Each of these options is viable. A software organization must choose the one that is most appropriate for each case.

**Reverse engineering.** The term *reverse engineering* has its origins in the hardware world. A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing "secrets." These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained. But these documents are proprietary and unavailable to the company doing the reverse engineering. In essence, successful reverse engineering derives one or more design and manufacturing specifications for a product by examining actual specimens of the product.

Reverse engineering for software is quite similar. In most cases, however, the program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier). The "secrets" to be understood are obscure because no specification was ever developed. Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of *design recovery.* Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

**WebRef**
An array of resources for the reengineering community can be obtained at **www.comp.lancs.ac.uk/projects/RenaissanceWeb/.**

**Code restructuring.** The most common type of reengineering (actually, the use of the term reengineering is questionable in this case) is *code restructuring.*[3] Some legacy systems have a relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted, and code is then restructured (this can be done automatically). The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

**Data restructuring.** A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, data architecture has more to do with the long-term viability of a program that the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined (Chapter 9). Data objects and attributes are identified, and existing data structures are reviewed for quality.

---

3 Code restructuring has some of the elements of "refactoring," a redesign concept introduced in Chapter 4 and discussed elsewhere in this book.

When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.

Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

**Forward engineering.** In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an "engine" will appear, but vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More important, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering, also called *renovation* or *reclamation* [CHI90], not only recovers design information from existing software, but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software reimplements the function of the existing system and also adds new functions and/or improves overall performance.

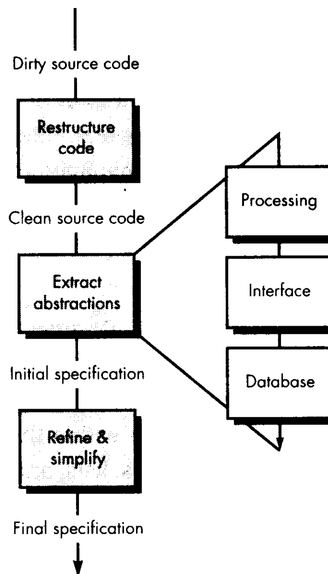
## 31.3 Reverse Engineering

*Reverse engineering* conjures an image of the "magic slot." We feed a haphazardly designed, undocumented source listing into the slot and out the other end comes a complete design description (and full documentation) for the computer program. Unfortunately, the magic slot doesn't exist. Reverse engineering can extract design information from source code, but the abstraction level, the completeness of the documentation, the degree to which tools and a human analyst work together, and the directionality of the process are highly variable.

The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code. Ideally, the abstraction level should be as high as possible. That is, the reverse engineering process should be capable of deriving procedural design representations (a low-level abstraction), program and data structure information (a somewhat higher level of abstraction), object models, data and/or control flow models (a relatively high level of abstraction), and UML class, state and deployment diagrams (a high level of abstraction). As the abstraction level increases, the software engineer is provided with information that will allow easier understanding of the program.

The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple design repre-

**FIGURE 31.3**
**The reverse engineering process**




sentations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.

**KEY POINT**
Three reverse engineering issues must be addressed: abstraction level, completeness, and directionality.

Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is "integrated" with automated tools to create an effective reverse engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.

If the *directionality* of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.

The reverse engineering process is represented in Figure 31.3. Before reverse engineering activities can commence, unstructured ("dirty") source code is restructured (Section 31.4.1) so that it contains only the structured programming constructs.[4] This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.

The core of reverse engineering is an activity called *extract abstractions*. The engineer must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

4 Code can be restructured using a *restructuring engine*—a tool that restructures source code.

**WebRef**
Useful resources for "design recovery and program understanding" can be found at **www.sel.iit.nrc.ca/projects/dr/dr.html.**

### 31.3.1 Reverse Engineering to Understand Data

Reverse engineering of data occurs at different levels of abstraction and is often the first reengineering task. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems). Reverse engineering of the current global data structures sets the stage for the introduction of a new system-wide database.

*Seemingly insignificant compromises in data structures can lead to potentially catastrophic problems in future years. Consider the Y2K problem as an example.*

**Internal data structures.** Reverse engineering techniques for internal program data focus on the definition of classes of objects. This is accomplished by examining the program code with the intent of grouping related program variables. In many cases, the data organization within the code identifies abstract data types. For example, record structures, files, lists, and other data structures often provide an initial indicator of classes.

**Database structure.** Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

The following steps [PRE94] may be used to define the existing data model as a precursor to reengineering a new database model: (1) build an initial object model, (2) determine candidate keys, (3) refine the tentative classes, (4) define generalizations, and (5) discover associations (use techniques that are analogous to the CRC approach). Once information defined in the preceding steps is known, a series of transformations [PRE94] can be applied to map the old database structure into a new database structure.

### 31.3.2 Reverse Engineering to Understand Processing

Reverse engineering to understand processing begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

The overall functionality of the entire application system must be understood before more detailed reverse engineering work occurs. This establishes a context for further analysis and provides insight into interoperability issues among applications within the system. Each of the programs that make up the application system represents a functional abstraction at a high level of detail. A block diagram, representing the interaction between these functional abstractions, is created. Each component performs some subfunction and represents a defined procedural abstraction. A processing narrative for each component is developed. In some situations, system,

program, and component specifications already exist. When this is the case, the specifications are reviewed for conformance to existing code.[5]

> "There exists a passion for comprehension, just as there exists a passion for music. That passion is rather common in children, but gets lost in most people later on."
>
> **Albert Einstein**

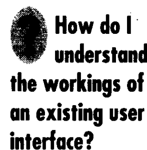
Things become more complex when the code inside a component is considered. The engineer looks for sections of code that represent generic procedural patterns. In almost every component, a section of code prepares data for processing (within the module), a different section of code does the processing, and another section of code prepares the results of processing for export from the component. Within each of these sections, we can encounter smaller patterns; for example, data validation and bounds checking often occur within the section of code that prepares data for processing.

For large systems, reverse engineering is generally accomplished using a semi-automated approach. Automated tools are used to help the software engineer understand the semantics of existing code. The output of this process is then passed to restructuring and forward engineering tools to complete the reengineering process.

### 31.3.3 Reverse Engineering User Interfaces

Sophisticated GUIs are now de rigueur for computer-based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur.

To fully understand an existing user interface, the structure and behavior of the interface must be specified. Merlo and his colleagues [MER93] suggest three basic questions that must be answered as reverse engineering of the UI commences:



**How do I understand the workings of an existing user interface?**

- What are the basic actions (e.g., keystrokes and mouse clicks) that the interface must process?
- What is a compact description of the behavioral response of the system to these actions?
- What is meant by a "replacement," or more precisely, what concept of equivalence of interfaces is relevant here?

Behavioral modeling notation (Chapter 8) can provide a means for developing answers to the first two questions. Much of the information necessary to create a behavioral model can be obtained by observing the external manifestation of the existing interface. But additional information necessary to create the behavioral model must be extracted from the code.

