security architecture is to identify how the system is administered, and minimize the number of users, or system functions, that can exercise that authority. The 'keep it simple' principle is a reminder that security guarantees and functional complexity are not easily combined; complex applications should not be security critical, and interfaces to security functions should be as simple as possible (coupling, again).

The security architecture must allow the project team to review the current implementation against these principles. It must also bridge the gap between local and system level properties; again we can draw advice from established security practice by proposing that the architecture should:

- Partition a system, identifying which parts are security-sensitive, and in what way.
- Show how security components combine to provide useful system level security.
- Communicate the structural logic of security; for example, ensuring that team members do not inadvertently build functionality that bypasses security.

Partitioning a system separates security relevant parts from those that are security-neutral. In practice this often means that an effective design places security in the infrastructure, which then constrains the behaviour of security-neutral applications.

2.2 How Should an Iterative Security Architecture Be Used?

In an agile project, building a security architecture – i.e. summarizing which components contribute to security, and how – supports these design principles by provoking a discussion, which may highlight the need for refactoring. This is the primary principle that we seek to establish: in each iteration the implementers must have space and reason to consider security; if not, the 'bad smells' simply accumulate to a level that is too difficult to refactor away when the system is presented with a demanding requirement. Security can only be developed iteratively if the development team continuously monitors its design.

In summary:

- An agile security architecture should be produced as the code is produced: it is a security view of the current system, and should not anticipate future iterations.
- The architecture should be as simple as possible, and be documented in such a way that it communicates the security intent to the whole development team (e.g. posters, simple UML diagrams).
- As the architecture is produced it should be reviewed against the criteria outlined in section 2.1. As with functional code, refactoring may be necessary to re-establish a system that satisfies the architectural criteria.

Many security architectures come ready-made, for example those of operating systems and web-servers; often these are well designed and implemented, and form a good basis for a security infrastructure. Without an explicit security architecture, however, as new security requirements are added the danger is that they will simply be adsorbed into application code. The use of ready-made security infrastructure is commendable, but it should not be used as a reason to defer making the system security architecture explicit.

The next section places these recommendations in context by providing a concrete example of a conventional architecture, then contrasting that with its incremental counterpart.

3 Architectures in Practice

This section provides practical examples of architectures, to illustrate the effectiveness of iterative development. There are many aspects to security architectures, including authentication, authorization, intrusion detection, audit, and communications security; depending on the system these many need to be combined into a single architecture, or presented as a small number of separate views. Because of space limita-

tions only authorization is discussed, but this has sufficient complexity to highlight the main issues[1].

The scenario used in our experimental development is an estate agent's business system; the first two iterations, and their security, have already been described [14], so this account does not detail the system or its design in detail.

The value of an iterative architecture is best demonstrated by comparing it with the established approach. We therefore start by outlining a typical, well-designed, top-down security architecture. This is followed by a summary of the much simpler architecture that developed in the first two iterations of our experimental system. The proposed third iteration adds a straightforward new security requirement, but this is sufficient to demonstrate the flexibility of the iterative approach, and the danger of committing too early to a security design.

3.1 A Top-Down Architecture

This architecture shows how applications are invoked, and how they are able to request fine-grain authorization decisions. A top-down design of this sort would typically be derived from representative generic scenarios, such as remote access to a business service, collaboration between businesses, or providing services to groups of similar users (e.g. at a remote institution). The result is a flexible architecture that can be divided into three tiers: the application container and its communications subsystem, the messaging layer and the application.

Web based authorization systems can be divided into coarse and fine controls. Coarse controls operate at the level of the network, transport protocol and container, while finer grain controls operate in message handler chains and in the application. The message layer is able to manage clients that invoke services via intermediate systems, and can support complex policies where a client presents an arbitrary claim (e.g. sufficient budget, organization membership) to be allowed access.

These authorization decisions can either be required by an infrastructure policy, in which case they are checked before the application is invoked, or be requested from the application via an authorization interface. Figure 1 shows a typical architecture.

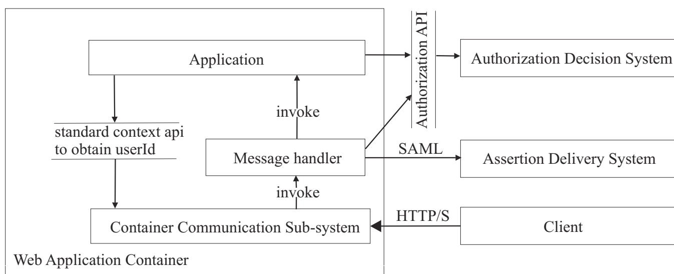


Fig. 1. A Top-Down Authorization Architecture

[1] One problem of restricting this discussion to a partial architecture is that we will not be able to show how the architecture contributes to the system viewpoint. We are essentially working at the mechanism level; the system viewpoint will be described elsewhere.

Space prevents an extensive account of this architecture, but the security principles in section 2 have been followed to define a small number of well-defined components, minimize coupling by defining suitable interfaces, and separate applications from security mechanisms. The architecture as it stands allows the independent development of its components, and the substitution of major sub-systems without change to applications (e.g. different user identity systems can be wrapped to provide SAML assertions).

This architecture also meets the criteria for communication – it provides an overall guide to security-relevant parts and their function. It would be a 'creative' implementer that, given this view, felt able to build a separate communications stack within an application without considering the security implications. One of the benefits of pair programming is that extreme individualism of this sort is unlikely.

This is modern, well designed, and meets the security criteria, so what can go wrong? This is a complex infrastructure: it requires considerable understanding and security sophistication within the implementation team, and if an attempt is made to implement it before the system, then there is a high start-up cost. One option is to design top-down and implement incrementally, but this still involves significant design cost and the learning process may result in a cultural commitment to a design which, as we show later, may not be useful.

3.2 Two Simple Iterations

The experimental system [14] provides services to house buyers; initially it allows a single estate agent to manage details of houses, but during the lifetime of the system the services offered increase to include searching multiple estate agents' properties, personalising the customer interface, and finally supporting the house-purchase process. We describe only external-delivery iterations here; the issue of how to schedule security in internal project iterations will be reported separately.

Because the growth of this system involves new users, organizations and collaborations, as well as services, security is a natural and essential part of the user requirement, and the requirements for security change and grow as the system develops.

First Iteration: Self Contained System. The first delivery is a database to support an Estate Agent; it provides local managers with the facility to store, manage, search and display details of houses for sale. The primary security requirement is to maintain the integrity of the hardware, software and database; this is met by physical access controls, and by backup and restore procedures.

This system does not therefore require an authorization system. The delivery still needs a security architecture (for backup and recovery) but the team must also consider authorization issues: have any security features been built into this application, or appear in the user stories? If the application code makes decisions based on the identity of the user, their role, or other access-based decisions the team would need to recognize that they have started to introduce authorization features.

Second Iteration: Extending the Service to a Second Organization. The second iteration of this system introduces a new organisation, the Broker, which provides customers with a search facility.

The estate agent therefore needs access control, but a simple regime that can be supported by web-server infrastructure is sufficient[2]: three roles are defined (manager, broker, guest). The total number of users is small enough for user accounts to be maintained in the Estate Agent's web-server.

The team has made good use of the built-in security architecture of the web-server. However, an explicit architecture is still useful: the team needs to collectively agree that this description is adequate (e.g. no stories need finer grain control), and be aware of their obligations (e.g. if necessary sub-divide application pages so that access can be on the basis of page URL, rather than within the application). Even with this simple system, therefore, there is a need for team-wide agreement and co-ordination. The architecture is now a subset of the top-down architecture shown in figure 1: the container, the communication sub-system and the context API.

3.3 The Third Iteration

The customer requirement for the third iteration is straightforward: the Estate Agent wishes to introduce a personalized service to allow users to register their details and be advised when suitable properties become available.

The application functions now need user identities, so there is an inevitable link between the security infrastructure and the application. A good security design would still minimize the coupling: the user authentication and access control is still located in the infrastructure, but the security context includes a user identity. Unfortunately this is not enough to determine the architecture; there are choices to be made about how users are authenticated and how user accounts are managed. We give just two scenarios of many:

The first scenario assumes that users are sophisticated e-commerce customers, who have registered with a user identity service. The required architecture is the same as Figure 1: user authentication is carried out by an external service, which provides SAML assertions on request to a local policy-decision system.

The second scenario anticipates casual users who are prepared to register and use password-based authentication. In this case the architecture is very different. Authentication and authorization functions are integrated with the container communications handler, using either a standard container option or, more likely, a commercial or bespoke adaptor that is linked to a user account database.

How does this compare with the top-down architecture? The first scenario is compatible with the top-down approach, and the second is not. In the first case the effort put into the early design may be recuperated by the third iteration; in the second the top-down design work has wasted time and skill. In both cases it is straightforward to add the new security features from the baseline of the iterative architecture of the second iteration.

There is, of course, a worse case. If instead of the architecture used for iteration two, the implementers had built access decisions into each (active) page, then at best every application page would need to be updated to accommodate the new security requirement. At this stage it may be unnecessarily expensive, but perhaps not prohibi-

---

[2] This is a simplification. The system also needs database access controls; this is a significant issue when the whole system perspective is considered, but adds little to this discussion.

tively so. However, if the implementers were to continue with this approach and place individual user authorization information in the application database, the coupling between security and functionality would soon be difficult to modify. For example, moving between the two scenarios above would be particularly difficult. Of course this design would violate the criteria of section 2.1, as well as other well known design principles, such as 'single choice'[3]; the value of an iterative architecture is that it would make the development team aware of the problem.

In summary, this worked example shows that there are two very bad choices of method: up-front architecture with implementation, or no architecture. Even a good top-down architecture may not be appropriate when the future requirements become known, so early implementation is at best expensive and at worst limits future design choices. No architecture at all runs the biggest risk: that the security becomes so entangled in the application that its quality is uncertain and the system is prohibitively expensive to refactor. A top-down architecture with incremental implementation may be a waste of design time, but will be disastrous only if it results in a cultural attachment to the wrong solution. On the other hand, the incremental architecture is clearly effective; it provides flexibility, quality security, and minimises up-front cost.

4 Conclusions

A key issue in agile system development is the prospect for incremental security, and the possibility that refactoring security is inherently difficult.

In order to integrate security in an agile development environment it is necessary for the team as a whole to have an overview of the security approach, and a trigger that causes them to consider and refactor the security design. This places security in the mainstream of agile development practices, and has strong parallels with the way that quality functional designs are produced.

We have shown that this can be achieved by using iterative security architectures. Such an architecture includes only what is necessary for the present delivery, but provides the necessary communication mechanisms and design triggers to ensure that the security solution remains well structured.

This paper outlines criteria for an incremental security architecture and describes how it can be used in system implementation. A concrete example shows that iterative architectures are superior to top-down architectures: they defer design costs until features are needed, and reduce the danger of cultural commitment to a faulty design.

References

1. Wäyrynen, J., M. Bodén, and G. Boström. Security Engineering and eXtreme Programming: An Impossible Marriage? in XP/Agile Universe 2004: 4th Conference on Extreme Programming and Agile Methods. 2004, Springer-Verlag,Lecture Notes in Computer Science. p. 117
2. Beznosov, K. and P. Kruchten. Towards Agile Security Assurance. in The New Security Paradigms Workshop. 2004

[3] Given a multiple choice (such as selection of user) the complete list of choices should only be in one place in a system.

3. Endiktsson, O., D. Dalcher, and H. Thorbergsson. Choosing a Development Life Cycle: Comparing Project and Product Measures. in Génie Logiciel & Ingénierie de Systèmes et leurs Applications (ICSSEA'04). 2004
4. Beznosov, K. Extreme Security Engineering: On Employing XP Practices to Achieve "Good Enough Security" without Defining It. in The first ACM Workshop on business Driven Security Engineering (BizSec). 2003, ACM Press
5. Risk Management Guide for Information Technology Systems. 2002, National Institute of Standards and Technology (NIST)SP 800-30. http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf
6. Chivers, H. and M. Fletcher. Adapting Security Risk Analysis to Service-Based Systems. in Grid Security Practice and Experience Workshop. 2004, University of York, Department of Computer Science,Technical Report YCS 380
7. Amey, P. and R. Chapman. Static Verification and Extreme Programming. in The 2003 annual ACM SIGAda international conference on Ada: the engineering of correct and reliable software for real-time & distributed systems using ada and related technologies. 2003, ACM Press. p. 4-9
8. Kevin Soo Hoo, A.W. Sudbury, and A.R. Jaquith, Tangible ROI through Secure Software Engineering. Secure Business Quarterly, 2001. **1**(2).
9. Fowler, M., Refactoring: Improving the Design of Existing Code. The Addison-Wesley Object Technology Series. 1999: Addison Wesley Longman.
10. Beck, K., Extreme Programming Explained. 2000: Addison Wesley Longman.
11. Murru, O., R. Deias, and G. Mugheddu, Assessing XP at a European Internet Company. IEEE Software, 2003. **20**(3): p. 37-43.
12. Shore, J., Continuous Design. IEEE Software, 2004. **21**(1): p. 20-22.
13. Viega, J. and G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way. Professional Computing. 2001: Addison-Wesley.
14. Paige, R.F., J. Cakic, X. Ge, and H. Chivers. Towards Agile Re-Engineering of Dependable Grid Applications. in Génie Logiciel & Ingénierie de Systèmes et leurs Applications (ICS-SEA'04). 2004

Quantifying Requirements Risk

Fred Tingey

10 Harewood Avenue, London, NW1 6AA
fred.tingey@bnpparibas.com

Abstract. It is possible to apply Information Theory to the Software Development process – an approach I have dubbed 'Iterative Theory'. Focusing on the user requirements Iterative Theory is introduced and then used to quantify how the choice of development methodology affects the 'value at risk' on a software project. The central theme is that end-user requirements cannot be described exactly resulting in an inherent uncertainty in the correctness of any specification. This uncertainty can only be removed by receiving feedback on working software. Iterative Theory, the application of Information Theory to the software development process, is certainly an area requiring further study.

1 Introduction

This paper has two purposes; firstly to propose a new area of research applying Information Theory to software development - called Iterative Theory, and secondly as an example to apply Iterative Theory to quantify the requirements risk associated with a development project.

To limit the scope we focus purely on quantifying the requirements risk associated with software development.

Before we begin I will describe in outline the main elements of the software development process.

We normally start by going out and gathering what are called 'User Requirements'. These represent statements and descriptions about what is wanted by the Users. They may be further analysed and refined into a specification about what the software system needs to be able to do. Iterative Theory requires that specific acceptance tests are prepared as part of the analysis for each Requirement. We can imagine many other steps to refine the analysis but ultimately we end up at a point where some software will be developed.

Software is built (or engineered or developed). There are many ways to do this; e.g. in steps or as a big bang, but the ultimate result is a piece of working software. Any tests that have been identified as part of the analysis must be passed by the software before it can be called 'working'.

At some point, and this is the crucial step, the working software is shown to the End Users and they then have the option to test it. Of course the software would already pass any tests that had been previously identified. The fact is that the initial analysis of the requirements will never be 100% correct except in the most trivial of cases. There will be tests or even whole requirements that are either missing or wrong that will only be discovered at this late stage.

The next step in the process is to put the software into 'production'. This may also include elements of user training, migration from an existing system to the new one, phasing the roll-out, interfacing with other systems etc.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 66–73, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Agile approaches make explicit the fact that the above steps are carried out many times whereas Traditional approaches would imply they are done once sequentially.

The final element in the process is called software maintenance. This is where issues with the production software or new requirements that arise are dealt with as changes requiring a subsequent re-run of the process above. This can be fairly lightweight for small bugs or enhancements but can expand into a major effort if a significant new version of the software is required incorporating large changes.

1.1 The Abstract Version

Focusing on the key points for Iterative Theory we get:

1. Talk to the Users to gather and prioritise User Requirements
2. Analyse the Requirements and produce Acceptance Tests
3. Develop Working Software that passes the Acceptance Tests
4. Verify that the Working Software is Fit For Purpose from the Users point of view
5. Release and Maintain the Working Software

The steps above form the basis of the model presented in this paper. A theory is developed to quantify the requirement risk.

Minimising the requirement risk turns out to mean that any collection of requirements should be developed iteratively i.e. we perform step 4. – User Verification – at multiple points in the process rather than just once for all the requirements at the end of the process.

2 The Iterative Theory of Software Development

We will go through the software development process introducing the concepts of Iterative Theory relevant to requirements risk as we go. We will build up a simple mathematical model and see what insights can be gained about adopting different software development strategies.

2.1 Talk to the Users, Gather and Prioritise User Requirements

Notice that there is a subtle difference between what the Users really need and what they actually ask for. What they ask for, the requirements, can be analysed and written down. What they really need is 'unobservable' in an absolute sense – there is always the risk of an error between what is written down and what will actually work. The accuracy of the requirements can only be verified once the software is developed and there is the chance for feedback from the Users regarding the implementation.

Iteration Theory. We will represent the user requirements as a set $\mathbf{R}=\{R_i\}$ of binary random variables where '1' means that the User is satisfied with the implementation of the requirement and it meets their need and '0' means the need is not satisfied as there is something wrong with the implementation of the requirement – not that the software has bugs but rather that what they asked for does not work in practice.

We will associate a probability distribution $p(R_i)$ with R_i:

$$p(R_i=1)=\mathit{Probability\ of\ R_i\ Passing\ User\ Acceptance\ Testing}$$

$$p(R_i=0)=\mathit{Probability\ of\ R_i\ Failing\ User\ Acceptance\ Testing}$$

Typically these requirements would be prioritised based on business value provided, estimated cost, risk of implementation etc. but in any case at some point the 'scope' of the project would be defined as a subset containing 'N' of the possible requirements which we can call $\mathbf{R}^{\mathbf{N}} = \{R_1, R_2, \ldots R_N\}$.

This set $\mathbf{R}^{\mathbf{N}}$ of requirements could form the 'minimal marketable feature set' for the development project i.e. The minimal amount of functionality required for the project to 'go live' or be regarded as useful[1].

This set $\mathbf{R}^{\mathbf{N}}$ forms an ensemble of not necessarily independent binary random variables. The binary variables are related by a joint probability distribution $p(\mathbf{R}^{\mathbf{N}}) = p(R_1, R_2, \ldots R_N)$. It is not easy to know or work out what this distribution is from the given requirements.

> *However this turns out not to matter too much as the results we will derive will be applicable whatever the distribution happens to be.*

2.2 Analyse the Requirements and Produce Acceptance Tests

So what does it mean to implement a requirement as working software? In the Iterative Model we interpret this statement as meaning: "define a set of pass / fail tests for each user requirement and call the software working when all of the tests pass simultaneously".

The job of analysis is to generate Acceptance Tests for User Requirements which in turn are an approximation to the 'real needs'. Notice the definite separation of interests here; Analysis is concerned with defining business relevant pass / fail tests; writing software is concerned with engineering a system to pass the tests in the most cost effective way possible.

> *By applying Information Theory we can prove that whatever the details of the engineering process and whatever the tests defined it is always less risky to develop iteratively.*

It is possible to define each requirement as a set of tests and thus assign an overall distinct pass / fail or binary value to it. This is important as it allows us to define 'Working Software' i.e. software that we believe contains no known errors with respect to the requirements as stated.

2.3 Develop Working Software that Passes All of the Acceptance Tests

So we have a set of tests for each requirement R_i such that if every test passes we can say we have working software i.e. We can say that the requirements are implemented – albeit with a possibility that they will fail end user acceptance testing. The only way to know for sure is to seek feedback on the working software.

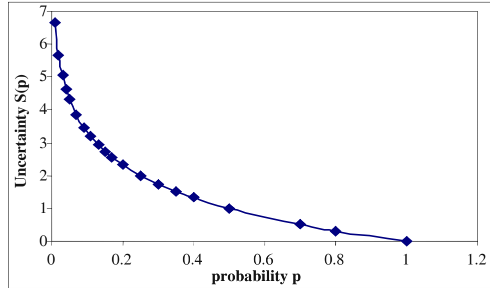
Iteration Theory. We start from the joint probability distribution $p(\mathbf{R}^{\mathbf{N}}) = p(R_1, R_2, \ldots R_N)$ of the requirements. We can subject this to analysis in terms of information theory by defining the uncertainty associated with a particular outcome as the Shannon information content [3] of that outcome:

---

[1] For the moment we will forbid going live with a subset of requirements; if we could then that would be the minimal marketable feature set.

$$S(\boldsymbol{R}^{N}) = -\log p(\boldsymbol{R}^{N}) \tag{1}$$

This uncertainty is a measure of the Risk associated with the project. Clearly if p($\mathbf{R}^{\mathbf{N}}$) is close to one then the risk is close to zero whereas if it is close to zero then the risk can be very large.



We are mostly interested in the particular outcome where the requirements are implemented i.e. $R_i = 1$ and this has an associated uncertainty of:

$$S(R_1 = 1, R_2 = 1, \ldots R_N = 1) = -\log p(R_1 = 1, R_2 = 1, \ldots R_N = 1) \tag{2}$$

One way to reduce the risk is to do more analysis. This is an attempt to increase the joint probability p($\mathbf{R}^{\mathbf{N}}$) to be closer to one[2] as expressed in terms of the acceptance tests for the working software. One would hope that the 'better specified' the requirements are the more likely they are to be accepted; note very carefully that 'better specified' has a technical meaning here in that these additional tests must bring *extra information* about the requirement.

Another way to reduce risk is to choose requirements that are most likely to be accepted in the initial set of N that were chosen. However this initial choice is often driven by the business value to be derived from implementing them so this is not always an option. Further it is probable[3] that there is a correlation between the requirements that add the most value and those that are harder to specify.

Development Approaches. Now we are ready to consider the impact of alternative development strategies. S($\mathbf{R}^{\mathbf{N}}$) represents the uncertainty or requirements risk associated with the functionality of the development project.

We could develop working software for all of the requirements before seeking feedback. This approach where all of the requirements are implemented as a set in a single deliverable is called **Waterfall Development**[4].

If we develop working software fully implementing one requirement at a time seeking user feedback after each one and gradually building up the full system then a different picture emerges. This approach is called **Iterative Development**[5].

Iteration Theory. Assuming that we are in possession of a fixed set of requirements $\mathbf{R}^{\mathbf{N}}$ and that we are going to develop them all as a single project we can now compare the risks of the two approaches from an information theory point of view.

---

2 We can define a 'risk free' cost associated with the software as the build cost when the uncertainty is zero i.e. that the requirements will certainly be accepted by the end users.

3 Consider; if it was easy to do and adds a lot of value it would have been done already!

4 This nomenclature dates back to an original 1970 paper by Dr Winston W Royce [2]; he in fact proposed a two iteration cycle but this seems to have been forgotten over the years.

5 The origins of Iterative Development are long and distinguished; despite the dominance of Waterfall based formal methodologies in academia and big business.

We can consider an 'iterative style' waterfall approach – developed iteratively but not show to the end users until it is 'finished' – as having the same risk as a true waterfall approach.

For the iterative approach we complete one requirement at a time and seek feedback in order to get it accepted before moving on to the next. In this case information is being added as we go and this extra information reduces the risks in the overall project. Let us quantify this.

The overall uncertainty in the project for a particular outcome is given by:

$$S(\boldsymbol{R}^N) = S(R_1) + S(R_2/R_1) + S(R_3/R_1, R_2) ... + S(R_N/R_1, R_2, ..R_{N-1}) \quad (3)$$

or in words:

Total uncertainty = uncertainty in R_1 plus that in R_2 given we know R_1 etc.

In Iterative development we can use the information gained as each requirement is verified to reduce the uncertainty remaining in subsequent requirements.

Imagine that we have completed the i'th iteration then we find:

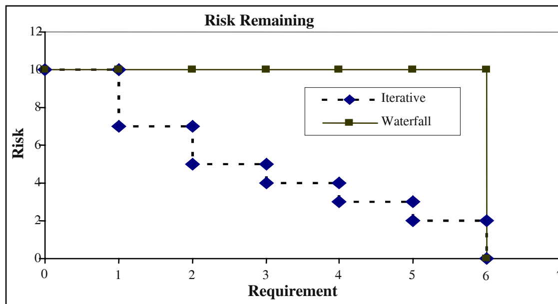
$$S(R_{i+1}, R_{i+2}, ..., R_N/R_1, R_2 ... R_i) = S(\boldsymbol{R}^N) - S(R_1, R_2 ... R_i) \quad (4)$$

or in words:

Risk Remaining = Risk we started with – Information Gained

For this to work we have to be able to show the users 'working software' and get feedback after each iteration. In addition we have to be able to guarantee that further iterations do not invalidate previous ones hence the need to have defined acceptance tests that can be repeated every iteration.

If we look at a graph of the uncertainty remaining as a function of the number of requirements implemented then we find a picture like:



In this case there are six requirements and as each is implemented the remaining risk is reduced. At each step R_i we get a reduction in risk of:

$$S(R_i/R_1, R_2, ... R_{i-1}) \quad (5)$$

For the Waterfall approach all the risk remains until the end.

Value at Risk. Consider the graph above and ask what it is telling us about costs and risks. If we assume the requirements are being developed sequentially[6] and are each of roughly the same size then the x axis represents a *time line* and as we move along that line development costs are increasing.

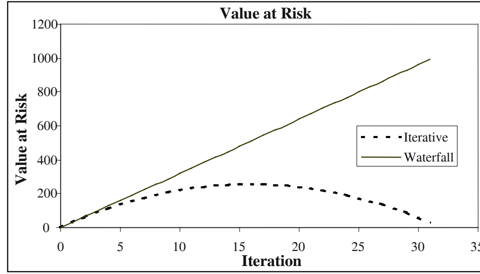
[6] We assume an 'iterative style' to our waterfall development (without the feedback of course!)

If we multiply the Costs so far by the Risk remaining we are defining a crude measure $V(\mathbf{R}^N)$ of the 'value at risk' for the project. The value at risk is best defined as a function of the number of iterations completed.

We will use $V(R_i)$ to represent the value at risk after the i'th iteration with 'C' to represent the cost per iteration. Then we find:

$$V(R_i) = i\,C(S(\boldsymbol{R}^N) - S(R_1, R_2 \ldots R_i)) \tag{6}$$

$$V(R_i) = i\,C\,S(R_{i+1}, R_{i+2}, \ldots, R_N / R_1, R_2 \ldots R_i) \tag{7}$$



As can be seen in the waterfall approach the value at risk increases up until the end of the project whereas for an iterative approach it peaks in the middle somewhere – the peak is lower than that of waterfall; with identical independent variables it is approximately ¼ and occurs halfway through the project.

Sub Projects. Another possibility is to split a big project up into sub projects in order to reduce the risk. If these projects are completed sequentially then this is the model where several requirements are completed per iteration.

Imagine splitting a system development up into independent components or sub projects. For simplicity let us assume that there are 'n' requirements per sub project so that there are N/n sub projects M_j all together. We also arrange it so that the first sub project has requirements 1 to n , the second n+1 to 2n etc. from $\mathbf{R}^N$. We can then write down the uncertainty of each sub project as:

$$S(M_j) = S(R_{(j-1)n+1}, R_{(j-1)n+2}, .. R_{jn}) \tag{8}$$

given the overall uncertainty as:

$$S(\boldsymbol{R}^N) = S(R_1, R_2 \ldots R_n) + S(R_{n+1}, R_{n+2}, \ldots, R_N / R_1, R_2 \ldots R_n) \tag{9}$$

we can pull out the N/n sub projects M_j by taking n requirements at a time to get:

$$S(\boldsymbol{R}^N) = S(M_1) + S(M_2 / M_1) + \ldots + S(M_{N/n} / M_1, \ldots, M_{N/n-1}) \tag{10}$$

We will now consider what happens if the projects are developed by separate teams (perhaps in parallel) so that feedback on the requirements for one are not available to the others. What this means for example is that when developing M_2 we are unable to take advantage of the feedback from M_1 . Taking a conditional uncertainty can only ever reduce the uncertainty:

$$0 \leq S(X/Y) \leq S(X) \tag{11}$$

Therefore:

$$S(\boldsymbol{R}^N) \leq S(M_1) + S(M_2) + \ldots + S(M_{N/n}) \tag{12}$$

i.e. the uncertainty or risk associated with a collection of sub projects is always greater than that of a single project of all the requirements; unless the requirements in each sub project are independent of those in any other sub project.

This may seem (and is) pretty obvious. The risk in developing a single system for instance to a) control a thermostat and b) do spell checking is the same as the risk in developing two separate systems one for each function.

The contrary is perhaps not so obvious; splitting a system up into separate independent projects when there clearly is mutual information between them will only ever increase the risks[7]. From a risk point of view:

> *It is far better to develop 'front to back' slices of functionality for each requirement than it is to split the requirement across projects.*

2.4 Verify that the Working Software Is Fit for Purpose from the Users Point of View

So far we have looked exclusively at the risks associated with a development project because of the inherent uncertainty in the requirements. We have concluded that it is not a good idea to split a project up into separate projects unless the mutual information between the requirements in different projects is small. We have also concluded that the 'value at risk' on a project is always lower for an iterative project than it is for a waterfall one under not unreasonable assumptions.

Given that the Iterative model requires regular user feedback that means we need 'working software' to be available after each iteration incorporating all the requirements so far. Further we must be able to show that there is no regression in the requirements developed so far. These lead naturally to the Agile practices of regular integration and automated acceptance testing.

2.5 Release and Maintain the Working Software

Assuming that we have completed the development of the working software and it has been accepted by the users and is deployed into production we are unlikely to be in the situation where all the work is finished.

Firstly we will recall that we initially chose a set of requirements to form the initial delivery maximising business value and forming the minimal marketable feature set.

Secondly additional requirements are bound to have arisen during the development of the working software that were not envisaged at the beginning of the project. Some of these requirements could even be needed, with hindsight, in the minimal marketable feature set.

Thirdly the world does not stand still. It may well be that some of the requirements already developed need to be changed or modified in order to reflect the current 'needs.

Ultimately there is an uncertainty related to time. The initial analysis and definition of the minimal marketable feature set is a snapshot of the real business needs[8]. Requirements that were once valuable may become no longer needed; new requirements

---

[7] Think Interfaces between systems perhaps?

[8] It is not unknown for a Waterfall project to fall into the trap of targeting a feature set where the rate of 'churn' of requirements is faster than the feature set can be developed.

can arise. The delivery of working software can stimulate or stumble across extra requirements that were not anticipated.

> *Altogether this inherent requirements uncertainty justifies an iterative approach in virtually all non-trivial situations.*

So far we have assumed that we could adopt an iterative like style to the Waterfall development; the only difference to Iterative being that we keep it back from the users until it is 'finished'. But if we do this we will miss out on any requirements that arise or change significantly as a result of the users seeing the working software - thus increasing risks.

3 Conclusion

Iterative Theory allows us to develop a theoretical understanding of the software development process. Further, it becomes possible to start quantifying various aspects of the process such as the Requirements Risk using Information Theoretic arguments. We can summarise what we have learnt so far:

Value At Risk: Iterative < Value at Risk: Waterfall
Risk of a project < Risk of splitting it into sub projects

These are important results and completely general. Given knowledge of the joint probability distribution Iterative Theory enables us to start quantifying exactly how much less risky various development approaches will be. Given the large number and large costs associated with many software projects this knowledge is very valuable.

One of the most striking features of Iteration Theory is how the theoretical conclusions tend to match up with what seems obvious to software development practitioners such as myself. In this paper we have focused on Requirements Risk and avoided the formal proofs and background material due to space limitations but there is plenty more to say on the subject.

My own ideas for further research suggest the following areas:

- Cost of Testing of software [forthcoming]
- Drilling down to Reducing Risk at the Acceptance / Unit test level
- Design of Tests for maximising gained Information / minimising risk
- Estimation of the joint and conditional probability distribution functions
- Seeking the best order of implementation to reduce risk as early as possible
- Quantifying the Design Risk between Iterative and Waterfall approaches.

An alternative approach is to consider the practices of Agile methods and see what the implications are from an Information Theory point of view.

Further developments of Iteration Theory have been mentioned in the text and many more can be imagined – I issue a call for more joint research by Information Theory and Information Technology professionals.

References

1. Mackay, David J. C.: Information Theory, Inference and Learning Algorithms. Cambridge University Press 2004.
2. Royce, Winston: Managing the Development of Large Software Systems. Proceedings of IEE Westcon, 1970.
3. Pierce, John R.: An Introduction to Information Theory. Dover Science Books, 1980.

Social Perspective of Software Development Methods: The Case of the Prisoner Dilemma and Extreme Programming

Orit Hazzan[1] and Yael Dubinsky[2]

[1] Department of Education in Technology & Science
Technion – Israel Institute of Technology
oritha@tx.technion.ac.il
[2] Department of Computer Science
Technion – Israel Institute of Technology
yael@cs.technion.ac.il

Abstract. One of the main dilemmas with which software development teams face is how to choose a software development method that suits the team as well as the organization. This article suggests a theory that may help in this process. Specifically, Extreme Programming (XP) is analyzed within the well known framework of the prisoner dilemma. We suggest that such an analysis may explain in what situations XP may fit for implementation and, when it is used, the way it may support software development processes.

Keywords: Agile software development methods, extreme programming, social theories, game-theory, the prisoner dilemma.

1 Introduction

Software development is a complicated task that is strongly based on the people carried it out. The task of deciding upon the appropriate software development method that suits the organization and the team should address the fitness of the method to the people involved (Dines, 2003). Usually methods are selected according to organization and team traditions and according to practical-professional-technical considerations like the programming languages and tools. This situation motivated us to check the analysis of software development methods from other perspectives, such as social and cognitive ones.

This article focuses on a social framework. Specifically, from the social perspective, we apply a game theory framework – the prisoner's dilemma – which is usually used for the analysis of cooperation and competition. Our arguments are illustrated by Extreme Programming (Beck, 2000). We view the perspective presented in this paper as part of our research about human aspects of software engineering in general, and our comprehensive research about cognitive and organizational aspects of XP, both in the industry and the academia, in particular (Hazzan and Dubinsky, 2003A, 2003B; Tomayko and Hazzan, 2004; Dubinsky and Hazzan, 2004).

2 A Game Theory Perspective: The Prisoner's Dilemma[1]

Game theory analyzes human behavior by using different theoretical fields such as mathematics, economics and other social and behavioral sciences. Game theory is

[1] This analysis is partially based on Tomayko and Hazzan, 2004.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 74–81, 2005.
© Springer-Verlag Berlin Heidelberg 2005

concerned with the ways in which individuals make decisions, where such decisions are mutually interdependent. The word "game" indicates the fact that game theory examines situations in which participants wish to maximize their profit by choosing particular courses of action, and in which each player's final profit depends on the courses of action chosen by the other players as well.

In this section, a well-known game theory framework is used, namely the prisoner's dilemma. This framework illustrates how the lack of trust leads people to compete with one another, even in situations in which they might gain more from cooperation. In the simplest form of the prisoner's dilemma game, each of two players can choose, at every turn, between cooperation and competition. The working assumption is that none of the players knows how the other player will behave and that the players are unable to communicate. Based on the choices of the two players, each player gains points according to the payoff matrix presented in Table 1, which describes the game from Player A's perspective. A similar table, describing the prisoners' dilemma from Player B's perspective, can easily be constructed by replacing the locations of the values 10 and (-10) in Table 1.

Table 1. The prisoner's dilemma from player A's perspective

	B cooperates	**B competes**
A cooperates	+ 5	-10
A competes	+10	-5

The values presented in Table 1 are illustrative only. They do, however, indicate the relative benefits gained from each choice. Specifically, it can be seen from Table 1 that when a player does not know how the other player will behave, it is advisable for him or her to compete, regardless of the opponent's behavior. In other words, if Player A does not know how Player B will behave, then in either case (whether Player B competes or cooperates), Player A will do better to compete. According to this analysis, both players will choose to compete. However, as can be seen, if both players choose the same behavior, they will benefit more if they both cooperate rather than if they both compete[2].

As it turns out, the prisoner's dilemma is manifested also in real life situations, in which people tend to compete instead of to cooperate, although they can benefit more from cooperation. The fact that people tend *not* to cooperate is explained by their concern that their cooperation will not be reciprocated, in which case they will lose even more. The dilemma itself stems from the fact that the partner's behavior (cooperation or competition) is an unknown factor. Since it is unknown, the individual does not trust her partner, nor does the partner trust her, and, as described in Table 1, both parties choose to compete.

In what follows, the prisoner's dilemma is used to show that competition among team members is the source of some of the problems that characterize software development processes. To this end, it should be noted first that, in software development environments, cooperation (and competition) can be expressed in different ways, such as, information sharing (or hiding), using (or ignoring) coding standards, clear and

[2] For more details about the Prisoner's Dilemma, see the GameTheory.net website at: http://www.gametheory.net/Dictionary/PrisonersDilemma.html.

simple (or complex and tricky) code writing, etc. It is reasonable to assume that such expressions of cooperation increase the project's chances of success, while such expressions of competition may add problems to the process of software development.

Since cooperation is so vital in software engineering processes, it seems that the quandary raised by the prisoner's dilemma is even stronger in software development environments. To illustrate this, let us examine the following scenario according to which a software team is promised that if it completes a project on time, a bonus will be distributed among the team members according to the individual contribution of each team member to the project. In order to simplify the story, we will assume that the team comprises of only two developers – A and B. Table 2 presents the payoff table for this case.

Table 2. The prisoner's dilemma in software teams

	B cooperates	**B competes**
A cooperates	The project is completed on time. A and B get the bonus. Their personal contribution is evaluated as equal and they share the bonus equally: **50% each.**	A's cooperation leads to the project's completion on time and the team gets the bonus. However, since A dedicated part of her time to understanding the complex code written by B, while B continued working on her development tasks, A's contribution to the project is evaluated as less than B's. As a result, **B gets 70% of the bonus and A gets only 30%**.
A competes	The analysis is similar to that presented in the cell 'A cooperates / B competes'. In this case, however, the allocation is reversed: **A gets 70% of the bonus and B gets 30%**.	Since both A and B exhibit competitive behavior, they do not complete the project on time, the project fails and they receive no bonus: **0% each**.

The significant difference between Table 2 and the original prisoner's dilemma table (Table 1) lies in the cell in which the two players compete. In the original table, this cell reflects an outcome that is better for both players than the situation in which one cooperates and the opponent competes. In software development situations (Table 2), the competition-competition situation is worst for both developers. This fact is explained by the vital need for cooperation in software development processes. It can be seen from Table 2, that in software development environments, partial cooperation (reflected in Table 2 by the cooperation of only one team member) is preferable to no cooperation at all.

Naturally, in many software development environments, team members are asked to cooperate. At the same time, however, they are unable to ensure that their cooperation will be reciprocated. In such cases, even if there is a desire to cooperate, there is no certainty that the cooperation will be reciprocated, and, as indicated by the prisoner's dilemma table (Table 1), each team member will prefer to compete. However, as indicated by Table 2, in software development situations, such behavior (expressed by the competition-competition cell) results in the worst result for *all* team members.

3 Application of the Prisoner Dilemma for the Analysis of XP

In a recent interview, Kent Beck[3] was asked when XP is not appropriate. Beck answered that "[i]t's more the social or business environment. If your organization punishes honest communications and you start to communicate honestly, you'll be destroyed." As can be seen, Kent talks in terms of punishment. In fact, Kent describes a situation in which a team member cooperates ("communicates honestly") while the environment competes ("punishes honest communications"). In terms of the prisoner's dilemma, this is the worst situation for the cooperator, and so, it is expected that no one will cooperate in such an environment. If the entire organization, however, communicates honestly, that is to say, cooperates, there is no need to worry about cooperating, since it is clear that the other members of the organization will cooperate as well. Consequently, the entire organization reaps the benefits of such cooperation.

In what follows we demonstrate the analysis of XP from the game theory perspective. This analysis illustrates how XP enhances trust among team members and methodologically leads them into cooperation-cooperation situations. Specifically, we explain how, from a game theory perspective, two of the XP values (simplicity and courage) as well as several of the XP practices, lead to the establishment of development environments, the atmosphere of which can be characterized by the cooperation-cooperation cell of the prisoner's dilemma (cf. Table 2).

In the case of the value of *courage*, cooperation implies, among other things, that all team members have the courage to admit and to state explicitly when something goes wrong. From the individual's point of view, this means that no one is conceived of as a complainer if one issues a warning when something goes wrong. As Beck says, from the individual's point of view, a warning is worthwhile only if one knows that the organization encourages such behavior and that the other members of the organization behave similarly. Otherwise, it is not worth expressing courage, as such behavior might be conceived of as a disruption, and eventually one might suffer certain consequences as a result of such behavior. In our case, the organization is the XP development environment. Consequently, all team members are committed to the value of courage, all of them can be sure that they are not the only ones to expresses courage, and no one faces the dilemma whether to cooperate (that is, to issue a warning when something goes wrong) or not. As described in Table 2, all team members benefit from this cooperation.

With respect to the value of simplicity, cooperation is expressed when all activities related to the software development are conducted in the simplest possible way. Thus, for example, all team members are committed not to complicate the code they write, but rather to develop clear code, which is understood by all team members. As all team members are committed to working according to this norm, the development environment is stabilized within the cooperation-cooperation cell, a less complicated development environment is established (for example, the code is simpler), and all team members benefit.

In what follows, several of the XP practices are analyzed by the prisoner's dilemma framework.

[3] An interview with Kent Beck, June 17, 2003, *Working smarter, not harder*, IBM website: http://www-106.ibm.com/developerworks/library/j-beck/

Test-First Programming: Cooperation in this case means writing tests prior to the writing of the code; competition means code writing that is not in accordance with this standard. The rational behind this practice relate to scope creep, coupling and cohesion, trust and rhythm (Beck, 2005, p. 50-51). Specifically, Beck says: "Trust – It's hard to trust the author of code that doesn't work. By writing clean code that works and demonstrating your intentions with automated tests, you give your teammates a reason to trust you" (Beck, 2005, p. 51). This perspective even fosters the trust element in XP team. Specifically, from the prisoner dilemma perspective, because all team members are committed to working according to XP in general, they are all committed to adopting the practice of test-first programming in particular. Naturally, this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them apply the test-first programming practice. Consequently, they all benefit from the quality of software that is developed according to test-first programming.

Collective Ownership: Since XP is the development environment, all team members are committed to apply all the XP practices and, in particular, the practice of collective ownership. The meaning of cooperation in the case of collective ownership is code sharing by all team members; whereas competition means concealing of information. This, however, is not the full picture. In addition, each team member knows that the other team members are also committed to working according to XP. Specifically, with respect to the practice of collective ownership this means that they are all committed to sharing their code. In other words, the practice of collective ownership implies that since all team members are committed to working according to XP, they all cooperate, and share their codes. Thus, the unknown behavior of the others, which is the source of the prisoner's dilemma, ceases to exist. Consequently, team members face no (prisoner's) dilemma whether to cooperate or not, and since they are guided by the practice of collective ownership, they all cooperate and share their code with no concern about whether their cooperation will be reciprocated or not. Since, for purposes of software quality, it is required that knowledge be passed on among team members, all team members benefit more from this practice than if they had chosen to be in competition with one another. Thus, the practice of collective ownership yields a better outcome for all team members.

Coding Standards: Cooperation in this case means writing according to the standards decided on by the team; competition means code writing that is not in accordance with these standards. Because all team members are committed to working according to XP in general, they are all committed to adopting the practice of coding standards in particular. Naturally, this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them write according to the code standards. Consequently, they all benefit from the quality of software that is developed according to decided coding standards.

Sustainable Pace: The rationale for this practice is that tired programmers cannot produce code of high quality. Cooperation in this case means that all team members

work only 8-9 hours a day; competition is expressed by working longer hours, in order, for example, to impress someone. The fact that all team members are committed to working according to XP, ensures that they all work at a sustainable pace in the above sense, and that no one "cheats" and stays longer so as to cause his or her contribution to be perceived as greater. Consequently, none of the team members is concerned about competing in this sense. Eventually, all team members benefit from the practice of sustainable pace.

Simple/Incremental Design: Cooperation in this case means to "strive to make the design of the system an excellent fit for the needs of the system that day (Beck, 2005, p. 51); competition means for example to add (sometimes) redundant features to the code, thus complicating the design. Because all team members are committed to working according to XP in general, they are all committed to adopting this practice in particular. Naturally, from the prisoner dilemma perspective this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them keep the design simple and incremental. Consequently, they all benefit from the quality of software that is developed according to simple/incremental design.

Pair-Programming: Cooperation in this case means to accept the rules of pair programming, such as switching between driving and navigating; competition means not to accept these rules, for example, to code without a pair. From the prisoner dilemma perspective, because all team members are committed to working according to XP in general, they are all committed to adopting the practice of pair programming in particular. Naturally, this commitment leads all team members to be in the cooperation-cooperation cell. Since the entire team works according to XP, all team members are sure that the other team members will cooperate as well. Thus, none of them face the dilemma of whether to cooperate or not and all of them apply the rule of the pair programming practice enabling a constant code inspection. Consequently, they all benefit from the quality of software that is developed according to the practice of pair programming.

We believe that at this stage the readership can apply a similar analysis with respect to the other XP practices. Clues for the need of such analysis of software development environments are presented in previous writings. For example, Yourdon (1997) says that "In the best of all cases, everyone will provide an honest assessment of their commitment and their constraints". (p. 65). Further, Yourdon tells about "Brian Pioreck [who] reminded me in a recent e-mail message that it's also crucial for the team members to be aware of each other's level of commitment, which the project manager can also accomplish through appropriate communication: I think you also have to make their commitments public through the use of project plan. Everyone sees the total involvement of all team members this way and what their own involvement means to the project." The contribution of the **Planning Game** to the achievement of this commitment of all team members is clearly derived from the analysis of the planning game from the prisoner dilemma perspective.

4 Conclusion

Cockburn (2002) says that "software development is a group game, which is goal seeking, finite, and cooperative. […] Software development is […] a cooperative game of invention and communication. There is nothing in the game but people's ideas and the communication of those ideas to their colleagues and to the computer". (p. 28). This article also applies a game oriented perspective at software development processes. Specifically, this article uses the prisoner dilemma, a well known game-theory framework, for the analysis of software development methods in general and for the analysis of XP in particular. It is suggested that this article demonstrates how a software development method can be analyzed, not only by referring to its technical benefits but, rather, by suggesting ways in which the software development method is viewed from a social perspective.

The scope of this paper is limited to the examination of a software development method in general and of XP in particular by one framework. It is suggested that similar examinations in at least three directions can be carried out.

First, other software development methods can be analyzed using the prisoner dilemma framework;

Second, the application of other game theory methods for the analysis of software development methods can be checked. Kent, for example, has demonstrated the benefits of Win-Win situations: "Mutual benefit in XP is searching for practices that benefit me now, me later and my customer as well". (Kent, 2005, p. 26). Further, we may analyze situation that should be avoided in software development methods from a game theory perspective. For example, we may analyze the influence of zero-sum situations of software development processes.

Third, it is worth examining the analysis of software development methods based on additional research frameworks and theories borrowed from other disciplines. For example, currently we explore the application of the cognitive theory constructivism (cf. Piaget, 1977; Davis, Maher and Nodding, 1990; Smith, diSessa and Roschelle, 1993), which examines the nature of learning processes, for the analysis of software develop methods.

References

1. Beck, K. (2000). Extreme *Programming Explained: Embrace Change*. Addison-Wesley.
2. Beck, K. (with Andres, C., 2005, second edition). Extreme *Programming Explained: Embrace Change*. Addison-Wesley.
3. Cockburn, A. (2002). *Agile Software Development*. Addison-Wesley.
4. Davis, R. B., Maher, C. A. and Noddings, N. (1990). Chapter 12: Suggestions for the improvement of mathematics education. In Davis, R. B., Maher, C. A. and Noddings, N. (eds.). *Journal for Research in Mathematics Education, Monograph Number 4, Constructivist Views on the Teaching and Learning of Mathematics*, The National Council of Teachers of Mathematics, Inc., pp. 187-191.
5. Dubinsky, Y. and Hazzan, O. (2004). Roles in agile software development teams, *Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering*, Garmisch-Partenkirchen, Germany, pp. 157-165.
6. Dines, B. (2003). What is a method?, an essay on some aspects of domain engineering. Monographs In *Computer Science - Programming methodology*, pp. 175 - 203.

7. Hazzan, O. and Dubinsky, Y. (2003A). Teaching a Software development methodology: The Case of Extreme Programming, The proceedings of the *16th International Conference on Software Engineering Education and Training*, Madrid, Spain, pp. 176-184.
8. Hazzan, O. and Dubinsky, Y. (2003B). Bridging cognitive and social chasms in software development using Extreme Programming, *Proceedings of the Fourth International Conference on eXtreme Programming and Agile Processes in Software Engineering*, Genova, Italy, pp. 47-53.
9. Piaget, J. (1977). Problems of Equilibration. In Appel, M. H and Goldberg, L. S. (1977). *Topics in Cognitive Development, Volume 1: Equilibration: Theory, Research and Application*, Plenum Press, NY, pp. 3-13.
10. Smith, J. P., diSessa, A. A. and Roschelle, J. (1993). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The Journal of the Learning Sciences,* **3**(2), pp. 115-163.
11. Tomayko, J. and Hazzan, O. (2004). *Human Aspects of Software Engineering*, Charles River Media.
12. Yourdon, E. (1999). *Death March*. Prentice Hall.

A Framework for Understanding the Factors Influencing Pair Programming Success

Mustafa Ally, Fiona Darroch, and Mark Toleman

Department of Information Systems, University of Southern Queensland
Toowoomba Qld 4350 Australia
{Mustafa.Ally,Fiona.Darroch,Mark.Toleman}@usq.edu.au

Abstract. Pair programming is one of the more controversial aspects of several Agile system development methods, in particular eXtreme Programming (XP). Various studies have assessed factors that either drive the success or suggest advantages (and disadvantages) of pair programming. In this exploratory study the literature on pair programming is examined and factors distilled. These factors are then compared and contrasted with those discovered in our recent Delphi study of pair programming. Gallis et al. (2003) have proposed an initial framework aimed at providing a comprehensive identification of the major factors impacting team programming situations including pair programming. However, this study demonstrates that the framework should be extended to include an additional category of factors that relate to organizational matters. These factors will be further refined, and used to develop and empirically evaluate a conceptual model of pair programming (success).

1 Introduction

Pair programming is a core (some would say mandatory) practice of eXtreme Programming (XP) [2], and commonly applied and or recommended for use in conjunction with many other Agile software development methods including Feature Driven Development, Scrum, Lean Software Development, Crystal, and Dynamic Systems Development Method.

Various definitions of pair programming have been proposed [12, 20, 23]. Jensen [12] describes it as 'two programmers working together, side by side, at one computer collaborating on the same analysis, design, implementation, and test'. Compared to traditional programming where typically one programmer is responsible for developing and testing their own code, in pair programming every code fragment is developed by a team of two programmers working at the same workstation. There are two roles, viz, a driver controlling the mouse, keyboard or other input device to write the code and unit tests, and a navigator, observing and quality assuring the code, asking questions, considering alternative approaches, identifying defects, and thinking strategically. The partners are considered equals and will regularly swap roles and partners [22].

The need for this study arises from the lack of an all-encompassing theory about the factors that influence pair programming success and the extent of this influence. Pair programming may have a basis in theories of group problem

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 82–91, 2005.
© Springer-Verlag Berlin Heidelberg 2005

solving and decision making [8, 14], but any explicit reference to such theories for its use seems difficult to locate. Most research has tended to be fragmented and restricted to specific issues. A useful framework for research on team programming situations is found in [8]. This paper builds on and extends their preliminary work by adopting a holistic approach and analyzing and synthesizing the literature findings and empirical data collected for this study, with the future aim of developing a conceptual model of pair programming success.

This paper reports an analysis of the literature involving empirical research on pair programming, to discover theoretical concepts and factors relevant to the practice. Since pair programming is one of the more contentious aspects of Agile system development (particularly eXtreme Programming), it has attracted much attention and consequently become the focus of numerous studies, both in the field with real-life examples and professionals, and in an educational context with students as subjects. These studies have employed a range of research methods to investigate this phenomenon including: case studies, experience reports, surveys and experiments. Space precludes all the literature being presented and represented here but that mentioned typifies the studies and views taken.

The paper also reports a comparison of the factors arising from the literature analysis and our recent Delphi study of pair programming [19]. Briefly, the Delphi technique emerged from work at the US Department of Defence and the RAND Corporation in the 1950s. It is a qualitative, structured group interaction technique and its use is well documented [5, 17]. The objective of the technique is to allow the researcher to obtain a reliable consensus from a panel of experts where the phenomenon or situation under study is political or emotional and where the decisions affect strong factional interests. The technique then collates, synthesizes and categorizes those opinions until general consensus is reached. In the Delphi study, 20 participants engaged in three (3) rounds to reach consensus on issues about pair programming from both an organizational and an individual's perspective. The Delphi participants, comprising academics, software developers and managers, were selected on the basis of their pair programming experience, software development expertise, and industry reputation. Participants were drawn from a range of different types and sizes of organizations to give a broad perspective to the study. In the text that follows, selected representative quotations from Delphi participants are shown in 'quotes'. This study reports partial findings from the Delphi study. Further in-depth analysis of the Delphi study data and the development of models to relate the factors identified with measures of pair programming success are yet to occur.

This paper is structured as follows. The next section identifies concepts found in both the literature and from the Delphi study and where the views relevant to pair programming coincide. In the third section concepts related to pair programming, where the views are opposing, are reviewed. The fourth section reviews concepts arising in only one source: either the literature or the Delphi study but not both. The fifth section aligns the factors identified in this study to the initial framework proposed by [8]. The final section offers conclusions and expectations for future work.

2 Concepts in Common: Literature and Delphi

This section identifies concepts found in both the literature and in the Delphi study, where the views relevant to pair programming coincide. Literature relevant to the concept is reported, as is representative mention of the concept in the Delphi study providing further validation of the practical implications of the concept. Fifteen concepts are identified.

Quality. There are many references in the literature that support the concept of improved quality arising from pair programming situations. Poole and Huisman [15] suggest that pair programming results in improved engineering practices and quality, as evidenced by low error rates. Improved quality was manifested in earlier bug detection/prevention [4]. Experiments with 'industrial' programmers showed error rates were reduced by two-thirds and needed less iterations to fix them when pairing [12, 13]. Equally, in the Delphi study the issue of improved quality was raised many times, both from organizational and individual perspectives. The general consensus was that code quality improved through the pairing process resulting in fewer bugs and better designs. Also on the issue of code maintenance 'usually if someone is programming with you, better choices are made for variable names, better structure, (and) programmers aren't as lazy keeping coding standards'.

Team Building and Pair Management. Two aspects of team management raised in the literature were that pair programming engenders a team spirit; and that there is a need for training in team building [12, 18]. The Delphi study affirmed the import of these team management concepts, emphasizing that pairing is a social activity where 'one has to learn how to work closely with others, (to) work effectively as a member of a team'. The need to develop these skills was highlighted in the Delphi study where 'traditionally, IT study/training alone does not equip individuals with the interpersonal skills required for effective pairing'. The Delphi also raised issues related to managing the paired programming process, including pragmatic issues such as pair rotation and 'what do you do when there are odd numbers of people on a team?'; resolving personality conflicts, for example where 'an obsessively neat person (is required to) work with a messy person' and 'people that no-one wants to work with'; and logistical issues such as when 'pairs need to start/end work at the same time'.

Pair Personality. Dick and Zarnett [6] identified personality traits as playing a vital role in the success, or otherwise, of pair programming. The Delphi study identified two specific issues that need to be addressed: personality conflicts 'when two people have different ideas, or different styles of dealing (with) problems' and 'some people just don't like accepting other people's suggestions/ideas'; and divergent personal styles where 'some programmers like to discuss with others, while some do like to work on issues by themselves' and '(pairing) strong personalities together with weak personalities'.

Threatening Environment. Both [6, 18] highlight the problem for individuals in the pair of the fear of feeling and/or appearing ignorant on some programming or system development aspect to one's partner. The Delphi Study supported

this indicating that working in pairs may expose an individual's weaknesses and competencies. Comments included: 'Most people new to pairing find the prospect frightening/threatening – will I appear stupid/ignorant?' and 'Sharing knowledge (and ignorance) on a daily basis can be threatening'.

Project Management. Pair programming raises issues for project management. On the positive side, it may act as a backup for absent or departing developers [7, 23]. This was also reflected in the Delphi study with 'no one person has a monopoly on any one section of the code, which should remove organizational dependencies on particular resources and mitigate risk to the business'. On a less positive note, there are challenges for project management in terms of planning and estimation. This was highlighted in the Delphi study by 'Methods of planning/estimating need to change when (a) team is pair-programming rather than tackling tasks as individuals'.

Design and Problem Solving. There is ample evidence that pair programming results in improved design and problem solving through the removal of 'tunnel vision' and the exchange of ideas [12, 16, 23]. It is especially suited to dealing with very complex problems that are too difficult for one person to solve [4, 21, 23]. Experiments have provided supporting empirical evidence about improved design [13]. This sentiment was affirmed in the Delphi study where it was felt that design decisions and difficult problem resolutions would be superior, and that 'it very often solves complex problems much more effectively than a single person would', as well as the potential to 'find problems in advance'.

Programmer Resistance. An important issue for management consideration is that many programmers resist (at least initially) pair programming. There are many facets to this issue including: a reluctance to share ideas; ego problems where some people think they are always right; and lack of trust where comments may be taken as personal criticism [18]. Therefore, there is a need for strategies to introduce pair programming 'softly', recognizing that even after coaching some programmers resist working in pairs [18]. In the Delphi study the resistance to pairing was also raised, especially among the 'old-school programmers who find it difficult to change habits'.

Communication. The literature cites communication as integral to pair programming [7], and that it helps to get people to work better [4]. The Delphi study also supported that pair programming requires and 'fosters communication skills in the team', and 'improves interactions between team members'.

Knowledge Sharing. The literature proposes that pair programming presents opportunities for improved knowledge transfer [7]. Pairs learn a great deal from one another [4] including changed behaviour, habits, ideas and attitudes [18]. The Delphi study also cited improved knowledge transfer in a variety of contexts including: that the programmers' knowledge became more broad-based; that it enabled a concurrent understanding rather than a post-explanation; that it resulted in more thorough domain knowledge; and that it could act as a backup/contingency plan in cases of illness or resignation. In contrast, the Delphi study also raised some negative aspects of knowledge sharing viz. that pairing

may result in programmers having a broader, but shallower understanding of the system; and that some may find knowledge sharing to be threatening.

Mentoring. The literature found that pair programming provides an ideal environment that greatly facilitates mentoring [6, 15]. Positive outcomes were enjoyed by senior staff [15], as well as less experienced programmers, who learned from their more experienced partners [12, 13]. Several responses in the Delphi study attested to the mentoring benefits arising out of their pair programming experiences: 'There is a fast tracking of skills development with careful choice of pairs (for example) mentor/ junior role' and 'it can really help with the development of new programmers'. However, 'it helps if you have a good programmer who is able to explain, or teach, an inexperienced programmer'. The point where mentoring becomes training was raised in the Delphi study as evidenced by 'that each (developer) has a say in how the task is to go ahead (that is) it is a team not a mentor/junior process. In this case I don't think it is pair programming but more like training'.

Environment Requirements. An unsuitable physical environment may act as a barrier to pair programming success [12]. This was reflected in the Delphi study. The physical environment should facilitate two programmers working at a single workstation because 'most single person desks are not comfortable for two people to sit at' and 'our desks are L-shaped, and as such do not allow two developers to sit side-by-side comfortably'. This work environment may be more disruptive as 'good pairs interact constantly'. Of course individual environmental preferences may vary, for example a liking for differing styles of keyboards.

Effective Pairs. While there is agreement that the dynamics of the pairs needs to be carefully considered, there is no agreement as to what constitutes the most effective pair combinations. For instance, [12] suggests that it is counterproductive to pair two programmers of equal skill. This sentiment was also reflected in the Delphi study: 'for two equally competent programmers I see this as a waste of resource'. The contrary view was also expressed that 'pair programming between experienced programmers is often more useful when it comes to making design decisions'. Two instances where effective pairing may produce beneficial results are (1) where a new developer is placed in a pairing situation and can start being productive immediately, and (2) where a junior programmer might need mentoring. However, the Delphi study revealed that the dynamics of the pairs needs to be considered carefully. Many combinations would not work well: a novice programmer could slow down (and potentially annoy) a skilled programmer, while lowering the self-esteem of the former; two skilled programmers working together could have the effect of negating productivity benefits, for instance when the navigator becomes increasingly frustrated at the lack of involvement, or when there is contant 'clashing of the minds'; two novice programmers could benefit from pair programming, but they run the risk of 'the blind leading the blind'.

Shared Responsibility. Both [1, 18] argued that by spreading responsibility and decision-making load, pairs effectively 'halve' the problem solving. Individuals feel more confident about the decisions made, and less overwhelmed by

decision-making responsibilities. The Delphi study respondents agreed, noting that 'new developers can feel more confident about attacking complex code because there is someone else there with them', and further that they are 'helping someone else with their assigned duties'.

Human Resource Management. Pair programming has implications for the recruitment process of hiring programmers [6]. It also challenges traditional human resource ideas of individual-based performance evaluation and remuneration. These team-based approaches need new management strategies to be considered that are significantly different from those traditionally used for software developers [18]. Many of these human resource issues were raised in the Delphi study generally: 'traditional performance measures focus on the individual – how do you map entrenched HR practices/requirements of a large organization to a collaborative team structure'; from an organizational perspective: 'emphasis moves to team success rather than individual success'; and from an individual perspective: 'a programmer cannot look at a subsystem and say "I did that"; success is now team-based, not individual-based'.

Attitude. A stereotypes of programmers is the 'lone-hacker'. Pair programming has been shown to change programmers' attitude from withdrawn, introverted and worried, to outgoing, gregarious and confident [1]. Delphi study participants agreed, noting that 'some people have entered the industry because it is one where they can be alone for long periods' but that 'one has to learn how to work closely with others' and 'work effectively as a member of a team'.

3 Opposing Perspectives: Literature and Delphi

A significant finding of this study is that some of the issues raised in the literature were also raised in the Delphi study, but from opposing viewpoints. It is notable that for these factors the literature is consistently positive about the concept in contrast to the Delphi study in which the same issues appear as barriers or hindrances. Thus the Delphi study effectively contradicts the practical implications of the concept as it is presented in the literature.

Morale. The literature suggests that morale can be improved by using pair programming, especially when working on a difficult or complex system. This morale 'boost' may be in the form of positive reinforcement by peers [15] but also because 'there's someone there to celebrate with when things go right' [1]. In the Delphi study, the impact of pair programming on morale was raised in a negative light for example when it came to a mismatch of skills: 'there's a high probability one member of the pair will resent the other one and lower their morale whilst working with this person'.

Productivity. The literature cites many examples of improved productivity arising from pair programming [12, 15, 18]. This includes experiments with 'industrial' programmers [12, 13]. This was in part attributed to a shared conscience where pairs are less likely to indulge in time-wasting activities [23]. Pairs wasted less time trying to solve problems compared to working alone [12]. However in the main, the Delphi study suggested lower productivity or at least perceptions

of lower productivity. In particular, management is yet to be convinced of the productivity benefits of pair programming: 'corporate viewed pairing as being ... twice as slow as traditional development'. In certain pairing scenarios pairing was seen to be less productive: 'two top programmers would (have) lower productivity'. It was even suggested that the quantity of code 'usually goes down per hour when taking into account the number of people working on it'. In contrast, it was suggested by one participant that the definition of code generation needed to be considered in developing any measure of productivity: 'if design reviews are accepted as being part ... then productivity gains are higher'.

Development Costs. Clearly, development costs are an important issue for software construction. The literature suggests code costs are slightly higher ([21] suggests 15%) with pair programming, but that it is offset by improved code quality, and minimization of and the earlier detection of bugs [4, 13]. Delphi study participants were far from convinced. They noted the problem for management of an apparent doubling of cost for development of the same feature: 'development throughput is reduced, not only by halving the number of people actively coding simultaneously, but also because there is additional collaboration on the design of the code'. An interesting take on the situation was that 'a pair wasting time costs twice as much as a single developer wasting time'.

Enjoyment of Work. Students and professional programmers report finding their work to be more enjoyable when pairing [23]. However, this is an opposing viewpoint to the Delphi study where it was described as an unpopular activity that resulted in lowered personal satisfaction.

4 One Source Concepts: Literature and Delphi

Some factors were raised in either the literature or the Delphi, but not both, which may suggest that saturation of all the issues involved has not yet occured.

Factors that appeared only in the literature included: **project schedule** potential where project timelines can be realistically shortened through a change in workflow to a more speedy iteration of plan, code, test and release [1, 3, 9, 24]; **fit of pair programming to project type** where experiments have shown that pair programming is especially suited to situations characterized by changing requirements, and unfamiliar, challenging or time-consuming problems [13]; **code readability** where source code readability is greatly enhanced by using pair programming [10]; and **distributed pair programming** where appropriate tools can assist distributed pair programming where co-location is not possible [11].

Factors that arose only in the Delphi study included: **collective code ownership** through pair programming minimizes the introduction of coding flaws and enhances concurrent understanding of the code base; **accountability** concerns the shift of responsibility from the individual to the pair through collective code ownership; **customer resistance** to pair programming through the perception of increased costs; **organizational culture** and its influence on the acceptance of pair programming, and the influence of pair programming on the organization; and **solitude and privacy opportunities** are reduced when pair programming, with the increased potential for stress and 'programmer burnout'.

These factors (and possibly others) will be more fully analyzed prior to incorporation into a conceptual model.

5 Extension of Gallis et al. (2003) Framework of Factors

An initial framework for research on pair programming has been proposed ([8] summarized in their Fig. 1). While their study was based on four different configurations of team programming, this study focuses specifically on pair programming. In addition, [8] considered a specific set of literature including their own pair programming studies in developing their research framework. This study extends their framework by considering additional literature, and individual and organizational issues identified in our Delphi study. They identified dependent variables (time, cost, quality, productivity, information and knowledge transfer, trust and morale, and risk) which were affirmed and context variables, which were affirmed but further elaborated in this study (see Table 1).

Table 1. Extension and elaboration of the Gallis et al. (2003) framework context variables (sections where the variable appears in this paper are shown in parenthesis)

Gallis et al. (2003)	This study
Subject variables	
Education & experience	Mentoring (2)
Personality	Pair personality (2) Programmer resistance (2)
Roles	Shared responsibility (2)
Communications	Communication (2)
Switching partners	Project management (2) Effective pairs (2) Attitude (2) Enjoyment of work (3) Knowledge sharing (2) Threatening environment (2)
Task variables	
Type of development activity	Design & problem solving (2) Code readability (4)
Type of task	Fit of pair programming to project type (4)
Environment variables	
Software development process	Project schedule (4)
Software development tools	Distributed pair programming (4)
Workspace facilities	Environment requirements (2) Solitude & privacy (4)
Organizational variables	
	Team building & pair management (2) Human resource management (2) Accountability (4) Customer resistance (4) Organizational culture (4) Collective code ownership (4)

Also, our findings reveal an additional category of context variables, namely organizational factors, which were repeatedly raised by the Delphi study participants. The impact of the adoption of pair programming on organizations and the impact of organizational culture on the practice of pair programming are clearly important issues for further consideration. This extended framework will form the basis for the development of a model of pair programming success.

6 Conclusions

Pair programming is controversial: the diverse literature on the practice and discussion with practitioners confirms the variety of factors affecting its success as a practice, how it is viewed by practitioners, and its impact on software development success. While there is a great deal of evidence in the literature about pair programming success, there is much work to be done by an organization to properly prepare for its implementation (especially overcoming resistance), and it is clear that many of the 'people' issues require in-depth consideration.

This study suggests that further research is required particularly to examine the breakdowns, that is, where the literature and practitioners hold opposing views. In addition a more complete analysis is required of those factors that appear in only one of either the literature or the practitioner experience.

Dependent and independent variables have been identified in the framework, but further refinement is necessary. The next step is to formulate a conceptual model of pair programming (success), which can be quantitatively tested. This work is in progress. There is a need for multi-disciplinary and mixed-method research that will uncover behavioural strategies for a more complete understanding of the complexities of the human aspects of pair programming. Other research includes pair programming experiments with students and practitioners.

References

1. Baer, M.: The New X-Men, viewed 1/12/04, `http://www.wired.com/wired/archive/11.09/xmen.html?pg=1&topic=&topic_s` (2003).
2. Beck, K.: *Extreme Programming Explained: Embrace Change*, Addison Wesley, Boston (1999).
3. Brooks, F.P.: *The Mythical Man-Month Essays on Software Engineering*, Anniversary Edition, Addison-Wesley, Boston (1995).
4. Cockburn, A. and Williams, L.: The Costs and Benefits of Pair Programming, *Proceedings of XP2000*, Sardinia, Italy, June 21–23 (2000).
5. Day, L.: Delphi Research in the Corporate Environment, in Linstone and Turoff (Eds) *The Delphi Method: Techniques and Applications*, Addison-Wesley, London (1975).
6. Dick, A.J. and Zarnett, B.: Paired Programming and Personality Traits, *Proceedings of XP2002*, Sardinia, Italy, May 26–29 (2002) 82–85.
7. Flies, D.D.: Is Pair Programming a Valuable Practice?, viewed 6/12/04, `http://csci. mrs.umn.edu/UMMCSciWiki/pub/CSci3903s03/StudentPaperMaterials/flies-pairprogramming03.pdf` (2003).

