

Security testing incorporates a series of tests designed to exploit vulnerabilities in the WebApp and its environment. The intent is to demonstrate that a security breach is possible.

Performance testing encompasses a series of tests that are designed to assess (1) how WebApp response time and reliability are affected by increased user traffic, (2) which WebApp components are responsible for performance degradation and what usage characteristics cause degradation to occur, and (3) how performance degradation impacts overall WebApp objectives and requirements.

TASK SET



WebApp Testing

1. Review stakeholder requirements. Identify key user goals and objectives. Review use-cases for each user category.
2. Establish priorities to ensure that each user goal and objective will be adequately tested.
3. Define WebApp testing strategy by describing the types of tests (Section 20.2) that will be conducted.
4. Develop a test plan.
 - Define a test schedule and assign responsibilities for each test.
 - Specify automated tools for testing.
 - Define acceptance criteria for each class of test.
 - Specify defect tracking mechanisms.
 - Define problem reporting mechanisms.
5. Perform "unit" tests.
 - Review content for syntax and semantics errors.
 - Review content for proper clearances and permissions.
 - Test interface mechanisms for correct operation.
 - Test each component (e.g., script) to ensure proper function.
6. Perform "integration" tests.
 - Test interface semantics against use-cases.
 - Conduct navigation tests.
7. Perform configuration tests.
 - Assess client-side configuration compatibility.
 - Assess server-side configurations.
8. Conduct performance tests.
9. Conduct security tests.

20.3 CONTENT TESTING



Although formal technical reviews are not a part of testing, content review should be performed to ensure that content has quality.

Errors in WebApp content can be as trivial as minor typographical errors or as significant as incorrect information, improper organization, or violation of intellectual property laws. *Content testing* attempts to uncover these and many other problems before the user encounters them.

Content testing combines both reviews and the generation of executable test cases. Review is applied to uncover semantic errors in content (discussed in Section 20.3.1). Executable testing is used to uncover content errors that can be traced to dynamically derived content driven by data acquired from one or more databases.

20.3.1 Content Testing Objectives

Content testing has three important objectives: (1) to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and

KEY POINT

Content testing objectives are (1) to uncover syntactic errors in content, (2) to uncover semantic errors, (3) to find structural errors.

What questions should be asked and answered to uncover semantic errors in content?

other media, (2) to uncover semantic errors (i.e., errors in the accuracy or completeness of information) in any content object presented as navigation occurs, and (3) to find errors in the organization or structure of content that is presented to the end-user.

To accomplish the first objective, automated spelling and grammar checkers may be used. However, many syntactic errors evade detection by such tools and must be discovered by a human reviewer (tester). As we noted in the preceding section, copy editing is the single best approach for finding syntactic errors.

Semantic testing focuses on the information presented within each content object. The reviewer (tester) must answer the following questions:

- Is the information factually accurate?
- Is the information concise and to the point?
- Is the layout of the content object easy for the user to understand?
- Can information embedded within a content object be found easily?
- Have proper references been provided for all information derived from other sources?
- Is the information presented consistent internally and consistent with information presented in other content objects?
- Is the content offensive, misleading, or does it open the door to litigation?
- Does the content infringe on existing copyrights or trademarks?
- Does the content contain internal links that supplement existing content? Are the links correct?
- Does the aesthetic style of the content conflict with the aesthetic style of the interface?

Obtaining answers to each of these questions for a large WebApp (containing hundreds of content objects) can be a daunting task. However, failure to uncover semantic errors will shake the user's faith in the WebApp and can lead to failure of the Web-based application.

Content objects exist within an architecture that has a specific style (Chapter 19). During content testing, the structure and organization of the content architecture is tested to ensure that required content is presented to the end-user in the proper order and relationships. For example, the SafeHomeAssured.com WebApp⁵ presents a variety of information about sensors that are used as part of security and surveillance products. Content objects provide descriptive information, technical specifications, a photographic representation and related information. Tests of the SafeHomeAssured.com content architecture strive to uncover errors in the presentation of this information (e.g., a description of Sensor X is presented with a photo of Sensor Y).

⁵ The SafeHomeAssured.com WebApp has been used as an example throughout Part 3 of this book.

20.3.2 Database Testing

Modern Web applications do much more than present static content objects. In many application domains, WebApps interface with sophisticated database management systems and build dynamic content objects that are created in real-time using the data acquired from a database.

For example, a financial services WebApp can produce complex text-based, tabular, and graphical information about a specific equity (e.g., a stock or mutual fund). The composite content object that presents this information is created dynamically after the user has made a request for information about a specific equity. To accomplish this, the following steps are required: (1) a large equities database is queried, (2) relevant data are extracted from the database, (3) the extracted data must be organized as a content object, and (4) this content object (representing customized information requested by an end-user) is transmitted to the client environment for display. Errors can and do occur as a consequence of each of these steps. The objective of database testing is to uncover these errors.

Database testing for WebApps is complicated by a variety of factors:

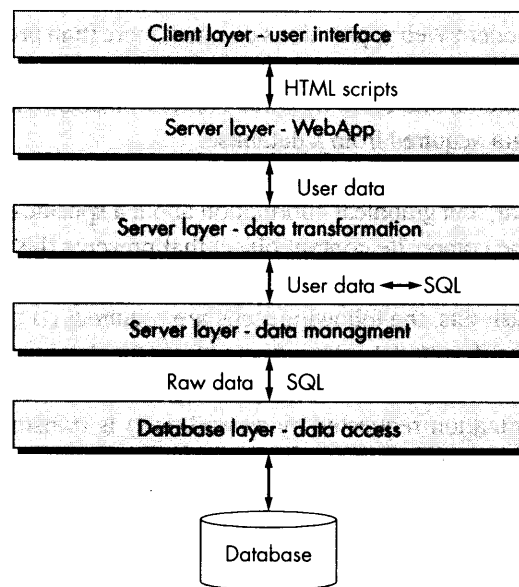
What issues complicate database testing for WebApps?

1. *The original client-side request for information is rarely presented in the form (e.g., structured query language, SQL) that can be input to a database management system (DBMS). Therefore, tests should be designed to uncover errors made in translating the user's request into a form that can be processed by these DBMS.*
2. *The database may be remote to the server that houses the WebApp. Therefore, tests that uncover errors in communication between the WebApp and the remote database should be developed.⁶*
3. *Raw data acquired from the database must be transmitted to the WebApp server and properly formatted for subsequent transmittal to the client. Therefore, tests that demonstrate the validity of the raw data received by the WebApp server should be developed, and additional tests that demonstrate the validity of the transformations applied to the raw data to create valid content objects must also be created.*
4. *The dynamic content object(s) must be transmitted to the client in a form that can be displayed to the end-user. Therefore, a series of tests should be designed to (a) uncover errors in the content object format, and (b) test compatibility with different client environment configurations.*

Considering these four factors, test case design methods should be applied for each of the "layers of interaction" [NGU01] noted in Figure 20.2. Testing should ensure that (1) valid information is passed between the client and server from the interface layer;

⁶ These tests can become complex when distributed databases are encountered or when access to a data warehouse (Chapter 10) is required.

FIGURE 20.2
Layers of interaction



(2) the WebApp processes scripts correctly and properly extracts or formats user data; (3) user data are passed correctly to a server side data transformation function that formats appropriate queries (e.g., SQL); (4) queries are passed to a data management layer⁷ that communicates with database access routines (potentially located on another machine).

Data transformation, data management, and database access layers shown in Figure 20.2 are often constructed with reusable components that have been validated separately and as a package. If this is the case, WebApp testing focuses on the design of test cases to exercise the interactions between the client layer and the first two server layers (WebApp and data transformation) shown in the figure.

The user interface layer is tested to ensure that HTML scripts are properly constructed for each user query and properly transmitted to the server side. The WebApp layer on the server side is tested to ensure that user data are properly extracted from HTML scripts and properly transmitted to the data transformation layer on the server side.

The data transformation functions are tested to ensure that correct SQL is created and passed to appropriate data management components.

A detailed discussion of the underlying technology that must be understood to adequately design these database tests is beyond the scope of this book. The interested reader should see [SCE02], [NGU01], and [BRO01].

⁷ The data management layer typically incorporates an SQL call-level interface (SQL-CLI) such as Microsoft OLE/ADO or Java Database Connectivity (JDBC).

"As e-customers (whether business or consumer), we are unlikely to have confidence in a Web site that suffers frequent downtime, hangs in the middle of a transaction, or has a poor sense of usability. Testing, therefore, has a crucial role in the overall development process."

Wing Lam

20.4 USER INTERFACE TESTING

Verification and validation of a WebApp user interface occurs at three distinct points in the Web engineering process. During formulation and requirements analysis (Chapters 17 and 18), the interface model is reviewed to ensure that it conforms to customer requirements and to other elements of the analysis model. During design (Chapter 19), the interface design model is reviewed to ensure that generic quality criteria established for all user interfaces have been achieved and that application-specific interface design issues have been properly addressed. During testing, the focus shifts to the execution of application-specific aspects of user interaction as they are manifested by interface syntax and semantics. In addition, testing provides a final assessment of usability.

20.4.1 Interface Testing Strategy

The overall strategy for interface testing is to (1) uncover errors related to specific interface mechanisms (e.g., errors in the proper execution of a menu link or the way data are entered in a form) and (2) uncover errors in the way the interface implements the semantics of navigation, WebApp functionality, or content display. To accomplish this strategy, a number of objectives must be achieved:



With the exception of WebApp-oriented specifics, the interface strategy noted here is applicable to all types of client/server software.

- *Interface features are tested to ensure that design rules, aesthetics, and related visual content are available for the user without error.* Features include type fonts, the use of color, frames, images, borders, tables, and related elements that are generated as WebApp execution proceeds.
- *Individual interface mechanisms are tested in a manner that is analogous to unit testing.* For example, tests are designed to exercise all forms, client-side scripting, dynamic HTML, CGI scripts, streaming content, and application specific interface mechanisms (e.g., a shopping cart for an e-commerce application). In many cases, testing can focus exclusively on one of these mechanisms (the "unit") to the exclusion of other interface features and functions.
- *Each interface mechanism is tested within the context of a use-case or NSU (Chapter 19) for a specific user category.* This testing approach is analogous to integration testing (Chapter 13) in that tests are conducted as interface mechanisms are integrated to allow a use-case or NSU to be executed.
- *The complete interface is tested against selected use-cases and NSUs to uncover errors in the semantics of the interface.* This testing approach is analogous to

validation testing (Chapter 13) because the purpose is to demonstrate conformance to specific use-case or NSU semantics. It is at this stage that a series of usability tests are conducted.

- *The interface is tested within a variety of environments (e.g., browsers) to ensure that it will be compatible.* In actuality, this series of tests can also be considered to be part of configuration testing.