5 Often, specifications written early in the life history of a program are never updated. As changes are made, the code no longer conforms to the specification.

It is important to note that a replacement GUI may not mirror the old interface exactly (in fact, it may be radically different). It is often worthwhile to develop new interaction metaphors. For example, an old GUI requests that a user provide a scale factor (ranging from 1 to 10) to shrink or magnify a graphical image. A reengineered GUI might use a slide-bar and mouse to accomplish the same function.

SOFTWARE TOOLS

**Reverse Engineering**

**Objective:** To help software engineers understand the internal design structure of complex programs.

**Mechanics:** In most cases, reverse engineering tools accept source code as input and produce a variety of structural, procedural, data, and behavioral design representations.

**Representative Tools**[6]

*Imagix 4D,* developed by Imagix (www.imagix.com), "helps software developers understand complex or legacy C and C++ software" by reverse engineering and documenting source code.

*Understand,* developed by Scientific Toolworks, Inc. (www.scitools.com), parses Ada, Fortran, C, C++, and Java "to reverse engineer, automatically document, calculate code metrics, and help you understand, navigate and maintain source code."

A comprehensive listing of reverse engineering tools can be found at http://scgwiki.iam.unibe.ch:8080/SCG/370.

## 31.4 RESTRUCTURING

Software restructuring modifies source code and/or data in an effort to make it amenable to future changes. In general, restructuring does not modify the overall program architecture. It tends to focus on the design details of individual modules and on local data structures defined within modules. If the restructuring effort extends beyond module boundaries and encompasses the software architecture, restructuring becomes forward engineering (Section 31.5).

Restructuring occurs when the basic architecture of an application is solid, even though technical internals need work. It is initiated when major parts of the software are serviceable and only a subset of all components and data need extensive modification.[7]

### 31.4.1 Code Restructuring

*Code restructuring* is performed to yield a design that produces the same function as the original program but with higher quality. In general, code restructuring techniques (e.g., Warnier's logical simplification techniques [WAR74]) model program logic using Boolean algebra and then apply a series of transformation rules that yield restructured

6 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.

7 It is sometimes difficult to make a distinction between extensive restructuring and redevelopment. Both are reengineering.

*Although code restructuring can alleviate immediate problems associated with debugging or small changes, it is not reengineering. Real benefit is achieved only when data and architecture are restructured.*

logic. The objective is to take "spaghetti-bowl" code and derive a procedural design that conforms to the structured programming philosophy (Chapter 11).

Other restructuring techniques have also been proposed for use with reengineering tools. A resource exchange diagram maps each program module and the resources (data types, procedures, and variables) that are exchanged between it and other modules. By creating representations of resource flow, the program architecture can be restructured to achieve minimum coupling among modules.

### 31.4.2 Data Restructuring

Before data restructuring can begin, a reverse engineering activity called *analysis of source code* must be conducted. All programming language statements that contain data definitions, file descriptions, I/O, and interface descriptions are evaluated. The intent is to extract data items and objects, to get information on data flow, and to understand the existing data structures that have been implemented. This activity is sometimes called *data analysis* [RIC89].

Once data analysis has been completed, *data redesign* commences. In its simplest form, a *data record standardization* step clarifies data definitions to achieve consistency among data item names or physical record formats within an existing data structure or file format. Another form of redesign, called *data name rationalization,* ensures that all data naming conventions conform to local standards and that aliases are eliminated as data flow through the system.

When restructuring moves beyond standardization and rationalization, physical modifications to existing data structures are made to make the data design more effective. This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.

SOFTWARE TOOLS

#### *Software Restructuring*

**Objective:** The objective of restructuring tools is to transform older unstructured computer software into modern programming languages and design structures.

**Mechanics:** In general, source code is input and transformed into a better structured program. In some cases, the transformation occurs within the same programming language. In other cases, an older programming language is transformed into a more modern language.

**Representative Tools[8]**

*DMS Software Reengineering Toolkit,* developed by Semantic Design (www.semdesigns.com), provides a variety of restructuring capabilities for COBOL, C/C++, Java, FORTRAN 90, and VHDL.

*FORESYS,* developed by Simulog (www.simulog.fr), analyzes and transforms programs written in FORTRAN.

*Function Encapsulation Tool,* developed at Wayne State University (www.cs.wayne.edu/~vip/RefactoringTools/), refactors older C programs into C++.

*plusFORT,* developed by Polyhedron (www.polyhedron.com), is a suite of FORTRAN tools that contains capabilities for restructuring poorly designed FORTRAN programs into the modern FORTRAN or C standard.

8 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.

## 31.5 Forward Engineering

A program with control flow that is the graphic equivalent of a bowl of spaghetti, with "modules" that are 2000 statements long, with few meaningful comment lines in 290,000 source statements and no other documentation must be modified to accommodate changing user requirements. We have the following options:

**What options exist when we're faced with a poorly designed and implemented program?**

1. We can struggle through modification after modification, fighting the implicit design and source code to implement the necessary changes.
2. We can attempt to understand the broader inner workings of the program in an effort to make modifications more effectively.
3. We can redesign, recode, and test those portions of the software that require modification, applying a software engineering approach to all revised segments.
4. We can completely redesign, recode, and test the program, using reengineering tools to assist us in understanding the current design.

There is no single "correct" option. Circumstances may dictate the first option even if the others are more desirable.

Rather than waiting until a maintenance request is received, the development or support organization uses the results of inventory analysis to select a program that (1) will remain in use for a preselected number of years, (2) is currently being used successfully, and (3) is likely to undergo major modification or enhancement in the near future. Then, option 2, 3, or 4 is applied.

This *preventative maintenance* approach was pioneered by Miller [MIL81] under the title *structured retrofit.* This concept is defined as "the application of today's methodologies to yesterday's systems to support tomorrow's requirements."

At first glance, the suggestion that we redevelop a large program when a working version already exists may seem quite extravagant. Before passing judgment, consider the following points:

*Reengineering is a lot like getting your teeth cleaned. You can think of a thousand reasons to delay it, and you'll get away with procrastinating for quite a while. But eventually, your delaying tactics will come back to cause pain.*

1. The cost to maintain one line of source code may be 20 to 40 times the cost of initial development of that line.
2. Redesign of the software architecture (program and/or data structure), using modern design concepts, can greatly facilitate future maintenance.
3. Because a prototype of the software already exists, development productivity should be much higher than average.
4. The user now has experience with the software. Therefore, new requirements and the direction of change can be ascertained with greater ease.
5. Automated tools for reengineering will facilitate some parts of the job.
6. A complete software configuration (documents, programs, and data) will exist upon completion of preventive maintenance.

When a software development organization sells software as a product, preventive maintenance is seen in "new releases" of a program. A large in-house software developer (e.g., a business systems software development group for a large consumer products company) may have 500–2000 production programs within its domain of responsibility. These programs can be ranked by importance and then reviewed as candidates for preventive maintenance.