8. Gallis, H., Arisholm, E. and Dybå, T.: An Initial Framework for Research on Pair Programming, *Proceedings of the 2003 International Symposium on Empirical Software Engineering-ISESE 2003* (2003) 132–142.
9. Glass, R.L.: *Software Runaways*, Prentice Hall, New Jersey (1998).
10. Grenning, J.: Launching Extreme Programming at a Process-Intensive Company, *IEEE Software*, **18**(6) (2001) 27–33.
11. Hanks, B.: Empirical Studies of Pair Programming, *2nd International Workshop on Empirical Evaluation of Agile Processes*, New Orleans, Louisiana (2003).
12. Jensen, R.W.: A Pair Programming Experience, *Journal of Defense Software Engineering*, March, viewed 1/12/04, `http://www.stsc.hill.af.mil/crosstalk/2003/03/jensen.html` (2003).
13. Lui, K.M. and Chan, K.C.C.: When Does a Pair Outperform Two Individuals? in M. Marchesi and G. Succi (Eds) *Extreme Programming and Aile Processes in Software Engineering – 4th International Conference, XP 2003* Lecture Notes in Computer Science **2675** (2003) 225–233.
14. Napier, R.W. and Gershenfeld, M.: *Groups: Theory and experience*, Sixth Edition. Boston, Houghton Mifflin Company (1999).
15. Poole, C. and Huisman, J.W.: Using Extreme Programming in a Maintenance Environment, *IEEE Software*, **18**(6) (2001) 42–50.
16. Pulugurtha, S., Neveu, J. and Lynch, F.: Extreme Programming in a Customer Services Organization, *Proceedings of XP2002*, Sardinia, Italy, May 26–29 (2002) 193–194.
17. Rowe, G., Wright, G. and Bolger, F.: Delphi: A Reevaluation of Research and Theory, *Technological Forecasting and Social Change*, **39**(3) (1991) 235–251.
18. Sharifabdi, K. and Grot, C.: Team Development and Pair Programming – Tasks and Challenges of the XP Coach, *Proceedings of XP2002*, Sardinia, Italy, May 26–29 (2002) 166-169.
19. Toleman, M., Ally, M. and Darroch, F.: A Delphi Study of Pair Programming, Working paper, Department of Information Systems, University of Southern Queensland (2005).
20. Wiki: viewed 1/12/04, `http://c2.com/cgi/wiki?PairProgramming` (2004).
21. Williams, L.: The Collaborative Software Process, unpublished PhD dissertation, Department of Computer Science, University of Utah (2000).
22. Williams, L. and Kessler, R.: *Pair Programming Illuminated*, Addison-Wesley, Boston (2002).
23. Williams, L., Kessler, R.R., Cunningham, W. and Jeffries, R.: Strengthening the Case for Pair Programming, *IEEE Software*, **17**(4) (2000) 19–25.
24. Yourdon, E.: *Death March: The Complete Developer's Guide to Surviving 'Mission Impossible' Projects*, Prentice Hall, New Jersey (1999).

Empirical Study on the Productivity of the Pair Programming

Gerardo Canfora, Aniello Cimitile, and Corrado Aaron Visaggio

Research Centre on Software Technology,
University of Sannio, viale Traiano 1, Palazzo ex-Poste, 82100 Benevento, Italy
{canfora,cimitile,visaggio}@unisannio.it

Abstract. Many authors complain the lack of empirical studies which assess the benefits and the drawbacks of agile practices. Particularly, pair programming rises an interesting question for managers: does pair programming mean to pay two developers in the place of one? We realized an experiment to compare the productivity of a developer when performing pair programming and when working as solo programmer. We obtained empirical evidence that pair programming decreases the effort of the single programmer.

Introduction

Pair programming is an agile practice [9] consisting of two developers working at the same code, side by side on the same machine: one develops the code, the other one reviews it. A number of benefits are enumerated in literature, such as:

- ***Pair pressure.*** Working in pairs drives the developer out of his personal 'comfort zone', and requires a major involvement, humility, and awareness [20].
- ***Economics.*** The continuous review of code while writing it, increases sensitively the rate of defects' removal [19].
- ***Satisfaction.*** Working in pairs increases enjoyment, and then commitment of the developers [17].
- ***Knowledge transfer.*** Pair working should increase the knowledge transfer among pair's members [11].
- ***Team building and communication.*** Developers learn to discuss, to communicate, and generally to work in group [20].

At our best knowledge, there is not a mature body of knowledge that validates these conjectures with empirical analyses, although some valuable studies have been accomplished in this direction, as the section Related Works shows. As a matter of fact, some authors remark [1, 10, 21] that there is a need for more case studies and field experiments, possibly with industries, in order to define the actual effects of this practice on the software development process.

Economics, in particular, is the focus of this paper: it is immediate to wonder whether pair programming means to sustain the cost of two developers for the same work that a developer could perform. Some previous papers discuss this concern, and outline that pair programming decreases the effort with respect to a solo programmer with the additional benefit to increase also the quality of the code produced.

The novelty of our study stands in the method of investigation that we have adopted. We analyze the effects of pair programming on each single programmer, by

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 92–99, 2005.
© Springer-Verlag Berlin Heidelberg 2005

comparing the performances when the developer programs in pair and when the same developer programs as solo. Conversely, literature reports researches in which the outcomes of pairs are compared with those of solo programmers, but, at our best knowledge, there is not consolidated results on the effect of pair programming on the performances of an individual programmer.

We realized an experiment in order to answer the research goal, that can be formulated, in accordance to the GQM paradigm [2] as follows: *Analyze* pair programming sessions *for the purpose of* evaluating *with respect to* its capability of reducing the developing time of each single programmer *from the point of view of* the researchers *in the context of* students' groups with different degrees.

The experiment is part of a larger research program that investigates not only the performances allowed by the practice of pair programming, but also: the quality of the code produced; the relationship between pair programming and knowledge [6]; and finally, how pair programming can improve the systems design[5, 7].

The paper continues as follows: the section *Related Work* reviews relevant literature and outlines the distinctive aspects of our work; the section *Experimental Design* describes the experiment; the section *Data Analysis* and the section *Statistical Tests* provide a commentary for the data obtained as results of the experiment; and, finally, the *Conclusion*'s section draws the observation and discusses the future work.

Related Work

The current work is part of a family of experiments, aiming at evaluating how pair programming affects effort and productivity of each programmer with respect to the solo programming.

When the term 'pair programming' was not yet widespread, Nosek investigated Collaborative Programming [16]. Collaborative Programming means 'two programmers working jointly on the same algorithm and code'. Basically, it was a form of pair programming *ante-litteram*. Nosek executed an experiment with experienced programmers and it showed that collaborative programmers outperformed the individual programmers. Interestingly, a secondary observation stemmed from the experiment: collaborative programmers reported higher values of enjoyment of the process and confidence about their work. With the growing interest for pair programming in the software engineering research community, more focalized investigations have been published. However, initially the attention of researchers focussed mainly on quality and productivity. In [19] the authors realized a structured experiment in order to understand the difference between pair and solo programming. The results showed that the time was reduced up to the 50%. The authors found other outcomes, concerning quality and enjoyment of the participants. In [15], the authors compared the pair programming with the Personal Software Process proposed by Humprey. They found that pair programming can decrease the development time but not up the 50% reported by Williams in[19]; pair programming is more predictable than solo programming in terms of development time; and the amount of rework was reduced with the pair programming. In [3] the authors investigate the performance of pair programming when the components of the pair are not co-located: they found that distributed pair programming seems to be comparable to colocated software development in terms of the productivity and quality. Productivity is measured as lines of code per hour.

Recently, the target of pair programming investigation is turning to learning and knowledge transfer. Many authors [11, 14, 17, 20, 22] found that pair programming fosters knowledge leveraging between the two programmers, particularly tacit knowledge and learning. Other studies investigate the benefit of pair programming for improving the overall project [18, 23].

By concluding, a remark deserves attention: at our best knowledge there is not an enough high number of studies comparing performances of pair programming and of solo programming on the same developer is yet too low and pretty close to zero.

Experimental Design

This section illustrates the experiment realized in order to achieve the research goal.

Definition. The experiment was executed with the purpose of testing the following null hypothesis:

> **Ho:** pair programming does not affect the developing time spent by each programmer with respect to the solo programming.
> The alternative hypothesis is:
> **H_1:** pair programming affects the developing time spent by each programmer with respect to the solo programming.

Characterization
Subjects. The experiment was executed with the collaboration of the students of the Master of Technologies of Software (MUTS), an high education university course for post-graduates, at University of Sannio(http://www.ing.unisannio.it/master/). Students of MUTS own a scientific graduation (engineering, mathematics, physics). The course provides the basic education in computer engineering (operating systems, programming languages, network, database, and software engineering) and the students attend theoretical classes and lab sessions; they develop a large and complex project in connection with an enterprise, participate to seminaries from international experts, perform a three month stage in software companies.
Each subject performed both pair programming and solo programming alternatively.

Variables. the dependent variable is the time and it was evaluated by a time sheet that the subjects filled in during the experimental runs. In the appendix there is an excerpt of the form.

Rationale for the sampling from the population. Students of the MUTS course are suitable for such an experiment because they were attending the course of object oriented programming held by one of the authors of this article.

Assignment. The students were required to develop two applications, one for each run: an excerpt is reported in the appendix.

The process. The students formed the pairs by themselves and they the pairs remained the same in both the runs. In each run the students performed pair programming as well as solo programming, working alternatively at two different applications. The students used ECLIPSE as development environment because they were trained to use it in the courses. The Table 1 shows the experimental design.

Table 1. Experimental Design

Subject	Treatment			
	1st Run		2nd Run	
	Pair	Solo	Pair	Solo
$S_{j.1}$	1.A	1.B	2.A	2.B
$S_{(j+1).1}$	1.B	1.A	2.B	2.A
$S_{j.2}$	1.A	1.B	2.A	2.B
$S_{(j+1).2}$	1.B	1.A	2.B	2.A

The Table 1 shows that the subjects $S_{j.1}$ and the subject $S_{j.2}$, make up the *j-th* pair: they had same assignment (1.A in the 1st run and 2.A in the 2nd run) as pair and as solo(1.B in the 1st run and 1.B in the 2nd run) in both the runs. *J* varied from 1 up to the total number of pairs, that is 12, for an amount of 24 subjects.

Data Analysis

The Table 2 reports the main descriptive statistics in hours. 'Moda' stands for the most frequent value, and in the last row we indicate the ratio between Standard Deviation and the mean in order to understand the normalized interval of variation of the values in the sample.

The statistic shows that when the subjects worked in pairs performed better than when they worked as solo in each run. This appears evident by the mean value: in the first run the subjects working as solo spent 61% more than the subjects working in pair; whereas in the second run the increment was the 3%. The huge difference between the runs (61-3) can be explained as follows: the assignment of the second run was more complex than that of the first run. Thus, the subjects spent more time in the second run for establishing the strategies to follow. By observing the other values, the difference between the effort spent in pair and that as solo is more significant. The most frequent value for the solos in the first run is two times that of subjects in pairs, and in the second run the ratio is 2,31. A similar ratio is maintained for the maximum value.

The standard deviation is smaller for pairs than for solos' samples: this suggests that working in pairs tends to limit the effort in a certain interval: that is the effort of pairs is much more stable and predictable than that of solos. In both the runs the ratios between the standard deviation and the mean is smaller for pairs than for solos.

The Figure 1 compares some of the most relevant indicators.

Table 2. Descriptive Statics of the Experiment

Statistical variable	Treatment			
	Pair 1st run	Solo 1st run	Pair 2nd run	Solo 2nd run
Mean	1,88	3,03	2,52	2,60
Max	2,1	5	4	5,1
Min	1,3	1,4	1,3	1
Std Dev	0,31	1,17	0,97	1,26
Mode	2	4	1,3	3
Std Dev /Mean	0,16	0,39	0,38	0,48

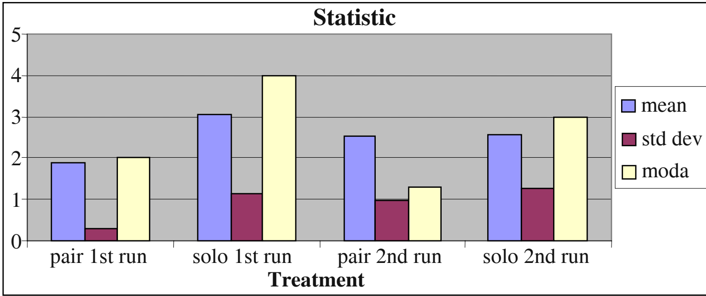


Fig. 1. Main Statistics of the Experiment

Another interesting indicator is the difference between the time each subject spent working solo and the time each subject spent working in pair and the descriptive statistic for this indicator is reported in Table 3. More specifically, the indicator is:

$$t_{solo,i,k} - t_{pair,i,k} \text{ ,}$$

where $t_{solo,i,k}$ is the time the i-th subject spent working as solo in the k-th run, whereas $t_{pair,i,k}$ is the time the i-th subject spent working in pair in the k-th run.

The 'negatives' row indicates the percentage of the negative values, that is the number of subjects who spent a longer time by working in pair. The percentage is relatively low; consider that the minimum is –0,85 hours for the first run and –1,6 hours for the second run.

It is interesting to notice that there are cases in which the maximum value of the difference is three hours: if we consider that the maximum value of solos in both runs is about 5 hours, it means that some subjects in pairs diminish of the 50% the effort. Anyway the mean is positive and this allows us to understand that by working in pairs does not requires increasing developing time and in some cases it halves the time required by the solo.

Table 3. Difference of the Efforts Spent by Subjects between the two Treatments

Statistical Variable	1st Run	2nd Run
Mean	0,69	0,58
Max	3	3
Min	-0,85	-1,6
Mode	0	0
Std dev	1,08	1,19
Negatives	12,5 %	16,67%

Another interesting note is that the most frequent value is zero: some subjects (three for the first run and six for the second run) did not varying the time between the pair and solo treatment. The Figure 2 shows the graph of the values.

Statistical Tests

The Table 4 reports the outcomes of the statistical tests used. We performed a Mann-Withney test because the data of samples did not have a normal distribution and we fixed the p-level at 0.05.

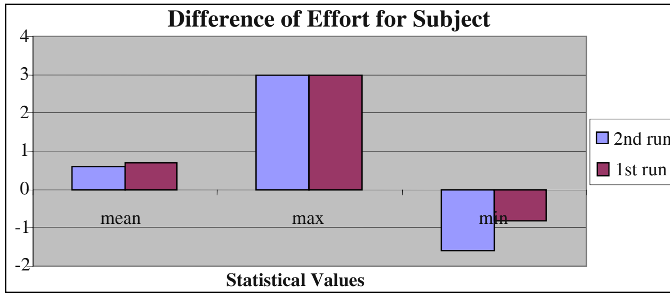


Fig. 2. Descriptive Statistics for each Subject when Performing Pair and Solo Programming

Table 4. Results of the statistical Test

Test Between	Rank sum (α)	Rank sum (β)	p-level
Subjects in Pair (α) Subject as solo (β) In the 1st run	274,000	164,000	**0,04427**
Subjects in Pair (α) Subject as solo (β) In the 2nd run	487,000	123,000	**0,03512**
Subjects in Pair in the 1st run (α) Subject in Pair in the 2nd run (β)	410,00	410,000	1,00000
Subjects as solos in the 1st run (α) Subject as solos in the 2nd run (β)	572,000	463,000	0,94537

The first and the second row shows an ***empirical evidence*** that the differences between the subjects‘ performances in pairs and as solos are statistical significant, both in the first run and in the second run.

The third and the fourth row show that there ***is not empirical evidence*** that the differences between the pairs’ performances between the two runs are significant: this demonstrates that the maturity treat did not affect the results of the experiment.

Conclusion

Many authors complain the lack of empirical evidence about the expected benefits of the agile practices. In this work we focus on the costs of pair programming: we realized and experiment with the aim of understanding the cost implication of a programmer working in pair. The research question intends to determine if pair programming means to sustain the cost of two programmers instead of one.

Our experiment provides ***empirical evidence*** *that the same developer decreases the time for developing a task when moves to pair programming from solo programming.* We executed two runs in which we obtained the same result.

The novelty of our work stands in the research method: we investigate the effect of pair programming on the each programmer, comparing the results when the developer moves from solo programming to pair programming.

An exhaustive discussion on the experimental treats cannot be presented in this paper, but in the following a brief discussion of the main two ones is afforded.

The first one concerns the maturation of the subjects. The statistical tests make us sure that the subjects during the two run did not mature a better way to perform pair programming. The second one concerns the use of student in the experimentation. Students play a very important role in the experimentation in software engineering: in situations in which the tasks to perform do not require industrial experience the experimentation with students is viable [4, 8, 12, 13].

As future work we plan to: (i) make further replicas in academic and industrial environments in order to enforce the external validity of the outcomes, and (ii) comparing the quality produced by solos and that produced by the pairs.

References

1. Abrahamsson P., and Koskela J. “Exterme Programming: A Survey of Empirical Data from a Controlled Case Study”, Proc. of International Symposium on Empirical Software Engineering, Redondo Beach, CA, 2004, IEEE CS Press, pp.73-82.
2. Basili V. R., “Software modeling and measurement: The Goal/Question/Metric paradigm”, Technical Report CS-TR-2956, Department of Computer Science, University of Maryland, College Park, MD 20742, 1992.
3. Baheti P., Williams L., Gehringer E., Stotts D., and McC.Smith J., “Distributed Pair programming: Empirical Studies and Supporting Environments”, Technical Report TR02-010, Department of Computer Science, University of North Carolina at Chapel Hill, March 2002.
4. Basili V., Shull F., and Lanubile F., “Building Knowledge Through Families of Experiments”, IEEE Transactions on Software Engineering, 25 (4), IEEE CS Press, 1999.
5. Bellini E., Canfora G., Cimitile A., Garcia F., Piattini M., and Visaggio C.A., “The impact of educational background on design knowledge sharing during pair programming: an empirical study”, Proc. of Knowledge Management of Distributed Agile Processes 2005, Kaiserslautern, Germany, April 2005, Springer LNCS.
6. Canfora G., Cimitile A., and Visaggio C.A., “Lessons learned about Distributed Pair programming: what are the knowledge needs to address?”, Proc. of Knowledge Management of Distributed Agile Process-WETICE, Linz, Austria, 2003, IEEE CS Press, pp. 314-319.
7. Canfora G., Cimitile A., and Visaggio C.A., “Working in Pairs as a Means for Design Knowledge Building: An Empirical Study ”, Proc. of 12th International Workshop on Program Comprehension, Bari, Italy, 2004, IEEE CS Press, pp.62-69.
8. Carver J., Jaccheri L., Morasca S., and Shull F., "Using Empirical Studies during Software Courses”, Experimental Software Engineering Research Network 2001-2003. LNCS 2765, 2003.
9. Cockburn A., Agile Software Development, Addison-Wesley Pub Co, 2001.
10. Gallis H., Arisholm E., and Dyba T., “An Initial Framework for Reasearch on Pair Progamming”, Proc. of International Symposium on Experimental Software Engineering,, Rome, Italy, 2003, IEEE CS Press, pp. 132-142.
11. McDowell C., Werner L., Bullock H., and Fernald J., “The Effects of Pair Programming on Performance in an introductory Programming Course”, Proc. of the 33rd Technical Symposium on Computer Science Education, Northern Kentucky - The Southern Side of Cincinnati, ACM, March 2002.
12. Kitchenham B., Pfleeger S., Pickard L., Jones P., Hoaglin D., El Emam K., and Rosenberg J., “Preliminary Guidelines for Empirical Research in Software Engineering”. IEEE Transactions on Software Engineering, 28 (8), 2002, IEEE CS Press.

13. Höst M., Regnell B., and Wholin C., "Using Students as Subjects – A comparative Study of Students & Professionals in Lead-Time Impact Assessment", Proc. of 4th Conference on Empirical Assessment & Evaluation in Software Engineering (EASE), 2000.
14. McDowell C., Werner L., Bullock H., and Fernald J., "The Impact of Pair Programming on Student Performance, Perception and persistence". Proc. of the Int'l Conference on Software Engineering (ICSE 2003),Portland, Oregon, USA, 2003, IEEE CS Press, pp.602-607
15. Nawrocki J. and Wojciechowski A., "Experimental Evaluation of Pair Programming", Proc. of European Software Control and Metrics, 2001.
16. Nosek J.T., "The case for collaborative programming", Communication of ACM, 41(3) ACM, 1998.
17. VanDerGrift T., "Coupling Pair Programming and Writing: Learning About Students' Perceptions and Processes" proc. of SIGCSE, ACM, 2004.
18. Wernick P. and Hall T., ""The impact of Using Pair Programming on System Evolution: a Simulation –Based Study", Proc.of the 2'th Int'l Conference on Software Maintenance (ICSM 2004) 2004, Chicago, Illinois, USA, IEEE CS Press, pp. 422-426.
19. L. Williams, W. Cunningham, R. Jeffries, and R. R. Kessler, "Straightening the case for pair programming", IEEE Software, 17(4), IEEE CS Press, 2000.
20. L. Williams and R. L. UpChurch, "In Support of Student Pair-Programming", Proc. of the thirty-second SIGCSE Technical symposium on Computer Science Education, Charlotte, NC, ACM, 2001.
21. Williams L., McDowell C., Nagappan N., Fernald J., and Werner L., "Building Pair programming Knowledge through a Family of Experiments", Proc. of International Symposium on Experimntal Software Engineering, Rome, Italy, 2003, IEEE CS Press, pp.143-153.
22. Williams L. and Kessler B., "The Effects of "Pair-Pressure" and "Pair-Learning"", Proc. of 13th Conference on Software Engineering Education and Training, Austin, Texas, 2000, IEEE CS Press, pp.59-65.
23. Williams L., Shukla A., and Anton A.I. "An Initial Exploration of the relationship between Pair programming and Brooks law" Agile Development Conference (ADC 04), Salt Lake City, Utah, USA, 2004, IEEE CS Press, pp.11-20.

Appendix

This appendix reports the assignment 1.A used in the experiment.

Assignment 1.A
Develop a water vending machine for the distribution of water's bottles with gas and without gas; the machine has to allow:
R1 to set, to modify, and to display the price of the water with gas and without gas;
R2 release the number of water's bottles of water with and witthout gas, display the total amount to pay and give the exact cash to the user.
R3 In case the request cannot be satisfied completely a message is displayed and the water's bottles are released in the possible number and type.

Name(1)____
Name(2) ____
Name of the pair ____________
Requirements realized: R1(Y|N) R2(Y|N) R3(Y|N)
Times:
Start R1__________ End R1__________
Start R2__________ End R2__________
Start R3__________ End R3__________

The Social Side of Technical Practices

Hugh Robinson and Helen Sharp

Centre for Empirical Studies of Software Development
The Open University, Walton Hall, Milton Keynes MK7 6AA UK
{h.m.robinson,h.c.sharp}@open.ac.uk

Abstract. XP is a social activity as well as a technical activity. The social side of XP is emphasized typically in the values and principles which underlie the technical practices. However, the fieldwork studies we have carried out with mature XP teams have shown that the technical practices themselves are also intensely social: they have social dimensions that arise from and have consequences for the XP approach. In this paper, we report on elements of XP practice that show the social side of several XP practices, including test-first development, simple design, refactoring and on-site customer. We also illustrate the social side of the practices in combination through a thematic view of progress.

1 Introduction

XP is an intensely technical activity, involving practices such as pair programming, continuous integration and test-first programming. It is also an intensely social activity with explicit values, such as communication and respect, and explicit principles, such as humanity and reflection [1]. XP is about social organization: how people organize themselves to develop working software in a manner that is effective in terms of human values as well as in terms of technical and economic values.

The social side of XP is emphasized typically via the values and principles as sustaining, and being sustained by, the practices. Our work on the culture of XP [2] and on the characteristics of XP teams [3] are examples of such an approach. However, the social side of XP is not restricted to the values and principles. There are social interactions involving individuals or groups of individuals which arise from the practices and which have consequences for the practices – the practices themselves have a social side which has significance for XP. In this paper we report on fieldwork studies of mature XP teams that reveal the detail of this social side of technical practices. Some practices (the on-site customer, for example) are more clearly socially-oriented than others, but we also found some surprises (such as refactoring and continuous integration).

The paper has the following structure. First, we outline the details of our fieldwork, the teams studied, and the methodology used. We then report our findings on the social aspects of two elements of XP practice which involve several XP practices. Pairing involves pair programming, refactoring, test-first development and simple design; customer collaboration involves the on-site customer and the planning game. Following this, we consider the social aspects of practices in combination from the thematic view of progress. We conclude by discussing the outcomes and consequences of our findings.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 100–108, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Fieldwork

Our fieldwork studies were of three XP teams, each of which carried out all 12 of the XP practices advocated in the first edition of Beck's book [4]. Each team was studied for a period of a week, with further follow-up meetings to discuss preliminary findings. The approach taken was that of the researcher immersing themselves in the day-to-day business of XP development, documenting practice by a variety of means that included contemporaneous field notes, photographs/sketches of the physical layout, copies of various documents and artefacts, and records of meetings, discussions and informal interviews with practitioners.

Team X were part of a small company developing web-based intelligent advertisements in Java for paying clients. At the time of the study, there were eight developers in the team, one graphic designer and one person who looked after the infrastructure. The company employed four marketing people who determined what was required in collaboration with clients. Marketing took the role of the on-site customer. Iterations were three weeks in length and often included a retrospective.

Team Y were part of a medium-sized company producing software products in C++ to support the use of documents in multi-authored work environments. At the time of the study, there were 23 people working in the team, including three programme managers (who took the role of the on-site customer), two testers, a technical author, a development team coach (who also managed the development team and pair programmed) and 16 developers. Iterations were two weeks in length. Within each iteration, the team organised itself into sub-teams oriented around the various software products or related issues.

Team Z were part of a large financial institution and produced software applications in Java to support the institution's management of operational risk. Whilst the team was a full XP team, the vast majority of the institution's software was originally developed and maintained using conventional plan-driven approaches. The overall team was organised into two sub-teams. At the time of the study, one sub-team comprised 7 developers (one of whom also managed the overall team and the sub-team) and a business analyst. The other sub-team comprised 5 developers (one of whom also managed the sub-team). The role of on-site customer to the overall team was carried out by two individuals with expertise in the institution's approach to the management of operational risk. Iterations were one week in length and each iteration included a retrospective.

Our analysis methodology was ethnographic in approach [5-7]. As far as possible, the natural setting of practice was observed without any form of control, intrusion or experiment. In our analysis, we sought to understand practice in its own terms, minimizing the impact of our own backgrounds, prejudices and assumptions.

3 Pairing

In our fieldwork, pairing involved four of XP’s practices: pair programming, test-first development, simple design and refactoring. In addition, Team Z used pairing to develop acceptance tests. The following observations therefore relate to all of these practices. We consider the social side of pairing from a number of characteristics.

Pairing Is a Conversation

Perhaps the most striking thing about the observed pairing was that it was not so much about code as about conversation. The two programmers involved were engaged in talk – purposeful talk where they discuss, explore, negotiate and progress the task in hand. This talk had a complex structure with identifiable episodes of exploration, creation, fixing & refining, overlaid with explaining, justifying & scrutinising. Much effort was expended on understanding code and it was demonstrably important that this understanding was shared between the pair.

Talk could be almost continuous with no significant periods of silence or it could be more infrequent with noticeable periods of silence. However, these periods of silence were part of the talk at that point in the conversation. They were natural (silence was expected) and in no sense indicated that the conversation was flagging: code was being run through a series of tests, an unexpected red bar had been encountered, or a question had been posed that demanded thought and attention to the screen.

There was a 'silent' partner in this talk: the code and its various manifestations in terms of the panes and windows on the screen that occupied the gaze of the two programmers. Their talk oriented to this partner as did their actions; sometimes rapidly summoning and dismissing panes, sometimes giving their detailed attention to what the code was demanding of them.

And all of this was a shared process. Typically, no one individual was in charge of the talk, no individual monopolised the keyboard or mouse.

These conversations between the two programmers in the pair (with the silent partner of the code) often became conversations involving other pairs, as part of a conversation was overheard and other individuals shared their knowledge and understanding to the task in hand. This was an accepted feature in all the teams and the layout of the pair programming areas allowed pairs to overhear discussions in another pair. The importance of peripheral awareness has been reported elsewhere [8], and there were many examples during our studies where one pair overheard another pair and joined in the conversation. This was not simple curiosity, but led to a sharing of experience and a pooling of ideas to resolve issues and bolster shared undertanding, thus illustrating the importance of these interactions.

Pairing Is Intense and Stressful

In the teams we observed, pairing was an intense and demanding activity, as team members acknowledged in their actions. For example, Team X and one of the subteams from Team Z took regular communal breaks away from the development area. Individuals in Team X would take it upon themselves to remind pairs to take a break from the intensity of pairing and the team coach in Team Y actively monitored intense bouts of pairing and encouraged pairs to go off for breaks. For all three teams, there was a rhythm to pairing during the day that acknowledged this intensity. The start of the day would be marked by activities that didn't require pair programming (personal email, discussions with the on-site customer, stand-up, etc.) and pairing would only begin about two to three hours before lunch (with a break mid-morning). A similar pattern would occur after lunch, so that pairing typically did not take up much in excess of five to six hours in the day. Indeed, Team Z reported that when

they attempted to pair routinely beyond that amount of time they had found it too stressful to sustain. Team X recognised the demands of pairing with a policy of not integrating code into the main system after about 5pm because the biggest problems have been caused by releases at that time of day. In addition, they allowed developers two days per month (known as 'gold card' days [9]) to carry out some individually-focussed work that was of value to the company.

This issue of the stress and intensity of pairing was also addressed in the organisation of space. Both Teams X and Y had a pair programming area as well as an area for activities that didn't involve pair programming, including 'personal' areas for email and 'phone use. Team Y dedicated significant space & facilities for *ad hoc* meetings. This did not always work as intended. For example, Team Y's 'personal' areas were rarely used and the vast majority of *ad hoc* meetings took place within the pair programming area. Team Z were constrained in their available space and had adapted the plan-driven 'developer sitting at a machine' set-up for XP use in an effective fashion, albeit with some frustration.

Pairing Involves a Variety of Styles

Pairs showed a variety of styles in their pairing. For example, an experienced programmer might be paired with a less-experienced colleague so that the experienced programmer could gain familiarity with portions of the code base that the less-experienced colleague had recently worked on. A developer whose approach is one of active exploration might be paired with a colleague whose instincts lead to a more reflective approach. Two developers with similar styles might be paired. Two developers who do not particularly like each other, for whatever reason, might be paired. All these possibilities bring extra demands to the social interaction that is pair programming and require a level of maturity and social management from the participants. This was recognised by our teams in a variety of ways. The development coach for Team Y monitored and adjusted pairing to ensure active and effective engagement. Team Z likened the pairing relationship to that of marriage and strove to display all the skills of compromise, sensitivity, negotiation and reflection that this required. Team X took the overall social health of the team very seriously, giving developers breaks from pairing and, on one occasion, making use of a qualified social worker [9].

Pairing Is Situated

Like all social interactions, pairing does not exist in isolation. Pairing exists, and makes sense, in the context of what has gone before: the creation of stories by the on-site customer, the estimating of those stories by developers, etc. Pairing involves several practices: pair programming, test-first programming, refactoring, and simple design. It also has consequences for other practices, including collective ownership, and the planning game. Pairing exists, and makes sense, in the context of what will happen next: the continuous integration of code and the next release. It is a situated action, in the sense of Suchman [10], and the social interactions of pairing depend upon and reinforce this situated nature.

4 Customer Collaboration

Customer collaboration inevitably revolves around interactions between individuals and between groups. Customer collaboration focuses mainly on the practices of on-site customer and the planning game. Here we concentrate on the former as being the key customer collaboration practice.

In an ideal XP world, the people filling the on-site customer role would be co-located with the developers; would 'speak with one voice'; would be potential users of the system; and would be collaborative, representative, authorised, committed and knowledgeable. Such an ideal on-site customer would bring social challenges to an XP team. But, of course, this ideal is rarely realised: client organisations may be unwilling or unable to spare people to become part of the development team; different customers may have conflicting requirements; potential users of the system may not have the authority to make decisions concerning the identification and prioritisation of system features, whereas decision makers may not understand the needs of users, and so on. XP practitioners have recognised this fact and devised approaches and methods to deal with the gaps between the ideal and the reality (see [11-13], for example). Our teams also had such strategies, and we explore these below, highlighting the social aspects and their consequences.

Team X had perhaps the closest to the on-site customer with marketing personnel who dealt directly with individual paying clients on a regular basis. This direct involvement with the client brought great clarity and authority to the generation of stories and to practices such as the Planning Game. However, the role of marketing personnel demanded that they respond quickly (minutes rather than hours) to requests from clients. Usually, such requests necessitated consultation (and hence considerable interaction) with developers. Much as the developers valued customer collaboration, as well as recognising that keeping the company's clients happy was a pragmatic necessity, the frequency of such interruptions proved too distracting given the intense nature of pairing. The solution explored was that of an 'exposed pair': each day a pair of developers was identified who could be interrupted if a client had an urgent request. Such a solution, of course, depended on the shared understanding and responsibility created by other XP practices such as pairing and collective ownership.

Team Y was somewhat different. The on-site customer role was carried out by programme managers who worked with marketing but were firmly part of the development team. As such, they understood both the market requirements and positioning of the company's various products and the needs of the software development that would create those products. Programme managers organised a considerable amount of the detail of software development and ran the daily stand-ups, as well as orchestrating and managing requests from marketing. They therefore managed a complex set of interactions between various groups and individuals.

It was noticeable that pairing was more 'interruptible' here: *ad hoc* discussions involving pairs and a programme manager would naturally occur and often would involve individuals from another pair, or testers, or the team coach. Once the particular issue was resolved, pairing would resume and there was no sense that what had occurred was an 'interruption'. Of course, the development team manager had an overarching role in this organisation of work but he also had time to devote to his dual role of development team coach, monitoring and managing interaction between relevant parties, the health and vitality of the XP team, and its practice.

Team Z had as the on-site customer two individuals with expertise in the institution's methodology for the management of operational risk. Operational risk at this statutory level (see http://www.fsa.gov.uk/pubs/press/2003/078.html, for example) is still a relatively new area, requiring an institutional 'methodology' for the management of operational risk. The details of such a methodology take time to emerge. As a consequence, stories (requirements) were often subject to a degree of change. In addition, whilst the on-site customer had expertise in the approach to the management of operational risk, they were not the intended user of the various applications. Furthermore, the institution had a strong tradition of conventional, plan-driven software development with all its expectations of how sponsors, stakeholders and users interact with software developers. Importantly, the on-site customer had significant responsibility for the overall success of the applications under development. The two individuals carrying out the on-site customer role were not co-located with the developers (unlike the case with both Team X and Team Y).

All of these factors made the role of on-site customer particularly demanding both for the individuals carrying out the role and for the developers. Both developers and customers worked actively to manage the relationship and overcome problems, and reported positively on this at a retrospective. Developers proactively involved the customer at a range of opportunities, from the Planning Game, to seeking them out after a stand-up, through to ensuring that the customer came with the team for a coffee break. Considerable effort was expended in developing a shared understanding of the risk methodology via *ad hoc* meetings. However, the relationship was demanding and had consequences for other practices. For example, whilst the 40-hour week practice was part of the XP team's approach, there was some evidence of pressure on the developers being exerted through interactions with the customer, in light of the institution's overall approach to software development. Whilst this had the obvious consequence of additional hours, it also worried the developers in terms of the quality of code being produced and the issue of 'more means less' was actively discussed by one sub-team and raised at a retrospective.

5 Progress

We now consider the social side of XP's practices in combination from the viewpoint of the theme of progress. This theme involves a variety of interactions and the full range of practices. XP is about making progress with software development – it stands in opposition to 'death march' projects where making progress is problematic. XP addresses the issue of progress through the combination of all 12 practices. The orchestration of these practices by an XP team to make progress is a powerful social achievement and operates in a number of ways.

XP life has a rhythm that operates at a range of levels and we concentrate on two that emerge from our fieldwork: one was a daily rhythm and one was a rhythm oriented around the iteration. These rhythms were marked by events that signalled openings and closings and were punctuated by other events which signalled progress.

The daily rhythm for all three teams was remarkably similar. The day would begin as people arrived for work and engaged in activities that did not require pair programming (as we have discussed above in Section 2). The daily stand-up was a significant event in this rhythm which discussed 'what we did yesterday, what we'll do

today' and briefly shared understanding and issues. Pairing, and its conversation within a pair and between pairs, typically began following the stand-up, punctuated by breaks and lunch. Stories would be successfully completed and code would be released (the continuous integration practice).

Team X marked integration in a public fashion. The machine used for integration had on top of it a small box with a picture of a cow on it and the box would produce a 'moo' sound when picked up and tilted. Each time a developer released code, he/she would pick up the box and make it 'moo', thus communicating to others that progress had been made. This illustrates a more indirect form of interaction. Furthermore, as already noted, the team had a policy of not carrying out integration after about 5pm: a recognition of the social side of pair programming and its consequences for continuous integration and meaningful progress.

Team X had a notice board dominated by a large organised space devoted to the active story cards and four 'to do' lists. Completion of a story would be reflected in a change to the board: the board 'kept the score'. Each developer had a responsibility, to themselves and to others, to check progress, to maintain progress, and to shout if progress was not happening. This indirect interaction between parties via a board of some sort has been noticed elsewhere in different settings such as that of patient care activity by nurses in ward settings [e.g. 14, Chapter 5].

Team Z adopted a similar approach with a notice board but were constrained by physical space and other factors outside their control in the amount of information that could be displayed. The institutional culture was also a factor here: jokes were made that displaying progress also displays what is still outstanding.

In contrast, Team Y used a custom-built computer documentation tool to record, communicate and progress stories, rather than index cards. Whilst this tool was a significant medium of communication in that it captured the detail of each story: the estimate, a brief description, the customer acceptance test, who was working on it, progress through development, integration, pre-quality assurance testing, etc. it did not easily facilitate a public display of progress. However, the completion of a story and the integration of its code was marked by an event – the signalling to one of the two testers that code was now ready for acceptance testing.

This daily rhythm concluded with the end of the day – a move over a short period of time from a full office to an empty one. Importantly, there was no sense of frustration at day's end, such as might be associated with missed targets in a rigid schedule.

This daily rhythm was embedded in the rhythm of the iteration, where the full range of practices would be in play, typically involving a retrospective for two of the teams. Within this iteration rhythm, the detail of many of the practices oriented towards the issue of progress in a way which illustrates some of the tensions and strengths of XP practices.

Refactoring is an example of tension. During one pair programming episode with Team X, significant frustration was shown by the developers who wanted to refactor the code, but were working on a story card that did not include an estimate for refactoring. The problem was exacerbated because the code base supported more than one product and refactoring would have involved making decisions about the older product which involved business strategy decisions not just coding ones. The pair reluctantly agreed to write a task card for refactoring this section of the code, and to focus only on the card in front of them. Their disappointment and frustration was palpable.

This suggests that sometimes there could be a tension between pragmatic considerations around progress and the desire to refactor and have the simplest code. Team Z exhibited a similar frustration and raised the issue of accumulated 'technical debt' at both the iteration Planning Game and the retrospective.

An example of illustrating the strengths of XP comes from Team Y. When we questioned a member of one sub-team about progress – about a crucial release date for a software product that was vital to the company's plans the reply was 'Don't know: I'm not working on that.' Rather than indifference, this was an expression of confidence in colleagues. Such confidence is a consequence of the practices: the Planning Game (and its mutually agreed estimates), pair programming (and the acceptance of responsibility and ownership it entails), of test-first programming (and the working code it ensures), of continuous integration, and so on. That is, the practices ensured that each sub-team actively agreed responsibility for work, respected the similar action of the other sub-teams, and made progress.

6 Conclusion

We have shown how the technical practices of XP have social interactions of consequence for the practices themselves, as exemplified by the various approaches taken by the teams we studied. We have made explicit that which is implicit in good XP practice.

We offer no simple rote rules. For example, the utility of pairing as a conversation (including the role of peripheral awareness) cannot be facilitated simply by arrangement of physical space. It requires attention to other aspects of pairing, such as its intensity and the variety of styles. It requires attention also to the relationship with – and the impact of – other practices (such as that of the on-site customer) and will not be realised with a limited set of practices. If a limited set of practices is adopted, then careful thought needs to be given to the interactions discussed here and how to ensure that the advantages of the practices are maintained. Good practice will, however, orient to the issues and factors that we have discussed in a fashion that reflects mastery of the values and principles behind the practices and of the specific nature of the practices in a particular team setting.

References

1. Beck K, Andres C. eXtreme Programming Explained: embrace change. San Francisco: Addison-Wesley, 2005.
2. Robinson HM, Sharp H. XP culture: why the twelve practices both are and are not the most significant thing. Proceedings of the Agile Development Conference. Salt Lake City, Utah, 25-28 June: IEEE Computer Society Press, 2003. pp. 12-21.
3. Robinson HM, Sharp H. The characteristics of XP teams. In: Eckstein J, Baumeister H, editors. Proceedings of the Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004). Garmisch-Partenkirchen, Germany, June 6-10: Springer-Verlag, 2004. pp. 135-47.
4. Beck K. eXtreme Programming Explained: embrace change. San Francisco: Addison-Wesley, 2000.

5. Robinson HM, Segal J, Sharp H. The case for empirical studies of the practice of software development. In: Jedlitschka A, Ciolkowski M, editors. Proceedings of the ESEIW Workshop on Empirical Studies in Software Engineering. Rome Castles, Italy, 29 September, 2003. pp. 98-107.
6. Sharp H, Robinson HM. An ethnographic study of XP practice. Empirical Software Engineering 2004;9 (4):353-75.
7. Sim SE. Evaluating the evidence: lessons from ethnography. Proceedings of the Workshop on Empirical Studies of Software Maintenance. Oxford, England, 1999. pp. 66 - 70.
8. Heath C, Luff P. Collaboration and control: crisis management and multimedia technology in London Underground line control rooms. Proceedings of CSCW'92, 1992. pp. 69 - 94.
9. Mackinnon T. XP - call in the social workers. In: Marchesi M, Succi G, editors. Proceedings of the Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2003). Berlin: Springer, 2003. pp. 288-97.
10. Suchman LA. Plans and Situated Actions. Cambridge: Cambridge University Press, 1987.
11. Martin A, Noble J, Biddle R. Being Jane Malkovitch: a look into the world of an XP customer. In: Succi G, editor. Proceedings of the Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2003). Genoa, Italy, May 25-29: Springer-Verlag, 2003.
12. Martin A, Biddle R, Noble J. The XP customer role in practice: three case studies. Proceedings of the Second Agile Development Conference. Salt Lake City, Utah, June 22-26, 2004.
13. Sharp H, Robinson HM, Segal J. eXtreme Programming and User-Centered Design: friend or foe? Proceedings of HCI2004. Leeds, 2004.
14. Davis H. The social management of computing artefacts in nursing work: an ethnographic account. PhD thesis, University of Sheffield, 2001. pp. 289.

A Survey of Test Notations and Tools for Customer Testing

Adam Geras[1], James Miller[2], Michael Smith[1], and James Love[1]

[1] University of Calgary, Calgary, Alberta, Canada T2N 1N4
{ageras,smithmr}@ucalgary.ca
[2] University of Alberta, Edmonton, Alberta, Canada T6G 2E1
jm@ee.ualberta.ca

Abstract. The term 'customer testing' typically refers to the functional, or correctness testing of software-intensive systems. The tests are typically described and automated in a test-first manner, that is, they exist before the target system is built. This is generally believed to have improved the testability, and perhaps the overall quality, of the systems under test. With the increasing utility and functionality of mobile and pervasive computing systems, we speculate that the need to include non-functional test cases (performance, security, usability, etc.) under the 'test-first' umbrella will increase. In this paper, we review the capability of existing test notations and tools to describe and execute, in a test-first style, non-functional test cases. This concept challenges the default agile position of delegating non-functional tests cases to traditional, test-last test techniques.

1 Introduction

The term 'customer testing' is commonly used in agile development projects and maps loosely to system and acceptance testing in traditional testing terminology [1]. In agile projects, customer tests are those test cases that reflect the end-user perspective, so they are usually black-box test cases oriented to finding defects in the overall system behavior. In contrast, 'developer tests' are those tests that reflect the developer perspective and typically map to the more detailed test levels in traditional testing terminology.

We have focused on customer testing in this study since results from early empirical studies on agile methods imply that the use of customer tests is critical to the success of test-driven development [2, 3]. In other words, the benefits of using both customer and developer tests in a test-first manner exceeds the benefits of using only developer tests test-first. The agile perspective is typically that non-functional tests are not necessarily candidates for test-first and that they might best be managed by traditional, test-last techniques [4]. We speculate that developers working on pervasive computing systems will challenge the view that customer testing is limited to functional or correctness testing. These applications come with limited functionality but an extensive set of non-functional requirements. To that end, non-functional test cases play a large role in the development of these systems, and it might make sense to look at those requirements from a test-first perspective as well [5]. Therefore, in this document, we survey various notations and tools for supporting 'non-functional customer testing' in agile software development projects.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 109–117, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The testing mechanisms and tools for developer testing are well-known and epitomized by the use of the xUnit testing frameworks. The issue that we wish to investigate here is that the mechanisms and tools question at the customer testing level is NOT well-known and it is probably NOT best represented by any single tool, notation, or automation idiom. In this paper, we will identify criteria for comparing the various types of customer testing notations and frameworks that are available; and conduct a brief qualitative analysis using those criteria. Our intent is to follow this current qualitative work with a quantitative analysis; once we have engaged more practitioners and are in a better position to precisely define what the objectives of such research might be. For now, we are content to initiate the discussion of the requirements of a customer testing notation and automation framework.

Our current analysis focus includes both *functional* and *non-functional* tests because of the probable overlap between the two types of tests. That is, non-functional tests typically re-use aspects of the functional tests in order to maximize their meaningfulness in the context of the system under test, SUT. Functional tests target user expectations within the domain that the system under development, SUD, or the SUT, will operate within. A non-functional test has an altogether different target – something such as performance, security, or usability objectives governing the SUT [6].

2 Survey Methodology

2.1 Notations/Tools Surveyed

It is important to recognize that a single commercial or open source tool may be represented in more than one of the following categories.

'Record-Playback Scripting Notations/Tools' describes the set of tools that are based on using a 'recorder' of some kind to write test scripts. The resulting output of the recorder can be saved and run later as required. Some tools generate scripts in a proprietary scripting language, while others generate scripts in a standard scripting language. Some tools generate keyword-driven scripts and store them in simple text formats, such as CSV.

The **'Standard Scripting Notations/Tools'** category represents the tools that describe a test case in a standard scripting language, such as VBScript, Ruby, Python, PHP, shell script, etc. These 'tools' may include function or class libraries oriented to testing. They may also be more elaborate and include the facilities for manipulating user interface elements for testing directly against a user interface.

'TTCN-3' is a test notation that was initially devised to support telecommunication testing [7]. For that reason, it provides excellent support for protocol testing and other communications-related testing tasks. It differs from the standard scripting languages in that the core language specification includes constructs oriented to testing, including verdict determination and reporting. We've included it because although we are certain that it was not designed for customer testing, we're less certain that it couldn't be used for such purposes. TTCN-3 has two formalized formats for describing test cases, a text-based format and a graphical format. We have not employed TTCN-3 on a real project in order to complete this analysis.

The **'XML-based Notations and Tools**' are tools that permit the test script authors to describe their test cases in an XML document. A test runner then parses the XML

document and treats its various elements as directives to run against the system under test. XML differs greatly from standard scripting languages even though these tools tend to use it in the same manner. For this reason, we've made it a separate category. Initiatives like the Automated Test Markup Language (ATML) project are included in this category.

'UML Testing Profile' represents the standardization of the mechanism for describing tests using the Unified Modeling Language. In order for the test cases described in this manner to be executed, a language mapping must also be employed. There are language mappings available for JUnit and for TTCN-3. We have not used the UML Testing Profile in a real-life software project in order to complete this analysis.

'FIT' represents the use of tools based on Ward Cunningham's FIT tool [8]. There are several variations available that support both data-driven testing and keyword-driven testing. This is perhaps the only category containing tools that were purposefully designed with automated customer testing in mind.

2.2 Research Setting

Being a qualitative study, our aim is to ask closed questions in each of 3 categories – 'ease of use', 'support for test-first use', and 'support for describing non-functional tests'. We conducted the survey by creating a reference functional test case in plain language, and then translating that into each of the specified notations. The system under test was a database-backed web application, and the test case itself was a "day in the life" type of a high-level (acceptance or functional) test case. For the record/playback tools, we relied on our industrial experience to answer the questions in lieu of creating a specific translation using one of the tools. Once the reference test case was translated, we were then able to inspect and assess the notations from a non-technical user's perspective to answer the ease-of-use questions and from a test-driven developer's perspective to answer the test-first questions. The questions were answered by scoring the notation with 2 (fully supported) 1 (somewhat supported), or.0 (no support for the specified criterion).

To answer the questions in the non-functional test case group, we assessed the test case in each notation by describing what would have to be done in order to adorn the test case so that it could target the specific non-functional objectives. For performance testing, for example, we assessed the work that would be required to cause the test to pass or fail depending on the resources it uses – CPU resources, memory resources, or elapsed time. It is important to recognize that the work to verify our assessments is not completed – we're not yet ready to conduct evaluations of the notations. At this time, we're only seeking direction and guidance for future research into automating non-functional tests.

3 Analysis

3.1 Ease of Use

In the ease-of-use category, we considered the importance of bypass-testing in ensuring that back-end functionality is correct with having to deal with the complications of the front-end. We also considered that both data-driven test cases and keyword-

driven test cases were useful for higher-level tests since customers can relate to them relatively easily.

Q1. Can the Test Cases Employ an API if it Exists and Bypass the User Interface?

Rec-Pl: 0	Script: 2	TTCN: 2	XML: 2	UMLTP: 2	FIT: 2

The record/playback tools were the only tools that were not able to employ the back-end API directly (unless the tool was used in scripting mode). This means that any notation except a record/playback tool would be appropriate if the test target is the back-end business functionality and not the user interface.

Q2. Can the Test Cases Employ the Same Interface that the Users Will Use?

Rec-Pl: 2	Script: 2	TTCN: 0	XML: 2	UMLTP: 0	FIT: 0

TTCN-3, the UML Test Profile, and FIT are not designed to work against an existing user interface. For this reason, if the test target is the integration of the front-end to the back-end, then an alternate notation should be used.

Q3. Is There Support for Storing Test Input and Expected Output Data Separate from the Test Execution Facility, that Is, to Support Data-Driven Testing?

Rec-Pl: 2	Script: 2	TTCN: 2	XML: 2	UMLTP: 2	FIT: 2

All of the tools can run test cases in a data-driven manner. Note that because of its table orientation, FIT is particularly effective for running the same test repeatedly with different inputs. Depending on the problem that you are trying to solve, the physical appearance of the various data values one after the other is an excellent overview of the test cases that some of the other notations are not able to provide.

Q4. Is There Support for Storing Test Input Data, Expected Output Data, and Test Directives Separate from the Test Execution Facility, i.e. Keyword-Driven Testing?

Rec-Pl: 0	Script: 1	TTCN: 1	XML: 1	UMLTP: 1	FIT: 2

Domain-specific keywords that represent application actions can be implemented with standard scripting languages since they all provide language constructs for building reusable components and methods. Considerable effort would have to be made to make the names of these reusable components consistent with application actions that the customer/end-user tester could relate to. TTCN-3 test cases could be managed in a similar manner. The XML-based notations can be extended to enable custom tags in the XML documents that represent application actions that the user can relate to at the price of a tag handler in the test runner. This isn't more effort than the effort required to build a DoFixture in FIT, but we assessed this as a yes/no since many of the XML-based notations we looked at were oriented to testing through the graphical user interface, and using the notation in "bypass GUI" mode would have been a considerable change from its intended use.

Q5. Is There Built-In Support for Recovering from a Test Failure, or Would One Failing Test Interrupt the Test Run? Is there a Built-In Facility for Determining Verdicts and Publishing Them?

Rec-Pl: 1	Script: 1	TTCN: 2	XML: 1	UMLTP: 2	FIT: 2

All of the test notations considered had language features or notation for describing how to handle unexpected results. All of the test notations considered had language features or notation that could be used to establish or determine the outcome of tests. Scripting languages were assessed as only partially meeting this requirement since the needed support might be distributed separately and is not part of the core language distribution. This is true of some common scripting languages such as VBScript. More powerful scripting languages such as Ruby and Python have xUnit-style support bundled with the core language run-time.

Ease of Use Category Totals

Rec-Pl: 6	Script: 8	TTCN: 7	XML: 8	UMLTP: 7	FIT: 8	/10

The ease of use category totals indicate that scripting, XML-based notations, or FIT seem to be the tools to choose for flexibility and ease-of-use at the customer test level.

3.2 Support for Test-First Use

In the support-for-test-first category, we considered the need to describe and run the test case before the target system exists. We also considered the complexity of the notation itself from the perspective of non-technical users; and if there were ways to use the notation to help manage the requirements.

Q6. Can the Tests Be Described Before the Target Exists [9]?

Rec-Pl: 0	Script: 2	TTCN: 2	XML: 2	UMLTP: 2	FIT: 2

The record/playback tools are the only ones that cannot be used before the target exists. The rest of the test notations could be used to specify the intended behaviour at the customer test level. In our reference test case of a database-backed web application feature, confirming that the database changed state appropriately requires additional work so that the test case (and not the system under test) is the oracle.

Q7. Can the Tests Be Run Before the Target Exists?

Rec-Pl: 0	Script: 2	TTCN: 2	XML: 2	UMLTP: 0	FIT: 2

Agile testing means that running the test case shows a FAILURE response instead of an ERROR response. Again record/playback tools are unable to comply with this requirement. Running the UMLTP test cases is also a bit of a stretch since it requires a language mapping, much further away from the direct execution like FIT or scripting languages have. Even to run FIT tests there is some work required since the back-end test fixtures need to exist in order to avoid an ERROR response.

Q8. Is There a Non-technical Artifact for Describing Test Cases? Is There an Editing Tool that End-User/Customer Testers Could Use?

Rec-Pl: 2	Script: 0	TTCN: 0	XML: 1	UMLTP: 1	FIT: 2

Tests should be readable by both technical and non-technical team members since the tests document requirements and describe intended functionality [10]. Scripting and TTCN-3 were assessed as being the furthest away from the ideal answer to these questions. We also assessed record/playback tools as being favourable to non-technical users since creating the test cases is akin to using the target application.

Still, the expectations of the target behaviour are less clear in record/playback tools unless you peek at the underlying script. The DoFixture in FIT once again was assessed as being highly amenable to non-technical users.

Q9. Does the Notation Provide for Requirements Coverage and Traceability?

Rec-Pl: 1	Script: 1	TTCN: 1	XML: 1	UMLTP: 1	FIT: 1

Test creators should be able to associate the test case to a requirement. This enables coverage analysis that ensures all requirements have an adequate number of test cases. It also enables determining the requirements or features that are affected if a test case is failing, that is, traceability. The UML package notation can be used to group the test cases accordingly. For the rest of the notations, a workaround in comments or storage location would approximate satisfying the requirement, but not completely satisfy it. There are open source and commercial tools for managing agile projects that associate user stories (requirements) to test cases but the association is not part of the test case notation itself. Some of the XML notations that we reviewed included tags for identifying the requirement that the test case was intended to support; this is a step in the right direction.

Test-First Use Category Totals

Rec-Pl: 3	Script: 5	TTCN: 5	XML: 6	UMLTP: 5	FIT: 7	/8

The totals for each of the tools in the suitability for test-first use category indicate that XML-based notations, or FIT seem to be work best for test-driven work at the customer test level.

3.3 Support for Describing Non-functional Tests

In this section, we will describe our assessments of the various notations for describing non-functional tests; that is, tests for determining the acceptability of the application from performance, security, usability, etc. perspectives. Assessing the UML Test Profile was particularly difficult in this category because it can be used to express all types of test cases, but requires a mapping to something else before the test case might become executable.

Q10. Is There Support for Measuring the Run-Time Performance of the Target Such as CPU, Memory Usage?

Rec-Pl: 0	Script: 1	TTCN: 1	XML: 1	UMLTP: 1	FIT: 1

All of the notations under consideration can only partially describe performance-oriented test cases, but none permit the test developer to describe resource consumption targets as maximum levels that would cause the test to fail if the maximum was exceeded. Without further evaluating any of the tools specifically for this purpose, we suggest that teams adopt developer testing tools to automate test cases that fail on excessive resource consumption (as opposed to customer testing tools). The extensibility of FIT and/or the XML-based tools may enable us to create a custom fixture that contains resource constraints as part of the test. The XML tools and FIT were assessed as somewhat satisfying performance testing requirements because of the ease of adding attributes to test cases, at least in the FIT implementations where annotations (meta data) can be added to test cases (.NET and Java).

Q11. Is There Support for Simulating Heavy Usage Volumes?

Rec-Pl: 2	Script: 1	TTCN: 2	XML: 2	UMLTP: 1	FIT: 0

As with resource metering, the support for load/stress testing in the existing tools mostly include simulating the load and then generating a report, leaving the test outcome to be decided by an attendant. The ideal answer would be again to support automated verdicts related to the desired performance under certain loads. None of the notations that we surveyed included the verdict automation that we envisioned, although the XML-based tools and the record/playback tools can simulate the required number of simultaneous users and transactions and TTCN has been used for quality of service testing under noisy conditions, which is analogous to heavy usage. There seems to be more tasks required to customize FIT tests for this purpose than any of the other notations.

Q12. Support for Security Testing such as Running Scenarios Under User Personas? For Checking Other Vulnerabilities?

Rec-Pl: 0	Script: 1	TTCN: 1	XML: 1	UMLTP: 1	FIT: 1

Some aspects of security testing could be integrated into the test runners for all of the notations. All of the tools, for example, could be coerced into running the various test cases under the auspices of different user roles. Similarly, analysis tools such as buffer overflow checkers can probably be integrated into any of the solutions. Automated verdict determination would still have to be built from scratch for most of these types of test cases. FIT and XML were again assessed as somewhat satisfying the requirements for security testing because of the relative ease of adding meta data that would prescribe the security context for running the test (for example).

Q13. Is There Support for Describing and Running Tests that Evaluate Usability of the Target Application (Usability Testing)?

Rec-Pl: 1	Script: 1	TTCN: 0	XML: 1	UMLTP: 0	FIT: 0

As with security testing, some aspects of usability testing can be implemented in any of the notations, especially test targets such as accessibility, internationalization, or user interface standard compliance. Other test targets related to usability are probably best left to manual testing. FIT is designed for writing test cases that bypass the user interface, so it is unlikely to be useful out-of-the-box for usability testing even if a custom FIT fixture could be devised to target usability concerns.

Q14. Is There Support for Describing and Running Tests that Evaluate the Ability of the SUT to Recover from Disaster (Recovery Testing)?

Rec-Pl: 0	Script: 1	TTCN: 0	XML: 0	UMLTP: 0	FIT: 0

None of the tools contain specific syntax for dealing with recovery testing, although most of them could be coerced into turning on and off certain services that the application requires. Using a scripting language, for example, it is relatively easy to turn off an underlying database service, run a test case, and then turn the database service back on again, especially through standardized instrumentation API's such as the Common Information Model (CIM) an aspect of the Web-Based Enterprise Management industry initiative. So for at least some of the targets of recovery testing, scripting seems to be the first choice from those analyzed.

Q15. Is There Support for Configurability/Interoperability Testing?

Rec-Pl: 0	Script: 1	TTCN: 2	XML: 0	UMLTP: 1	FIT: 0

Only TTCN supports interoperability testing (conformance testing of communications protocols is one of its primary purposes) directly. To a lesser extent, scripting could be used to build tests of this type.

Non-functional Testing Category Totals

Rec-Pl: 3	Script: 6	TTCN: 6	XML: 5	UMLTP: 4	FIT: 2	/12

The totals for each of the tools in the suitability for non-functional testing category indicate that TTCN, XML-based notations, or scripting seem to show the most promise for non-functional testing work at the customer test level,. Keep in mind, however, that many commercial record-playback tools also support scripting; making them good candidates for this work.

The common thread in the preceding analysis is that none of the notations fully support non-functional testing. After inspecting the various artifacts that generate task lists of what we would have to do to make these tests fully automated, we have come to the tentative conclusion that adorning functional cases with non-functional test attributes is the research direction worth exploring. This would enable customer test developers to describe functional test cases that run under certain other constraints that could also cause the test to fail. In our application of FIT for high-level testing of an embedded digital signal processing system, for example, we could run a functional test and place a constraint on that test run that it does not consume more than 25 kb of memory. At this time, we're thinking that this will require a custom FIT fixture for adorning the functional test with additional test conditions at a minimum.

4 Conclusion

This qualitative analysis suggests that using FIT, a scripting language, or an XML-based notation/tool will optimize the effectiveness of a test-driven development project; and using those tools to build keyword-driven test cases might be one of the most effective ways to run customer testing today. To expand the set of customer tests to include non-functional requirements, then scripting languages, TTCN-3, or XML-based notations seem to be the easiest to work with, at least based on this cursory analysis.

Agile testing means departing from the traditional separation of describing test cases from running them, and this places severe demands on the test notation used in a project. At the same time, agile projects cannot rely exclusively on functional or correctness testing and as we get increasingly familiar with the benefits of test-driven development, the impetus for including non-functional test cases in the test-first umbrella is correspondingly increasing as well, especially in application development domains where there are a large number of non-functional requirements. Further work is required to expose the limitations of thinking that only functional test cases should be done test-first and to further refine the requirements for an all-inclusive, test-first, test notation.

References

1. Beck, K., *Extreme Programming Explained*. 1999, Don Mills: Addison-Wesley Publishing Co.
2. Geras, A., M. Smith, and J. Miller. *A Prototype Empirical Evaluation of Test Driven Development*. in *10th International Software Metrics Symposium*. 2004. Chicago: IEEE Computer Society.
3. Pancur, M., et al. *Towards empirical evaluation of test-driven development in a university environment*. in *EUROCON 2003. Computer as a Tool. The IEEE Region 8*. 2003: IEEE Press.
4. Marick, B., *Roadmap for Agile Testing*, in *Agile Times Newsletter #5*. 2004, Agile Alliance.
5. Beznosov, K. and P. Kruchten. *Towards Agile Security Assurance*. in *New Security Paradigms Workshop*. 2004. White Point Beach Resort, Nova Scotia, Canada.
6. Bourque, P. and R. Dupuis, *Guide to the Software Engineering Body of Knowledge*. 2004: IEEE Computer Society.
7. ETSI, *Part 1: TTCN-3 Core Language*. 2003, European Telecommunications Standards Institute.
8. Cunningham, W., *Framework for Integrated Test*. 2002, Cunningham & Cunningham, Inc.
9. Crispin, L., *eXtreme Rules of the Road*, in *STQE*. 2001. p. 24-29.
10. Mugridge, R., B. MacDonald, and P. Roop. *A Customer Test Generator for Web-Based Systems*. in *4th International Conference on eXtreme Programming and Agile Processes in Software Engineering*. 2003. Genova, Italy: Spinger-Verlag Heidelberg.

Testing with Guarantees and the Failure of Regression Testing in eXtreme Programming

Anthony J.H. Simons

Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
A.Simons@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk/~ajhs/

Abstract. The eXtreme Programming (XP) method eschews all formal design, but compensates for this by rigorous unit testing. Test-sets, which constitute the only enduring specification, are intuitively developed and so may not be complete. This paper presents a method for generating complete unit test-sets for objects, based on simple finite state machines. Using this method, it is possible to prove that saved regression test-sets do not provide the expected guarantees of correctness when applied to modified or extended objects. Such objects, which pass the saved tests, may yet contain introduced faults. This puts the whole practice of regression testing in XP into question. To obtain the same level of guarantee, tests must be regenerated from scratch for the extended object. A notion of guaranteed, repeatable quality after testing is defined.

1 Introduction

The popular eXtreme Programming (XP) method throws away the formal analysis and design stages of conventional software engineering [1, 2, 3] in reaction to the high overhead of standard development processes and the extra documentation that these require [4, 5]. However, the lack of formal design in XP has received some constructive criticism [6]: its "extremeness" may be characterized by this and the fact that the software base is subject to constant modification during the lifetime of the project, with all developers granted the freedom to change any part of the software.

To guard against inadvertently damaging the code base, a great emphasis is placed upon *testing*. Unit tests are developed, sometimes directly from the requirements (*test-first design*), but mostly in parallel with the code (*test-driven development*). Tools supporting the unit testing approach have been developed, such as JUnit, a popular tool for unit testing single classes [7, 8]. Every time the object under test (OUT) is changed, the software must be exercised again with all the existing test-sets, to ensure that no faults have been introduced (known as *regression testing*). This, with the emphasis on regular builds in small increments, is intended to guard against the effects of entropy. (XP also advocates *acceptance testing*, whereby the system is evaluated against user requirements prior to delivery; this aspect is not under investigation in the current paper).

1.1 Parallel Design and Test

Practical object-oriented unit testing has been influenced considerably by the *non-intrusive* testing philosophy of McGregor *et al* [9, 10]. In this approach, every OUT

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 118–126, 2005.
© Springer-Verlag Berlin Heidelberg 2005

has a corresponding test-harness object (THO), which encapsulates all the test-sets separately, but may have privileged access to observe the internal states of the OUT (e.g. via *friend* declarations in C++). The OUT is therefore uncluttered by test code and may be delivered as-is, after testing. This separation of concerns lies behind the *parallel design and test architecture* (see figure 1), in which an isomorphic inheritance graph of test harness classes shadows the graph of production classes [10].

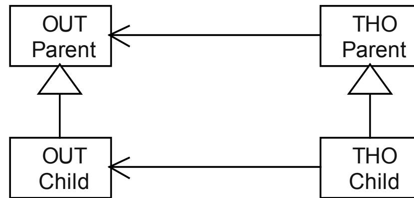


Fig. 1. McGregor's software architecture for parallel design and test abstractions. Motivated by the separation of concerns between production and test code, this leads to the notion of inheritable test-sets, which are reused when testing refined classes

From this arises a beguiling notion: since a child class is an extension of its parent, so the test-sets for the child must be simple extensions of the test-sets for the parent. The presumed advantage is that most test-sets can be *inherited* from the parent THO and reapplied, *as a suite*, to the child OUT, in a kind of regression test. The child THO is understood to supply additional test-sets to exercise the new methods introduced in the child OUT [9, 10]. Below, we show just how unreliable this intuition is.

1.2 Recycling Unit Test Sets in XP

The JUnit tool fosters a similar strategy for re-testing classes that are subject to continuous *refactoring* (internal reorganization yielding the same external behaviour) and *subclassing* (extension without invalidating old behaviour) [7, 8]. JUnit allows programmers to develop and save *test suites*, which may be executed repeatedly. One of the key benefits of JUnit is that it makes the re-testing of refactored and subclassed objects semi-automatic, so it is widely used in the XP community, in which the recycling of old test-sets has become a major part of the quality assurance strategy.

XP claims that a programmer may incrementally modify code in iterative cycles, so long as each modification passes all the original unit tests: "Unit tests enable refactoring as well. After each small change, the unit tests can verify that a change in structure did not introduce a change in functionality" [11]. There are two sides to this claim. Firstly, if the modified code *fails* any tests, it is clear that faults have been introduced (tests as diagnostics). Secondly, there is the assumption that modified code which *passes* all the tests is still as secure as the original code (tests as guarantees). This assumption is unsound and unsafe. After passing the recycled tests, objects may yet contain faults introduced by the refinement. The guarantee provided by passing the recycled tests is strictly weaker than the original guarantee that the same test-set provided in its original context. Using simple state-based models of objects, it is possible to show why recycled test-sets provide incommensurate, and therefore inadequate, test coverage in the new context.

2 Generating Complete Unit Test-Sets

State machines have frequently been used as models of objects, both to articulate their design [12, 5], and more formally as part of a testing method [13, 14, 15, 16]. In the more rigorous approaches, it is possible to develop a notion of *complete test coverage*, based on the exhaustive exploration of the object's states and transitions. The following is an adaptation of the X-Machine testing method [16, 17], which offers stronger guarantees than other methods, in that its testing assumptions are clear and it tests *negatively* for the absence of all undesired behaviour as well as *positively* for the presence of all desired behaviour. It is well known that programmers fail to consider the former, since it is hard to think of all the unintended ways in which a system might possibly be abused, when the focus is on positive outcomes. No matter how simple and direct the hand-crafted tests might appear, if they don't cover the object's state space, they are *incomplete*, and only offer a false sense of security.

Earlier companion work showed how easy and practical it is to generate simple state machines directly from XP story cards [18] and UML use cases [19]. This supported the automatic generation of complete *user-acceptance tests*. In [20] the automatic generation of complete *unit tests* from state machines was shown to outperform ad-hoc test scripting. The focus of the current paper is *regression testing* and complete test *regeneration* in the context of object evolution.

2.1 State-Based Design

An object exists in a series of states, which are chosen by the designer to reflect modes in which its methods react differently to the same messages. The number of states an object can have depends on the independently-varying values of all of its attributes. The theoretical maximum is the Cartesian product of its attribute domains. Typically, this fine-grained attribute space is partitioned into coarser states, in which behaviour is qualitatively "the same" in each partition [15, 21]. We define states more abstractly, as partitions of the product of the ranges of an object's access methods [22]. This permits the design of state machines for fully abstract classes and interfaces, which by definition have no concrete attributes.

Figure 2 illustrates a simple state machine, describing the modal behaviour of a *Stack*. This could represent the design of a concrete class, or an abstract interface. Transitions for the state-modifying methods {*push, pop*} are shown explicitly. For completeness, a finite state machine must define a transition for *every* method in *every* state. We assume by convention that the access methods {*top, empty, size*}, which are not shown, all have implicit transitions looping from each state back to itself.