20.4.2 Testing Interface Mechanisms

When a user interacts with a WebApp, the interaction occurs through one or more interface mechanisms. In the paragraphs that follow, we present a brief overview of testing considerations for each interface mechanism [SPL01].



External link testing should occur throughout the life of the WebApp. Part of a support strategy should be regularly scheduled link tests.

Links. Each navigation link is tested to ensure that the proper content object or function is reached.⁸ The Web engineer builds a list of all links associated with the interface layout (e.g., menu bars, index items) and then executes each individually. In addition, links within each content object must be exercised to uncover bad URLs or links to improper content objects or functions. Finally, links to external WebApps should be tested for accuracy and also evaluated to determine the risk that they will become invalid over time.

Forms. At a macroscopic level, tests are performed to ensure that (1) labels correctly identify fields within the form and that mandatory fields are identified visually for the user; (2) the server receives all information contained within the form and that no data are lost in the transmission between client and server; (3) appropriate defaults are used when the user does not select from a pull-down menu or set of buttons; (4) browser functions (e.g., the “back” arrow) do not corrupt data entered in a form; and (5) scripts that perform error checking on data entered work properly and provide meaningful error messages.

At a more targeted level, tests should ensure that (1) form fields have proper width and data types; (2) the form establishes appropriate safeguards that preclude the user from entering text strings longer than some predefined maximum; (3) all appropriate options for pull-down menus are specified and ordered in a way that is meaningful to the end-user; (4) browser “auto-fill” features do not lead to data input errors; and (5) the tab key (or some other key) initiates proper movement between form fields.



Client-side scripting tests and tests associated with dynamic HTML should be repeated whenever a new version of a popular browser is released.

Client-side scripting. Black-box tests are conducted to uncover any errors in processing as the script (e.g., Javascript) is executed. These tests are often coupled with forms testing, because script input is often derived from data provided as part of forms processing. A compatibility test should be conducted to ensure that the scripting language that has been chosen will work properly in the environmental configuration(s) that supports the WebApp. In addition to testing the script itself, Splaine

⁸ These tests can be performed as part of either interface or navigation testing.

and Jaskiel [SPL01] suggest that “you should ensure that your company’s [WebApp] standards state the preferred language and version of scripting language to be used for client-side (and server-side) scripting.”

Dynamic HTML. Each Web page that contains dynamic HTML is executed to ensure that the dynamic display is correct. In addition, a compatibility test should be conducted to ensure that the dynamic HTML works properly in the environmental configuration(s) that supports the WebApp.

Pop-up windows.⁹ A series of tests ensure that (1) the pop-up is properly sized and positioned; (2) the pop-up does not cover the original WebApp window; (3) the aesthetic design of the pop-up is consistent with the aesthetic design of the interface; and (4) scroll bars and other control mechanisms appended to the pop-up work, are properly located, and function as required.

CGI scripts. Black-box tests are conducted with an emphasis on data integrity (as data are passed to the CGI script) and script processing once validated data have been received. In addition, performance testing can be conducted to ensure that the server-side configuration can accommodate the processing demands of multiple invocations of CGI scripts [SPL01].

Streaming content. Tests should demonstrate that streaming data are up-to-date, properly displayed, and can be suspended without error and restarted without difficulty.

Cookies. Both server-side and client-side testing are required. On the server side, tests should ensure that a cookie is properly constructed (contains correct data) and properly transmitted to the client side when specific content or functionality is requested. In addition, the proper persistence of the cookie is tested to ensure that its expiration date is correct. On the client side, tests determine whether the WebApp properly attaches existing cookies to a specific request (sent to the server).

Application specific interface mechanisms. Tests conform to a checklist of functionality and features that are defined by the interface mechanism. For example, Splaine and Jaskiel [SPL01] suggest the following checklist for shopping cart functionality defined for an e-commerce application:

- Boundary test (Chapter 14) the minimum and maximum number of items that can be placed in the shopping cart.
- Test a “check out” request for an empty shopping cart.
- Test proper deletion of an item from the shopping cart.
- Test to determine whether a purchase empties the cart of its contents.

⁹ Pop-ups have become pervasive and are a major irritant to many users. They should be used judiciously or not at all.

- Test to determine the persistence of shopping cart contents (this should be specified as part of customer requirements).
- Test to determine whether the WebApp can recall shopping cart contents at some future date (assuming that no purchase was made) if the user requests that contents be saved.

20.4.3 Testing Interface Semantics

Once each interface mechanism has been “unit” tested, the focus of interface testing changes to a consideration of interface semantics. Interface semantics testing “evaluates how well the design takes care of users, offers clear direction, delivers feedback, and maintains consistency of language and approach” [NGU01].

A thorough review of the interface design model can provide partial answers to the questions implied by the preceding paragraph. However, each use-case scenario (for each user category) should be tested once the WebApp has been implemented. In essence, a use-case becomes the input for the design of a testing sequence. The intent of the testing sequence is to uncover errors that will preclude a user from achieving the objective associated with the use-case.

As each use-case is tested, the Web engineering team maintains a checklist to ensure that every menu item has been exercised at least one time and that every embedded link within a content object has been used. In addition, the test sequence should include improper menu selection and link usage. The intent is to determine whether the WebApp provides effective error handling and recovery.

20.4.4 Usability Tests

Usability testing is similar to interface semantics testing (Section 20.4.3) in the sense that it also evaluates the degree to which users can interact effectively with the WebApp and the degree to which the WebApp guides users’ actions, provides meaningful feedback, and enforces a consistent interaction approach. Rather than focusing intently on the semantics of some interactive objective, usability reviews and tests are designed to determine the degree to which the WebApp interface makes the user’s life easy.¹⁰

Usability tests may be designed by a Web engineering team, but the tests themselves are conducted by end-users. The following sequence of steps is applied [SPL01]:

1. Define a set of usability testing categories and identify goals for each.
2. Design tests that will enable each goal to be evaluated.

WebRef

A worthwhile guide to usability testing can be found at www.dhref.com/guides/design/199806/0615jef.html.

¹⁰ The term “user-friendliness” has been used in this context. The problem, of course, is that one user’s perception of a “friendly” interface may be radically different from another’s.

3. Select participants who will conduct the tests.
4. Instrument participants' interaction with the WebApp while testing is conducted.
5. Develop a mechanism for assessing the usability of the WebApp.

Usability testing can occur at a variety of different levels of abstraction: (1) the usability of a specific interface mechanism (e.g., a form) can be assessed; (2) the usability of a complete Web page (encompassing interface mechanisms, data objects, and related functions) can be evaluated; or (3) the usability of the complete WebApp can be considered.

The first step in usability testing is to identify a set of usability categories and establish testing objectives for each category. The following test categories and objectives (written in the form of a question) illustrate this approach:¹¹

What characteristics of usability become the focus of testing, and what specific objectives are addressed?

Interactivity—Are interaction mechanisms (e.g., pull-down menus, buttons, pointers) easy to understand and use?

Layout—Are navigation mechanisms, content, and functions placed in a manner that allows the user to find them quickly?

Readability—Is text well-written and understandable?¹² Are graphic representations easy to understand?

Aesthetics—Do layout, color, typeface, and related characteristics lead to ease of use? Do users “feel comfortable” with the look and feel of the WebApp?

Display characteristics—Does the WebApp make optimal use of screen size and resolution?

Time sensitivity—Can important features, functions, and content be used or acquired in a timely manner?

Personalization—Does the WebApp tailor itself to the specific needs of different user categories or individual users?

Accessibility—Is the WebApp accessible to people who have disabilities?

Within each of these categories, a series of tests is designed. In some cases, the “test” may be a visual review of a Web page. In other cases interface semantics tests may be executed again, but in this instance usability concerns are paramount.

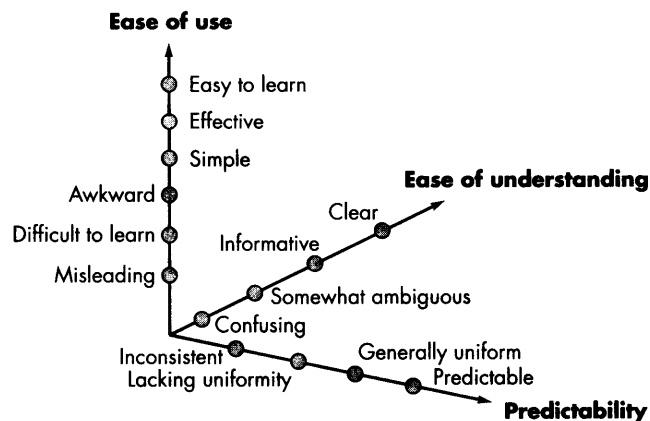
As an example, we consider usability assessment for interaction and interface mechanisms. Constantine and Lockwood [CON03] suggest that the following list of interface features should be reviewed and tested for usability: animation, buttons, color, control, dialogue, fields, forms, frames, graphics, labels, links, menus,

¹¹ For additional usability questions, see the “usability” sidebar in Chapter 12.

¹² The FOG Readability Index and others may be used to provide a quantitative assessment of readability. See <http://developer.gnome.org/documents/usability/usability-readability.html> for more details.

FIGURE 20.3

Qualitative assessment of usability



messages, navigation, pages, selectors, text, and tool bars. As each feature is assessed, it is graded on a qualitative scale by the users who are doing the testing. Figure 20.3 depicts a possible set of assessment “grades” that can be selected by users. These grades are applied to each feature individually, to a complete Web page, or to the WebApp as a whole.

20.4.5 Compatibility Tests

WebApps must operate within environments that differ from one another. Different computers, display devices, operating systems, browsers, and network connection speeds can have a significant influence on WebApp operation. Each computing configuration can result in differences in client-side processing speeds, display resolution, and connection speeds. Operating system vagaries may cause WebApp processing issues. Different browsers sometimes produce slightly different results, regardless of the degree of HTML standardization within the WebApp. Required plug-ins may or may not be readily available for a particular configuration.

In some cases, small compatibility issues present no significant problems, but in others, serious errors can be encountered. For example, download speeds may become unacceptable, lack of a required plug-in may make content unavailable, browser differences can change page layout dramatically, font styles may be altered and become illegible, or forms may be improperly organized. *Compatibility testing* strives to uncover these problems before the WebApp goes on-line.

The first step in compatibility testing is to define a set of “commonly encountered” client-side computing configurations and their variants. In essence, a tree structure is created, identifying each computing platform, typical display devices, the operating systems supported on the platform, the browsers available, likely Internet connection speeds, and similar information. Next, the Web engineering team derives a series of compatibility validation tests, derived from existing interface tests, naviga-

KEY POINT

WebApps execute within a variety of client-side environments. The objective of compatibility testing is to uncover errors associated with a specific environment (e.g., browser).

tion tests, performance tests, and security tests. The intent of these tests is to uncover errors or execution problems that can be traced to configuration differences.