The forward engineering process applies software engineering principles, concepts, and methods to recreate an existing application. In most cases, forward engineering does not simply create a modern equivalent of an older program. Rather, new user and technology requirements are integrated into the reengineering effort. The redeveloped program extends the capabilities of the older application.

### 31.5.1 Forward Engineering for Client/Server Architectures

Over the past few decades, many mainframe applications have been reengineered to accommodate client/server architectures (including WebApps). In essence, centralized computing resources (including software) are distributed among many client platforms. Although a variety of different distributed environments can be designed, the typical mainframe application that is reengineered into a client/server architecture has the following features:

- Application functionality migrates to each client computer.
- New GUI interfaces are implemented at the client sites.
- Database functions are allocated to the server.
- Specialized functionality (e.g., compute-intensive analysis) may remain at the server site.
- New communications, security, archiving, and control requirements must be established at both the client and server sites.

*In some cases, migration to a client/server architecture should be approached not as reengineering, but as a new development effort. Reengineering enters the picture only when specific functionality from the old system is to be integrated into the new architecture.*

It is important to note that the migration from mainframe to client/server computing requires both business and software reengineering. In addition, an "enterprise network infrastructure" [JAY94] should be established.

Reengineering for client/server applications begins with a thorough analysis of the business environment that encompasses the existing mainframe. Three layers of abstraction can be identified. The *database layer* sits at the foundation of a client/server architecture and manages transactions and queries from client applications. Yet these transactions and queries must be controlled within the context of a set of business rules (defined by an existing or reengineered business process). Client applications provide targeted functionality to the user community.

The functions of the existing database management system and the data architecture of the existing database must be reverse engineered as a precursor to the redesign of the database layer. In some cases a new data model (Chapter 8) is created. In every case, the client/server database is reengineered to ensure that transactions

are executed in a consistent manner, that all updates are performed only by authorized users, that core business rules are enforced (e.g., before a vendor record is deleted, the server ensures that no related accounts payable, contracts, or communications exist for that vendor), that queries can be accommodated efficiently, and that full archiving capability has been established.

The *business rules layer* represents software that is resident at both the client and the server. This software performs control and coordination tasks to ensure that transactions and queries between the client application and the database conform to the established business process.

The *client applications layer* implements business functions that are required by specific groups of end-users. In many instances, a mainframe application is segmented into a number of smaller, reengineered desktop applications. Communication among the desktop applications (when necessary) is controlled by the business rules layer.

A comprehensive discussion of client/server software design and reengineering is best left to books dedicated to the subject. The interested reader should see [VAN02], [COU00], and [ORF99].

### 31.5.2 Forward Engineering for Object-Oriented Architectures

Object-oriented software engineering has become the development paradigm of choice for many software organizations. But what about existing applications that were developed using conventional methods? In some cases, the answer is to leave such applications "as is." In others, older applications must be reengineered so that they can be easily integrated into large, object-oriented systems.

Reengineering conventional software into an object-oriented implementation uses many of the same techniques discussed in Part 2 of this book. First, the existing software is reverse engineered so that appropriate data, functional, and behavioral models can be created. If the reengineered system extends the functionality or behavior of the original application, use-cases (Chapters 7 and 8) are created. The data models created during reverse engineering are then used in conjunction with CRC modeling (Chapter 8) to establish the basis for the definition of classes. Class hierarchies, object-relationship models, object-behavior models, and subsystems are defined, and object-oriented design commences.

As object-oriented forward engineering progresses from analysis to design, a CBSE process model (Chapter 30) can be invoked. If the old application exists within a domain that is already populated by many object-oriented applications, it is likely that a robust component library exists and can be used during forward engineering.

For those classes that must be engineered from scratch, it may be possible to reuse algorithms and data structures from the existing conventional application. However, these must be redesigned to conform to the object-oriented architecture.

### 31.5.3 Forward Engineering User Interfaces

As applications migrate from the mainframe to the desktop, users are no longer willing to tolerate arcane, character-based user interfaces. In fact, a significant portion of all effort expended in the transition from mainframe to client/server computing can be spent in the reengineering of client application user interfaces.

Merlo and his colleagues [MER95] suggest the following model for reengineering user interfaces:

*What steps should we follow to reengineer a user interface?*

1. *Understand the original interface and the data that move between it and the remainder of the application.* The intent is to understand how other elements of a program interact with existing code that implements the interface. If a new GUI is to be developed, the data that flow between the GUI and the remaining program must be consistent with the data that currently flow between the character-based interface and the program.
2. *Remodel the behavior implied by the existing interface into a series of abstractions that have meaning in the context of a GUI.* Although the mode of interaction may be radically different, the business behavior exhibited by users of the old and new interfaces (when considered in terms of a usage scenario) must remain the same. A redesigned interface must still allow a user to exhibit the appropriate business behavior. For example, when a database query is to be made, the old interface may require a long series of text-based commands to specify the query. The reengineered GUI may streamline the query to a small sequence of mouse picks, but the intent and content of the query remain unchanged.

**WebRef**

A 300+ page handbook on reengineering patterns (developed as part of the FAMOOS ESPRIT project can be downloaded from **www.iam.unibe.ch/~scg/Archive/famoos/patterns/index3.html.**

3. *Introduce improvements that make the mode of interaction more efficient.* The ergonomic failings of the existing interface are studied and corrected in the design of the new GUI.
4. *Build and integrate the new GUI.* The existence of class libraries and automated tools can reduce the effort required to build the GUI significantly. However, integration with existing application software can be more time consuming. Care must be taken to ensure that the GUI does not propagate adverse side effects into the remainder of the application.

> "You can pay a little now, or you can pay a lot more later."
>
> **Sign in an auto dealership suggesting a tune up**

## 31.6 The Economics of Reengineering

In a perfect world, every unmaintainable program would be retired immediately, to be replaced by high-quality, reengineered applications developed using modern software engineering practices. But we live in a world of limited resources. Reengineering

drains resources that can be used for other business purposes. Therefore, before an organization attempts to reengineer an existing application, it should perform a cost/benefit analysis.

A cost/benefit analysis model for reengineering has been proposed by Sneed [SNE95]. Nine parameters are defined:

$P_1$ = current annual maintenance cost for an application
$P_2$ = current annual operation cost for an application
$P_3$ = current annual business value of an application
$P_4$ = predicted annual maintenance cost after reengineering
$P_5$ = predicted annual operations cost after reengineering
$P_6$ = predicted annual business value after reengineering
$P_7$ = estimated reengineering costs
$P_8$ = estimated reengineering calendar time
$P_9$ = reengineering risk factor ($P_9 = 1.0$ is nominal)
$L$ = expected life of the system

The cost associated with continuing maintenance of a candidate application (i.e., reengineering is not performed) can be defined as

$$C_{\text{maint}} = [P_3 - (P_1 + P_2)] \times L \qquad (31\text{-}1)$$

The costs associated with reengineering are defined using the following relationship:

$$C_{\text{reeng}} = [P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9)] \qquad (31\text{-}2)$$

Using the costs presented in Equations (31-1) and (31-2), the overall benefit of reengineering can be computed as

$$\text{cost benefit} = C_{\text{reeng}} - C_{\text{maint}} \qquad (31\text{-}3)$$

The cost/benefit analysis presented in the equations can be performed for all high-priority applications identified during inventory analysis (Section 31.2.2). Those applications that show the highest cost/benefit can be targeted for reengineering, while work on others can be postponed until resources are available.