The state machine is developed by considering the modes in which certain methods behave differently. Here, *pop* and *top* are legal in the *Normal* state, but undefined in the *Empty* state. There is a single transition to the initial state (representing object construction). Transitions are then added for *every* method in *every* state. If an object is no longer useable, it may enter a final state (such as the error state after the illegal *pop* from *Empty*). If multiple transitions could fire for the same message request, the designer should resolve this nondeterminism by placing guards, mutually exclusive and exhaustive conditions, on the conflicting transitions (such as on the two *pop* tran-

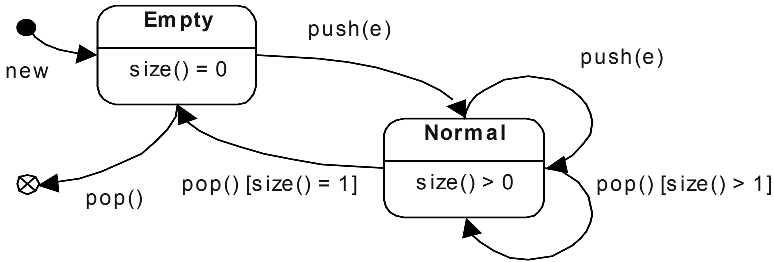


Fig. 2. State machine for a *Stack*. The two states (*Empty, Normal*) are defined on a partition of the range of the *size* access method. Only state-modifying transitions are shown explicitly

sitions from *Normal*). Certain design-for-test conditions may apply, to ensure that the machine can be driven deterministically through all of its states and transitions [16]. For example, in order to know when the final *pop* transition from *Normal* to *Empty* is reached, the accessor *size* is required as one of *Stack*'s methods, so that suitable guards may be constructed. The above design is logically *complete*, in the sense that the modal behaviour of every method from every state is known.

2.2 Elements of the State Model

In formal approaches to state-based test generation, test-sets are constructed from *languages,* sequences of symbols drawn from the total *alphabet* of transition labels. The *alphabet* is the set of methods $m \in M$ that can be called on the interface of the object (including any inherited methods). For the *Stack* shown in figure 2, the alphabet M = {*push*, *pop*, *top*, *empty*, *size*}. Note that *new* is not technically in the method-interface of *Stack*, but rather constructs the *Stack* instance. The object responds to all $m \in M$, and to no other methods (which are ruled out by compile-time syntax checking). This puts a useful upper bound on the scope of negative testing.

The state-transition diagram defines a number states $s \in S$. The states of the *Stack* in figure 2 are S = {*Empty, Normal*}. The "final state" is not treated as a first-class state, but as an exception raised by *pop* from the *Empty* state. It is assumed that the test harness can construct a predicate p : *Stack* → *Boolean* for each state $s \in S$, to detect whether the OUT is in that state. Predicates may freely use public access methods internally: for example, *isEmpty(s) ≡ s.size() = 0*. Our earlier definition of state (see 2.1) ensures that such predicates may always be defined.

Sequences of methods, denoted $<m_1, m_2, \ldots>$, $m \in M$, may be constructed. *Languages* M^0, M^1, M^2, … are sets of sequences of specific lengths; that is, M^0 is the set of zero-length sequences: $\{<>\}$ and M^1 is the set of all unit-length sequences: $\{<m_1>, <m_2>, \ldots\}$, etc. The infinite language M^* is the union $M^0 \cup M^1 \cup M^2 \cup \ldots$ containing all arbitrary-length sequences. A predicate language $P = \{<p_1>, <p_2>, \ldots\}$ is a set of unit-length predicate sequences, testing exhaustively for each state $s \in S$.

2.3 Complete Unit-Test Generation

When testing from a state-based design, the tester drives the OUT into all of its states and then attempts every possible transition (both expected and unwanted) from each

state, checking afterwards which destination states were reached. The OUT should exhibit indistinguishable behaviour from the design, to pass the tests. It is assumed that the design is a *minimal* state machine (with no duplicate, or redundant states), but the tested implementation may be non-minimal, with more than the expected states. Testing must therefore take this into account, exercising more than the minimal state machine. These notions are formalised below.

The *state cover* is a set $C \subseteq M^*$ consisting of the shortest sequences that will drive the OUT into all of its states. C is chosen by inspection, or by automatic exploration of the model. For the *Stack* shown in figure 2 above, C = {<>, <*push*>} is the smallest state cover, which will enter the *Empty* and *Normal* states. An initial test-set T^0 aims to reach and then verify every state. Verification is accomplished by invoking each predicate in the predicate language P after exploring each path in the state cover C, a test set denoted by: $C \otimes P$, the *concatenated product* that appends every sequence in P to every sequence in C. Exactly one predicate in each sequence should return true, and all the others false, as determined from the model.

$$T^0 = C \otimes P \tag{1}$$

A more sophisticated test-set T^1 aims to reach every state and also exercise every single method in every state. This is constructed from the *transition cover,* a set of sequences $K^1 = C \cup C \otimes M^1$, which includes the state cover C and the concatenated product term $C \otimes M^1$, denoting the attempted firing of every single transition from every state. The states reached by the transition cover are validated again using all singleton predicate sequences $<p> \in P$.

$$T^1 = (C \cup C \otimes M^1) \otimes P \tag{2}$$

An even more sophisticated test-set T^2 aims to reach every state, fire every single transition and also fire every possible pair of transitions from each state (sometimes known as the *switch cover*). This is constructed from the augmented set of sequences $K^2 = C \cup C \otimes M^1 \cup C \otimes M^2$ and the reached states are again verified using the predicates. The product term $C \otimes M^2$ denotes the attempted firing of all pairs of transitions from every state.

$$T^2 = (C \cup C \otimes M^1 \cup C \otimes M^2) \otimes P \tag{3}$$

In a similar fashion, further test-sets are constructed from the state cover C and low-order languages $M^k \subseteq M^*$. Each test-set subsumes the smaller test-sets of lesser sophistication in the series. In general, the series can be factorised and expressed for test-sets of arbitrary sophistication as:

$$T^k = C \otimes (M^0 \cup M^1 \cup M^2 \ldots M^k) \otimes P \tag{4}$$

The general formula describes all test-sequences starting with a state cover, augmented by firing all single, pairs, triples etc. of transitions and then verifying the reached states using state predicates.

3 Test Completeness and Guarantees

The test-sets produced by this algorithm have important completeness properties. For each value of k, specific guarantees are obtained about the implementation, once testing is over. Below, we show how regression testing in XP does not provide the

same guarantees after retesting. However, for simple extensions to a state-based design, tests may be re-generated by this algorithm and all the same guarantees upheld.

3.1 Guarantees After Testing

The set T^0 guarantees that the implementation has *at least* all the states in the specification. The set T^1 guarantees this, and that a *minimal* implementation provides exactly the desired state-transition behaviour. The remaining test-sets T^k provide the same guarantees for *non-minimal* implementations, under weakening assumptions about the level of duplication in the states and transitions.

A non-minimal, or *redundant* implementation is one where a programmer has inadvertently introduced extra "ghost" states, which may or may not be faithful copies of states desired in the design. Test sequences may lead into these "ghost" states, if they exist, and the OUT may then behave in subtle unexpected ways, exhibiting extra, or missing transitions, or reaching unexpected destination states. Each test-set T^k provides complete confidence for systems in which chains of duplicated states do not exceed length k-1. For small values of k, such as k=3, it is possible to have a very high level of confidence in the correct state-transition behaviour of even quite perversely-structured implementations.

Both *positive* and *negative* testing are achieved; for example, it is confirmed that access methods do not inadvertently modify object states. Testing avoids any *uniformity assumption* [23], since no conformity to type need be assumed in order for the OUT to be tested. Likewise, testing avoids any *regularity assumption* that cycles in the specification necessarily correspond to implementation cycles. When the OUT "behaves correctly" with respect to the specification, this means that it has all the same states and transitions, or, if it has extra, redundant states and transitions, then these are semantically identical duplicates of the intended states in the specification.

3.2 Refactoring, Subclassing and Test Coverage

Figure 3 illustrates the state machine for a *DynamicStack*, an array-based implementation of a *Stack*. We may think of this either as a subclass design for a concrete class that implements an abstract *Stack* interface [22], or as a refactored design for a *Stack*, after the decision is taken to switch to an array-based implementation. The main difference between this and the earlier machine in figure 2 is that the old *Normal* state, now only shown as a dashed region, has been partitioned into the states {*Loaded, Full*}, in order to capture the distinct behaviour of *push* in the *Full* state, where this triggers a memory reallocation. Though this design appears more complex, it is easily generated by following the simple rule: show the behaviour of *every* method in *every* state (self-transitions for the access methods are inferred, as above).

Increasing the state-space has important implications for test guarantees. Consider the sufficiency of the T^2 test-set, generated from the abstract *Stack* specification in figure 2. This robustly guarantees the correct behaviour of a simple *LinkedStack* implementation with S = {*Empty, Normal*}, even in the presence of "ghost" states. T^2 will generate one sequence <*push, push, push, isNormal*>, which robustly exercises <*push, push*> from the *Normal* state and will even detect a "ghost" copy of the *Normal* state.

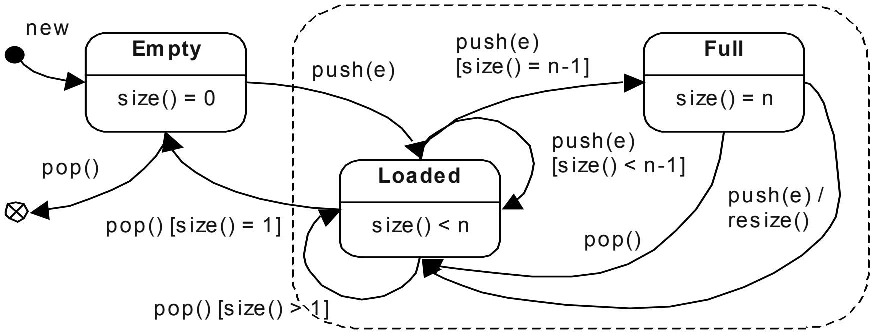


Fig. 3. State machine for a *DynamicStack*, which conforms to the behaviour of the *Stack* in fig. 2. The two states (*Loaded, Full*) partition the old *Normal* state in fig. 2, resulting in the replication of its transitions. The behaviour of *push* in the *Full* state must be tested

In XP regression testing, saved test-sets are reapplied to modified or extended objects in the expectation that passing all the saved tests will guarantee the same level of correctness. If the *Stack's* T^2 test-set were reused to test the *DynamicStack* in figure 3, with S = {*Empty, Loaded, Full*}, the resizing *push* transition would never be reached, since this requires a sequence of four *push* methods. To the tester, it would appear that the *DynamicStack* had passed all the saved T^2 tests, even if a fault existed in the resizing *push* transition. This fault would be undetected by the saved test-set.

In general, subclasses have a larger state-space than superclasses, due to their introduction of more methods affecting object states. In Cook, Daniels [24] and McGregor's [15, 21] state models, subclasses could introduce new states. In Simons' more careful model of state refinement [22, 25], subclassing was shown always to result in the *subdivision of existing states* (as in figure 3). The current paper shows that saved tests not only miss the new states formed in the child, but also fail to test all of the transitions that were tested in the parent, due to the splitting [22] of these transitions. This is important for regression testing, since it means that saved tests exercise *significantly less* of the *same behaviour* in the child as they did in the parent.

To achieve the same level of coverage, it is not sufficient to supply extra tests for the new methods, but rather it is vital to test *all the interleavings* of new methods with the inherited methods, so exploring the state-transition diagram completely. This simply cannot be done reliably by human intuition and manual test-script creation. However, it could be done reliably if the test-sets were *regenerated* for the modified state machine designs, using the algorithm from section 2.3 above.

4 Conclusions

The strength of the guarantee obtained after regression testing is overestimated in XP. Recycled test-sets always exercise significantly less of the refined object than the original, such that re-tested objects may be considerably less secure, for the same testing effort. As the state-space of the modified or extended object increases, the guarantee offered by retesting is progressively weakened. This undermines the validity of popular regression testing approaches, such as parallel design-and-test, test set inheritance and reuse of saved test scripts in JUnit.

By comparison, if complete test-sets are always generated from simple state machine designs, it is possible to provide specific guarantees, for example up to the k=2 or k=3 confidence level, in the OUT. After retesting a subclassed or refactored OUT, the same guarantees may be upheld by generating from the revised state machine to the same confidence level. Not only this, but *all* the objects in a software project may be unit-tested to a given confidence level. This notion of *guaranteed, repeatable quality* is a new and important concept in object-oriented testing.

This research was undertaken as part of the MOTIVE project, supported by UK EPSRC GR/M56777.

References

1. Wells, D.: The Rules and Practices of Extreme Programming. Website and hypertext article http://www.extremeprogramming.org/rules.html (1999)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, New York (2000)
3. Beck, K. and Fowler, M.: Planning Extreme Programming. Addison-Wesley, New York (2000)
4. Kruchten, P.: The Rational Unified Process: an Introduction. 3rd edn. Addison-Wesley, Reading (2003)
5. Object Management Group, UML Resource Page. Website http://www.omg.org/uml/ (2004)
6. Stephens, M. and Rosenberg D.: Extreme Programming Refactored: The Case Against XP. Apress, Berkley (2003)
7. Beck, K. Gamma E. et al.: The JUnit Project. Website http://www.junit.org/ (2003)
8. Stotts, D., Lindsey, M. and Antley, A.: An Informal Method for Systematic JUnit Test Case generation. Lecture Notes in Computer Science, Vol. 2418. Springer Verlag, Berlin Heidelberg New York (2002) 131-143
9. McGregor, J. D. and Korson, T.: Integrating Object-Oriented Testing and Development Processes. Communications of the ACM, Vol. 37, No. 9 (1994) 59-77
10. McGregor, J. D. and Kare, A.: Parallel Architecture for Component Testing of Object-oriented Software. Proc. 9th Annual Software Quality Week, Software Research, Inc. San Francisco, May (1996)
11. Wells, D.: Unit Tests: Lessons Learned, in: The Rules and Practices of Extreme Programming. Hypertext article http://www.extremeprogramming.org/rules/unittests2.html (1999)
12. Schuman, S. A. and Pitt, D. H.: Object-oriented Subsystem Specification. In: Program Specification and Transformation. Elsevier Science, North Holland (1987)
13. Chow, T.: Testing Software Design Modeled by Finite State Machines. IEEE Transactions on Software Engineering, Vol. 4 No. 3 (1978) 178-187
14. Binder, R. V.: Testing Object-Oriented Systems: a Status Report. 3rd edn. Hypertext article http://www.rbsc.com/pages/oostat.html (2001)
15. McGregor, J. D.: Constructing Functional Test Cases Using Incrementally-Derived State Machines. Proc. 11th International Conference on Testing Computer Software. USPDI, Washington (1994)
16. Holcombe, W. M. L. and Ipate, F.: Correct Systems: Building a Business Process Solution. Applied Computing Series. Springer Verlag, Berlin Heidelberg New York (1998)
17. Ipate, F. and Holcombe, W. M. L.: An Integration Testing Method that is Proved to Find All Faults. International Journal of Computational Mathematics, Vol. 63 (1997) 159-178

18. Holcombe, M., Bogdanov, K. and Gheorghe, M.: Functional Test Generation for Extreme Programming. Proc. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering. XP 2001, Sardinia, Italy (2001) 109-113
19. Dranidis, D., Tigka, K. and Kefalas, P.: Formal Modelling of Use Cases with X-machines. Proc. 1st South-East European Workshop on Formal Methods. South-Eeast European Research Centre, Thessaloniki (2004)
20. Holcombe, M.: Where Do Unit Tests Come From? Proc. 4th. International Conference on Extreme Programming and Flexible Processes in Software Engineering. XP 2003, Genova, Italy, Lecture Notes in Computer Science, Vol. 2675. Springer Verlag, Berlin Heidelberg New York (2003) 161-169
21. McGregor, J. D. and Dyer, D. M.: A Note on Inheritance and State Machines. Software Engineering Notes, Vol. 18, No. 4 (1993) 61-69
22. Simons, A. J. H., Stannett, M. P., Bogdanov, K. E. and Holcombe, W. M. L.: Plug and Play Safely: Behavioural Rules for Compatibility. Proc. 6th IASTED International Conference on Software Engineering and Applications. SEA-2002, Cambridge (2002) 263-268
23. Bernot, B., Gaudel, M.-C. and Marre, B.: Software Testing Based on Formal Specifications: a Theory and a Tool. Software Engineering Journal, Vol. 6, No. 6 (1991) 387-405
24. Cook, S. and Daniels, J.: Designing Object-Oriented Systems: Object-Oriented Modelling with Syntropy. Prentice Hall, London (1994)
25. Simons, A. J. H.: Letter to the Editor, Journal of Object Technology. Received December 5, 2003. Hypertext article http://www.jot.fm/general/letters/comment_simons_html (2003)

Examining Usage Patterns of the FIT Acceptance Testing Framework

Kris Read, Grigori Melnik, and Frank Maurer

Department of Computer Science, University of Calgary, Calgary, Alberta, Canada
{readk,melnik,maurer}@cpsc.ucalgary.ca

Abstract. Executable acceptance testing allows both to specify customers' expectations in the form of the tests and to compare those to actual results that the software produces. The results of an observational study identifying patterns in the use of the FIT acceptance testing framework are presented and the data on acceptance-test driven design is discussed.

1 Introduction

Acceptance testing is an important aspect of software development. Acceptance tests are high level tests of business operations and are not meant to test internals or technical elements of the code, but rather are used to ensure that software meets business goals. Acceptance tests can also be used as a measure of project progress. Several frameworks for acceptance testing have been proposed (including JAccept [5], Isis [6], and FIT [2]). The use of acceptance tests have been examined in recent studies from members of both academia [8, 7] and industry [5,10]. We are interested in determining the value of executable acceptance tests, both for quality assurance as well as to represent functional requirements in a test-first environment. To this end we have arranged several experiments and observational studies using tools such as FIT [2] and FitNesse [3] to work with executable acceptance tests. FIT is an acceptance testing framework which has been popularized by agile developers. FIT allows tests to be specified in tables, in multiple formats such as HTML, Excel, or on a wiki page. Although users can specify the test case tables, developers must later implement fixtures (lightweight classes calling business logic) that allow these tables to be executed. Suitability of acceptance tests for specifying functional requirements has been closely examined in our previous paper [4]. Our hypothesis that tests describing customer requirements can be easily understood and implemented by a developer who has little background on this framework was substantiated by the evidence gathered in our previous experiment. Over 90% of teams delivered functioning tests and from this data we were able to conclude that the learning curve for reading and implementing executable acceptance tests is not prohibitively steep.

In this paper we expand on our previous results and investigate the ways in which developers use executable acceptance tests. We seek to identify usage patterns and gather information that may lead us to better understand the strengths and weaknesses of acceptance tests when used for both quality control and requirements representation. Further, examining and identifying patterns may allow us to provide recommendations on how acceptance tests can best be used in practice, as well as for future development of tools and related technologies. In this paper we report on results of observations in an academic setting. This exploratory study will allow us to refine hypotheses and polish the experimental design for future industrial studies.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 127–136, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Context of Study

Data was gathered from two different projects in two different educational institutions over four months. The natures of the two projects were somewhat different; one was an interactive game, and another a Web-based enterprise information system. The development of each project was performed in several two to three week long iterations. In each project, FIT was introduced as a mandatory requirement specification tool. In one project FIT was introduced immediately, and in the other FIT was introduced in the third iteration (half way through the semester). After FIT was introduced, developers were required to interpret the FIT-specified requirements supplied by the instructor. They then implemented the functionality to make all tests pass, and were asked to extend the existing suite of tests with additional scenarios.

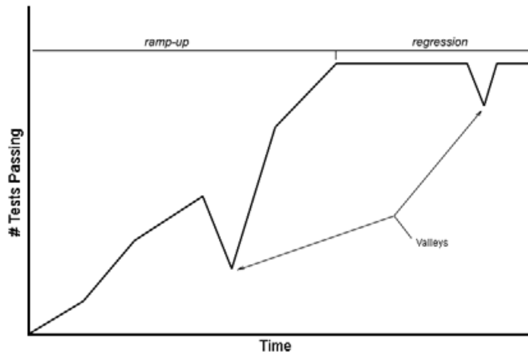


Fig. 1. Typical iteration life-cycle

The timeline of both projects can be split into two sections (see Figure 1). The first time period begins when students received their FIT tests, and ends when they implemented fixtures to make all tests pass. Henceforth this first time period will be called the "*ramp up*" period. Subjects may have used different strategies during ramp up in order to make all tests pass, including (but not limited to) implementing business logic with in the test fixtures themselves, delegating calls to business logic classes from test fixtures, or simply mocking the results within the fixture methods (Table 1). The second part of the timeline begins after the ramp up and runs until the end of the project. This additional testing, which begins after all tests are already passing, is the use of FIT for *regression testing*. By executing tests repeatedly, developers can stay alert for new bugs or problems which may become manifest as they make changes to the code. It is unknown what types of changes our subjects might make, but possibilities range from refactoring to adding new functionality.

3 Subjects and Sampling

Students of computer science programs from the University of Calgary (UofC) and the Southern Alberta Institute of Technology (SAIT) participated in the study. All individuals were knowledgeable about programming, however, no individuals had any knowledge of FIT or FitNesse (based on a verbal poll). Senior undergraduate

Table 1. Samples of how a given fixture could be implemented

Example: In-fixture implementation
`public class Division extends ColumnFixture { public double numerator, denominator; public double quotient() { return numerator/denominator; } }`
Example: Delegate implementation
`public class Division extends ColumnFixture { public double numerator, denominator; public double quotient() { DivisionTool dt = new DivistionTool(); return dt.divide(numerator, denominator); } }`
Example: Mock implementation
`public class Division extends ColumnFixture { public double numerator, denominator; public double quotient() { return 8; } }`

UofC students (20) who were enrolled in the *Web-Based Systems*[1] course and students from the Bachelor of Applied Information Systems program at SAIT (25) who enrolled the *Software Testing and Maintenance* course, took part in the study. In total, 10 teams with 4-6 members were formed.

4 Hypotheses

The following hypotheses were formulated prior to beginning our observations:

H_A: No common patterns of ramp up or regression would be found between teams working on different projects in different contexts.

H_B: Teams will be unable to identify and correct “bugs” in the test data or create new tests to overcome those bugs (with or without client involvement).

H_C: When no external motivation is offered, teams will not refactor fixtures to properly delegate operations to business logic classes.

H_D: When no additional motivation is given, students will not continue to the practice of executing their tests in regression mode (after the assignment deadline).

H_E: Students will not use both suites and individual tests to organize/run their tests.

5 Data Gathering

A variety of data gathering techniques were employed in order to verify hypotheses and to provide further insight into the usage of executable acceptance testing. Subjects used FitNesse for defining and executing their tests. FitNesse [3] is an open-source wiki-based tool to manage and run FIT tests. For the purposes of this study, we provided a binary of FitNesse that was modified to track and record a history of FIT test executions, both successful and unsuccessful. Specifically, we recorded:

- Timestamp;
- Fully-qualified test name (with test suite name if present);
- Team;
- Result: number right, number wrong, number ignored, number exceptions.

---

[1] http://mase.cpsc.ucalgary.ca/seng513/F2004

The test results are in the format produced by the FIT engine. *Number right* is the number of passed assertions, or more specifically the number of "green" table cells in the result. *Number wrong* is the number of failed assertions, which are those assertions whose output was different from the expected result. In FIT this is displayed in the output as "red" table cells. *Ignored* cells were for some reason skipped by the FIT engine (for example due to a formatting error). *Number exceptions* records exceptions that did not allow a proper pass or fail of an assertion. It should be noted that a single exception if not properly handled could halt the execution of subsequent assertions. In FIT exceptions are highlighted as "yellow" cells and recorded in an error log. We collected 25,119 different data points about FIT usage.

Additional information was gathered by inspecting the source code of the test fixtures. Code analysis was restricted to determining the type of fixture used, the non-commented lines of code in each fixture, the number of fields in each fixture, the number of methods in each fixture, and a subjective rating from 0 to 10 of the "fatness" of the fixture methods: 0 indicating that all business logic was delegated outside the fixture (desirable), and 10 indicating that all business logic was performed in the fixture method itself (see Table 1 for examples of fixture implementations).

Analysis of all raw data was performed subsequent to course evaluation by an impartial party with no knowledge of subject names (all source code was sanitized). Data analysis had no bearing or effect on the final grades.

6 Analysis

This section is presented in four parts, each corresponding to a pattern observed in the use of FIT. *Strategies of test fixture design* looks at how subjects construct FIT tables and fixtures; *Strategies for using test-suites vs. single tests* examines organization of FIT tests; *Development approaches* identifies subject actions during development; and *Robustness of test specification* analyzes how subjects deal with exceptional cases.

6.1 Strategies of Test Fixture Design

It is obvious that there are multitudes of ways to develop a *fixture* (a simple interpreter of the table) such that it satisfies the conditions specified in the table (test case). Moreover, there are different strategies that could be used to write the same fixture. One choice that needs to be made for each test case is what type of FIT fixture best suits the purpose. In particular, subjects were introduced to RowFixtures and ActionFixtures in advance, but other types were also used at discretion of the teams (see Table 2). Some tests involved a combination of more than one fixture type, and subjects ended up developing means to communicate between these fixtures.

Another design decision made by teams was whether to develop "fat", "thin" or "mock" methods within their fixtures (Table 3). "Fat" methods implement all of the business logic to make the test pass. These methods are often very long and messy, and likely to be difficult to maintain. "Thin" methods delegate the responsibility of the logic to other classes and are often short, lightweight, and easier to maintain. Thin methods show a better grasp on concepts such as good design and refactoring, and facilitate code re-use. Finally, "mock" methods do not implement the business logic

Table 2. Common FIT fixtures used by subjects

Fixture Type	Description	Frequency of Use
RowFixture	Examines an order-independent set of values from a query.	12
ColumnFixture	Represents inputs and outputs in a series of rows and columns.	0
ActionFixture	Emulates a series of actions or events in a state-specific machine and checks to ensure the desired state is reached.	19
RowEntryFixture	Special case of ColumnFixture that provides a hook to add data to a dataset.	2
TableFixture	Base fixture type allowing users to create custom table formats.	30

or functionality desired, but instead return the expected values explicitly. These methods are sometimes useful during the development process but should not be delivered in the final product. The degree to which teams implemented fat or thin fixtures was ranked on a subjective scale of 0 (entirely thin) to 10 (entirely fat).

The most significant observation that can be made from Table 3 is that the UofC teams by and large had a much higher fatness when compared to the SAIT teams. This could possibly be explained by commonalities between strategies used at each location. At UofC, teams implemented the test fixtures in advance of any other business logic code (more or less following Test-Driven Development philosophy [9]). Students may not have considered the code written for their fixtures as something which needed to be encapsulated for re-use. This code from the fixtures was further required elsewhere in their project design, but may have been "copy-and-pasted". No refactoring was done on the fixtures in these cases. This can in our opinion be explained by a lack of external motivation for refactoring (such as additional grade points or explicit requirements). Only one team at the UofC took it upon themselves to refactor code without any prompting. Conversely, at SAIT students had already implemented business logic in two previous iterations, and were applying FIT to existing code as it was under development. Therefore, the strategy for refactoring and maintaining code re-use was likely different for SAIT teams. In summary, acceptance test driven development failed to produce reusable code in this context. Moreover, in general, teams seem to follow a consistent style of development – either tests are all fat or tests are all thin. There was only one exception in which a single team did refactor some tests but not all (see Table 2, UofC T2).

6.2 Strategies for Using Test Suites vs. Single Tests

Regression testing is undoubtedly a valuable practice. The more often tests are executed, the more likely problems are to be found. Executing tests in suites ensures hat all test cases are run, rather than just a single test case. This approach implicitly forces developers to do regression testing frequently. Also, running tests as a suite ensures that tests are compatible with each other – it is possible that a test passes on its own but will not pass in combination with others.

In this experiment data on the frequency of test suite vs. single test case executions was gathered. Teams used their own discretion to decide which approach to follow (suites or single tests or both). Several strategies were identified (see Table 4).

Table 3. Statistics on fixture fatness and size

Team	Fatness (subjective)		NCSS[a]	
	Min	Max	Min	Max
UofC T1	7	10	28	145
UofC T2	0	9	8	87
UofC T3	8	10	40	109
UofC T4	9	10	34	234
SAIT T1	0	1	7	57
SAIT T2	0	2	22	138
SAIT T3	0	0	24	57
SAIT T4	0	0	15	75
SAIT T5	1	2	45	91
SAIT T6	0	1	13	59

[a] NCSS is Non-Comment Source Lines of Code, as computed by the JavaNCSS tool: http://www.kclee.de/clemens/java/javancss/

Table 4. Possible ramp up strategies

Strategy	Pros	Cons
(*) Exclusively using single tests	– fast execution – enforces baby steps development	– very high risk of breaking other code – lack of test organization
(**) Predominantly using single tests	– fast most of the time execution – occasional use of suites for regression testing	– moderate risk of breaking other code
(***) Relatively equal use of suites and single tests	– low risk of breaking other code – immediate feedback on the quality of the code base – good organization of tests	– slow execution when the suites are large

Exclusively using single tests may render faster execution; however, it does not ensure that other test cases are passing when the specified test passes. Also, it indicates that no test organization took place which may make it harder to manage the test base effectively in the future. Two teams (one from UofC and one from SAIT) followed this approach of single test execution (Table 5). Another two teams used both suites and single tests during the ramp up. A possible advantage of this strategy may be a more rapid feedback on the quality of the entire code base under test. Five out of nine teams followed the strategy of predominantly using single test, but occasionally using suites. This approach provides both organization and infrequent regression testing. Regression testing using suites would conceivably reduce the risk of breaking other code. However, the correlation analysis of our data finds no significant evidence that any one strategy produces fewer failures over the course of the ramp up. The ratio of peaks and valleys (in which failures occurred and then were repaired) over the cumulative test executions fell in the range of 1-8% for all teams. Moreover, even the number of test runs is not deterministic of strategy chosen.

During the regression testing stage we also measured how often suites versus single test cases were executed (Table 6). For UofC teams, we saw a measured difference in how tests were executed after the ramp up. All teams now executed single test cases more than suites. Team 1 and Team 2 previously had executed suites more than single cases, but have moved increasingly away from executing full test suites. This may be due to troubleshooting a few problematic cases, or may be a result of increased deadline pressure. Team 3 vastly increased how often they were running test

suites, from less than half the time to about three-quarters of executions being performed in suites. Team 4 who previously had not run any test suites at all, did begin to run tests in an organized suite during the regression period. For SAIT teams we see a radical difference in regression testing strategy: use single test case executions much more than test suites. In fact, the ratios of single cases to suites are so high as to make the UofC teams in retrospect appear to be using these two types of test execution equally. Obviously, even after getting tests to pass initially, SAIT subjects felt it necessary to individually execute far more individual tests than the UofC students. Besides increased deadline pressure, a slow development environment might have caused.

Table 5. Frequency of test suites versus single test case executions during ramp up

Team	Suite Executions	Single Case Executions	Single/Suite Ratio
UofC T1 (***)	650	454	0.70
UofC T2 (***)	314	253	0.80
UofC T3 (**)	169	459	2.72
UofC T4 (*)	0	597	Exclusively Single Cases
SAIT T1 (**)	258	501	1.94
SAIT T2 (**)	314	735	2.40
SAIT T3 (**)	49	160	3.27
SAIT T4 (*)	8	472	59.00
SAIT T5 (**)	47	286	6.09
SAIT T6 (not included due to too few data points).	8	25	3.13

Table 6. Frequency of suites versus single test case executions during regression (post ramp up)

Team	Suite Executions	Single Case Executions	Single/Suite Ratio
UofC T1	540	653	1.21
UofC T2	789	1042	1.32
UofC T3	408	441	1.08
UofC T4	72	204	2.83
SAIT T1	250	4105	16.42
SAIT T2	150	3975	26.50
SAIT T3	78	1624	20.82
SAIT T4	81	2477	30.58
SAIT T5	16	795	49.69
SAIT T6	31	754	24.32

6.3 Development Approaches

The analysis of ramp up data demonstrates that all teams likely followed a similar development approach. Initially, no tests were passing. As tests are continued to be executed, more and more of the assertions pass. This exhibits the iterative nature of the development. We can infer from this pattern that features were being added incrementally to the system (Figure 2, left). Another approach could have included many assertions initially passing followed by many valleys during refactoring. That would illustrate a mock-up method in which values were faked to get an assertion to pass and then replaced at a later time (Figure 2, right).

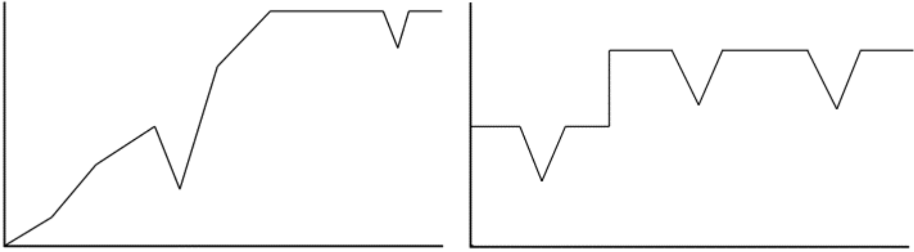


Fig. 2. A pattern of what incremental development might look like (left) versus what mocking and refactoring might look like (right)

Noticeably, there were very few peaks and valleys[2] during development (Table 7). A valley is measured when the number of passing assertions actually goes down from a number previously recorded. Such an event would indicate code has broken or an error has occurred. These results would indicate that in most cases as features and tests were added, they either worked right away or did not break previously passing tests. In our opinion, this is an indication that because the tests were specified upfront, they were driving the design of the project. Because subjects always had these tests in mind and were able to refer to them frequently, they were more quality conscious and developed code with the passing tests being the main criteria of success.

Table 7. Ratio of valleys found versus total assertions executed

Team	"Valleys" vs. Executions in Ramp Up Phase	"Valleys" vs. Executions in Regression Phase
UofC T1	0.03	0.05
UofC T2	0.07	0.10
UofC T3	0.03	0.10
UofC T4	0.01	0.05
SAIT T1	0.06	0.12
SAIT T2	0.03	0.10
SAIT T3	0.04	0.09
SAIT T4	0.05	0.06
SAIT T5	0.05	0.09
SAIT T6	0.03	0.14

6.4 Robustness of the Test Specification

Several errors and omissions were left in the test suite specification delivered to subjects. Participants were able to discover all such errors during development and immediately requested additional information. For example, one team posted on the experience base the following question: *"The acceptance test listed … is not complete (there's a table entry for "enter" but no data associated with that action). Is this a leftover that was meant to be removed, or are we supposed to discover this and turn it into a full fledged test?"* In fact, this was a typo and we were easily able to clarify the requirement in question. Surprisingly, typos or omissions did not seem to affect subjects' ability to deliver working code. This demonstrates that even with errors in the

[2] The number of peaks equals the number of valleys. Henceforth we refer only to valleys.

test specification, FIT adequately describes the requirements and makes said errors immediately obvious to the reader.

7 Conclusion

Our observations lead us to the following conclusions. Our hypothesis that no common patterns of ramp up or regression would be found between teams working on different projects in different contexts was only partly substantiated. We did see several patterns exhibited, such as incremental addition of passing assertions and a common use of preferred FIT fixture types. However, we also saw some clear divisions between contexts, such as the relative "fatness" of the fixtures produced being widely disparate. The fixture types students used were limited to the most basic fixture type (TableFixture) and the two fixture types provided for them in examples. This may indicate that rather than seeing a pattern in what fixture types subjects chose, we may need to acknowledge that the learning curve for other fixture types discouraged their use. Subjects did catch all "bugs" or problems in the provided suite of acceptance tests, refuting our hypothesis and demonstrating the potential for implementing fixtures despite problems. Our third hypothesis, that teams would not refactor fixtures to properly delegate operations to business logic classes, was confirmed. In the majority of cases, when there was no motivation to do so students did not refactor their fixture code but instead had the fixtures themselves perform business operations. Subjects were aware that this was bad practice but only one group took it upon themselves to "do it the right way". Sadly, the part of our subject pool that was doing test-first was most afflicted with "fat" fixtures, while those students who were writing tests for existing code managed by large to reuse that code. In all cases, students used both suites and individual test cases when executing their acceptance tests. However, we did see that each of the groups decided for themselves when to run suites more often than single cases and vice versa. It is possible that these differences were the result of strategic decisions on behalf of the group, but also possible that circumstance or level of experience influenced their decisions.

Our study demonstrated that subjects were able to interpret and implement FIT test specifications without major problems. Teams were able to deliver working code to make tests pass and even catch several bugs in the tests themselves. Given that the projects undertaken are similar to real world business applications, we suggest that lessons learned in this paper are likely to be applicable to an industrial setting. Professional developers are more experienced with design tools and testing concepts, and, therefore, would likely overcome minor challenges with as much success as our subjects (if not more).

References

1. Chau, T., Maurer, F. Tool Support for Inter-Team Learning in Agile Software Organizations. Proc. LSO 2004, Springer, LNCS, Vol. 3096: 98-109, 2004.
2. Cunnigham, W. FIT: Framework for Integrated Test. Online http://fit.c2.com. Last accessed on Jan 15, 2005.
3. Fitnesse. Online http://fitnesse.org. Last accessed on Jan 15, 2005.

4. Melnik, G., Read, K., Maurer, F. Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective. Proc. XP/Agile Universe 2004, LNCS, Vol. 3134, Springer Verlag: 60–72, 2004.
5. Miller, R., Collins, C. Acceptance Testing. Proc. XPUniverse 2001, July, 2001.
6. Mugridge, R., MacDonald, B., Roop, P. A Customer Test Generator for Web-Based Systems. Proc. XP2003, LNCS, Vol.2675, Springer Verlag: 189-197, 2003.
7. Mugridge, R., Tempero, E. Retrofitting an Acceptance Test Framework for Clarity. Proc. Agile Development Conference 2003, IEEE Press: 92-98, 2003.
8. Steinberg, D. Using Instructor Written Acceptance Tests Using the Fit Framework, Lecture Notes in Computer Science, LNCS, Vol. 2675, Springer Verlag: 378-385, 2003.
9. Test Driven Development. Online http://c2.com/cgi/wiki?TestDrivenDevelopment. Last accessed on Jan 15, 2005.
10. Watt, R., Leigh-Fellows, D. Acceptance Test Driven Planning, Proc.XP/Agile Universe 2004, LNCS, Vol. 3134, Springer Verlag: 43-49, 2004.

Agile Test Composition

Rick Mugridge[1] and Ward Cunningham[2]

[1] University of Auckland, New Zealand
r.mugridge@auckland.ac.nz
[2] Cunningham & Cunningham Inc, and Microsoft Corporation
ward@c2.com

Abstract. Storytests in storytest driven development serve two interrelated goals. On the one hand, they are used to formulate and communicate business rules. On the other, they are used to verify that a story has been completed and that it hasn't been subsequently broken.

There is a small conflict between these views. For their communicative role, storytests are better to be concise and independent. For automated testing, speed is important in providing fast feedback, and so it makes sense to combine storytests. We show how this conflict can be avoided by automatically combining storytests. Hence the value of storytests for defining the needs of the system is not diminished when it comes to automated testing.

1 Introduction

In the story-driven development of an agile project [1,2], many storytests (customer tests) will be created. The initial aim of the storytests is to communicate (and formulate) the business objects, processes and rules that are important for the system under development.

There are many good reasons to keep the storytests concise and concerned with a single business rule. However, doing this can be at odds with the efficiency and effectiveness of those storytests when they are run against the system under test. In particular, fast feedback from storytests is crucial to support the storytest-driven development process. This applies particularly to business processes, or workflow, where there can be many interrelated storytests.

We next introduce workflow-based storytests in Fit [3, 4], through a trivial example, and show the desire for combining them. We then show how we can instead think of storytests as defining transitions within a directed graph. This allows us to define storytests independently, but to combine them automatically in interesting ways. We discuss a tool that does this, and conclude with some thoughts on combining storytests in other, more complex ways.

2 Workflow Storytests

Workflow storytests tend to have three components:

- The *setup*, when the system under test is put into a particular state.
- The *change* or *transition*, when something happens to the system.
- The *check* of the expected resulting state.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 137–144, 2005.
© Springer-Verlag Berlin Heidelberg 2005

For example, here's a simple storytest in Fit that carries out a video rental and checks that the number of remaining copies of that video has changed appropriately:

StartVideo

VideosSetUp	
name	count
A Vampire in Auckland	3
Notre Damned	2

Rent	
name	rent()
A Vampire in Auckland	true

VideosList	
name	count
A Vampire in Auckland	2
Notre Damned	2

The first two tables define the *setup*, where the system is started and data about two videos is entered. The third table carries out a *change*, when the rental takes place. The last, *check* table ensures that the rental counts have changed correctly.

Another storytest for video rentals checks when a video is returned:

StartVideo

VideosSetUp	
name	count
A Vampire in Auckland	2
Notre Damned	2

Rent	
name	returns()
A Vampire in Auckland	true

VideosList	
name	count
A Vampire in Auckland	3
Notre Damned	2

With many storytests in a test suite, there can be considerable duplication of the *setup* tables. Some testing frameworks provide support for sharing the *setup* between storytests. For example, FitNesse [5] and the FolderRunner [6] allow for `SetUp` (and `TearDown`) pages/files, which are automatically included in the storytests of a test suite as they are run.

However, the *check* of the first storytest above is almost identical to the *setup* of the second storytest. The state of the system under test at the end of the first storytest is expected to be the same as the state at the start of the second storytest. With the extra effort of defining the *setup* and *check* for the two storytests, there's an incentive to simply combine the two storytests. This avoids test-writing effort, and also speeds up test execution because there is less *setup* required. Here's the resulting storytest:

StartVideo

VideosSetUp	
name	count
A Vampire in Auckland	3
Notre Damned	2

Rent	
name	rent()
A Vampire in Auckland	true

VideosList	
name	count
A Vampire in Auckland	2
Notre Damned	2

Rent	
name	returns()
A Vampire in Auckland	true

VideosList	
name	count
A Vampire in Auckland	3
Notre Damned	2

But this tangles the storytests and the combined storytest is longer to read as a unit. If a business change requires that rentals are tested differently, the combined storytest needs to be changed. If we want other storytests to follow after the first one, we'll still need the *setup* defined in the second storytest on the previous page.

How can we keep the storytests independent, while making better use of the *setup*s and *check*s that we've defined?

3 Storytests as Transitions

Let's explicitly treat each storytest as defining a *transition* from a named *start state* to a named *final state*. For example, our first storytest is then written as:

start state	StateInitial

Rent	
name	rent()
A Vampire in Auckland	true

final state	StateOne

Our second storytest is written as:

start state	StateOne

Rent	
name	returns()
A Vampire in Auckland	true

final state	StateInitial

States are then defined independently. The common state *StateOne* is defined as follows:

Videos	
name	count
A Vampire in Auckland	2
Notre Damned	2

When a single storytest is to be run as a test, the *start state* is used as a *setup* and the *final state* as a *check*. So the interpretation of a state depends on whether it's used before or after the *transition*. If before, the state definition is treated as a *setup*. If after, as a *check*. This is achieved in Fit through having the fixture for any table in a *state* choosing to act as either a *setup* or *check*, depending on whether it appears before or after the table *switch setup check*.

When two storytests are combined into a sequence as a single test, they will need to share a common state. The *final state* of the first one is the same as the *start state* of the second. Then the resulting test only needs to mention the intermediate state once, as a check.

For an example of a single storytest, our rental storytest will be generated as a test with the *start state*, the *transition*, *switch setup check* and the *final state*, as follows:

Videos	
name	count
A Vampire in Auckland	3
Notre Damned	2

Rent	
name	rent()
A Vampire in Auckland	true

switch setup check

Videos	
name	count
A Vampire in Auckland	2
Notre Damned	2

Given this mechanism, a test consisting of a sequence of transitions (storytests) can be constructed automatically. We could do this by specifying the sequence explicitly. But that's not necessary, as we can tell which transitions can follow by the previous *final state*.

4 Tests as Graphs

Once we've written lots of storytests as *transition*s which use, and reuse, a set of states, we end up with a directed graph (actually, possibly a set of unconnected graphs).

We can now make use of the connectivity of the *transition*s in the graph. For example, consider the graph shown in Fig 1, where the states are shown as boxes containing the numbers of the two videos. The *transition*s are shown as labelled arcs. For example, our first transition is labelled *t1*.

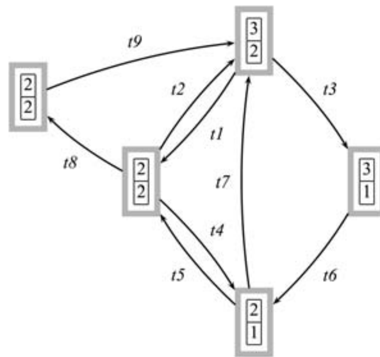


Fig. 1. The State Graph

We can automatically construct a sequence of tests (or *path*s), starting from some state and following the *transition*s. For example, the following are possible paths from the graph in Fig. 1: 1-2, 1-4-5-8, and 6-5.

With a given *path*, we don't duplicate the states between the *transition*s. The first state in the *path* is for *setup* and the rest are *check*s. If we assume that a *path* will usually suceed, we can easily drop out all the intermediate *check*s. This will provide poor diagnostic information when things go wrong, so we want to be able to easily run a failing *path* with all the *check*s in place, so we find out as early as possible in the *path* where things went wrong.

5 Which *Paths*?

With cycles in the graph, as in Fig. 1, there are an infinite number of possible *path*s (transition sequences). How do we choose which ones? How do we choose some *path*s that give good coverage of the storytests (and thus the system under test)? What do we mean by coverage?

In testing, there are several notions of coverage. We consider three here in relation to *path*s in a graph:

- Each *transition* occurs (at least) once. This is the simplest and weakest coverage because it doesn't take account of different *transition*s. This can be achieved by having a separate *path* for each *transition*. Or it can be achieved by building several paths that together visit all of the transitions, with the advantage of efficiencies in test execution.
- Each *transition* is followed by all *transition*s that are "strongly" affected by that change (corresponding to *def-use*[1]). This will pick up interaction faults, although we can't determine the *def-use* dependencies automatically.
- Each *transition* is followed by all *transition*s that are reachable from that *transition* (including itself if there are suitable cycles).

[1] Thanks to Brian Marick for pointing this possibility out

Several *path*s are likely to be needed to achieve the second and third forms of coverage. Let's compare the value of the second and third approaches. If the graph is very large, we may need lots of *path*s to achieve the third, which will take a long time to test.

So the second approach gives us the possibility of smaller running times, by selecting the pairs of *transition*s that need to be tested together. But this incurs the manual cost of deciding on and recording the *def-use* dependencies between storytests. in addition, each time the storytests change, as the stories, storytests and system evolve, we have to reconsider our decisions.

In general, more expensive tests can be run less often, so it's possible to reach a balance between fast feedback from tests while getting good coverage. The advantage of selecting paths automatically is that the approach used can be tailored easily to fast feedback or to strong cover. For example, faster feedback will be provided when we just require that each *transition* occurs (at least) once.

6 Choosing *Paths*

It is straightforward to select paths that ensure that each *transition* occurs (at least) once. We now consider the more complex case, with pairs of transitions, as much the same approach is used.

We have two ways to choose which pairs of *transition* are to be tests: either by someone explicitly stating them or by automatically basing it on *reachability*. With *reachability*, we determine all transitions that can be reached through the graph from a particular transition, including itself. We now consider the choice of the *path*s to satisfy those pairs.

In general, it's impossible to find an optimal set of paths to cover the pairs required. For each such pair we can randomly walk the graph from the initial to the final *transition*, possibly taking the shortest route. Once we have a set of *path*s, we can eliminate any that are redundant (ie, that don't improve transition-pair coverage).

But how do we know what's better? If the tests run very fast, it doesn't matter that there are hundreds of millions of them. But often, this many tests will take too long to execute.

7 Searching for Better *Paths*

The way forward is to use search. A simple approach is to repeatedly generate the paths for a graph, keeping the best set one after many trials. This process will be cheap compared to the cost of running the tests, so it's worth doing.

This however, won't necessarily give us anything like the best solution, as it's constrained by the topology of the graph. In general, we can do better by taking the paths that result from the first stage, as above, and processing them further.

We do this using a *hill-climbing* technique. The idea is to start with a valid solution (to satisfying the pairs coverage in our case), and make changes to that solution to see whather we can reduce the cost. This search can be run many times to find the best one so far.

Now hill climbing suffers from local maxima (if we continue with the metaphor and treat cost as going down). We can avoid this limitation by using *simulated annealing*. This allows the search to break out of local maxima by occasionally making changes to the solution that make things worse (increase the cost). The probability of making such changes is reduced as the search proceeds, so that it eventually makes a choice. In general, this leads to better solutions faster than with hill-climbing.

8 The Whole Process

Let's summarise the steps in terms of a tool that we've written that does this. It carries out the following steps:

1. Read the set of *state* and *transition* files for all of the storytests.
2. Construct a graph (or set of graphs) for them, based on the common *state*s.
3. For each graph, use the reachability of the graph to generate all *transition* pairs (or, read them from a file that defines the desired ones).
4. For each graph, using the *transition* pairs, generate a set of *path*s that satisfy those pairs.
5. For the *path*s and corresponding pairs, use search to find a cheaper solution.
6. Translate each *path* by mapping each *transition* back into the *transition* files and generate each one into a Fit test which is written to a file.

The generated tests can be run at any time, as often as desired. They will need to be updated each time the storytests change. This can be arranged to be done automatically. If the graph structure is unaltered, the *path* generation doesn't need to be done again; the previously-generated *path*s can be reused to map the current independent tests afresh.

9 Experience

We have used this approach on small sets of storytests to test-drive the tool and gain experience in its use. Further work is needed to make the tool generally applicable and available.

We have looked at two large test suites to see whether this approach could be used well with them, assuming that the workflow storytests were restructured into *state*s and *transition*s. It appears that the test suites would be clearer and simplier with this change. It may not be possible to automatically restructure such tests, but it may be possible to provide some support in their migration.

We have several new projects under way in which workflow storytests are being written in this style from the start. Early feedback implies that this is a useful way of thinking about writing workflow tests.

10 Conclusions and Future Work

We have shown how storytests can be written independently and combined to improve test coverage. This means that there doesn't need to be a conflict between the needs of

the storytest writer with regard to test-driven development [2]. Storytests can be written simply and independently. They are combined automatically to satisfy the testing coverage criteria.

A similar, graph-based approach is used by Holcombe and Ipate, although they are yet to integrate it with XP development [7].

This approach is fine for single-user systems, but what about when several users are using a system concurrently? Or when a user, through a GUI, can be going back and forth between several interaction sequences? It would be great to automatically combine storytests for these situations. However, the composed tests that we generate don't address the issue of interactions between those users.

We are developing an extension of our tool that takes two paths generated from test graphs and interleaves them. Clearly, there are constraints on what paths can be interleaved. For example, a video can't be rented in one storytest if they have all already been rented in an interleaved storytest. There are also constraints on what checks can be made, because the transitions are written to be independent. We approach these issues by using a simple model (a simulator) to ensure that only valid interleavings are generated, and by making use of cycles in the graph.

References

1. *Storytest* was coined by Joshua Kerievsky, http://www.industrialxp.org.
2. *Test Driven Development: By Example*, Kent Beck, Addison Wesley, 2002.
3. *Fit for Developing Software: Framework for Integrated Tests*, Rick Mugridge and Ward Cunningham, Prentice-Hall, 2005.
4. *Fit* http://fit.c2.com.
5. *FitNesse* http://www.fitnesse.org.
6. *FitLibrary*, available from https://sourceforge.net/projects/fitlibrary.
7. *Complete test generation for Extreme Programming*, Mike Holcombe and Florentin Ipate, procs. *XP2004*, pp274-277.

E-TDD – Embedded Test Driven Development a Tool for Hardware-Software Co-design Projects

Michael Smith[1], Andrew Kwan[1], Alan Martin[1], and James Miller[2]

[1] Department of Electrical and Computer Engineering
University of Calgary, Calgary, Alberta, Canada T2N 1N4
smithmr@ucalgary.ca
[2] Department of Electrical and Computer Engineering
University of Edmonton, Edmonton, Alberta, Canada T6G 2V4
jm@ee.ualberta.ca

Abstract. Test driven development (TDD) is one of the key Agile practices. A version of *CppUnitLite* was modified to meet the memory and speed constraints present on self-contained, high performance, digital signal processing (DSP) systems. The specific characteristics of DSP systems required that the concept of refactoring be extended to include concepts such as "refactoring for speed". We provide an experience report describing the instructor-related advantages of introducing an embedded test driven development tool E-TDD into third and fourth year undergraduate Computer Engineering Hardware-Software Co-design Laboratories. The TDD format permitted customer (instructor) hardware and software tests to be specified as "targets" so that the requirements for the components and full project were known "up-front". Details of *CppUnitLit* extensions necessary to permit tests specific for a small hardware-software co-design project, and lessons learnt when using the current E-TDD tool, are given. The next stage is to explore the use of the tool in an industrial context of a video project using the hybrid communication-media (HCM) dual core Analog Devices ADSP-BF561 Blackfin processor.

1 Introduction

Many commentators agree that we are on the threshold of new era of computing characterized not by the expansion of desktop or mainframe systems but by a new breed of embedded (and invisible) product entering the marketplace. Third era (3E) computing products will contain a processor, rather than "a computer", and include everything from wearable phones, to guiding the blind using intelligent GPS, to drive-by-wire automobiles, to wireless remote sensing of patient life signs. These 3E systems promise to become all-pervasive in modern life of the industrialized world. No longer will there be the traditional relationship of one computer per person; but instead each individual will be served by many machines that will be highly embedded, fully interconnected and often highly mobile.

While these new systems provide a highly exciting view of the next era, the Information Technology (I.T.) industry has a steep hill to climb to play its part in the pervasive computing revolution. These products will often have extreme characteristics when compared with current existing systems. On the software side, dramatic increases in, for example: the safety-related properties; the reliability and availability of software-intensive systems. The very expected mobility of the software (including

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 145–153, 2005.
© Springer-Verlag Berlin Heidelberg 2005

object-code) and data imply increased security problems that will move the software industry into new intellectual and technical domains. On the hardware side, the issues are equally demanding whether with embedded systems or the envisioned more complex and diverse ubiquitous systems. The problems include the system level validation of mixed signal computations (analog, digital, wireless), hard real-time and throughput requirements, size, weight, area and power (SWAP) constraints, quality of service, environmental effects, *etc*.

According to a newly released study commissioned by the National Institute of Standards and Technology (NIST) [1], software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually. The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved verification infrastructure that enables earlier and more effective identification and removal of software defects. Early identification of defects is more than just desirable with the new systems, as the firmware in the product may not be upgradeable; being burnt directly into the custom silicon. The concepts of "Getting it right the first time" [2] and the "Blue screen of death" gain a real personal interpretation when applied to the firmware inside YOUR surgically-implanted heart defibrillator!

Although challenges exist right across the board, we argue that if these types of systems are to be successful then the production methodology needs to be concentrated on the verification and validation of these systems and that these activities must drive the production process! Many traditional production processes, such as the waterfall or spiral models defer the key faultfinding verification and validation activities too late in the life cycle, resulting in these components often being considered as an "afterthought" to the production component. Further the "testing" activities often get disassembled into artificial sub-activities such as unit testing, integration testing and system testing. The popular "V model" of an integrated testing process is a good example of this type of structuring; this model results in the production team being forced into conducting these sub-activities regardless of the relative cost-benefit issues associated with each of these sub-activities. Clearly in any arbitrary project, the costs and benefits associated with any verification and validation activity will be highly dependent on the domain of operation and the product under development.

Since testing now accounts for more than half of the costs on many projects, any production methodology that fails to actively consider the costs and benefits of the testing activities is potentially wasting an extremely large amount of resources and failing to perfect the product for the marketplace. In addition, Pervasive Computing Systems are highly likely to present a different and potentially more demanding verification and validation puzzle, further suggesting that an alteration to current practices are required. Even in comparatively simple embedded systems, system testing is an undertaking that is often beyond the capacity and ability of many organizations; and hence this component is often less than satisfactory on release date. This implies that even large well-organized companies, utilizing the latest technology, struggle to meet the demands of system testing. Unfortunately, these demands are set to explode as the numbers of critical non-functional requirements multiply.

Recently the software engineering community has started to migrate towards a design process specifically intended to minimize the number of faults during development of software. Considerable effort has been directed towards Test Driven Devel-

opment (TDD) and Extreme Programming (XP) as initial attempts towards establishing such a fault-intolerant design process. However the focus to date has been directed towards functional correctness for business / desk-top applications. We propose to adapt and extend these initial ideas to develop and demonstrate a fault-intolerant design framework initially for embedded systems, and subsequently for ubiquitous 3E devices described in the earlier paragraphs of this introduction.

However, it is very quickly obvious that embedded system Extreme Programming (E-XP) and embedded system Test Driven Development (E-TDD) require the acquisition of a new mind set from the traditional Agile development team familiar with delivering functional incremental software releases combined with improvement ("refactoring") of existing code. By comparison, the new systems may well be monolithic with system components having low cross-cohesion and cross-coupling, but yet "nothing works unless everything works!" The real time functionality, mobile nature, high-volume and low margin found with these systems implies that refactoring may refer to "speed improvements" and "reduced power consumption", even when these factors involves many tradeoffs between portability, clarity and modularity [2].

In this paper, we describe the modifications necessary to permit a test driven development tool, *CppUnitLite* [3], to be used with embedded systems where every last cycle, watt and developer's work-second count in releasing a product. We detail the advantages of using the prototype E-TDD tool as a teaching tool within an undergraduate "hands-on" hardware-software co-design laboratory course in Fall 2004.

2 Implementation of a *CppUnitLite* TTD Tool Variant to Meet the Requirements of Embedded System

A key identified problem is that embedded systems software development is frequently limited to the compiler environment and (very low level) debugging tools that examine the hardware interface directly. In addition, as embedded systems are real-time and performance constrained, it is difficult to gather extensive execution tracing logs, or capture the entire execution context [2] without completely disrupting the functionality of the system.

We chose to adapt the small *CppUnitLite* developed by Michael Feathers [3]. A key element in adapting this tool was to minimize memory usage to ensure that it would fit within the embedded system environment. In particular, any print statements which involved formatting were replaced by simpler statements (***puts(astring)***). The sheer generality of the formatting associated with statements such as **cout << value** or **printf("%d", value)** can generate sufficient instructions to occupy most of the available program memory space available (on-chip) within the target system's processor; possibly leaving little room for other code.

Further reason for choosing *CppUnitLite* as a basis for the E-TDD tool was the fact that it was "non-scripted" means that the framework detects the tests to be run automatically, freeing the developer from working with a script running on an external development environment (PC). Such external scripts require that the embedded system be stopped in order to run, or report on, or be interrogated about, a specific test. Messages are sent back from the target to the development environment over a serial, USB or JTAG connection. Basically the target must be stopped and switched into "emulator interrupt" mode. This permits the message to be "wangled" out of the tar-

get “one character at time”, but completely disrupts the real-time operation of the embedded systems. For some of the development environments examined, there is a “background telemetry channel (BTC)” [4] which we have investigated as a mechanism to permit reports, in particular failure reports, from the non-scripted tests to be interchanged with the external development environment with no, or minimal, disruption of real time operation.

The following listing, **test macro elements bolded,** can be considered classical (embedded) TDD. The test is for the validation of the response of a finite impulse response filter. Initial values are established, before the test macro **CHECK( )** assert is used to generate a report on the validity of the result from the *FilterASM( )* function under test. For an FIR filter, the output is equal to the jth filter coefficients when an impulse vector ($v_i = 0 \leq I < NEEDED$, $v_j = 1$) is input to the filter.

```
#define NEEDED FIR_length
TEST(FilterASM_impulse, DEVELOPER_TEST) {
  int value, test[NEEDED],coeffs[NEEDED];
  // Impulse response tested
  Set_FilterCoeffs(coeffs, NEEDED);
  for (int i = 0; i < NEEDED; i++) {
     for (int j = 0; j < NEEDED; j++) {
         test[j] = 0; coeffs[j] = j;
     }
     test[i] = 1;
     value = FilterASM(test, coeffs, NEEDED);
     CHECK(value == coeffs[i]);
  }
}
```

In the following listing, some new features of E-TDD are introduced. The real time nature of the embedded environment implies that it is critical that certain functions be guaranteed to perform within a specified time period.

```
void FilterRelease(float* , float*, int);
void FilterASM(float* , float*, int);

TEST(SPEED_REPORT_Filterfloat, CUSTOMER) {
  float *pt, test [NEEDED];
  float coeffs[NEEDED];
  unsigned long int timeRELEASE, timeASM;

  Set_FilterCoeffs(coeffs, NEEDED);
  EstablishTestData(test, NEEDED);
  MEASURE_EXECUTION_TIME(timeRELEASE,
       FilterRelease(test, coeffs, NEEDED));
  MEASURE_EXECUTION_TIME(timeASM,
       FilterASM(test, coeffs, NEEDED));

  for (int i = 0; i < 100; i++) {
    MAXTIME_ASSERT(timeRELEASE,
      FilterASM(test, coeffs, NEEDED));
  }
}
```

Here the **MEASURE_EXECUTION_TIME( )** macro automatically uses the on-chip clock to measure the execution (performance) time of *Function(parameters)*. Whether this execution time meets critical performance characteristics can be checked through the **MAXTIME_ASSERT( )** and **MAXPOWER_ASSERT( )** statements. These are the first of a series of functional and non-functional tests for embedded systems planned for development.

The following test macros demonstrate some specialized embedded system extensions **WatchDataClass( )** and **WATCH_MEMORY_RANGE( )**

```
#define SCALE 0
#define PERIOD 0x2000
#define COUNT 0x2000
  typedef ulong unsigned long int;
  void SetCoreTimerASM(ulong, ulong, ushort);
  TEST(Test_SetCoreTimer, SET_UP) {
    __SaveUserRegAndReset ( );
    WatchDataClass <unsigned long> coretimer_reg(
            4,  pTCNTL, pTPERIOD, pTSCALE, pTCOUNT);
  // Setup expected final memory mapped register values
    ulong expected_value[] = {0x0, PERIOD, SCALE, COUNT};
    WATCH_MEMORY_RANGE(coretimer_reg,
             (SetCoreTimerASM(COUNT, PERIOD, SCALE)),
                                READ_CHECK | WRITE_CHECK);
    __RecoverUserReg( );
    CHECK(coretimer_reg.getReadsWrites( ) == 4);
    ARRAYS_EQUAL(expected_value,
                       coretimer_reg.getFinalValue( ), 4);
  }
```

The **WatchDataClass( )** and **WATCH_MEMORY_RANGE( )** macros to provide the ability to watch (in real time) the performance of specific processor and peripheral memory mapped registers during system initialization and time critical sections of code. These tests should be considered as an embedded extension to, rather than a strict departure from, the “oracle” style of classical TDD since a given computation is run and its output, the number of memory mapped register hardware operations performed and the results of those operations, compared to values predicted in advance. These tests make use of “hardware instruction and data breakpoints” [4] on a running system rather than via “static profiling” on an architectural simulator or “statistical profiling” (occasional snapshots of the program counter) on a running system. The timer and watch-data E-TDD classes are general in concept, but must be specifically implemented using processor resources , and at the same time, not remove resources need for the normal development of code.

The **WatchDataClass( )** class was initially developed for use by an expert (*e.g.* an instructor with very intimate prior knowledge of the system architecture) might write such an EXPERT test to automatically examine whether those developing code for the processor have properly configured the registers of a peripheral correctly. Both register values and register access operations are evaluated *e.g.* checking that registers have been specifically set to the required values rather than left with the default (re-

set) values which "just happen" to be the same as the required values. However, in practice, this test class proved to be unexpectedly much more utilitarian.

- From personal experience, this has proved to be a test that is useful when the developer knows exactly what needs to be done, and how to do it, but gets distracted by an interruption in the middle of the creative process.
- The 3rd year undergraduate class, in the very first assignment after being trained with E-TDD in a hardware-software co-design laboratory, discovered unexpected (undocumented) behaviour of a new processor's core timer resources. Such behaviour could play (could be playing) havoc within industrial products if it remained unrecognized. The students, through the methods of the **WatchDataClass** test class, were able to identify, and then stabilize, the error's behaviour (to prove that it existed) so that the problem could be reported to, and recognized as an issue by, the chip manufacturer [5][6].

3 Experiences and Lessons Learned

The modified test-driven development environment source code for E-TDD has been compiled, linked and downloaded to a number of embedded systems; a single instruction single data (SISD) processor (ADSP-BF533 Blackfin), a single instruction multiple data (SIMD) processor (ADSP-21161) and a variable length instruction word (VLIW) processor (ADSP-TS201 TigerSHARC). Given that the basic TDD code was written in "C++", we were not expecting any compatibility issues across these processors from Analog Devices, nor across processors from other manufacturers. However, we were concerned about code size issues associated with the memory constraints across such a wide range of processors designed for different applications.

A key issue element of the TDD is that all tests be available for running at all times. The non-scripted environment, with its test macros expanded, proved to make use of a larger amount of heap space than had been anticipated, and not necessarily available within the constraints of an embedded system. Three approaches are being used to solve this issue. First a menu driven system has been created to provide automated selective linking to the test library banks. Use of a second C++ heap within external memory for test storage is also being explored. External memory can be two to ten times slower than the internal chip memory because of bandwidth and other issues. However this speed difference is not expected to be critical, since it is the tests themselves, rather than the functions being tested, that are stored in the slower memory. Finally, the report information can be reduced to the passage of tokens requiring minimal memory storage, which are then expanded for use within reports generated by the development environment running on an external work-station.

The current E-TDD prototype provides the ability to set the hardware environment to a known state prior to issuing a series of tests. To capture this functionality, three new hardware oriented TDD procedures were developed

__CaptureKnownState( ) – This procedure is called as the first line of a *main( )* function run on a system that has just been powered up. It automatically captures and saves the "C++" initial environmental setup to a file.
__SaveUserRegAndReset( ) saves the current user processor state, and resets the system to the known state established by the *__CaptureKnownState( )* procedure. *RecoverUserReg( )*, restores the initial user processor state.

Although a fundamental assumption underlying the TDD "oracle" style of testing is to set the system into a known state, the jury is out on the utility of applying this style of testing in an embedded environment. It is true that having the system in a known state before testing prevents many errors. However, testing with a system that is unintentionally in an unknown state does also uncover unexpected system configuration issues. In addition, resetting the system's registers prior to individual tests can be "a highly delicate, and difficult to perform, process" as we discovered recently when adjusting multiple instruction and system caches on the very long instruction word (VLIW), highly parallel, high speed TigerSHARC (ADSP-TS201S) digital signal processor [4]. The resetting of running hardware attached to the embedded system is also not something that can be treated casually.

The long term goal is to use E-TDD within an industrial environment using hybrid communication-media (HCM) processors. However the prototype tool has already proved to offer many immediate advantages to the undergraduate instructor [5][6]. The tests used by the instructor when developing "hands-on" embedded system laboratories make excellent presentation tools to provide the students with an insight of the requirements of all aspects of a hardware software co-design laboratory for both high and low level tasks. The instructor developed tests then form the basis for a bank of "customer" tests that students must satisfy to demonstrate their own design. For tests to have any meaning, their code must be made available for inspection by the tester rather indicating the failure of "secret" test conditions. However, the availability of the provided "high-level" customer test bank meant that many students did not generate their own developer tests during the initial stages of their product development; defeating the purpose of TDD training. We added successful, as well as failed, tests reports; providing the students with an initial comfort zone. These experiences have provided us with sufficient confidence to move to the next stage: using the E-TDD tool within an industrial context of a video project on the hybrid communication-media, dual core, Analog Devices ADSP-BF561 Blackfin processor.

By definition, since tests are developed before the code, there are many error messages from the immediately available "customer" tests covering all stages of the project generated when the tests are first run. This problem was significantly reduced after refactoring the testing environment so that associated groups of tests could be gathered into individual object files within a project library file, and then linked at will. In this manner, a programming pair would be able to activate the "customer" tests for the current and earlier incremental development phases of a project without being over-whelmed by expected failure messages from tests for later project stages.

While we believe that we have made great progress, and that the direction is right, we also believe that we are just scratching the surface of what is required to adapt agile processes into an embedded environment. Of considerable interest is extending the ideas of Feathers [9] on adding tests to "legacy systems", into the environment of testing the custom off the shelf (COTS) software commonly used with embedded systems. Adding nonfunctional unit testing facilities for hard deadlines

e.g.
```
MAXTIME_ASSERT(timeRELEASE,
                       FilterASM(test, coeffs, NEEDED));
```

was relatively straight forward. However adding nonfunctional tests for soft deadlines is more demanding. Soft deadlines can often be examined at the unit level, but the test

is passed / failed over a number of executions rather than a single run. Expression of the pass / failed statement is problematic, as this decision usually involves some level of performance tolerance rather than any strict hard limit.

In addition, refactoring [7] becomes a more complex issue. While embedded systems will be refactored to improve code quality; it is equally likely that they will be refactored to increase performance, or decrease power consumption, or modify other specific and unique, but not code-related characteristics. Again, this takes us beyond the available literature and the definition of the technique. And, in fact, the complex relationship between code quality / maintainability and performance becomes a core concern in many embedded applications, while often ignored as non-critical in many desktop situations.

4 Conclusions

A key identified problem is that embedded systems software development is frequently limited to the compiler environment and (very low level) debugging tools that examine the hardware interface directly. In addition, as embedded systems are real-time and performance constrained, it is difficult to gather extensive execution tracing logs, or capture the entire execution context without completely disrupting the functionality of the system. These problems will become exacerbated as these embedded systems become evermore ubiquitous in nature and require the ability to run for extended periods without experiencing any significant faults.

We believe that agile methodologies, including pair programming [8], will play an important role in developing these zero-defect oriented devices of tomorrow. To-date, we have experimented with test driven development and unit testing. Evidence has been provided to demonstrate that current unit testing tools can be successfully adapted to work in an embedded environment. While the adaptation is not necessarily straightforward; it is viable. Our experiences in adapting these tools also strongly suggests that they need to be extended to explicitly cover the testing of non-functional requirements which often play a critical role in embedded and ubiquitous environments. We have demonstrated these needs by examining the testing of hard deadlines within unit testing. In our future work, we plan to undertake a rigorous experiment to validate the hypothesis that we are already witnessing a more rigorous production process, especially targeted at the elimination of defects during embedded system design and development.

Acknowledgements

Financed under a collaborative research and development research grant involving Analog Devices (Canada) and the Natural Sciences and Engineering Research Council (Canada). MS was awarded the Analog Devices University Ambassadorship for 2002 – 2005. Discussions with A. Geras on agile development were appreciated.