SAFEHOME



WebApp Testing

The scene: Doug Miller's office.

Doug: Hey, Doug Miller (manager of the *SafeHome* software engineering group) and Vinod Raman, a member of the product software engineering team.

The participants:

Doug: What do you think of the *SafeHomeAssured.com* e-commerce WebApp V0.0?

Vinod: The outsourcing vendor's done a good job.

Doug: [Development manager for the vendor] tells me you're looking as we speak.

Doug: I'd like you and the rest of the team to do a little bit of testing on the e-commerce site.

Vinod: Good thing. I thought we were going to hire a professional testing company to validate the WebApp. We're still killing ourselves trying to get the product out the door.

Doug: We're going to hire a testing vendor for functional and security testing, and our outsourcing vendor is doing usability testing. Just thought another point of view would be helpful, and besides, we'd like to keep our budget down.

Vinod: What are you looking for?

Doug: I want to be sure that the interface and all

the processes we can start with the use-cases for the e-commerce interface functions:

Learn about *SafeHome*

Specify the *SafeHome* system you need

Purchase a *SafeHome* system

Get technical support

Doug: Good. But take the navigation paths all the way to their conclusion.

Vinod (looking through a notebook of use-cases): Yeah, when you select **Specify the *SafeHome* system you need**, that'll take you to

Select *SafeHome* components

Get *SafeHome* component recommendations

We can exercise the semantics of each path.

Doug: While you're there, check out the content that appears at each navigation node.

Vinod: Of course . . . and the functional elements as well. Who's testing usability?

Doug: Oh . . . the testing vendor will cover the usability testing. We've hired a usability vendor to line up 20 typical users for the usability testing. If the guys uncover any usability issues

Vinod: I know, pass them along.

Doug: Thanks, Vinod.

20.5 COMPONENT-LEVEL TESTING

Component-level testing, also called *function testing*, focuses on a set of tests that attempt to uncover errors in WebApp functions. Each WebApp function is a software module (implemented in one of a variety of programming or scripting languages) and can be tested using black-box (and, in some cases, white-box) techniques discussed in Chapter 14.

Component-level test cases are often driven by forms-level input. Once forms data are defined, the user selects a button or other control mechanism to initiate execution. The following test case design methods (Chapter 14) are typical:

- *Equivalence partitioning*—The input domain of the function is divided into input categories or classes from which test cases are derived. The input form is assessed to determine what classes of data are relevant for the function. Test cases for each class of input are derived and executed while other classes of input are held constant. For example, an e-commerce application may implement a function that computes shipping charges. Among a variety of shipping information provided via a form is the user's postal code. Test cases are designed in an attempt to uncover errors in postal code processing by specifying postal code values that might uncover different classes of errors (e.g., an incomplete postal code, a correct postal code, a nonexistent postal code, an erroneous postal code format).
- *Boundary value analysis*—Forms data are tested at their boundaries. For example, the shipping calculation function noted previously requests the maximum number of days required for product delivery. A minimum of 2 days and a maximum of 14 are noted on the form. However, boundary value tests might input values of 0; 1, 2, 13, 14, and 15 to determine how the function reacts to data at and outside the boundaries of valid input.¹³
- *Path testing*—If the logical complexity of the function is high,¹⁴ path testing (a white-box test case design method) can be used to ensure that every independent path in the program has been exercised.

In addition to these test case design methods, a technique called *forced error testing* [NGU01] is used to derive test cases that purposely drive the WebApp component into an error condition. The purpose is to uncover errors that occur during error-handling (e.g., incorrect or nonexistent error messages, WebApp failure as a consequence of the error, erroneous output driven by erroneous input, side-effects that are related to component processing).

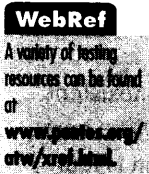
Each component-level test case specifies all input values and the expected output to be provided by the component. The actual output produced as a consequence of the test is recorded for future reference during support and maintenance.

In many situations, the correct execution of a WebApp function is tied to proper interfacing with a database that may be external to the WebApp. Therefore, database testing becomes an integral part of the component-testing regime. Hower [HOW97] discusses this when he writes:

Database-driven Web sites can involve a complex interaction among Web browsers, operating systems, plug-in applications, communications protocols, Web servers, databases, [scripting language] programs . . . , security enhancements, and firewalls. Such

¹³ In this case, a better input design might eliminate potential errors. The maximum number of days could be selected from a pull-down menu, precluding the user from specifying out-of-bounds input.

¹⁴ Logical complexity can be determined by computing cyclomatic complexity of the algorithm. See Chapter 14 for additional details.



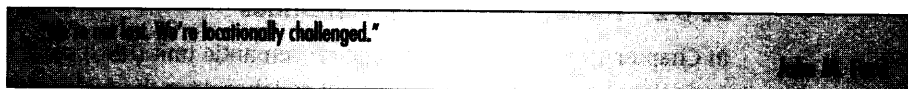
complexity makes it impossible to test every possible dependency and everything that could go wrong with a site. The typical Web site development project will also be on an aggressive schedule, so the best testing approach will employ risk analysis to determine where to focus testing efforts. Risk analysis should include consideration of how closely the test environment will match the real production environment. . . . Other typical considerations in risk analysis include:

- Which functionality in the Web site is most critical to its purpose?
- Which areas of the site require the heaviest database interaction?
- Which aspects of the site's CGI, applets, ActiveX components, and so on are most complex?
- What types of problems would cause the most complaints or the worst publicity?
- What areas of the site will be the most popular?
- What aspects of the site have the highest security risks?

Each of the risk-related issues discussed by Hower should be considered when designing test cases for WebApp components and related database functions.

20.6 NAVIGATION TESTING

A user travels through a WebApp in much the same way as a visitor walks through a store or museum. There are many pathways that can be taken, many stops that can be made, many things to learn and look at, activities to initiate, and decisions to make. As we have already discussed, this navigation process is predictable in the sense that every visitor has a set of objectives when he arrives. At the same time, the navigation process can be unpredictable because the visitor, influenced by something he sees or learns, may choose a path or initiate an action that is not typical for the original objective. The job of navigation testing is (1) to ensure that the mechanisms that allow the WebApp user to travel through the WebApp are all functional and (2) to validate that each navigation semantic unit (NSU) can be achieved by the appropriate user category.



20.6.1 Testing Navigation Syntax

The first phase of navigation testing actually begins during interface testing. Navigation mechanisms are tested to ensure that each performs its intended function. Splaine and Jaskiel [SPL01] suggest that each of the following navigation mechanisms should be tested:

- *Navigation links*—internal links within the WebApp, external links to other WebApps, and anchors within a specific Web page should be tested to

ensure that proper content or functionality is reached when the link is chosen.

- *Redirects*—these links come into play when a user requests a nonexistent URL or selects a link whose destination has been removed or whose name has changed. A message is displayed for the user, and navigation is redirected to another page (e.g., the home page). Redirects should be tested by requesting incorrect internal links or external URLs and assessing how the WebApp handles these requests.
- *Bookmarks*—although bookmarks are a browser function, the WebApp should be tested to ensure that a meaningful page title can be extracted as the bookmark is created.
- *Frames and framesets*—each frame contains the content of a specific Web page; a frameset contains multiple frames and enables the display of multiple Web pages at the same time. Because it is possible to nest frames and framesets within one another, these navigation and display mechanisms should be tested for correct content, proper layout and sizing, download performance, and browser compatibility.
- *Site maps*—entries should be tested to ensure that the link takes the user to the proper content or functionality.
- *Internal search engines*—complex WebApps often contain hundreds or even thousands of content objects. An internal search engine allows the user to perform a keyword search within the WebApp to find needed content. Search engine testing validates the accuracy and completeness of the search, the error-handling properties of the search engine, and advanced search features (e.g., the use of Boolean operators in the search field).

Some of the tests noted can be performed by automated tools (e.g., link checking) while others are designed and executed manually. The intent throughout is to ensure that errors in navigation mechanics are found before the WebApp goes on-line.

20.6.2 Testing Navigation Semantics

In Chapter 19 we defined a navigation semantic unit (NSU) as “a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements” [CAC02]. Each NSU is defined by a set of navigation paths (called “ways of navigating”) that connect navigation nodes (e.g., Web pages, content objects, or functionality). Taken as a whole, each NSU allows a user to achieve specific requirements defined by one or more use-cases for a user category. Navigation testing exercises each NSU to ensure that these requirements can be achieved.

As each NSU is tested, the Web engineering team must answer the following questions:

- Is the NSU achieved in its entirety without error?

- Is every navigation node (defined for a NSU) reachable within the context of the navigation paths defined for the NSU?
- If the NSU can be achieved using more than one navigation path, has every relevant path been tested?
- If guidance is provided by the user interface to assist in navigation, are directions correct and understandable as navigation proceeds?
- Is there a mechanism (other than the browser “back” arrow) for returning to the preceding navigation node and to the beginning of the navigation path?
- Do mechanisms for navigation within a large navigation node (i.e., a long Web page) work properly?
- If a function is to be executed at a node and the user chooses not to provide input, can the remainder of the NSU be completed?
- If a function is executed at a node and an error in function processing occurs, can the NSU be completed?
- Is there a way to discontinue the navigation before all nodes have been reached, but then return to where the navigation was discontinued and proceed from there?
- Is every node reachable from the site map? Are node names meaningful to end-users?
- If a node within a NSU is reached from some external source, is it possible to process to the next node on the navigation path? Is it possible to return to the previous node on the navigation path?
- Does the user understand his location within the content architecture as the NSU is executed?



If NSUs have not been created as part of Web engineering analysis or design, you can apply use-cases for the design of navigation test cases. The same set of questions are asked and answered.

Navigation testing, like interface and usability testing, should be conducted by as many different constituencies as possible. Early stages of testing are conducted by Web engineers, but later stages should be conducted by other project stakeholders, an independent testing team, and ultimately, by nontechnical users. The intent is to exercise WebApp navigation thoroughly.

20.7 CONFIGURATION TESTING

Configuration variability and instability are important factors that make Web engineering a challenge. Hardware, operating system(s), browsers, storage capacity, network communication speeds, and a variety of other client-side factors are difficult to predict for each user. In addition, the configuration for a given user can change (e.g., OS updates, new ISP and connection speeds) on a regular basis. The result can be a client-side environment that is prone to errors that are both subtle and significant. One user’s impression of the WebApp and the manner in which he

interacts with it can differ significantly from another user's experience, if both users are not working within the same client-side configuration.

The job of *configuration testing* is not to exercise every possible client-side configuration. Rather, it is to test a set of probable client-side and server-side configurations to ensure that the user experience will be the same on all of them and to isolate errors that may be specific to a particular configuration.