## 31.7 SUMMARY

Reengineering occurs at two different levels of abstraction. At the business level, reengineering focuses on the business process with the intent of making changes to improve competitiveness in some area of the business. At the software level, reengineering examines information systems and applications with the intent of restructuring or reconstructing them so that they exhibit higher quality.

Business process reengineering defines business goals, identifies and evaluates existing business processes (in the context of defined goals), specifies and designs revised processes, and prototypes, refines, and instantiates them within a business.

BPR has a focus that extends beyond software. The result of BPR is often the definition of ways in which information technologies can better support the business.

Software reengineering encompasses a series of activities that include inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability—programs that will be viable well into the twenty-first century.

Inventory analysis enables an organization to assess each application systematically, with the intent of determining which are candidates for reengineering. Document restructuring creates a framework of documentation that is necessary for the long-term support of an application. Reverse engineering is the process of analyzing a program in an effort to extract data, architectural, and procedural design information. Finally, forward engineering reconstructs a program using modern software engineering practices and information learned during reverse engineering.

The cost/benefit of reengineering can be determined quantitatively. The cost of the status quo, that is, the cost associated with ongoing support and maintenance of an existing application, is compared to the projected costs of reengineering and the resultant reduction in maintenance costs. In almost every case in which a program has a long life and currently exhibits poor maintainability, reengineering represents a cost-effective business strategy.

## REFERENCES

[CAN72] Canning, R., "The Maintenance 'Iceberg'," *EDP Analyzer,* vol. 10, no. 10, October 1972.

[CAS88] "Case Tools for Reverse Engineering," *CASE Outlook,* CASE Consulting Group, vol. 2, no. 2, 1988, pp. 1–15.

[CHI90] Chikofsky, E. J., and J. H. Cross, II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software,* January 1990, pp. 13–17.

[COU00] Coulouris, G., J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design,* 3rd ed., Addison-Wesley, 2000.

[DAV90] Davenport, T. H., and J. E. Young, "The New Industrial Engineering: Information Technology and Business Process Redesign," *Sloan Management Review,* Summer 1990, pp. 11–27.

[DEM95] DeMarco, T., "Lean and Mean," *IEEE Software,* November 1995, pp. 101–102.

[HAM90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate," *Harvard Business Review,* July–August 1990, pp. 104–112.

[HAN93] Manna, M., "Maintenance Burden Begging for a Remedy," *Datamation,* April 1993, pp. 53–63.

[JAY94] Jaychandra, Y., *Re-engineering the Networked Enterprise,* McGraw-Hill, 1994.

[MER93] Merlo, E., et al., "Reverse Engineering of User Interfaces," *Proc. Working Conference on Reverse Engineering,* IEEE, Baltimore, May 1993, pp. 171–178.

[MER95] Merlo, E., et al., "Reengineering User Interfaces," *IEEE Software,* January 1995, pp. 64–73.

[MIL81] Miller, J., in *Techniques of Program and System Maintenance,* (G. Parikh, ed.) Winthrop Publishers, 1981.

[ORF99] Orfali, R., D. Harkey, and J. Edwards, *Client/Server Survival Guide,* 3rd ed., Wiley, 1999.

[OSB90] Osborne, W. M., and E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle," *IEEE Software,* January 1990, pp. 10–11.

[PRE94] Premerlani, W. J., and M. R. Blaha, "An Approach for Reverse Engineering of Relational Databases," *CACM,* vol. 37, no. 5, May 1994, pp. 42–49.

[RIC89] Ricketts, J. A., J. C. DelMonaco, and M. W. Weeks, "Data Reengineering for Application Systems," *Proc. Conf. Software Maintenance—1989*, IEEE, 1989, pp. 174–179.

[SNE95] Sneed, H., "Planning the Reengineering of Legacy Systems," *IEEE Software*, January 1995, pp. 24–25.

[STE93] Stewart, T. A., "Reengineering: The Hot New Managing Tool," *Fortune*, August 23, 1993, pp. 41–48.

[SWA76] Swanson, E. B., "The Dimensions of Maintenance," *Proc. Second Intl. Conf. Software Engineering*, IEEE, October 1976, pp. 492–497.

[VAN02] Van Steen, M., and A. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Prentice-Hall, 2002.

[WAR74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.

## Problems and Points to Ponder

**31.1.** Research the literature and/or Internet sources to find one or more papers that discuss case studies of mainframe to client/server reengineering. Present a summary.

**31.2.** How would you determine $P_4$ through $P_7$ in the cost-benefit model presented in Section 31.6?

**31.3.** Your instructor will select one of the programs that everyone in the class has developed during this course. Exchange your program randomly with someone else in the class. Do not explain or walk through the program. Now, implement an enhancement (specified by your instructor) in the program you have received.

a. Perform all software engineering tasks including a brief walkthrough (but not with the author of the program).
b. Keep careful track of all errors encountered during testing.
c. Discuss your experiences in class.

**31.4.** Using information obtained via the Internet, present characteristics of three reverse engineering tools to your class.

**31.5.** Explore the inventory analysis checklist presented at the SEPA Web site and attempt to develop a quantitative software rating system that could be applied to existing programs in an effort to pick candidate programs for reengineering. Your system should extend beyond economic analysis presented in Section 31.6.

**31.6.** Some people believe that artificial intelligence technology will increase the abstraction level of the reverse engineering process. Do some research on this subject (i.e., the use of AI for reverse engineering), and write a brief paper that takes a stand on this point.

**31.7.** Suggest alternatives to paper and ink or conventional electronic documentation that could serve as the basis for document restructuring. (Hint: Think of new descriptive technologies that could be used to communicate the intent of the software.)

**31.8.** Consider any job that you've held in the last five years. Describe the business process in which you played a part. Use the BPR model described in Section 31.1.3 to recommend changes to the process in an effort to make it more efficient.

**31.9.** Why is completeness difficult to achieve as abstraction level increases?

**31.10.** There is a subtle difference between restructuring and forward engineering. What is it?

**31.11.** Do some research on the efficacy of business process reengineering. Present pro and con arguments for this approach.

**31.12.** Why must interactivity increase if completeness is to increase?

## Further Readings and Information Sources

Like many hot topics in the business community, the hype surrounding business process reengineering has given way to a more pragmatic view of the subject. Hammer and Champy (*Reengineering the Corporation,* HarperBusiness, revised edition, 2001) precipitated early interest with their best-selling book. Later, Hammer (*Beyond Reengineering: How the Processed-Centered Organization Is Changing Our Work and Our Lives,* HarperCollins 1997) refined his view by focusing on "process-centered" issues.