References

1. G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing", National Institute of Standards and Technology Report, 2002.
2. D. Dahlby, "Applying Agile Methods to Embedded Systems Development", *Embedded Software Design Resources.,* Vol.41, pp.101-123, 2004.
3. M. Feathers, "CppUnitLite Source code", http://c2.com/cgi/wiki?CppUnitLite (Dec. 2004).
4. Analog Devices Blackfin and TigerSHARC DSP "User and hardware manuals', http://www.analog.com/dsp (Dec.2004)
5. M. Smith, A. Martin, L. Huang, M. Bariffi, A. Kwan, W. Flaman, A. Geras, J. Miller, "A look at test driven development (TDD) in the embedded environment: Part 1", Circuit Cellar magazine, Vol. 176, pp 34 - 39, March 2005.
6. M. Smith, A. Martin, L. Huang, M. Bariffi, A. Kwan, W. Flaman, A. Geras, J. Miller, "A look at test driven development (TDD) in the embedded environment: Part 2" Circuit Cellar magazine, Vol. 177, pp 60 - 67, April 2005.
7. M. Fowler, "Refactoring: Improving the Design of Existing Code", The Addison-Wesley Object Technology Series, 1999.
8. L. Williams, R.R. Kessler, W. Cunningham, R. Jeffries, "Strengthening the case for pair programming", IEEE Transactions on Software Engineering, pp. 19 – 25, 2000.
9. M. C. Feathers "Working Effectively with Legacy Code" Upper Saddle River: Prentice Hall PTR, 2005.

Multi-criteria Detection of Bad Smells in Code with UTA Method

Bartosz Walter[1] and Błażej Pietrzak[1,2]

[1] Institute of Computing Science, Poznań University of Technology, Poland
{Bartosz.Walter,Blazej.Pietrzak}@cs.put.poznan.pl
[2] Poznań Supercomputing and Networking Center, Poland

Abstract. Bad smells are indicators of inappropriate code design and implementation. They suggest a need for refactoring, i.e. restructuring the program towards better readability, understandability and eligibility for changes. Smells are defined only in terms of general, subjective criteria, which makes them difficult for automatic identification. Existing approaches to smell detection base mainly on human intuition, usually supported by code metrics. Unfortunately, these models do not comprise the full spectrum of possible smell symptoms and still are uncertain. In the paper we propose a multi-criteria approach for detecting smells adopted from UTA method. It learns from programmer's preferences, and then combines the signals coming from different sensors in the code and computes their utility functions. The final result reflects the intensity of an examined smell, which allows the programmer to make a ranking of most onerous odors.

1 Introduction

Software, both while development and maintenance, require health-preserving activities. In Extreme Programming (XP) refactoring is the key practice addressing this issue [1]. The term denotes constant restructuring the program code in order to make it more readable and understandable while preserving its behavior [2].

Applying the treatment is just next to diagnosing the illness. Prior to performing an action, a programmer should find the suspected code and identify the problem to apply an appropriate solution. XP coined the term *code smells* to describe the common structures and constructs in the code that possibly require refactoring. There are over 20 different smells known, but their detection is still based on human intuition and experience. The lack of formal criteria for detecting smells effectively prevents it from being fully automated, which seriously affects the refactoring performance. Existing approaches to smells detection are based on a semi-automated analysis of metrics values, with a programmer making the final assessment and deciding whether the flawed code should be refactored or not.

Unfortunately, metrics fail to uncover many smell symptoms. They just deliver an aggregate measure of the code, whereas smells often reveal also by improper statements, dangerous program behavior or presence of other smells. Besides, metrics represent individual code properties, like cohesion or class size, while ignoring other aspects. Thus, there is a need for a detector that would simultaneously attain two goals: combine different signals indicating the smell presence and let the programmer to adjust the detection process to individual preferences.

In the paper we present a multi-criteria approach to smell detection. Data coming from six sources we identified are quantified and aggregated into a single value using

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 154–161, 2005.
© Springer-Verlag Berlin Heidelberg 2005

a multi-criteria method UTA [3]. The method learns from the programmer's subjective preferences of smells intensity, and then creates a ranking of detected odors, which reflects the preferences. The stored preference model can be reused later.

The paper is structured as follows: In chapter 2 we present the identified sources of smell symptoms. Chapter 3 gives a short description of the UTA method with an example of defining utility functions for every data source for a *Large Class* smell. In chapter 4 we present the evaluation on selected classes taken from Jakarta Tomcat project [13], and conclude with a summary in chapter 5.

2 Data Sources for Smell Detection

In the real world, smells are detected with a single sense only: a nose. In programming, however, code smells usually are a combination of different subtle odors coming from various sources. Some of them, like publicly exposed attributes of a class, can be effectively localized with only one sensor, while others – the vast majority – require smarter detection. This comes from the high-level nature of smells: each of them actually discloses several symptoms. That yields several data sources that indicate smell presence.

As an example, consider the *Data Class* smell [2]. It deals with breaking the encapsulation and limited or no responsibility of a class. It is characterized by three violations that describe not-so-closely related programming faults:

- A class with no functionality (methods) but holding data,
- A public attribute of a class,
- A collection returned by class methods can be modified independently from the owning object.

Each of these violations has different symptoms and requires appropriate detectors: the number of methods in a class can be found with metrics, a public field – with analysis of a syntax tree, and an insecure collection – by running the code.

In general, we identified six distinct sources of data useful for smell detection:

- programmer's intuition and experience,
- metrics values,
- analysis of a source code syntax tree,
- history of changes made in code,
- dynamic behavior of code,
- existence of other smells.

2.1 Intuition and Experience

At present stage the human factor in refactoring is unavoidable in smell detection. While applying refactorings is subject to partial automation, the identification of smells is a creative activity that cannot be fully formalized. According to Fowler "no set of metrics rivals informed human intuition" and the objective is to "give indications that there is trouble that can be solved by a refactoring" [2].

Attempts toward supporting smells detection [4, 5] are based on pointing at most distinct cases, but finally leaving the decision whether to refactor or not to the programmer. Smells are the matter of subjective aesthetics.

2.2 Metrics

Measuring the software code is a common method of taking quick, compact picture of its structure. It is also prevalently and primarily used for detecting smells (e.g. [4]). Most smells affect multiple code measures, and their impact is relatively easy to detect. The term 'metrics driven development', introduced by Simon et al. [5], describes a process, in which the programmer makes choice of code transformations to be done depending on the measurement outcome. Pieces of code with the most deviated metrics are refactored first.

Code metrics, while useful in many cases, cannot sense several smells. It should be noticed, however, that metrics produce mere numbers, not actual suggestions for refactoring. They cannot perform dynamic analysis or look for incorrect constructs. Effectively, doubtless following the metrics may lead to wrong decisions, so they need support from other sensors.

2.3 Syntax Trees

Analysis of abstract syntax trees (AST) is another frequently used source of smells indicators. It allows for finding inappropriate or deprecated statements, improper data types etc. In particular, it provides context information, like inherited members or access modifiers, that otherwise missing could affect the outcome from other sensors.

This method is often combined with metrics, because they are usually computed from the syntax trees. In this case the AST analysis plays an auxiliary role to metrics.

2.4 Code Behavior

There are a few smells that actually result from design flaws, but are difficult to find with a static analysis only. The smells are subject to dynamic analysis, requiring running the code. We propose to utilize unit tests [6] for that purpose. Unit test take a single code unit (usually a class) and examine if its methods react appropriately to the input values. For the sake of smell detection, unit tests simulate conditions in which a smell can be sensed. For example, to detect if a collection is well protected, they attempt to execute all its mutating methods, and check whether the collection actually changes [7]. The tests results indicate the smell presence.

Since the expected outcome is known in advance, the tests could be generated in semi-automated way from templates defined for the given smell (similar approach was applied in for verification of refactorings correctness [8]). In this case the cost of creating the test is significantly reduced. The prototype of such detector we built for *Data Class* smell [7] proved the applicability of the concept.

2.5 Code History

Some smells, like *Shotgun Surgery* or *Divergent Change*, are called maintenance smells [4]. They reveal the design flaws by analyzing how the code reacts to changes. The changes cannot be detected with analysis of a single piece of code, so they compare the subsequent code versions [9]. Configuration management system's entries make a good example for that.

2.6 Presence of Other Smells

Most smells co-exist with others, which means that presence of one smell may suggest presence of related ones. This is due the common origins of some smells: a single code blunder gives rise to many design faults, and the resulting smells are not independent of each other. For example, *Long Methods* often contain *Switch Statements* that decide on the control flow, or have *Long Parameter Lists*. The latter case indicates also the *Feature Envy*, since most of data must be delivered from outside. Therefore, if a method is considered long, the danger of *Long Parameter List* gets higher. However, there is a need for statistical analysis that would confirm or reject this hypothesis.

3 Multi-criteria Model of Smells

Multiple smell sensors give different, partial views of a single odor. To obtain a complex image of the smell, there is a need for a detector that both combines the heterogeneous signals and still preserves the initial programmer preferences concerning their impact on the resulting grade. This chapter illustrates the multi-criteria model of smells based on an example of detecting the *Large Class* [2] smell.

3.1 UTA Method

In general, there are two approaches to create the intensity function: traditional *aggregation* approach and *disaggregation-aggregation* approach [3]. In the traditional aggregation approach the utility function (intensity) is created first *a priori* and then it is evaluated.

In the latter approach the algorithm is split into two phases. The disaggregation phase aims at reconstructing a preference model from the decision maker's limited set of reference alternatives. Next, the aggregation phase, basing on the information induced from the disaggregation one, concludes to construct the measurable utility functions reflecting the verbal criteria used initially.

UTA [3] is a method for ranking a finite set of alternatives, evaluated by the finite set of criteria. The comprehensive preference model used by the method is an additive utility function. The decision maker creates a subjective ranking of small subset of reference alternatives. In UTA, the reference ranking involves two relations: strict preference and indifference. The main goal of the method is the construction of additive utility function compatible with the reference ranking. The construction proceeds according to ordinal regression approach. The utility function is then used on the entire set of alternatives, giving a ranking value to each alternative. The Kendall's coefficient describes the correlation between the reference ranking and the utility function. Values greater than 0.75 indicate that the utility function is compatible with reference ranking obtained from the decision maker. The maximum value for Kendall's coefficient is 1.0.

This method is highly interactive and permits the decision-maker to select the best solution according to his/her subjective point of view. It is also conducive to changes in product preferences and generally copes well with noisy or inconsistent data [10]

UTA is also a good solution in cases where there exist difficulties in obtaining values of the preference model directly from the user.

3.2 Multi-criteria Model for Detecting *Large Class*

According to Fowler, a *Large Class* is a class that is trying to do too much [2]. To be more specific, we decided that following symptoms indicate existence of a *Large Class* smell:

- The class has too much functionality,
- A *Temporary Field* smell [2] occurs in the class,
- *Data Clumps* or *Long Parameter List* smells [2] occur in the class,
- *Inappropriate Intimacy* [2] occurs in the class.

3.2.1 Measuring Class Functionality

The metrics used for measuring functionality offered by a class are presented in Table 1. The recommended threshold values for these metrics are taken from the Nasa Software Assurance Technology Center [11]. We assumed that a class has too much functionality when one of the metrics exceeds the accepted maximum.

Table 1. Metrics used for measuring functionality (source: [11, 12])

Metric	Description	Maximum accepted value
Number Of Methods (NOM)	Number of methods in the class.	20
Weighted Methods per Class (WMC)	Sum of cyclomatic complexities of class methods.	100
Response For Class (RFC)	Number of methods + number of methods called by each of these methods (each method counted once).	100
Coupling Between Objects (CBO)	Number of classes referencing the given class.	5

3.2.2 Detecting *Temporary Fields*

A field is considered temporary when it is set only in certain circumstances [2]. We decided not to detect this smell, because the detector gives many confusing results: e.g. setting methods in a *Data Class* [2] by design use only one field at a time. Such field could be falsely considered as temporary by the detector.

3.2.3 Detecting *Inappropriate Intimacy*

Over-intimate classes are classes that spend too much time delving in each other's private parts [2]. We assume that *Inappropriate Intimacy* is present when:

- There are bi-directional associations between classes, or
- Subclasses know more about their parents than they should.

To detect this smell the detector traverses the Abstract Syntax Tree and counts the number of bi-directional associations for every class (the NOB metric).

3.2.4 Finding *Data Clumps* and *Long Parameter Lists*

The *Data Clumps* smell [2] relates to a group of fields that appear in several classes in the same context. *Long Parameter List* smell [2] is a special case of *Data Clumps*, but dealing with methods: it is a group of method parameters that appears in several methods.

Since this smell requires knowledge of the code semantics (which is unavailable at the code level), we do not detect it.

4 Experimental Evaluation

The proposed approach to detecting *Large Class* [2] smell was evaluated on Jakarta Tomcat 5.5.4 [13] project. Tomcat is well known to the open-source community for its good code quality resulting from constant refactoring.

In order to model the programmer preferences, we needed a learning ranking. It can be either constructed with a real variants or imaginary data, if it only reflects the programmer sense of smell. We chose the latter option, since the criteria for detecting *Large Class* smell had been already defined in Chapter 3. In such cases the learning set is included into the variants set, and then removed from final ranking.

The detection rules are presented in Table 2. Relation *S* denotes the strict preference relation and *I* stands for the indifference relation. We prefer excessive functionality to *Inappropriate Intimacy* smell, because the latter one not necessarily implies the *Large Class*. We also distinguish different aspects of over-functionality. Any metric describing functionality that is exceeding the accepted value is considered to be smelly, but classes with numerous methods stink more.

Table 2. Smell symptoms reference ranking for *Large Class*

Relation	Variant name	NOM	RFC	WMC	CBO	NOB
	Imaginary_NOM	21	21	21	0	0
S						
	Imaginary _CBO	1	1	1	6	0
I						
	Imaginary _RFC	1	101	1	1	0
I						
	Imaginary _WMC	1	1	101	0	0
S						
	Imaginary _NOB	1	2	1	1	1

Table 3 presents top ten smelly classes in the ranking generated by UTA method after analyzing 829 classes of Tomcat code base. The *U* column represents the aggregate utility value.

However, the resulting ranking is not satisfactory. It reflects the weak order of the learning ranking, but there are several doubtful cases. For example, analyzing the final ranking we noticed that the *Request* class smells more than *Digester* class. These classes have similar values on every criterion except for the NOB metric. The *Digester* class have over three times higher value on that criterion and is considered to be less smelly. We looked through the code to assess which class actually smelled more intensively and decided that the *Digester* class was a better candidate for refactoring. The resulting value of utility function for *Request* class was higher

Table 3. Smell intensity ranking

U	Class name (org.apache.*)	NOM	RFC	WMC	CBO	NOB
1.00	*catalina.core.StandardContext*	250	967	353	103	9
0.97	*catalina.connector.Request*	129	426	161	69	4
0.92	*tomcat.util.digester.Digester*	129	387	110	57	13
0.88	*coyote.tomcat4.CoyoteRequest*	113	333	111	62	2
0.82	*jasper.compiler.Parser*	57	407	272	60	1
0.81	*catalina.core.StandardServer*	58	379	224	75	2
0.80	*catalina.core.StandardWrapper*	79	337	109	71	4
0.78	*catalina.servlets.WebdavServlet*	29	441	302	57	0
0.77	*catalina.servlets.DefaultServlet*	36	402	202	62	0
0.76	*catalina.connector.Response*	72	280	109	52	3

than for *Digester* class due to compensation on other criteria. The other interesting aspect of intensity ranking is that the *Parser* class is preferred to the *StandardWrapper* class, although *Parser* class has many complex methods that could be extracted. This would result in higher number of methods (the NOM metric).

Luckily, these drawbacks can be easily fixed. The advantage of UTA method is that it can incrementally refine the preference model until the result is satisfactory. Of course, the modifications can affect the correspondence between learning and final ranking. We decided to extend the reference ranking by adding four relations: *Digester* is preferred to *Request, Request* is preferred to *Parser*, *Parser* is preferred to *StandardWrapper* and the latter one is preferred to the artificial *Imaginary_NOM* variant.

Table 4. Smell intensity ranking for modified reference ranking

U	Class name (org.apache.*)	NOM	RFC	WMC	CBO	NOB
0.96	*catalina.core.StandardContext*	250	967	353	103	9
0.84	*tomcat.util.digester.Digester*	129	387	110	57	13
0.82	*catalina.connector.Request*	129	426	161	69	4
0.74	*coyote.tomcat4.CoyoteRequest*	113	333	111	62	2
0.66	*jasper.compiler.Parser*	57	407	272	60	1
0.63	*catalina.core.StandardWrapper*	79	337	109	71	4
0.62	*catalina.core.StandardServer*	58	379	224	75	2
0.60	*catalina.loader.WebappClassLoader*	50	301	256	63	0
0.60	*catalina.connector.Response*	72	280	109	52	3
0.59	*catalina.servlets.WebdavServlet*	29	441	302	57	0

The intensity ranking for modified references (Table 4) reflects our expectations. The *Inappropriate Intimacy* smell is finally taken into consideration, *Digester* class is preferred to *Request* class and *StandardWrapper* is class preferred to *Parser* class. The ranking is still compatible with the reference ranking at acceptable level (the Kendall's coefficient is equal 0.98).

5 Conclusions

Detecting bad smells in the code is a complex problem. The high-level and vague description given by Fowler makes it difficult to define strict and clear rules of what is and what is not a smell. Programmers can merely observe and measure smell symptoms, which are ambiguous and may lead to wrong conclusions. Effectively it is the programmer who makes the decision basing on intuition and experience.

The multi-criteria model of smells that we proposed, based on the UTA method, responds to the problems mentioned above. It aggregates multiple sources of smell symptoms, not only the metrics, which broadens the spectrum of potential sensors. We determined six such sources, and most of them can be objectively examined and measured.

Every smell is defined by a set of measurable symptoms that suggest its presence. Our model utilizes UTA method to combine these measures and concludes with a ranking of smell intensity. It is important that the method learns from the programmer preferences and constructs the ranking according to them.

The experiment with detecting *Large Class* smell showed that multi-criteria approach allows for more thorough representation of programmer preferences than individual metrics. A combination of various symptoms provides more versatile insight into the nature of a smell. The proposed approach is highly interactive and permits the programmer to define smells according to his/her subjective point of view.

Acknowledgements

This work has been supported by the Polish State Committee for Scientific Research as a part of the research grant KBN/91-0824.

References

1. Beck K.: Extreme Programming Explained. Embrace Change. Addison-Wesley, 2000.
2. Fowler M.: Refactoring. Improving Design of Existing Code. Addison-Wesley, 1999.
3. Jacquet-Lagreze E., Siskos J.: Assessing a Set of Additive Utility Functions for Multicriteria Decision-making, the UTA Method. European Journal on Operational Research, Vol. 10, No. 2, 1982, 151-164.
4. van Emden E., Moonen L.: Java Quality Assurance by Detecting Code Smells. In: Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer Press, 2003.
5. Simon F., Steinbrueckner F., Lewerentz C.: Metrics Based Refactoring. In: Proceedings of CSMR Conference. Lisbon, 2001.
6. JUnit, http://www.junit.org, January 2005.
7. Pietrzak B., Walter B.: Automated Detection of Data Class smell. Inżynieria Oprogramowania. Nowe wyzwania. WNT, 2004, 465-477 (in Polish).
8. Walter B., Pietrzak B.: Automated Generation of Unit Tests for Refactoring. Lecture Notes in Computer Science, Vol. 3092. Springer Verlag, Berlin Heidelberg New York, 2004, 211–214.
9. Ratju D., Ducasse S., Gybra T., Marinescu R.: Using History Information to Improve Design Flaws Detection. In: Proceedings of 8th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, Washington D.C., 2004, 223-232.
10. Beuthe M., Scannella G.: Comparative Analysis of UTA Multicriteria Methods. European Journal of Operational Research, Vol. 130, No. 2, 2001, 246-262.
11. NASA Software Assurance Technology Center: SATC Historical Metrics Database, http://satc.gsfc.nasa.gov/metrics/codemetrics/oo/java/index.html, January 2005.
12. Chidamber S.R., Kemerer C.F.: A Metrics Suite from Object-Oriented Design. IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994, 476-493.
13. The Apache Jakarta Project: Tomcat 5.5.4, http://jakarta.apache.org/tomcat/index.html, January 2005.
14. Pietrzak B.: XSmells: Computer Aided Refactoring of Software. M. Sc Thesis, Poznań University of Technology. Poznań, Poland, 2003.

An Eclipse Plugin to Support Agile Reuse*

Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick

Department of Computer Science, University College Dublin,
Belfield, Dublin 4, Ireland
{frank.mccarey,mel.ocinneide,nick}@ucd.ie

Abstract. Reuse in an Agile context is largely an unexplored research topic. On the surface, these two software engineering techniques would appear to be incompatible due to contradictory principles. For example, Agile components are usually accompanied with little or no support materials, which is likely to hamper their reuse. However we propose that Agile Reuse is possible and indeed advantageous.
We have developed an Eclipse plug-in, named RASCAL, to support Agile Reuse. RASCAL is a recommender agent that infers the need for a reusable component and proactively recommends that component to the developer using a technique consistent with Agile principles. We present the benefits and the challenges encountered when implementing an Agile Reuse tool, paying particular to attention to the XP methodology, and detail our recommendation technique. Our overall results suggest RASCAL is a promising approach for enabling reuse in an Agile environment.

1 Introduction

The demand for organisations to produce new or enhanced software implementations quickly in response to an ever-changing environment has fuelled the use of Agile processes, with Extreme Programming (XP) [1] perhaps the best known and most widely-used Agile methodology. Reuse of software components is another popular software engineering practice. Software reuse has proven to be an effective means of reducing development time and costs whilst benefiting the overall quality of the software [2, 3]. It is not clear however how Reuse and Agile engineering approaches can be carried out in tandem and very little literature exists on this specific issue. It would be desirable to employ Agile principles to produce simple clear software which is easily adaptable to changing requirements while also employing reuse techniques to improve the software quality and reduce development effort, time and cost. We introduce the term *Agile Reuse* to describe such an approach. In practice several inherent difficulties arise when considering the compatibility of Agile and reuse techniques due to differences, often contradictory, in their fundamental principles. For example Agile software tends to be simple and domain specific accompanied with minimal support documentation. Reuse relies on support documentation and favors more generalised components.

* Funding for this research was provided by IRCSET under grant RS/2003/127

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 162–170, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In addition to the above challenges, several other factors hamper reuse independent of the development process used. A mature software development organisation is likely to possess a large, growing repository of components from previous projects. As this repository increases in size, so too does the challenge for developers to remain conversant with all components. Often the effort and time taken to locate and integrate reusable components will be perceived to be costly and to outweigh any potential reuse benefits. Indeed, the reality of strict schedules and tight deadlines may mean a developer has simply not the time to search for components; Frakes *et al.* [4] document other barriers to reuse.

In response to these challenges, various intelligent component retrieval techniques have been developed to assist a developer discover or locate components in an efficient manner [5]. These techniques share a common shortcoming though; the developer must initiate the retrieval process. In our work, we shift the attention from component retrieval to component recommendation. We have developed a recommender tool, named RASCAL, for software components. RASCAL has been developed for two purposes. Firstly we wish to recommend software components that the developer is interested in. Secondly, and more importantly, we wish to recommend useful components which the developer may not be familiar with or aware of. We believe recommendations will assist and encourage developers in making full use of large component repositories in an efficient manner and in turn will help to promote software reuse. Our work is geared towards supporting Agile Reuse, paying particular attention to XP. The goal of RASCAL then is to recommend useful components to a developer in a way which is consistent with the principles of XP development; reusable components currently being developed should not need any additional documentation and reuse of such components should be appealing, straightforward and require little additional effort from the developer.

In this paper we introduce Agile Reuse, present our support tool RASCAL and explain the AI recommendation technique employed. An overview of RASCAL's implementation is given in the following section. In section 3 we detail Agile Reuse; we discuss the benefits of such reuse in an XP context and identify the difficulties of providing an XP tool to support this concept. Two recommendation techniques are discussed in section 4; we then present our hybrid approach followed by a short analysis of the experimental results. In section 5 we review related work in the area of component search, retrieval and recommendation. Finally we discuss how RASCAL can be extended and draw general conclusions in section 6.

2 System Overview

RASCAL is implemented as a plug-in for the Eclipse IDE, as illustrated in figure 1. As a developer is writing code, RASCAL monitors the methods currently invoked and uses this information to recommend a candidate set of methods to this developer. Recommendations are then presented to the developer in the recommendations view at the bottom right hand corner of the IDE window. Currently, RASCAL recommends methods from the Swing and AWT toolkits.

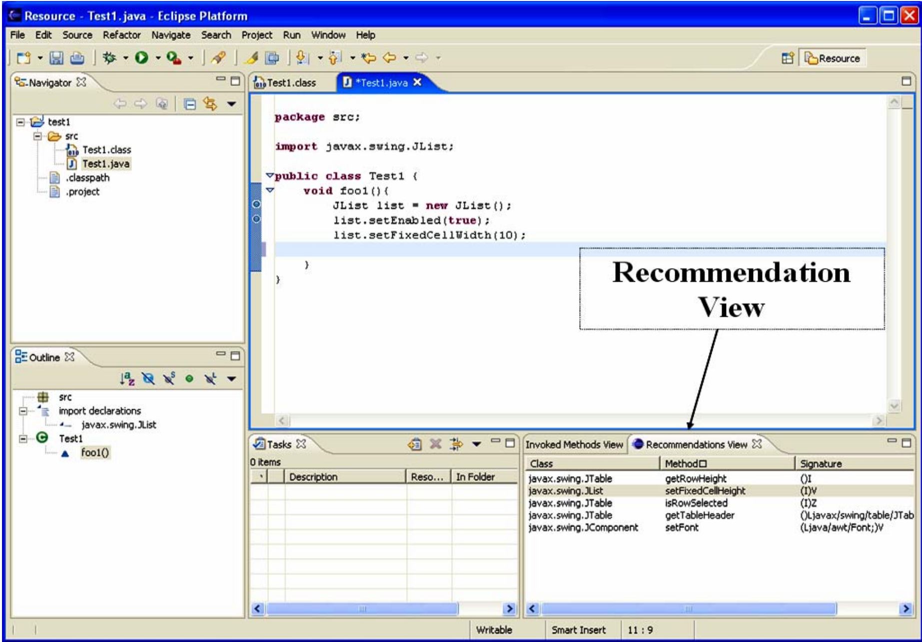


Fig. 1. Eclipse Prototype

Figure 2(a) display a general overview of our system which consists of four components: the *active user*, the *code repository*, the *usage history collector* and the *recommender agent*. The active user can be defined as the developer of the current active class or the current active class itself; the distinction will be clear from the context of the discussion. When monitoring user preferences we only consider the usage history of the current active class and not any other classes this developer may have previously written. The code repository maintains code from all previous projects and all newly created classes will be added to this repository. In our work, we built a code repository using open-source software available from *Sourceforge* [6].

The usage history collector automatically mines the code repository to extract component usage histories for all the stored Java classes. This will need to be done once initially for each class and subsequently when a class is added to the repository. Component usage histories for all the users are then transformed into a user-item preference database, as shown in figure 2(b), which can be used to establish similarities between users. Also, for each individual user we store a list of components based on their actual usage order. The latter information is used for Content-Based filtering as discussed in section 4.1. Finally the recommender agent actively monitors the Java class that the developer is coding, noting in particular the components used in this class. The agent attempts to establish a set of neighbouring users who are similar to the active user by searching the user-item preference database. A set of ordered Java methods is then recommended to the active user based on the neighbouring users.

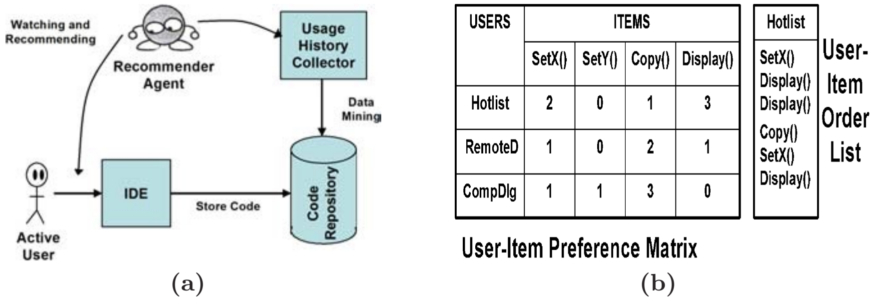


Fig. 2. (a) System Overview (b) Sample user-item database

3 Agile Reuse

Software reuse refers to the use of existing artifacts from previous projects as part of a new development project. Ad hoc reuse has always existed. However as enterprises invest in developing and maintaining large software systems in an increasingly competitive environment, there exists the need for an effective and structured reuse strategy. Ten reusable aspects of any given software project are presented by Frakes *et al.* [4] in their discussion of reuse metrics and models, including requirements and design reuse. In keeping with Agile principles, we are only concerned with *source code* reuse in our present work. Successful reuse has been shown to improve software quality and developer productivity while reducing overall costs [3] and time to market [2].

Despite these desirable advantages several factors hamper reuse as discussed in the introductory section. Factors vary from technical difficulties such as support environments to more pragmatic issues such as managerial and developer attitudes. As reuse becomes more prominent and accepted in industry, systems and tools that aid and support reuse become key aspects in achieving successful reuse of software artifacts [7]. This notion is reflected by the shift in software reuse research from initially focusing on techniques to develop reusable components and component libraries to a focus on supporting reuse through intelligent storage and retrieval strategies [5].

We have mentioned the benefits of reuse-based software development, however, it is unclear how this software engineering approach can be carried in tandem with Agile development. There is an absence of literature and tools to support this concept. It would be desirable to employ Agile principles to produce simple clear software which is easily adaptable to changing requirements while also employing reuse techniques to improve the software quality and reduce development effort, time and cost. We describe such an approach as *Agile Reuse*. Our work focuses on the technical issues involved in implementing this approach; we pay particular attention to Agile Reuse in an XP environment though the issues raised are relevant to all Agile processes. For the following reasons it is the authors position that Agile Reuse using XP is possible and indeed makes sense:

- The simple nature of XP software makes its reuse appealing to developers. Software is produced in small increments and these small units of software may actually be more reusable than software developed under traditional rigorous methodologies.
- XP development advocates quick frequent releases of working code. Reuse will help to achieve this.
- XP developers refactor their code on a regular basis and these very skills are ideal for integrating and tailoring reusable components to match specific needs.

In practice several inherent difficulties arise when considering the compatibility of XP and reuse techniques due to differences, often contradictory, in their fundamental principles. Table 1 on the following page displays a sample of such difficulties that may be encountered and illustrates why providing tool support for reuse in an XP context is difficult. In addition to this we also explain how our support tool, RASCAL, can be employed to address these issues. In the next section we describe how RASCAL automatically retrieves and recommends components, and present experimental results.

4 Recommendations

4.1 Recommendation Technique

Recommendations are produced using a hybrid of two popular filtering techniques, namely collaborative filtering and content-based filtering. The goal of Collaborative Filtering (CF) algorithms is to suggest new items or predict the utility of a certain item for a particular user based on the user's previous preference and the opinions of other like-minded users [8]. CF systems are founded on the belief that users can be clustered. Users in a cluster share preferences and dislikes for particular items and are likely to agree on future items. In the context of this paper, a user can be considered a Java class and an item refers to a software component and more specifically a Java method. Like CF, the goal of Content-Based Filtering (CBF) [9] is to suggest or to predict the utility of certain items for a particular user. CBF recommendations are based solely on an analysis of the items for which the current user has shown preference. Unlike CF, users are assumed to operate independently. Items which correlate closely with the user's preference are likely to be recommended. For example in a news recommender system we would analyse the keywords from the current user's preference to recommend news stories which contain similar keywords; keywords could be "business" or "sport". In our work, instead of analysing keywords or categories we analyse the order in which components are used. In our hybrid recommendation technique we produce our primary recommendation set using CF. We then make use of CBF to order the initial recommendation set. The component which we believe to be most useful to the current developer at this time will appear first in the recommendation set.

4.2 Evaluation

We have conducted experiments to investigate the accuracy of our hybrid algorithm. The component repository used in these experiments contained 1888

Table 1. XP Reuse Challenges and RASCAL

Practice/Belief	Challenge	RASCAL
Working software is the primary measure of success. Less emphasis is placed on comprehensive design or support documentation and quite often the source code is the only available documentation	Reuse relies on support documentation. Locating an undocumented component is problematic, attempting to reuse this component can be daunting and unappealing to a developer.	As the developer writes code our agent is continually searching for reusable components. Newly developed XP components do not need support documentation or commenting for our agent to locate or recommend them. These components just need to have been employed at some stage. Based on the context of such employment, our agent will be able to determine when this component is suitable for recommendation. No additional developer effort is required.
Customer satisfaction is the main priority. This is achieved through early and continuous delivery of working code.	The developer is focused on producing small working units of software as early as possible. If effective reuse support tools do not exist then a developer will perceive the time taken to locate a reusable component as too costly and a burden to achieving their overall goal.	Developers need not initiate the process of component search and retrieval. Instead RASCAL automatically recommends or delivers a suitable component to reuse. We believe component delivery will enhance, promote and increase the feasibility of software reuse to XP developers as they can quickly and easily employ reusable components and thus produce working code quickly.
Simplicity is essential.	Software developed with simplicity in mind will often tend to be very domain specific and perhaps not as reusable as software developed for a more general or abstract task.	We propose that the simplicity of XP components fosters their reuse. RASCAL will help to support and encourage such reuse which otherwise may not have occurred. Despite their simplicity, some components may still be initially challenging to understand and integrate with existing work. RASCAL produces a recommendation for a component to a class by examining similar classes which employ this component. Code snippets taken from the similar classes could prove to be an effective addition to the minimal documentation which often accompanies XP components.

methods from the standard Java Swing library and the Abstract Window Toolkit (AWT). Recommendations were made for a total of 508 Java classes (users)

which invoked on average 60 methods. These classes were taken from 60 GUI applications in SourceForge [6].

For each class several sets of recommendations were made. For example, if a fully developed class used 10 Swing methods, then we removed the 10th method from the class and a recommendation set was produced for the developer based on the preceding 9 methods. Following this recommendation, the 9th method was removed from the class and a new recommendation set was formed for this developer based on the preceding 8 methods. This process was continued until just 1 method remained. Each recommendation set contained a maximum of 5 methods as we believe this to be a sufficient lookahead for a developer. We evaluated the results using *Precision* and *Recall* [10]. Precision represents the probability that a recommended method is relevant. Recall represents the probability that a relevant method will be recommended. Based on our repository of original classes, we also evaluate whether the actual next method a particular developer invoked is in our recommendation set. This is an important evaluation as we wish to recommend methods in an realistic and meaningful order.

4.3 Results

Figure 3 displays the results of our recommendation technique. We also present a baseline result based simply on recommending the five most commonly used methods at each recommendation stage. The recommendation precision is displayed in figure 3(a); the average precision of our technique is 20% which compares favorably with our baseline result. Recall is displayed in figure 3(b); the average recall, based on our recommendation algorithm, is 36%. That is, if we were to recommend ten methods, then on average almost four of those recommended methods would be relevant. Finally, in figure 3(c) we display the likelihood that the next method the developer will actually invoked will be in our recommendation set; on average there is 43% likelihood that it will be. Further to this encouraging result, we see that RASCAL can make reasonably accurate predictions at a relatively early stage in the class's development. For example, when a developer has invoked 20% or less of the total methods she will employ then there is 42% likelihood that RASCAL will correctly recommend the next invocation. We only present the results of our hybrid approach here as we have ascertained that this algorithm leads to the most accurate predictions; [11] details the implementation details, benefits and accuracy of the individual CF and CBF algorithms.

5 Related Work

Much research on tool support for software reuse has concentrated on intelligent search and retrieval techniques which are dependent on developer initiation, for example [5]. However, to effectively and realistically support component reuse it is tremendously important that component retrieval be complemented with unsolicited component delivery/recommendation. One technique to address this issue is *CodeBroker* [12]. CodeBroker infers the need for components and proactively

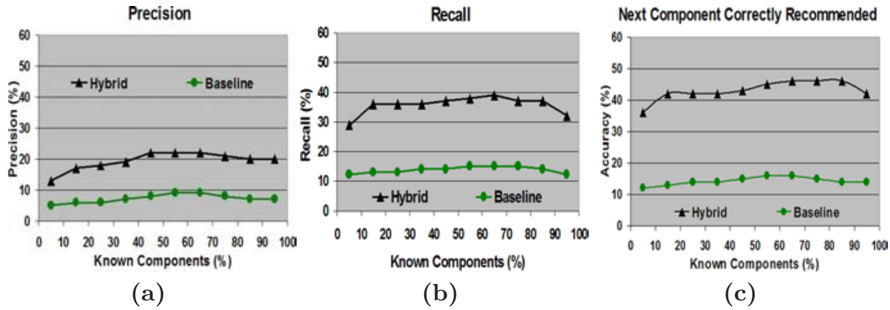


Fig. 3. (a) Precision (b) Recall (c) Next found

recommends components, with examples, that match the inferred needs. The need for a component is inferred by monitoring developer activities, in particular developer comments and method signature. This solution greatly improves on traditional retrieval approaches, but it does not address the requirements of Agile Reuse. The reusable components in the repository must be sufficiently commented to allow matching, this may exclude many components. Developers must actively and correctly comment their code which currently they may not do. Active commenting is an additional strain placed on developers which may make the use of *CodeBroker* less appealing and particularly unsuitable for XP and other Agile methodologies.

Ohsugi *et al.* [13] propose a system to allow users discover useful functions at a low cost in application software such as MS Word and MS Excel for the purpose of improving the user's productivity. For clarity, *Convert Text to Table* or *Insert Picture* are examples of MS Word functions. A set of candidate functions is recommended to the individual, based on the opinions of like-minded users. The technique proposed is an extension of traditional collaborative filtering algorithms used in mainstream recommender systems such as *Amazon*. In our work we apply Ohsugi's principle to a different problem domain, namely reusable software components. Similar to *CodeBroker* [12] our goal is to recommend a set of candidate software components to a developer; however our recommendations are based on the opinions of like-minded developers and not the developer's comments/method signature. Unlike the related works, our technique is specifically designed to assist reuse in an XP environment.

6 Conclusions

In this paper we introduced the concept of Agile Reuse and identified specific issues which hamper such reuse. In addressing these issues, we evaluated collaborative and content-based filtering and found a hybrid approach to be most effective. Our recommendation scheme addresses various shortcomings of previous solutions to the component retrieval problem; user context and problem domain are considered while no additional requirements are placed on the de-

veloper. Opportunities exist to expand RASCAL's scope though. Firstly, we will develop RASCAL into a general recommender capable of recommending various component types. RASCAL will then be extended to allow greater user interaction; for example an accepted recommendation will be automatically added to the user's code. With any unsolicited recommender, delivery is important. Using established industrial links, extensive user trials are planned which we hope will foster a more usable application.

Recommender systems are a powerful technology that can cheaply extract knowledge for a software company from its code repositories and then exploit this knowledge in future developments. We have demonstrated that RASCAL offers real promise for allowing developers discover reusable components and is well suited to Agile development. When little information is known about the user we can nevertheless make reasonably good predictions and future work will likely strengthen recommendations. We believe RASCAL will aid developers whilst improving their productivity, enhance the quality of their code and promoting software reuse.

References

1. Beck, K.: XP explained: embrace change. Addison-Wesley Publishing Co. (2000)
2. Yongbeom, K., Stohr, E.: Software reuse: Survey and research directions. Management Information Systems **14** (1998) 113–147
3. Hooper, J., Chester, R. In: Software Reuse: Guidelines and Methods. Plenum Press, NY (1991)
4. Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Surv. **28** (1996)
5. Yao, H., Etzkorn, L.: Towards a semantic-based approach for software reusable component classification and retrieval. In: Proceedings of the 42nd annual Southeast regional conference, ACM Press (2004) 110–115
6. OSTG: Open source technology group inc (ostg). `http://sourceforge.net`. (2004)
7. Daudjee, K.S., Toptsis, A.A.: A technique for automatically organizing software libraries for software reuse. In: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research, IBM Press (1994) 12
8. Sarwar, B.M., Karypis, G., Konstan, J.A., Reidl, J.: Item-based collaborative filtering recommendation algorithms. In: World Wide Web. (2001) 285–295
9. Oard, D., Marchionini, G.: A conceptual framework for text filtering process. Technical report, University of Maryland, College Park (1996)
10. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison Wesley, New York (1999)
11. McCarey, F., Cinneide, M., Kushmerick, N.: Knowledge reuse for software reuse. In: Submitted to the 17th International Conference on Software Engineering and Knowledge Engineering. (2005)
12. Yunwen, Y., Fischer, G.: Information delivery in support of learning reusable software components on demand. In: Proceedings of the 7th international conference on Intelligent user interfaces, ACM Press (2002) 159–166
13. Ohsugi, N., Monden, A., Matsumoto, K.: A recommendation system for software function discovery. In: Proceedings of the 9th Asia-Pacific SE Conference. (2002)

An Approach for Assessing Suitability of Agile Solutions: A Case Study

Minna Pikkarainen[1] and Ulla Passoja[2]

[1] VTT Technical Research Centre of Finland, P.O. Box 1100, FIN-90571 Oulu, Finland
Minna.Pikkarainen@vtt.fi
[2] Department of Information Processing Science, Hantro Products OY, Nahkatehtaankatu 2, FIN-90100 Oulu, Finland, Research and development Department
Ulla.Passoja@Hantro.com

Abstract. Dynamic market situation and changing customer requirements generate more demands for the product development. Product releases should be developed and managed in short iterations answering to the rapid external changes and keeping up a high quality level. Agile practices (such as the best practices in Extreme Programming and Scrum) offer a great way of monitoring and controlling rapid product development cycles and release development. One problem in product development projects, however, is how to apply agile methods and principles as a part of the complex product development. The purpose of this paper is to describe, how Agile Assessment was conducted in a case company in order to support product development and customer support improvement. During the experiment it was found that Agile Assessment is an efficient method to clarify what agile practices are suitable for the organization's product development and customer co-operation. Another finding was that the use of the best suitable agile practices would improve incremental development monitoring and traceability of requirements.

1 Introduction

Agile SW development and Agile methodologies (e.g. XP [1] or Scrum [2]), used to improve organization or project team ability to manage projects, have been widely discussed in literature [3]. Changing customer requirements, dynamic market situation and new technical challenges generate more demands for the product development [1]. Therefore, functional increments in the faster development cycle can be seen as one solution for the efficient product development. The purpose of Agile Project Management is to develop products based on customer demands, iteratively, simply and using the team experiences [4]. Incremental agile development with iteration planning [4] and Post-Iteration Workshops [5] offers the practices for monitoring the project status and for making sure that the customer requirements are obtained in the project. Many organizations have already utilized the agile methods and principles in their SW development [3, 4]. However, only few organizations can take a specific agile method (e.g. XP) and use it as such. The purpose is rather to apply the best agile practices as a part of the organization's current SW development practices [6]. This would need evaluation which charts the areas of the improvement of the organization's current SW development mapping with the most suitable agile practices. This is the starting point for our study on defining an Agile Assessment Approach. Agile Assessment is SW development evaluation which is done in agile way,

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 171–179, 2005.
© Springer-Verlag Berlin Heidelberg 2005

focusing on finding the most suitable agile practices for the SW development organizations.

The purpose of this paper is to describe the approach to Agile Assessment and practical experiences of Agile Assessment in Hantro. The goal is to find possibilities how to improve Project (PM) and Requirements Management (RM) processes of the case company with agile practices. This paper presents the empirical data from the case study and suggests an approach conducting Agile Assessment.

This paper is composed as follows: section 2 presents Agile Assessment Approach. The third section provides a description of the main results; what benefits the agile methods have and would bring to Hantro's product development, and what kind of experiences we got from Agile Assessment. The last section concludes the paper with final remarks and outlines for future actions.

2 Agile Assessment Approach

The agile community has widely reported the assessments of the agile SW development needs [7, 8]. Existent methods are, however, mainly focused on the metric data based comparison between the traditional and agile SW development. For example, Boehm and Turner present five agility factors (critically, personnel, dynamism, culture and project size) which affect the agile or plan-driven method selection [8]. The Boehm and Turner's model [8] provides a good starting point for agility evaluation but does not address any specifics regarding the application of an agile method. The main challenges for the organizations are still how to tailor agile methods as a part of the product development [6] and how to assess product development agility [8]. Agile Assessment provides a solution for these challenges.

Agile practices described in this paper include both agile principles and methods. The focus of the agile principles is customer satisfaction, rapid answer to changes and close co-operation with motivated business people and programmers. Agile methods (e.g. Scrum and XP) aim at answering the challenge of the rapid development and changing customer demands [8]. Typically, agile methods require close collaboration with the external and internal stakeholders including the processes that employ short iterative life cycles and self organizing teams [8].

The traditional assessments as well as an Agile Assessment is possible based on some well known assessment models (e.g. CMMI [9] or SPICE [10]). In fact, assessment models provide the principal requirement for the assessment planning. These models, however, lack the needed reference information for the agile based SW development efficiency evaluation [11]. One problem is that, even if the traditional assessment is often seen as an opposite to the agile thinking, the agile SW development should be based on the best SW development practices. Simplifying does not mean not documented or not existence processes (e.g. the question of the CMMI 2 level achievement with XP development are recently argued in many studies [12, 13]). Thus, Agile Assessment does not need to be a complex evaluation including the full analysis of CMMI base practices. It should be light-weight and based on agile principles, such as face-to-face communication, rapid feedback to interviewees and organization management and include the simple documentation.

The agile assessment approach is based on well-known SW process improvement paradigms (e.g. QIP[14]) which have a strong theoretical and practical background in

improving the project performance. Agile Assessment includes 1. Goal definition utilizing agile practices; 2. Interview planning based on agile practices; 3. Interviews and improvement analysis; 4. Improvement idea mapping with the best agile practices; 5. Workshops and learning steps (Figure 1).

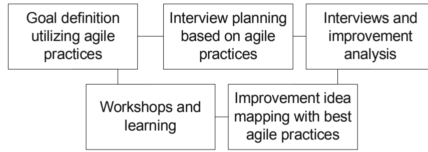


Fig. 1. An Agile Assessment Approach

Agile Assessment is started with the **goal definition utilizing the agile practices** (Figure 2). The first additional task of which, compared to the traditional assessment, is to define what agile methods and principles the organization already uses. Another point is to notice that the planning of Agile Assessment differences if the assessed projects are at high agility level. Maturity of the high agility level project could be difficult to define because the idealized list of the agile practices has not been proven to work. The goal of the agility evaluation is to examine how (and if) the project can be improved applying the agile practices. **The interviews are planned by studying the agile practices** (Figure 1) in the selected process areas (e.g. RM, PM). The idea is to use the agile practices as a basis for the "ideal" Agile Situation definition. **Interviews and improvement analysis** (Figure 1) are mainly developed using the traditional interviewing techniques. However, on Agile Assessment, the aim is to discuss possible agile practices and their using possibilities in the company. The **"**ideal" Agile Situation definition is used as background information in the improvement analysis. The purpose of the Agile Assessment is to find the improvement ideas and analyze how the projects could be improved by applying suitable agile methods for the assessed organization's current needs (Figure 1).

Table 1. PM and RM challenges that can be answered with Agile PM and RM [4]

Challenges	Agile Project Management Answers
Requirement changes	Priorisation in iteration planning
Unpredictable effort	Short scheduled iterations and technical excellence
Continuous innovation	If you want to innovate then iterate
Changing technology and architecture	Short iterations, innovative team, dynamic architecture, technical excellence
Complex documentation	Simplicity
Customer interface	Deliver customer value, deliver early benefits
Requirements traceability	Requirement status discussion in the end-of-iteration reviews
Risk management	Risk discussions in the end-of-iteration reviews
Project visibility	Status and requirement discussion in the end-of-iteration reviews

The analysis of most suitable agile practices resolving the problems requires much background information about the benefits of agile practices in different situations. In table 1, an example of the project and requirement management challenge "agile tool box" for the agile analysis is described.

The main results of Agile Assessment are defined in the **workshops** (Figure 1) where the best suitable agile practices are analyzed. The workshops can be prepared

presenting possible problem solution alternatives in the organization's previous process descriptions and guidelines based on the agile assessment results. After Agile Assessment, the improvements and defined agile practices are priorised and further analysed in the internal meetings. Suitable practices are piloted in projects the selection of which could be based on projects' agility.

3 Experiences and Key Findings

Hantro develops video technology for mobile devices to enable multimedia applications. Typically, product development to mobile devices includes the changing of operational environment and fast time to market that could be well supported with "agile thinking". Hantro has been developing its embedded product development processes continuously for three years. Recently, agile principles and methods have been taken actively into account in the development work.

3.1 Focus of Agile Assessment

The purpose of Agile Assessment was to objectively bring out the most critical improvements in Hantro product development and customer integration projects. Another aim was to analyze, how the improvement ideas could be supported with the agile methods and principles. The scope of Agile Assessment was to evaluate the PM and RM in product development and customer integration projects and to define which agile practices would best support the Hantro RM and PM work in practice. Assessment was made for three projects which had a different agility level (Figure 2). In the evaluation, the Boehm and Turner agility dimensions were tailored based on our needs[1].

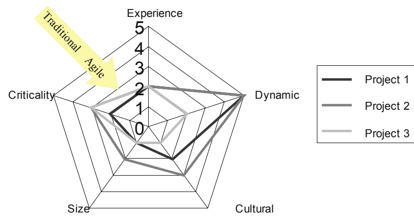


Fig. 2. The Agility of the Hantro projects according to the tailored agility dimensions [8]

One focus in Hantro Agile Assessment was to clarify the most suitable agile practices in diverse projects. Project 1 was customer integration project, project 2 developed an embedded product with new technologies, and project 3 developed product parallel with integration work for several customers. Project 1 did not use any specific agile methods. However, it used the simplified development and rapid short release iterations for customers. Project 2 had successfully used Rapid 7 [15] method

[1] Dynamic[many changes 1p-Stabile 5p], Size[Under 10/1, 10-20/2p, 20-50/3p, 50-100/4p, more 5p],Critically [Not critical 1p-Critical for human life/5p] Experience [more than 7 years/1p,3-7 years/2p, 2-3 Years/3p, 1-2 Years/4p, trainers/5p],Cultural [Based on agile principles 1p- traditional 5p]).

in specification and documentation work. The project was quite stabile and its agility was quite low. According to this evaluation, the culture and amount of change in Project 3 supported the agile principles best. (Figure 2). Information of the project's agility (Figure 2) can be utilized when selecting the projects for agile assessment and for the most suitable agile practice piloting. Agile Assessment was planned by comparing the agile and CMMI practices in PM and RM (e.g. Table 2).

Table 2. An example of the agile practice and CMMI analysis

CMMI	Agile principles	Agile Practices based on literature
SG 1 Manage requirements	Customer satisfaction,	User stories definition and analysis for subsequent iteration. Product and sprint Backlog requirements analysis.[1, 2]
SG 1 Establish estimates in PM	Short iterations	Project planning including the working tasks and schedule estimations for subsequent iteration.[1, 2]
SG 2 Develop a Project Plan	Short iterations	Schedule, risks, resources, needed knowledge and skills definition for the subsequent iteration.[1, 2]
SG 3 Obtain commitment to the Plan	Face-to-Face communication	Iteration planning and reviewing together with the relevant stakeholders [15]
SG 2 Monitor project against plan	Rapid answer to change	Checking of the previous iteration status in Mobile-D planning days [16].Information Radiators (IR) [1] in the task definition and monitoring

According to this analysis detailed requirement definition for each increment as well as the product backlog lists are the key activities in the agile RM. Agile PM includes, for instance, the increment planning and face-to-face communication with developers (e.g. Scrum daily meetings). The analysis of the agile practices (Table 2) worked as a basis for the Agile Assessment question (Table 3) creation and result analysis.

Table 3. An example of Agile Assessment questions

Agile Practices based on literature	Questions
Requirements and change requests definition for subsequent iteration. Product and sprint Backlogs	1. How iterative is the product development? 2. How are the iterations and release development planned? 3. How is the requirement changes analysed? 4. How are the requirements and change requests defined for each release?
Short iterations. Project planning for subsequent iteration. Requirement status review for the previous iteration	5. How often are the releases delivered? 6. How are the project tasks and effort estimations defined? 7. How are the tasks for each release selected? 8. How is the requirement traceability ensured? 9. Who participates in the task and requirement definition?

Agile Assessment included five interviews for project managers, developers and product manager. Hantro’s quality management participated in all interviews and overall agile analysis. In the assessment result analysis, the improvement ideas were evaluated based on defined Agile RM and PM situation. In this phase, the agile solutions for the improvements were defined, lightly documented and analyzed in the workshop together with the interviewees and management. The main results of the analysis were the definition of the best agile practices that would be suitable and useful for Hantro product development and customer integration projects.

As a result of Agile Assessment, it was found that Agile Assessment offered the agile based solutions for improving the organization's PM and RM. It took about one

month working effort (two weeks from Hantro and two weeks from VTT) but gave the objective ideas (both agile and non-agile solutions) how to start to improve product development and customer integration work. After Agile Assessment, the improvement ideas and defined agile practices were priorised and further analysed at Hantro's process improvement meeting. The practices will be tailored to suit Hantro processes and piloted in suitable projects. Their deployment will be supported by quality management. New practices and their benefits will be discussed in post-iteration workshops [5]. If new practices improve PM and RM, they will be deployed in other suitable projects. Later, when the new practices have been taken into use, new Agile Assessment to relevant process areas will be organized.

In future, it would be interest to repeat the assessment resulting in the difference on improvements that happened so far. Other experiences of the agile assessment approach creation are out of the scope of this paper. It will be, however, discussed and analysed as a part of the future Agile Assessment research.

3.2 Findings of Agile Assessment at Hantro

Hantro's research and development culture supports agile principles. Hantro has technically experienced development employees, who all work at the same site, when face-to-face communication is the preferred way of communication. Project teams are relatively small in size and team members and teams seem to be co-operative. Specification workshops, using Rapid7 method, have improved the communication between the HW and SW teams in product development (Table 4). Close cooperation with the business department and sales ensures taking the latest needs for change into account. Working releases are delivered to customers in short cycles. Requirements are specified on a proper level in the beginning of the project. Change requests are handled with change management flow. Dynamic architecture facilitates concurrent product development and integration work. Hantro has also effective quality monitoring practices. Quality audits before releases and at the end of the projects ensure the quality of the delivered releases and final products. Key findings of the agile assessment were improvement ideas that could be supported with certain agile practices. At Hantro, PM and RM are mostly done with traditional methods, where selected agile practices could bring some benefits. Improvement ideas were focused on incremental project monitoring, risk management in co-operation with customer, and requirement traceability (Table 4).

Table 4. An example of the agile practice findings at Hantro

Improvements	Expected benefits of the agile practices for Hantro
Release planning in incremental development	Dynamic release plan (which is updated in co-operation with the product management, programmers and customers for the subsequent release iteration) answers in the challenge of the rapid change demands.
Project and risk management in incremental work	The iteration planning and post-iteration workshop meetings would make the project monitoring more effective. Risk identification and monitoring for each development increment would improve the risk management.
Requirement visibility and traceability	Use of the product backlog list would improve the requirement visibility (requirement status easy to find, requirements in the same place). Requirements definition and priorisation for subsequent iteration and requirement status checking in post-iteration workshop would strengthen the requirements traceability.
Continue documentation team work	Specification workshops with Rapid7 method has already improved the communication between the both HW, SW and system and testing groups in embedded product development

The first improvement idea was to systematize the iterative release planning. For example, in XP [23] a customer defines user stories that are built for the next release. Programmers estimate the task efforts, communicate with the customer about technical risks and measure the progress to provide the customer a budget. The release plan is updated at the beginning of each increment. In Hantro's customer projects, releases were planned as a part of the contract. In the evaluated projects, release plan could not, however, answer to the challenges that new customers and technical requirement changes bring to projects. The solution proposal would be to create a product release plan which is updated in the increment planning meetings where developers would estimate the task effort for the changes and discuss the technical risks. (Table 4).

The second improvement idea was to emphasize iterative project monitoring and risk management. It could be done using, for example, the Mobile-D [16] where the status of the requirements, tasks, effort estimates and risks are discussed for the subsequent iteration in the iteration planning meetings. The third improvement idea was to improve the visibility and traceability of requirements. This could be done using iteration backlog lists in focusing the requirements for the subsequent release iteration or to checking the requirement status in the iteration planning meeting.

Agile assessment provided an analysis of what agile practices could fit the Hantro environment. At the moment, Rapid 7 method for the embedded product specification has already been successfully used at Hantro. Unfortunately, empirical data on the other agile practices actually used in Hantro does not yet exist. Agile analysis was, however, the basis for process description updates and most of the proposed solutions will be tailored best to suit Hantro’s working environment and current processes. According to an initial tailor plan, release plan will be a part of the project plan and it will be updated in connection with increment meetings. Increment reviews will be combined to milestone checks, where product management and possible external customer accept or give feedback about the release. To systematize incremental project management increment planning and post-iteration workshops will be combined and held internally after increment reviews. In addition, there will still be periodic project meetings and status reports. Requirements will be listed in an RM tool, where one parameter is a planned release to help incremental planning and status traceability. Implementation and piloting of the best agile practices in Hantro projects will require continuous support from quality management.

4 Conclusions and Further Study

Agile methods and principles offer tools to improve product management and development activities [1]. The purpose of agile practices is to answer the challenge of the dynamic market situation and late changes [8, 17]. The best suitable agile practices can be used to improve the workload estimations, product validation cycles and requirement change management time [1]. Only a few organizations can, however, use a certain agile method such as Scrum or XP as such. Rather, they may deploy the most suitable agile practices as a part of the organization’s existing product development practices. An Agile Assessment is an approach that helps organizations to find the best suitable agile practices to improve a specific aspect of the SW development work. It does not take much work effort but provides an objective viewpoint and understanding of the needed improvements and available agile solutions.

The purpose of this paper was to introduce an Agile Assessment approach and to describe experiences and key findings from Agile Assessment work at Hantro. Based on experiences of the case study, the Agile Assessment is an efficient and objective way to find what agile practices would improve efficiency of working methods and what agile practices would fit the organizational culture and current working methods and environment. The findings of the case study support the assumption that the use of agile practices would improve project monitoring, risk management and requirement traceability in the incremental product development. The empirical data from the case study shows that the process assessment can be done effectively using close communication, rapid feedback, and simple documentation.

However, there is no experience available yet about validating the improvements in the assessed organization. Agile Assessment should also be researched using a deeper literature analysis of agile methods' view of agile suitability in different SW development processes and with more case study experiences from multiple organizations. One reason for this is that the existing study lacks important aspects of defining, how the specific agile methods and practices could be tailored in different product development contexts. Also, it has not been defined earlier, how the assessment implementation can be done in agile way using the agile principles.

Acknowledgements

The research was conducted in co-operation with VTT Technical Research Centre of Finland and Hantro Products Oy in Agile ITEA project funded by National Technology Agency of Finland. Hantro is the leading provider of video enabling technology for mobile devices. It develops video solutions in both ASIC and SW implementation. Acknowledgements to Jarkko Nisula and Harri Hyväri of Hantro and Outi Salo and Pekka Abrahamsson of VTT of their support and valuable comments.

References

1. K. Beck, *Extreme Programming Explained: Embrace Change*: Addison Wesley Longman, Inc., 2000.
2. K. Schwaber and M. Beedle, *Agile Software Development With Scrum*. Upper Saddle River, NJ: Prentice-Hall, 2002.
3. B. Greene, "Agile Methods Applied to Embedded Firmware Development," presented at Agile Development Conference, Salt-Lake city, 2004.
4. J. Highsmith, *Agile Project Management, Creating innovative products*: Addison-Wesley, 2004.
5. O. Salo, K. Kolehmainen, P. Kyllönen, J. Löthman, S. Salmijärvi, and P. Abrahamsson, "Self-Adaptability of Agile Software Processes: A Case Study on Post-Iteration Workhops," presented at 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), Garmisch-Partenkirchen, Germany, 2004.
6. P. Manhart and K. Schneider, "Breaking the Ice for Agile Development of Embedded Software: An Industry Experience report," presented at 26 th International Conference of Software Engineering, 2004.
7. P. Lappo and C. T. A. Henry, "Assessing Agility," presented at Extreme programming and Agile Processes in Software Engineering, Germany, 2004.
8. B. Boehm and R. Turner, "Balancing Agility and Discipline," in *Balancing Agility and Discipline -A Guide for the Perplexed*: Addison Wesley, 2003.

9. P. t. CMMI, "Capability Maturity Model Integration (CMMI)," CMU/SEI-2002-TR-002 ed: Software Engineering Institute, 2001.
10. R. m. o. S. C. T. ISO, "ISO TR 15504, Information technology-software process assessment -Part 2, A Reference model of Software Capability." Geneva: International organisation for standardisation, 1997.
11. P. Kettunen and M. Laanti, "How to steer an embedded software project: tactics for selecting the software process model," *Information and Software Technology*, 2004.
12. C. Vriens, "Certifying for CMM Level 2 and ISO9001 with XP@Scrum," presented at Agile Development Conference, 2003.
13. M. C. Paulk, "Extreme Programming from a CMM Perspective," *Software*, vol. 18, pp. 19-26, 2001.
14. S. H. Kan, V. R. Basili, and L. N. Shapiro, "Software Quality: An Overview from the perspective of total quality management," *IBM Systems Journal*, vol. 33, 1994.
15. R. Kylmäkoski, "Efficient Authoring of Software documentation Using RaPiD7," presented at ICSE 2003, 2003.
16. P. Abrahamsson, A. Hanhineva, H. Hulkko, T. Ihme, J. J., M. Korkala, J. Koskela, P. Kyllönen, and O. Salo, "Mobile-D: An Agile Approach for Mobile Application Development," presented at 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), Vancouver, British Columbia, Canada, 2004.
17. P. Gould, "What is agility? [agile manufacturing]," *Manufacturing Engineer*, vol. 76, pp. Pages:28 - 31, 1997.

XP Expanded: Distributed Extreme Programming

Keith Braithwaite and Tim Joyce

WDS Global, Forelle House, Marshes End, Upton Road, Poole
{keith.braithwaite,tim.joyce}@wdsglobal.com

Abstract. Colocation has come to be seen as a necessary precondition for obtaining the majority of the benefits of XP. Without colocation teams expect to struggle, to compromise and to trade off the benefits of XP *vs* the benefits of distributed development. We have found that you can stay true to the principles and not compromise the practices of XP in a distributed environment. Thus, business can realize both the benefits of distributed and of truly agile development.

Keywords: Agile, XP, Extreme Programming, Scrum, distributed, multi-site, outsourcing

1 Introduction

The small but growing literature on "Distributed Agile development" takes a largely pessimistic view. Often, writers assume that having the members of a team distributed widely in space (and/or time) would deal a fatal blow to the communication mechanisms upon which agile development relies. They then infer that, should one attempt to use an Agile method (for example, XP) in a distributed environment, those practices which embody the communication value in a colocated setting would be near–fatally compromised. Thus that XP itself would be compromised and various additional practices of a questionable nature (often forms of documentation, or of process automation) would need to be introduced. So it is believed that much of the benefit of "agile" development would be lost, and that winning much of the rest would be very challenging.

Note the subjunctive mood throughout the previous paragraph. Much of the Distributed Agile literature is speculative. The few reports of distributed agile development in practice are stories of hedging, compromise, and of profoundly mixed results. These are valuable data points, but do not address the question: what happens if a team distributed in space and time works as fully as possible in alignment with the principles of, say, XP?

Our observation is that such a team can succeed, and win for its business sponsors both the advantages of agile development and of distributed working.

2 Distributed Agile Development

Various terms, such as Distributed Agile Development and Distributed Extreme Programming are currently terms used to refer to a variety of process and prac-

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 180–188, 2005.
© Springer-Verlag Berlin Heidelberg 2005

tices associated with non-colocated development teams. When we need to refer to teams that are not colocated we prefer to say "cross-site".

We distinguish these three general cases for non-colocated, Agile aligned development:

Agile Outsourcing (AO): Where an agile team is created at an appropriately low cost offshore location. Requirements are generated onshore, and communicated offshore using documents, people and tests. There may be some code sharing between the onshore and offshore team, but not shared ownership as commonly understood. This approach has a degree of popularity and has been widely discussed, notably in [10] and [14].

Agile Dispersed Development (ADD): As practiced by much of the Open Source community and some commercial companies [6]. Developers tend to be physically alone, but connected through a variety of communication channels. Practices such as frequent releases and continuous integration are employed, Pair Programming[1] and other team based activities are not (or only in a very limited form). Because of this, aspects of shared code ownership are often. In the open source case, this results in practices such as Benign Dictator, and Trusted Lieutenant.

Distributed Agile Development (DAD): Customers are distributed. One development team is distributed evenly over several sites to remain close to the customers. Rich, high density communication ensures that Agile principles and practices are not compromised, locally or globally.

2.1 Wireless Data Services Case

WDS is a global business providing various services to mobile telephone network operators and handset manufacturers. These include web based software for self-serve device management. At the end of 2003, core services were developed and deployed as APIs by a UK based team using XP. In each region a (non XP) team would use these internal APIs to deliver on locally generated requirements.

At the beginning of 2004, we brought all developers together in the UK for an XP and Java "boot camp". This time was also used to establish a single global team. Developers were then dispersed to three sites (UK, Seattle, Singapore), and Distributed Extreme Programming (DXP) begun in April 2004. The business considers the change to be successful.

3 How to Remain Extreme Around the World

Given that there is a business need to have developers around the world work together, how can agility be preserved?

[1] We distinguish the names of practices by using this face

At one level, the answer is quite simple: maintain a commitment to the value judgments that characterise the core of all agile methods, the Agile Manifesto [2]. Some writers take it that in applying agile approaches to distributed working must require sophisticated tools and complicated process models [12]. This seems at odds with the spirit of the manifesto. Others report some success on small–scale pilot projects using a much more direct approach [9]. We have shown that the direct application of XP to full–scale commercial development can be successful.

The implementation of the Agile Manifesto that we prefer is XP. As described in [1], this uses various Primary and Corollary Practices to embody Principles which manifest Values. We also apply many project management ideas adopted from Scrum [3]. Of the XP Values, we find that in the DXP case, Communication and Respect are especially emphasized. The key problems in DXP are to maintain sufficiently rich communication, and a sufficient level of respect, between colleagues separated widely in time and space.

Our experience is that these problems are soluble in a way wholly aligned with Agile principles. We consider that the defining characteristics of DXP are the use of: One Team, Balanced Sites and Distributed Standup, One Team, One Codebase.

4 Can Distributed Development Be Truly Agile?

The question is rather: can successful Distributed Development be anything other than Agile?

4.1 Traditional Distributed Development

We did not investigate the literature particularly thoroughly before experimenting with DXP [2]. Instead, we expressed the XP value of Courage. Confident that the principles of Agile development, and the practices of XP and Scrum, were fundamentally sound we started with the obvious first steps to implement DXP.

Subsequently we read Carmel [4]. The most interesting feature of this work (which predates the Agile revival) is that most of the content is aimed at convincing a plan–following, top–down, command–and–control manager of the value judgments captured in the Agile Manifesto. Carmel identifies "loss of teamness" and "loss of communication richness" as two of the five centrifugal forces that will damage a global team. His solutions revolve around such items as "lateral communication" (communication between co–workers across the width of the organization chart), encouraging a common team culture, building trust through face–to–face meetings, and so on.

It's our claim that almost all of the best practices presented by Carmel for building dispersed traditional development teams will be second nature to an organisation practiced at XP.

[2] Perhaps if we had, would have been scared off

4.2 The Colocation Shibboleth

Having all team members in one room is a defining characteristic of default XP. Beck gives Sit Together as the first Primary Practice of XP. However, he also states clearly that "[...] teams can be distributed and do XP".

By definition, distributed teams cannot achieve Sit Together as a whole, although each regional group can and should be colocated itself (as developers in WDS's regions are). However, if we look beyond the one–room practice to the value of Communication it manifests we can see the possibility of expressing that value in other ways.

We submit that the injunction to put everyone in one room is an absolutely necessary rule to apply when *introducing* XP. Non-agile development practice often trains developers to be solitary, uncommunicative and non collaborative. Together with pairing, Sit Together is very effective at breaking those habits—as required to roll out the rest of XP. But, if a body of developers are available already trained to work gregariously, to communicate as much as possible, to seek out collaborators, then perhaps the need for colocation as the prime mechanism to manifest the Communication value is weakened.

Beck's discussion describes the XP practices as *theories*, with attached predictions. The Sit Together theory predicts that "[...] the more face time you have, the more humane and productive the project." We would generalise this to state that the more, more rich, communication you have, the more humane and productive the project. Face time is still much preferred, but it turns out not to be a *uniquely* valuable medium.

5 Practices for DXP

We have identified a number of practices for cross-site development, with particular emphasis on agile techniques. We present them here as candidate patterns in something like Portland Form [5]. The patterns are organised by thematic area, and ordered by significance within an area—as indicated by the number of *'s suffixed to the name.

5.1 People

These practices relate mostly to human interactions, the most difficult and also most crucial aspect of DXP. Each site implementing these practices needs people with experience of co-located XP.

One Team ***

Business needs lead to development resources distributed widely in space and time. Communication between members is compromised. Trust and cooperation can break down.

Therefore: Maintain as far as possible a singe team identity across all locations. Encourage non-business communication, encourage any activity that lets team

members share a joke or a cultural reference. Cherish every successful interaction. Let people play. A high level trust is maintained, resulting in fewer conflicts. When work related conflicts arise, a joke can defuse the tension.

Relates to: Kickoff; Multiple Communication Modes; One Team, One Codebase; One Team, One Build
As seen in: Seems to be novel as stated, we'd love to learn otherwise

Balanced Sites ***

Team members at one location are sometimes blocked while waiting for actions or decisions taken at another site. This creates resentment on both sides; the dependent site resents the productivity impact and the loss of decision making power; the depended site resents the interruption of thought and activity.
Therefore: Make all sites equal in skill and numbers, and empowered to take any decision, so that inter–site dependencies are minimized. There is no delay between when a decision or action is required and when it is performed, maintaining flow. All team members feel fully engaged.