20.7.1 Server-Side Issues

On the server side, configuration test cases are designed to verify that the projected server configuration (i.e., WebApp server, database server, operating system(s), firewall software, concurrent applications) can support the WebApp without error. In essence, the WebApp is installed within the server-side environment and tested with the intent of finding configuration-related errors.

As server-side configuration tests are designed, the Web engineer should consider each component of the server configuration. Among the questions that need to be asked and answered during server-side configuration testing are:

What questions must be asked and answered as server-side configuration testing is conducted?

- Is the WebApp fully compatible with the server OS?
- Are system files, directories, and related system data created correctly when the WebApp is operational?
- Do system security measures (e.g., firewalls or encryption) allow the WebApp to execute and service users without interference or performance degradation?
- Has the WebApp been tested with the distributed server configuration¹⁵ (if one exists) that has been chosen?
- Is the WebApp properly integrated with database software? Is the WebApp sensitive to different versions of database software?
- Do server-side WebApp scripts execute properly?
- Have system administrator errors been examined for their affect on WebApp operations?
- If proxy servers are used, have differences in their configuration been addressed with on-site testing?

20.7.2 Client-Side Issues

On the client side, configuration tests focus more heavily on WebApp compatibility with configurations that contain one or more permutation of the following components [NGU01]:

¹⁵ For example, a separate application server and database server may be used. Communication between the two machines occurs across a network connection.

- *Hardware*—CPU, memory, storage, and printing devices.
- *Operating systems*—Linux, Macintosh OS, Microsoft Windows, a mobile-based OS.
- *Browser software*—Internet Explorer, Mozilla/Netscape, Opera, Safari, and others.
- *User interface components*—Active X, Java applets, and others.
- *Plug-ins*—QuickTime, RealPlayer, and many others.
- *Connectivity*—cable, DSL, regular modem, T1.

In addition to these components, other variables include networking software, the vagaries of the ISP, and applications running concurrently.

To design client-side configuration tests, the Web engineering team must reduce the number of configuration variables to a manageable number.¹⁶ To accomplish this, each user category is assessed to determine the likely configurations to be encountered within the category. In addition, industry market share data may be used to predict the most likely combinations of components. The WebApp is then tested within these environments.

WebApp security is a complex subject that must be fully understood before effective security testing can be accomplished.¹⁷ WebApps and the client-side and server-side environments in which they are housed represent an attractive target for external hackers, disgruntled employees, dishonest competitors, and anyone else who wishes to steal sensitive information, maliciously modify content, degrade performance, disable functionality, or embarrass a person, organization, or business.

...places where to conduct business or store assets. Hackers, crackers, spammers, and rogue program purveyors run loose."

Security

Security tests are designed to probe vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and the server-side environment. Each of these domains can be attacked, and it is the job of the security tester to uncover weaknesses that can be exploited by those with the intent to do so.

¹⁶ Running tests on every possible combination of configuration components is far too time consuming.

¹⁷ Books by Trivedi [TRE03], McClure and his colleagues [MCC03], and Garfinkel and Spafford [GAR02] provide useful information about the subject.



If the WebApp is business critical, maintains sensitive data, or is a likely target of hackers, it's a good idea to outsource security testing to a vendor who specializes in it.

On the client-side, vulnerabilities can often be traced to pre-existing bugs in browsers, e-mail programs, or communication software. Nguyen [NGU01] describes a typical security hole:

One of the commonly mentioned bugs is Buffer Overflow, which allows malicious code to be executed on the client machine. For example, entering a URL into a browser that is much longer than the buffer size allocated for the URL will cause a memory overwrite (buffer overflow) error if the browser does not have error detection code to validate the length of the input URL. A seasoned hacker can cleverly exploit this bug by writing a long URL with code to be executed that can cause the browser to crash or alter security settings (from high to low), or, at worst, to corrupt user data.

Another potential vulnerability on the client-side is unauthorized access to cookies placed within the browser. Web sites created with malicious intent can acquire information contained within legitimate cookies and use this information in ways that jeopardize the user's privacy, or worse, set the stage for identity theft.

Data communicated between the client and server are vulnerable to *spoofing*. Spoofing occurs when one end of the communication pathway is subverted by an entity with malicious intent. For example, a user can be spoofed by a malicious Web site that acts as if it is the legitimate WebApp server (identical look and feel). The intent is to steal passwords, proprietary information, or credit data.

On the server-side, vulnerabilities include denial-of-service attacks and malicious scripts that can be passed along to the client side or used to disable server operations. In addition, server-side databases can be accessed without authorization (data theft).

To protect against these (and many other) vulnerabilities, one or more of the following security elements is implemented [NGU01]:

- *Firewalls*—a filtering mechanism that is a combination of hardware and software that examines each incoming packet of information to ensure that it is coming from a legitimate source, blocking any data that are suspect.
- *Authentication*—a verification mechanism that validates the identity of all clients and servers, allowing communication to occur only when both sides are verified.
- *Encryption*—an encoding mechanism that protects sensitive data by modifying it in a way that makes it impossible to read by those with malicious intent. Encryption is strengthened by using *digital certificates* that allow the client to verify the destination to which the data are transmitted.
- *Authorization*—a filtering mechanism that allows access to the client or server environment only by those individuals with appropriate authorization codes (e.g., userID and password)



Security tests should be designed to exercise firewalls, authentication, encryption, and authorization.

The intent of security testing is to expose holes in these security elements that can be exploited by those with malicious intent. The actual design of security tests

requires in-depth knowledge of the inner workings of each security element and a comprehensive understanding of a full range of networking technologies. In many cases, security testing is outsourced to firms that specialize in these technologies.

20.9 PERFORMANCE TESTING

Nothing is more frustrating than a WebApp that takes minutes to load content when competitive sites download similar content in seconds. Nothing is more aggravating than trying to log-in to a WebApp and receiving a “server-busy” message, with the suggestion that you try again later. Nothing is more disconcerting than a WebApp that responds instantly in some situations, and then seems to go into an infinite wait-state in other situations. All of these occurrences happen on the Web every day, and all of them are performance-related.

Performance testing is used to uncover performance problems that can result from lack of server-side resources, inappropriate network bandwidth, inadequate database capabilities, faulty or weak operating system capabilities, poorly designed WebApp functionality, and other hardware or software issues that can lead to degraded client-server performance. The intent is twofold: (1) to understand how the system responds to *loading* (i.e., number of users, number of transactions, or overall data volume), and (2) to collect metrics that will lead to design modifications to improve performance.

20.9.1 Performance Testing Objectives

Performance tests are designed to simulate real-world loading situations. As the number of simultaneous WebApp users grows, or the number of on-line transactions increases, or the amount of data (downloaded or uploaded) increases, performance testing will help answer the following questions:



Some aspects of WebApp performance, at least as it is perceived by the end-user, are difficult to test, including network loading, the vagaries of network interfacing hardware, and similar issues.

- Does the server response time degrade to a point where it is noticeable and unacceptable?
- At what point (in terms of users, transactions, or data loading) does performance become unacceptable?
- What system components are responsible for performance degradation?
- What is the average response time for users under a variety of loading conditions?
- Does performance degradation have an impact on system security?
- Is WebApp reliability or accuracy affected as the load on the system grows?
- What happens when loads that are greater than maximum server capacity are applied?

To develop answers to these questions, two different performance tests are conducted:

- *Load testing*—real world loading is tested at a variety of load levels and in a variety of combinations.
- *Stress testing*—loading is increased to the breaking point to determine how much capacity the WebApp environment can handle.

Each of these testing strategies is considered in the sections that follow.

20.9.2 Load Testing

The intent of *load testing* is to determine how the WebApp and its server-side environment will respond to various loading conditions. As testing proceeds, permutations to the following variables define a set of test conditions:

N , the number of concurrent users

T , the number of on-line transactions per user per unit time

D , the data load processed by the server per transaction



If a WebApp uses multiple servers to provide significant capacity, load testing must be performed in a multiserver environment.

In every case, these variables are defined within normal operating bounds of the system. As each test condition is run, one or more of the following measures are collected: average user response, average time to download a standardized unit of data, or average time to process a transaction. The Web engineering team examines these measures to determine whether a precipitous decrease in performance can be traced to a specific combination of N , T , and D .

Load testing can also be used to assess recommended connection speeds for users of the WebApp. Overall throughput, P , is computed in the following manner:

$$P = N \times T \times D$$

As an example, consider a popular sports news site. At a given moment, 20,000 concurrent users submit a request (a transaction, T) once every two minutes on average. Each transaction requires the WebApp to download a new article that averages 3 K bytes in length. Therefore, throughput can be calculated as:

$$\begin{aligned} P &= [20,000 \times 0.5 \times 3 \text{ Kb}] / 60 = 500 \text{ Kbytes/sec} \\ &= 4 \text{ megabits per second} \end{aligned}$$

The network connection for the server would therefore have to support this data rate and should be tested to ensure that it does.

20.9.3 Stress Testing

Stress testing (Chapter 13) is a continuation of load testing, but in this instance the variables, N , T , and D are forced to meet and then exceed operational limits. The intent of these tests is to answer each of the following questions:

- Does the system degrade “gently” or does the server shut down as capacity is exceeded?



The intent of stress testing is to better understand how a system fails as it is stressed beyond its operational limits.

- Does server software generate “server not available” messages? More generally, are users aware that they cannot reach the server?
- Does the server queue requests for resources and empty the queue once capacity demands diminish?
- Are transactions lost as capacity is exceeded?
- Is data integrity affected as capacity is exceeded?
- What values of N , T , and D force the server environment to fail? How does failure manifest itself? Are automated notifications sent to technical support staff at the server site?
- If the system does fail, how long will it take to come back on-line?
- Are certain WebApp functions (e.g., compute intensive functionality, data streaming capabilities) discontinued as capacity reaches the 80 or 90 percent level?

A variation of stress testing is sometimes referred to as *spike/bounce testing* [SPL01]. In this testing regime, load is spiked to capacity, then lowered quickly to normal operating conditions, and then spiked again. By bouncing system loading a tester can determine how well the server can marshal resources to meet very high demand and then release them when normal conditions reappear (so that they are ready for the next spike).

SOFTWARE TOOLS



Tools Taxonomy for WebApp Testing

In his paper on the testing of e-commerce systems, Lam [LAM01] presents a useful taxonomy of automated tools that have direct applicability for testing in a Web engineering context. We have appended representative tools in each category.¹⁸

Configuration and content management tools manage version and change control of WebApp content objects and functional components.

Representative tool(s):
Comprehensive list at www.daveeaton.com/scm/CMTools.html

Database performance tools measure database performance, such as the time to perform selected database queries. These tools facilitate database optimization.