Books by Smith and Fingar (*Business Process Management (BPM): The Third Wave,* Meghan-Kiffer Press, 2003), Jacka and Keller (*Business Process Mapping: Improving Customer Satisfaction,* Wiley, 2001), Sharp and McDermott (*Workflow Modeling,* Artech House, 2001), Andersen (*Business Process Improvement Toolbox,* American Society for Quality, 1999), and Harrington et al. (*Business Process Improvement Workbook,* McGraw-Hill, 1997), present case studies and detailed guidelines for BPR.

Feldmann (*The Practical Guide to Business Process Reengineering Using IDEF0,* Dorset House, 1998) discusses a modeling notation that assists in BPR. Berztiss (*Software Methods for Business Reengineering,* Springer, 1996) and Spurr et al. (*Software Assistance for Business Reengineering,* Wiley, 1994) discuss tools and techniques that facilitate BPR.

Secord and his colleagues (*Modernizing Legacy Systems,* Addison-Wesley, 2003), Ulrich (*Legacy Systems: Transformation Strategies,* Prentice-Hall, 2002), Valenti (*Successful Software Reengineering,* IRM Press, 2002), and Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software,* Fitzroy Dearborn Publishers, 1999) focus on strategies and practices for reengineering at a technical level. Miller (*Reengineering Legacy Software Systems,* Digital Press, 1998) "provides a framework for keeping application systems synchronized with business strategies and technology changes." Umar (*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies,* Prentice-Hall, 1997) provides worthwhile guidance for organizations that want to transform legacy systems into a Web-based environment. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems,* Prentice-Hall, 1996) discusses the bridge between BPR and information technology. Aiken (*Data Reverse Engineering,* McGraw-Hill, 1996) discusses how to reclaim, reorganize, and reuse organizational data. Arnold (*Software Reengineering,* IEEE Computer Society Press, 1993) has put together an excellent anthology of early papers that focus on software reengineering technologies.

A wide variety of information sources on software reengineering 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.**

CHAPTER

# 32 THE ROAD AHEAD

**KEY CONCEPTS**

- data
- ethics
- information
- knowledge
- people
- process
- scope of change
- software revisited
- technology trends

In the 31 chapters that have preceded this one, we explored a process for software engineering. We presented both management procedures and technical methods, basic principles and specialized techniques, people-oriented activities and tasks that are amenable to automation, paper and pencil notation, and software tools. We argued that measurement, discipline, and an overriding focus on quality will result in software that meets the customer's needs, software that is reliable, software that is maintainable, software that is better. Yet, we have never promised that software engineering is a panacea.

As we continue our journey into a new century, software and systems technologies remain a challenge for every software professional and every company that builds computer-based systems. Although he wrote these words with a twentieth century outlook, Max Hopper [HOP90] accurately describes the current state of affairs:

> Because changes in information technology are becoming so rapid and unforgiving, and the consequences of falling behind are so irreversible, companies will either master the technology or die . . . Think of it as a technology treadmill. Companies will have to run harder and harder just to stay in place.

**QUICK LOOK**

**What is it?** The future is never easy to predict—pundits, talking heads, and industry experts not-withstanding. The road ahead is littered with the carcasses of exciting new technologies that never really made it (despite the hype) and is often shaped by more modest technologies that somehow modify the direction and width of the thoroughfare. Therefore, we won't try to predict the future. Rather we'll discuss some of the issues that you'll need to consider to understand how software and software engineering will change in the years ahead.

**Who does it?** Everyone!

**Why is it important?** Why did ancient kings hire soothsayers? Why do major multinational corporations hire consulting firms and think tanks to prepare forecasts? Why does a substantial percentage of the public read horoscopes? We want to know what's coming so we can ready ourselves.

**What are the steps?** There is no formula for predicting the road ahead. We attempt to do this by collecting data, organizing it to provide useful information, examining subtle associations to extract knowledge, and from this knowledge, suggest probable occurrences that predict how things will be at some future time.

**What is the work product?** A view of the near-term future that may or may not be correct.

**How do I ensure that I've done it right?** Predicting the road ahead is an art, not a science. In fact, it's quite rare when a serious prediction about the future is absolutely right or unequivocally wrong (with the exception, thankfully, of predictions of the end of the world). We look for trends and try to extrapolate them ahead in time. We can assess the correctness of the extrapolation only as time passes.

Changes in software engineering technology are indeed "rapid and unforgiving," while at the same time progress is often quite slow. By the time a decision is made to adopt a new method (or a new tool), conduct the training necessary to understand its application, and introduce the technology into the software development culture, something newer (and even better) has come along, and the process begins anew.

In this chapter, we examine the road ahead. Our intent is not to explore every area of research the holds promise. Nor is it to gaze into a "crystal ball" and prognosticate about the future. Rather, we explore the scope of change and the way in which change itself will affect the software engineering process in the years ahead.

## 32.1 The Importance of Software—Revisited

The importance of computer software can be stated in many ways. In Chapter 1, software was characterized as a differentiator. The function delivered by software differentiates products, systems, and services and provides competitive advantage in the marketplace. But software is more than a differentiator. The programs, documents, and data that are software help to generate the most important commodity that any individual, business, or government can acquire—*information*. Pressman and Herron [PRE91] describe software in the following way:

> Computer software is one of only a few key technologies that will have a significant impact on nearly every aspect of modern society . . . It is a mechanism for automating business, industry, and government, a medium for transferring new technology, a method of capturing valuable expertise for use by others, a means for differentiating one company's products from its competitors, and a window into a corporation's collective knowledge. Software is pivotal to nearly every aspect of business. But in many ways, software is also a hidden technology. We encounter software (often without realizing it) when we travel to work, make any retail purchase, stop at the bank, make a phone call, visit the doctor, or perform any of the hundreds of day-to-day activities that reflect modern life.

The pervasiveness of software leads us to a simple conclusion: Whenever a technology has a broad impact—an impact that can save lives or endanger them, build businesses or destroy them, inform government leaders or mislead them—it must be handled with care.

> **"Predictions are very difficult to make, especially when they deal with the future."**
>
> **Mark Twain**

## 32.2 The Scope of Change

The changes in computing over the past 50 years have been driven by advances in the hard sciences—physics, chemistry, materials science, and engineering. This trend will continue during the first quarter of the twenty-first century. The impact of new technologies is pervasive—on communications, energy, healthcare, transportation, entertainment, economics, manufacturing, and warfare, to name only a few.

INFO

### *Technologies to Watch*

The editors of *PC Magazine* [PCM03] prepare an annual "Future Tech" issue that "[sorts] through all the chatter (there's a lot of it) to identify 20 of the most promising technologies of tomorrow." The technologies noted run the gamut from healthcare to warfare. However, it's interesting to note that software and software engineering have a significant role to play in every one, either as an enabler for the technology or an integral part of it. The following represents a sampling of the technologies noted:

**Carbon nanotubes**—with a tiny graphite-like structure, carbon nanotubes can serve as wires to transmit signals from one point to another and as transistors, using signal changes to store information. These devices show promise for use in the development of smaller, faster, lower energy, and less expensive electronic devices (e.g., microprocessors, memory, displays).