Relates to: One Team; Distributed Standup (even though dependencies are minimized, force communication anyway); Ambassador
As seen in: Seems to be novel, as described. We'd love to learn otherwise

Ambassador **

Members of a team in one location find it hard to understand the point of view of members in another location. Trust and cooperation break down, it is hard for one local group to work effectively with another.
Therefore: Send an ambassador from one region to another, for an extended period. Such a local representative can interpret the communications of the remote group, demonstrate that "they" are just like "us", and influence locally on behalf of the remote group when required. Ambassadors also carry business domain knowledge between sites.

Relates to: Visits Build Trust; One Team; Balanced Sites; Multiple Communication Modes
As seen in: [8], [11], [7]

Visits Build Trust **

Team members find it hard to have faith in the good intentions of remove colleagues. Blamestorming replaces collaboration, fingerpointing replaces problem solving.
Therefore: Have team members rotate through locations continually. Always have at least one team member away from their home location. Trust in a team member currently remote can be maintained, based on the experience of having worked with them colocated in the past.

Relates to: AmbassadorOne Team, Multiple Communication Modes
As seen in: [4], [11], [8]

Kickoff *

A new project is to start. All team members involved must synchronise their ideas about it.
Therefore: Bring everyone involved in the project together in one place at the same time. Future distributed working is informed by a cohesive view of the project, and secure interpersonal relationships, formed while the advantages of Sit Together were available.

Relates to: One Team; Multiple Communication Modes
As seen in: [4], [8], [11]

5.2 Communication

These patterns consider communication, the lifeblood of agile development and the greatest challenge for DXP.

Distributed Standup ***

Team members remote from one another cannot easily see each other's story board, overhear technical discussions, share in resolving issues. Remote members' idea of the state of the team fall out of sync, damaging the cohesion of the team.
Therefore: Have a video conference session running whenever possible, but at least once a day, every day for each pair of sites adjacent in time. Force an overlap if required. No member can forget that the remote members have a stake, status is shared (perhaps transitively).

Relates to: One Team; Balanced Sites
As seen in: [**?**]

Multiple Communication Modes ***

The members of a team cannot be colocated. Face–to–face communication (explicit and "overhearing") cannot be used to maintain tacit knowledge. Many different kinds of knowledge must be shared, often during sharply time–limited handover slots.
Therefore: Provide team members with as many communication media as possible. At least these: individual and conference telephone, teleconference, video conference, email, IM, wiki, VNC. Communication is fostered greatly, and many different modes of communication can be applied in parallel. A good conversation to hear at a videoconference standup meeting would be: *site 1: We had an idea for that problem, I've just jabbered you the URL for the wiki page that discusses our example code, see what you think. Site 2: Great! Let's remote–pair on this tomorrow.*

Relates to: One Team; Wiki as Shared Location; Remote Pair Ambassador; Code is Communication
As seen in: [8], [7], [4]

Wiki as Shared Location **

Team members can meet neither at the same time nor at the same place. Communication has to be both over low bandwidth channels and asynchronous. Members do not feel members of “one team” due to disjointed communication.
Therefore: Use a wiki. A shared virtual place is created where notices may be posted, asynchronous conversations take place in a persistent form, and a feeling of community fostered.

Relates to: Multiple Communication Modes; One Team; Code is Communication is dual to this practice
As seen in: Many sources mention team wiki’s, but the notion of wiki as shared virtual location is not explicit. The walls of public lavatories.

Remote Pair **

Developers that need to Pair are remote. Code changes need to be shared. A familiar shared environment needs to be available to allow pair programming between sites.
Therefore: Establish an easy-to-start environment with rich communication (video, text and sound) and a shared development tool (use VNC or similar to share an IDE). Agree a regular time when remote pairing will occur. Complex, code-level decisions and communication will occur in a familiar way.

Relates to: One Team, One Codebase; Many Communication Modes; Code is Communication
As seen in: [12], many informal mentions on newsgroups, etc.

5.3 Code

These patterns address what is perhaps the easiest aspect of DXP, the technical.

One Team, One Codebase ***

Widely separated team members need to maintain a common identity as technical problem solvers. They need to share rights and responsibilities toward each others’ work, just as colocated workers do.
Therefore: Have all team members everywhere use a single, shared codebase. Technical problems and their solutions are shared. The whole team always has a common point of reference.

Relates to: One Team; Code is Communication
As seen in: Seems to be novel as stated, we’d love to learn otherwise. This practice largely opposed to much of the advice given in [13].

Functional Tests Capture Requirements **

Requirements need to be transmitted from one site to another. A great deal of time and energy would be consumed to make requirements documents work as a medium.

Therefore: Use failing functional tests to express the required functionality. The requirement is expressed unambiguously.

Relates to: Code is Communication; Tests Announce Intention
As seen in: [8]

One Team, One Build **

All team members need to share responsibility for maintaining all code in a working state. With multiple integration machines, inevitable environment skew can hide the reasons for build failure. Arguments break out between sites as to whether a break is because of "your build machine" or "our code", destroying shared responsibility and respect.
Therefore: Have a single build server. Set up an RSS feed so that each site can hear the build passing or failing. The build status of the global codebase is a single boolean value.

Relates to: One Team, One Team, One Codebase
As seen in: Seems to be novel as stated, we'd love to learn otherwise. This contradicts the advice given in [13] for large teams: §3.4 states "Each [sub]team can and should set up their own automated integration server"

Code is Communication **

Colleagues far apart cannot discuss technical issues, design ideas, requirements face–to–face. This can threaten the conceptual integrity of a code base (and team).
Therefore: Use the code base as a communications medium between sites. Converse with remote colleagues via the codebase. Express problems as failing tests in a suite outside the build, express design ideas as working code in a scratch area of the repository. *Code is written for humans to read and only incidentally for computers to execute* — attributed to Knuth. A unique, unambiguous, shared artifact exists to transmit technical ideas.

Relates to: Tests Announce Intention; Wiki as Shared Location is dual to this practice.
As seen in: Seems to be novel as described, we'd love to learn otherwise. Although "ask the code" is a common XP slogan, it would seem to mean something rather different.

Tests Announce Intention *

Colleagues working on the same code base cannot "overhear" that they are about to collide on the same region of the code and so coordinate their efforts. Remote teams suffer integration races.
Therefore: Use functional and/or acceptance tests to publish the intention to work in a particular area. Remote colleagues can identify, asynchronously and unambiguously, what areas of the code others are likely to be changing soon.

Relates to: Code is Communication

6 Conclusion

A need for Distributed development exists in business, a desire to remain Extreme exists in the developer community. There is no need to compromise the second to accommodate the first.

Remaining firmly aligned to Agile principles allows development teams to grow with businesses as they globalize.

References

1. Beck, K. with Andres, C.: Extreme Programming Explained: Embrace Change (2nd Edition) Addison–Wesley (2004)
2. Beck, K. *et al*: The Agile Manifesto http://www.agilemanifesto.org/
3. Beedle, M., Schwaber, K.: Agile Software Development with Scrum Prentice Hall (2002)
4. Carmel, E.: Global Software Teams Prentice Hall (1999)
5. Cunningham, W.: About the Portland Form http://c2.com/ppr/about/portland.html (as at end 2004)
6. Daniels, J., Dyson, P.: CS2 Dispersed Development OT2004 http://www.spa2005.org/ot2004/programme.shtml (2004)
7. Jensen, B., Zilmer, A.: Cross-Continent Development Using Scrum and XP http://www.informatik.uni-trier.de/~ley/db/conf/xpu/xp2003.html (2003)
8. Fowler, M.: Using an Agile Software Process with Offshore Development http://martinfowler.com/articles/agileOffshore.html (as at April 2004)
9. Kircher, M., Jain, P., Corsaro, A., Levine, D.: Distributed eXtreme Programming
10. Martin, A., Biddle, R., Noble, J.: When XP Met Outsourcing XP 2004, LNCS 3092 (2004)
11. Poole, C. J.: Distributed Product Development using Extreme Programming XP 2004, LNCS 3092 (2004)
12. Reeves, M., Zhu, J.: Moomba–A Collaborative Environment for Supporting Distributed Extreme Programming in Global Software Development. XP 2004, LNCS 3092 (2004)
13. Rogers, R. O.: Scaling Continuous Integration XP 2004, LNCS 3092 (2004)
14. Simons, M.: Internationally Agile Informit.com (2002) http://www.informit.com/articles/article.asp?p=25929

A Case Study on Naked Objects in Agile Software Development

Heikki Keränen[1,2] and Pekka Abrahamsson[1]

[1] VTT Technical Research Centre of Finland, PO Box 1100, FIN-90571 Oulu, Finland
{heikki.keranen,pekka.abrahamsson}@vtt.fi
[2] Department of Information Processing Science, FIN-90014 University of Oulu, Finland

Abstract. Naked Objects and agile software development have been suggested to complement each other. Very few empirical studies to date exist where a product has been developed using the Naked Objects technologies in an agile development environment. This study reports results of a case study where a mobile application was developed using the Naked Objects Framework. Qualitative and quantitative data was collected systematically throughout the project. The empirical results offer support for the argument that the Naked Objects approach is suitable for agile software development. The results also reveal weaknesses in the current Naked Object Framework, namely, that it is not yet mature enough for applications that require intense database operations. The results also show that the development team was able to create an operational user-interface just in five hours, which demonstrates the applicability of the Naked Object Framework in practical settings.

1 Introduction

Naked Objects [4] is an architectural pattern which exposes core business objects to the user. Naked Objects Framework is a software framework which supports implementing this pattern. Pawson and Wade [5] have proposed that the use of the Naked Objects architectural pattern complements the Extreme Programming's (XP) set of practices. Developers can, for example, make use of the Naked Object Framework to render the requirements in a concrete form that is immediately usable by the end-users [5]. Originally, Pawson and Wade suggested that only exploration phase would benefit from the use of the Naked Objects Framework. Yet, the long term goal is to develop the Naked Objects Framework to a state that the development of the working prototype can be continued to a full working release.

Very few empirical studies exist today where a product has been developed using the Naked Object technologies in an agile development environment (i.e., XP or others). The lack of empirical studies hinders the ability of other practitioners to evaluate the proposed approach and makes it difficult for a researcher to pinpoint the weaknesses and strengths of Naked Objects pattern and framework. This study reports the results of a case study where a mobile application, enabling users around the world access the Helsinki Stock Exchange for trading and viewing stock market development, was developed. Fig. 1 shows a part of the user interface.

The project involved student developers working 8 weeks in calendar time and using a total development effort of 810 hours. While this is the first of the kind empirical study on the Naked Objects Framework, we characterize the study as of the 'empirical exploratory' type. No hypotheses were made to be tested. The results produced

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 189–197, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Fig. 1. Part of the stock application user interface: Main menu, branch view and stock view

in this study will therefore set some references for other developers and researchers, which can be tested or refuted. This paper concentrates on the process aspect of the development. In addition, the weaknesses and strengths of the Naked Objects based development are proposed including lessons learned from the project.

2 Naked Objects

In the Naked Objects Framework, the core business objects encapsulate all business data and behaviour. They implement the Naked Object Java interface and obey some simple coding conventions. The framework has an Object Viewing Mechanism (OVM) which autogenerates a desktop user interface based on information in the business objects. Core interfaces implemented by the application and the Java reflection mechanism are utilised to do that. [4] Due to the abstract nature of Naked Objects, it is possible to create OVM's for different kind of devices. In this project, we developed an OVM for 'Java Mobile Information Device Profile (MIDP) 2.0'-capable mobile phones called MIDP-OVM. The Naked Objects Framework also contains a set of Object Stores, which provide automatic persistence for the business objects. In this project, we used the XML Object Store, which saves the business objects into XML files, because it was considered to be the most mature object store. Due to the automatic user interface generation and object stores, the implementation of an application is supposed to be rapid.

3 Research Design

The research method utilized was a controlled case study approach [6], which is an approach drawn from the action research, case study research and experimentation. In a controlled case study, the development environment is a laboratory setting. Yet, the approach strives at an industry-like business and delivery pressure where the development of a particular system is of the highest importance.

The Naked Objects development team involved four students as the development resources. The team worked for 24 hours a week in the development facilities. Both quantitative and qualitative data were collected. The developers collected the used effort for each defined task with a precision of 5 minutes using paper/pen and an in-house tool. The size of the development work, i.e. logical lines of code, was collected on daily basis using automated counters. The qualitative data includes the developer team interview.

The development was guided by an adapted version of the Extreme Programming approach called Mobile-D [1]. The adaptation has been performed taking into account the specific demands of the mobile development environment. The development cycle involves 5 system releases having the duration of one to two weeks. Before the project started, a rough version of MIDP-OVM existed[1]. It was, however, only capable of browsing objects and needed a lot of development before a real application could be used through it. The idea was to refine it in the beginning of the project and after that develop an application on top of it. MIDP-OVM is application independent and was developed for the purposes of enabling other developers to use Naked Objects on mobile platforms.

4 Results

In this section the results of the case study are presented.

4.1 Effort Distribution

Effort distribution is presented in Fig. 2. The coding phase consists of tasks related to the implementation of a feature. The management includes the collection of metrics, daily meetings and the project management work. The section 'Other' includes the environment setup, studying, coaching and the documentation activities. The planning activities include the planning game in the beginning of each iteration, as well as the architectural planning during development iterations. The quality assurance includes the tasks for verifying the user stories and related tasks. The defect fixing includes the refactoring and bug fixing activities. The testing includes writing test cases and a pre-release testing session, which is performed prior to the release. The release formalities include the formation of baseline and the acceptance test performed by the customer.

As shown in Fig. 2, the development profiles are slightly different in the MIDP-OVM development as compared to the application development. A lot of defect fixing (15%) was done in the application construction phase. The management activities also took more effort in the application development phase than in the framework development phase. More testing (i.e. 10%) was required for the platform development than application (i.e. 5%) part.

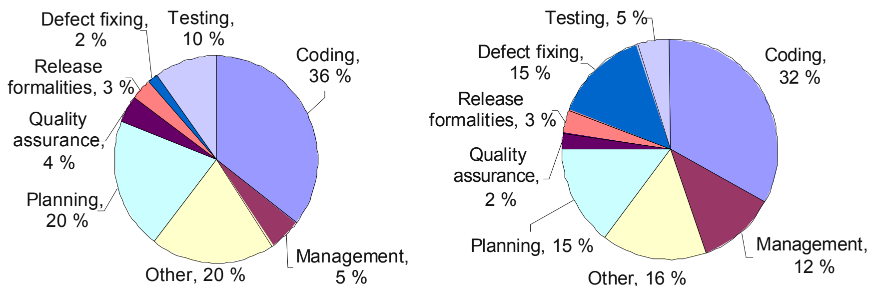


Fig. 2. Effort distribution of the MIDP-OVM construction phase (left) and the application construction phase (right)

[1] http://opensource.erve.vtt.fi/pdaovm/midp-ovm/

4.2 Estimation Accuracy and Precision

Estimation accuracy is presented in Fig. 3 using box plots[2]. The data used for drawing the box plots is based on the tasks that the developers' identified for the user story level implementation. The data below the thicker line indicates overestimation and data above the line refers to underestimation of the tasks.

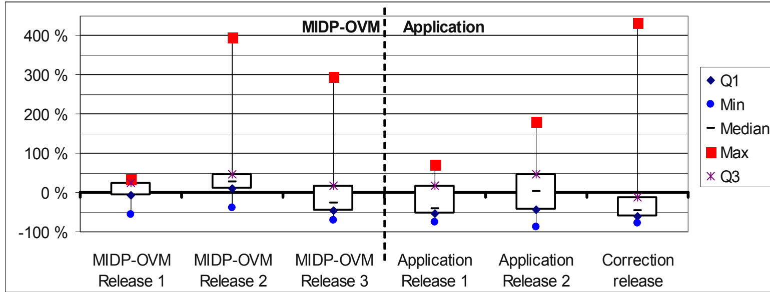


Fig. 3. Estimation accuracy

Overall, the data shows that the estimation error is a bit higher in the application creation than creating the MIDP-OVM. There is a high variance in the second and third releases of both MIDP-OVM and application. What is especially high, is the highest overspending in task time (Max-value), over 400 percent, which tells about unexpected problems in the development.

Figure 4 presents the estimation precision development, i.e. how many actual hours the developers lost by faulty estimates. The thick line indicates a loss of zero hours. The data points below the line indicate that an implementation of the task took less time than expected. The data points above the thicker line indicate that a particular task took longer than expected.

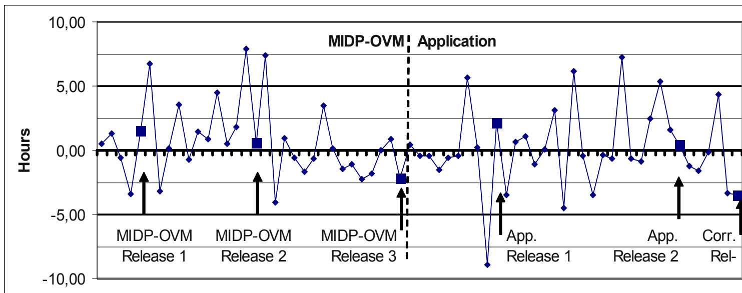


Fig. 4. Hours lost by faulty estimates

---

[2] A box plot diagram visualises the 5 number summary of a data set. Median value is the line in the shaded box area. A1 (first or lower quartile) shows the median of the lower 50% of data points. Q3 (third or upper quartile) shows the median of upper 50% of data points. The minimum value indicates the lowest and the maximum the highest values in the respective data sets.

By observing Fig. 4, we can see that in the implementation of the MIDP-OVM the estimation precision enhanced over time. On the other hand, in the application creation phase the implementation of the basic application went smoothly with very small task sizes, and thus, small errors in the absolute error estimates, but towards the end of the project, the estimates become less accurate.

4.3 Distribution of Task Sizes and User Story Effort

In the planning game, the team, together with a customer, identifies the user stories to be included in the iteration. The team divides each user story into a set of tasks preferably between 2-10 hours. The distribution of the actual task sizes are presented in Fig. 5.

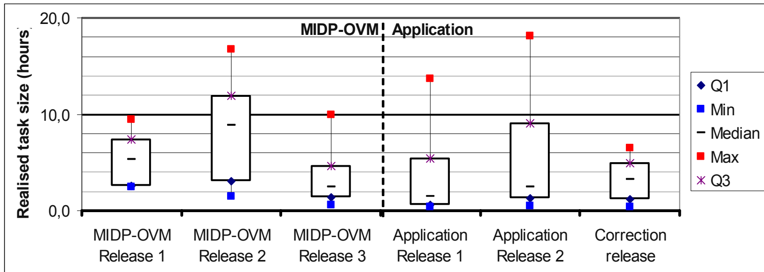


Fig. 5. Actual task sizes in each release

As can be deduced from the earlier figures, the estimation error caused some tasks to exceed the suggested limits. Yet, the data reveals that in spite of the complexity of creating OVM's, the team was able to split the user stories into reasonable sized tasks.

When creating the application, there were more tasks that took less than two hours, which indicates some difficulties in combining tasks into larger ones.

The user story effort correlates to how fast the project is able to generate visible results, meaningful to the customer. The user story effort is presented in Table 1. In the planning game of Application Release 1, there were first several user stories concerning the application requirements, introduced by the customer. As these were, however, considered trivial to be implemented using Naked Objects, the team decided to group these stories to one big story called “Create application” and define those stories as tasks.

Table 1. Actual effort used in the implemented user stories

Release	OVM R1	OVM R2	OVM R3	App. R1	App. R2	Corr. Rel.
User story effort (median, h)	13,7	49,1	9,7	0,6	9,7	9,8
User story effort (max, h)	9,5	98,2	24,7	2,4	23,7	14,5
# User stories implemented	2	2	5	6	9	2
# User stories postponed for next rel.	0	2	0	1	1	0

The size of this story was estimated at 8 hours and it was implemented in 5 hours, which is the size of a typical task in the normal development. To make it fair to com-

pare the implementation speed from the user point of view, in table 1 this "Create application" -story is split back into the original user stories.

In the MIDP-OVM Release 2, there is only one huge story and another with zero effort because it became a side effect of the huge story. This huge story is due to complete rewriting of the MIDP-OVM. The implementation speed of the user stories in Application Release 1 is remarkable (median 0.6 hours). Most of the effort in Application Release 1 was used to a story that took over 30 hours and still had to be postponed for the next release.

4.4 Growth of the Code Base

The development of the code base is important, since it portrays how the project progressed over time in terms of actual product development. Figure 6 presents the code size development during the project.

The initial version of MIDP-OVM was roughly two thousand lines of code, but after the experiences of the first release, the team decided to discard the old version and rewrite the MIDP-OVM starting from scratch, which explains the amount of code going down to zero. Bugs in the MIDP-OVM had to be fixed and some features had to be added in it, which explains the little changes in the amount of code after the application implementation started.

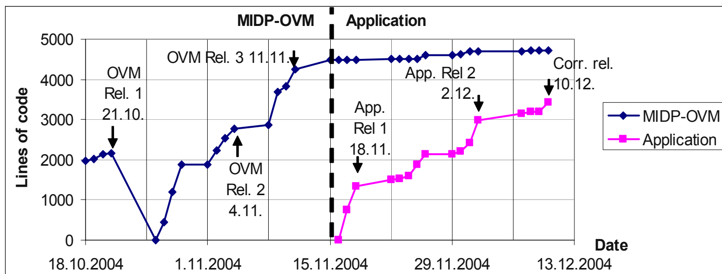


Fig. 6. Growth of the code base (LOC) during the project

4.5 Naked Objects from the Developers' Viewpoint

Any new software process innovation needs to be accepted by the developers. Otherwise the impact of the new technology is limited. In the interview, we asked the development team about the positive and negative experiences of using the Naked Objects Framework, as well as how easy or difficult they perceived the use of the framework and the pattern to be.

The programmers found the Naked Objects principle simple, but they felt that it requires quite little practice to learn how the framework actually works:

> "*For just writing the Naked Object application I think [it] is very easy, [it takes] just one or two days, and [a good way to do it is] writing couple of things and testing them if they work and if not try in another way and after three or four trials you get it, basically because there are new concepts, but they are not so many.*" *[developer 1]*

Developers viewed that the implementation an OVM was especially challenging:

> "*But programming the MIDP-OVM is like getting really inside the framework, how it works, and not just using in [in a way] the ordinary programmer would do it." [developer 1]*

As a part of the application, the team had to do a module which periodically parses the stock data from the public web pages and feeds it into the application. Due to limited time and lack of examples of using application specific direct connection to an SQL database, the developers were consulted in using XMLObjectStore to store data. Other object stores were considered but they seemed either not to be compatible with the framework version we used, or were labeled as "proof of concept" -version. Also, there were no examples on how to make application tailored persistence of Naked Objects. In the application release 2, the developers told that the XMLObjectStore is not capable of handling a vast amount of automatic updates. There was no time to explore other approaches.

The developers also reported other problems with the Object Store concept. The integration of the web page parser to the Object Store was found problematic:

> "*There is just not a clean interface to do that and that is why we had so many problems with it." [developer 1]*

There were also less severe problems which were considered as bugs in the framework:

> "*We found out that although the book said we could disable some actions or [editing] associations, but at least this framework we used did not support those." [developer 3]*
> *"The framework itself uses a strange way to call all the classes in the application once in a while and load the objects. Finally, we managed to solve the problem by putting the actual method calls in the title-method, so we had to be creative." [developer 3] "At first we ended up in an endless loop." [developer 2]*

The developers' opinion was that Naked Objects suits well for agile development:

> "*I even think that its biggest advantage is that after a very short time of coding, maybe two or three hours, you already see the result." [developer 1]*

5 Discussion

As it has been stated a few times, there are very few case studies on the Naked Object technology that would be comparable to the study presented in this paper. The closest one on the Java technology and on a similar research setting can be found in [2] (i.e., the "eXpert" project) where a web based system was developed using the same cyclic approach and development rhythm. Also, the practices and tools are mostly the same as well as some of the support team members.

One of the greatest differences compared to the eXpert project is that in that project the maximum estimation error varies greatly: In their study, the highest estimation error in the release was always from 100% to 170%. In our case, there were releases

well under 100% and the three releases experienced over 290% estimation errors. One reason, we suspect, is the defects in the Naked Objects Framework. Another is the lack of documentation and code examples on extending the framework and integrating it with legacy software. The third reason might be the complexity of writing extensions to the framework due to the reflection properties and abstract behaviour - it is easier to hard code things, as it is done in traditional software development, than in this case making the MIDP-OVM application independent.

It seems that the creation of MIDP-OVM follows the characteristics of traditional agile software project in the sense that the task size estimations become more accurate. On the other hand, when creating an application using Naked Objects, it seems that the beginning of the project goes very fast, with very small errors in task time estimations, but after a while, the development slows down and the estimation accuracy deteriorates, due to the fact that some parts of the software need to be implemented without Naked Objects and integration of those parts to the Naked Objects application may be hard. Due to the very limited time to implement the application, we cannot predict if the estimation accuracy would enhance over time.

Compared to the results of Pawson [3], in this case, we did not need business agility in this project, as the desired functionality was pretty much fixed before the project started. This study suggests that although the business agility of the Naked Objects applications is good [3], difficulties integrating Naked Objects into the traditional systems might cause a risk in that sense, since when decision to use the Naked Objects technology is done, all the future business changes may not be known.

As a conclusion, we can identify three principal lessons learned:

- Based on this study, it seems that the Naked Objects Framework (version 1.2) is not yet mature for making serious business applications having a lot of objects and multiuser security requirements. Generally, the problem seems to be "*leaving those predefined paths the Naked Objects people were defining",* as one of the team members commented; this is often necessary because the framework is not designed to address all problems of all applications.
- The rapid development of the first versions of the Naked Objects applications makes it possible to use Naked Objects in an exploratory phase, as Pawson and Wade suggested [5]. In this project, the requirements for the application were pretty well known, so we used the default Mobile-D way of not creating code during the planning phase, but normally, the first versions could be created with close interaction with the customer.
- Naked Objects enables a fast realization of user stories, as the data clearly showed.

6 Conclusions and Future Work

This paper has presented a first-of-a-kind empirical case study on a project using the Naked Objects Framework. The results show that the current Objects Stores are not mature for applications that need a high number of objects and high throughput. The lack of documentation and knowledge on how to do sample code, and lack of time in the project made it impossible to try the application tailored persistence for Naked Objects. The ability to generate applications fast is a remarkable feature of the Naked Objects technology and it may change the software business, if the current problems can be solved.

The usability of the autogenerated user interface was outside the scope of this paper and needs to be studied. It should be noted that the current development of the Naked Objects Framework is addressing the problems found in this study.

References

1. Abrahamsson, P., Hanhineva, A., Hulkko, et al. Mobile-D: An Agile Approach for Mobile Application Development, OOPSLA 2004, Poster session, Vancouver, Canada, 2004
2. Abrahamsson, P. and Koskela, J., Extreme programming: A survey of empirical results from a controlled case study, ISESE 2004.
3. Pawson, R., Naked Objects. PhD thesis, Trinity College, Dublin, 2004.
4. Pawson, R. and Matthews, R., Naked Objects. 2002: J Wiley.
5. Pawson, R. and Wade, V., Agile Development Using Naked Objects, XP 2003, p. 97-103.
6. Salo, O. and Abrahamsson, P. Empirical evaluation of agile software development: The controlled case study approach, Profes 2004.

Extreme Programming for Critical Systems?

Ian Sommerville

Lancaster University
is@comp.lancs.ac.uk

Abstract. From the perspective of a 'sympathetic sceptic', this talk will discuss the issues around the development of critical systems - systems where the costs of failure are very high - and whether or not extreme programming practices can be adapted and used in critical systems engineering. I will start by discussing the characteristics of critical systems development, such as the need to justify claims about the system dependability, and the differences in development culture between XP and critical systems development. I will then go on to discuss how different XP practices reduce or increase the risks of software failure, especially when the reality of implementing XP is considered. I will identify weaknesses in the XP process, such as the use of user stories for requirements definition, that have to be addressed before XP practices will be considered by the critical systems community. I will then suggest how the cultural barriers between the communities might be broken down and will propose how it might be possible to adapt XP practice to the development of some types of critical system by introducing 'dependability spikes' into the XP process.

Presenter

Ian Sommerville is Professor of Software Engineering at Lancaster University and has 25 years of experience in software engineering teaching and research. His textbook on software engineering, now in its 7th edition, has been widely used and adopted. Ian has research interests in system dependability, requirements engineering, service-centric systems and social and human issues in software engineering. He is convinced that conventional software engineering has much to learn from XP if only the communities could talk to rather than at each other.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 198, 2005.
© Springer-Verlag Berlin Heidelberg 2005

That Elusive Business Value: Some Lessons from the Top

John Favaro

jfavaro@tin.it

Abstract. Amid the enthusiasm generated by the success of agile approaches to software development in recent years, we have begun to extrapolate the principles of agile methodologies to propose innovative new ways of managing entire businesses, such as "self-reflective fractal organizations." However, it is important to resist the tendency toward an attitude that managers have more to learn from us than we do from them. Such an attitude is based on the assumption of a direct and automatic link between agile development and business value creation that is often more imaginary than real, and certainly harder for them to recognize than us. Even a perfectly organized, successful agile project may fail to deliver any business value at all; even more often, we are unable even to judge whether any business value has been created. The potential of agile approaches to contribute to value creation is large, but only if we recognize that we must reach beyond them to study and learn from the best managers. This starts with an appreciation of how difficult (and rare) it is to create a sustainable competitive advantage with IT investment. Some lessons from top management can help us understand the sources of competitive advantage and how we can help our customers in their search for that elusive business value. In so doing, we enable ourselves to create better business cases for managers to invest in software development and we sharpen our own focus on what software development will create the most value for the business.

Presenter

John Favaro is the founder of Consulenza Informatica in Pisa, Italy. In 1996 he introduced the principles of Value Based Management in software engineering in an article in IEEE Software on the relationship between quality management and value creation. In 1998 he introduced Value Based Software Reuse Investment, applying the ideas of Value Based Management and option pricing theory to the analysis of investments in software reuse. Recently he has investigated the relationship of Value Based Management to agile development processes. He is a founding member of the International Society for the Advancement of Software Education (ISASE), and was Guest Editor of the May/June 2004 Special Issue of IEEE Software on "Return on Investment in the Software Industry." He took his degrees in computer science at Yale University and the University of California at Berkeley.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 199, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Agility – Coming of Age

Jutta Eckstein

www.jeckstein.com

Abstract. XP exists for almost ten years now. The latest Standish Report characterizes agile development as a top success factor. The new certified, standardized, and official process model of the German government, V-Modell XT, includes an agile project strategy. These are all clear signs that the agile approach is coming of age.
Being a grown-up doesn't mean being fully developed – in contrary the need still exists to continuously seek for opportunities for improvement. Thus, instead of being dogmatic about practices, we have to use the agile value system as our guidance for improvement and for creating and customizing practices that help the teams to succeed.

Presenter

Jutta Eckstein (www.jeckstein.com, info@jeckstein.com) is an independent consultant and trainer from Braunschweig, Germany. She has a unique experience in applying agile processes within medium-sized to large mission-critical projects. This is also the topic of her book Agile Software Development in the Large. Besides engineering software she has been designing and teaching OT courses in industry. Having completed a course of teacher training and led many 'train the trainer' programs in industry, she focuses also on techniques which help teach OT and is a main lead in the pedagogical patterns project. She has presented work in her main areas at ACCU (UK), OOPSLA (USA), OT (UK), XP (Italy and Germany) and XP and Agile Universe (USA).

She is a member of the board of the AgileAlliance and a member of the program committee of many different European and American conferences in the area of agile development, object-orientation and patterns.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 200, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Another Notch

Kent Beck

Three Rivers Institute
kent@threeriversinstitute.org

Summary. How do we take XP to the next level? How do we get respect and freedom to work as we'd like? How do we get them to listen to us? XP has successfully raised expectations for what is technically possible with software development. The next challenge is realizing that potential. Doing so requires not more technical skills, but better relationships within the organization, a shift in attitude and perspective.
It's natural to want to have impact in the world. How can we best have impact on organizations? Counterintuitively, the way we gain influence is to listen. The way to gain freedom is to be accountable. The way to get respect is to give it. The way to get them to listen to us is to eliminate the dichotomy between us. We are all on the same side working towards a more effective software development process for the good of our company.
What we need is a change in perspective. XPers should first demonstrate that others have impact on them, by listening and acting on what others say. We need to offer accountability. With a record of careful listening and trustworthiness, we will be well-positioned to be heard when the organization has a problem and we have an idea.
A common barrier to organizational impact for programmers is our sense of "being special". The days of the prima donna programmer are over. What would happen if we treated everyone we talked to as if their ideas, needs, and perspectives had equal value with our own? That would be extreme. It would require a shift in our beliefs about organizations and our contributions to them.
XP has improved programmers work. Influence at the next level uses the same principles apply that have guided XP thus far. Respect, mutual benefit, improvement in baby steps from where we are today; these are principles that can guide the maturation of an XP team to be a full partner in business. These are the principles that, if applied with humility and awareness, will help us take XP up another notch in impact and influence.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 201, 2005.
© Springer-Verlag Berlin Heidelberg 2005

A Process Improvement Framework for XP Based SMEs

Muthu Ramachandran

School of Computing, The Headingley Campus
Leeds Metropolitan University, Leeds LS6 3QS
m.ramachandran@leedsmet.ac.uk

Abstract. XP has introduced best practices into software development. However we need to adopt and monitor those practices continuously to maximise its benefits. Our previous research has focussed on software process improvement model for SMEs (Small-to-Medium Enterprises). This paper introduces a process improvement framework for assessing and improving XP best practices. We have also developed a web based tool support to assess, improve, and suitability of introducing XP into SMEs.

1 Introduction

Extreme Programming is a methodology which is based on a number of key basic principles such as rapid feedback, incremental change, and quality work. Extreme Programming projects make use of a minimum of up-front design, instead continuously revisiting requirements. There is a strong focus on team working with the emphasis on personal communication and morale rather than on documentation and this helps to build a good team spirit. The client is seen as a colleague and as such is integrated into team discussions. There are plenty of sources of materials and online web sites on current XP practices [2], [3]. Cohn and Ford [2] provide a more focused research articles on current XP practices. Who is suitable for XP? We consider it is suitable for all including SMEs. Any organisation with fewer to less than or equal to fifty employees can be considered as SMEs.

There is also an excellent survey on current XP practices provided by Shine Technologies [1]. The survey results have been unexpectedly positive:

- 93% said team productivity improved
- 88% found the quality of applications was better
- 83% experienced better business satisfaction with the software

Research is need to address many issues that are remain unresolved such as how do we improve current XP practices and how do we transform and adopt XP for a large scale software projects? How do we adopt and extend XP practices into a more systems engineering projects? How do we adapt existing software process improvement such as CMM and CMMI and quality methods such as SPICE, etc.?

Our earlier work has concentrated on adopting CMM-like-model for SMEs [4]. In this work we have addressed the issue of how to adopt and assess process improvement methods for XP based SMEs. We have also developed a knowledge based tool support for assessment of XP practices and provide what is best for that company.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 202–205, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 XP Based Process Improvement Framework

In general software process improvement framework is based around a number of levels; providing a list of key practice areas at each level. This is quite unacceptable for SMEs and XP based organisation, although research claims that level 2 CMM is always compatible with SMEs and any other organisation. Whatever it may be the case as long as we provide a clear feedback on current XP practices and make opportunity for possible improvements to XP principles then we can systematically adopt and practice XP. We believe CMM-like heavy-weight model may be is unsuitable for SMEs and XP based projects. Therefore we have proposed a model for process improvement known as XPI (XP based process improvement) as shown in Figure 1. It considers XP based practices right from the beginning and it is quite intuitive for automation when adopted. This model starts identifying current XP practices against well known XP principles. This can be achieved using built-in XP experts' knowledge. Then it identifies improvements that are necessary to adopt XP systematically. It then includes planning, monitoring, and improvements.

In the following section we will look at how this can be made possible using automated tool support for continuous improvement and knowledge sharing of best practices across the organization.

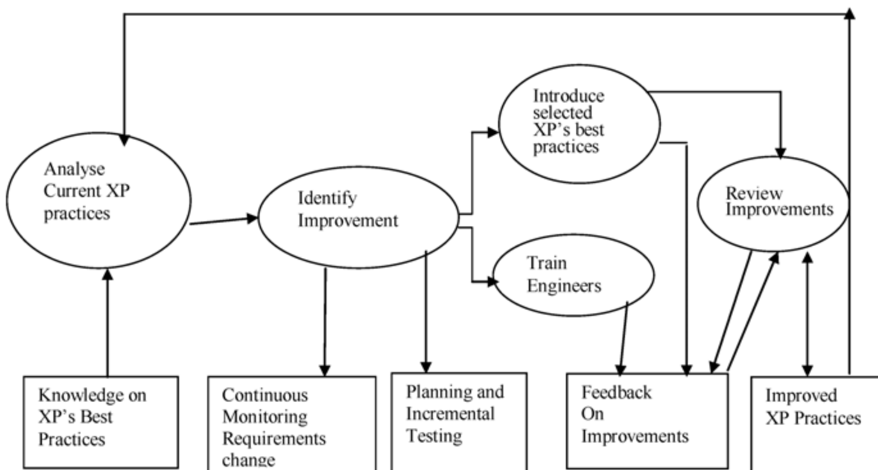


Fig. 1. XP based Process Improvement Framework

3 Knowledge Based System Support

We have identified a possible model for automated assessment and monitoring XP best practices continuously as shown in Figure 2. This consists of three major aspects to this system, suitability assessment, best practice assessment, and improvement. We aim to encode the best practices as knowledge to the system.

It is important to conduct an assessment before implementing XP into your organisation. We have developed a tool, XP maturity model for measuring the success of XP and also its impact on software process improvement models like CMM (Capabil-

ity Maturity Model). The purpose of this form is to enable people with little or even no knowledge of XP, to estimate quickly easily whether XP will fulfil their needs and requirements. The program consists of a form containing a handful of simple questions. The answers from these questions will provide immediate feedback on whether XP is appropriate for the person who answered the question.

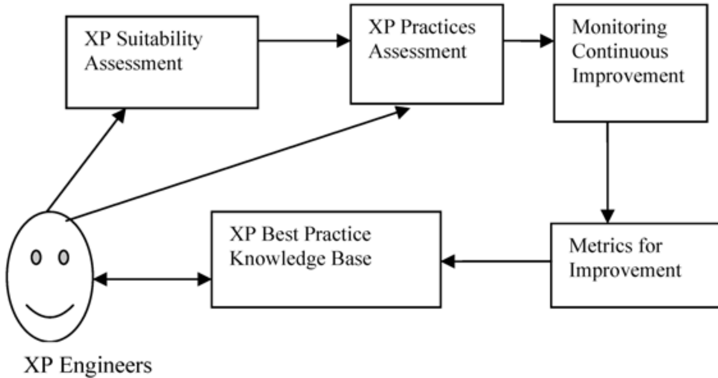


Fig. 2. A System Support for Knowledge Based XP Improvement

4 Checklist Based Assessment

The form will ask questions about the critical areas surrounding XP. We need to identify with as few questions as possible whether XP is, or is not appropriate. The following aspects have been identified as critical for XP:

- team size, client on site, team location

In order to provide a somewhat more subtle analysis, the following (less critical) aspects have also been selected:

- requirements volatility, facilities strategy

Figure 3 is the illustration of our tool support which provides a web interface and online assessment forms to assess suitability for introducing XP into SMEs or any organisation. The interface has been made simple thus allowing a first time user to fill in the form right away and getting a result within a few minutes. The results will be colour coded to help result interpretation and a summarised result will also be available. We have developed a web based tool which provides an assessment and analysis for migrating to XP.

5 Conclusion

XP has introduced best practices into software development. XP can help to speed up production and delivery of our software systems. However research is needed to minimise and introduce XP best practices systematically across projects and organisations. In the work we have proposed a model for improvement of XP practices and also have developed a prototype tool to support introducing XP into SMEs.

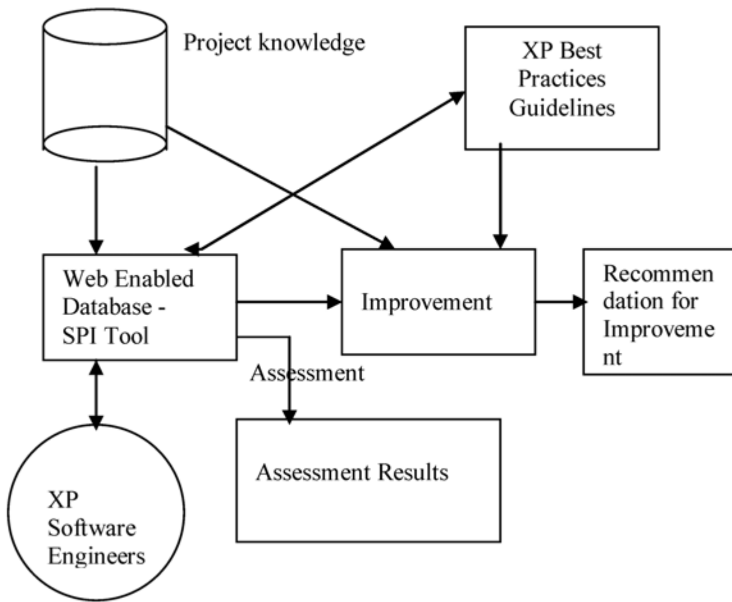


Fig. 3. Web Based Assessment Tool

References

1. Shine Technologies: Agile Methodologies Survey, www.shinetech.com/agile_survey_results.jsp, Jan.2003
2. Cohn, M., Ford, D. Introducing an Agile Process to an Organization, Special Issue, IEEE Computer, June 2003
3. Beck, K. Extreme Programming Explained: Embrace Change, Addison-Wesley, 2000
4. Allen, P., Ramachandran, M., Abushama, H., PRISM: A model for process improvement for SMEs, Intl Conference on Software Quality, QSIC, November, 2003

Standardization and Improvement of Processes and Practices Using XP, FDD and RUP in the Systems Information Area of a Mexican Steel Manufacturing Company

Luis Carlos Aceves Gutiérrez[1], Enrique Sebastián Canseco Castro[2], and Mauricio Ruanova Hurtado[3]

[1] Universidad de Monterrey, Ciencias Computacionales, Av. Morones Prieto 4500 Pte, 66220 San Pedro Garza García, Nuevo León, México
laceves@udem.edu.mx
http://www.udem.edu.mx
[2] ecanseco@yahoo.com
[3] mruanova@yahoo.com

Abstract. This work focuses on standardization and improvement of processes and practices using a combination of methodologies including Agile Methodologies (AM). It was implemented at a Mexican steel manufacturing company using FDD, XP and RUP. The main goal was to improve the software systems production, maintenance and support.

1 Introduction

This document identifies the needs and problems that the company faced. There were no customary procedures in daily operation, as well as lack of systems documentation. Regarding the documentation, it was inadequate or nonexistent. To correct this processes redesign through the combination of RUP, XP, and FDD methodologies, was adopted including documentation and practices.

2 Diagnosis

Three main stages were identified*: New Projects, Project Change and Improvement, and General Support*. Following is the description of the three main stages:

- *New projects:* By New Projects we refer to the development of a solution which does not involve existing modules. [3] [7]
- *Project change and improvements:* Change and Improvement are those modifications that users request, and involve non-structural changes to existent applications and queries. [3] [7]
- *General support projects:* The support area is in charge of registering and following up system errors notifications, and technical questions. [3] [7]

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 206–209, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Table 1. Summarization of problems with its respective indicators, according to each stage, which are identified as follows: New Projects *(N)*, Change and Improvements *(C)*, and General Support *(S)*

Indicator / Problem	Department is highly dependent on personnel	Difficult to adapt to organizational changes	Difficulty systems maintenance	Existent systems affectation errors.	Job posts without scope responsibilities	Non-documented or informal documented systems	Nonexistent formal development methodology.	Non-standardized development processes	Too much time invested studying system's code
Coworkers' collaboration.								N,C,S	
Responsibilities scope.					N,C,S		N,C	N,C,S	
Tasks continuation.					N,C,S			N,C,S	
Interface between personnel tasks.		N,C,S					N,C	N,C,S	
Opportune, truthful, and reliable documentation.				N,C		N,C	N,C	N,C	
Input and output documentation.							N,C	N,C	
Subordinate coordination.							N,C	N,C	
Version control.				N,C		N,C			
Centralized documentation.				N,C		N,C	N,C		
Personnel training time.	C		C,S						C

3 Redesign Through XP, FDD and RUP

The redesign included so much the documentation and the processes flow. The documentation redesign identified the RUP, XP, and FDD artifacts and practices to implement and so the actual way of work of the enterprise. [2][4][5]

Artifacts were filtrated so the ones that really had added value were chosen. All of them were adapted and approved by consensus. Finally it was cataloged according to RUP disciplines. [1] Also a set of software tools were implemented in each discipline.

4 Results on the Use of XP, FDD and RUP

In order to carry out the redesign implementation, and opportunity of putting in practice the redesigned processes was reached. This opportunity was through different

projects that matched the implementation needs, a new project, a change and improvement project and a general support request.Implementation results reported that developing time was greater than previous developing process, which was expected, however, lets remember that the objective of this project was not to accelerate the software developing time but to standardize the way of work of the three departments.

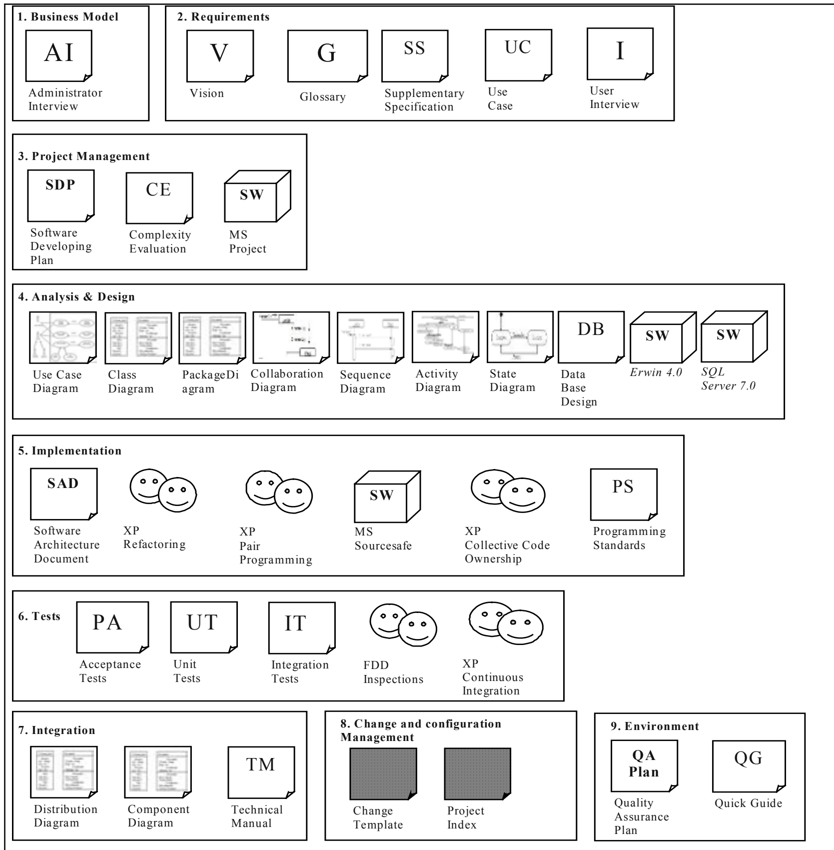


Fig. 1. Artifacts, practices and software tools implemented according RUP disciplines

Table 2. Comparative time of development between original and redesigned process in New Projects and Change and Improvement stages

Subprocess	New Projects		Change and Improvements	
	Redesigned	Original	Redesigned	Original
Needs identification	10	36	8	6
Requirements description and schedule	10	10	8	6
Analysis and Design elaboration	40	40	15	3
Implementation	20	24	4	2
Total Hours	80	110	35	17

Table 3. Comparative time of attention between original and redesigned process in General Support stage

Subprocess		Redesigned	Original
User attention		2	2
System errors		8	8
	Total Hours	10	10

5 Conclusion

Although not all of the artifacts were included neither this redesign speeded up the development process, it certainly helped the company to standardize its development processes as well as the tasks among the development team.

References

1. Ambler, Scott, “Agile Modeling“ , Agile Documentation, (2005), http://www.agilemodeling.com/essays/agileDocumentation.htm
2. De Luca, J. y Coad, P., “The Community Portal for al things FDD”, Feature Driven Development, (2003), http://www.featuredrivendevelopment.com
3. González, Pablo, Personal interview, Project leader Deacero S.A. de C. V., Monterrey, N. L., 2003.
4. Jeffries, R., “What is Extreme Programming?”, XProgramming.com An Extreme Programming Resource, (2001) ronjeffries@acm.org, http://xprogramming.com/xpmag/whatisXP.htm
5. Kruchten, P., The Rational Unified Process an Introduction, Addison-Wesley Pub Co., United States, (2000).
6. Larman, C., Applying UML and Patterns, Pearson Education, Second Edition, India, (2002).
7. Ravizé, Javier, Personal interview, Systems Managements, Deacero S.A. de C. V., Monterrey, N. L., 2003

Multithreading and Web Applications: Further Adventures in Acceptance Testing

Johan Andersson, Geoff Bache, and Claes Verdoes

Carmen Systems AB, Odinsgatan 9, SE-41103 Göteborg, Sweden
geoff.bache@carmensystems.com

Abstract. At XP2004, two of the authors presented an "agile record/replay" approach[1] to GUI Acceptance Testing based on recording high level use-cases. In the past year we have run a project to attempt to write tests using this approach for three different Carmen Systems products.[2] During this project we have met new challenges presented by multi-threaded GUIs and web GUIs, and in the process we have produced JUseCase[5] – a Java Swing equivalent of PyUseCase[5], presented last year, and for web application testing we produced WebUseCase[6] – a browser-like use-case recorder based on JUseCase. Via these use-case recorders, we have found that we can fit both these challenges comfortably into our existing approach.

1 Summary of Our Acceptance Testing Approach and Tools

These are presented more fully in our paper from XP2004[1] and to some extent XP2003[3] as well. This is a short summary of the ideas presented there.

1.1 TextTest: Verification by Textual Differences

We verify program correctness by the simple mechanism of comparing plain text produced by a program against a previously accepted version of that text - essentially comparing log files with a graphical difference tool like tkdiff. This way, writing tests never involves writing test code, and plain text is readable, portable and very easy to change – important aspects when it comes to maintenance of large test suites.

TextTest[4] essentially assumes a batch program that will perform a task and produce text files without human intervention, and then exit.

1.2 xUseCase: GUI Usage Simulation with Use Case Recorders

When it comes to testing GUIs, we advocate an agile "record/replay approach" based around automatically recording and replaying a high level 'domain language' script that models the use case that the test writer is performing with the GUI in question. This combines the strength of traditional record/replay

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 210–213, 2005.
© Springer-Verlag Berlin Heidelberg 2005

approaches (rapid creation of tests, few skills required to create them) with the strength of data-driven approaches (easily tweaked high-level test representations). In short, the core assumption is that recording is great but re-recording is horrible.

This is achieved by providing a GUI library-specific layer that can listen for every event that the application listens for, and can be told by the application what the intent behind the event is in the language of the domain. This means it can record this high-level statement instead of something based on the screen layout or other GUI mechanics.

Two such libraries currently exist: PyUseCase[5], for the Python library PyGTK and JUseCase[5], for Java Swing.

2 Testing Multi-threaded Programs

When we replay a test without human intervention, it may well be necessary to wait for things to happen before proceeding. Otherwise the test will fail because further use case actions rely on data loaded in a separate thread being present. In this case a traditional record/replay tool is basically stuck: it knows nothing of application intent and all it can do is ask the test writer to hand-insert 'sleep' statements into the script after recording it. Needless to say, this is both inefficient and error-prone.

Our use-case recorders handle this situation by introducing the notion of an "application event": the application can simply notify the use-case recorder when a significant event has occurred that is worth waiting for. At places in the code where such events occur, the programmer adds calls to xUseCase, which will then record a "wait for <name of application event>" command. During replay the replay thread will halt until the application reaches the point where the application event occurs, i.e., when the use-case recorder is notified of the event having occurred.

For example, assume we have the following use case script from a Swing App, using JUseCase:

```
load movie data into list
select movie Die Hard
```

Also assume that the first command starts a separate thread that loads a large amount of data from a database and displays it on the screen. Unless there is a way of telling the replayer when this has completed, it would perhaps try to select "Die Hard" before that item was present in the list, causing the simulation to fail. To solve this, the programmer inserts

```
ScriptEngine.instance().applicationEvent("data to be loaded");
```

at the appropriate point in his application. The recorded use case will now look like this:

```
load movie data into list
wait for data to be loaded
select movie Die Hard
```

In record mode the applicationEvent method just records the "wait for" command to the script file. In replay mode, the replayer halts replaying on reading this "wait for" command, and the applicationEvent call then acts as a notifier to tell it to resume when the data has been loaded.

3 Use-Case Recording for Web Applications

The GUI for a web app is presented in an external application: a web browser. This means that recording use cases presents new and different challenges.

3.1 Simulating User/Browser Interaction

To obtain scripts at the use case level, we need to approach the application from the browser side, rather than HTTP level. In this domain a number of web application testing frameworks exist, e.g., actiWATE, Canoo WebTest, HttpUnit and HtmlUnit [7]. They are all in some sense browser simulators, allowing the creation of web page objects that support clicking on links, filling out forms etc. They mostly expect that tests will be written using a provided Java API, although Canoo WebTest makes use of XML scripts instead.

3.2 A Simple Browser Based on HtmlUnit

As we already had a use-case recorder for Java Swing (JUseCase), we decided to take one of the Java frameworks and build a simple web browser on top of it, plugging it into JUseCase at the same time. It would then be possible to record and replay use cases for the web application within the browser. Of the available options, HtmlUnit seemed to be the most browser-like framework that was also open source, so we decided to build on that and created "WebUseCase"[6]. Aside from being actively developed and maintained, HtmlUnit can also be configured to mimic and behave as known browsers like Microsoft Internet Explorer, Mozilla and Netscape – an important feature when testing web applications from the browser side.

3.3 Getting Relevant Use Case Descriptions

Having a Swing browser for web apps was one step towards being able to record and simulate user/GUI interactions, but there was more to do: all components had to be connected to JUseCase. At a first glance this didn't seem to be much of a problem since the GUI only used standard components already supported by JUseCase – for each link or form component an equivalent component in Swing could be connected up to JUseCase as discussed previously. That was the easy part. The hard part was to choose relevant names for the use case commands connected to the component events.

When the GUI isn't part of the application code that we want to create use cases for, how are we to set proper use case command names, i.e., names that

tell something about the intent of the application? Since the only connection between the browser and the web app is a number of HTML pages, that's where the use case command names have to come from. So how do we put the use case command names into an HTML document, without destroying it?

Fortunately, most HTML tags support the 'title' attribute which can be used to define use case command names. This attribute is optional and "offers advisory information about the element for which it is set" [8]. Using it to set use case command names thus doesn't conflict with its intended use.

The title attribute gives the application developer full control over which use case command names to use at which places, and since the attribute is available for both links and form controls, it provides a consistent mechanism for defining use case command names.

4 Conclusion

The acceptance testing approach advocated has shown itself to be versatile and applicable well outside of the realm of the batch applications where it began. Use-case recording has proved itself for 'fat client' multi-threaded GUIs in both Java and Python, as well as for web applications.

References

1. Andersson, J. and Bache, G.: "The Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing" in Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004). Germany, 2004.
2. Verdoes, C.: "Use case recording and simulation : Automating acceptance tests for GUI applications". Chalmers University of Technology, Sweden, 2005.
3. Andersson, J., Bache, G. and Sutton, P.: "XP with Acceptance-Test Driven Development: A Rewrite Project for a Resource Optimization System" in Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2003). Italy, 2003.
4. TextTest is free and open source. It can be downloaded from `http://sourceforge.net/projects/texttest`
5. Both PyUseCase and JUseCase are free and open source. They can be downloaded from `http://sourceforge.net/projects/pyusecase` and `http://sourceforge.net/projects/jusecase` respectively.
6. WebUseCase will be released as open source pretty soon...
7. The HtmlUnit Java library for automatic simulation and testing of web applications is open source and can be downloaded at `http://htmlunit.sf.net/`
8. The HTML 4.01 specification, `http://www.w3.org/TR/html4/`

Using State Diagrams to Generate Unit Tests for Object-Oriented Systems

Florentin Ipate[1] and Mike Holcombe[2]

[1] IFSoft, Romania
fipate@ifsoft.ro
www.ifsoft.ro

[2] Department of Computer Science, University of Sheffield, UK
m.holcombe@dcs.shef.ac.uk

Abstract. Traditionally, finite state machines and their extensions, such as stream X-machines, have been used for modelling and testing of graphical user interfaces (GUI) and for acceptance testing. This paper shows how these testing techniques can be successfully extended to unit test generation for object-oriented systems and integrated into Extreme Programming in a simple and designer-friendly way. The approach has been used by MSc students in Computer Science at the Pitesti University to write JUnit tests for XP projects and the effectiveness of these tests has been compared with that of tests produced using ad-hoc and traditional functional methods. The conclusions show that over 90 % of the faults found by other methods have also been found by the stream X-machine based approach, whereas less than 75 % (in many instances less than half) of the faults uncovered by the stream X-machine based testing have been found by other methods. As the finite state machine based test generation has been automated, the time spent using the two testing strategies was roughly equal.

Keywords: unit testing, functional testing, state diagrams, finite state machines, stream X-machines

1 A Simple Example

Suppose that we are building a simple computer system for a library. We might identify a number of stories such as the following: Borrow book (a book can be borrowed if the customer does not have the maximum permitted number of books on loan), Return book, Reserve book (a book that is currently on loan can be reserved) Extend loan (the loan can be extended if the borrowed book has not been reserved by another customer).

From the above user stories we can identify two obvious class candidates, *Book* and *Customer*, and their operations; for *Book*, these are *borrow*, *return*, *extend*, *reserve*. Three states of a book can also be identified: *Available*, *Borrowed*, *Reserved*. Having identified the operations and the states, we can now proceed to drawing a state diagram for the *Book* class. As it turns out, the state diagram (Fig. 1) has actually 4 states, since, once a book has been reserved,

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 214–217, 2005.
© Springer-Verlag Berlin Heidelberg 2005

it has to be known whether the book is still on loan by the current customer (*Borrowed&Reserved*) or has been returned (*Reserved*) and the new customer can proceed with the borrowing. Furthermore, there has to be a way of progressing from the *Reserved* state, so we can decide that the reservation has to be cancelled before the new customer can borrow the book. If the book is no longer of interest for the customer who has made the reservation, s/he can just *cancel* the reservation, either from the *Reserved* state or from the *Borrowed&Reserved* state. The state diagram in Fig. 1 describes the possible sequences of operations that the class can perform in correct use, that must be obeyed by any class users or clients. On the other hand, the class cannot control these users, so it is never known when an operation will be called. Thus, in order to insure the correctness of the system, programmers use suitable error handling to deal with incorrect or unexpected use. This situation can be modelled by adding to the diagram erroneous transitions to an *Error* state.

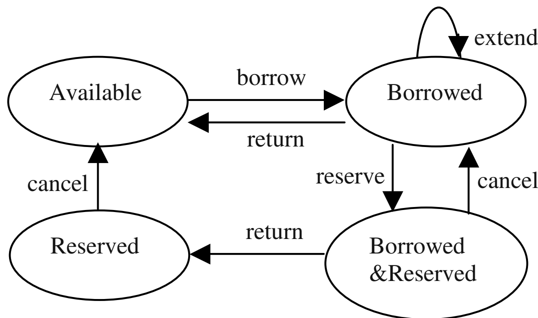


Fig. 1. The state diagram for *Book*

We can now proceed in a similar way with the *Customer* class and identify 3 potential states: *Empty* – when the customer has no books on loan, *Full* – when the customer has the maximum permitted number of books on loan, and *Partial* – when neither of these situations apply. There is, however, an important difference in comparison with the diagram for *Book*, since, in this case, the transitions from the *Partial* state will be *conditioned* by predicates.

As classes form associations, it is not sufficient to test them independently and some testing has to be done for some small groups of classes with high coupling between them. This kind of testing cannot be left until system integration and has to be performed during the unit testing phase. When an object in a class A can send messages to an object in a class B, we can use a state diagram to show the effect of the operations of A on the object in B, i.e. apply these operations in the states of the class B diagram. Obviously, only those operations of A that can affect the state of B (a message form A to B is sent as a result of the operation) need to be considered, the others may be omitted. When the association between the two classes is bi-directional, two diagrams (one for each

direction) will normally be needed. We can safely assume that a *Customer* object will send messages to a *Book* object, but not the other way around, so the association between these two classes can be modelled by a single state diagram.

2 Finite State Machine and X-Machine Based Testing

Once we have produced the states diagrams, we can use them to generate test cases in a rigorous manner and to automate the testing process, by applying existing finite state machine based strategies. One of the most general approaches is the W-method [2], that generates sequences of symbols to reach every state in the diagram, check all the transitions from that state and identify all destination states to ensure that their counterparts in the implementation are correct.

It is straightforward to apply the W-method to a finite state machine, but the state diagrams that describe the behaviour of a class or a class association are not, strictly speaking, finite state machines, as transition labels are not mere symbols from an abstract alphabet, but have some functionality attached to them – they can be represented by *mathematical functions*. This more general model, in which the labels of the transitions are mathematical functions is called a *stream X-machine* [1]. Each such function is driven by some input (operation name, input parameters), performs some processing on the object data and may produce some output (output parameters, display messages). Thus, the following information is associated with each transition label:

– **Input:** the *name* of the operation performed, the *input parameters* (if any) and their domains.
– **Data domain:** the domain of the data values for which the operation is valid.
– **New data:** the updated data values.
– **Output:** the *output parameters* (if any) of the operation and their values and any other *observable outputs*. If the state diagram shows the effect of the operations of a class A in the states of a class B, the object in A will also be assimilated as input parameter for each operation.

For each diagram, the four components for each label will be identified and the results will be placed into a table. Each user story will correspond to one or more rows in this table. The table can then be used to identify the sequence of inputs (operation names and input parameters) that drives a sequence of transitions so that each sequence of transition labels generated by the W-method can be translated into a sequence of program statements and an appropriate test program can be written. Furthermore, the expected outputs can also be derived from the table and these can be compared with the outputs produced by the test program. However, this testing method is only effective if the diagram (stream X-machine) satisfies some *design for test* conditions: observability and controllability [1]. In unit testing, these requirements can be achieved by splitting a transition into two or more transitions (observability) and by designing special operations to set up the appropriate context for the conditioned transitions (controllability).

3 Conclusions

State diagrams are intuitive and easy to use. They require little formal training but, at the same time, are rigorous means of describing the behaviour of a class or a system, since they are based on mathematical models such as finite state machines and their extensions. State diagrams can help to clarify and refine design details (in our example, a new method, *cancel*, that did not came out directly from the user stories, was identified when the *Book* class diagram was drawn up). Finite state machines and stream X-machines provide the basis for rigorous testing, without any other kind of (semi-) formal specification being necessary, which is an important advantage in the context of Extreme Programming.

Testing must be automated as much as possible in Extreme Programming and functional tests themselves are written as computer programs [3]. It is straightforward to convert a finite state machine into a computer program and this process can be easily automated. Furthermore, the process of generating test sequences form a finite state machine can also be automated and appropriate tools exist. Obviously, the tester will have to look up in the table which stores the transition label details and produce appropriate sequences of code statements to drive the sequences of labels that come out of the finite state machine test generation tool, but, on the other hand, functional tests cannot be fully automated unless a complete (formal) specification and tools for writing and executing it are available, which is not the case in Extreme Programming, nor in most development approaches in the core software industry.

As the method thoroughly tests a class or a system, it usually produces a larger number of test cases in comparison with traditional functional methods, such as such as category-partition. However, the test cases produced are easier to run in comparison with these methods, since there is no need to explicitly establish the context (the state) before actually running the tests (the test sequences reach the state and identify it before checking all the transitions that come out from it).

The key benefit of the stream X-machine based testing method [1] is that sets generated *fully tests* the class, as all possible transitions, including the error handling part of the operations, will be checked in every possible context (state of the diagram). Furthermore, the method does not only test the operations individually, it also tests their coordination within the class.

References

1. Holcombe, M. and Ipate, F. 1998. *Correct Systems: Building a Business Process Solution*. Springer Verlag: Berlin.
2. Ipate, F. and Holcombe M. 1997. An Integration Testing Method That is Proved to Find all Faults. *Intern. J. Computer Math.* **63**: 159-178.
3. Jeffries, R., Anderson, A., Hendrickson, C. 2000. *Extreme programming installed*, Addison-Wesley.

The Positive Affect of the XP Methodology

Sharifah Lailee Syed-Abdullah, John Karn, Mike Holcombe,
Tony Cowling, and Marian Gheorge

Dept of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
{s.abdullah,j.karn,m.holcombe,a.cowling,m.gheorge}
@dcs.shef.ac.uk

Abstract. This paper describes a longitudinal study on how the XP methodology acts as a positive mood inducer to SE teams. The results provide empirical evidence of the ability of these practices to alleviate the positive feeling amongst SE teams and there is a strong relationship between the positive moods experienced by the teams and the number of the XP practices used.

Keywords: Agile methodology, positive affect, empirical evidence, humanistic factor.

1 Introduction

Past research has shown that a positive affect or temperament induction leads to a greater cognitive flexibility and facilitates creative problem solving across a broad range of settings. It is the intention of this paper to explore the possibility of XP as a positive affect inducer and to discuss the finding of a longitudinal study on the possible impact of XP practices on the developers. Positive affectivity refers to an individual's disposition to be happy across time and situations [1]. To achieve this, comparison studies were conducted on students embarking on a real life project in the Sheffield Software Engineering Observatory. The second section of this paper discusses the experiments and the result of the study made on the XP methodology in action. The findings revealed that the XP methodology does have an impact on the positive temperament of the developers when most of the practices were used. The study was carried out to determine, whether the teams using XP practices experienced a higher positive affectivity than the teams using the design-based methodology. The hypotheses for this study are as follow:

- **H_1:** The XP team will experience a higher increase in the positive temperament than the design team at the end of the project.
- **H_2:** The number of the XP practices used is positively related to the level of the positive temperament experienced by the SE members.

To test these hypotheses, experiments to compare the effect of using two different methodologies, were carried out in 2003 and 2004. There were six projects involved in this experiment.

2 The Experiment

The experiment was carried out by requiring half of the development teams to use the XP methodology [2] and the other half to use the Discovery [3]. During the first

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 218–221, 2005.
© Springer-Verlag Berlin Heidelberg 2005

week, all of the students were given a team management course to familiarize with managing a group project. Each project was overseen by a manager who monitored the progress of both types of team. The questionnaire on the positive temperament was administered during this week because the assignment of the team was completed but the application of the methodology has not commenced. The data collected represented the level of the positive feeling experienced by the SE teams before the methodology treatment. Throughout the next 10 weeks, the SE teams attended separate lecture and lab sessions. The XP teams were introduced to the methodology during the lectures and were coached on applying the practices during the lab sessions. When the XP teams attended the lectures or lab sessions, the design-based teams met the client. After the first hour, the two types of teams switched places. The second reading of the positive temperament was taken before the final delivery of the system.

To measure the developers state of positive temperament, a questionnaire on the positive temperament scale of the Positive and Negative Affect Schedule (PANAS)[4] was used. The scale was developed and improved by Watson and colleagues [1]. The validity and reliability of this scale has been demonstrated in other studies [4-6]. Measurement was on a 5-point scale, and the respondents were required to indicate the degree of agreement and disagreement with each item. Responses range from 1 “very slightly” or “not all” to 5 “very” or “extremely”. The experiments do not include the negative moods because positive moods operated as a single construct indicating that the fluctuation of positive moods has no effect on the negative moods of a person. During the experiment, the Positive Affect scale showed satisfactory internal consistency coefficient, Cronbach $\alpha = 0.9023$ during the first reading and $\alpha = 0.8700$ during the second reading. Cronbach alpha provides an assessment of the internal consistency of all the items used in the scale.

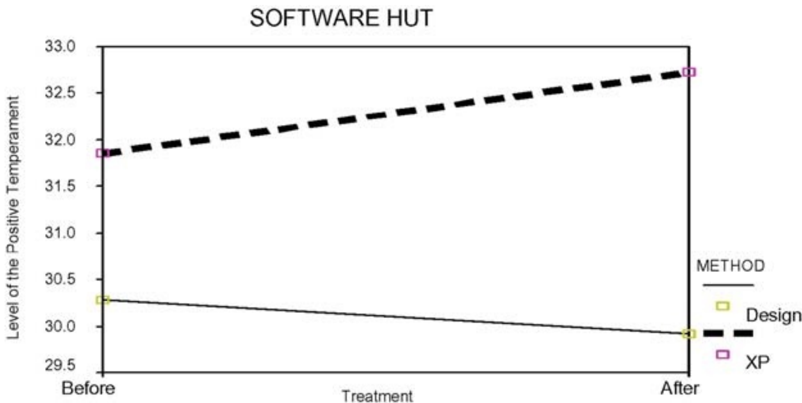


Fig. 1. Bar graph showing the difference in the positive moods of the two teams before and after the methodology treatment

SPSS package was used to analyze the data. To determine whether there was any difference in the level of the positive temperament between the two types of teams, an independent T–test was conducted. At the beginning of the experiment, analysis

showed that all of the developers have a similiar temperament [the design-based team (N = 50, Mean score = 30.28, SD = 11.85) and the XP team (N = 47, Mean score = 31.85, SD = 9.25)]. The analysis after the methodology treatment, indicated an increase in the positive affectivity level among the XP developers [the design-based team (Mean score = 29.92, SD = 7.876) and the XP team (Mean score = 32.72, SD = 8.070), significant at 0.087 (< 10 % level)]. The findings supported the hypothesis, that by incorporating most of the practices, the XP team was able to experience a significantly higher positive temperament than the design-based team. The relationship between XP practices and positive temperament, was investigated using the Pearson Correlation test. There was a significant relationship between the two variables for every project [r = 0.331, p = 0.030] indicating that the higher the number of XP practices used, the higher is the level of positive temperament experienced. This result supported the second hypothesis that the more XP practices used, there is greater tendency for the team to experience the feeling of positive emotions such as joy, interest, contentment, pride, broadening the scope of thinking and actions.