Representative tool(s): BMC Software (www.bmc.com)

Debuggers are typical programming tools that find and resolve software defects in the code. They are part of most modern application development environments.

Representative tool(s):
Accelerated Technology (www.acceleratedtechnology.com)
IBM *VisualAge Environment* (www.ibm.com)
JDebugTool (www.debugtools.com)

Defect management systems record defects and track their status and resolution. Some include reporting tools to provide management information on defect spread and defect resolution rates.

Representative tool(s):
EXCEL *Quickbugs* (www.excelsoftware.com)

¹⁸ Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In addition, tool names are registered trademarks of the companies noted.

McCabe *TRUETrack* (www.mccabe.com)

Rational *ClearQuest* (www.rational.com)

Network monitoring tools watch the level of network traffic. They are useful for identifying network bottlenecks and testing the link between front- and back-end systems.

Representative tool(s):

Comprehensive list at www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html

Regression testing tools store test cases and test data and can reapply the test cases after successive software changes.

Representative tool(s):

Compuware *QARun* (www.compuware.com/products/qacenter/qarun)

Rational *VisualTest* (www.rational.com)

Seque Software (www.seque.com)

Site monitoring tools monitor a site's performance, often from a user perspective. Use them to compile statistics such as end-to-end response time and throughput, and to periodically check a site's availability.

Representative tool(s):

Keynote Systems (www.keynote.com)

Stress tools help developers explore system behavior under high levels of operational usage and find a system's breakpoints.

Representative tool(s):

Mercury Interactive (www.merc-int.com)

Scapa Technologies (www.scapatech.com)

System resource monitors are part of most OS server and Web server software; they monitor

resources such as disk space, CPU usage, and memory.

Representative tool(s):

Successful Hosting.com (www.successfulhosting.com)

Quest Software *Foglight* (www.quest.com)

Test data generation tools assist users in generating test data.

Representative tool(s):

Comprehensive list at www.softwareqatest.com/qatweb1.html

Test result comparators help compare the results of one set of testing to that of another set. Use them to check that code changes have not introduced adverse changes in system behavior.

Representative tool(s):

Useful list at www.aptest.com/resources.html

Transaction monitors measure the performance of high-volume transaction processing systems.

Representative tool(s):

QuotiumPro (www.quotium.com)

Software Research *eValid* (www.soft.com/eValid/index.html)

Web-site security tools help detect potential security problems. You can often set up security probing and monitoring tools to run on a scheduled basis.

Representative tool(s):

Comprehensive list at www.timberlinetechnologies.com/products/www.html

20.10 SUMMARY

The goal of WebApp testing is to exercise each of the many dimensions of WebApp quality with the intent of finding errors or uncovering issues that may lead to quality failures. Testing focuses on content, function, structure, usability, navigability, performance, compatibility, interoperability, capacity, and security. Testing also incorporates reviews that occur as the WebApp is designed.

The WebApp testing strategy exercises each quality dimension by initially examining "units" of content, functionality, or navigation. Once individual units have been validated, the focus shifts to tests that exercise the WebApp as a whole. To accomplish this, many tests are derived from the users' perspectives and are driven by information contained in use-cases. A Web engineering test plan is developed

and identifies testing steps, work products (e.g., test cases), and mechanisms for the evaluation of test results. The testing process encompasses seven different types of testing.

Content testing (and reviews) focus on various categories of content. The intent is to uncover both semantic or syntactic errors that affect the accuracy of content or the manner in which it is presented to the end-user. Interface testing exercises the interaction mechanisms that enable a user to communicate with the WebApp and validates aesthetic aspects of the interface. The intent is to uncover errors that result from poorly implemented interaction mechanisms, or omissions, inconsistencies, or ambiguities in interface semantics.

Navigation testing applies use-cases, derived as part of the analysis activity, in the design of test cases that exercise each usage scenario against the navigation design. Navigation mechanisms are tested to ensure that any errors impeding completion of a use-case are identified and corrected. Component testing exercises content and functional units within the WebApp. Each Web page encapsulates content, navigation links, and processing elements that form a "unit" within the WebApp architecture. These units must be tested.

Configuration testing attempts to uncover errors and/or compatibility problems that are specific to a particular client or server environment. Tests are then conducted to uncover errors associated with each possible configuration. Security testing incorporates a series of tests designed to exploit vulnerabilities in the WebApp and its environment. The intent is to find security holes. Performance testing encompasses a series of tests that are designed to assess WebApp response time and reliability as demands on server-side resource capacity increase.

REFERENCES

- [BRO01] Brown, B., *Oracle9i Web Development*, McGraw-Hill, 2nd ed., 2001.
- [CAC02] Cachero, C., et al., "Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification," *Proc. 2nd Intl. Workshop on Web-Oriented Technology*, June 2002, download from www.dsic.upv.es/~west/iwwost02/papers/cachero.pdf.
- [CON03] Constantine, L., and L. Lockwood, *Software for Use*, Addison-Wesley, 1999; see also <http://www.foruse.com/>.
- [GAR02] Garfinkel, S., and G. Spafford, *Web Security, Privacy and Commerce*, O'Reilly & Associates, 2002.
- [HOW97] Hower, Rick, "Beyond Broken Links," *Internet Systems*, 1997 available at <http://www.dbmsmag.com/9707i03.html>.
- [LAM01] Lam, W., "Testing E-Commerce Systems: A Practical Guide," *IEEE IT Pro*, March/April 2001, pp. 19-28.
- [MCC03] McClure, S., S. Shah, and S. Shah, *Web Hacking: Attacks and Defense*, Addison-Wesley, 2003.
- [MIL00] Miller, E., "WebSite Testing," 2000, available at <http://www.soft.com/eValid/Technology/WhitePapers/website.testing.html>.
- [NGU00] Nguyen, H., "Testing Web-Based Applications," *Software Testing and Quality Engineering*, May/June, 2000, available at <http://www.stqemagazine.com>.

- [NGU01] Nguyen, H., *Testing Applications on the Web*, Wiley, 2001.
[SCE02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.
[SPL01] Splaine, S., and S. Jaskiel, *The Web Testing Handbook*, STQE Publishing, 2001.
[TRE03] Trivedi, R., *Professional Web Services Security*, Wrox Press, 2003.
[WAL03] Wallace, D., I. Raggett, and J. Aufgang, *Extreme Programming for Web Projects*, Addison-Wesley, 2003.

- 20.1.** What is the objective of security testing? Who performs this testing activity?
- 20.2.** In your own words, discuss the objectives of testing in a Web engineering context.
- 20.3.** Compatibility is an important quality dimension. What must be tested to ensure that compatibility exists for a WebApp?
- 20.4.** Which errors tend to be more serious—client-side errors or server-side errors? Why?
- 20.5.** Are there any situations in which WebApp testing should be totally disregarded?
- 20.6.** Is it always necessary to develop a formal written test plan? Explain.
- 20.7.** Is it fair to say that the overall WebApp testing strategy begins with user-visible elements and moves toward technology elements? Are there exceptions to this strategy?
- 20.8.** Assume that you are developing an on-line pharmacy (CornerPharmacy.com) that caters to senior citizens. The pharmacy provides typical functions, but also maintains a database for each customer so that it can provide drug information and warn of potential drug interactions. Discuss any special usability tests for this WebApp.
- 20.9.** Describe the steps associated with database testing for a WebApp. Is database testing predominantly a client-side or server-side activity?
- 20.10.** Is it possible to test every configuration that a WebApp is likely to encounter on the server-side? On the client-side? If it is not, how does a Web engineer select a meaningful set of configuration tests?
- 20.11.** What elements of the WebApp can be “unit tested”? What types of tests must be conducted only after the WebApp elements are integrated?
- 20.12.** CornerPharmacy.com (Problem 20.8) has become wildly successful and the number of users has increased dramatically in the first two months of operation. Draw a graph that depicts probable response time as a function of number of users for a fixed set of server-side resources. Label the graph to indicate points of interest on the “response curve.”
- 20.13.** What is the difference between testing for navigation syntax and for navigation semantics?
- 20.14.** What is the difference between testing that is associated with interface mechanisms and testing that addresses interface semantics?
- 20.15.** Is content testing *really* testing in a conventional sense? Explain.
- 20.16.** Assume that you have implemented a drug interaction checking function for CornerPharmacy.com (Problem 20.8). Discuss the types of component-level tests that would have to be conducted to ensure that this function works properly. [Note: a database would have to be used to implement this function.]
- 20.17.** In response to its success CornerPharmacy.com (Problem 20.8) has implemented a special server solely to handle prescription refills. On average, 1000 concurrent users submit a re-

refill request once every two minutes. The WebApp downloads a 500 byte block of data in response. What is the approximate required throughput for this server in megabits per second?

20.18. What is the difference between load testing and stress testing?

The literature for WebApp testing is still evolving. Books by Ash (*The Web Testing Companion*, Wiley, 2003), Dustin and his colleagues (*Quality Web Systems*, Addison-Wesley, 2002), Nguyen [NGU01], and Splaine and Jaskiel [SPL01] are among the most complete treatments of the subject published to date. Mosley (*Client-Server Software Testing on the Desktop and the Web*, Prentice-Hall, 1999) addresses both client-side and server-side testing issues.

Useful information on WebApp testing strategies and methods, as well as a worthwhile discussion of automated testing tools is presented by Stottlemeyer (*Automated Web Testing Toolkit*, Wiley, 2001). Graham and her colleagues (*Software Test Automation*, Addison-Wesley, 1999) present additional material on automated tools.

Nguyen and his colleagues (*Testing Applications for the Web*, second edition, Wiley, 2003) have developed a major update to [NGU01] and provide unique guidance for testing mobile applications. Although Microsoft (*Performance Testing Microsoft .NET Web Applications*, Microsoft Press, 2002) focuses predominantly on its .NET environment, its comments on performance testing can be useful to anyone interested in the subject.

Splaine (*Testing Web Security*, Wiley, 2002), Klevinsky and his colleagues (*Hack I.T.: Security through Penetration Testing*, Addison-Wesley, 2002), Chirillo (*Hack Attacks Revealed*, second edition, Wiley, 2003), and Skoudis (*Counter Hack*, Prentice-Hall, 2001) provide much useful information for those who must design security tests.

A wide variety of information sources on testing for Web engineering is available on the Internet. An up-to-date list of World Wide Web references can be found at the SEPA Web site:

<http://www.mhhe.com/pressman>.

Four



MANAGING SOFTWARE PROJECTS

In this part of *Software Engineering: A Practitioner's Approach*, we consider the management techniques required to plan, organize, monitor, and control software projects. In the chapters that follow, we address the following questions:

- How must people, process, and problems be managed during a software project?
- How can software metrics be used to manage a software project and the software process?
- How do we estimate effort, cost, and project duration?
- What techniques can be used to formally assess the risks that can impact project success?
- How does a software project manager select a set of software engineering work tasks?
- How is a project schedule created?
- What is quality management?
- Why are formal technical reviews so important?
- How is change managed during the development of computer software and after delivery to the customer?