**Biosensors**—external or implantable microelectronic sensors are already in use for detecting everything from chemical agents in the air we breathe to blood levels in a cardiac patient. As these sensors become more sophisticated, they may be implanted in medical patients to monitor a variety of health-related conditions or attached to a soldier's uniform to monitor the presence of biological and chemical weapons.

**OLED displays**—An OLED "uses a carbon-based designer molecule that emits light when an electric current passes through it. Piece lots of molecules together and you've got a superthin display of stunning quality—no power-draining backlight required." [PCM03] The result—ultra-thin displays that can be rolled up or folded, sprayed onto a curved surface, or otherwise adapted to a specific environment.

**Grid computing**—this technology (available today) creates a network that taps the billions of unused CPU cycles from every machine on the network and allows exceedingly complex computing jobs to be completed without a dedicated supercomputer. For a real-life example encompassing over 4.5 million computers, visit http://setiathome.berkeley.edu/.

**Cognitive machines**—the 'holy grail' in the robotics field is to develop machines that are aware of their environment, that can "pick up on cues, respond to ever-changing situations, and interact with people naturally" [PCM03]. Cognitive machines are still in the early stages of development, but the potential (if ever achieved) is enormous.

**WebRef**
For predictions about the future of technology and other matters, see **www.futurefacing.com.**

Over the longer term, revolutionary advances in computing may well be driven by soft sciences—human psychology, sociology, philosophy, anthropology, and others. The gestation period for the computing technologies that may be derived from these disciplines is very difficult to predict, but early influences have already begun (e.g., the communities—an anthropological construct—of users that are an off-shoot of peer-to-peer networks).

The influence of the soft sciences may help mold the direction of computing research in the hard sciences. For example, the design of future computers may be guided more by an understanding of brain physiology than an understanding of conventional microelectronics.

In the shorter term, the changes that will affect software engineering over the next decade will be influenced by four simultaneous sources: (1) the people who do the work, (2) the process that they apply, (3) the nature of information, and (4) the underlying computing technology. In the sections that follow, each of these components—people, the process, information, and the technology—is examined in more detail.

## 32.3 People and the Way They Build Systems

The software required for high-technology systems becomes more and more complex with each passing year, and the size of resultant programs increases proportionally. The rapid growth in the size of the "average" program would present us with few problems if it wasn't for one simple fact: As program size increases, the number of people who must work on the program must also increase.

Experience indicates that as the number of people on a software project team increases, the overall productivity of the group may suffer. One way around this problem is to create a number of software engineering teams, thereby compartmentalizing people into individual working groups. However, as the number of software engineering teams grows, communication between them becomes as difficult and time consuming as communication between individuals. Worse, communication (between individuals or teams) tends to be inefficient—that is, too much time is spent transferring too little information content, and all too often, important information "falls into the cracks."

> **"Future shock [is] the shattering stress and disorientation that we induce in individuals by subjecting them to too much change in too short a period of time."**
>
> **Alvin Toffler**

If the software engineering community is to deal effectively with the communication dilemma, the road ahead for software engineers must include radical changes in the way individuals and teams communicate with one another. E-mail, Web sites, and centralized video conferencing are now commonplace as mechanisms for connecting a large number of people to an information network. The importance of these tools in the context of software engineering work cannot be overemphasized. With an effective electronic mail or instant messaging system, the problem encountered by a software engineer in New York City may be solved with the help of a colleague in Tokyo. In a very real sense, focused chat sessions and specialized newsgroups become knowledge repositories that allow the collective wisdom of a large group of technologists to be brought to bear on a technical problem or management issue.

Video personalizes the communication. At its best, it enables colleagues at different locations (or on different continents) to "meet" on a regular basis. But video also provides another benefit. It can be used as a repository for knowledge about the software and to train newcomers on a project.

> **"The proper artistic response to digital technology is to embrace it as a new window on everything that's eternally human, and to use it with passion, wisdom, fearlessness and joy."**
>
> **Ralph Lombreglia**

*More and more "nonprogrammers" are building their own (small) applications. This on-going trend is likely to accelerate into the future. Should these "civilians" apply the technology discussed in this book? Probably not. But they should adopt an agile software engineering philosophy, even if they don't adopt the practice.*

The evolution of intelligent agents will also change the work patterns of a software engineer by dramatically extending the capabilities of software tools. Intelligent agents will enhance the engineer's ability by cross-checking engineering work products using domain-specific knowledge, performing clerical tasks, doing directed research, and coördinating human-to-human communication.

Finally, the acquisition of knowledge is changing in profound ways. On the Internet, a software engineer can subscribe to newsgroups that focus on technology areas of immediate concern. A question posted within a newsgroup precipitates answers from other interested parties around the globe. The World Wide Web provides a software engineer with the world's largest library of research papers and reports, tutorials, commentary, and references in software engineering.

If past history is any indication, it is fair to say that people themselves will not change. However, the ways in which they communicate, the environment in which they work, the way in which they acquire knowledge, the methods and tools that they use, the discipline that they apply, and therefore, the overall culture for software development will change in significant and even profound ways.

## 32.4 The "New" Software Engineering Process

It is reasonable to characterize the first two decades of software engineering practice as the era of "linear thinking." Fostered by the classic life cycle model, software engineering was approached as a linear activity in which a series of sequential steps could be applied in an effort to solve complex problems. Yet, linear approaches to software development run counter to the way in which most systems are actually built. In reality, complex systems evolve iteratively, even incrementally. It is for this reason that a large segment of the software engineering community is moving toward agile, incremental models for software development.

Agile, incremental process models recognize that uncertainty dominates most projects, that timelines are often impossibly short, and that iteration provides the ability to deliver a partial solution, even when a complete product is not possible within the time allotted. Evolutionary models emphasize the need for incremental work products, risk analysis, planning and then plan revision, and customer feedback. In many instances, the software team applies an "agile manifesto" (Chapter 4) that emphasizes "individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation, and responding to change over following a plan" [BEC01].

> **"The best preparation for good work tomorrow is to do good work today."**
>
> **Elbert Hubbard**

Object technologies, coupled with component-based software engineering (Chapter 30), are a natural outgrowth of the trend toward incremental and evolu-

tionary process models. Both will have a profound impact on software development productivity and product quality. Component reuse provides immediate and compelling benefits. When reuse is coupled with CASE tools for application prototyping, program increments can be built far more rapidly than through the use of conventional approaches. Prototyping draws the customer into the process. Therefore, it is likely that customers and users will become much more involved in the development of software. This, in turn, may lead to higher end-user satisfaction and better software quality overall.