3 Discussion

When a person experiences positive temperament, they show a greater preference for a larger variety of actions and are able think of more possibilities and options to solve whatever problem is faced [7]. People with a positive temperament are more likely to take action because they are proactive, thus they tend to approach and explore novel objects, people and situations under such emotions. Studies have shown that people must have a surplus of resources such as time, energy and attention to engage in a proactive behaviour. The XP teams experienced a surplus of time during the coding phase because less time was engaged in the designing phase [8]. It was also observed, that the practice of pair programming started with the initial socializing among the pair, thus creating a positive mood amongst them before any formal programming commenced. The positive mood, which is experienced and the attention of two developers allowed the pair to engage in a more proactive behaviour. The ability to discuss the advantages and disadvantages of certain coding ideas enabled the pair to seek improvements and to avoid specific weaknesses. Previous work suggests that when people experience joy and contentment, they are more likely to think of a wider range of action [9] as an earlier study on the productivity of XP teams has shown [10], become more resilient over time and are more likely to develop long-term plans and goals. A study by Brief at al [11] has shown a consistent finding with past research, that positive mood induction has been found to yield increases in pro-social behaviour and has been found to be positively associated with job satisfaction. The second finding showed a strong relationship between positive moods experienced and the number of XP practices used. The findings support earlier studies, that the more practices were incorporated in the development process, the higher is the positive emotion experienced by the developers and therefore they were more productive [10]. This is expected because of the existence of the respective practices such as simple design, pair programming, continuous testing, continuous integration and frequent review (release) that command feedback. This finding helps to provide empirical evidence that the XP methodology does have an impact on the positive temperament of developers.

References

1. Watson, D. and L.A. Clark, Negative Affectivity: The disposition to experience aversive emotional states. Psychological Bulletin, 1984. **96**: p. 465-490.
2. Beck, K., Extreme Programming Explained: Embrace Change. 2000, NJ: Addison-Wesley. 190.
3. Simons, A.J.H. Object Discovery: A process for developing medium sized applications. in Tutorial 14, ECOOP 1998. 1998. Brussels :: AITO/ACM.
4. Watson, D. and A. Tellegen, Towards a consensual structure of mood. Psychological Bulletin, 1985. **98**: p. 219-235.
5. Watson, D., J.W. Pennebaker, and R. Folger, Beyond negative affectivity: Measuring stress and satisfaction in the workforce. Journal of Organizational Behavior Management, 1987. **8**(2): p. 141-157.
6. Watson, D. and L.A. Clark, Extraversion and its positive emotional core, in Handbook of Personality Psychology, R. Hogan and J.A. Johnson, Editors. 1997, Academic Press: San Diego, Ca. p. 767-793.
7. Isen, A.M., K.A. Daubman, and G.P. Nowicki, Positive affect facilitates creative problem-solving. Journal of Personality and Social Psychology, 1987. **52**: p. 1122-1131.
8. Macias, F., Emprical Assessment of the Extreme Programming (PhD thesis), in Dept of Computer Science. 2004, University of Sheffield: Sheffield. p. 189.
9. Syed-Abdullah, S., M. Holcombe, and M. Gheorge, The Impact of an Agile Methodology on the Well being of Development Teams. Empirical Software Engineering (to appear), 2005.
10. Syed-Abdullah, S., M. Holcombe, and M. Gheorge, Practice makes Perfect, in Lecture Notes in Computer Science LNCS 2675, M. Marchesi and G. Succi, Editors. 2003, Springer-Verlag: Berlin Heidelberg, Germany. p. 354-356.
11. Brief, A.P., A.H. Butcher, and L. Roberson, Cookies, Disposition, and Job Attitudes:The Effects of Positive Mood-Inducing Events and Negative Affectivity on Job Satisfaction in a Field Experiment. Oranizational Behavior and Human Decision Processes, 1995. **62**(1): p. 55-62.

Adjusting to XP: Observational Studies of Inexperienced Developers

John Karn, Tony Cowling, Sharifah Lailee Syed-Abdullah, and Mike Holcombe

Department of Computer Science, University of Sheffield,
Regent Court, Portobello Street, Sheffield S1 4DP, UK
{a.cowling,j.karn,s.abdullah,m.holcombe}@dcs.shef.ac.uk

Abstract. Extreme programming (XP) has been introduced in various scenarios primarily because some in industry argued for a move away from what they feel are rigid documentation-based development techniques. This has usually taken place with experienced developers. This paper describes attempts by researchers from the University of Sheffield to introduce XP to relatively inexperienced student developers. This paper describes some of the important findings and provides evidence relating to common problems encountered when students attempt to adjust to XP.

Keywords: Extreme Programming, Agile Methods, Empirical Software Engineering

1 Introduction

This paper describes ongoing attempts to introduce XP into highlights some of the main problems encountered along the way. student group projects at the University of Sheffield and

The context for this study is the Software Engineering Observatory [1], a research facility which is run by the Verification and Testing (VT) research group. The Observatory constitutes of several projects: the ones relevant to this research are the Software Hut (SH), which is a second year undergraduate project that runs for one semester, and Genesys Solutions (GS), which is an MSc project spanning the entire academic year.

XP was first introduced in 2000. Holcombe *et al* [2], [1], examined some effects of this by comparing the systems produced in the SH project by XP teams and those using Discovery [3], and found no significant difference between the groups. Also Syed-Abdullah [4], [5] identified a number of initial problems with XP, particularly concerning levels of understanding of this new methodology and of motivation to use it.

This paper presents the results of these investigations, section 2 of the paper describes research methods used, section 3 presents the results, section 4 discusses the results and section 5 describes conclusions.

2 Research Methods

The primary method used was ethnography, and then others namely focus group interviews, document analysis, and questionnaires provided triangulation. The use of

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 222–225, 2005.
© Springer-Verlag Berlin Heidelberg 2005

ethnographic methods provided evidence both of how the team had operated and their attitudes to the work, with particular focus on their attitude toward XP.

3 Results

3.1 Introduction of XP

Analysis of the 12 practices adopted by the teams in the SH 2002 and GS 2001 showed that teams readily adopted practices that were descended from a previous methodology, including coding standards, continuous integration and testing. Practicing test-first programming proved to be a difficult task. This was due to the fact that most of the students had been taught that testing is the final phase in the software development life cycle. Students also found it difficult to understand the 12 practices of XP because of the inter-relationship of each practice with each other, therefore it was essential to reduce the complexity of the relationship in order for them to gain a greater understanding of the methodology. Another common problem was that lack of experience prevented students from developing and producing complete story cards. This particularly led to problems within GS. As GS is made up of MSc and fourth year students the entire work force is replaced every year. Therefore the company is in dire need of effective communication methods to convey suffiicient information from previous developers to the new teams.

XP literature streses that in order to see the full advantage of XP all 12 practices must be adopted. During the SH 2002 teams were required to apply full adoption. At the end of the project group interviews found that they encountered difficulties in understanding all of the practices. This prompted a change, in 2003 the researchers decided on a partial adaptation, poor results forced the project managers to revert to full XP practices in 2004. This particular issue remains unresolved although the current preference is to opt for all XP practices. The onsite customer also proved to be problematic. Throughout the study all of the clients were reluctant to be stationed on site for more than one third of the development time. The second factor was the existence of several user groups. The process of achieving compromise between groups can be a harrowing experience for developers. The other problematic practice is the system metaphor. The students preference was to use a display-based problem solving approach as opposed to a metaphor as they felt this would aid them in acheiving a common vision with their clients. This failure to use the metaphor provided support to the proponents of XP, who dropped this practice from the latest version of XP.

3.2 XP and Software Hut Teams

XP proved to be a major source of discontent for the selected SH team. Discussing story-cards caused a lot of stress. The primary factors for this problem were sarcasm, a lack of understanding and hair-splitting pedantry which kept forcing people to backtrack and explain the same thing 2 or 3 times. The other problems relating to XP were general team complaints about the methodology. One member advocated doing what needed to be done rather than being so rigid with XP. In the case of the XP team, they were new to XP and they didn't feel that it was fair that they had been allocated this methodology. In comparison the Discovery group did not have any explicit method-

ology problems. A crucial factor was that the students had studied Discovery in great depth prior to the SH project. This meant that they didn't have to worry about learning a new methodology as well as trying to adjust to high-pressure group work and trying to meet the requirements of a real industrial client.

3.3 XP and Genesys Teams

3.3.1 Genesys Team 1

This team never encountered any serious problems with XP. The test first strategy proved to be hard for some members of the team to master. Other team members could see the potential problems, but worked hard to enforce the test-first practice. An automated testing environment (PHP Unit) was eventually used to help with this. This team also had initial problems with pair programming. This was mainly due to the management enforcing that all teams swapped pairs at regular intervals throughout the project. Due to this, the team had to organize 8 hours per week when they would all be in the office together. There was an interesting range of opinions on story-cards. A general opinion was that it would have been helpful to have had a project management tool, so the cards could be swapped around according to complexity. Despite this, they agreed that they were using XP effectively. The main XP problem facing them revolved around the perception they were being thrown in at the deep end. These concerns were eased by other members who were experienced with XP.

3.3.2 Genesys Team 2

This team had a lot of problems getting to grips with the issue of documentation in an XP project; another problem was that there was a lot of disagreement about the use of story-cards. More story-cards were needed as the system went on, the team was unwilling to show additional cards to the client, changes had to be made to supplement the original story-cards and the team was forced to admit that they were not using story-cards correctly. Pair programming was not fully practiced whereby the team members were supposed to pair as much as possible and in addition to change the partners. Timetable and personality clashes caused problems and made effective pair programming difficult. The team argued that it would have been good to have done an upfront design and forcing the client to stick to it, they argued that keeping to XP had encouraged the client to keep changing his mind. They had a lot of trouble adjusting to some of the XP practices for a variety of reasons including being trained in other methodologies, personality differences, time-table clashes, and apathy.

4 Conclusions

There is a clear need for more consideration to be given to a careful introduction of XP to students coming to it for the first time. A common theme was that students felt they were being thrown in at the deep end, and that following a new methodology, as well as working towards the completion of a project was an intolerable burden. A positive finding was that some students were knowledgeable enough to debate aspects of XP. This showed that they were thinking about the methodology and were in a position to form cogent opinions on it, whether. It is important to specify that one cannot isolate problems with the methodology from overall project problems. It is

possible to experience communication, personality, client or managerial problems, which would blight any team regardless of the methodology.

References

1. M. Holcombe, M. Gheorge, and F. Macias, "Functional Test Set Generation for Extreme Programming," Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP 2001), pp. 109-113, 2001.
2. F. Macias, M. Holcombe, and M. Gheorge, "Empirical experiments with XP," Proceedings of XP 2002, pp. 225-228, 2002.
3. A. J. H. Simons, Object Discovery- A Process for Developing Medium Sized Applications. Brussels Belgium, 1998.
4. S. Syed-Abdullah, M. Holcombe, and M. Gheorge, "Practice makes perfect," Proceedings of XP 2003, pp. 161-169, 2003.
5. S. Syed-Abdullah, M. Holcombe, and M. Gheorge, "Agile Methodology in Practice," Department of Computer Science Research Report CS-03-04, University of Sheffield, 2004.

An Agile and Extensible Code Generation Framework

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York, UK
{dkolovos,paige,fiona}@cs.york.ac.uk

Abstract. Code generation automatically produces executable code by software. Model-driven code generation is currently the most flexible and scalable generative technique, but there are many complaints about the complexity it introduces into the development process, and the design decisions imposed on the code. Here, an agile code-oriented model-driven generative methodology is outlined that reduces complexity and allows the engineer to define the exact form of the produced code and embrace change in the requirements in an automated manner. A flexible tool, ECGF, supports this methodology, and a case study in rapid generation of large-scale HTML documents is outlined.

1 Introduction

Code generation is the automated production of code by software. In the context of this paper, a special type of code generation called *model-driven code generation* is examined. This concept is to design systems using abstract representations (models) and then, based on the models, to automatically derive working code.

There are two approaches to model-driven code generation. In the first model-driven approach, the engineer designs the system in terms of abstract models (e.g., in UML [10]) and then generates code based on them. In the *code-oriented* approach (COMMD), the engineer designs part of the code of the system and then creates models that can be used to generate the rest of the code needed.

Code generation tools available today, such as Rational Rose, support generation from abstract models, e.g., in UML, and for particular popular application domains, such as enterprise data-driven systems. Limited tool support exists for the code-oriented model-driven approach. A generic, unified tool support structure would be beneficial for this approach, and to support agile development techniques such as Extreme Programming [8]. COMMD is compatible with the tenets of agile development, as it is code-based and accommodates changing requirements as well as the need for flexibility in software.

We describe a COMDD methodology and a supporting tool. The tool, called the Extensible Code-Generation Framework (ECGF), is tailored for use in agile development approaches. We demonstrate the flexibility and power of ECGF via a case study in the agile development of large-scale systems.

2 Code Generation and Agility

There are many reasons why the structure or the functionality of code should be modified (even radically) during the development process. In these situations it is

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 226–229, 2005.
© Springer-Verlag Berlin Heidelberg 2005

important to have an agile process that embraces change. The advantage of template based code generators is that the information on the functionality and structure of the code exists in the models and the templates respectively. To embrace a change the engineer has to modify the templates and/or the models and regenerate the code to fit the new requirements.

3 A Code-Oriented Model-Driven Methodology

Agile development shifts attention from models (e.g., in UML) to code. The rationale of this approach is that the quality of the code, and not that of the models, finally determines the success of the development process. With this principle in mind a COMDD methodology is outlined.

1. **Write a prototype by hand:** this is radically different to most model-driven development techniques, but similar to agile methods, since development starts with coding. The aim with this approach is to allow the generation of hand-written quality source code rather than non-human-modifiable and tool-specific code.
2. **Decide on applicability:** In order for a system to be suitable for code generation it should contain large amounts of repetitive code, which cannot or is not preferable to be eliminated by changes in the design.
3. **Identify common and variable parts of the code:** Models will represent the variable code parts, while templates will represent the common parts.
4. **Identify scope:** Thus, in this step the system is analyzed to determine which parts to assign to the code generator, keeping in mind that the more tasks that a code generator must do, the more complex it – and the resulting code – will become.
5. **Create models and templates:** Our methodology implies that the engineers compose the templates and the models themselves. Composing the templates and models manually ensures that they fit the desired structure of the system and that produced code follows the coding and quality standards that the engineer has set.
6. **Reproduce prototype using the code generator:** the next step is to reproduce the prototype system using the models and the templates. This reveals aspects that might have been mistreated, and builds confidence that the code generator can actually generate the desired code.
7. **Generate the final system:** The final step is to build the models and generate the rest of the modules of the system.

As the process is agile, it is important that the generated code is compiled and tested after each generation, to ensure that it does not contain syntactical and logical errors. Test skeletons, but not implementations, could be derived from the models in order to avoid error propagation from models to tests.

There are two additional points to note about the methodology. First, the development process needs careful management and coaching in terms of the use of templates and models using a configuration management system so that all the developers work with the same versions of the templates. The second point is that models and templates from a successful development process can be reused to generate similar software in the future. Of course, these artifacts need to be viewed critically and may need refactoring for different projects.

4 ECGF: The Extensible Code Generation Framework

ECGF is a tool to support the agile COMMD methodology outlined in Section 3. The tool has been designed to be usable by technical developers; this is an important quality attribute, since many developers are anecdotally averse to existing code generators, considering them inflexible to use. Usability is achieved by providing a simple integrated development environment (IDE). Since template and model construction, as well as code generation, are error-prone tasks, ECGF also provides detailed feedback about errors detected.

4.1 ECGF Framework

ECGF builds on an open-source template engine, the part of the system that interprets and executes the templates in order to generate the code. A simple scripting language called Velocity Template Language is used for writing templates. Scripting languages are most appropriate for rapid template development because they provide a brief and powerful syntax without the type-casting overhead of strongly typed languages.

General requirements for a suitable language for agile modeling include: extensibility, mechanisms for model validation, and a substantial existing user basis. The two alternatives that meet the requirements to some extents are XML and UML. UML is undesirable because of its semantic fragmentation, and the need to use heavyweight UML metamodel extension. XML has a simple syntax and provides mechanisms for validation, queries and transformations. Moreover, there are numerous high-quality open-source XML parsers that implement these features. Thus, ECGF supports XML as a modelling notation.

In order to integrate XML models with Velocity Templates we use a mechanism that combines the XML Document Object Model (DOM) [7] with pluggable Velocity introspection [5]. Pluggable introspection allows overriding the default method invocation behavior of the template engine. In this way we achieve an elegant API for iterating and managing the DOM that makes it look and feel like a custom object model. We briefly explain this mechanism below.

Consider a simple Velocity expression: **`$a.b`**. This expression is interpreted as following *"if `a` is an XML element, first see if it has an attribute named `b` and if so return the value of `b`"*. By this mechanism, we shield the user from metamodelling but still obtain the benefits from them, via introspection.

5 Generating Browsable Documentation for Modelware

To illustrate the usability of the methodology, we briefly outline a recent application of ECGF in an ongoing EU project, "Modelware" [11]. As part of Modelware, we need to spend many hours traversing the UML metamodel [9] as represented in XMI. Given the complexity of the physical structure of XMI, this is time-consuming and tedious, and it was decided to generate a browsable HTML-based navigation utility.

The first step was to manually create a sample class description page and a sample package description page as well as the frame page that would contain class and package pages (prototyping). When we stabilized the desired content and format of each page type, we composed the respective templates and produced the rest of the navigation utility.

The product consists of 147 web-pages which would have possibly taken weeks to compose by hand while the design, implementation and testing of the demonstrated solution required less than a day. We then used the same templates to generate identical mechanisms for the navigation of Common Warehouse Metamodel (CWM) and Meta Object Facility (MOF 1.4).

6 Conclusions

We have outlined an agile code generation technique, along with a supporting tool, that enables rapid development in an agile way. We have applied the approach in a number of practical case studies. The tool, ECGF, has proven to be usable, efficient, and practical in the large. We are currently using ECGF in the Modelware project and elsewhere for a variety of code generation tasks. We are also considering adding further features to ECGF, particularly pattern recognition for lightweight assistance in detecting common code, and also better visual support for model editing.

References

1. Sal Mangano. XSLT Cook Book. O' Reilly, 2003.
2. Jack Herrington. *Code generation in Action*. Manning, 2004.
3. OMG. Model driven architecture official web-site. http://www.omg.org/mda/.
4. XDoclet development team. http://xdoclet.sourceforge.net.
5. Velocity development team. http://jakarta.apache.org/velocity.
6. Code generation is a design smell. http://c2.com/cgi/wiki?CodeGenerationIsaDesignSmell.
7. Eric van der Vlist. XML Schema. O' Reilly, 2002.
8. Kent Beck. *Extreme Programming Explained*. AWL, 1999.
9. OMG. UML 1.4 Metamodel. http://www.omg.org/uml.
10. OMG. CWM Metamodel. http://www.omg.org/cwm.
11. Modelware Integrated Project. http://www.modelware-ist.org

UC Workbench – A Tool for Writing Use Cases and Generating Mockups

Jerzy Nawrocki and Łukasz Olek

Poznań University of Technology, Institute of Computing Science,
ul. Piotrowo 3A, 60-965 Poznań, Poland
{Jerzy.Nawrocki,Lukasz.Olek}@cs.put.poznan.pl

Abstract. Agile methodologies are based on effective communication with the customer. The ideal case is XP's on-site customer. Unfortunately, in practice customer representatives are too busy to work with the development team all the time. Moreover, frequently there are many of them and each representative has only partial domain knowledge. To cope with this we introduced to our projects a proxy-customer role resembling RUP's Analyst and we equipped him with a tool, called UC Workbench, that supports the communication with the customer representatives and the developers. Analyst collects user stories from customer representatives and 'translates' them into use cases. UC Workbench contains among other things a use-case editor and a generator of mockups (a mockup generated by UC Workbench animates use-cases and illustrates them with screen designs).

1 Introduction

Many agile methodologies use informal stories for requirements description [3]. Perhaps the most popular are XP's user stories [2] [5]. An interesting alternative to them are use cases invented by Ivar Jacobson [7] and incorporated into Rational Unified Process. They provide "*a semiformal framework for structuring the stories*" [1]. Although they are more formal than user stories, they are not contradictory with the agile approach to software development. There are some agile methodologies that use written requirements or permit them (e.g. the Crystal methodologies [4], DSDM [14] or Scrum [13]). For people applying those methodologies, use cases can be very useful. As Craig Larman put it: "*when written functional requirements are needed, consider use cases*" [9].

Use-cases engineering comprises *editing* the use cases (e.g. inserting a step can require re-numbering all the subsequent steps and extensions associated with them – that could be done automatically) and *generating mockups* that would animate the use cases (that would support customer-developers communication), and *preparing effort calculators* based on Use-Case Points [11] and adjusted to the current set of use cases (that could be a valuable tool supporting XP's Planning Game or backlog's effort estimation in Scrum). Unfortunately, there is no tool offering that range of functionality. The only thing we have found was a use-case editor, called CaseComplete, offered by Serlio Software [16].

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 230–234, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The aim of the paper is to describe our approach to use-case engineering. It is based on a tool called *UC Workbench* (Use Case Workbench) developed at the Poznan University of Technology. It is a use-case editor combined with a mockups generator and an effort-calculators generator. It is important for an agile team that after changing some use cases a new mockup and updated effort calculators are obtained 'at the press of a button'. UC Workbench has been successfully used at the university's Software Development Studio and in a commercial project run by PB Polsoft, a medium size software company with headquarters in Poznan. In the paper we focus on the language for use-cases description (Sec. 2.), and generation of mockups (Sec. 3.). Other functionality provided by the tool includes automatic inspections and support for effort estimation.

2 FUSE: A Language for Use-Cases Description

Use cases collected by an analyst are edited with the help of UC Editor, a part of UC Workbench. The editor uses FUSE (Formal USE cases) language. It is a simple language formalizing structure of use-cases description to allow generating of mockups and effort calculators (actor descriptions and steps within the use cases are expressed in a natural language).

FUSE is based on use-case patterns collected by Adolph and his colleagues [1]. Use cases are accompanied with actor descriptions (that is adviced by the Clear-CastOfCharacters pattern). FUSE allows for two forms of use cases (the MultipleForms pattern): *casual* and *formal*. The former has no steps, just plain text and resembles XP's user stories. The latter corresponds to the ScenarioPlusFragments pattern: the description consists of a main scenario split into a number of steps and extensions (that form is very popular – see e.g. [6]). When applying the breadth-before-depth strategy and spiral development, first the casual form is used and at the next cycle some use cases are refined and written down using the formal form.

To make formal descriptions of use cases more precise and readable we have decided to introduce the Either-Or construct to FUSE. Moreover, for the sake of readability FUSE allows nested steps that are especially helpful when combined with Either-Or.

2.1 Either-Or

Sometimes one needs *nondeterministic choice* between alternative steps. Assume, for instance, that someone has to describe how to buy books in an Internet-based bookstore. A customer can either *add a book to the cart* or *remove one from it*.

Putting those two steps in a sequence

```
1. Customer adds a book to the cart.
2. Customer removes a book from the cart.
```

is not correct, as it suggests that one has always to remove a book adding one to the cart. A remedy could be to add an extension like the following one

```
2a. Customer does not want to remove a book from the cart.
    2a1. Customer skips the step.
```

to each of the above steps. Unfortunately, that would clutter the description and other really important extensions would be less visible.

To solve the problem we have decided to introduce the Either-Or construct to the formal form of use cases. Using it one could describe customer's options in the following way:

```
1. Either: Customer adds a book to the cart.
2. Or:     Customer removes a book from the cart.
```

One can argue that the Either-Or construct can be difficult for some end-users to understand. In the case of UC Workbench it should not be a problem, as the use cases are accompanied with an automatically generated mockup which visualizes the control flow by animating of use cases.

2.2 Nested Steps

Sometimes a step can be decomposed into 2 or 3 other steps. Then it can be convenient to have the "substeps" shown directly in the upper level use case (according to the LeveledSteps pattern a scenario should contain from 3 to 9 steps). For instance, assume that adding a book to the cart consists of the following "substeps":

```
1. Customer selects a book.
2. System shows new value of the cart.
```

Then buying books could be described in FUSE in the following way (use-case header and extensions have been omitted):

```
1. Either: Customer adds a book to the cart.
           1.1. Customer selects a book.
           1.2. System shows new value of the cart.
2. Or:     Customer removes a book from the cart.
```

Again, end-user will be supported with a mockup helping him to understand the control flow, but if the analyst thinks it is not enough she can always choose not to use new constructs.

An example of use case written in FUSE is presented below:

```
Main scenario:
  1. Customer opens main page of a Bookshop.
  2. System presents a list of categories and all new positions.
  3. Customer is composing his order:
    3.1. Either: Customer adds a book to his cart:
      3.1.1. Customer chooses a desired book.
      3.1.2. System shows the book details.
      3.1.3. Customer adds the book to cart.
    3.2. Or: Customer removes a book from the cart.
  4. Customer finalizes the order.
Extensions:
  4.A. The cart is empty.
    4.A.1. System shows appropriate message and returns to step 3.
```

3 Generating of Mockups

There are two kinds of prototypes [10]: throwaway prototypes (mockups) and evolutionary ones. The latter are core of every agile methodology. The former could be used to support customer-developers communication about requirements but their development had to be very cheap and very fast.

The mockups generated by UC Workbench are simple but effective. They focus on presenting functionality. They combine use cases (i.e. behavioural description) with screen designs associated with them (that complies with the Adornments pattern [1]). A generated mockup is based on a web browser and it consists of two frames (see Fig. 1):

– the *scenario window* presents the currently animated use cases (it is the left frame in Fig. 1) and the current step is shown in bold;
– the *screen window* shows the screen design associated with the current step (it is the right frame in Fig. 1).

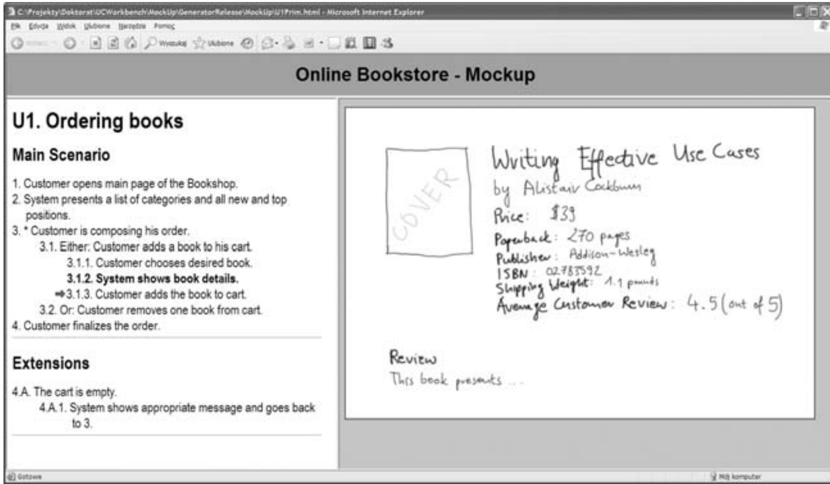


Fig. 1. A Mockup Screen

To generate a mockup the analyst has to associate with use-case steps names of files containing screen designs. The fidelity levels of screen designs are up to the analyst (an interesting discussion about fidelity levels can be found in [12], [15], [8]). The screen design shown in Fig. 1 is at the low fidelity level and it has been created with a tablet connected to PC. After decorating 'difficult' use-cases with screen designs one can generate a mockup "at the press of a button". Using UC Workbench one can produce a mockup (i.e. re-write use cases and adorn them with screen designs) within a few hours – that goes well with agile methodologies and can really support customer-developers communication, especially if there is a danger that the requirements are ambiguous or contradictory.

4 Conclusions

UC Workbench presented in the paper supports editing use cases (we were supprized that Rational Requisite Pro much more supports 'traditional' requirements than use cases) and generates mockups that animate use cases. What is important for agile developers a mockup can be obtained automatically, so it is always consistent with changing requirements. *UC Workbench* supports also automatic inspections, effort estimation based on Use Case Points [11] and generation of the software requirements document from a set of use cases.

Acknowledgements. We would like to thank Grzegorz Leopold and Piotr Godek from PB Polsoft – their bravity allowed us to get a feedback from industry and helped us to improve the tool. This work has been financially supported by the State Committee for Scientific Research as a research grant 4 T11F 001 23 (years 2002-2005).

References

1. Adolph S., Bramble P., Cockburn A., Pols A.: Patterns for Effective Use Cases. Addison-Wesley (2002)
2. Beck, K.: Extreme Programming Explained. Embrace Change. Addison-Wesley, Boston, 2000
3. Boehm, B., Turner, R: Balancing Agility and Discipline. A Guide for the Perplexed. Addison-Wesley, Boston, 2004
4. Cockburn, A.: Agile Software Development. Addison-Wesley, Boston, 2002.
5. Cohn, M.: User Stories Applied. Addison-Wesley, Boston, 2004.
6. Fowler, M., Scott, K.: UML Distilled. Addison-Wesley, Boston, 2000.
7. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading MA, 1992.
8. Landay J.A.: SILK: Sketching Interfaces Like Crazy IEEE Computer-Human Interaction (April 13-18, 1996)
9. Larman, C.: Agile And Iterative Development. A Manager's Guide. Addison-Wesley, Boston, 2004.
10. Pressman, R.: Software Engineering. A Practitioner's Approach. McGrow-Hill, New York, 1997.
11. Ribu K.: Estimating Object-Oriented Software Projects with Use Cases Master of Science Thesis, University of Oslo 2001
12. Rittig M.: Prototyping for Tiny Fingers. Communications of the ACM, April 1994/Vol. 37, No. 4 p. 21-27
13. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press, Redmond, 2004.
14. Stapleton, J.: DSDM. Business Focused Development. Addison-Wesley, London, 2003.
15. Walker M., Takayama L., Landay J.A.: High-fidelity or Low-fidelity, Paper or Computer? Choosing Attributes When Testing Web Prototypes. Proceedings of the Human Factors and Ergonomics Society 46th Annual Meeting: HFES2002. pp. 661-665
16. http://www.serliosoft.com/casecomplete/

Desperately Seeking Metaphor

Ben Aveling

Alcatel Australia
ben.aveling@alcatel.com.au

Abstract. This paper shows how System Metaphor delivers a coherent system of names that carry more information than unrelated names would.

Keywords: Extreme Programming, Metaphor, Names.

Introduction

In the White Book (Beck, 2000) System Metaphor was 'the story everyone could tell about the system.' While the term is not used in the second edition it remains explicit that 'names are drawn from a consistent set of metaphors' (2004a, p26). Metaphor has been widely studied, particularly by linguists. It has many uses beyond the scope of this paper, particularly for communicating and exploring concepts and as a rhetorical device. This paper investigates the use of metaphor as a source of names and seeks to demystify System Metaphor.

Metaphor

A metaphor is an expression taken out of its normal context, or *source* context, whilst preserving some aspect of its original meaning in the new context, or *target* context, as shown in figure 1. For example, 'trees have branches, which have leaves'. Trees arboreal and tree data structures are different but the sentence makes sense in both contexts and has basically the same meaning in each context, even though the meaning of almost every word used differs with context. The shared vocabulary of trees encourages us to see ways in which all trees are alike.

To linguists, the source context is the *source domain* or the *vehicle domain* and the target context is the *target domain* or *topic domain*. Terms used in both domains are *bridge terms*. Terms only used in the source domain are *unused terms*. The reader seeking a formal treatment may wish to consult, amongst many others, Kittay (1987) or Lakoff and Johnson (1980).

The System Metaphor

In a System Metaphor, the system being modelled is metaphorically seen as an specific instance of another system, or *unifying concept*. For example, a payroll system is an assembly line that assembles paychecks, a command line processor is a bakery that cooks requests, and a testing framework is a play that stars software.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 235–238, 2005.
© Springer-Verlag Berlin Heidelberg 2005

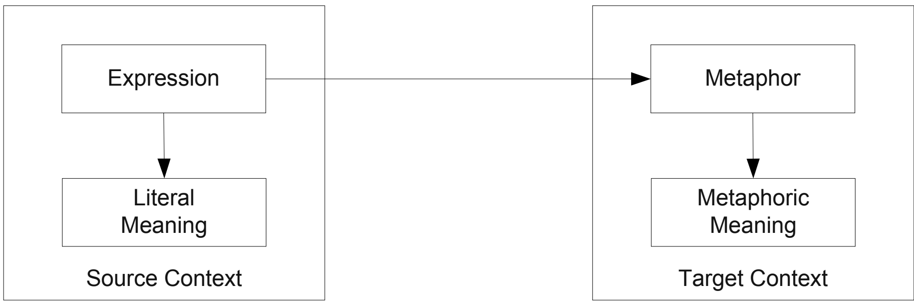


Fig. 1. A Metaphor is a expression that draws analogies between unlike things

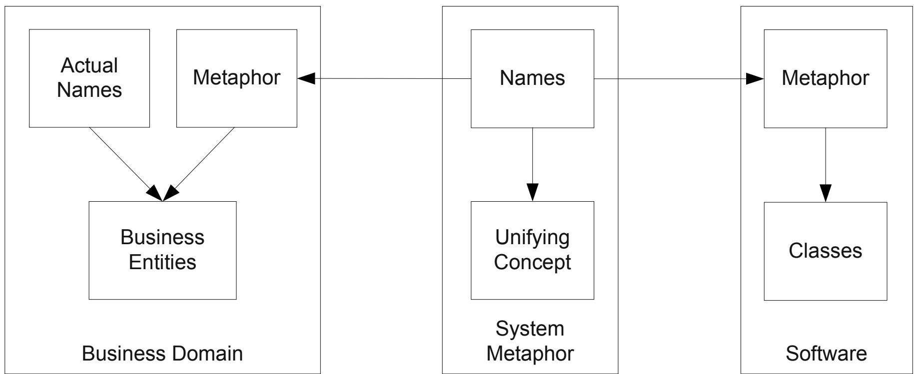


Fig. 2. Metaphor in XP

Figure 2 shows how the System Metaphor employs a unifying concept. The classes that compose the system's software are a model of the unifying concept. The business domain is, metaphorically, an instance of the unifying concept. Thus, the software is a model of the business domain even though it does not use actual names from the business domain.

The Stage – A Sample System Metaphor

By way of example, consider that a testing framework is, as mentioned earlier, a type of play (Richer, 2004). Each test has its Cast of Actors, such as HTTP actor, EJB actor, file system actor and JDBC actor. In a Scene, each Actor has a Script to follow. The Director and Stage Manager are responsible for getting Actors and Props on and off the Stage. The familiarity of the source domain allows software developers, and the reader, to intuit the role of each class with no further explanation.

The Native Metaphor

Finding a good unifying concept to use as a metaphor can be difficult. Instead, the Naïve Metaphor is often used. Like Ivar Jacobson's Naïve Object Model (cited in

Constantine 1995) and Eric Evan's Ubiquitous Language (2004), Beck's Naïve Metaphor (2000) uses terms from the business vocabulary to model the system. This provides no new insight into the business domain, does not introduce a syntactic distinction between the components of business domain and the model, and it moves the balance of power from developers to business. And yet, the word naïve is inappropriate.

The naïve metaphor is the only metaphor that can be guaranteed to map exactly to the business domain. It is understood by the business, proven thorough usage, and often well documented in academic, business and technical literature. I suggest instead the term *native metaphor*.

It has been suggested that the native metaphor is not really a metaphor because calling a thing by its own name is not metaphorical. As noted by Beck (2004b), an object is a collection of bits, not the entity it models. An object is a distinct thing that shares a common name. Conversely, the object exists only to model the thing; it lacks intrinsic purpose. Whether the native metaphor is a proper metaphor or merely metaphor like, it provides developers with a system of names into which to place classes.

The Value of Metaphor

Kent Beck (1997, p 168) writes that good names are: short, convey as much information as possible, are familiar and unique. The right word captures exactly a concept, any other word requires a supporting sentence. Finding the right word is often difficult, finding a good metaphor is harder still. However, a good metaphor can contain numerous names.

Names drawn from a good metaphor will be short, familiar and unique. They convey a great deal of information. They are not intrinsically better than other names but the metaphor gives extra meaning to its names. Instead of understanding each name in isolation, the reader need only understand the unifying theme to make sense of every name.

Conclusion

Metaphor has been described as "the XP practice that everyone excludes" (Beck, cited in Betty 2002). It is not so much that Metaphor is controversial. Rather, the role of metaphor is poorly understood (Aveling, 2004). This paper has attempted to address that confusion.

Metaphor provides a story everyone can tell about the system: how it decomposes into components, the responsibilities of each and the relationships between them. Programming is sometimes compared with building. If so, our raw material is names. Metaphor does not guarantee that our names will always capture concepts exactly but it does help us find names that click together.

> The question is not whether you will think metaphorically or not. The question is whether you will become aware of your metaphors and choose them consciously.
> — Kent Beck (2004b)

The programmer who is not consciously aware of metaphor will use metaphor sporadically, to choose a name here and there. The programmer with a deeper understanding of metaphor will actually use less different metaphors and they will draw many more terms from those metaphors.

Acknowledgments

This paper has benefited greatly from the input of Louis Richer, the discoverer of the test framework as play metaphor, and from Ron Jeffries' review of, and comments on, earlier versions of this paper. Thanks also to the anonymous reviewers of this paper.

References

Aveling, Ben (2004), 'XP Lite Considered Harmful?' In *Extreme Programming and Agile Processes in Software Engineering* (2004), Eds Eckstein & Baumeister, Berlin, Springer.
Beck, Kent (1997), *Smalltalk Best Practice Patterns*, New Jersey, Prentice Hall.
Beck, Kent (2000), *Extreme Programming Explained* (1st edition), Reading, Addison-Wesley.
Beck, Kent (2004a), *Extreme Programming Explained* (2nd edition), Reading, Addison-Wesley.
Beck, Kent (2004b), *RE: [XP] Metaphor Practice*,
emails to extremeprogramming@yahoogroups.com, 11/11/2004 and 16/12/2004.
Betty, Greg (2002), 'The Metaphor Metaphor',
Available at http://www.intelliware.ca/individual_messages/2002/11/400.html
Constantine, Larry (1995), *Constantine on Peopleware*, NJ, Yordon Press.
Evans, Eric (2004), *Domain Driven Design*, Boston, Addison Wesley.
Kittay, Eva (1987), *Metaphor, its Cognitive Force and Linguistic Structure*, New York, OUP.
Lakoff, George and Mark Johnson (1980), *Metaphors We Live By*, Chicago, University of Chicago Press.
Richer, Louis (2004), Personal communication.

Agile Testing of Location Based Services

Jiang Yu[1], Andrew Tappenden[1], Adam Geras[2], Michael Smith[2], and James Miller[1]

[1] University of Alberta, Edmonton, Alberta, Canada T6G 2E1
jm@ee.ualberta.ca
[2] University of Calgary, Calgary, Alberta, Canada T2N 1N4
{ageras,smithmr}@ucalgary.ca

Abstract. Mobile applications are increasingly location-based; *i.e.* their functionality is becoming both interactive and context-aware. Combined with an overall increase in the complexity of the devices delivering such services, and a growth in the number of possible networks that they can participate in, these systems require more than just the average approach to testing. The principles and practices of agile testing may serve development teams well here; since the systems ultimately end up being developed and deployed in an iterative and evolutionary manner. In this paper, we explore a testing framework for location-based services that can be employed test-first and yet also offers the full range of non-functional tests that these applications require.

1 Introduction

Location based services (LBS) are services that utilize information about the user's current geographical location [1]. LBS have received heightened interest since October 1994 when the FCC issued "FCC 94-102". This mandate was issued to ensure that any mobile phone sold in the US could be located geographically for wireless emergency services; hence every mobile phone sold in the US will possess the potential to utilize LBS. Strategy Analytics (March 2003) estimates that in 2008, LBS will generate over $8 Billion in global service revenues [2]. There is a need to match these changes, and this potential for success, with effective development and testing methods. Otherwise, the negative effects of low software usability experienced by e-commerce [3] will likely extend to m-commerce and LBS.

In this paper, we present a new framework for the testing of LBS that supports both agile and traditional testing. Agile testing means a number of things, but primarily it means that the entire team be involved in quality assurance. There is no single 'tester' role played assuming sole responsibility for quality; the responsibility is shared amongst all team members. Conversely, traditional testers focus on acceptance testing; and on transferring their testing skills to the rest of the team [4]. Agile testing also means automating acceptance tests and other tests as much as possible. Customers and end-users clarify their requirements by writing tests, and developers express their design intentions again by writing tests. By making these tests automated, the team enjoys both clear and precise direction on what is needed and a head start on their testing effort.

2 The Proposed Framework

The proposed test framework combines the mobile, network, and context emulators and allows for the testing of LBS to be automated. The test framework interfaces with

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 239–242, 2005.
© Springer-Verlag Berlin Heidelberg 2005

the different emulators using the TTCN-3 specification and implementation language [5]. TTCN-3 provides testers with the ability to rapidly prototype the test environment and thus to generate test cases quickly. The fundamental idea behind the test framework is to introduce re-usable components, well-defined system interfaces, and a scalable test environment, creating a suitable methodology for testing distributed systems such as LBS.

2.1 Test Framework Architecture

As shown in Figure 1, the test framework is divided into three layers: the functional layer, the control layer, and the system under test (SUT) layer. The functional layer includes several components that support the test framework, such as the test case generator, system centre, performance analyzer, frequency controller, context simulator, and user interface. The control layer contains the test system console and the system adapters. These components manage interactions between the functional layer components and the SUT. The SUT layer consists of the server and location based applications which run on handheld emulators.

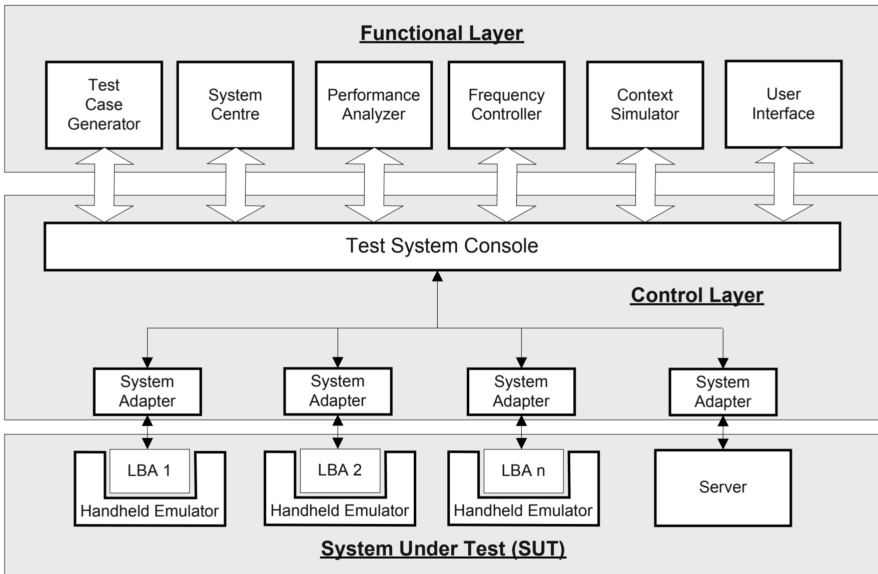


Fig. 1. Architecture view of the LBS test framework

A location-based application (LBA) is a mobile application that is developed for a particular mobile device and can be executed on a device emulator. A LBA communicates with the server through the test framework. The device emulator provides the ability to simulate wireless networks, and the test framework simulates context events. It is thus possible to simulate a wireless environment in which the mobile device will operate under different network conditions. The server also interacts with the LBA exclusively through the test framework. This allows for the testing of the server as well as the testing of the LBAs.

The context simulator (CS), implemented by integrating a third-party tool, Context Toolkit [6] is responsible for providing all location information to the LBAs via the test framework. The Context Toolkit provides the ability for the LBAs to access context information while hiding the details of context generation; permitting the LBAs to act as though they are interacting with the real environment, not an emulated one. The Context Toolkit uses HTTP and XML to support the transmission to the test system, allowing communication across a wide variety of platforms and devices.

The frequency controller (FC) enables the test framework to place the SUT under stress. The FC is used to control the frequency at which the CS provides context-events to the LBAs. This will put both the LBAs and the Server under stress as the frequency increases. By increasing the requests from the LBAs to the server, the breaking point of the system can be found and the system's performance evaluated.

The system centre (SC) component contains a description of the behavior of the SUT and rules for transferring the specified behavior into test cases. Using this information, the Test Case Generator (TCG) component can automatically generate test cases. The system centre is the primary location for input into the testing framework. With the rules and behavior set in the system centre, the test framework will automatically generate test cases and perform these tests on the system.

The performance analyzer (PA) is the component responsible for the measurement of performance for both the LBA and the server. The PA records the response times between the server and client and provides the tester with the ability to collect and interpret all performance data.

The test system console (TSC) manages the activities in the test framework and coordinates testing activities with the SUT and the other components such as the FC and CS. The TSC communicates with the SUT through ports defined in the SC. The TSC is responsible for the execution of all test cases provided by the TCG.

The system adapter (SA) is responsible for adapting all messages and procedure calls from the test environment to the LBAs and the server. The SA permits the test framework to be used for a wide variety of platforms and devices. The SA allows for the abstraction of the hardware specific components of the mobile devices or mobile device emulators and allows for the testing framework to be extended to new hardware elements.

2.2 Test Framework Implementation

The primary role of a tester in the software development process is the encoding of the behavior and the rules that apply to the system. This can be done before or after the SUT exists; a key aspect of agile testing. The required behavior is injected into the system using a scripting language and stored inside the SC. With the initial rules and behavior encoded, the test system will retrieve the information and extract a set of test cases along with other relevant information, such as the number of test adapters and specific test components required to execute the associated test cases. This data generated by the TCG is then wrapped in XML and used for system testing. These test cases can be executed automatically by the testing framework and can be used as acceptance tests for working code. Upon execution, the test framework will automatically bring together all relevant components including the LBAs, server, CS, FC and PA.

The test framework can also accept the manual creation of test cases. This allows for very specific test case generation, for example, writing test cases in a test-first manner. As mentioned earlier, tests written before the SUT may elaborate on requirements or express a design intention. The test framework can also stand alone or be used as a part of the daily build process in support of the agile practice of 'continuous integration'.

3 Conclusion

LBS and m-commerce are becoming an area of increasing interest to both software developers and academics. In the future, services that do not utilize context information to provide custom-tailored functionality will simply not meet users' expectations. The proposed testing framework provides an environment through which LBS can be verified and validated thus allowing for the future development of LBS and mobile applications. In addition to refining the test framework itself, future work will focus on providing an executable test notation that both customer/end-user testers and developer tests find readable and useable in an agile project context.

References

1. Adusei, I.K., K. Kyamakya,F. Erbas, *Location-based services: advances and challenges*, 2004.
2. Taylor, P., *Location Based Services: Strategic Outlook for Mobile Operators and Solutions Vendors*, in *Strategy Analytics*. 2003.
3. Nielsen, J., *Did Poor Usability Kill E-Commerce?* 2001, Jakob Nielsen.
4. Crispin, L., *XP Testing Without XP: Taking Advantage of Agile Testing Practices, in Methods and Tools*. 2003, Martinig & Associates.
5. ETSI, *Part 1: TTCN-3 Core Language*. 2003, European Telecommunications Standards Institute.
6. Dey, A.K. and G.D. Abowd. *The Context Toolkit: Aiding the Development of Context-Aware Applications*. in Workshop on Software Engineering for Wearable and Pervasive Computing. 2000. Limerick, Ireland.

Source Code Repositories and Agile Methods

Alberto Sillitti and Giancarlo Succi

Free University of Bozen
{Alberto.Sillitti,Giancarlo.Succi}@unibz.it

Abstract. Source repositories are a promising database of information about software projects. This paper proposes a tool to extract and summarize information from CVS logs in order to identify whether there are differences in the development approach of Agile and non-Agile teams. The tool aims to improve empirical investigation of the Agile Methods (AMs) without affecting the way developers write code. There are many claims about the benefits of AMs; however, these claims are seldom supported by empirical analysis. Configuration management systems contain a huge amount of quantitative data about a project. The retrieval and part of the analysis can be automated in order to get useful insights about the status and the evolution of the project. However, this task poses formidable challenges because the data source is not designed as a measurement tool. This paper proposes a tool for extracting and summarizing information from CVS (Concurrent Versions System) repositories and a set of analysis that can be useful to identify common or different behaviors.

1 Introduction

The main purpose of version control systems is to collect and manage source code effectively when there are several people modifying the same set of files. However, such systems collect a huge amount of information that does not include only source code.

Version control systems store precise information regarding the files created, keep track of the modifications introduced storing both timestamps and user names, etc. Such data are process data that can be useful to study the development process of an Agile team. Moreover, analyzing code repositories presents several advantages, such as:

1. **Data collection is non-invasive:** It does not require effort from the developers [2].
2. **Data are "for free":** All development teams are already using a version control system, therefore they already have a database that can be analyzed.
3. **Plenty of data available:** It is possible to analyze projects that are already finished or at any time of the development process. It is not required that the data collection starts with the project.

However, there are some drawbacks as well:

1. It is not possible to collect data regarding the effort spent for developing a piece of code.
2. It is not possible to trace the code that is developed and not checked in the repository (spikes, code developed and than deleted, etc.)

This paper presents CodeMart (CM), a tool for retrieving and analyzing data related to the software development process stored inside version control systems. The

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 243–246, 2005.
© Springer-Verlag Berlin Heidelberg 2005

paper is organized as follows: section 2 analyzes problems analyzing software repositories; section 3 presents the architecture of CodeMart; section 4 propose a set of analysis for data stored in code repositories; finally, section 5 draws the conclusions.

2 Analyzing Software Repositories

Source code repositories store useful process information, but they do not collect any data regarding the modifications that are stored. Systems such as CVS ask the developer to insert a short comment before checking in the code, but this approach presents two problems:

1. Most of the times developers do not insert any data.
2. If data are inserted, they are free text. Therefore, it is hard to understand for an automated system.

In order to overcome this limitation, we have added a classification mechanism to our data extraction tool [3].

A simple classification identifies three main types of modifications that developers can introduce in source code (Table 1).

Table 1. Classification of code modifications

Type	Code Identifier
Comment	Any changes in the code comments
Structural modification	Any changes in the code effecting execution paths (statements such as: if-then-else, for, do-while, switch, etc.)
Non-structural modification	Any changes in the code not effecting execution paths (any statements rather than statement included in the previous type)

Comments modifications include all changes that affect source code comments and do not modify any executable instructions.

Non-structural modifications include modifications of the source code instructions that do not change any execution paths of the program (all the function calls and the instructions except the flow control ones: if-then-else, for, do-while, switch, etc.).

Structural modifications include modifications of the source code that change execution paths of the program.

3 Architecture of CodeMart

CodeMart is a tool for data collection and analysis of data stored in version control systems. It is able to connect to software repositories and analyze the structure of the code and the sequences of operations performed by all the developers [1, 4].

The system includes:

- ***Data extractor:*** It accesses a version control system, extracts all available data, parses the source code, and finally stores both raw collected and processed data into the data warehouse system;
- ***Data analyzer:*** It performs analysis and shows results to the user through dynamically generated web pages, queries the data warehouse, collects answers, and displays data in different ways according to user preferences.

4 Sequence Analysis

In order to identify interesting development patterns, we propose to use a behavioral analysis of the developers in time. In particular, we have applied the concepts of the gamma analysis [5] used in the social sciences.

In this kind of analysis, a set of phases of the model are identified, than the sequence of the phases is analyzed. Phases are associated to the classification of the modifications described in Table 1.

The gamma analysis describes the order of the phases in the sequence and provides a measure of their overlapping. It is based on the gamma score defined as follows:

$$\gamma_{(A,B)} = \frac{P-Q}{P+Q}$$

where P is the number of A-phases preceding the B-phases and Q is the number of A-phases following the B-phases. The γ calculated in this way is symmetric and varies between -1 and +1. If $\gamma_{(A,B)} < 0$, the A-phases follow the B-phases; if $\gamma_{(A,B)} > 0$, the A-phases precede the B-phases; finally, if $\gamma_{(A,B)} = 0$, the A-phases and the B-phases are independent.

The gamma score is used to calculate the precedence score and the separation score. The precedence score id defined as follows:

$$\gamma_A = \frac{1}{N}\sum_i \gamma_{(A,i)}$$

where N is the number of phases and $\gamma_{(A,i)}$ is the gamma score calculated between the phases A and i. This score varies between -1 and +1.

The separation score is defined as follows:

$$s_A = \frac{1}{N}\sum_i \left|\gamma_{(A,i)}\right|$$

This score varies between 0 and +1. A separation score of 0 means that the phases are independent, while +1 means that there is a separation among the phases.

The values of γ and s are calculated for each file in the projects considered. Then, their values for the whole project are calculated as a weighted average as follows:

$$\gamma = \frac{\sum_i v_i \gamma_i}{\sum_i v_i} \qquad s = \frac{\sum_i v_i s_i}{\sum_i v_i}$$

where γ_i, s_i, v_i are the precedence score, the separation score, and the number of versions of the file i.

5 Conclusions and Future Work

The system presented in this paper is the first implementation of an automated system for the identification of relevant sequences in source code repositories. And its goal is to identify if there are differences in the development patterns between Agile and traditional developers.

The work presented is only an exploratory phase on the analysis on sequence patterns in software development.

References

1. R. Cooley, B. Mobasher, J. Srivastava, “Web Mining: Information and Pattern Discovery on the World Wide Web”, *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97)*, November 1997.
2. P.M. Johnson, “You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering”, *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*, Nashville, TN, USA, December 2001.
3. S.J. Metsker, “Building Parsers with Java”, Adison-Wesley, 2001.
4. J. Myllymaki, J. Jackson, “Web-based data mining, Automatically extract information with HTML, XML, and Java”, *IBM developerWorks*, http://www-106.ibm.com/developerworks/web/library/wa-wbdm/?dwzone=web
5. D.C. Pelz, “Innovation Complexity and Sequence of Innovating Strategies”, *Knowledge: Creation Diffusion, Utilization*, Vol. 6, 1985, pp. 261-291.

Writing Coherent User Stories with Tool Support

Michał Śmiałek[1,2], Jacek Bojarski[1],
Wiktor Nowakowski[1], and Tomasz Straszak[1]

[1] Warsaw University of Technology, Warsaw, Poland
[2] Infovide S.A., Warsaw, Poland
smialek@iem.pw.edu.pl

Abstract. Writing good user stories for software systems seems to be a hard task. Story writers often tend to mix real stories (sequences of events) with descriptions of the domain (notion definitions). This often leads to inconsistencies and confusion in communication between the users and the developers. This paper proposes a tool that could support writing coherent user stories. The tool clearly separates the domain notion definitions from the actual stories. This leads to a consistent requirements model that is more readable by the users and also easier to implement by the developers.

1 Introduction

User stories [1] can be compared to novels in literature. Good novels communicate stories treated as sequences of events, and place these stories in a well described environment. Unfortunately, writing stories that describe requirements for software systems seems to be equally hard as writing good novels. However, unlike for writing novels, lack of coherence and ambiguities may cause disaster when developing a system based on such stories.

Finding inconsistencies in a set of several tenths or hundreds of stories is quite a hard task. Especially, when these stories are written by different people and at different times. It seems that some kind of a tool support for the task of writing stories could significantly help in keeping coherence of requirements. This tool should have two major characteristics: it should enable keeping coherent style (possibly by different writers) and it should prevent from introducing conflicts between different stories.

2 The Concept

Most inconsistencies in requirements are caused by contradictory definitions of terms in different stories. To eliminate the source of such inconsistencies we need to have a single repository of notions (a vocabulary) that can be used in various stories. The stories could then use definitions already found in the repository and just concentrate on the actual sequence of events (Fig. 1).

Another source of confusion lies in the interpretation of story sentences. It can be argued that the story can be unambiguously presented with very simple

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 247–250, 2005.
© Springer-Verlag Berlin Heidelberg 2005

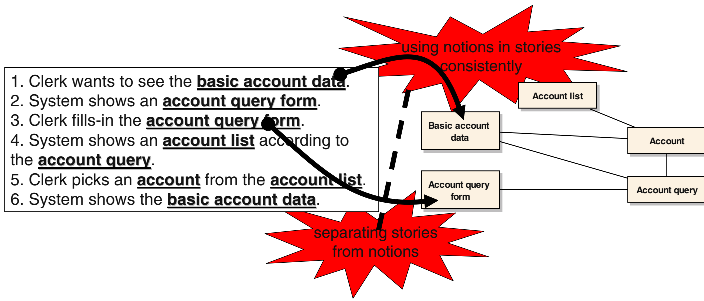


Fig. 1. Story with a separate notion vocabulary

sentences containing just a subject, a verb and one or two objects (SVO[O] - see [2]). This gives us very consistent style, and promotes discovery of new notions. The story writers are obviously somewhat constrained with such notation, however, practice shows that such self-constraint leverages introduction of notions that would not be discovered otherwise (Fig. 2).

3 The Tool

The basic characteristic of a scenario construction tool would be to promote consistent style with proper SVO sentences and link sentence elements with the notions in the vocabulary [3] (see Fig. 3). When writing a story, the story writer would have instant access to notions grouped by subject. All the notions could be easily introduced as sentence objects (or subjects or verbs). When a particular

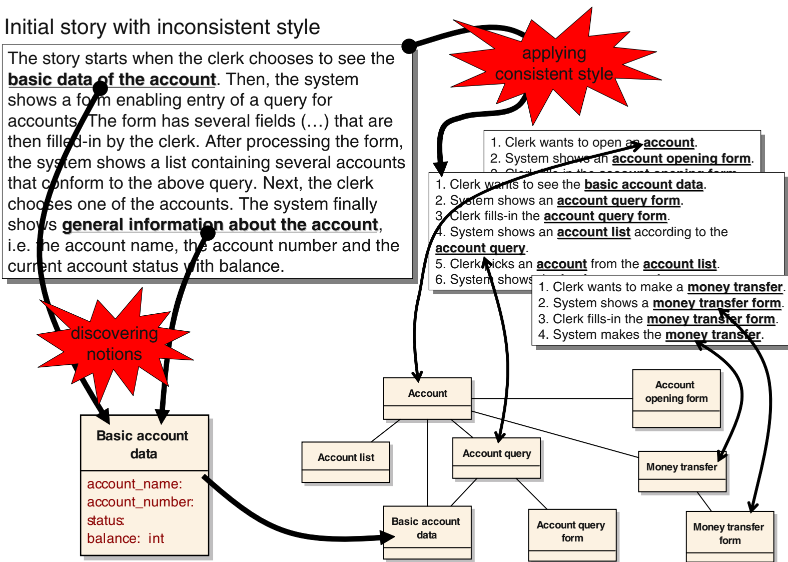


Fig. 2. Story with a separate notion vocabulary

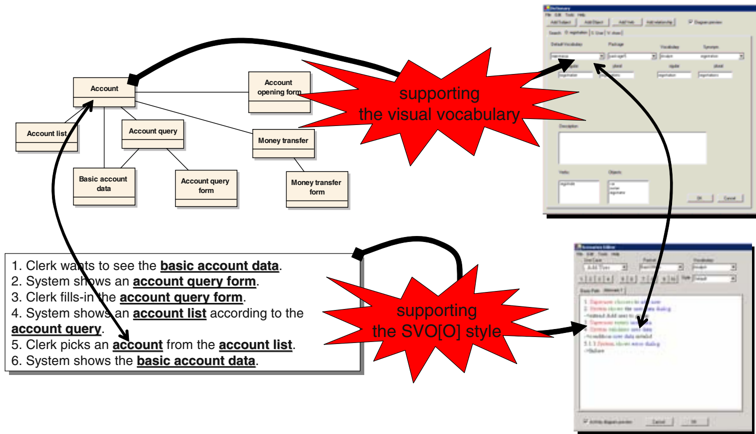


Fig. 3. Supporting consistent stories with notions in a dedicated tool

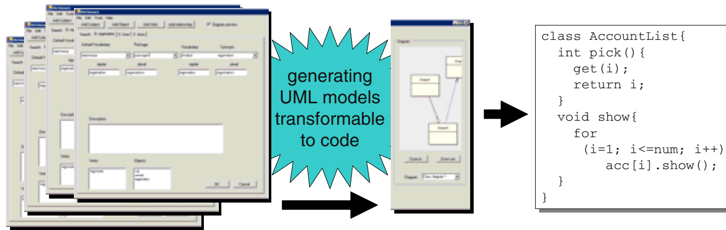


Fig. 4. Generating a UML class model for direct transformation into code

notion used by the writer is not present in the vocabulary - it can be defined immediately. An important feature when defining notions is the possibility to introduce synonyms and forms. This allows for building different vocabularies for different groups of users.

A very important feature of the story construction tool is the possibility to generate visual models (see Fig. 4). This gives the software developers means to synchronize their efforts with the actual requirements. The tool makes this possible by allowing to create visual UML [4] diagrams in parallel with writing notions and story sentences. These diagrams can be directly translated into code or transformed into more platform specific design models.

4 The Tool and the Process

As every tool, the current story-writing tool has to used in the development process with care. Figure 5 illustrates possible usage of the tool in a lightweight, iterative process based on XP [5] and FDD [6]. It has to be noted, that initial stories coming from the user are created without tool support. Also, the initial story-writing session can be best organized with very simple "tools" like index cards. The interactive session should produce a set of index cards with clarified

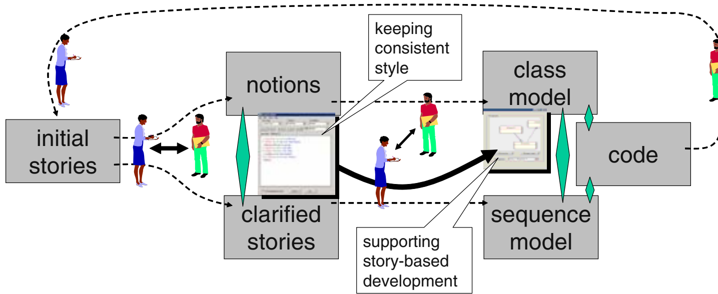


Fig. 5. Applying SVO story tool support in an agile process

stories and notions (see Fig. 5). Only then, when development begins, the stories and notions can be introduced into a repository organized around the presented tool. The tool can verify the results of the interactive session, allowing for their coherence and synchronization with the results of previous iterations. This leads to further clarification and interactions with the users.

During development, the tool facilitates engineering tasks by binding them better with the stories to be implemented. It has to be stresses, that the proposed UML class model generator does not suppress human effort in structuring code, but only supports it to some extent. The models that come from the tool can only initiate the activities associated with construction of the final system.

5 Conclusions

The proposed tool can significantly improve consistency of requirements written using stories. The tool allows the story writers to apply uniform style, and to synchronize notion definitions between different stories. Very simple notation for stories and a separate notion repository gives additional impulse for creativity - leverages finding new notions that can then be used as candidates for classes in code. At the same time, the tool can effectively promote a lightweight approach to developing software that is based directly on the user's needs.

References

1. Cohn, M.: User Stories Applied. Addison-Wesley (2004)
2. Graham, I.: Object-Oriented Methods Principles & Practice. Pearson Education (2001)
3. Śmiałek, M.: Profile suite for model transformations on the computation independent level. Lecture Notes on Computer Science **3297** (2005) 269–272
4. Fowler, M., Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman (2000)
5. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
6. Palmer, S.R., Felsing, J.M.: A Practical Guide to Feature-Driven Development. Prentice Hall PTR (2002)

BPUF: Big Picture Up Front

Frank Keenan[1] and David Bustard[2]

[1] Department of Computing and Mathematics,
Dundalk Institute of Technology, Dundalk, Co. Louth, Ireland
frank.keenan@dkit.ie
[2] School of Information and Software Engineering,
University of Ulster, Coleraine, BT52 1SA, UK
dw.bustard@ulster.ac.uk

Abstract. XP recommends that developers focus on the software product ignoring issues that do not contribute directly to its construction. All wider issues are assumed to be the responsibility of the customer representative. This paper argues that there is benefit in the full development team considering the 'big picture' before beginning implementation as long as this can be undertaken in a suitably agile way. Evidence of the need for this wider approach is presented. Aspects of Soft Systems Methodology (SSM) are then proposed as a means of performing the necessary analysis. Two analysis techniques of SSM, rich pictures and conceptual models, are evaluated against agile criteria defined by Ambler.

1 Introduction

Jackson argues that when tackling new projects it is important to understand the problem domain adequately before focusing on the solution [1] because the requirements are effectively determined by this environment. Essentially, an environment can be defined as *that part of the world that affects or is affected by the computing system*. Ideally, a computing system should be 'aligned' with its environment to provide optimal support and such alignment should be maintained as the situation evolves. The XP process begins at the *requirements specification* phase, with no suggestion of any pre-specification activities [2, 3]. This paper considers this apparent deficiency and how it might be resolved.

2 The XP Approach

Taking time to stand back and look at the *big picture* can be beneficial. Why are we doing this project? What problem is it trying to solve? How does it fit in with the overall context? What progress are we trying to make? How will we judge success? XP on the other hand encourages a focus on short-term product development, emphasized by the YAGNI mantra (*You Aren't Going to Need It*). It could be argued that knowledge of the environment is a responsibility of the *on-site customer*. However, various difficulties have been reported. Jepsen [4], for example, recommends spending a great deal of time and energy in investigating contrasting stakeholder view-

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 251–254, 2005.
© Springer-Verlag Berlin Heidelberg 2005

points in complex situations. This helps in "understanding the business" [5]. Stephens and Rosenberg also highlight the value of spending time 'up front' with as many customer representatives as possible to get "down into the trenches and find out the real problems that need to be solved" [6]. Without this activity, an XP project can become too inwardly focused. Van Deursen [7] similarly identifies benefit for the customer representative in having "systematic ways for diagnosing problem situations", which again means analyzing the environment adequately.

The premise of this paper is that conducting an appropriate analysis of a problem situation, before development begins, can enhance the XP process. However, such analysis must be consistent with the other agile practices of XP in terms of efficiency and linkage.

3 Soft Systems Methodology (SSM)

Soft Systems Methodology (SSM) [8] is a well-established technique for analysing problem situations, including the development of information systems [9, 10]. It is especially helpful in complex 'messy' situations - so called 'soft' problems. This goal-driven approach is used to "establish a vision" of what the business might be which helps identify desirable changes to the current system. SSM can be used by itself and in combination with other approaches [11]. For example, the combination of SSM and use case modeling has been described [12].

In SSM, problem situations are usually captured diagrammatically as *rich pictures*. These are subjective, with no rules defined for drawing them, but they help achieve a shared understanding of a situation among stakeholders. Based on that understanding, models of 'relevant systems' are developed. These are expressed as *root definitions* and *conceptual models*. A root definition is a short textual statement that defines the important elements of the 'relevant system'. A *conceptual model* is a behavioral description identifying each activity required for a system to achieve its purpose as described in the root definition. More specifically, it is a directed graph with nodes denoting activities and arcs indicating logical dependencies.

Conceptual models are compared with the current real-world situation to identify 'gaps' for improvement. This facilitates analysis between the model and the perceived real world in a structured way.

As a step towards integrating SSM with XP, Table 1 compares the SSM models against criteria defined by Ambler [13]. The assessed degree of conformance, '✓' or '✓✓', to each activity is recorded.

The table indicates that both rich pictures and conceptual models are as simple as possible while fulfilling their purpose, sufficiently accurate and provide positive value. However, rich pictures are not necessarily consistent as they are very subjective. Also, they can only be used to represent very high-level detail. If more detail is required their usefulness diminishes.

Agile models should be "understandable by their intended audience". With conceptual models the intended audience may include different stakeholders from diverse backgrounds. At first the terminology used may be less well understood by each party. However, the notations used are easily understandable. Conceptual models can become cluttered if too much detail is represented. To overcome this nodes can be reorganized so that each is exploded at a lower level.

Table 1. Evaluation of SSM Models

Agile Model traits	Rich Picture	Conceptual Model
Fulfill Purpose	✓✓	✓✓
Understandable	✓✓	✓
Sufficiently Accurate	✓✓	✓✓
Sufficiently consistent	✓	✓✓
Sufficiently detailed	✓	✓
Provide positive value	✓✓	✓✓
Simple as possible	✓✓	✓✓

Overall, the models of SSM are relatively consistent with the agile philosophy. However, to link with XP more support will be required. Also, SSM is a traditionally a slow process [10], designed for effectiveness rather than efficiency, so some streamlining is required for use with XP.

4 Conclusion and Further Work

This paper has proposed that XP can benefit, in certain situations, from an analysis of the problem situation before commencing development. Potentially SSM can meet that need. SSM models seem reasonably consistent with expected agile criteria if the overall analysis process can be made sufficiently lean. This is being examined. The research includes tool development and an observation of patterns of XP usage. The role of on-site customers and interaction designers (listed as a team role) will also have to be clarified. Tools, such as personas and goals that help *make sense of the real world* are also being considered. The overall goal is to help derive and clarify user stories on which system requirements are based.

Acknowledgements

The work described in this paper was supported by the Centre for Software Process Technologies (CSPT), funded by Invest NI through the Centres of Excellence Programme. In particular, it is a contribution to collaborative work on the ‘agile approaches’ theme of the Irish Software Engineering Research Consortium (ISERC) (http://www.iserc.ie/).