Once these questions are answered, you'll be better prepared to manage software projects in a way that will lead to timely delivery of a high-quality product.

CHAPTER 21

PROJECT MANAGEMENT

KEY CONCEPTS

agile teams

coordination

critical practices

people

problem

decomposition

process

project

software scope

software team

stakeholders

team leaders

W²HH principle

In the preface to his book on software project management, Meiler Page-Jones [PAG85] makes a statement that can be echoed by many software engineering consultants:

I've visited dozens of commercial shops, both good and bad, and I've observed scores of data processing managers, again, both good and bad. Too often, I've watched in horror as these managers futilely struggled through nightmarish projects, squirmed under impossible deadlines, or delivered systems that outraged their users and went on to devour huge chunks of maintenance time.

What Page-Jones describes are symptoms that result from an array of management and technical problems. However, if a post mortem were to be conducted for every project, it is very likely that a consistent theme would be encountered: project management was weak.

In this chapter and the six that follow, we consider the key concepts that lead to effective software project management. This chapter considers basic software project management concepts and principles. Chapter 22 presents process and project metrics, the basis for effective management decision making. The techniques that are used to estimate cost, define a realistic schedule, and establish an effective project plan are discussed in Chapters 23 and 24. The management activities that lead to effective risk monitoring, mitigation, and management are presented in Chapter 25. Finally, Chapters 26 and 27 consider techniques for ensuring quality as a project is conducted and managing changes throughout the life of an application.

QUICK LOOK

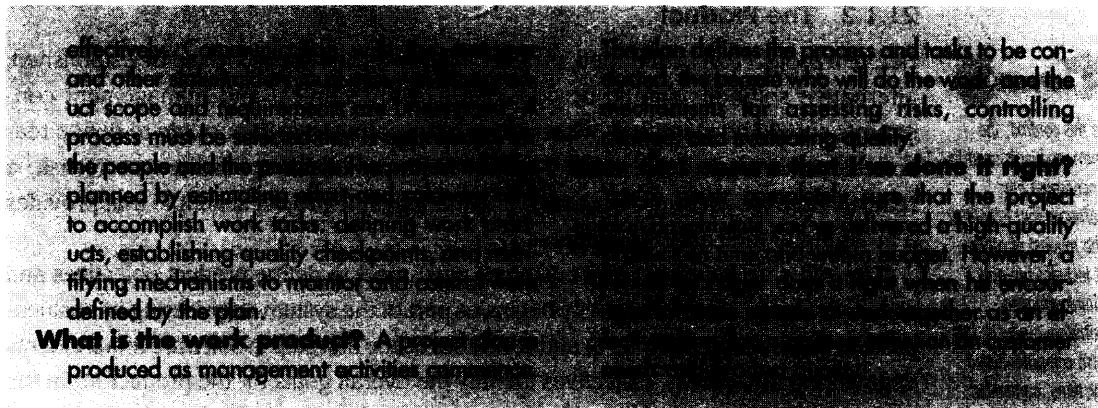
computer-based systems. Project management involves monitoring and controlling the progress of software development and events that occur in software development. A preliminary concept to an abstract problem is a preliminary concept to an abstract problem.

Who does it? Everyone manages systems to some extent, but the scope of management activities varies among people involved in a software project. A software engineer manages her day-

to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and software development.

What is it? Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long period. Why software projects need to be managed.

What are the steps? Understand the four P's—people, process, product, and project. People must be organized to perform software work



21.1 THE MANAGEMENT SPECTRUM

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a project risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the product.

21.1.1 The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s (e.g., [COU80], [WIT94], [DEM98]). In fact, the “people factor” is so important that the Software Engineering Institute has developed a people management capability maturity model (PM-CMM), “to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability” [CUR94].

The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The PM-CMM is a companion to the Software Capability Maturity Model Integration (Chapter 2) that guides organizations in the creation of a mature software process. Issues associated with people management and structure for software projects are considered later in this chapter.



In this context, the term product is used to encompass any software that is built at the request of others. It includes not only shrink-wrapped software products, but also computer-based systems, embedded software, WebApps, and problem-solving software (e.g., programs for engineering/scientific problem solving).

21.1.2 The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

The software developer and customer must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering (Chapter 6) and continues as the first step in software requirements engineering (Chapter 7). Objectives identify the overall goals for the product (from the customer's point of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more importantly, attempts to bound these characteristics in a quantitative manner.

Once the product objectives and scope are understood, alternative solutions are considered. Although relatively little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

21.1.3 The Process



Those who adhere to the agile process philosophy (Chapter 4) argue that their process is leaner than others. That may be true, but they still have a process, and agile software engineering still requires discipline.

A software process (Chapters 2, 3, and 4) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

21.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, we still struggle. In 1998, industry data indicated that 26 percent of software projects failed outright and 46 percent experienced cost and schedule overruns [REE99]. Although the success rate for software projects has improved somewhat, our project failure rate remains higher than it should be.¹

¹ Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially. Part of the answer, I think, is that a substantial number of these "failed" projects are ill-conceived in the first place. Customers lose interest quickly (because what they've requested wasn't really as important as they first thought), and the projects are cancelled.

Some projects are like a road trip. Some projects are simple and routine, like driving to the store in broad daylight. Other projects are more like driving a truck off-road, in the mountains, at night.”
 Cem Kaner, James Bach, and Gabe Potts

To avoid project failure, a software project manager and the software engineers who build the product must heed a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 21.5 and in the chapters that follow.

21.2 PEOPLE

In a study published by the IEEE [CUR88], the engineering vice presidents of three major technology companies were asked the most important contributor to a successful software project. They answered in the following way:

VP 1: I guess if you had to pick one thing out that is most important in our environment, I'd say it's not the tools that we use, it's the people.

VP 2: The most important ingredient that was successful on this project was having smart people . . . very little else matters in my opinion . . . The most important thing you do for a project is selecting the staff . . . The success of the software development organization is very, very much associated with the ability to recruit good people.

VP 3: The only rule I have in management is to ensure I have good people—real good people—and that I grow good people—and that I provide an environment in which good people can produce.

Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue (as the preceding group had) that people are primary, but their actions sometimes belie their words. In this section we examine the stakeholders who participate in the software process and the manner in which they are organized to perform effective software engineering.

21.2.1 The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

1. *Senior managers* who define business issues that often have significant influence on the project.
2. *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
3. *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.

4. *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
5. *End-users* who interact with the software once it is released for production use.

Every software project is populated by people who fall within this taxonomy.² To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

21.2.2 Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers" [EDG95].

In an excellent book of technical leadership, Jerry Weinberg [WEI86] suggests a MOI model of leadership:

What do we look for when choosing someone to lead a software project?

Motivation. The ability to encourage (by "push or pull") technical people to produce to their best ability.

Organization. The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

"A leader is one who knows where he wants to go, and gets up, and goes."

Another view [EDG95] of the characteristics that define an effective project manager emphasizes four key traits:

Problem solving. An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply

² When Web applications are developed (Part 3 of this book), other nontechnical people may be involved in content creation.

lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity. A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement. To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actions that controlled risk taking will not be punished.

Influence and team building. An effective project manager must be able to “read” people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

21.2.3 The Software Team

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager’s scope of responsibility. However, the organization of the people directly involved in a software project is within the project manager’s purview.

“A group is not a team, and not every team is effective.”

The “best” team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [MAN81] describes seven project factors that should be considered when planning the structure of software engineering teams:

What factors should be considered when the structure of a software team is chosen?

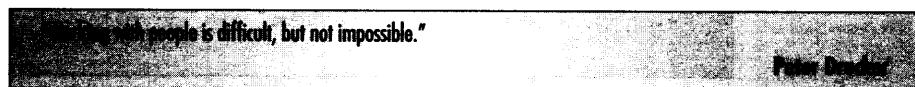
- The difficulty of the problem to be solved.
- The “size” of the resultant program(s) in lines of code or function points (Chapter 22).
- The time that the team will stay together (team lifetime).
- The degree to which the problem can be modularized.
- The required quality and reliability of the system to be built.
- The rigidity of the delivery date.
- The degree of sociability (communication) required for the project.

“To be incrementally better. Be competitive. If you want to be exponentially better, be competitive.”

What options do we have when defining the structure of a software team?

Constantine [CON93] suggests four “organizational paradigms” for software engineering teams:

1. A *closed paradigm* structures a team along a traditional hierarchy of authority. Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. A *random paradigm* structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when “orderly performance” is required.
3. An *open paradigm* attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively. Heavy communication and consensus-based decision making are the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.
4. A *synchronous paradigm* relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.



As an historical footnote, one of the earliest software team organizations was a closed paradigm structure originally called the *chief programmer team*. This structure was first proposed by Harlan Mills and described by Baker [BAK72]. The nucleus of the team is composed of a *senior engineer* (the chief programmer), who plans, coordinates, and reviews all technical activities of the team; *technical staff* (normally two to five people), who conduct analysis and development activities; and a *backup engineer*, who supports the senior engineer in his or her activities and can replace the senior engineer with minimum loss in project continuity.

The chief programmer may be served by one or more *specialists* (e.g., telecommunications expert, database designer), *support staff* (e.g., technical writers, clerical personnel), and a *software librarian*.

As a counterpoint to the chief programmer team structure, Constantine’s random paradigm [CON93] suggests a software team with creative independence whose approach to work might best be termed *innovative anarchy*. Although the free-spirited approach to software work has appeal, channeling creative energy into a high-performance team must be a central goal of a software engineering organization. To achieve a high-performance team:

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.

Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book, *Peopleware*, DeMarco and Lister [DEM98] discuss this issue:

**What is a
"jelled"
team?**

We tend to use the word *team* fairly loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't seem like teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell*.

A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts . . .

Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

**Why do
teams fail
to jell?**

But not all teams jell. In fact, many teams suffer from what Jackman [JAC98] calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

To avoid a frenzied work environment, the project manager should be certain that the team has access to all information required to do the job and that major goals and objectives, once defined, should not be modified unless absolutely necessary. A software team can avoid frustration (and stress) if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built and the people doing the work, and by allowing the team to select its own process model. The team itself should establish mechanisms for accountability (formal technical reviews and pair programming are excellent ways to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving.

"Do or do not. There is no try."

Yoda from Star Wars

In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some team members are extroverts; others are introverted. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on “feel.” Some practitioners want a detailed schedule populated by organized tasks that enable them to achieve closure for some element of a project. Others prefer a more spontaneous environment in which open issues are okay. Some work hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are energized by the rush to make a last minute deadline. A detailed discussion of the psychology of these traits and the ways in which a skilled team leader can help people with opposing traits to work together is beyond the scope of this book.³ However, it is important to note that recognition of human differences is the first step toward creating teams that jell.