The rapid growth in network and multimedia technologies (e.g., the exponential increase in WebApps over the past decade) is changing both the software engineering process and its participants. Again, we encounter an agile, incremental paradigm that emphasizes immediacy, security, and aesthetics as well as more conventional software engineering concerns. Modern software teams (e.g., a Web engineering team) often meld technologists with content specialists (e.g., artists, musicians, videographers) to build an information source for a community of users that is both large and unpredictable. The software that has grown out of these technologies has already resulted in radical economic and cultural change. Although the basic concepts and principles discussed in this book are applicable, the software engineering process must adapt.

## 32.5 New Modes for Representing Information

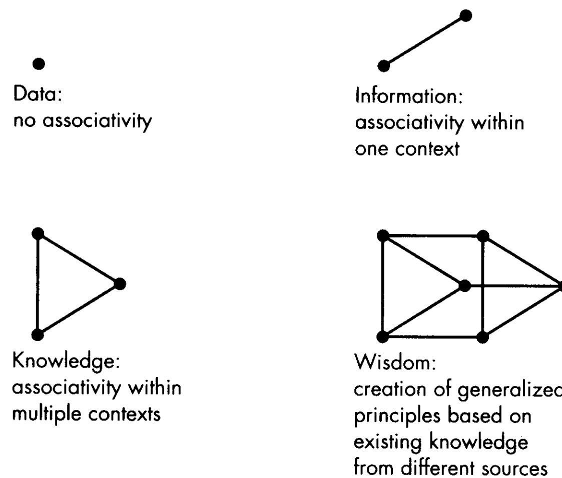
Over the history of computing, a subtle transition has occurred in the terminology that is used to describe software development work performed for the business community. Forty years ago, the term *data processing* was the operative phrase for describing the use of computers in a business context. Today, data processing has given way to another phrase—*information technology*—that implies the same thing but presents a subtle shift in focus. The emphasis is not merely to process large quantities of data but rather to extract meaningful information from this data. Obviously, this was always the intent, but the shift in terminology reflects a far more important shift in management philosophy.

When software applications are discussed today, the words *data* and *information* occur repeatedly. We encounter the word *knowledge* in some artificial intelligence applications, but its use is relatively rare. Virtually no one discusses *wisdom* in the context of software applications.

Data is raw information—collections of facts that must be processed to be meaningful. Information is derived by associating facts within a given context. Knowledge associates information obtained in one context with other information obtained in a different context. Finally, wisdom occurs when generalized principles are derived from disparate knowledge. Each of these four views of "information" is represented schematically in Figure 32.1.

To date, the vast majority of all software has been built to process data or information. Software engineers are now equally concerned with systems that process

**FIGURE 32.1**
**An "information" spectrum**




knowledge.[1] Knowledge is two-dimensional. Information collected on a variety of related and unrelated topics is connected to form a body of fact that we call *knowledge.* The key is our ability to associate information from a variety of different sources that may not have any obvious connection and combine it in a way that provides us with some distinct benefit.

> **"Wisdom is the power that enables us to use knowledge for the benefit of ourselves and others."**
> **Thomas J. Watson**

To illustrate the progression from data to knowledge, consider census data indicating that the birthrate in 1996 in the United States was 4.9 million. This number represents a data value. Relating this piece of data with birthrates for the preceding 40 years, we can derive a useful piece of information—aging "baby boomers" of the 1950s and early 1960s made a last gasp effort to have children prior to the end of their child-bearing years. In addition "gen-Xers" began their childbearing years. The census data can then be connected to other seemingly unrelated pieces of information. For example, the current number of elementary school teachers who will retire during the next decade, the number of college students graduating with degrees in primary and secondary education, the pressure on politicians to hold down taxes and therefore limit pay increases for teachers.

All of these pieces of information can be combined to formulate a representation of knowledge—there will be significant pressure on the education system in the United States in the first decade of the twenty-first century, and this pressure will

1 The rapid growth of data mining and data warehousing technologies reflect this growing trend.

continue for over a decade. Using this knowledge, a business opportunity may emerge. There may be significant opportunity to develop new modes of learning that are more effective and less costly than current approaches.

The road ahead for software leads to systems that process knowledge. We have been processing data using computers for over 50 years and extracting information for more than three decades. One of the most significant challenges facing the software engineering community is to build systems that take the next step along the spectrum—systems that extract knowledge from data and information in a way that is practical and beneficial.

## 32.6 TECHNOLOGY AS A DRIVER

The people who build and use software, the software engineering process that is applied, and the information that is produced are all affected by advances in hardware and software technology. Historically, hardware has served as the technology driver in computing. A new hardware technology provides potential. Software builders then react to customer demands in an attempt to tap the potential.

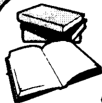
The road ahead for hardware technology is likely to progress along two parallel paths. Along one path, hardware technologies will continue to evolve at a rapid pace. With greater capacity provided by traditional hardware architectures, the demands on software engineers will continue to grow.

But the real changes in hardware technology may occur along another path. The development of nontraditional hardware architectures (e.g., carbon nanotubes, EUL microprocessors, cognitive machines, grid-computing) may cause radical changes in the kind of software that we build and fundamental changes in our approach to software engineering. Since these nontraditional approaches are only now maturing, it is difficult to determine which will have broad-based impact and even more difficult to predict how the world of software will change to accommodate them.

The road ahead for software engineering is driven by software technologies. Reuse and component-based software engineering offer the best opportunity for order of magnitude improvements in system quality and time to market. In fact, as time passes, the software business may begin to look very much like the hardware business of today. There may be vendors that build discrete devices (reusable software components), other vendors that build system components (e.g., a set of tools for human/computer interaction), and system integrators that provide solutions (products and custom-built systems) for the end-user.

Software engineering will change—of that we can be certain. But regardless of how radical the changes are, we can be assured that quality will never lose its importance and that effective analysis and design and competent testing will always have a place in the development of computer-based systems.

INFO



### Technology Trends

P. Cripwell Associates (www.jpcripwell.com), a consulting firm specializing in knowledge management and information engineering, discusses five technology drivers that will influence technology directions in the coming years:

**Combination technologies.** When two important technologies are merged, the impact of the merged result is often greater than the sum of the impact of each taken separately. For example, GPS satellite technology, coupled with on-board computing capability, coupled with LCD display technologies has resulting in sophisticated automobile mapping systems. Technologies often evolve along separate paths, but significant business or societal impact occurs only when someone combines them to solve a problem.

**Data fusion.** The more data we acquire, the more data we need. More importantly, the more data we acquire, the more difficult it is to extract useful information. In fact, we often need to acquire still more data to understand (1) what data are important; what data are relevant to a particular need or source, and what data should be used for decision making. This is the data fusion problem. J. P. Cripwell uses an advanced automobile traffic monitoring system as an example. Digital speed sensors (in the roadway) and digital cameras sense an accident. The severity of the accident must be determined (via camera?). Based on severity, the monitoring system must contact police, fire, or ambulance; traffic must be rerouted; media (radio) must broadcast warnings; and individual cars (if equipped with digital sensors or wireless communication) must be informed. To accomplish this, a variety of decisions, based on data acquired from the monitoring system (data fusion), must be made.