References

1. Jackson, M., 2004, Seeing More of the World, IEEE Software, November/December 21(6).
2. Abrahamsson P., Warsta J, Siponen M., Ronkainen J., New Directions on Agile Methods: A Comparative Analysis, ICSE 2003
3. Beck K., Andres C., “Extreme Programming Explained”, Second Edition, Addison Wesley, 2005.
4. Jepsen O., “Customer Collaboration – challenges and experiences”, Agile Development Conference, Salt Lake City, 2004.
5. Thorup L, Jepsen O., “Improving customer developer collaboration”, JAOO 2003, 25/09/03, www.bestbrains.dk/xpvip-jaoo2003-report.html [viewed 16/08/04].
6. Stephens M., Rosenberg D., “Extreme Programming Refactored: The Case Against XP”, Apress, 2003

7. van Deursen A., "Customer Involvement in Extreme Programming", http://www.cwi.nl/~arie/wci2001/wci-report.pdf, May 2001, pp 1-2, XP2001.
8. Checkland, P., Systems Thinking, Systems Practice (with 30-year retrospective), John Wiley & Sons, 1999
9. Stowell, F.A. (ed.), Information Systems Provision: The Contributions of SSM, McGraw-Hill, London, 1995.
10. Mingers J., Taylor S., "The Use of Soft Systems Methodology in Practice," Journal of the Operational Research Society, 43(4), 1992, pp. 321-332.
11. Munro I., Mingers J., "The use of multimethodology in practice – results of a survey of practitioners", Journal of the Operational Research Society, Palgrave MacMillan, Mar 18 2002, pp. 369 – 378
12. Bustard, D W, Zhonglin He, Wilkie, F G, "Soft Systems and Use-Case Modelling: Mutually Supportive or Mutually Exclusive?" Proc 32nd Hawaii International Conference System Sciences, 1999.
13. Ambler, S.,: When is a Model Agile?,. http://www.agilemodeling.com/essays/whenIsAModelAgile.htm, [viewed 7th Feb 2005]

Agile Development Environment for Programming and Testing (ADEPT) – Eclipse Makes Project Management eXtreme

Mike Holcombe and Bhavnidhi Kalra

Department of Computer Science, University of Sheffield
M.Holcombe@dcs.shef.ac.uk, acp03bk@shef.ac.uk

Abstract. Genesys Solutions is a bespoke IT company, first of its kind, run by MSc and fourth year students of Department of Computer Science, University of Sheffield under the supervision of Prof. Mike Holcombe and Dr. Marian Gheorghe. Genesys follows the eXtreme Programming (XP) methodology for software development based on client requirements. The commitment towards XP and its 'good software practices' can be considered as the greatest strength of Genesys.

Agile Development Environment for Programming and Testing (ADEPT) is our contribution towards supporting the XP methodology by adopting the Eclipse platform along with its associated tools and frameworks within Genesys Solutions. It aimed to teach good software practices in Genesys to support XP by providing a software development life cycle management tool that will encompass the best practices of XP. It comprises of tools based on the principles of XP such as story cards, system metaphor, estimations, testing and quality assurance. ADEPT was the result of the IBM Eclipse Innovation 2004 awarded to the University of Sheffield. Also, based on the previous year's performance and more innovative ideas to implement more principles of XP we have been awarded another grant under the IBM Eclipse Innovation 2005 programme.

Keywords: project management, software life cycle, extreme programming, eclipse

1 The Foundation – Genesys Solutions

Genesys Solutions is a commercial IT company specialising in bespoke solutions for business. It is the first of its kind that is run by university students as part of their degree. The MSc and fourth year students of the Department of Computer Science, University of Sheffield operates the company under the supervision of Prof. Mike Holcombe and Dr. Marian Gheorghe. It is a self-funded company that also provides high quality IT consultancy and training at highly competitive rates.

Genesys has existed for 8 years and has received very favourable comments from Industry and the UK Government. The main reason for this is due to the fact that this company provides students with invaluable experience in modelling business processes and the engineering of high quality software solutions, which has proved to be its key strength in close collaboration with their business clients.

Genesys has followed the eXtreme Programming (XP) methodology for software development over the last 5 years. This requires the company to work closely with the client to ensure that the client will receive a solution, which fits their requirements. Any teaching required by the students is a mixture of formal lectures and presentations, tutorials and demonstrations, supervision and group project management. Thus,

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 255–258, 2005.
© Springer-Verlag Berlin Heidelberg 2005

the commitment towards XP and its ‘good software practices’ can be considered as one of the greatest strength’s of Genesys.

In Genesys we place much emphasis on using and supporting Open Source initiatives. Therefore, as far as possible, we run Linux both on our servers and workstations and use Java in the majority of our software solutions. The Eclipse IDE is used within Genesys for software development.

2 The Eclipse Platform

Eclipse is an open framework to integrate different types of application development tools. It is a platform on top of which an integrated development environment (IDE) can be created to work on almost any programming language or resource. A software developer can use Eclipse to combine different tools to design, compile, debug and analyze code to get a single integrated development environment. Eclipse achieves this with an open, extensible architecture based on plug-ins.

Eclipse is the initiative of the Eclipse Foundation. It is built by an open community of tool providers that uses an open source model. Eclipse was originally an IBM project. In 2001, IBM released the source code and the Eclipse Foundation was started by a group of companies including IBM, Borland and RedHat. Since then its membership has grown to include HP, Intel, Ericsson, SAP and many other companies. Now, the Eclipse Foundation is an independent, not-for-profit corporation that will steer the evolution of Eclipse. Thus, IBM supports the activities of Eclipse and does not possess it.

The Eclipse project has three main components.

- the Eclipse Platform
- Java Development Tools (JDT)
- Plug-in Development Environment (PDE)

3 IBM Eclipse Innovation Grant 2004

The *IBM Eclipse Innovation Grant 2004* was awarded to the University of Sheffield to encourage the active use of the Eclipse open source software for academic curricula and research. This was achieved by adopting the Eclipse platform along with its associated tools and frameworks within Genesys Solutions.

Agile Development Environment for Programming and Testing (ADEPT) is our contribution towards supporting the XP methodology by adopting the Eclipse platform along with its associated tools and frameworks within Genesys Solutions. It aimed to teach good software practices in Genesys to support XP by providing a software development life cycle management tool that will encompass the best practices of XP. It comprises of tools based on the principles of XP such as story cards, system metaphor, estimations, testing, and quality assurance.

The first step towards incorporating Eclipse within Genesys was to ensure that it is being used as the primary IDE for software development. This was achieved by having shared networked installations of versions 2.3.1 and 3.0.0M8 for both Linux GTK and Windows. Additional facilities for the Eclipse framework was provided in the form of PHP, code colouring and database plug-ins.

As part of the proposal, teaching material was developed to introduce Eclipse along with the role of Java in Eclipse. Also, principles such as source code manage-

ment, test first programming, continuous integration and code refactoring were covered as other tutorials.

The research conducted for Eclipse and its practical implementation can be seen through the Eclipse plug-in developed. The endeavour of the ADEPT plug in, a suite of computer aided software engineering plug ins, is to ensure that the software development life cycle (SDLC) incorporates the best practices provided by XP. The following diagram represents the complete project management tool:

Fig. 1. Block Diagram for ADEPT: Shows all the plug-ins that constitutes that project management tool along with their dependencies representing the complete SDLC cycle

The practices and its corresponding plug-ins are as listed below:

- **Story Card Editor**
  This plug-in has been designed keeping *the planning game* principle of XP in mind. It supports gathering the client's requirements and documenting them for the use by the other tools.
- **Software Metrics Management**
  This plug-in has been designed considering the importance of *software estimations* in project management. It uses story cards to estimate the time, effort and cost, using the function and feature.
- **System Metaphor Management**
  The development of this plug-in is attributed to the system metaphor principle of XP. It will translate the story cards into X-Machine diagrams.
- **Test Management**
  The *test first programming* principle of XP has lead to the development of this plug-in. It will help in planning and managing system test cases. The tests will be generated based on story cards.
- **Quality Assurance**
  Though, quality assurance is not as highlighted as it should be, it is a vital requirement of the XP project management life cycle. A *quality check* for any software is an imperative prerequisite before its release. Thus, this tool supports tracking and monitoring of the quality aspects of the whole system. It assures consistency between story cards, X-Machines and the generated test cases. It also generates checklists to support the inspection process.

A section on the Genesys website (http://www.genesys.shef.ac.uk/eclipse) comprises of all the information related to Eclipse within Genesys. This includes the documentation produced till date and the download of ADEPT plug-in with a provision to report bugs in any of the plug-ins.

4 IBM Eclipse Innovation Grant 2005

On the basis of our performance and our innovative ideas to implement more principles of XP we have been awarded another grant under the *IBM Eclipse Innovation 2005* programme**.** This year we intend to make further enhancements to the current version of ADEPT. Teaching material on the process of plug-in development will be created based on the experiences gained and problems faced during the same.

As a part of the Eclipse Innovation Grant 2005, we are progressing in the direction of implementing XP more practically by:

- improvising on the current version of ADEPT
- developing tools on other principles of XP such as coding standards
- to help new students in the understanding of the agility required in software development we aim to develop a planner tool
- with new innovative ideas like mind maps we intend on understanding in depth the current principle of system metaphor

The following diagram represents the second version of ADEPT with all the new additions.

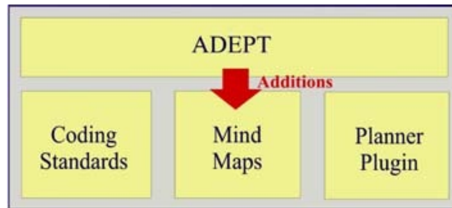


Fig. 2. Block Diagram for ADEPT ++: Continuation of ADEPT supplemented with additional tools to improve software development that follow other XP practices

5 Conclusion and Future Work

ADEPT along with new additions will result in a far more comprehensive and robust project management tool for use in Genesys and other software houses following XP. It is aiming to be a stepping stone for software development companies and developers who are using XP, by automating project management following XP practices as much as possible.

ADEPT has laid foundation for a project management tool that can be enhanced through an iterative process year after year within Genesys Solutions. ADEPT is our contribution to the Eclipse Community and marks its success by being the first ever project management tool in Eclipse using XP.

References

1. Beck, K. (1999), Extreme Programming Explained
2. Holcombe, M. (2003) Extreme Programming for Real: a disciplined, agile approach to software engineering.
3. Extreme Programming: http://www.extremeprogramming.org/
4. Eclipse: http://www.eclipse.org
5. Genesys Solutions: http://www.genesys.shef.ac.uk/eclipse

Tailoring Agile Methodologies to the Southern African Environment

Ernest Mnkandla[1], Barry Dwolatzky[2], and Sifiso Mlotshwa[3]

[1] Monash University, South Africa, School of Information Technology, Private Bag X60, Roodepoort, Johannesburg, 1725, South Africa
Ernest.Mnkandla@infotech.monash.edu.au
[2] University of the Witwatersrand, School of Electrical and Information Engineering, P.O. Box 542, Wits, 2050, Johannesburg, South Africa
b.dwolatzky@ee.wits.ac.za
[3] eSolutions, 55 Rossal Rd, Greendale, Harare, Zimbabwe
smlotshwa@yahoo.com

Abstract. The present movement in the adoption of agile methodologies as a contemporary approach to the management of the software development processes has seen a growing trend towards the selection of relevant practices from the agile family as opposed to the adoption of specific methods. This paper reports work-in-progress of a proposed novel modeling technique for tailoring methodologies to a particular environment using the family of methodologies approach. The tool is being applied by one software development organisation in Southern Africa and the partial results are included in this paper.

1 Introduction

Methodology selection and classification techniques that exist in the available literature are mainly academic meaning that when analysing a methodology the choice of elements that fully represent the methodology though subjective [2] gives the user a philosophical understanding of the methodologies see; [1, 2] for such frameworks. For a software developer trying to traverse the methodology jungle, the value of the methodology is in those practices that are relevant to the given project's needs.

This paper presents work-in-progress. The entire research work involves case studies in Southern Africa, one in Zimbabwe and another in South Africa. However, what is reported here are the partial results of the Zimbabwean case. The final results are expected to provide evidence that the proposed Agile Methodologies Generic (AMG) model which should be generally usable in any software development environment has been used successfully in the Southern Africa environment.

While tailoring of methodologies to a specific situation according to cultural, organisational, and individual factors is not a key observation because similar work has been done before, [3, 4], there are two key things about this contribution: first the specific point of view from the Southern African culture which will appear in the future release of the work, and secondly the attempt of the model to derive a development approach from the agile group rather than specific methods.

2 Family of Methodologies Model

The family of methodologies concept proposed in this paper considers agile methodologies as a group of methodologies with common parameters that can be used to

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 259–262, 2005.
© Springer-Verlag Berlin Heidelberg 2005

model the entire group. Based on this model, methodology parameters can be identified that are common among the different agile methodologies making it possible to create a set of relevant agile practices that can be used in an organization. The original concepts of the model are based on two foundations: 1) the philosophy of Jim Highsmith's Adaptive Software Development (ASD) methodology. ASD focuses on the speculate, collaborate and learn cycle iteratively which is fundamental to agile development, and 2) the concept of organizational maturity levels which says that mature organizations families of repeatable and automated processes. It is from such a perspective that AMG was born. AMG considers agile methodologies at an abstract level where the four values of the Agile Manifesto are assumed to collectively constitute basic philosophy of all agile methodologies. The phases of AMG (mechanistic, organic, and synergistic adaptation) are therefore analyzed in light of the values of the project at hand.

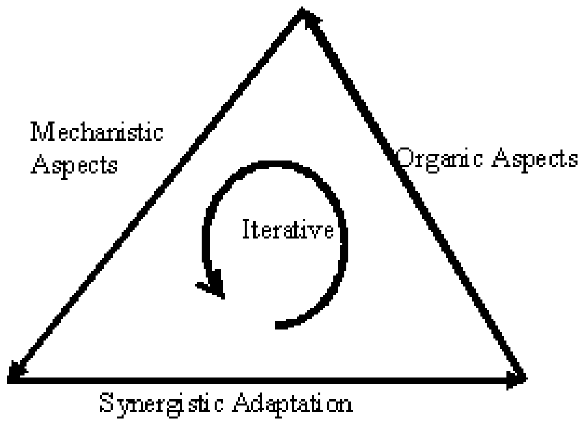


Fig. 1. Agile Methodologies Generic (AMG) Model

Organic Aspects: How the methodology caters for social issues and the way these social issues affect the software architecture and how these human issues drive project outcomes. The emphasis is on the importance of individual members of the software development team and the way they interact. At this phase the fundamental question should be; how the methodology caters for team building, organization culture etc.

Mechanistic Aspects: How the methodology values the technical aspects of software development such as the form of the deliverables (is it a document or code etc?). The emphasis is on the importance of working software as opposed to document deliverables. At this phase the technical details of how the methodology spells out the development environment should be considered.

Synergistic Adaptation: How the methodology values the software development activity as a collaborative process between the developers and the customer. The Adaptive part means that the methodology regards instability of requirements as a welcome change that must be embraced to improve the relevance of the product to the customer. The emphasis is on the importance of customer collaboration instead of contract negotiation, and embracing requirements change instead of following a plan.

The *iterative* centre means that the model is applicable to the chunks of iterations that the problem domain is broken down into in the agile philosophy.

3 Case Study

eSoultions Southern Africa is an IBM Partner organisation that started in 2001. They specialise in software development of web-based solutions to general business and computing problems. Among other products the organisation has developed applications for the health insurance sector, revenue authority, wireless banking applications, etc. The company has group of six developers at senior and junior levels. In 2004 they decided to review their development methodologies, seeking one that was most suitable for their customers' ever changing business environments.

3.1 Techniques

The process leading to the adoption of the AMG model involved discussion sessions on agile technologies and eventually studying of books on specific agile processes by the developer. To study the problems that the developer company was encountering interviews and questionnaires were used. Some of the responses to; e.g., questions on reasons for wanting to change the method of development were:

- Need to exploit the opportunities of ever changing business requirements.
- The massive application of Object Technology (J2EE) led to the need to apply a methodology that would cultivate the spirit of solving problems the Object Oriented way.
- Object Orientation technology benefits such as, lower costs of factoring changes demands the implementation of a methodology that is based on iterative development.
- A dire need to use a methodology that would allow learning of new and emerging technologies, and business processes during the development process.

3.2 Tentative Results

Through the application of the AMG model the organization managed to select practices that have a stronger alignment with Extreme Programming (XP) and Scrum. The results will be analyzed by looking for patterns, contrasts and commonalities in the methodology selection and tailoring behavior of the three selected organizations.

3.3 Problems Encountered

The team members: In some instances it was not possible to hold scrum meeting at a scheduled times. This was solved through variation of time for meetings. The variation was however fixed to within an hour say; between 14:00 and 15:00.

The customers: Demand for deliverables that were not inline with the new methodology. Another pressure point was in the alignment of the actual project costs to the budget. To deal with this challenge a workshop was conducted with the client executives on the project management methodology (tailored version of Scrum and XP).

4 Conclusions

The paper summarized a novel thinking tool for the tailoring of agile methodologies to a given environment. The major contribution of the proposed technology is to align the software developer's thinking process with a more generic perspective of agile methodologies. This tool can benefit not only developers but, researchers seeking to model agile methodologies and executives seeking to understand the agile family of methodologies. This is work-in-progress but so far the results show that challenges of tailoring agile methodologies to given environments have different flavors in this part of the world such as need to work around the clock, costing projects in very unstable currencies, complete lack of knowledge on agile methodologies by most customers, very old technologies and systems still in use etc.

References

1. Abrahamson, P., Salo, O., Ronkainen, J. and Warsta, J. (2002), Agile Software Development Methods: Review and Analysis, *VVT Publications,* No. 478, pp. 7-94.
2. Avison, D.E. and Fitzgerald, G. (1995), Information Systems Development: Methodologies Techniques and Tools. McGraw-Hill.
3. Song, X. and Osterweil, L.J. (1992), Comparing Design Methodologies Through Process Modeling, *IEEE Software*, pp. 29-44.
4. Osterweil, L.J. and Song, X. (1996), Toward objective, systematic design-method comparisons, *IEEE Proceedings of IWSSD-8*, pp. 170-171.

XP/Agile Education and Training

Angela Martin, Steven Fraser, Rachel Davies, Mike Holcombe,
Rick Mugridge, Duncan Pierce, Tom Poppendieck, and Giancarlo Succi

Abstract. XP/Agile education and training remains a challenge from the perspective of determining relevant content; identifying effective methods for delivery; and maintaining the focus and motivation of students. This panel brings together academic and industry professionals to share their perspectives and experiences. Anticipated points for discussion include: education/training delivery strategies, curriculum definition, certification challenges, marketing issues, collaboration strategies to engage industry sponsorship, value assessments for students and sponsoring organizations, and program success stories. This will be a highly interactive panel and the audience should come prepared to both ask and answer questions.

Steven Fraser (sdfraser@acm.org) – Panel Co-moderator
Steven Fraser recently joined Qualcomm's Learning Center as a senior member of staff in San Diego California with responsibilities for technical learning and development. From 2002 to 2004 Steven was a consultant on tech transfer and disruptive technologies. Previous to 2002 Steven held a variety of diverse software technology program management roles at Nortel Networks and BNR/Bell Northern Research – including: Process Architect, Senior Manager (Disruptive Technology and Global External Research), and Process Engineering Advisor. In 1994 he spent a year as a Visiting Scientist at the Software Engineering Institute (SEI) collaborating with the Application of Software Models project on the development of team-based domain analysis techniques. Steven has a PhD in EE from McGill University (Montreal) and is an avid Operatunist/videographer

Angela Martin (angela@mcs.vuw.ac.nz) – Panel Co-moderator
Angela has over ten years of wide ranging information systems experience and has a firm grounding in all aspects of systems integration and development. She is a PhD Candidate at Victoria University of Wellington, New Zealand, supervised by James Noble and Robert Biddle. Her PhD research utilizes in-depth case studies of the XP Customer Role, on a wide range of projects across Australasia, the US and Europe.

Rachel Davies (rachel@agilexp.com)
The best way to learn how to do something is to try it for yourself, ideally with support from an experienced practitioner. Theory gives us a mental map that helps us understand how process steps connect together. Experience helps us understand how the techniques work from the inside. Coaching can accelerate the learning process but if you can maintain the energy, then practice and refinement will get you there in the end. Agile software development is a collaborative activity and this must be reflected in any training. My preferred training approach is to give a little theory and practice

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 263–266, 2005.
© Springer-Verlag Berlin Heidelberg 2005

on a sample problem then follow up by using the technique on a real project with coaching support. It is vital in this training approach to have frequent reflection points where the trainees and the trainer share their observations with the group.

Rachel Davies, Agile Experience Ltd – www.agilexp.com. Rachel is an XP practitioner and makes her living training and coaching agile teams in industry. She is also a director of the Agile Alliance.

Mike Holcombe (M.Holcombe@dcs.shef.ac.uk)

We have two basic courses. The first of these is a second year course, Software Hut, where teams of 4 or 6 students compete to build a solution for an external business client. Typically each client will work with 4-6 teams and they select the best solution for use in their company at the end of the 12 week exercise. A prize is given to the best teams. For several years we have had half the teams using XP and the rest a traditional design-led method. This year all students are using XP. It is their first exposure to XP and having a real client makes it all make sense! I believe that having a real client – who doesn't know what they want precisely – is the only way to learn real software engineering. Lectures on Software engineering are so boring, doing it is fun. You cannot imitate a real client credibly. The need for outstanding quality is also a major driver to learn. The second course is “The Genesys Company” – a real software company entirely run by masters students. XP is the company approach. We have won IBM Eclipse Innovation Awards in 2004 and 2005 to develop ADEPT our Eclipse environment for XP. My work with Genesys (http://www.genesys.shef.ac.uk/) is unique and has been highly praised by the UK Education Minister and others. Visit: http://www.dcs.shef.ac.uk/vt/projects/softwareobservatory.html for more details.

Mike Holcombe is Professor of Computer Science at university of Sheffield since 1988. Research interests in software engineering include: software testing, agile methodologies, empirical software engineering and computational biology. Mike is a Fellow of British Computer Society, Inst. Math. & Its Applications and has published 7 books and 130 research papers. Mike has taught XP for the past 5 years and his recent work is based around project work as a learning mechanism.

Rick Mugridge (r.mugridge@auckland.ac.nz)

There are interesting challenges in introducing agile software development into a university curriculum. Some of these challenges arise when introducing agility into any organisation, such as building interest and an agile culture, taking the larger organisational contexts into account, and making progress step by step. As with any organisational change, there are forces which act to eject new ideas. In the university, these forces include the following: waterfall-based attitudes, several courses going in parallel, and individual assessment. Introducing agility can easily lead to outstretched resources. I have had to apply my time wisely to introducing agility into the right areas of the curriculum. Finally, it takes time to introduce and evolve new ideas. It's not possible to develop good skills in agile development in a single course. I started with a project class of 45 at year 2 in 2001. Year by year I have tried different approaches. In 2004 I ran six-hour labs each week with 85 students. I was customer and coach, writing Fit tests. This was successful but tiring.

Rick Mugridge has been teaching agile software development in various university courses in Software Engineering since 2001. As well as running XP projects with large classes, he has taught TDD and agile UI at years 2 and 3. Rick has also coached teams in XP, and consulted and run industry tutorials in Storytest driven development and automated testing. He is an author of “Fit for Developing Software” and is on the Program Committee for XP2005 and Agile 2005.

Duncan Pierce (duncan@duncanpierce.org)
Ok, I've only been given two paragraphs to cover a lot of ground, so there's no room for shades of meaning here – it's all black and white! Most universities are teaching two major streams that I'd call "computer science" and "software engineering". The computer science is good stuff, and I think we need a broader understanding of it, especially in industry. The software engineering, on the other hand, is mostly based on views and techniques that are outdated, misunderstood and misapplied. I'll go further: software engineering causes projects to fail. We should stop encouraging people to think this way. On the other hand, I think there's some back-filling to be done. I don't doubt that agile techniques work very well in practical terms, but we need to build a better theoretical basis for understanding and explaining agility. I think the conceptual framework that ties together teaching of every aspect of software development should be built around agile values like business focus, goal setting and so on. It's not sufficient to settle for teaching a few practitioner skills like test-driven development, vital though these are. I also think we should strive for convergence in teaching and training techniques. I can't see why these should really be that different. Learning well requires theoretical understanding, demonstrated examples, practise at applying it and mentoring in realistic situations. And what about training? Well, the basic problem with most training courses is that they're boring. The students yawn their way through endless presentations, then fumble their way through mindless exercises. Training courses should be based on robots destroying each other, simulating ant colonies, or whatever, but *not* payroll systems!

Duncan Pierce (www.duncanpierce.org) has been helping companies including Egg and British Telecom improve their software development using agile techniques for the last 4 years. Duncan regularly presents at conferences, including XPDays in the UK, Benelux and Germany, XP2004, Agile Business Conference, ACCU and SPA. He is a long-standing member of the Extreme Tuesday Club (XTC), and was a founding organizer of the first XPDay conference.

Tom Poppendieck (tom@poppendieck.com)
All software is embedded, some in hardware, some in a business process. The value of software derives wholly from the operation of whatever it is embedded in. Any attempt to separate software design from the design of what generates value will naturally lead to the sort of sub-optimizing pathologies the software community continues to struggle with. Product or process development teams need to include everyone involved in conceptualizing, developing, marketing, deploying and supporting the product or business process so that the overall value chain delivers the best result the **whole** team is capable of. Traditional software engineering, divorced from a product or process design context, inevitably optimizes one small segment of the value chain at the expense of dramatically less valuable outcomes. Contemporary

software engineering must catch up on lean thinking principles which have over the last 20 years doubled the productivity of manufacturing, logistics, retail, and other fields. Agile methods are a proper application of lean principles but still fall sadly short of addressing the entire value chain. The bottom line is software engineering, in isolation, is likely to destroy value. Software engineering practiced in a cross-disciplinary manner in collaborations with Operations Management, Psychology, Business, and other appropriate disciplines (depending on what the software is embedded in) can become a productivity enhancing instead of a productivity destroying discipline.

Tom Poppendieck is co-author with his wife Mary of the 2003 Jolt Productivity Award winning *Lean Software Development – an Agile Toolkit.* He has worked as a physics professor, in product development at large and small organizations and worked as an enterprise architect and business analyst. His recent focus is on tools and practices to support the customer role on agile development teams.

Giancarlo Succi (Giancarlo.Succi@unibz.it)

In our university we teach both the Agile and the Traditional approach. We use a draft of a text that we are producing. The agile approach is the "reference" approach in our classes, as it appears more suited for a learning environment (fast feedback & constant improvement), but we also present the more plan-based methodologies. The students are evaluated according to three criteria: midterm (28%), final oral (12%), and project (60%). However, each part must be passed for the student to pass the course. The project is evaluated weekly with questions to each team member to ensure that there are individual contributions to the project being developed. The project employs an external customer, which has a say on the project grade. Using an agile approach has simplified the way projects are handled. Now, we need to ensure that also more traditional aspects are being taken care of – which perhaps is our weak part now.

Giancarlo Succi is Professor with Tenure at the Free University of Bolzano-Bozen, Italy, where he directs the Center for Applied Software Engineering. Before joining the Free University of Bolzano-Bozen, he has been Professor with Tenure at the University of Alberta, Edmonton, Alberta, Associate Professor at the University of Calgary, Alberta, and Assistant Professor at the University of Trento, Italy. He was also chairman of a small software company, EuTec. Prof. Succi holds a Laurea degree in Electrical Engineering (Genova, 1988), an MSc in Computer Science (SUNY Buffalo, 1991) and a PhD in Computer and Electrical Engineering (Genova, 1993). Prof. Succi is a consultant for several private and public organizations worldwide in the area of software system architecting, design, and development; strategy for software organizations; training of software personnel. Giancarlo Succi is a Fulbright Scholar.

Off-Shore Agile Software Development

Steven Fraser, Angela Martin, Mack Adams, Carl Chilley,
David Hussman, Mary Poppendieck, and Mark Striebeck

Abstract. Off-shore development is increasing in popularity. Off-shoring affects many things in our environment: what and where we build and deploy; how we budget and deliver services; and how and when we communicate. Can the high touch, high bandwidth model that Agile purports be applied to a situation where one of the fundamental tenants – a co-located team – is shattered? This panel will offer a forum to share and learn from industry practitioners and researchers on how to make off-shore software development work in an agile context.

Steven Fraser (sdfraser@acm.org) – Panel Moderator
A version of this panel was first organized for the ACM's OOPSLA'04 conference in Vancouver and was very well received. It will be run at XP2005 with a somewhat different cast in an Agile (rather than object-oriented) conference context. Motivation for outsourcing and off-shoring is driven by three factors: cost avoidance, time-to-market, and customer proximity. Karmarkar (Harvard Business Review, June 2004) has observed that cost-avoidance in the knowledge/IT industry seems to work best when there is a significant bimodal wealth distribution among a global community sharing similar linguistic and cultural contexts. According to a recent A. T. Kearney survey, the top 10 countries for US-based companies with offshore development include: India, China, Mexico, Brazil, Canada, Czech Republic, Philippines, Australia, Hungary, and Ireland. The report ranked countries on the basis of their financial structure, business environment, staff talent and availability. While increased bandwidth has facilitated the sharing of information, it has not necessarily improved the sharing of context and culture.

Steven Fraser recently joined Qualcomm's Learning Center as a senior member of staff in San Diego California with responsibilities for technical learning and development. From 2002 to 2004 Steven was a consultant on tech transfer and disruptive technologies. Previous to 2002 Steven held a variety of diverse software technology program management roles at Nortel Networks and BNR/Bell Northern Research - including: Process Architect, Senior Manager (Disruptive Technology and Global External Research), and Process Engineering Advisor. In 1994 he spent a year as a Visiting Scientist at the Software Engineering Institute (SEI) collaborating with the Application of Software Models project on the development of team-based domain analysis techniques.Steven has a PhD in EE from McGill University (Montreal) and is an avid Operatunist/videographer.

Angela Martin (angela@mcs.vuw.ac.nz)
Can offshore agile software development succeed? Absolutely! Can it also fail? Absolutely! I have encountered both outcomes in my research and in my personal ex-

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 267–272, 2005.
© Springer-Verlag Berlin Heidelberg 2005

perience – along with a few more that are not quite so clear cut. I think the key question is: which factors make offshore agile software development more likely to succeed – despite the odds being against it?

The successful offshore projects that I have observed find a way to build connections between people, both in work and personal settings. These connections are harder to build when you are offshore. Co-location lets the whole team learn to understand one another, and to see each others' points of view. We may not be "walking in the other persons shoes" but we are close enough to "see them walking in their own shoes". We can begin to observe the difficulties associated with that "walk", whether it is as a programmer, business sponsor, project manager, tester or a customer. Offshore development requires travel, as well as many phone calls, informal emails and whatever other communication opportunities you have at your disposal on this project. At least one person on this team actively needs to seek out opportunities to build these connections: they do not happen by accident.

One concern I have is that is the customer or customer proxy role typically takes on the responsibility of building these connections, especially the traveling. We already know that the customer in a co-located team is overloaded: how much more strain can the customer really take? What other hidden costs to off-shore agile development have we yet to discover?

Angela has over ten years of wide ranging information systems experience and has a firm grounding in all aspects of systems integration and development. She is a PhD Candidate at Victoria University of Wellington, New Zealand, supervised by James Noble and Robert Biddle. Her PhD research utilizes in-depth case studies of the XP Customer Role, on a wide range of projects across Australasia, the United States and Europe.

Mack Adams (mca@thoughtworks.com)

Under most circumstances, writing business systems software is a difficult and complex challenge. Splitting the team into multiple locations, with 5000 miles and 12 hours separating them only serves to exacerbate this. Concurrently, as the offshore argument becomes more compelling for many companies, can the high touch, high bandwidth model that Agile/XP purports be applied to a situation where one of the fundamental tenants – a co-located team – is shattered?

By rigorously applying techniques and practices, common on all Agile/XP projects, in a modified manner, and wrapping the project in way that promotes maximum human interactions, the uncertainty of offshore development can be addressed. Increased feedback, visibility and communication lie at the heart of the struggle. On the people side, a battery of channels, both informal and formal, need to increase the communications bandwidth within the team. This piece is underscored with the early and regular cross-pollination of people through each of the locations. Additionally, a robust technical infrastructure facilitating a transparent development process that provides unequivocal status on progress and code quality is critical.

At the end of the day, the economic equation for offshore development will only get stronger as companies seek increasing labour arbitrage, access to talent and operational optimization. For complex offshore systems development the Distributed Agile delivery model can de-risk projects significantly and result in software that’s both economical and on target for clients.

Mack Adams is a Project & Program Manager with ThoughtWorks in London, where he helps clients deliver software using Agile/XP techniques. Having worked on projects in Canada, the United States, India, and the United Kingdom, Mack has been involved in the development and execution of the Distributed Agile delivery model in the offshore context. Prior to his 3 years at ThoughtWorks, he was involved in two start-up software companies, with the later being acquired.

Carl Chilley (Carl.Chilley@xansa.com)

That the software industry is following the trends of many production industries in “off-loading” (out sourcing and/or off-shoring) the design, delivery and maintenance of business solutions is not a surprise. What maybe is a surprise is the astonishment of many software professionals that it should have happened to their profession at all – is software development really a development process?

It is certainly true that out-sourcing of systems – be they soft or hard – has been ongoing for many years: why should an organisation have to focus time and effort on non core processes and systems when there are economies of scale in letting more specialist organisations manage them? But the intellectual process of software development a real production process? Surely not!

In some respects the IT industry, along with its close buddy the telecommunications industry, can be said to have laid the foundations for off-loading to be possible and effective. More cost-effective international bandwidth and integrated systems aid in the communication process. Also, better processes and methods all add to the foundation for making software development and delivery a global activity.

Add into this mix highly educated, experienced and motivated people, coupled with linguistic similarity and cultural familiarity to provide the raw materials for the intellectual effort, globalisation and “glocalisation” (the ability to make a global offering, be it a service or product, appear high localised) to minimise any cultural anomalies and potential misunderstandings and the mélange starts to become a business possibility. Blend into this the cost equation – day rates for software engineers in offshore environments can be 75% to 80% less than that of their on-shore equivalents – and you have the drive for the offshore phenomenon.

But are we creating a possible future problem here? Are there social ramifications that are seldom considered when talking about the role of off-shoring in the local IT industry?

One of the clearest lessons learnt over the last couple of decades of software development is the need to get close to the customer, understand intimately not just what they think they want but help them understand what they really want. Indeed, this is one of the central tenets of extreme programming. To be effective in this requires individuals who are not only close to the customer but also capable of appreciating the nuances of articulation from often disparate members of the client organisation and be able to clearly feed-back such ideas etc. to the benefit and client and producer alike.

Even with “glocalisation” and the cultural homogenisation being experienced throughout the world, there is often no substitute for “local knowledge” when it comes to working with clients. Such knowledge is not just at the business level but also at the production and cultural levels as well.

And here we have the possible paradox. If we continue to off-shore the IT production capability – for good economic reasons as the customer perspective is that they will get the same quality of solution as they did before but at a much cheaper price – where will the next generation of customer-local IT solution designers, architects and consultants gain their production experience and knowledge?

Carl Chilley is an Executive Consultant with Xansa in the UK and primarily focuses on business architecture across all vertical and horizontal market sectors. Carl has been in the industry for over 25 years, having been the European software project manager for the iconic Xerox Star development, working his way through various architectural forms and styles in pursuit of the delivery of usable and viable business solutions. Most recently Carl has been applying the precepts and concepts of service-driven architecture in the development of solutions for his clients. Noting that Xansa has a market-leading Indian capability and a solid reputation for both outsourcing and off-shoring process and IT systems, Carl has also been looking at how to best integrate the dispersed capabilities of organisations to effect effective delivery in a cost effective manner.

David Hussman (david.hussman@sgfco.com)
There are many cultures associated with any project. Healthy agile communities know they must respect the culture of the company, the project community, and the individuals. This is no small challenge, and needs specific focus from one or more members of the communities. Adding a community which is physically disconnected (off shore) or culturally different (off shore – out sourced) only increases the challenge. Each development community has deep beliefs (values) as well as ways of working and talking (principles and practices). Simply adding a new community to the mix does not work (and never has). Successful addition of new players (off shore or outsourced) should start with some players meeting and working together if possible. Going forward, the coaches (and other cultural guardians) must mind any rifts that start to grow because people are not together. Technologies can help, but the impersonal nature of telephones and teleconferencing cannot replace the information exchanged in a subtle expression that says “I don’t know if I agree.”

David has worked with distributed agile communities, some of which included the added challenges of off-shore and / or out-sourced players. Mostly, this made David’s coaching experience more difficult. Similar to any agile community, we do not think this is solved with a recipe. Instead, someone must work the land daily, trying to create a fertile ground to grow trust and respect that helps bond the community as a whole. David Hussman is co-owner of SGF Software, a U.S. based company that promotes agile practices and provides agile training / coaching. David has worked on large, mission-critical systems as well as small boutique applications across various industries for more than a decade. Motivated to help software teams succeed and smile, David has taught and shared agile ways for 5+ years. Recently, he has been working with large companies starting to use agile practices on a variety of projects. David has participated and presented at various agile conferences, contributing to several books about agile, and the Cutter Consortium Agile Project Advisory Service.

Mary Poppendieck (mary@poppendieck.com)

There are two kinds of work in a typical information systems department: the routine work that is necessary to keep the infrastructure running (servers, e-mail, security, etc.) and the work of software development. An information systems organization should separate these two types of work into two separate organizations. The infrastructure organization can be managed with good operations practices aimed at reducing the cost, and off-shoring may be an appropriate cost reduction strategy.

The software development organization, on the other hand, should not be focused on reducing cost, it should pay attention to bringing increased economic value to the businesses. For software development organizations aimed at increasing business value, off-shoring is usually penny wise and pound foolish. The most creative combination of software and business capability comes from a close collaboration between the people who understand the business and element of an agile those who understand how technology can best support business objectives. Separating these two groups of people by distance, time, paperwork and hand-offs will greatly reduce the potential for innovation in software-supported business processes.

Mary Poppendieck a Cutter Consortium Consultant, is a seasoned leader in both operations and new product development with more than 25 years' of IT experience. She has led teams implementing lean solutions ranging from enterprise supply chain management to digital media, and built one of 3M's first Just-in-Time lean production systems. Mary is currently the President of Poppendieck LLC and located in Minnesota. Her book Lean Software Development: An Agile Toolkit, which brings lean production techniques to software development won the Soft-ware Development Productivity Award in 2004.

Mark Striebeck (mstriebeck@vasoftware.com)

It has often been said that XP works only for co-located development teams. In fact, people sometimes use co-location of a team as simple criteria to use XP or not. This of course leads to the next assumption that it is even "less possible" to use off-shoring with XP – "if I can't make it work with my own distributed team, how could I make it work with a development partner on the other side of the world?"

During the 2 ½ years that VA Software has been using XP, we have had several different approaches to off-shoring our development efforts. And – as always – XP has produced "extreme" results. In some cases it worked very well, in other cases it did not work at all. Looking back, we can see that the following factors were very important for a successful XP off-shoring initiative:

- The local and offshore teams have to be compatible. The user story based approach requires that both sides communicate well and can cooperate without heavy documentation and processes
- Both sides need to trust and respect the other. We had the case were the local engineers did not trust the abilities of the offshore engineers or where offshore engineers thought of local engineers to as too critical and biased
- The traditional index card project management of XP needs to be complemented by one or more tools

The good news is that XP shows very quickly if the off-shoring development approach works or not. Instead of waiting for several weeks/months for the first release,

an XP team sees progress and issues after a few short iterations. In our case, we completely stopped an off-shoring initiative after 2 iterations when it became visible that there were too many issues to continue. Recognizing the problem early, we were able to adjust our project schedule accordingly instead of waking up a few days before the release was due and realizing that the application was not ready.

Mark Striebeck is the Director of SourceForge Engineering for VA Software. VA Software is the provider of SourceForge Enterprise Edition, the award–winning enterprise software that helps companies to improve the efficiency and effectiveness of globally distributed application development teams. Under his direction, VA Software adopted XP for all software development activities. He is working closely with VA Software's customers to adopt XP/agile methodologies using SourceForge Enterprise Edition. Prior to VA Software, Striebeck worked for Cambridge Technology Partners as project manager on several fixed-time, fixed-price projects. He worked in a variety of technical positions since 1989. Striebeck is a certified Project Manager and Scrum Master, he holds two Masters degrees in Computer Science and Mathematics.

The Music of Agile Software Development

Karl Scotland

Two Way Media Ltd
kscotland@twowaytv.co.uk

Value Proposition

A significant number of members of the Agile community seem to have a musical background, leading to a hypothesis that there are commonalities between the two disciplines, both of which draw on diversely skilled individuals, collaborating to create a common vision. While this notion is purely anecdotal and subjective, it is interesting enough to explore further, and this activity session is intended to delve into those commonalities in more detail.

The aim is to allow participants to:

- have fun,
- making music,
- exploring new ideas, and
- providing new insights into Agile software development

While musical skills, background, or interest would be beneficial, they are by no means required.

Format Description

The following items are planned:

- Clapping Music – rehearsal and performance of an existing composition, using nothing but hand claps.
- Improvisation – creation of new compositions to a theme, using simple toy instruments.
- Fishbowl discussion – sharing new or existing metaphors between music and agile software development.
- Smaller group discussions – exchanging experiences and collating results.

The intended outcome is a set of new ways of thinking and talking about agile software development, to be published on a wiki or other web site.

Biographies

Karl realised that computers were his forte while studying for his music degree, and went on to make his career in this field. He has worked on domains ranging from multimedia to neural network to interactive TV, and has experienced both a complete lack of process, and an overly rigorous one. When he discovered XP, and was given the opportunity to use it, he embraced it enthusiastically, and has never looked back. Karl is currently Production Manager with a team of 15 developers, responsible of ensuring smooth delivery of all projects. Until recently, he was a Team Leader with BBC Interactive, with a team which developed software which delivered 78 services in 12 months, a feat which could not have been achieved without agility.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 273, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The XP Game

Pascal Van Cauwenberghe[1], Olivier Lafontan[2], Ivan Moore[3], and Vera Peeters[4]

[1] Nayima bvba, St Pancratiuslaan 49, 1933 Sterrebeek, Belgium
pvc@nayima.be
[2] Egg plc, Pride Park, Riverside Road, Derby DE99 3GG, UK
Olivier.Lafontan@egg.com
[3] ThoughtWorks, Peek House, 20 Eastcheap, London EC3M 1EB, UK
ivan@thoughtworks.com
[4] Tryx bvba, Colomastraat 28, B-2800 Mechelen, Belgium
vera.peeters@tryx.com

Abstract. The XP Game is a playful way to familiarize the players with some of the more difficult concepts of the XP Planning Game, like velocity, story estimation, yesterday's weather and the cycle of life. Anyone can participate. The goal is to make development and business people work together, they both play both roles. It's especially useful when a company starts adopting XP.

1 Audience and Benefits of Attending

Anyone can participate. Developers and customers benefit from experiencing both sides of the planning game. The XP Game explains velocity, in particular, showing how velocity is not the same as business value. This tutorial demonstrates how you can quickly learn to make predictable plans.

2 Content Outline

In real life Planning Game, development and business people are sitting on opposite sides of the table. Both participate, but in different roles. The XP Game makes the players switch between developer and customer roles, so that they understand each other's behaviour very well.

Some of the concepts in the Planning Game are difficult to grasp, for developers and for customers. What exactly is the meaning and background of stories, iterations, consequent estimations, velocity, yesterday's weather, planning game, feedback? The XP Game is a simulation of the XP Planning Game, which includes the following phases: story estimation, story prioritization, planning, implementation and feedback. No knowledge of coding is required.

This XP Game is a practical way to demonstrate how the rules of the XP Planning Game make up an environment in which it becomes possible to make predictable plans. After all, the easiest way to get a feeling for the way it works is to experience it. This tutorial will differ slightly from the XP Game available from http://www.xp.be/xpgame/download/ with the inclusion of innovations from its use at Egg.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 274–275, 2005.
© Springer-Verlag Berlin Heidelberg 2005

3 Presenter Bios

Pascal Van Cauwenberghe is an independent consultant with company NAYIMA. He's worked in IT for more than 10 years, applying agile software development methods (mostly XP) for the last 5. He co-founded the Belgian XP users group (http://www.xp.be) and is one of the organizers of the XP Day Benelux (http://www.xpday.net) and Agile Open (http://www.agileopen.net) conferences. He has spoken and organized session at several conferences like XP2001, XP Universe, Object Technology and XP Day. He's the co-author of the XP Game (http://www.xp.be/xpgame).

Olivier Lafontan has spent the last six years working in programme/project management, specializing on Customer Relationship Management business aspects. He has alternated roles from both "Technology" and "Business" sides of the fence in companies such as BT, Lexmark, Unipart and Freesbee. Olivier currently works for Egg plc, and has been using the XP Game as a tool to aid the Agile transformation Egg has undertaken.

Ivan Moore has been programming for over 20 years and yet he still regularly makes mistakes. That's why he's interested in test driven development, refactoring, iterative and incremental development, and drinking tea. He has a PhD in automated refactoring (1996), and has presented papers, tutorials and workshops at numerous international conferences, such as OOPSLA, XP, XPDay, ACCU, TOOLS and ECOOP. He works for ThoughtWorks as a coach, developer and tea boy, helping teams to "get agile". He hopes that his Cruise Control monitoring tray icon will allow people to forgive him for developing Jester, a mutation testing tool for Java/JUnit.

Vera Peeters is an independent consultant. She runs her own company TRYX. She has 15 years experience in developing software systems, especially object-oriented development in all kinds of high-technological environments. She has been practicing agile ways of working for several years now. Vera has presented workshops at several conferences (XP200X, XPUniverse, OT200X, different XPDays). She is a co-organizer of the XPDay Benelux (http://www.xpday.be) and of the Javapolis conferences (http://www.javapolis.org). In 2001, she founded the Belgian XP User Group (http://www.xp.be) together with Pascal Van Cauwenberghe.

4 History of Tutorial

This XP Game has been presented at XP2001 in Sardinia, XPUniverse 2001 in North Carolina, OT2002 in St. Neots, XPDay03 in London and XP2004 in Germany.

5 Examples of Supporting Material

The XP Game materials are available from http://www.xp.be/xpgame/download/

Leadership in Extreme Programming

Panellists:
Kent Beck, Fred Tingey, John Nolan, and others

Convener:
Steve Freeman

Abstract. A panel of expert practitioners will offer advice to members of the audience on how to address the issues they are facing when applying XP. The format is to consider concrete cases, to talk about what we would do in those cases, and discuss the principles behind the actions. The goal is to help the audience to be more effective in response to their "leadership moments" as they apply XP.

Nobody said that XP was going to be easy, just effective. Anyone who is involved in applying XP will encounter a range of obstacles that tests their abilities to deliver code *and* create an effective organisation, and some of the ideas in XP contradict conventional wisdom about software development. The growing number of books offer guidance but an outside, human perspective can also be helpful.

The panel will discuss cases proposed by members of the audience. It will address the real issues that practitioners face in their day-to-day workw and offer advice and experience. The members of the panel are either managers who have applied XP, or consultants who have converted many different organisations to XP.

If you have a topic you would like the panel to discuss, then please submit a brief outline describing your situation and the difficulties you would like the panel to discuss. We will pick a representative selection before the session and work with the authors to clarify the issues for presentation. During the session, we will present each case and the panellists will discuss ideas, possible solutions, and principles with the author.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 276, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Agile Project Management

Ken Schwaber

Scrum Alliance
ken.schwaber@controlchaos.com

1 Description

We can read about Agile processes in books and articles. However, the management of projects using an Agile process represents a significant shift for both the project team(s) and the organization as a whole. The shift internal to the team occurs as the project manager teaches the customer how to drive the project iteration by iteration to maximize ROI and minimize risk, with no intermediaries between the customer and team. The other internal shift happens as the team realizes that self-management means exactly that – the team has to figure out how to mane its own work cross-functionally. These are trivial words, but the realization of their impact on career paths, relationships, and performance reviews is profound. Even more difficult is helping the team and organization overcome the bad habits they had acquired prior to implementing the Agile process – waterfall thinking, command-and-control management, and abusive relationships. Ken Schwaber, the instructor, has addressed these problems in numerous organizations and will share his insights with the attendees, along with a framework for thinking about the new role of a project manager. Since it is easy to think one knows what Agile processes are like without knowing what they really feel like, two case studies are used to help the class experience the differences.

2 Instructor

Ken Schwaber has addressed these problems in numerous organizations and will share his insights with the attendees, along with a framework for thinking about the new role of a project manager. Ken has been a software professional for thirty years, and was a signatory to the Agile Manifesto, founder and chairman of the board of the Agile Alliance, and founder of the Scrum Alliance. He is one of the co-developers of the Scrum process.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 277, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Expressing Business Rules

Rick Mugridge

University of Auckland, New Zealand, and Rimu Research Ltd
r.mugridge@auckland.ac.nz

Abstract. Learn how to express business rules as storytests [1], with a focus on expressing the business domain with clarity and brevity.
Writing storytests (Customer tests) is usually complicated by several factors: The business domain needs to be understood, and often needs to be clarified. Storytests need to evolve to help this understanding evolve. Emphasis is often placed too early on the testing aspects, rather than on expressing the business domain as clearly as possible. The storytests often make premature commitments to details of the application being developed, or are not written until those details are known.
This tutorial will give participants experience in expressing business rules well as storytests. We'll see that such storytests evolve as the whole team's understanding of the business needs and the system evolve.

1 Audience and Benefits of Attending

Audience: Customers, business analysts, project managers, testers, programmers.

Benefit: Learn how to express storytests with clarity and see the benefits of this. Avoid the pitfalls that many find when writing such storytests. Learn how to focus on understanding and communicating the domain, formulating a *ubiquitous language* [2] in the process. Learn how to use concrete examples to specify *calculation* and *workflow* business rules in that language.

This is suitable for those with little experience in storytests, right through to those who want to refine and advance their techniques. It is as relevant to programmers as it is to others in a project team.

2 Content Order and Process

After introducing the main ideas, participants will work in small groups on a series of exercises. These exercises involve examples and focus on specific practices.

Process:

- Introduction
- Main Ideas: expressing business rules (*calculation* and *workflow*) through examples in *Fit* [3, 4, 5, 6]. Evolving a *ubiquitous language*. Smells and refactorings.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 278–279, 2005.
© Springer-Verlag Berlin Heidelberg 2005

– First exercises, with guidance. Small groups will tackle focussed exercises that clarify approaches and smells in expressing business rules as storytests.
– Retrospective on experience with exercises. Abstracting and generalising from those exercises.
– Second exercises. Groups will tackle bigger, more realistic exercises with issues that are tangled together. There will be less upfront guidance.
– Retrospective on range of experiences on the exercises.
– Open questions.
– Conclusions.

3 Presenter Resume

Rick Mugridge is the author, with Ward Cunningham, of *Fit for Developing Software*, Prentice-Hall, June 2005. He developed the *FitLibrary* as he explored ways of expressing business rules well under change. He is an experienced software developer and a business analyst, and has been teaching, coaching and researching into agile software development for some years. He ran tutorials on *Fit* at XP2004, ADC2004 and XPAU2004.

References

1. *Storytest* and *Storytest Driven Development* were coined by Joshua Kerievsky, http://www.industrialxp.org.
2. Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison Wesley, 2004.
3. Rick Mugridge and Ward Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, Prentice-Hall, 2005.
4. *Fit*, http://fit.c2.com.
5. *FitNesse*, http://www.fitnesse.org.
6. *FitLibrary*, https://sourceforge.net/projects/fitlibrary.

Introduction to Lean Software Development

Practical Approaches for Applying Lean Principles to Software Development

Mary Poppendieck[1] and Tom Poppendieck[1]

Poppendieck.LLC, 7666 Carnelian Lane, Eden Prairie, MN, USA

Abstract. Long feedback loops are the biggest cause of waste in software development. They are the reason why well over 50% of all newly developed software is seldom or never used. Long feedback loops are the cause of seriously delayed projects, unmanageable software defect counts, and code bases that calcify because of their complexity.
Lean Software Development is all about shortening information feedback loops in the software development process and creating flow. The result is increased speed and quality along with lower cost. If this sounds unlikely, consider that in manufacturing, operations, and logistics, lean processes routinely deliver the highest speed, highest quality and lowest cost in extremely competitive environments. This tutorial will show you how to apply the principles that underlay lean manufacturing, lean logistics and lean product development to software development.

Format: The tutorial will be interactive. You will create a current value stream map of a real software development environment, learn how to apply lean tools to the environment, and then design a future value stream map.

Intended Audience: This program is designed for senior software development practitioners, team leads and managers who are considering lean software development for their organizations.

Learn how to:

1. Develop a value stream map for your software development organization and what to do once you have the map.
2. Assess the state of the basic disciplines which determine your software development process capability and organize a visual workplace so that everyone knows the most important thing to do next without being told.
3. Apply lessons from queuing theory to manage the software development pipeline.
4. Discover what's wrong with change approval and defect management systems, and how to reframe the development workflow to address both areas more effectively.
5. Use a financial model to avoid sum-optimization.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 280, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The Courage to Communicate: Collaborative Team Skills for XP/Agile Teams

Diana Larsen

FutureWorks Consulting, LLC
Portland, Oregon, USA
dlarsen@futureworksconsulting.com

Abstract. *Communication, Feedback, Courage, Simplicity. Individuals and interactions over process and tools.* Applying the values of XP/Agile approaches to software development projects highlights the shift to the critical importance of functioning well in highly collaborative team environments. The excitement of trying something new and the intense learning curve of understanding and applying the practices tends to overshadow team member interactions through the first stages of project team development. However, once XP/Agile practices become the usual way of doing business, team members frequently discover the limits of their ability to communicate and work collaboratively. Effective, collaborative communication becomes the next challenge. Three skills in particular help a team make the move from adequate work performance to high performance. Effective XP team members learn the critical collaborative skills of group decision-making, active listening and interpersonal feedback – seeking it, giving it, and receiving it well.

Introduction

Effective teamwork is an essential ingredient for success in Extreme Programming (XP) and other Agile software development environments. In these workplaces, teamwork strengthens feedback loops and fosters collaboration across functions and boundaries. Collaborative communication skills are a critical contribution to the software development process. Top-quality teamwork requires a commitment to common goals, effective working relationships, and full participation in planning and decision-making; all of which are improved by people who can communicate collaboratively interpersonally and as part of a group. Development and IT professional who are drawn to Extreme Programming work best in an environment where ideas and information are shared, where the spirit of "team-ness" is strong, and where team members work together to accomplish common goals – an environment of collaborative communication.

When XP, or another Agile methodology, is introduced on a project, team members face the challenge of learning and applying the values, principles and practices. The values and principles highlight the importance of collaboration and the practices provide some structure for increasing effective communication, but eventually the team discovers the limits of its members' skills as communicators and needs to institute communication practices as well. Three collaborative communication skills that are important for teams to use well are: active listening; seeking, giving and receiving interpersonal feedback; and group decision-making.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 281–284, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Collaborative Communication Skills

Active Listening

Listening is our most under-rated, under-appreciated communication skill. As we have moved from oral communities to literate societies, the skills of speaking, reading and writing have taken precedence over listening. When we are awake, we communicate about 80% of the time. As we communicate, research shows that we spend far more of our time listening than in the other four communication modes, 50%-500% more. However few of us take the time to learn to listen well.

On XP and Agile projects, the emphasis on understanding customer needs, collaborating with customers and others, pairing, planning as a team, and other communication intensive activities, makes effective listening an even higher priority.

We understand others best by listening and attending to their thoughts, feelings, and needs as expressed in their speech, behavior, and body language. We find ways to discover how they have interpreted our meaning when we speak by listening to how they respond to us and asking for clarification. Truly listening involves developing skills and staying curious about others.

Three important parts of good listening are a focus on the speaker, demonstrating our curiosity through Active Listening, and interpreting non-verbal behavior.

Focus on the Speaker
Focusing on the speaker involves making choices about the way we listen. A listener commits to doing only one thing – listening – at a time and eliminates other distractions. A listener lets the speaker complete his thoughts without interruption and allows him time and space in the conversation to think by being comfortable with the silence between their expressions. A listener checks in to ask whether the speaker has finished her thoughts before jumping into share his own.

Active Listening
In Active Listening, the listener summarizes, in his own words, the content and feeling of the speaker's message, states it to the speaker to confirm understanding, and may ask questions to clarify understanding. The listener puts aside her own frame of reference and looks at the world from the speaker's perspective. Active listening acknowledges another person by telling her what you understand her to be feeling and thinking. When listening actively, the listener uses skills to paraphrase, clarify, and summarize what was said to learn whether she understood them correctly. Through Active Listening skills, team members communicate that everyone will be heard when speaking up and, thereby, encourages creative interaction within the team.

Interpreting Non-verbal Behavior
For an effective listener, non-verbal behaviors may supply the key to unspoken emotions or thoughts or they may add emphasis to the spoken words. Non-verbal behavior includes facial expression, tone/volume of voice, gesture, body language, sub-verbal expressions (laughter, uh-huh, unh-huh, so-so-so) and others.

As we listen, it is tempting to think we know what certain non-verbal behaviors mean, but as with any assumptions we make about other people, we are wrong as often as we are right. Many factors can influence the use of non-verbal behavior: culture, environment, personality, physiology, and more.

Effective listeners pay attention to the assumptions they make about the non-verbal behaviors of others; they make them conscious and check them out. "There are frown lines between your eyes, does something about this worry you?" Active Listeners ask clarifying questions to check their assumptions particularly when the non-verbal behaviors seem incongruous to the spoken words. For example, people laugh when they are happy or find something funny or as a nervous response to anxiety or fear. Interpreting nervous laughter as an indicator that a team member finds a serious subject pleasing or humorous, without checking the assumption, could lead to major miscommunication.

Caring, Respectful Feedback

Of all the difficult conversations teams may engage in, seeking, giving, and receiving feedback are among the most difficult. Interpersonal Feedback is providing information to another about the impact, affect or outcome of their behavior. To paraphrase Seashore and Wienberg, feedback offers information about the past, in the present, to affect future behavior.

An XP open workspace is an intimate place. Working closely with others on a team, especially colleagues who are committed to the highest performance, means revealing essential details about who we are, what we know and what we believe. Team members put their ideas, values and passions on the line with their colleagues – and hold a collaborative stance as they do the same. In order to foster such intimacy and high performance, teams need regular, open, honest, direct communication, including interpersonal feedback. When information and feedback flow freely, team members pool their strengths and make progress toward mutual goals. Effective team members reveal details about the way they interpret and integrate the behavior of others, describe the impact of that behavior, and seek information about how their own behavior might affect the rest of the team.

The role of interpersonal feedback is to strengthen relationships. Interpersonal feedback is effective among team members when several factors are in place. Feedback is given with caring and respect for the person and the project. Each person on the team has the skill to ensure the feedback he gives or gets helps to improve his performance. Continual feedback has become an acceptable part of how work is conducted; the team may have working agreements explicitly encouraging feedback. The feedback given includes information about desirable as well as undesirable behavior.

Team Decision-Making

Many team members have learned to make effective independent, persona decisions. However, making decisions as a part of a team that the whole group can support adds geometric complexity with the increasing size of the team. Often teams will default to one decision-making approach. They will defer to the Coach or Scrum Master. They may procrastinate on important, time critical decisions. They may decide that every solution must have the unanimous agreement and enthusiastic support of every member – a time-consuming approach.

Teams with good collaborative communication skills know that different decisions require different approaches and learn to flexibly choose the appropriate approach for the importance and criticality of the issue. Teams select among approaches that include unanimity, consensus, autocratic, consultative, and majority rule, as each best

suits the situation. Effective teams have pre-determined working agreements about generic classes of decisions and how solutions will be derived.

Conclusion

Collaborative communication emerges when information is shared in ways that enhance working relationships and support team members' ability to do a good job. It occurs when conversations and questioning are open, honest, and action-oriented. Listening, interpersonal feedback and team decision-making serve to build trust and strengthen relationships, to focus on the work to be done, and to convey a sense of caring and commitment. When project community members feel in the know, have the information needed to do their job, and feel listened to and heard, they are better able to collaborate and co-create in the team. Collaborative communication gives members of XP teams a sense of connection and belonging. It enables project members to work together in ways that are mutually satisfying and inspiring. It is the backbone of a successful project.

Author

Identified by clients as a standard-setting consultant and by colleagues as an exceptional facilitator, Diana Larsen works in partnership with project and team leaders to strengthen their ability to create and maintain culture, manage change and improve project performance. Diana speaks at conferences and authors articles on topics affecting project teams, team leadership and organizational change. She is a certified ScrumMaster, a partner in FutureWorks Consulting LLC and collaborates closely with a network of consultant colleagues. Contact Diana at
dlarsen@futureworksconsulting.com or by phone +1 503-288-3550.

Test-Driven User Interfaces

Charlie Poole

Poole Consulting L.L.C.
charlie@pooleconsulting.com

Abstract. Learn techniques for Test-Driven Development of user interfaces.

1 Description

Certain kinds of code have a reputation of being quite difficult to test. One of the most frequently cited examples is testing of user interfaces - particularly GUIs.

This tutorial will begin with a quick review of the principles of unit-testing and test-driven development but will then drill down to the issues surrounding user interface testing. We'll look at design issues, identification of what needs to be tested, how test-driven development can be applied to the UI and specific testing techniques for GUIs.

The tutorial will include both presentation and hands on exercises. Participants are encouraged to bring laptops for use in the exercises. We will arrange pairing for those without a laptop. Class examples will be presented in C# and Java, but the exercises may be done in other languages if desired.

Duration

Three hours (half day)

1.1 Audience

This tutorial is aimed at programmers and project leaders who want to apply Test-Driven Development to the user interface. This includes both those with significant TDD experience, who need to develop testable user interfaces, and those who are just adopting TDD.

1.2 Outline

Topics covered will include

– A brief review of unit testing practices and test-first development, and of the facilities of standard unit-testing frameworks, to get us all on the same page.
– A series of motivating examples showing why UI testing seems to be difficult.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 285–286, 2005.
© Springer-Verlag Berlin Heidelberg 2005

- Demonstration of TDD for a simple user interface
- General enabling techniques for difficult test situations
  - Design patterns
  - Use of Mock Objects
  - Specialized testing frameworks
- Specifics for testing user interface code
  - Designing for testability: two principles for good UI separation
  - Defining what needs to be tested at the unit level
  - Testing events and event handling
  - Acceptance testing of user interfaces
- User interface problem-solving exercises

2 Presenter Resume

Charlie Poole has spent more than 30 years as a software developer, designer, project manager, trainer and coach. He works through his own company, Poole Consulting, in the US and recently joined Dublin-based Exoftware as a mentor, in order to work with European clients. He is one of the authors of the NUnit testing framework for .NET.

Contact Information

- charlie@pooleconsulting.com
- cpoole@exoftware.com
- www.pooleconsulting.com
- www.charliepoole.org

The XP Geography: Mapping Your Next Step, a Guide to Planning Your Journey

Kent Beck

Three Rivers Institute
kent@threeriversinstitute.org

Summary. We will explore the primary practices of XP in detail using mind mapping exercises. You will examine your needs and find practices to address them. We will discuss the change process, how to reach agreement on goals and principles, how to implement new practices and how to sustain them. You will make a plan to share with your team and set up incentives for accountability. This tutorial will be interactive and involve many colored felt pens.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 287, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Lightning Writing Workshop Exchange Ideas on Improving Writing Skills

Laurent Bossavit[1] and Emmanuel Gaillot[2]

[1] Exoftware
laurent@bossavit.com
http://bossavit.com/
[2] Octo Technology
egaillot@octo.com

Abstract. All software-related jobs also require writing about software as an ongoing duty, in one form or another – from writing articles evangelizing particular methods or technologies, to writing end user or technical documentation, to writing comments in code. Writing well increases your effectiveness in spreading crucial ideas, and focuses your own thinking as well. Writing is a complex technology in its own right, but it can be mastered through the diligent use of simple practices. This workshop focuses on one such practice, and invites discussion of other practices that develop writing skills.

1 Audience and Benefits

This workshop is intended for anyone who writes for professional or personal reasons, and who'd like to improve their writing skills – at any level, from novices to seasoned writers. Participants and presenters will

- learn and/or practice two specific techniques for improving writing skills
- exchange tips and techniques with other participants
- have fun

2 Content and Process

Beginning programmers often ask, "Which language should I learn ?" Some programming languages indeed provide more leverage than others – but the greatest possible leverage comes from improving mastery in a language you already know – written English (or the written form of your native language if you're not an English speaker).

All software-related jobs also require writing about software as an ongoing duty, in one form or another – from writing articles evangelizing particular methods or technologies, to writing end user or technical documentation, to writing comments in code. Writing well increases your effectiveness in spreading crucial ideas, and focuses your own thinking as well. Writing is a complex technology in own right – and there are specific practices which can help tame that technology. The workshop will

- introduce participants to techniques to improve writing skills, including freewriting and the use of audience feedback
- elicit further techniques from the participants.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 288–289, 2005.
© Springer-Verlag Berlin Heidelberg 2005

A longer-term objective is to assess the feasability of online equivalents of group techniques for building writing skills. **After the workshop**, a wiki will be provided for session outputs (such as samples of freewriting) and ongoing post-conference work.

The workshop itself will run, in outline, as follows:

09:00-09:30	**Entry; introductions**
09:30-09:45	**Freewriting exercise – "private" writing**
09:45-10:00	**Debriefing**
10:00-10:15	**Freewriting exercise – with constraint**
10:15-10:30	**Debriefing**
10:30-11:00	**Break**
11:00-11:30	**Uses of freewriting**
11:30-11:45	**Freewriting exercise – reprise**
11:45-12:30	**Getting and giving feedback exercise**

3 Presenters

Laurent Bossavit is a developer with over 20 years of coding experience, 15 of which on a professional basis. Laurent's focus as an external consultant is on working with teams and keeping them supplied with the raw materials of change and effectiveness – clarity of purpose and a constant infusion of fresh ideas. Laurent stewards (but does not by any stretch manage) several communities in both real and virtual space.

Emmanuel Gaillot is a software engineer and an experienced designer for theatre and dance. He has adapted XP practices and principles to the theatrical production process, and he currently works on instilling theatre practices back into the field of software making. Emmanuel's areas of expertise and interests include self-organizing teams, sofware making and Extreme Programming. He is involved in the conduct of a Coder's Dojo Experiment in Paris, France, where he also works for Octo Technology as both an IT consultant and XP coach.

4 History

Ran once at EuroFoo:
http://wiki.oreillynet.com/eurofoo/wikis.conf?WritingTheUltimateTechnology
Accepted at SPA2005: http://www.spa2005.org/sessions/session39.html

The Coder's Dojo – A Different Way to Teach and Learn Programming

Laurent Bossavit[1] and Emmanuel Gaillot[2]

[1] Exoftware
laurent@bossavit.com
http://bossavit.com/
[2] Octo Technology
egaillot@octo.com

Abstract. If I want to learn Judo, I will enroll at the nearest dojo, and show up for one hour every week for the next two years, at the end of which I may opt for a more assiduous course of study to progress in the art. Years of further training might be rewarded with a black belt, which is merely the sign of ascent to a different stage of learning. No master ever stops learning. If I want to learn object programming... my employer will pack me off to a three-day Java course picked from this year's issue of a big training firm's catalog. Nuts to that – acquiring coding skills is not an "instant gratification" process. This workshop proposes to discover a way of teaching and learning programming in a more appropriate manner, respecting the depth and subtlety of the craft.

1 Audience and Benefits

This workshop is intended for programmers and developers – people who value their programming skills and have a strong motivation to improve; programmers at any skill level from novice to master will benefit from attending. Participants and presenters will:

- discover, and help refine, a specific format for teaching and building coding skills
- explore distinctions between three aspects of coding – method, insight, and form
- contribute to the development of a "curriculum of good programming form"
- have fun

Participants should bring their laptop, and be prepared to code.

2 Content and Process

The Coder's Dojo is a weekly programming class. Programmers of varying skill levels meet as equals. They come together – in physical, not virtual space – around an ongoing series of coding challenges, usually small in scope, often patterned after "pragmatic" Dave Thomas' idea of "coding Kata".

The intent of this **half-day** workshop is to reproduce the mechanics of one Dojo session, so that participants may experience the benefits (and possible deficiencies) of this particular form of teaching and learning coding skills. Presenters intend to use participants' feedback to improve the Dojo, and perhaps inspire others to start their own.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 290–291, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Pre-workshop, a small number of “kata” challenges will be made available to participants. Participants are invited to work at their own pace on one of these challenges.

The workshop itself will be organized around a “typical” session of the Coder's Dojo, according to the format and rituals in force at that time. The Dojo is deliberately an evolving format, but in the main the workshop will run as follows:

09:00-09:15	**Bootstrapping: introduction to the Coder's Dojo**
09:15-09:30	**Fuseki: nominating the Dojo-Cho and setting the agenda**
09:30-10:15	**One Pair Presents a Solution**
10:30-11:00	**Break**
11:00-11:15	**Students Continue The Solution In Pairs**
11:15-12:00	**Original Pair Finishes The Solution - or - New Solution**
12:00-12:30	**Discussion: how to improve the Dojo?**

After the workshop, the participants will be invited to post blog entries to the Dojo's communal blog, as honorary members.

3 Presenters

Laurent Bossavit is a developer with over 20 years of coding experience, 15 of which on a professional basis. Laurent's focus as an external consultant is on working with teams and keeping them supplied with the raw materials of change and effectiveness – clarity of purpose and a constant infusion of fresh ideas. Laurent stewards (but does not by any stretch manage) several communities in both real and virtual space.

Emmanuel Gaillot is a software engineer and an experienced designer for theatre and dance. He has adapted XP practices and principles to the theatrical production process, and he currently works on instilling theatre practices back into the field of software making. Emmanuel's areas of expertise and interests include self-organizing teams, sofware making and Extreme Programming. He is involved in the conduct of the Coder's Dojo Experiment in Paris, France, where he also works for Octo Technology as both an IT consultant and XP coach.

4 History

The Coder's Dojo in Paris has been running on a weekly basis since January 2005. The workshop itself has no previous conference history.

Informative Workspace

"Ways to Make a Workspace that Gives Your Team Useful FeedBack"

Rachel Davies[1] and Tim Bacon[2]

[1] Agile Experience Limited, UK
Rachel@agilexp.com
[2] Prime Eight, UK
timBacon@primeeight.co.uk

Abstract. Informative Workspace is one of the new XP practices launched in the second edition of XP Explained. The practice is to build feedback mechanisms around an agile team that support them in their daily work. These feedback mechanisms can take the form of visual displays (Information Radiators) that are manually updated by the team or electronic eXtreme Feedback Devices (XFD) such as lava lamps or audio signals linked to automated processes. It is vital to ensure that feedback mechanisms are easy to interpret, low maintenance and adapted to local practices. This workshop aims to answer how to implement this practice and explore ways to make workspaces more informative.

1 Audience and Benefits

Participants should have some experience of working in agile software development either as members of a project team (programmers, testers, customers) or managers encouraging informative workspaces within their organizations.

The benefits of attending are that participants will learn new ways of creating an Informative Workspace that enhances their development activity.

1.1 Participation

The presenters have created a web page dedicated to workshop preparation at http://www.agilexp.com/XP2005_InformativeWorkspace.php and circulated a call for position papers to agile discussion lists and user groups. This session can accept an upper limit of 25 participants.

1.2 Deliverables

During the workshop, participants aim to produce examples of Informative Workspace innovations, such as ideas for automated feedback devices, chart formats or workspace layouts. These will be illustrated on flipcharts during the session.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 292–293, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The presenters will photograph all the outputs and arrange for these to be uploaded to the workshop web page or conference wiki website.

2 Content and Process

The workshop will start with introductions followed by a short presentation to introduce the topic. Next participants will share their experience and read their position papers.

The workshop will then move on to an exercise. Participants will be divided into teams and given a manual task to perform as a group. Teams then experiment with simple tracking to see if this improves performance.

Following a break for coffee, we will debrief the exercise and discuss emerging themes.

The session participants will form affinity work groups around the themes that they are interested in. Each work group will select an idea to elaborate and explore implementation constraints. Each work group will take a turn to present their findings to the session group.

2.1 Timetable

00:00 - 00:10	Introductions
00:10 - 00:30	Presentation
00:30 - 01:00	Position papers
01:00 - 01:30	Activity
	Tea break
01:30 - 02:00	Debrief and discussion
02:00 - 03:30	Affinity groups each explore an idea
02:30 - 03:00	Each group presents what they learned to the session group.

3 Presenters

Rachel Davies is an independent consultant, who provides training and coaching in agile software development. In her work she has facilitated many heartbeat retrospectives and requirements workshops in domains as varied as: bioinformatics, digital TV, drug safety, financial services, harbor board finance operations and government support of community waste recycling initiatives. Rachel is also a frequent presenter at industry conferences and a serving director of the Agile Alliance.

Tim Bacon has been involved in professional software development for over 12 years in locations ranging from Switzerland to Slough. He is a self-confessed 'people person' and a passionate advocate of agile processes, software craftsmanship, and Extreme Programming. Tim has been a speaker at several agile events and is currently working as an independent coach and consultant.

Exploring Best Practice for XP Acceptance Testing

Geoff Bache[1], Rick Mugridge[2], and Brian Swan[3]

[1] Carmen Systems AB, Göteborg, Sweden
geoff.bache@carmensystems.com
[2] University of Auckland, New Zealand
r.mugridge@auckland.ac.nz
[3] Exoftware, Dublin, Republic of Ireland
bswan@exoftware.com

Abstract. A few years ago, Acceptance Testing was one of the more poorly understood concepts of XP, with both tools and advice thin on the ground. This has meant that different people have gone different ways with it and an overview of knowledge gathered in the process has been lacking.

The presenters are three such people, each of whom has developed a different tool for acceptance testing: TextTest+xUseCase, Fit+FitLibrary and Exactor, respectively. We are aware that there are lots of other tools around, both within the XP community and outside it. The aim of this workshop is to gather together all this disparate knowledge and start to work towards a common understanding of 'best practice'.

1 Intended Participation

We hope that between 15-20 people will participate, though the format is fairly flexible in terms of numbers attending. Each participant should submit a brief position paper, which should aim to identify what experiences they have of Acceptance Testing, both tools and practices. It should especially highlight any technique they thought particularly helpful, or any problem that they found difficult to get around. These will be used as potential discussion topics in the workshop.

2 Audience and Benefits of Attending

Intended audience is anyone with some experience of acceptance testing who wants to improve what they do and learn from others. This probably includes developers, testers and customers. The benefits are gaining a broad view of a complex field without becoming bogged down in too many tool details. As testing framework designers, we hope to learn more about each other's tools and be able to discuss and push the boundaries of what is possible today in acceptance testing.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 294–295, 2005.
© Springer-Verlag Berlin Heidelberg 2005

3 Outline of the Workshop

The workshop will carry on for a half a day, most of which is reserved for small group discussion.

(1) Presentation. The presenters will aim to give a whirlwind history and overview of what they know of XP acceptance testing: what techniques have been developed and what tools there are to implement them.

(2) Discussion. We will take the topics provided by the attendees' position papers, and the presenters will add some of their own as they think appropriate. We will then choose the most interesting ones by some suitably democratic process.

We will then break into small groups of 5 or 6 people. Each will be headed by a co-ordinator (preferably the attendee who raised the issue in the first place) who will stay for the length of the discussion. Participants can move freely between the groups as they see fit. If the co-ordinator feels the discussion has reached a natural conclusion, he should try to pick another suitably interesting one from the list.

(3) Summary. Each co-ordinator will present (5 minutes max) for the whole group the findings of his small group in discussing the issues raised.

4 Presenter Resumes

Geoff Bache is an experienced software developer and XP coach, working for the software product company Carmen Systems in Gothenburg, Sweden. He has been interested in XP acceptance testing since 2000 and working on developing the approach that has become TextTest and xUseCase since then. He has presented papers on Acceptance Testing at the XP conferences each year since 2003.

Rick Mugridge teaches and runs projects in agile software development for software engineering students at the University of Auckland. He has presented various papers on acceptance testing and other topics at agile conferences over the last two years, and is on the program committee for XP2005 and Agile 2005. He ran tutorials on Fit at several agile conferences in 2004, as well as running several workshops. He is an experienced developer, and consults to industry. He is the author, with Ward Cunningham, of "Fit for Developing Software", Prentice-Hall, June 2005. He developed the FitLibrary, which extends Fit.

Brian Swan is an Agile mentor with Exoftware, and has extensive experience in both the technical and the management aspects of Agile. He has successfully led a variety of teams transitioning to Agile, and trained both developers and managers in Agile thinking and practice. Brian has specific technical experience in the financial services and telecoms sectors, and his work with Exoftware and Agile has taken him to a variety of companies.

Hands-on Domain-Driven Acceptance Testing

Geoff Bache[1], Rick Mugridge[2], and Brian Swan[3]

[1] Carmen Systems AB, Göteborg, Sweden
geoff.bache@carmensystems.com
[2] University of Auckland, New Zealand
r.mugridge@auckland.ac.nz
[3] Exoftware, Dublin, Republic of Ireland
bswan@exoftware.com

Abstract. A recent phenomenon in the world of acceptance testing is tools that emphasize the creation of a domain language in which to express tests. The benefits of this are twofold: customers and testers are more likely to get involved in tests expressed in a language they understand. Also, tests that express intentions rather than mechanics tend to be much easier to maintain in the long run as they do not break when circumstantial things change.
The aim of this workshop is to see how tools that support this work in practice. The presenters have each been involved in the development of such a tool, TextTest+xUseCase, Fit+FitLibrary and Exactor, respectively, and there is room for attendees to bring their own tools along too. We aim to learn enough about these tools to compare and contrast them with each other, as well as with agile approaches that are less focussed on the creation of a domain language.

1 Intended Participation

We hope that between 15-20 people will participate, though the format is fairly flexible in terms of numbers attending. Each participant should identify briefly what experiences they have of Acceptance Testing, particularly of the domain-driven variety. Participants are encouraged to describe in a few words an application that they wish to test (including which operating system it runs on), install it on a laptop and bring the laptop to the workshop. Participants doing so will be given priority over those that do not.

If potential participants know of other tools that also support creation of a domain language for acceptance testing, they can also submit these along with a brief description. Subject to review, these can then also be tried out in the workshop.

2 Materials Needed

A laptop is useful but not essential. Should be provided by those that provide test applications, as above.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 296–298, 2005.
© Springer-Verlag Berlin Heidelberg 2005

3 Audience and Benefits of Attending

Intended audience is anyone wanting to learn about domain-driven approaches to acceptance testing. This includes the presenters who want to learn about each others' approaches! This probably includes developers, testers and customers. People without experience in the tools (or even in acceptance testing at all) are welcome. The benefits are gaining insight into the available tools from those who designed them without the danger of a 'hard sell' on a particular tool alone.

4 Outline of the Workshop

The workshop will carry on for a half a day. It is subdivided as follows:

(1) Presentations. Each framework will be briefly presented (15 minutes) by its designers.

(2) Practice. We will list the candidate applications for testing and choose the most interesting ones by some suitably democratic means. For each one chosen, it will be paired as desired with a tool and the application owner and framework designer will sit down and try to explore what testing that application with that tool would involve, either writing tests in practice or discussing what is involved and particular challenges that might be thrown up.

Any attendees who feel sufficiently experienced in using one of the tools can also help application owners to write tests. Anyone without an application will be encouraged to take an active part in these sessions, suggest tests, techniques, offer opinions. There is no requirement that they should remain in one session, they are free to wanderaround and compare and contrast.

(3) Summarising. The application owners will briefly (max 5 minutes) present for the whole group what they discovered in attempting to write tests (with whatever tools they managed to try) for their application.

5 Presenter Resumes

Geoff Bache is an experienced software developer and XP coach, working for the software product company Carmen Systems in Gothenburg, Sweden. He has been interested in XP acceptance testing since 2000 and working on developing the approach that has become TextTest and xUseCase since then. He has presented papers on Acceptance Testing at the XP conferences each year since 2003.

Rick Mugridge teaches and runs projects in agile software development for software engineering students at the University of Auckland. He has presented various papers on acceptance testing and other topics at agile conferences over the last two years, and is on the program committee for XP2005 and Agile 2005. He ran tutorials on Fit at several agile conferences in 2004, as well as running several workshops. He is an experienced developer, and consults to industry. He is the author, with Ward Cunningham, of "Fit for Developing Software", Prentice-Hall, June 2005. He developed the FitLibrary, which extends Fit.

Brian Swan is an Agile mentor with Exoftware, and has extensive experience in both the technical and the management aspects of Agile. He has successfully led a variety of teams transitioning to Agile, and trained both developers and managers in Agile thinking and practice. Brian has specific technical experience in the financial services and telecoms sectors, and his work with Exoftware and Agile has taken him to a variety of companies.

How to Sell the Idea of XP to Managers, Customers and Peers

Jan-Erik Sandberg and Lars Arne Skår

BEKK, Norway

Summary. Attending this workshop will guide you in starting an XP project for all the right reasons, while maximizing the potential for success.

Abstract. This workshop aims to gather developers with XP experience, project managers and customers. We want this group to share and discuss experiences, thoughts and ideas on starting an XP project with the right foundation. The goal is to enable the participants to bring an understanding of the agile process and the values it gives back to their own organization and customers. In order to do so, we have to understand what is pressing the customer and what she cares about. The organizers have been through several successful processes in delivering the starting and ongoing values of an XP approach, both in their own organization and the customer side. Most of them were used to waterfall and similar approaches. After experiencing more control and confidence in their projects during the agile process, they report back to us with an eagerness to continue and further develop the process. In addition, they experienced a flexibility that they just could not get with their former habits. The most effective way of explaining the benefits of an agile process successfully is to communicate in a way that is meaningful for both the customer and your organization.

Participants: The organizers wishes to gather between 12 and 14 people to discuss and share ideas on what values XP projects actually provides for customers and managers.

Please submit submission paper to: larsarne@extremeprogramming.no.

Materials: The workshop will use flip over and whiteboards for discussions and presenting the ideas in an effective manner within a small group.

Audience: The participants should be motivated to discuss and share ideas on what values XP projects actually provides for customers and managers. The organizers want participants from the following groups:

- Developers with XP/agile experience
- Project managers with a strong interest in XP/Agile
- Customers

Organizer expectations: Jan-Erik Sandberg, one of the organizer is working on a book on how to sell XP and we wish to use this workshop to gather realistic situation experiences as well as thought and ideas on how to overcome social, practical and other challenges in starting XP or agile projects. This includes identifying the right kind of projects to run agile processes on. Thus, the goal is to gather people who have

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 299–301, 2005.
© Springer-Verlag Berlin Heidelberg 2005

a solid understanding or interest in agile processes, and have thought through what kind of projects and teams it will work successfully on. By gathering their experiences, thoughts and ideas on how such processes actually will benefit their customers, the goal is to make sure XP projects are started with the right foundation and right understanding. We also believe that sound reflections on the true values in such processes for these stakeholders are necessary in order to reap the true benefits of an agile process.

Outline of theme and goals: Although most programmers appreciate and buy into the practices and techniques behind XP after learning about it, most of us have experienced that there still can be difficult to communicate or even identify the value propositions for the business side in a language meaningful to them. In addition we have seen analyst companies writing critical papers on XP/agile, and authors have written complete books on the subjects, for example, the case against XP by Matt Stephens, the fundamental problem with extreme programming by Greg Vaughn. Although critical, they bring up some valid points that can make customers even more concerned when faced with the question on running agile or not. Some customers may not even know, understand or appreciate what the value propositions of XP/agile are.

To overcome this, we have to understand what is pressing the customers and what they really care about.

Another interesting observation is that the market and the business side now demand a more flexible system delivery approach. Analyst companies are writing papers on the need for IT departments to respond more quickly and more correctly to such demands. One of them states that it is no longer accepted to blame delivery problems on shifting requirements; shifting requirements is now a fact of life and needs to be dealt with accordingly. Thus, an agile process may actually be a necessity to respond to such expectations.

The workshop is planned to run as follows:

- Introduction from the organizers on why this subject is important, including summary of the position papers
- Break out in 2 groups of 5-7 people to discuss for ½ hour:
  - What are characteristics of projects that works best on agile processes?
  - What values do agile projects provide; with respect to the customer
- Each group present findings with feedback and debate from other group for ½ hour. The groups should challenge each other to make sure that their points are understood the way they should be.
- Coffee break
- Facilitated common discussion with a gold fish format, 1 hour
  - The discussion starts with 4 people in a panel to debate what values agile projects actually provide
  - The rest of the participants may ask questions or challenge the panel
  - If a participant challenge the panel, or raise an opinion, he/she is to sit in the panel, and one of the panelists is taking a seat among the rest
- Summary from the organizers

We have seen that this format is very effective in creating energy in the discussion as well as leaving space for anybody who has an experience or opinion to share.

Organizer 1: Jan-Erik Sandberg founded extremeprogramming.no, 4 years ago, with a strong dedication to evangelize extreme programming to Norwegian customers and developers. He has presented XP on several international events - at TechED Europe 2004 he was rated one of the "Top Speakers". For several years, he has been in charge of the agile track at one of Norway's largest software conferences. As a Microsoft Most Valuable Professional, he has a strong focus on .Net-based solutions and is a requested speaker at many Microsoft Events. He runs the Microsoft Developer group at BEKK, a Norwegian consultancy company. He has also founded The Norwegian SQLServer User Group (NSUG.NO).

Organizer 2: Lars Arne Skår joined extremeprogramming.no in 2003, and shares Jan-Eriks ambition to evangelize extreme programming to as many as possible. As a CTO in BEKK, he has worked actively on using XP as a base for system development in BEKK. He typically acts as an agile coach on projects to ensure the projects follows the most important practices in order to get the true benefits of an agile project. This has created a strong interest among project managers in BEKK who now are actively promoting agile projects. Lars actively participates at OOPSLA and Norwegian conferences as conference speaker and in workshops. During his 15 years of experience in the consultancy industry, he has always had a strong focus on training and coaching others. In addition to act as a coach, he has also been an instructor on several international training events.

Agile Contracts

How to Develop Contracts that Support Agile Software Development

Mary Poppendieck and Tom Poppendieck

Poppendieck.LLC, 7666 Carnelian Lane, Eden Prairie, MN, USA

Abstract. Agile Development sounds great, but it depends on the ability to determine the details of scope as the system is developed, driven by feedback from customers and users. Much software development is done under contract, where there is often a requirement to determine the details of the system early in the development process.

Goal: The goal of this workshop is to discuss, discover, and document specific details about how contracts can be structured and worded to support agile software development, and make this information available for posting on a web site.

Process: The workshop itself will have three parts.

Part 1: Each participant will be asked to lead a 15-20 minute discussion about contracts, focusing on how contacts have been either successfully used for agile development, or how contracts have prevented agile development from being successful.

Part 2: We will discuss the main types of contract forms, their weaknesses when used for Agile Development, and recommended mechanisms or modifications for using each type of contract to make it more amenable to agile software development. We will also categorize the main parties to software development contracts and the most common parings of these parties.

Part 3: We will map contract forms to contracting parties. This section involves group discussion about the implications of the earlier part of the meeting and a summary of the results.

Benefits of attending: This workshop is aimed at people grappling with the issues that result when a development organization attempts to do agile development under contract. Anyone involved in this type of situation will benefit from sharing their experience with colleagues in similar situations.

What we expect to learn from the workshop: To date we have seen successful use of contracts in agile development on a case-by-case basis, but have found little universal guidance for how to proceed with agile contracts. As the use of agile development grows, we believe that more universally applicable practices will arise. The purpose of this workshop is to provide a forum for the agile community to work toward this end.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, p. 302, 2005.
© Springer-Verlag Berlin Heidelberg 2005

When Teamwork Isn't Working

Tim Bacon[1] and Dave Hoover[2]

[1] Prime Eight Ltd, UK
tBacon@primeEight.co.uk
[2] Thoughtworks, USA
dHoover@thoughtworks.com

Abstract. XP is a team game. It relies on teamwork to be successful. But sometimes our teams don't work as well as we would like. This workshop examines real projects from different angles to explore answers to three powerful questions:

- How can we tell that our team isn't working?
- Which are the root causes of our problems?
- What actions can we take to improve our teams effectiveness?

1 Audience and Benefits

This workshop will help anyone who has ever worked in a poorly performing team to learn from their experiences, and to apply that learning in their current team. Participants will come away with a greater awareness of why teams become stuck in cycles of ineffective behaviour and how intervention strategies can help them to become unstuck.

2 Theme and Goals

There is plenty of learning material available for XP teams who wish to improve skills such as refactoring or testing. But there are fewer sources to turn to when it comes to improving important interpersonal skills such as collaboration and communication. Teams can be at a loss on how to proceed when problems surface in these areas, and their effectiveness becomes more and more compromised the longer that issues are not addressed.

However research in fields such as social and behavioural psychology is directly applicable to improving the personal interactions in software teams. This workshop introduces several techniques from other fields and relates them to real software development situations where they can provide positive benefits.

3 Content and Process

The workshop uses a series of facilitated story-telling activities to explore contextual anecdotes that illustrate the teamwork problems faced by XP teams. At the start of each activity guidance is provided using examples from the authors' experience.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 303–304, 2005.
© Springer-Verlag Berlin Heidelberg 2005

3.1 Timetable

Introduction and group warm-up: 25 minutes

Story-telling, clue detection, and reaction mapping: 50 minutes

- Working in small groups, swap anecdotes about teams that you feel weren't working at their full potential
- Identify the clues in the story that demonstrate that teamwork wasn't working
- Map the reactions of the people in the team to the clues in the story

Exploration: 45 minutes

- Look again at the clues in the stories and dig into their possible causes
- For each cause, suggest actions that might address the root of the problem
- Select an action that is likely to have the most benefit without causing harm

Sharing the insights: 40 minutes

- One person in the group stays with the outputs to explain them to visitors
- Other members visits other groups to see their outputs
- Roles swap so that all group members have the chance to visit and be visited

Wrap up & appreciations: 20 minutes

Time required 3 hours, excluding breaks

3.2 Participants

The number of participants is limited to 20. Registration is required in advance. Participants should prepare for the workshop by selecting teamwork stories to share and explore with others.

3.3 Materials

At least four flipcharts plus coloured marker pens, sticky tape, wall space, and chairs and tables that can be rearranged to suit various small group sizes.

4 Presenters

Tim Bacon is a self-confessed "people person". He is a passionate advocate of Agile processes, software craftsmanship, and test-driven development. Tim has been a public speaker for a number of years and is currently working as an independent coach and consultant.

Dave Hoover has been developing software for 15 minutes. He used to have a respectable job as a family therapist. Dave still wonders how he got here.

The Origin of Value: Determining the Business Value of Software Features

David L. Putman[1] and David Hussman[2]

[1] Centaur Communications plc, London W1F 7AX
david.putman@centaur.co.uk
http://www.centaur.co.uk
[2] SGF Software,3010 Hennepin Ave,. S. Suite 115, Minneapolis, MN 55408-2614, USA
david.hussman@sgfco.com
http://www.sgfco.com

Abstract. Many authors have presented techniques for using business value to measure and prioritise requirements. Unfortunately, although there is much written about using value in this way, there is very little written about how to measure value itself. This workshop is intended to help the participants answer questions such as: Where does value come from? How do we know what is valuable and what is not? Who determines value? Are there different types of value and, if so, how can we compare them?

Participation

Open to anyone attending XP2005.

Materials Required

A room large enough to hold the participants, a whiteboard, flipboard with pads, a beamer and screen. All other materials will be provided by the organisers.

Intended Audience

Customers, Managers, Developers, Testers and anyone else involved in software development from newcomers to masters.

Benefits of Attending

After attending this workshop, the participants should have some understanding of where value is derived from and how to apply the techniques and calculations demonstrated to their own projects.

Session Duration: Half Day

Session Type: Workshop

Session Theme and Goals:

The goal of this workshop are to demonstrate, with the active participation of the attendees, a range of techniques that can be used to discover the origins of value in

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 305–307, 2005.
© Springer-Verlag Berlin Heidelberg 2005

their own projects and organisations. The workshop leaders will show the participants how approaches like persona modelling and value scorecards can be used together to identify and objectively measure the value of the requirements for a project. Once measured, the value can be used to help identify the features that should be included in a Minimum Marketable Feature Set (MMFS). The workshop is intended to be fun and consists of practical exercises with much emphasis being placed on the active participation of the attendees.

Process

Introduction (20 mins):

- Introduce presenters and give an overview of the workshop.
- Warm up exercise
- Divide the participants into groups of four or maybe more.
- Give each group an imaginary project

The Groups Identify the Values (60 mins):

- Identify the roles of all the stakeholders in the project ()
- For each role the a group identifies, they will create a persona who they will name and draw a picture of
- The pictures of the personas will be stuck on the wall
- For each persona, the group will then identify a set of values that they write on post-it notes and stick to the personas.
- The group then spends some time evaluating the personas' values and clustering them
- By counting the number of personas related to it, each value is given a weighting and the participants create a scorecard for the top five values for their project.

Story Writing (30 mins):

- The groups write story cards for their projects

Story Scoring 40 mins

- The stories are assessed one by one
- Each participant has one vote which he must assign to a value for the story
- The number of votes each value has is multiplied by its weighting to give a subtotal.
- The value subtotals are summed to give a total value for the story.
- When all the stories have been scored, the ones with the highest score are (obviously) of the most value and should be part of the MMFS. (there are usually some surprises here)

Discussion and Wrap Up (30 mins)

- There are many questions to be answered and there are usually many things learned during this workshop, i.e. hidden values in organisation that are not overtly expressed but that sometimes govern projects and behaviour.

Presenter Learning

The presenters hope to learn more about the participants' values, their reactions to the workshop and expect to discover many ways of improving it.

David Putman

David was previously a mentor for Exoftware, where his role took him to a variety of organisations, and he has acted as an advisor on the management of software development projects to companies in three continents. He is now the manager of CentaurNet. David regularly presents papers and tutorials on the management and practice of software development at national and international events, including XP2002, XP2003 and XP2004. He writes the "Models and Methodologies" column for Application Development Advisor magazine and has had articles published in other publications. His main interests are the management of people and software development, learning organisations, and making work satisfying to all those involved.

David Hussman

David has been employed as a software geek for over 10 years. In that time, he has developed in the following fields: medical, digital audio, digital biometrics, retail, and educational. Somewhere along the way, David moved toward an odd way of defining, communicating, and developing software that worked. When someone organised a better version and started calling it XP / Agile, he felt right at home. Motivated to see IT folks succeed and smile, David has evangelised XP by working as developer, coach, customer, and manager on XP projects for 3.5 years. David presented a poster session at XP2002, had several papers accepted and presented at XP2003 and held workshops at XP2004.

The Drawing Carousel: A Pair Programming Experience

Vera Peeters[1] and Peter Schrier[2]

[1] Tryx, Colomastraat 28, B-2800 Mechelen, Belgium
Vera.Peeters@tryx.com
[2] TriCAT Agileon, Meyerweg 24, 8456 GG DE KNIPE, The Netherlands
pl.schrier@tricat.nl

Abstract. The participants will *experience* how a pair programming team works. Working in a single pair is different from working on your own: You have to articulate what you want to do, the other person (probably) has some different insights, and the result will be different from what you would have done on your own. But.. this is only half of the story: working in a Pair Programming Team has many more advantages: knowledge is spread, the team creates an own style, parallel development becomes easy, truck factor (http://c2.com/cgi/wiki?TruckNumber) is reduced. The team creates the product (instead of individual programmers creating the parts), the product is a whole, not a mixture of individual results. You will learn about the necessity of agile tools like daily standup meetings, pair rotation, coding standards and collective code ownership, how they work, and what their effects are. You will experience improvements for the quality of you product, and for the productivity of the team.

1 Audience and Benefits of Attending

Intended audience: Developers, managers, coaches, teachers

Experience level: Beginners, intermediate and experts

Benefits: Real life experience, but no programming involved! See and feel how this can possibly work without putting your real project at risk!

2 Session Type and Duration

½ day tutorial.

3 Content Outline

How is it possible to gain "real-life-experience" without programming?

In this simulation, we want to focus on some of the practices, and we make abstraction of everything that is not crucial. Programming is not crucial, being part of the team, that is what matters. The intention is that everybody can experience how it feels to be part of a Pair Programming team, we don't want to exclude people who don't know how to program.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 308–310, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In this simulation, you will get an assignment to visualize a given story.

When you start the implementation, you only have a rough idea about the story content and plot. You will not get all the thiny details up front, you will find out the exact requirements while you proceed with your tasks.

And they will change... You will have to change the details! But you are not doing this on your own, when you want to succeed, you have to surrender to the emerging "team spirit". You will see how the team can become an entity, how the whole is more than the parts, how individual decisions always contribute to the team's awareness of the product.

This session is not about *how to work well with the person next to you*, it is about how it feels to be assimilated in a real team (resistance is futile!).

Process and Timetable

0:00	0:05	intro: explain about pair progamming
0:05	0:10	Pair Draw exercise (cfr http://www.industriallogic.com/games/pairdraw.html)
0:10	0:20	Reflection
0:20	0:30	explain about pair programming teams
0:30	0:35	Excercise: define strategy
0:35	0:40	Reflection
0:40	0:50	day 1
0:50	1:05	reflection and coaching
1:05	1:15	day 2
1:15	1:30	reflection and coaching
1:30	1:40	day 3
1:40	1:50	reflection
1:50	2:00	day 4
2:00	2:10	reflection
2:10	2:20	day 5
2:20	2:30	reflection
2:30	2:40	day 6
2:40	3:00	final reflection

After 3 "days", the team will have developed an identifyable "team spirit", an entity that is more and different from its parts, and the team will have created the start of a product.

The next "days", they will be introduced with different problems, all if which will require a lot of communication, collaboration and self-organisation if they want to solve them.

4 Presenter Resumes

Vera Peeters is an independent consultant. She runs her own company TRYX. She has 15 years experience in developing software systems, especially object-oriented development n all kinds of high-technological environments. She has been practicing agile ways of working for several years now. Vera has presented workshops at sev-

eral conferences (XP200X, XPUniverse, OT200X). She is a co-organizer of the XPDay Benelux (http://www.xpday.be) and of the Javapolis conferences (http://www.java polis.org). In 2001, she founded the Belgian XP User Group (http://www.xp.be) together with Pascal Van Cauwenberghe.

Peter Schrier is founder of TriCAT Agileon. Driven by his passion for sustainable product-development, he supports software development teams create clean working software using Agile methods and techniques. Peter has been active in software-engineering since 1992.

5 History of Tutorial

This tutorial has been presented at XPDay Benelux 2004, Mechelen, Belgium and XPDay Germany 2004, in Karlsruhe, Germany.

Agile Development with Domain Specific Languages

Scaling Up Agile – Is Domain-Specific Modeling the Key?

Alan Cameron Wills[1] and Steven Kelly[2]

[1] Microsoft
+44 122 347 9719
awills@microsoft.com
http://blogs.msdn.com/alan_cameron_wills/
[2] MetaCase
+358 14 4451 401
stevek@metacase.com
http://www.metacase.com/blogs/stevek/

Abstract. This workshop will investigate the application of Domain Specific Languages within Agile development. A Domain Specific Language (DSL) is designed to express the requirements and solutions of a particular business or architectural domain. SQL, GUI designers, workflow languages and regular expressions are familiar examples. In recent years, Domain-Specific Modeling has yielded spectacular productivity improvements in domains such as telephony and embedded systems. By creating graphical or textual languages specific to the needs of an individual project or product line within one company, DSM offers maximum agility. With current tools, creating a language and related tool support is fast enough to make DSM a realistic possibility for projects of all sizes.

1 Introduction

A Domain Specific Language (DSL) is designed to express the requirements and solutions of a particular business or architectural domain. SQL, GUI designers, workflow languages and regular expressions are familiar examples in 'horizontal' domains. Each allows its user to concentrate on expressing what is required in terms directly related to the domain, leaving the platform to apply the most appropriate implementation patterns – a feat made possible by its restricted scope. Problems that were very substantial projects before the advent of these languages and their implementing engines, are now the work of an afternoon.

Can individual projects or product lines within one company, make similar gains in agility and productivity by creating graphical or textual languages specific to their own domain?

Domain Specific Modeling is the creation and use of DSLs, often graphical DSLs with domain-specific generators that create full production code directly from models [1–3]. In recent years, DSM has yielded spectacular productivity improvements, particularly in vertical domains where many similar variants of a generic system are to be developed, e.g. telephony and embedded systems [4–9].

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 311–314, 2005.
© Springer-Verlag Berlin Heidelberg 2005

A particular concern in agile methodology is how to scale the approach to large projects [15]. DSM is a particularly effective tool to help decouple top-level requirements from implementation layers, allowing a large project to be separated into several well-decoupled smaller projects. With current tools, creating a language and related tool support is fast enough to make DSM a realistic possibility for projects of all sizes [11, 12]. By creating languages and generators specific to the needs of an individual project or product line within one company, DSM offers maximum agility [10].

This workshop will investigate the application of Domain Specific Modeling within Agile development.

2 Workshop Topics

Topics to be tackled include:

- Process and Roles surrounding DSM
  - DSL users and DSL designers – are they separate?
  - Does DSM affect the development process?
  - How effective is DSM for high-level design of big projects? Or is it better at focused aspects inside a design?
- How do you design a DSL?
  - Identifying variable aspects of the domain – do you make a model or do you look at existing code?
  - Gradual evolution or big upfront design? – optimizing investment
  - Maintaining compatibility as the language evolves
  - Creating the execution platform from existing systems
- Ecology of DSLs
  - Designing a DSL from fragments of others
  - DSL repositories and markets
- Return on Investment
  - When to use DSM
  - Are DSLs syntactic sugar on a framework API?
- The development cycle
  - Testing in terms of the DSL and its abstractions
  - Debugging via a DSL
- Choosing a type of DSL
  - Graphical, textual, interactive
  - Uses of the DSL in an agile process; user roles
- Defining DSLs
  - Grammars and editing tools
  - Constraints and validation
- Generating code and artifacts
  - Generating or mapping code and other artifacts
  - DSLs used for validation and testing
  - Composing aspects of multiple DSLs

3 Audience and Benefits of Attending

The intended audience consists of developers and technical managers interested in finding out more about DSM. Experience of product lines, building product frameworks or creating DSLs is a bonus, but by no means necessary. Benefits of participation include a better understanding of:

- When to use DSM, and what for
- How to adjust the development process to use DSM
- How to create DSLs and use DSM

4 Submissions

Position papers are invited from each prospective participant. Each paper should:

- Have one author, and state his/her experience or interest in the area
- Explore a single topic related to the field outlined above
- Be no longer than two pages
- Clearly distinguish solid experience from wild ideas (both of which are welcome)
- Pose interesting questions for the workshop to consider
- Include web references to relevant material

Accepted position papers will be circulated to participants on the workshop website. In addition, the organizers will circulate short example DSLs and applications of them. These will be used as a basis for discussion at the workshop.

5 Workshop Format and Schedule

- Short presentation from workshop leaders.
  - Overview of positions and topics of interest
  - Setting vocabulary
- General goldfish-bowl discussion: “Can DSM scale up Agile?”
- “Cocktail party” of topics
  - This is like a poster session: some people put up headings on flipcharts, and then people wander between the discussions
- Separation into Working Groups based on popularity of topics
- Working Group discussions
- Reports of Working Groups
- General discussion

6 Workshop Leaders

Alan Cameron Wills (Microsoft) works on DSL tools for the Visual Studio IDE. Before joining Microsoft, he was a consultant in development methods, and co-developed the Catalysis approach to software development. He has run successful workshops and tutorials in related topics at SPA2005 and OT97-04.

Steven Kelly is CTO at MetaCase, and has been the lead on the MetaEdit+ DSM tool since 1996. He is also co-founder of the DSM Forum, has served on the committee of the OOPSLA workshops on DSM since 2001 [13–14], and has been giving metamodeling and DSM tutorials around the world since 1993.

References

1. Kelly, S., Tolvanen, J-P, "Visual domain-specific modeling: Benefits and experiences of using metaCASE tools", *Proceedings of International workshop on Model Engineering, ECOOP 2000, (ed. J. Bezivin, J. Ernst), 2000.*
2. Pohjonen, R., Kelly, S., "Domain-Specific Modeling", *Dr. Dobb's Journal*, August 2002.
3. Greenfield, J., Short, S, Cook, S., Kent, S., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley, 2004.
4. Kieburtz, R. et al., "A Software Engineering Experiment in Software Component Generation," *Proceedings of 18th International Conference on Software Engineering*, Berlin, IEEE Computer Society Press, March, 1996.
5. Long, E., Misra, A., and Sztipanovits, J., "Increasing Productivity at Saturn," *IEEE Computer*, August 1998, pp. 35-43.
6. Sztipanovits, J., Karsai, G., and Bapty, T., "Self-Adaptive Software for Signal Processing," *Communications of the ACM*, May 1998, pp. 66-73.
7. MetaCase, MetaEdit+ Revolutionized the Way Nokia Develops Mobile Phones, http://www.metacase.com/papers/MetaEdit_in_Nokia.pdf, 1999.
8. Weiss, D., Lai, C. T. R., *Software Product-line Engineering*, Addison Wesley Longman, 1999.
9. Moore, M., Monemi, S., Wang, J., Marble, J., and Jones, S., "Diagnostics and Integration in Electrical Utilities," *IEEE Rural Electric Power Conference*, Orlando, FL, May 2000.
10. Cook, S., "Domain-Specific Modeling and Model Driven Architecture", MDA Journal, http://www.bptrends.com, January 2004
11. Nordstrom, G., Sztipanovits, J., Karsai, G., and Ledeczi, A., "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *IEEE ECBS Conference*, April 1999.
12. Kelly, S., "Tools for Domain-Specific Modeling", *Dr. Dobb's Journal*, September 2004.
13. Tolvanen, J-P., Kelly, S. Gray, J., Lyytinen, K., (eds.) Proceedings of OOPSLA workshop on Domain-Specific Visual Languages, Tampa Bay, Florida, USA, University of Jyväskylä, Technical Reports, TR-26, Finland, 2001.
14. Gray, J., Rossi, M., Tolvanen, J-P., (eds.) Domain-Specific Modeling with Visual Languages, Special issue of Journal of Visual Languages and Computing, Vol. 15 (3-4), Elsevier, Jun-Aug, 2004.
15. Eckstein, J. *Agile Software Development in the Large*, Dorset House, 2004.
16. Czarnecki, K., Eisenecker, U. *Generative Programming*, Addison Wesley, 2000.

A Thinking Framework for the Adaptation of Iterative Incremental Development Methodologies

Ernest Mnkandla

Monash University, South Africa, School of Information Technology, Private Bag X60, Roodepoort, Johannesburg, 1725, South Africa
Ernest.Mnkandla@infotech.monash.edu.au

Abstract. Methodology adoption issues in the agile world are faced with challenges such as the growing trend towards the use of a group of relevant cross-methodology practices from the agile family as opposed to the adoption of individual methodologies. While this may see agile processes precipitating towards more mature software engineering processes, the challenge is that some of the specific agile methodologies may become extinct over time. This research therefore contributes a key addition to knowledge by developing a thinking tool that will guide system developers to informatively select agile practices from the entire agile methodologies family that are relevant to the project at hand. This paper proposes a novel modeling technique for tailoring methodologies to a particular environment using the family of methodologies approach. The Agile Methodologies Generic (AMG) model considers agile methodologies as a group of methodologies with common parameters that can be used to model the entire group. Based on this model, methodology parameters can be identified that are common among the different agile methodologies making it possible to create a set of relevant agile practices that can be used in an organization. The original concepts of the model are based on two foundations: 1) the philosophy of Jim Highsmith's Adaptive Software Development (ASD) methodology. ASD focuses on the speculate, collaborate and learn cycle iteratively which is fundamental to agile development, and 2) the concept of organizational maturity levels which says that mature organizations families of repeatable and automated processes. It is from such a perspective that AMG was born. AMG considers agile methodologies at an abstract level where the four values of the Agile Manifesto are assumed to collectively constitute basic philosophy of all agile methodologies. The phases of AMG (mechanistic, organic, and synergistic adaptation) are therefore analyzed in light of the values of the project at hand. The benefits of such a technology is the provision of a thinking tool that assists software development teams to effectively tailor agile methodologies to their project environments. The tool has been applied by two software development organisations and the results are being analysed.

Significant Problems: Selection and tailoring of agile methodologies is moving towards choice of most relevant practices from the agile family rather than selecting specific methods. What would be best way to select and tune the practices from such a perspective?

Solution: investigate behavior of software developers in the selection of methodologies. Propose a family of methodologies framework for the selection and tailoring of agile processes.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 315–316, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Proposed Approach: Do a theoretical replication using case studies (one successful and one failure). Formulate tentative theory and test it using Action research.

Results so Far: The project in which action research was applied eventually failed. The reason may be due to failure to apply the concepts proposed in the model.

Illustration of the AGM Model

The reasoning process that can be followed for example when performing a selection of tools applicable to a given project is shown in the graph on figure 1.

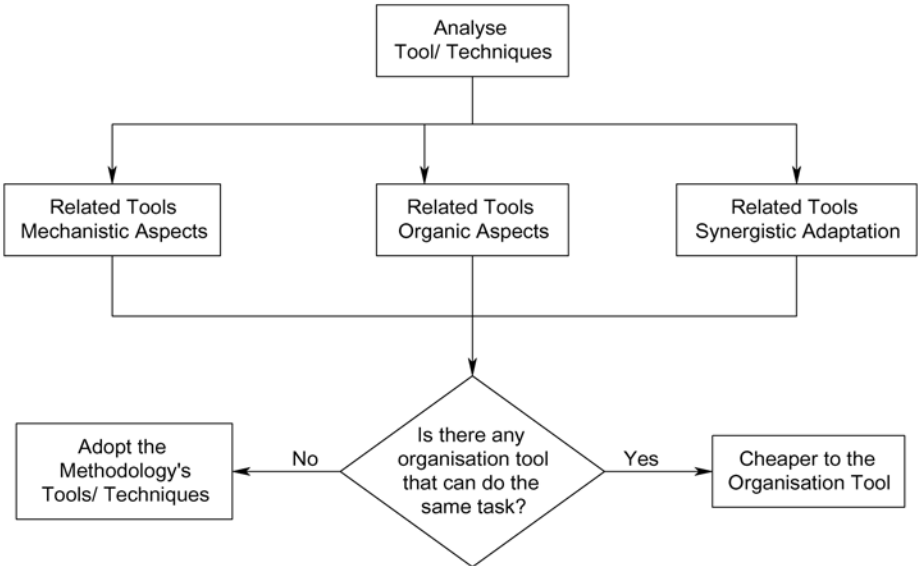


Fig. 1. AMG Reasoning Process Graph

Exploring XP's Efficacy in a Distributed Software Development Team*

Alessandra Cau

DIEE, Universitá di Cagliari
alessandra.cau@diee.unica.it
http://agile.diee.unica.it

Abstract. Since the first edition of Beck's book [1], the Extreme Programming (XP) has attracted attention from academia and industry, and its values, principles and practices are becoming increasingly popular. Strong interest in the software engineering community has generated substantial literature and debate over Extreme Programming. However, current research on the applicability and effectiveness of Extreme Programming is still very scarce and researchers and practitioners need to assess concretely XP's advantages and drawbacks. One disadvantage, which has been noted, is that Extreme Programming is more effective for small to medium size projects with co-located team. Despite such observation, Beck asserts that XP can work with teams of any size and also multi-site [2]. The main goal of this research is to evaluate the effectiveness of Extreme Programming, when the size of development team is large and distributed.

There are two different approaches to investigate the applicability and effectiveness of a software method: empirical studies and simulation process modeling. These approaches are usually applied separately, but there are many interdependencies between simulation and empirical research. On the one hand simulation model generalizes empirical studies and provides a framework for the evaluation of empirical models. On the other hand, empirical studies provide the necessary fundament for simulation models because through empirical studies it is possible to collect real data to validate the simulation model. In the present research these two approaches are combined.

My research is grounded on the following steps:

1. Understand if XP values are suitable to multi-site development as they are for co-located teams. Analyze how to apply XP practices for distributed and large team and seek which among XP principles will become more important.

* This work was supported by MAPS (Agile Methodologies for Software Production) research project, contract/grant sponsor: FIRB research fund of MIUR, contract/grant number: RBNE01JRK8.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 317–318, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2. Evaluate the applicability and efficiency of Extreme programming for large and multi-site projects using:
   - XP-Evaluation Framework (XP-EF). It is an ontology-based benchmark which defines the metrics that must be collected for each case study and to assess the efficacy of XP [3]. The XP-EF has been used to structure several XP case studies. The XP-EF will be updated to new XP.
   - Discrete Event Simulation Model. My research group and I have developed a simulation model to evaluate the applicability and effectiveness of XP process, and the effects of some of its individual practices.

3. Validate results. My research will be validated on a real academic case study. This project [4] involves almost 30 undergraduates students, who work as a distributed team in an open source project using some agile practices.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
2. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change- Second Edition. Addison-Wesley (2004)
3. Williams, L., Layman, L., Kreb, W.: Extreme Programming Evaluation Framework for Object-Oriented Languages–Version 1.4. North Carolina State University, Department of Computer Science, TR-2004-18 (2004)
4. MAD: Methodology agile distribuited. (http://www.unica.it/concas/MAD/)

Agile Methods for Embedded Systems

Dirk Wilking

Chair for Computer Science XI,
RWTH Aachen University

Abstract. The main goal to be answered by this Ph.D. thesis is whether there is a potential for a successful and powerful application of agile methods and related techniques to embedded systems development or not (cf. [2]). Regarding the special context of embedded system, there are some aspects to be mentioned as stated in [3]. These include the function oriented development which lead to early testing of the system, the use of target-processor simulation and the problem of hardware software co-design.

The first problem being addressed is the evaluation of well known sub-techniques like refactoring, TDD, fast development cycles, short design horizon, or similar methods in the context of embedded systems. A complementary approach consists of the elaboration of underlying root causes which make agile methods appear as a sound alternative to classic techniques. For example assumptions like source code degrading over time, non-costumer oriented development, overly complex systems, and wrong development focus could be checked. A possible subdivision of the causes can be done by distinguishing effects that are generated by agile methods and effects that typically exist in embedded system engineering. This can be regarded as an alternative upside down procedure which will more likely yield a justification for agile methods in embedded system development. Finding a causation with an appropriate prioritization appears more challenging and thus will be used only to verify techniques which have a strong effect.

The first step toward an assessment of agile methods has been started by executing a study during a lab course which is guided by the ideas described in [1]. Here, the students are divided into a planning group and an agile group, each developing a pre-crash system based on ultrasonic sensors. The two data collection mechanisms are a biweekly survey and a time recording log. The underlying aim is to show the influence of the planning horizon on embedded system development. This approach already sketches the main validation technique, which will be quantitative and composed of case studies and experiments. In addition, a case study with a high degree of variable control as proposed by [4] will be executed in order to guide the evaluation process to the most promising aspects of agile methods for embedded system development.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 319–320, 2005.
© Springer-Verlag Berlin Heidelberg 2005

References

1. Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. Using empirical studies during software courses. In R. Conradi and A. I. Wang, editors, *ESERNET 2001-2003*, LNCS 2765, 2003.
2. Peter Manhart and Kurt Schneider. Breaking the ice for agile development of embedded software: An industry experience report. *ICSE04*, pages 6–14, 2004.
3. Jussi Ronkainen and Pekka Abrahamsson. Software development under stringent hardware constraints: Do agile methods have a chance? In Michele Marchesi and Giancarlo Succi, editors, *Proceedings of 4th International Conference XP 2003*, 2003.
4. Outi Salo and Pekka Abrahamsson. Empirical evaluation of agile software development: The controlled case study approach. In F. Bomarius and H. Iida, editors, *PROFES 2004*, LNCS 3009, 2004.

Tool Support for the Effective Distribution of Agile Practice*
(Extended Abstract)

Paul Adams and Cornelia Boldyreff**

Department of Computing and Informatics, University of Lincoln

1 Introduction

Agile methods are quickly gaining notoriety amongst software engineers. Having been developed over the past decade, they now present a mature, lightweight alternative to the "classic" approaches to software engineering. Although agile methods have solved some of the problems of established software engineering practice, they have created some problems of their own. Most importantly, we can infer a, potentially problematic, requirement of collocation.

The usefulness of the agile methods is restricted by this requirement of collocation and by the requirement of small development teams. If these requirements can be loosened then it would be possible to apply agile methods to a larger arena of software development. This research intends to extend the usefulness of agile methods by defining a new paradigm for software engineering practice, the "Liberal" paradigm and providing tool support for processes within this paradigm.

2 Tool Support for the "Liberal" Paradigm

This new paradigm is based on the principles of agility combined with the experience of libre software process distribution. This paradigm offers important advantages as it will encompass all important features of both agile and libre software engineering practice in order to facilitate the distribution of effective agile practice. This paradigm will provide a new set of processes that allow the distribution of agile practice within the libre software paradigm, but also potentially improve the performance of collocated agile teams.

In this research the intention is to develop a distributed software engineering support system that will allow the effective distribution of agile practice within the "Liberal" paradigm.

Tool support for the "Liberal" environment will be provided in the form of a plug-in for the Eclipse IDE. Existing support for distributed agile practice is often formed as a naïve, ad hoc composition of existing tools. Environments such as this can offer effective solutions to this problem but are restricted in that they have not been specifically developed for this purpose. Unlike other research

* A complete edition of this paper can be found at http://eprints.lincoln.ac.uk/48/

** Project Supervisor

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 321–322, 2005.
© Springer-Verlag Berlin Heidelberg 2005

where process has been the focus, the principle contribution of this research shall be a tool set for supporting processes within the "Liberal" paradigm. This tool set must be rich enough to support distributed agile practice, but flexible enough to allow libre software practitioners the low-level freedoms they are familiar.

There are three types of tool that will be developed within this project: awareness tools, communication tools and task support tools. All of these shall be encapsulated within an Eclipse plug-in environment. The tasks initially identified for support include distributed story cards, virtual daily meetings, component integration and most importantly, pair programming.

3 Research Method

This research is largely based on empirical process. After thoroughly researching the requirements of agile programmers and distributed software engineering, I intend to iteratively implement and evaluate features for the distributed agile environment. Once the entire system is complete a thorough evaluation of the system through experimentation is planned followed by improvement (where required) and then further experimentation etc.

This research is focused on a bottom-up approach, that is, the development of a tool for supporting distributed agile practice; the development of a process for this tool and ensuring this process fits within the "Liberal" paradigm are secondary. However, there is an element of top-down approach, in that it has been possible to form some high-level descriptions of the "Liberal" paradigm and processes within it.

From the agile paradigm the "Liberal" paradigm inherits some of the high-level principles of the Agile Manifesto. However, these processes need to be more adaptable than the current agile processes to ensure that the principles do not conflict. For example, it may not be desirable to allow customer collaboration to restrict code production. From the libre paradigm the "Liberal" paradigm inherits the low-level freedoms that libre practitioners are afforded. It is also important for the "Liberal" paradigm to inherit scalability from the libre software paradigm.

4 Conclusions

We can infer from many agile methods that communication must take place in a collocated manner. This project aims to allow the distribution of effective agile practice by providing tool support that fits the "Liberal" paradigm without constraining its usage within specific software processes.

The main focus of this research is the creation of a tool that aids the distribution of effective agile practice. This tool must not only support agile practice but also offer support for the communication and awareness overheads that distribution causes. It is intended that the tool developed within this research will aid the distribution of effective agile practice in a manner that is relevant to both industrial practitioners and libre software practitioners and thus broaden the usefulness of the agile methods.

The Software Hut – A Student Experience of eXtreme Programming with Real Commercial Clients

Bhavnidhi Kalra, Chris Thomson, and Mike Holcombe

Department of Computer Science, University of Sheffield, Regent Court,
Portobello Street, Sheffield, S1 4DP
{c.Thomson,m.holcombe}@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk

Abstract. The University of Sheffield provides undergraduate students with a real experience of software engineering through a module entitled the Software Hut. Here, 2nd year students work in teams competing to build a real business solution for a real commercial client. In this exercise, eXtreme Programming is used. This article provides a few details of this innovative educational programme.

1 Software Hut

The Software Hut is a course module for the second year students where they have to develop software for real time clients and interact with them to understand their requirements and produce the desired results. It is used as a vehicle for investigating the processes of engineering a real software system for a real client in a competitive environment. It covers how to manage software development projects successfully and how to deliver software products that meet both client expectations and quality standards. The method of functioning is based on software development techniques such as the requirements engineering process; developing prototypes using different approaches such as visual builders; group project management; coding standards. Other important aspects of software development such as team management, conduct of meetings and action minutes are also focused on.

1.1 The Educational Objectives

The main goals to be achieved by the students are as follows:

- to gain experience in dealing with external clients from industry and in understanding and managing clients' expectations
- to experience the practical problems of constructing and managing a medium-sized software project in a competitive environment
- to examine selectively and use some of the tools and techniques available to solve these practical problems
- to understand the processes involved in the quality assurance of software and accompanying documentation
- to apply programming standards consistently

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 323–324, 2005.
© Springer-Verlag Berlin Heidelberg 2005

- to develop team working skills
- to prepare students for higher level project work, such as the individual Research Project and the Genesys Solutions software company

The students are divided into teams of four to six. Generally three or four business clients are found, either by advertising or word of mouth. Each client works with four to six different teams, the teams competing to build the best solution for their client. The results of this are high quality systems with complete documentation for users and maintenance. The students also have to sign a Non Disclosure Agreement to protect their client's confidential business details. Professional standards that are imposed include coding standards, a lightweight version of some IEEE document standards, testing standards, punctuality, etc.

The Software Hut has been running for the past 14 years and some of the clients that have worked with them are Doncaster Police, Pharmacovigilance, Fizzilink, Debt collection agent planner, SAP Recruitment Agency, Spanish Holiday property agent, Occupational Therapist PDA systems project. The Software Hut is therefore the first step to introduce students to the industry and its processes for software development in a highly professional framework.

2 Software Hut Framework

Each project team uses CVS and an Eclipse development framework with some in-house tools for managing an XP project such as a story card editor, metaphor designer and system test set generator as well as JUnit and related test tools. Weekly client and management meetings are held and students are required to maintain their project plans using the management tool and timesheets. The projects must be completed in 12 weeks of the University term and each student is expected to spend 15 hours per week on this activity – they are attending other classes in their course at the same time. Although students will have done programming and design modules before entering the Software Hut this will be the first time that they have encountered XP. An introductory lecture and some practical sessions, including a team building exercise and the course book, [1], are the only preparation for starting the project. Meeting their clients on a weekly basis is a strong motivation to learn about the techniques and practices of XP and has proved to be extremely successful.

Reference

1. M. Holcombe. The book of Genesys Solutions. http://www.dcs.shef.ac.uk/~wmlh/GenesysBook.pdf

Eclipse Platform Integration of Jester – The JUnit Test Tester

Simon Lever

3rd year student, Department of Computer Science, University of Sheffield
Regent court, Portobello Street, Sheffield, S1 4DP, UK

Abstract. Extreme programming (XP) emphasises the test-first strategy of developing software where if code passes unit tests developers gain more confidence in their software. Jester is a test tester for JUnit tests and thus allows developers to confirm their confidence in their tests and consequently in their code. Jester finds code that is not covered by JUnit tests and thus indicates either missing test cases or the redundancy of code that currently exists. The Eclipse IDE enables developers in any language to independently build tools that when combined together work as if they are part of a single integrated tool set. The implications of this open source IDE as an aid to software engineering are infinite and thus provide an ideal platform to nurture XP practices on.

1 Background

JUnit has already been seamlessly integrated with Eclipse, but Jester has not and many developers find it difficult and frustrating to use due to setting up its independent use. An excellent way to incorporate Jester within XP practices for the benefit of confidence in code, as well as making it more attractive for XP developers to use is by integrating it within the ADEPT plug-in and thus also Eclipse. The project aims to do this in a sophisticated way by making use of Eclipse's advanced internal IDE elements and therefore taking advantage of the test testing idea of Jester to the full. This integration of Jester within the Eclipse environment must take into account and maintain the independence of Jester, as replacing older versions by newer more sophisticated versions can then be easily done. This also maintains Jester's open source nature rather than customising it for Eclipse.

1.1 The Jester Plug-In

The main functionality of the Jester plug in can essentially be used by the Jester Launch View to test any JUnit test class (which is defined to be either a class that extends TestCase or has a "suite" method that composes TestCase's to test. The Jester plugin can be run using most of the Eclipse standard ways (i.e. by right clicking a test class and then selecting **Run->Run Jester JUnit Test Tester** and also by manually using the Jester Launch View's browse buttons to enter parameters). When the tool has been run it adds Jester Tasks to the Tasks view allowing the user to double click on them and be transported to the original source file showing the user the location at which Jester made a mutation. A high level Jester task is also added to each Java

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 325–326, 2005.
© Springer-Verlag Berlin Heidelberg 2005

source file Jester was able to make mutations to, without the tests failing. It allows a user to right click and then select “Compare with .jester mutation”. This then opens up a compare dialog between the original source file and a source file with the same name in the same location, but with a .jester file extension. This is created when Jester has successfully been run which is a copy of the original source but with all the changes applied to it.

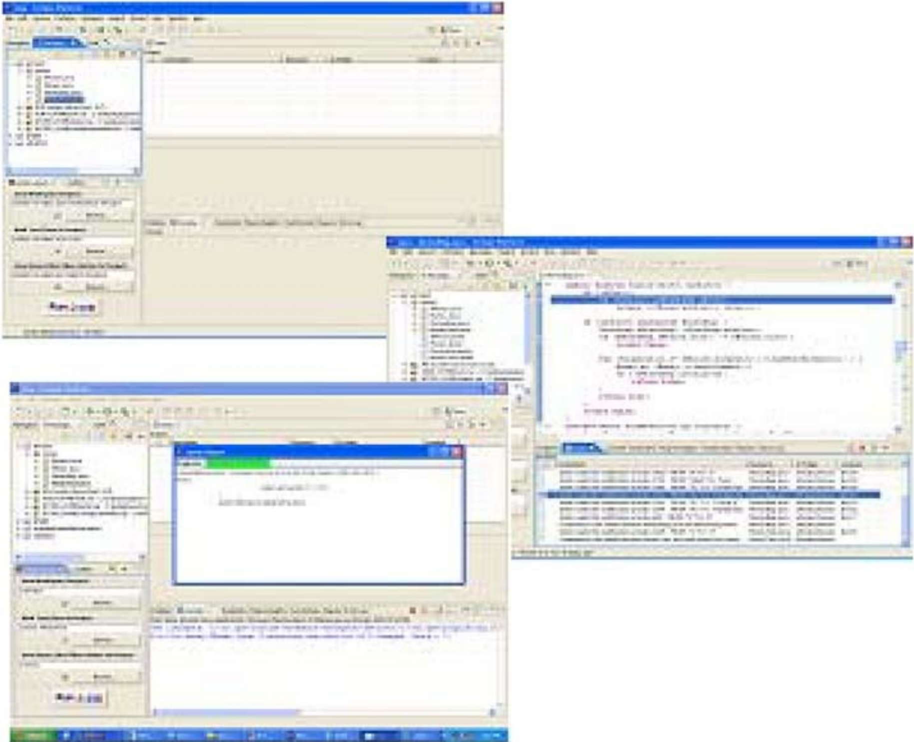


Fig. 1. Jester screen shots

Acknowledgements

Ivan Moore, the inventor of Jester, for help and support

Extreme Programming: The Genesys Experience

Susheel Varma and Mike Holcombe

Genesys Solutions, Department of Computer Science, 211 Regent Court,
University of Sheffield, S1 4DP
M.Holcombe@dcs.shef.ac.uk
http://www.genesys.shef.ac.uk/

Abstract. This paper attempts to describe the author's experience of working in an XP company, namely Genesys Solutions® and its use of the XP principles. Before understanding how XP is being used in Genesys, the process and the people of the organization must be taken into account. The process used is a formalized XP approach and the people considered are students, since the company is entirely run by students, which is unique to Sheffield. The paper also highlights some of the problems and challenges faced by the practitioners at Genesys Solutions® and also some of the real world advantages of using the XP approach in an academic and industrial background.

1 Introduction

Genesys Solutions® is a company of many contradictions. It is an XP company, and has for many years been pioneering the use of mature and formalized XP practices for software development, and yet it has not. It is an informal yet commercial establishment and the company is run entirely by the students of the Department of Computer Science at the University of Sheffield and yet they are professionals in their own right. Every academic year the current students manage every single aspect of the company, from project acquisition to project delivery and at the end of which they are assessed. The main idea is to provide an experience to the students on how a real world software development company is being run and managed which cannot be provided by reading textbooks and attending lectures alone. The XP practitioners at Genesys have been developing core business applications for clients and stake holders, and business solutions to enhance the use of XP both within and outside their company. The company has been involved in more than 50 projects and has had notably many successes and a few failures over the past years but stands in good stead for the years to come. Some of the major successes were a project for *skatesmart.com* which resulted in a new company being set up to market and develop the product further – it employs several of the student team involved – and an IBM funded Eclipse plug-in called ADEPT (**A**gile **D**evelopment **E**nvironment for **P**rogramming and **T**esting) which is a toolset aimed at streamlining the XP process. We have had some failures, however, one when the clients, a consortium of 8 public authorities and NGOs fell apart due to political/organisational reasons. The student team did a magnificent job trying to keep the project together but ultimately failed – they had a valuable learning experience and it demonstrates that political issues need to be addressed in any development methodology.

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 327–328, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Most Genesys projects succeeded without an **On Site Customer**, showing that the practice although desirable is not mandatory for success. There are also many significant examples of success showing that **Small Releases** is a valuable discipline but is more important to maintain release-quality code at all times. The **Planning Game** activity was very difficult to implement as estimates are often wrong, it also shows that customer interaction at this stage is desirable but not entirely necessary and has hindered progress in some projects. A **Simple Design** and **Test First** have been used as an inter-related activity and have shown that using test first has allowed developers to produce simple designs. Although studies and Kent Beck's new book downplays the use of **Metaphors**, it has been extremely useful in the form of X-machines and has given greater visibility for the developers into the solution. X-machines have also helped developers produce simple and testable designs. There has been substantial evidence to show that **Testing** was very difficult to implement as the students were relatively new to its concept, but agrees that testing is extremely desirable. It is hoped that the Automation of testing can increase the progress of the project. **Refactoring** was seen to be very important to maintain a simple design, but a more systematic approach was needed. **Continuous Integration** was also extremely important if the practices of small releases and Refactoring were being followed. **Pair Programming** was extremely useful to all the developers but some problems were noted mainly due to the individualistic nature of some developers. It was one of the most difficult practices to adopt and placed an immense responsibility on the organization to augment, due to the nature of the company. **Collective Ownership** was also an immense task that is extremely desirable and needed active support and is now being looked into by the company. The **40-hour week** (in our case 15 hours) or rhythmicity was also important to the company as it allowed students to maintain discipline and allowed them to reflect on their work. **Coding Standards** is important if pair programming and collective ownership is being followed. It is extremely beneficial to the company and allows easier maintenance and rapid prototyping.

Two significant hidden values of XP were also discovered, namely **respect** and **trust** which are very important to maintain a concrete relationship between the other values of **communication**, **simplicity**, **feedback** and **courage**. The hidden values are actually very important, and were observed to sustain the pair programming/planning activities. Two other practices observed in Genesys apart from the 12 were a **less bureaucratic approach** leading to a humane approach to software engineering and **early adoption of innovation** allowing the company to continuously improve.

Although XP is seen to be beneficial to small and medium sized company's, it must be noted that XP's de-emphasis on analysis and design in the very beginning means that expensive Refactoring must be considered when problems in design are discovered. It is compensated by tightly coupling the practices together; hence a lot of organizational support is required to use a tailored XP.

Shared Code Repository: A Narrative

Susheel Varma and Mike Holcombe

[1] Genesys Solutions, Department of Computer Science, 211 Regent Court,
University of Sheffield, S1 4DP
M.Holcombe@dcs.shef.ac.uk
http://www.genesys.shef.ac.uk/

Abstract. Software Developers face significant challenges including increasing software and development environment complexity, time/economic pressures and high expectations on software quality. This paper discusses a possible collaborative framework to establish and maintain the integrity of software products and processes throughout the development cycle and beyond. It is hoped that by protecting and managing these assets effectively, it enables development teams to produce higher-quality software, faster and with lower cost and risk. The toolset will go beyond version control and provide advanced capabilities that enable effective parallel development, reduced release/build cycles, promote ruse and reduce risk.

1 Introduction

Collaborative Software development is an increasingly complex and dynamic activity. Increased economic/time pressures, project/process management, increased focus on auditory and regulatory compliance further intensify this challenge. It has been observed in Genesys that development teams frequently perform concurrent development on the same or similar applications, with less emphasis on collaboration. This leads to many problems such as (1) Previous projects are impossible to find or cannot be rebuilt; (2) Files change or disappear mysteriously; (3) Corrected bugs reappearing; and many others. In Genesys Solutions the entire workforce changes every year and hence the teams do not have a chance to collaborate except within a 10 month period. This places an immense responsibility on the organization as a whole to support and augment collaboration. The shared code repository will help in simplifying and managing the entire development process by not only providing capabilities such as process control, release/build management, defect/change tracking and version control but also provide a set of tools to collaborate and communicate effectively and to act as a central source of standardized products, expertise, and best practices for developing, testing, deploying and optimizing new applications.

2 Proposed Collaboratory

Genesys has begun the process for implementing an integrated or integrate-able tool that can simplify and manage change, integrate and manage all software products and processes, support the entire development teams, easily integrate with existing sys-

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 329–330, 2005.
© Springer-Verlag Berlin Heidelberg 2005

tems and IDEs, adapt quickly to business and customer needs, and deliver quick business results.

It should also provide the entire organization with the visibility into the solution in its current state, visibility into the process used to create the solution and visibility into the quality and performance parameters of the solution helping to keep everyone informed and keep the solution aligned with the business objectives of the organization and the clients. This collaborative vision has numerous advantages over the current approach: (a) **Increased productivity** (b) **Increased operational efficiency** (c) **Reduced cost**: (d) **Faster delivery times**: **Improved customer retention**: (e) **Improved ability to meet compliance requirements**: (f) **Continuous improvement; and** (f) **Better software quality**.

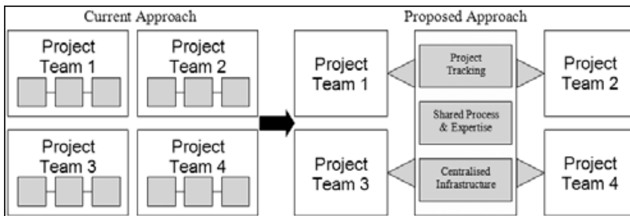


Fig. 1. A Shared Collaborative Framework

Our mission is **to help developers write great software** while **staying out of the way** and **to help reduce the learning curve** for future developers. All aspects of the proposed system would be designed with two main goals in mind: **to simplify tracking and communication of software issues, enhancements and monitoring the overall software progress** and **to provide a central repository of standardized products and processes** to help in the development process. The proposed toolset will go beyond version control and provide capabilities that accelerate productivity and adapt quickly to changing demands. The toolset will have the following features: (a) Story Card/Metaphor Viewer (b) Source Code Browser (c) Changeset Viewer (d) Issue/Bug Tracker (e) Custom Reporter (f) RoadMap/Timeline (g) Notification (h) Administration and Logging.

The shared code repository will then be able to pull out the required information automatically or manually. The automated version could have a possible xml file attached to the source code or the project directory. A possible example DTD is shown below:

```
<?xml version="1.0" encoding="ISO-8859-1">
<!DOCTYPE PROJECT [
   <!ELEMENT PROJECT (NAME,PROJPATH,SCID,TESTID,
NOTES,AUTHORS,DATE,FUNCTIONS+)>
   <!ELEMENT NAME (#PCDATA)>…
(NAME,CODETYPE,PRIORITY,DIFFICULTY,KEYWORDS<INPUTS,OUTPUTS,ME
MORY,DESCRIPTION)>…
]>
```

Such an xml file can then be validated, parsed and information extracted for inclusion into the repository. The code could also be validated for compliance with coding standards.

Author Index

Abrahamsson, Pekka 189
Aceves Gutiérrez, Luis Carlos 206
Adams, Mack 267
Adams, Paul 321
Ally, Mustafa 82
Andersson, Johan 210
Aveling, Ben 235

Bache, Geoff 210, 294, 296
Bacon, Tim 292, 303
Beck, Kent 201, 276, 287
Bojarski, Jacek 247
Boldyreff, Cornelia 321
Bossavit, Laurent 288, 290
Braithwaite, Keith 180
Bustard, David 251

Canfora, Gerardo 92
Canseco Castro, Enrique Sebastián 206
Cau, Alessandra 48, 317
Chilley, Carl 267
Chivers, Howard 57
Cimitile, Aniello 92
Concas, Giulio 48
Cookson, Ammon 1
Cowling, Tony 218, 222
Cunningham, Ward 137

Darroch, Fiona 82
Davies, Rachel 263, 292
Dubinsky, Yael 19, 74
Dwolatzky, Barry 259

Eckstein, Jutta 200

Favaro, John 199
Flaxel, Amy 1
Fraser, Steven 263, 267
Freeman, Steve 276
Freire da Silva, Alexandre 10

Gaillot, Emmanuel 288, 290
Ge, Xiaocheng 57
Geras, Adam 109, 239
Gheorge, Marian 218

Hazzan, Orit 19, 74
Holcombe, Mike 214, 218, 222, 255, 263, 323, 327, 329
Hoover, Dave 303
Hussman, David 267, 305

Ipate, Florentin 214

Joyce, Tim 180

Kalra, Bhavnidhi 255, 323
Karn, John 218, 222
Keenan, Frank 251
Kelly, Steven 311
Keränen, Heikki 189
Keren, Arie 19
Kolovos, Dimitrios S. 226
Kon, Fábio 10
Kushmerick, Nicholas 162
Kwan, Andrew 145

Lafontan, Olivier 274
Larsen, Diana 281
Lever, Simon 325
Love, James 109

Martin, Alan 145
Martin, Angela 263, 267
Maurer, Frank 127
McCarey, Frank 162
McMunn, Dave 28
Melis, Marco 48
Melnik, Grigori 127
Middleton, Peter 1
Miller, James 109, 145, 239
Mlotshwa, Sifiso 259
Mnkandla, Ernest 259, 315
Moore, Ivan 274
Mugridge, Rick 137, 263, 278, 294, 296

Nawrocki, Jerzy 230
Nielsen, Jeff 28
Nolan, John 276
Nowakowski, Wiktor 247

Ó Cinnéide, Mel 162
Olek, Łukasz 230

Paige, Richard F. 57, 226
Passoja, Ulla 171
Peeters, Vera 274, 308
Pierce, Duncan 263
Pietrzak, Błażej 154
Pikkarainen, Minna 171
Polack, Fiona A.C. 226
Poole, Charlie 285
Poppendieck, Mary 267, 280, 302
Poppendieck, Tom 263, 280, 302
Putman, David L. 305

Ramachandran, Muthu 202
Read, Kris 127
Robinson, Hugh 100
Ruanova Hurtado, Mauricio 206

Sandberg, Jan-Erik 299
Schrier, Peter 308
Schwaber, Ken 277
Scotland, Karl 273
Sharp, Helen 100
Sillitti, Alberto 243
Simons, Anthony J.H. 118
Skår, Lars Arne 299
Śmiałek, Michał 38, 247
Smith, Michael 109, 145, 239
Sommerville, Ian 198
Straszak, Tomasz 247
Striebeck, Mark 267
Succi, Giancarlo 243, 263
Swan, Brian 294, 296
Syed-Abdullah, Sharifah Lailee 218, 222

Tappenden, Andrew 239
Thomson, Chris 323
Tingey, Fred 66, 276
Toleman, Mark 82
Torteli, Cicero 10
Turnu, Ivana 48

Van Cauwenberghe, Pascal 274
Varma, Susheel 327, 329
Verdoes, Claes 210
Visaggio, Corrado Aaron 92

Walter, Bartosz 154
Wilking, Dirk 319
Wills, Alan Cameron 311

Yu, Jiang 239