21.2.4 Agile Teams

In recent years, agile software development (Chapter 4) has been proposed as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software; small highly motivated project teams; informal methods; minimal software engineering work products, and overall development simplicity.

The small, highly motivated project team, also called an *agile team*, adopts many of the characteristics of successful software project teams discussed in the preceding section and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team. Cockburn and Highsmith [COC01] note this when they write:

If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy—“people trump process” is one way to say this. However, lack of user and executive support can kill a project—“politics trump people.” Inadequate support can keep even good people from accomplishing the job . . .

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are *self-organizing*. A self-organizing team does not necessarily maintain a single team structure but in-

³ An excellent introduction to these issues as they relate to software project teams can be found in [FER98].

stead uses elements of Constantine's random, open, and synchronous paradigms discussed in Section 21.2.3.

"Collective ownership is nothing more than an instantiation of the idea that products should be attributable to the [agile] team, not individuals who make up the team."

KEY POINT

An agile team is a self-organizing team that has autonomy to plan and make technical decisions.

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct brief daily team meetings to coordinate and synchronize the work that must be accomplished for that day.

Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self-organization and collaboration move the team toward a completed software increment.

21.2.5 Coordination and Communication Issues

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software—scale, uncertainty, and interoperability—are facts of life. To deal with them effectively, a software engineering team must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively noninteractive and impersonal communication channels" [KRA95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

SAFEHOME

**Team Structure**

The scene: Doug Miller's office prior to the start of the *SafeHome* software project.

Doug: Welcome, Doug Miller (manager of the *SafeHome* software project), Vinod Raman, Jamie Lopez, and other members of the product software engineering team.

Doug: The client's great. They've given us the requirements.

Jamie: Have you guys had a chance to look over the requirements for *SafeHome* that marketing's prepared?

Doug: Yeah, and looking at his

requirements. Yeah, but we have a bunch of questions.

Jamie: Let's hold on that for a moment. I'd like to talk about what we're going to structure the team, who's responsible for what.

Doug: I'm really into the agile philosophy, Doug. I think we should be a self-organizing team.

Jamie: I agree. Given the tight time line and some of the uncertainty, and that fact that we're all really experienced [laughs], that seems like the right way to go.

Doug: That's okay with me, but you guys know the drill.

Jamie (smiling and talking as if she were reciting something): We make logical decisions about who does what and when, but it's our responsibility to get product out the door on time.

Vinod: And with quality.

Doug: Exactly. But remember there are constraints. Marketing defines the software increments to be produced—in consultation with us, of course.

Jamie: And?

Doug: And, we're going to use UML as our modeling approach.

Vinod: But keep extraneous documentation to an absolute minimum.

Doug: Who is the liaison with me?

Jamie: We decided that Vinod will be the tech lead—he's got the most experience, so Vinod is your liaison, but feel free to talk to any of us.

Doug (laughing): Don't worry, I will.

21.3 THE PRODUCT

A software project manager is confronted with a dilemma at the very beginning of a software engineering project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now!"

Therefore, we must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

21.3.1 Software Scope

The first software project management activity is the determination of *software scope*. Scope is defined by answering the following questions:

Context. How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?



If you can't bound a characteristic of the software you intend to build, list the characteristic as a project risk (Chapter 25).

Information objectives. What customer-visible data objects (Chapter 8) are produced as output from the software? What data objects are required for input?

Function and performance. What functions does the software perform to transform input data into output? Are there any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, size of mailing list, maximum allowable response time) are stated explicitly; constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in C++) are described.

21.3.2 Problem Decomposition

Problem decomposition, sometimes called *partitioning* or *problem elaboration*, is an activity that sits at the core of software requirements analysis (Chapters 7 and 8). During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality that must be delivered and (2) the process that will be used to deliver it.

Human beings tend to apply a divide-and-conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation (Chapter 23). Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful.

As an example, consider a project that will build a new word-processing product. Among the unique features of the product are continuous voice as well as keyboard input, extremely sophisticated "automatic copy edit" features, page layout capability, automatic indexing and table of contents, and others. The project manager must first establish a statement of scope that bounds these features (as well as other more mundane functions such as editing, file management, document production, and the like). For example, will continuous voice input require that the product be "trained" by the user? Specifically, what capabilities will the copy edit feature provide? Just how sophisticated will the page layout capability be?

As the statement of scope evolves, a first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing: (1) spell checking, (2) sentence grammar checking, (3) reference checking for large documents (e.g., Is a reference to a bibliography entry found in the list of entries in the bibliography?), and (4) section and chapter reference validation for large



To develop a reasonable project plan, you must decompose the problem. This can be accomplished using a list of functions, or with use-cases, or for agile work, user stories.

documents. Each of these features represents a subfunction to be implemented in software. Each can be further refined if the decomposition will make planning easier.



The framework activities (Chapter 2) that characterize the software process are applicable to all software projects. The problem is to select the process model that is appropriate for the software to be engineered by a project team.

The project manager must decide which process model is most appropriate for (1) the customers who have requested the product and the people who will do the work, (2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities, must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 24.



An automated project scheduling tool can be used to create a "task network" (Chapter 24). The network is loaded with estimated resource requirements, start/end dates, and other pertinent data. This resource loaded network can then be used for project tracking and control.

21.4.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization. Assume that the organization has adopted the following set of framework activities (Chapter 2): communications, planning, modeling, construction, and deployment.

The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 21.1 is created. Each major product function (the figure notes functions for the word-

FIGURE 21.1

Melding the problem and the process

COMMON PROCESS FRAMEWORK ACTIVITIES	communication	planning	modeling	construction	deployment
Software Engineering Tasks					
Product Functions					
Text input					
Editing and formatting					
Automatic copy edit					
Page layout capability					
Automatic indexing and TOC					
File management					
Document production					

KEY POINT

The process framework establishes a skeleton for project planning. It is adapted by allocating a task set that is appropriate to the project.

processing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.⁴ The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each task. These activities are considered in Chapter 24.

21.4.2 Process Decomposition

A software team should have a significant degree of flexibility in choosing the software process model that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If very tight time constraints are imposed and the problem can be heavily compartmentalized, the RAD model is probably the right option. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models.⁵

Once the process model has been chosen, the process framework is adapted to it. In every case, the generic framework discussed earlier—communication, planning, modeling, construction, and deployment—can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The process framework is invariant and serves as the basis for all software work performed by a software organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks, “How do we accomplish this framework activity?” For example, a small, relatively simple project might require the following work tasks for the communication activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

⁴ It should be noted that work tasks must be adapted to the specific needs of the project.

⁵ Recall that project characteristics also have a strong bearing on the structure of the software team (Section 21.2.3).

Now, we consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the communication activity:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a “working document” and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software. Alternatively, develop use-cases that describe the software from the user’s point of view.
7. Review each mini-spec or use-case for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document or collection of use-cases with all concerned.
10. Modify the scoping document or use-cases as required.

Both projects perform the framework activity that we call “communication,” but the first project team performed half as many software engineering work tasks as the second.

21.5 THE PROJECT

To manage a successful software project, we must understand what can go wrong (so that problems can be avoided). In an excellent paper on software projects, John Reel [REE99] defines 10 signs that indicate that an information systems project is in jeopardy:

1. Software people don’t understand their customer’s needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change [or are ill-defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

? What are the signs that a software project is in jeopardy?

Jaded industry professionals often refer (half-facetiously) to the 90–90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes the other 90 percent of the allotted effort and time [ZAH94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceding list.

"We don't have time to stop for gas, we're already late."

M. Cloran


But enough negativity! How does a manager act to avoid the problems just noted? Reel [REE99] suggests a five-part common-sense approach to software projects:

1. *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 21.2.3) and giving the team the autonomy, authority, and technology needed to do the job.
2. *Maintain momentum.* Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.⁶
3. *Track progress.* For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 22) can be collected and used to assess progress against averages developed for the software development organization.
4. *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).
5. *Conduct a postmortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

⁶ The implication of this statement is that bureaucracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is eliminated. The team should be self-organizing and autonomous.

21.6 THE W⁵HH PRINCIPLE

In an excellent paper on software process and projects, Barry Boehm [BOE96] states: “you need an organizing principle that scales down to provide simple [project] plans for simple projects.” Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the W⁵HH principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

 **How do we define key project characteristics?**

Why is the system being developed? The answer to this question enables all parties to assess the validity of business reasons for the software work. Stated in another way, does the business purpose justify the expenditure of people, time, and money?

What will be done? The answer to this question establishes the task set that will be required for the project.

When will it be done? The answer to this question helps the team to establish a project schedule by identifying when project tasks are to be conducted and when milestones are to be reached.

Who is responsible for a function? Earlier in this chapter, we noted that the role and responsibility of each member of the software team must be defined. The answer to this question helps accomplish this.

Where are they organizationally located? Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 23) based on answers to earlier questions.

Boehm’s W⁵HH principle is applicable regardless of the size or complexity of a software project. The questions noted provide an excellent planning outline for the project manager and the software team.

21.7 CRITICAL PRACTICES

The Airlie Council⁷ has developed a list of “critical software practices for performance-based management.” These practices are “consistently used by, and considered critical by, highly successful software projects and organizations whose ‘bottom line’ performance is consistently much better than industry averages” [AIR99].

⁷ The Airlie Council is a team of software engineering experts chartered by the U.S. Department of Defense to help develop guidelines for best practices in software project management and software engineering.

Critical practices⁸ include: metrics-based project management (Chapter 22), empirical cost and schedule estimation (Chapters 23 and 24), earned value tracking (Chapter 24), formal risk management (Chapter 25), defect tracking against quality targets (Chapter 26), and people-aware management (Section 21.2). Each of these critical practices is addressed throughout Part 4 of this book.

SOFTWARE TOOLS



Software Tools for Project Managers

The “tools” listed here are generic and apply to a broad range of activities performed by project managers. Specific project management tools (e.g., scheduling tools, estimating tools, risk analysis tools) are considered in later chapters.

Representative Tools⁹

The Software Program Manager’s Network (www.spmn.com) has developed a simple tool called *Project Control Panel* which provides project managers

with a direct indication of project status. The tool has “gauges” much like a dashboard and is implemented with Microsoft Excel. It is available for download at http://www.spmn.com/products_software.html. *Gantthead.com* has developed a set of useful *checklists for project managers* that can be downloaded from <http://www.gantthead.com/>. *Ittoolkit.com* (www.ittoolkit.com) provides “a collection of planning guides, process templates and smart worksheets” available on CD-ROM.