**Technology push.** In years past, a problem surfaced and technology was developed to solve it. Because the problem was evident to many people, the market for the new technology was well-defined. Today, some technologies evolve as solutions looking for problems. A market must be pushed to recognize that it needs the new technology (e.g., mobile phones, PDAs). As people recognize the need, the technology accelerates, improves, and often morphs as combination technologies evolve.

**Networking and serendipity.** In this context networking implies connections between people or between people and information. As the network grows, the likelihood of synergy between two network nodes (e.g., people, information sources) also grows. A chance connection (serendipity) can lead to inspiration and a new technology or application.

**Information overload.** A vast sea of information is accessible by anyone with an Internet connection. The problem, of course, is to find the right information, determine its validity, and then translate it into practical application at a business or personnel level.

## 32.7 The Software Engineer's Responsibility

Software engineering has evolved into a respected, worldwide profession. As professionals, software engineers should abide by a code of ethics that guides the work that they do and the products that they produce. An ACM/IEEE-CS Joint Task Force has produced a *Software Engineering Code of Ethics and Professional Practices* (Version 5.1). The code [ACM98] states:

**WebRef**
A complete discussion of the ACM/IEEE code of ethics can be found at **seeri.etsu.edu/Codes/default.shtm.**

> Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:
>
> 1. PUBLIC—Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.

5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.

8. SELF—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Although each of these eight principles is equally important, an overriding theme appears: a software engineer should work in the public interest. On a personal level, a software engineer should abide by the following rules:

- Never steal data for personal gain.
- Never distribute or sell proprietary information obtained as part of your work on a software project.
- Never maliciously destroy or modify another person's programs, files, or data.
- Never violate the privacy of an individual, a group, or an organization.
- Never hack into a system for sport or profit.
- Never create or promulgate a computer virus or worm.
- Never use computing technology to facilitate discrimination or harassment.

Over the past decade, certain members of the software industry have lobbied for protective legislation that [SEE03]:

1. Allows companies to release software without disclosing known defects;
2. Exempts developers from liability for any damages resulting from these known defects;
3. Constrains others from disclosing defects without permission from the original developer;
4. Allows the incorporation of "self-help" software within a product that can disable (via remote command) the operation of the product;
5. Exempts developers of software with "self-help" from damages should the software be disabled by a third party.

Like all legislation, debate often centers on issues that are political, not technological. However, many people (including this author) feel that protective legislation, if improperly drafted, conflicts with the software engineering code of ethics by indirectly exempting software engineers from their responsibility to produce high-quality software.

## 32.8 A Concluding Comment

It has been 25 years since the first edition of this book was written. I can still recall sitting at my desk as a young professor, writing the manuscript (by hand) for a book on a subject that few people cared about and even fewer really understood. I remember the rejection letters from publishers, who argued (politely, but firmly) that there would never be a market for a book on "software engineering." Luckily, McGraw-Hill decided to give it a try,[2] and the rest, as they say, is history.

Over the past 25 years, this book has changed dramatically—in scope, in size, in style, and in content. Like software engineering, it has grown and (I hope) matured over the years.

An engineering approach to the development of computer software is now conventional wisdom. Although debate continues on the "right paradigm," the importance of agility, the degree of automation, and the most effective methods, the underlying principles of software engineering are now accepted throughout the industry. Why, then, have we seen their broad adoption only recently?

The answer, I think, lies in the difficulty of technology transition and the cultural change that accompanies it. Even though most of us appreciate the need for an engineering discipline for software, we struggle against the inertia of past practice and face new application domains (and the developers who work in them) that appear ready to repeat the mistakes of the past.

To ease the transition we need many things—an agile, adaptable, and sensible software process; more effective methods; more powerful tools; better acceptance by practitioners and support from managers; and no small dose of education and "advertising." Software engineering has not had the benefit of massive advertising, but as time passes, the concept sells itself. In a way, this book is an "advertisement" for the technology.

You may not agree with every approach described in this book. Some of the techniques and opinions are controversial; others must be tuned to work well in different software development environments. It is my sincere hope, however, that *Software Engineering: A Practitioner's Approach* has delineated the problems we face, demonstrated the strength of software engineering concepts, and provided a framework of methods and tools.

As we move into the twenty-first century, software has become the most important product and the most important industry on the world stage. Its impact and im-

2 Actually, credit should go to Peter Freeman and Eric Munson, who convinced McGraw-Hill that it was worth a shot.

portance have come a long, long way. And yet, a new generation of software developers must meet many of the same challenges that faced earlier generations. Let us hope that the people who meet the challenge—software engineers—will have the wisdom to develop systems that improve the human condition.

## References

[ACM98] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice,* 1998, available at http://www.acm.org/serving/se/code.htm.

[BEC01] Beck, K., et al., "Manifesto for Agile Software Development," http://www.agilemanifesto.org/.

[BOL91] Bollinger, T., and C. McGowen, "A Critical Look at Software Capability Evaluations," *IEEE Software,* July 1991, pp. 25–41.

[GIL96] Gilb, T., "What Is Level Six?" *IEEE Software,* January 1996, pp. 97–98, 103.

[HOP90] Hopper, M. D., "Rattling SABRE, New Ways to Compete on Information," *Harvard Business Review,* May–June 1990.

[PAU93] Paulk, M., et al., *Capability Maturity Model for Software,* Software Engineering Institute, Carnegie Mellon University, 1993.

[PCM03] "Technologies to Watch," *PC Magazine,* July 2003, available at http://www.pcmag.com/article2/0,4149,1130591,00.asp.

[PRE91] Pressman, R. S., and S. R. Herron, *Software Shock,* Dorset House, 1991.

[SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates," 2003, available at http://seeri.etsu.edu/default.htm.

## Problems and Points to Ponder

**32.1.** Review the discussion of the agile, incremental process models in Chapter 4. Do some research, and collect recent papers on the subject. Summarize the strengths and weaknesses of agile paradigms based on experiences outlined in the papers.

**32.2.** Attempt to develop an example that begins with the collection of raw data and leads to acquisition of information, then knowledge, and finally, wisdom.

**32.3.** Get a copy of this week's major business and news magazines (e.g., *Newsweek, Time, Business Week*). List every article or news item that can be used to illustrate the importance of software.

**32.4.** Write a brief description of an ideal software engineer's development environment circa 2010. Describe the elements of the environment (hardware, software, and communications technologies) and their impact on quality and time to market.

**32.5.** One of the hottest software application domains is Web-based systems and applications. Discuss how people, communication, and process have to evolve to accommodate the development of "next generation" WebApps.

**32.6.** Provide specific examples that illustrate one of the eight software engineering ethics principles noted in Section 32.7.

## Further Readings and Information Sources

Books that discuss the road ahead for software and computing span a vast array of technical, scientific, economic, political, and social issues. Sterling (*Tomorrow Now,* Random House, 2002) reminds us that real progress is rarely orderly and efficient. Teich (*Technology and the Future,* Wadworth, 2002) presents thoughtful essays on the societal impact of technology and how changing