21.8 SUMMARY

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development, and support of computer software.

Four P’s have a substantial influence on software project management—people, product, process, and project. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication. The product requirements must be communicated from customer to developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

The pivotal element in all software projects is people. Software engineers can be organized in a number of different team structures that range from traditional control hierarchies to “open paradigm” teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general,

⁸ Only those critical practices associated with “project integrity” are noted here.

⁹ Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

formal reviews and informal person-to-person communication have the most value for practitioners.

The project management activity encompasses measurement and metrics, estimation and scheduling, risk analysis, tracking, and control. Each of these topics is considered in the chapters that follow.

REFERENCES

- [AIR99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.
- [BAK72] Baker, F. T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, vol. 11, no. 1, 1972, pp. 56-73.
- [BOE96] Boehm, B., "Anchoring the Software Process," *IEEE Software*, vol. 13, no. 4, July 1996, pp. 73-82.
- [COC01] Cockburn, A., and J. Highsmith, "Agile Software Development: The People Factor," *IEEE Computer*, vol. 34, no. 11, November 2001, pp. 131-133.
- [CON93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization," *CACM*, vol. 36, no. 10, October 1993, pp. 34-43.
- [COU80] Cougar, J., and R. Zawacki, *Managing and Motivating Computer Personnel*, Wiley, 1980.
- [CUR88] Curtis, B., et al., "A Field Study of the Software Design Process for Large Systems," *IEEE Trans. Software Engineering*, vol. SE-31, no. 11, November 1988, pp. 1268-1287.
- [CUR94] Curtis, B., et al., *People Management Capability Maturity Model*, Software Engineering Institute, 1994.
- [DEM98] DeMarco, T., and T. Lister, *Peopleware*, 2nd ed., Dorset House, 1998.
- [EDG95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager," *Application Development Trends*, vol. 2, no. 5, May 1995, pp. 37-42.
- [FER98] Ferdinandi, P. L., "Facilitating Communication," *IEEE Software*, September 1998, pp. 92-96.
- [JAC98] Jackman, M., "Homeopathic Remedies for Team Toxicity," *IEEE Software*, July 1998, pp. 43-45.
- [KRA95] Kraul, R., and L. Streeter, "Coordination in Software Development," *CACM*, vol. 38, no. 3, March 1995, pp. 69-81.
- [MAN81] Mantei, M., "The Effect of Programming Team Structures on Programming Tasks," *CACM*, vol. 24, no. 3, March 1981, pp. 106-113.
- [PAG85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. vii.
- [REE99] Reel, J. S., "Critical Success Factors in Software Projects," *IEEE Software*, May, 1999, pp. 18-23.
- [WEI86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.
- [WIT94] Whitaker, K., *Managing Software Maniacs*, Wiley, 1994.
- [ZAH94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development*, March 1994, pp. 35-38.

PROBLEMS AND POINTS TO PONDER

- 21.1.** The Software Engineering Institute's people management capability maturity model (PM-CMM) takes an organized look at "key practice areas" that cultivate good software people. Your instructor will assign you one KPA for analysis and summary.
- 21.2.** Describe three real-life situations in which the customer and the end-user are the same. Describe three situations in which they are different.
- 21.3.** Based on information contained in this chapter and your own experience, develop "10 commandments" for empowering software engineers. That is, make a list of 10 guidelines that will lead to software people who work to their full potential.

- 21.4.** The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.
- 21.5.** You have been appointed a project manager for a major software products company. Your job is to manage the development of the next generation version of its widely used word-processing software. Because new revenue must be generated, tight deadlines have been established and announced. What team structure would you choose and why? What software process model(s) would you choose and why?
- 21.6.** Do a first level functional decomposition of the page layout function discussed briefly in Section 21.3.2.
- 21.7.** You have been asked to develop a small application that analyzes each course offered by a university and reports the average grade obtained in the course (for a given term). Write a statement of scope that bounds this problem.
- 21.8.** You have been appointed a project manager for a small software products company. Your job is to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Because competition for the home entertainment market is intense, there is significant pressure to get the job done. What team structure would you choose and why? What software process model(s) would you choose and why?
- 21.9.** You have been appointed a software project manager for a company that services the genetic engineering world. Your job is to manage the development of a new software product that will accelerate the pace of gene typing. The work is R&D oriented, but the goal is to produce a product within the next year. What team structure would you choose and why? What software process model(s) would you choose and why?
- 21.10.** Review a copy of Weinberg's book [WEI86] and write a two- or three-page summary of the issues that should be considered in applying the MOI model.
- 21.11.** You have been appointed a project manager within an information systems organization. Your job is to build an application that is quite similar to others your team has built, although this one is larger and more complex. Requirements have been thoroughly documented by the customer. What team structure would you choose and why? What software process model(s) would you choose and why?

FURTHER READINGS AND INFORMATION SOURCES

The Project Management Institute (*Guide to the Project Management Body of Knowledge*, PMI, 2001) covers all important aspects of project management. Murch (*Project Management: Best Practices for IT Professionals*, Prentice-Hall, 2000) teaches basic skills and provides detailed guidance for all phases of an IT project. Lewis (*Project Managers Desk Reference*, McGraw-Hill, 1999) presents a 16-step process for planning, monitoring, and controlling any type of project. McConnell (*Professional Software Development*, Addison-Wesley, 2004) offers pragmatic advice for achieving "shorter schedules, higher quality products, and more successful projects."

An excellent four-volume series written by Weinberg (*Quality Software Management*, Dorset House, 1992, 1993, 1994, 1996) introduces basic systems thinking and management concepts; explains how to use measurements effectively; and addresses "congruent action," the ability to establish "fit" between the manager's needs, the needs of technical staff, and the needs of the business. It will provide both new and experienced managers with useful information. Futrell and his colleagues (*Quality Software Project Management*, Prentice-Hall, 2002) present a voluminous treatment of project management.

Phillips (*IT Project Management: On Track from Start to Finish*, McGraw-Hill/Osborne, 2002), Charvat (*Project Management Nation*, Wiley, 2002), Schwalbe (*Information Technology Project Management*, second edition, Course Technology, 2001) and Holtsnider and Jaffe (*IT Manager's Handbook*, Morgan Kaufmann Publishers, 2000) are representative of the many books that have been written on software project management. Brown and his

colleagues (*AntiPatterns in Project Management*, Wiley, 2000) discuss what not to do during the management of a software project.

Brooks (*The Mythical Man Month*, Anniversary Edition, Addison-Wesley, 1995) has updated his classic book to provide new insight into software project and management issues. McConnell (*Software Project Survival Guide*, Microsoft Press, 1997) presents excellent pragmatic guidance for those who must manage software projects. Purba and Shah (*How to Manage a Successful Software Project*, second edition, Wiley, 2000) present a number of case studies that indicate why some projects succeed and others fail. Bennatan (*On Time Within Budget*, third edition, Wiley, 2000) presents useful tips and guidelines for software project managers.

It can be argued that the most important aspect of software project management is people management. Cockburn (*Agile Software Development*, Addison-Wesley, 2002) presents one of the best discussions of software people written to date. DeMarco and Lister [DEM98] have written the definitive book on software people and software projects. In addition, the following books on this subject have been published in recent years and are worth examining:

Beaudouin-Lafon, M., *Computer Supported Cooperative Work*, Wiley-Liss, 1999.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones*, Prentice Hall, 1999.

Constantine, L., *Peopleware Papers: Notes on the Human Side of Software*, Prentice-Hall, 2001.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process*, Addison-Wesley, 1997.

Humphrey, W. S., *Introduction to the Team Software Process*, Addison-Wesley, 1999.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development*, McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments*, IEEE Computer Society, 1998.

Ensworth (*The Accidental Project Manager*, Wiley, 2001) provides much useful guidance to those who must survive "the transition from techie to project manager." Another excellent book by Weinberg [WEI86] is must reading for every project manager and every team leader. It will give you insight and guidance that will enable you to do your job more effectively.

Even though they do not relate specifically to the software world and sometimes suffer from over-simplification and broad generalization, best-selling "management" books by Bossidy (*Execution: The Discipline of Getting Things Done*, Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century*, Harper Business, 1999), Buckingham and Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently*, Simon and Schuster, 1999) and Christensen (*The Innovator's Dilemma*, Harvard Business School Press, 1997) emphasize "new rules" defined by a rapidly changing economy. Older titles such as *Who Moved My Cheese?*, *The One-Minute Manager*, and *In Search of Excellence* continue to provide valuable insights that can help you to manage people and projects more effectively.

A wide variety of information sources on software project management is available on the internet. An up-to-date list of World Wide Web references can be found at the SEPA Web site:

<http://www.mhhe.com/pressman>.

METRICS FOR PROCESS AND PROJECTS

22

KEY CONCEPTS

metrics

function-oriented

object-oriented

private

project

process

public

quality

size-oriented

use-case

WebApp

metrics baseline

metrics programs

SSPI

Measurement enables us to gain insight into the process and the project by providing a mechanism for objective evaluation. Lord Kelvin once said:

When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.

The software engineering community has taken Lord Kelvin's words to heart. But not without frustration and more than a little controversy!

Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of work products and to assist in tactical decision-making as a project proceeds (Chapter 15).

In their guidebook on software measurement, Park, Goethert, and Florac [PAR96] note the reasons that we measure: (1) to *characterize* in an effort to gain an understanding "of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments"; (2) to *evaluate* "to determine status with respect to plans"; (3) to *predict* by "gaining understandings

QUICK LOOK

What is it? Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

Who does it? Software metrics are analyzed and assessed by software managers. Measures are often collected by software engineers.

Why is it important? If you don't measure, judgment can be based only on subjective evaluation. With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished.

What are the steps? Begin by defining a limited set of process and project measures that are easy to collect. These measures are often normalized using either size- or function-oriented metrics. The result is analyzed and compared to

past averages for similar projects performed within the organization. Trends are assessed and conclusions are generated.

What is the work product? A set of software metrics that provides insight into the process and an understanding of the project.

How do I ensure that I've done it right?

By applying a consistent, yet simple measurement scheme that is never used to assess, reward, or punish individual performance.

of relationships among processes and products and building models of these relationships"; and (4) to *improve* by "identify[ing] roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance."

Measurement is a management tool. If conducted properly, it provides a project manager with insight. And as a result, it assists the project manager and the software team in making decisions that will lead to a successful project.

22.1 METRICS IN THE PROCESS AND PROJECT DOMAINS

KEY POINT

Process metrics have long-term impact. Their intent is to improve the process itself. Project metrics often contribute to the development of process metrics.

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement. *Project metrics* enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

Measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domain.

22.1.1 Process Metrics and Software Process Improvement

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement. But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance" [PAU94].

Referring to Figure 22.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown [BOE81] to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact.