QUICK LOOK

encompasses 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.

What is the work product? A variety of reengineering work products (e.g., analysis models, design models, test procedures) are produced. The final output is the reengineered business process and/or the reengineered software that supports it.

How do I ensure that I've done it right? Use the same SQA practices that are applied in every software engineering process—formal technical reviews assess the analysis and design models, specialized reviews consider business applicability and compatibility, testing is applied to uncover errors in content, functionality, and interoperability.

and small. The nexus between business reengineering and software engineering lies in a "system view."

Software is often the realization of the business rules that Hammer discusses. As these rules change, software must also change. Today, major companies have tens of thousands of computer programs that support the "old business rules." As managers work to modify the rules to achieve greater effectiveness and competitiveness, software must keep pace. In some cases, this means the creation of major new computer-based systems.[1] But in many others, it means the modification or rebuilding of existing applications.

In this chapter, we examine reengineering in a top-down manner, beginning with a brief overview of business process reengineering and proceeding to a more detailed discussion of the technical activities that occur when software is reengineered.

30.1 BUSINESS PROCESS REENGINEERING

Quote:

"To face tomorrow with the thought of using the methods of yesterday is to envision life at a standstill."

James Bell

Business process reengineering (BPR) extends far beyond the scope of information technologies and software engineering. Among the many definitions (most somewhat abstract) that have been suggested for BPR is one published in Fortune magazine [STE93]: "the search for, and the implementation of, radical change in business process to achieve breakthrough results." But how is the search conducted, and how is the implementation achieved? More important, how can we ensure that the "radical change" suggested will in fact lead to "breakthrough results" instead of organizational chaos?

30.1.1 Business Processes

A *business process* is "a set of logically related tasks performed to achieve a defined business outcome" [DAV90]. Within the business process, people, equipment, mate-

1 The Web-based systems and applications discussed in Chapter 29 are contemporary examples.

rial resources, and business procedures are combined to produce a specified result. Examples of business processes include designing a new product, purchasing services and supplies, hiring a new employee, and paying suppliers. Each demands a set of tasks and each draws on diverse resources within the business.

Every business process has a defined customer—a person or group that receives the outcome (e.g., an idea, a report, a design, a product). In addition, business processes cross organizational boundaries. They require that different organizational groups participate in the "logically related tasks" that define the process.

As a software engineer, your reengineering work occurs at the bottom of this hierarchy. Be sure, however, that someone has given serious thought to the level above. If this hasn't been done, your work is at risk.

In Chapter 10, we noted that every system is actually a hierarchy of subsystems. A business is no exception. The overall business is segmented in the following manner:

The business
  business systems
    business process
      business subprocesses

Each business system (also called *business function*) is composed of one or more business processes, and each business process is defined by a set of subprocesses.

BPR can be applied at any level of the hierarchy, but as the scope of BPR broadens (i.e., as we move upward in the hierarchy), the risks associated with BPR grow dramatically. For this reason, most BPR efforts focus on individual processes or subprocesses.

30.1.2 Principles of Business Process Reengineering

In many ways, BPR is identical in focus and scope to business process engineering (Chapter 10). In an ideal setting, BPR should occur in a top-down manner, beginning with the identification of major business objectives and goals and culminating with a much more detailed specification of the tasks that define a specific business process.

Extensive information on business process reengineering can be found at **www.brint.com/BPR.htm**

Hammer [HAM90] suggests a number of principles that guide BPR activities when they begin at the top (business) level:

Organize around outcomes, not tasks. Many companies have compartmentalized business activities so that no single person (or organization) has responsibility (or control) of a business outcome. It such cases, it is difficult to determine the status of work and even more difficult to debug process problems if they do occur. BPR should design processes that avoid this problem.

Have those who use the output of the process perform the process. The intent of this recommendation is to allow those who need business output to control all of the variables that allow them to get the output in a timely manner. The fewer separate constituencies involved in a process, the smoother is the road to a rapid outcome.

Incorporate information processing work into the real work that produces the raw information. As IT becomes more distributed, it is possible to locate most information processing within the organization that produces the raw data. This localizes control, reduces communication time, and puts computing power in the hands of those that have a vested interest in the information that is produced.

Quote:

"As soon as we are shown the existence of something old in a new thing, we are pacified."

Friedrich Wilhelm Nietzsche

Treat geographically dispersed resources as though they were centralized. Computer-based communications have become so sophisticated that geographically diverse groups can be placed in the same "virtual office." For example, instead of running three engineering shifts at a single location, a global company can run one shift in Europe, a second shift in North America, and a third shift in Asia. In each case, engineers will work during daylight hours and communicate via high-bandwidth networks.

Link parallel activities instead of integrating their results. When different constituencies perform work in parallel, it is essential to design a process that demands continuing communication and coordination. Otherwise, integration problems are sure to result.

Put the decision point where the work is performed, and build control into the process. Using software design jargon, this principle suggests a flatter organizational architecture with reduced factoring.

Capture data once, at its source. Data should be stored on-line so that once collected it need never be re-entered.

Each of these principles represents a "big picture" view of BPR. Guided by these principles, business planners and process designers must begin process redesign. In the next section, we examine the process of BPR in a bit more detail.

30.1.3 A BPR Model

Like most engineering activities, business process reengineering is iterative. Business goals and the processes that achieve them must be adapted to a changing business environment. For this reason, there is no start and end to BPR—it is an evolutionary process. A model for business process reengineering is depicted in Figure 30.1. The model defines six activities:

Business definition. Business goals are identified within the context of four key drivers: *cost reduction, time reduction, quality improvement,* and *personnel development and empowerment.* Goals may be defined at the business level or for a specific component of the business.

Process identification. Processes that are critical to achieving the goals defined in the business definition are identified. They may then be ranked by importance, by need for change, or in any other way that is appropriate for the reengineering activity.

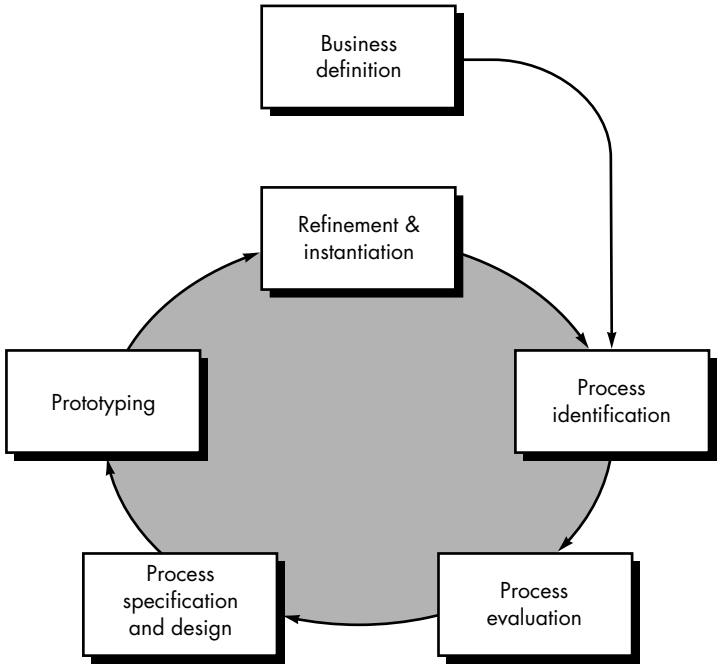

FIGURE 30.1
A BPR model

Process evaluation. The existing process is thoroughly analyzed and measured. Process tasks are identified; the costs and time consumed by process tasks are noted; and quality/performance problems are isolated.

Process specification and design. Based on information obtained during the first three BPR activities, use-cases (Chapter 11) are prepared for each process that is to be redesigned. Within the context of BPR, use-cases identify a scenario that delivers some outcome to a customer. With the use-case as the specification of the process, a new set of tasks (which conform to the principles noted in Section 30.2.1) are designed for the process.

Prototyping. A redesigned business process must be prototyped before it is fully integrated into the business. This activity "tests" the process so that refinements can be made.

Refinement and instantiation. Based on feedback from the prototype, the business process is refined and then instantiated within a business system.

These BPR activities are sometimes used in conjunction with workflow analysis tools. The intent of these tools is to build a model of existing workflow in an effort to better analyze existing processes. In addition, the modeling techniques commonly associated with business process engineering activities such as information strategy planning and business area analysis (Chapter 10) can be used to implement the first four activities described in the process model.

30.1.4 Words of Warning

It is not uncommon that a new business approach—in this case, BPR—is at first hyped as a panacea, and then criticized so severely that it becomes a pariah. Over the years, debate has raged about the efficacy of BPR (e.g., [BLE93], [DIC95]). In an excellent discussion of the case for and against BPR, Weisz [WEI95] summarizes the argument in the following way:

> It is tempting to bash BPR as another silver-bullet fad. From several points of view—systems thinking, peopleware, simple history—you'd have to predict high failure rates for the concept, rates which seem to be borne out by empirical evidence. For many companies, the silver bullet has apparently missed. For others, though, the reengineering effort has evidently been fruitful.

BPR can work, if it is applied by motivated, trained people who recognize that process reengineering is a continuous activity. If BPR is conducted effectively, information systems are better integrated into the business process. Reengineering older applications can be examined in the context of a broad-based business strategy, and priorities for software reengineering can be established intelligently.

But even if business reengineering is a strategy that is rejected by a company, software reengineering is something that *must* be done. Tens of thousands of legacy systems—applications that are crucial to the success of businesses large and small—are in dire need of refurbishing or rebuilding.

30.2 SOFTWARE REENGINEERING

The SEI offers a variety of software reengineering resources at **www.sei.cmu.edu/reengineering/**

The scenario is all too common: An application has served the business needs of a company for 10 or 15 years. During that time it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (the press of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software reengineering has been spawned by a software maintenance "iceberg" that has been building for more than three decades.

30.2.1 Software Maintenance

Thirty years ago, software maintenance was characterized [CAN72] as an "iceberg." We hope that what was immediately visible is all there is to it, but we know that an enormous mass of potential problems and cost lies under the surface. In the early 1970s, the maintenance iceberg was big enough to sink an aircraft carrier. Today, it could easily sink the entire navy!

The maintenance of existing software can account for over 60 percent of all effort expended by a development organization, and the percentage continues to rise as more software is produced [HAN93]. Uninitiated readers may ask why so much maintenance is required and why so much effort is expended. Osborne and Chikofsky [OSB90] provide a partial answer:

> Much of the software we depend on today is on average 10 to 15 years old. Even when these programs were created using the best design and coding techniques known at the time [and most were not], they were created when program size and storage space were principle concerns. They were then migrated to new platforms, adjusted for changes in machine and operating system technology and enhanced to meet new user needs—all without enough regard to overall architecture.
>
> The result is the poorly designed structures, poor coding, poor logic, and poor documentation of the software systems we are now called on to keep running . . .

The ubiquitous nature of change underlies all software work. Change is inevitable when computer-based systems are built; therefore, we must develop mechanisms for evaluating, controlling, and making modifications.

Quote:

"Program maintainability and program understandability are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain."

Gerald Berns

Upon reading the preceding paragraphs, a reader may protest: "but I don't spend 60 percent of my time fixing mistakes in the programs I develop." Software maintenance is, of course, far more than "fixing mistakes." We may define maintenance by describing four activities [SWA76] that are undertaken after a program is released for use. In Chapter 2, we defined four different maintenance activities: *corrective maintenance, adaptive maintenance, perfective maintenance* or *enhancement,* and *preventive maintenance* or *reengineering.* Only about 20 percent of all maintenance work is spent "fixing mistakes." The remaining 80 percent is spent adapting existing systems to changes in their external environment, making enhancements requested by users, and reengineering an application for future use. When maintenance is considered to encompass all of these activities, it is relatively easy to see why it absorbs so much effort.

30.2.2 A Software Reengineering Process Model

Reengineering takes time; it costs significant amounts of money; and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software reengineering.

A workable strategy is encompassed in a reengineering process model. We'll discuss the model later in this section, but first, some basic principles.

Reengineering is a rebuilding activity, and we can better understand the reengineering of information systems if we consider an analogous activity: the rebuilding of a house. Consider the following situation.

You have purchased a house in another state. You've never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt. How would you proceed?

- Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.
- Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).
- Before you start rebuilding be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.
- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

The steps noted here for a house are "obvious." Be sure that your consideration of legacy software applies steps that are just as obvious. Think about it. A lot more money is at stake.

Although these principles focus on the rebuilding of a house, they apply equally well to the reengineering of computer-based systems and applications.

To implement these principles, we apply a software reengineering process model that defines six activities, shown in Figure 30.2. In some cases, these activities occur in a linear sequence, but this is not always the case. For example, it may be that reverse engineering (understanding the internal workings of a program) may have to occur before document restructuring can commence.

The reengineering paradigm shown in the figure is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revisited. For any particular cycle, the process can terminate after any one of these activities.

Inventory analysis. Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

Inventory Analysis

It is important to note that the inventory should be revisited on a regular cycle. The status of applications (e.g., business criticality) can change as a function of time, and as a result, priorities for reengineering will shift.

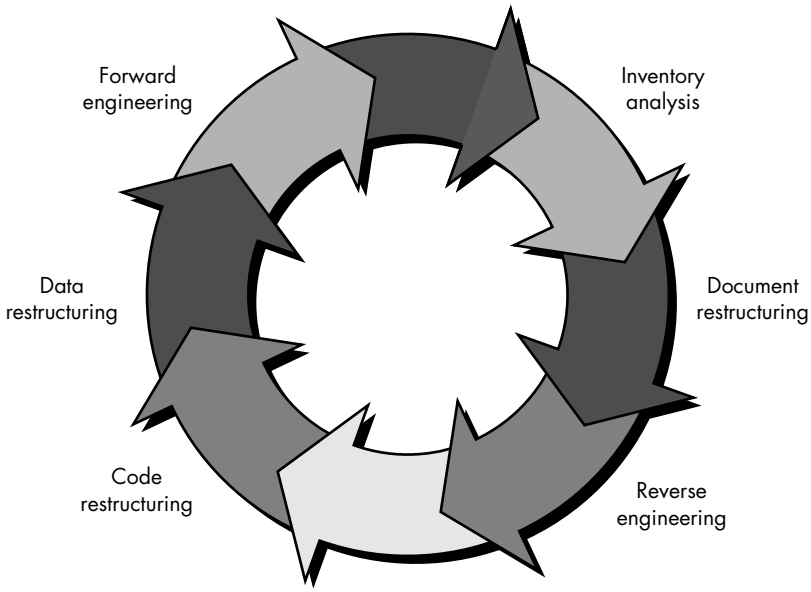

FIGURE 30.2
A software reengineering process model

Document restructuring. Weak documentation is the trademark of many legacy systems. But what do we do about it? What are our options?

1. *Creating documentation is far too time consuming. If the system works, we'll live with what we have.* In some cases, this is the correct approach. It is not possible to re-create documentation for hundreds of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change, let it be!
2. *Documentation must be updated, but we have limited resources. We'll use a "document when touched" approach.* It may not be necessary to fully redocument an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.
3. *The system is business critical and must be fully redocumented.* Even in this case, an intelligent approach is to pare documentation to an essential minimum.

Create only as much documentation as is required to enhance understanding of the software, not one page more.

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.

Code restructuring. The most common type of reengineering (actually, the use of the term *reengineering* is questionable in this case) is code restructuring. 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.

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

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 12). Data objects and attributes are identified, and existing data structures are reviewed for quality.

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 a 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 CASE vendors have introduced tools that provide a limited subset of these capabili-

ties 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.

30.3 REVERSE ENGINEERING

Reverse engineering conjures an image of the "magic slot." We feed an unstructured, undocumented source listing into the slot and out the other end comes 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 [CAS88].

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), data and control flow models (a relatively high level of abstraction), and entity relationship models (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.

XRef

The notation discussed here is explained in detail in Chapter 12.

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

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 data flow representations may also be derived, but it is far more difficult to develop a complete set of data flow diagrams or entity-relationship models.

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

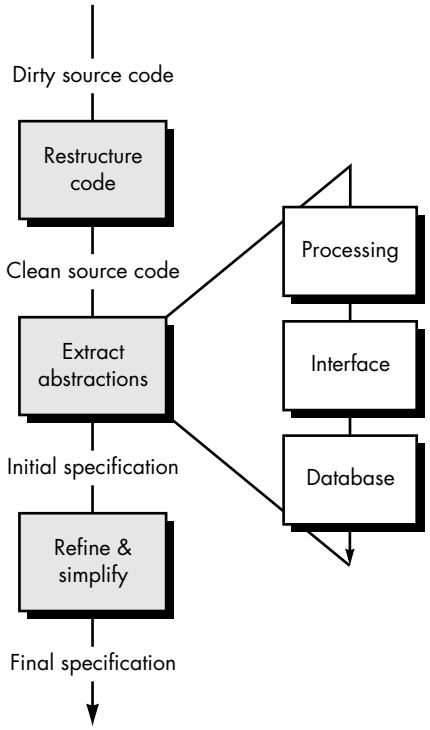

FIGURE 30.3 The reverse engineering process

fed to a reengineering tool that attempts to restructure or regenerate the old program.

The reverse engineering process is represented in Figure 30.3. Before reverse engineering activities can commence, unstructured ("dirty") source code is *restructured* (Section 30.4.1) so that it contains only the structured programming constructs.[2] This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.

WebRef

The Design Recovery and Program Understanding page provides useful resources for reengineering work: **wwwsel.iit.nrc.ca/projects/dr/dr.html**

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, extract 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.

30.3.1 Reverse Engineering to Understand Processing

The first real reverse engineering activity 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.

2 Code can be restructured automatically using a "restructuring engine"—a CASE tool that restructures source code.

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 created. In some situations, system, program and component specifications already exist. When this is the case, the specifications are reviewed for conformance to existing code.[3]

Quote:

"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. CASE tools are used to "parse" the semantics of existing code. The output of this process is then passed to restructuring and forward engineering tools to complete the reengineering process.

30.3.2 Reverse Engineering to Understand Data

Reverse engineering of data occurs at different levels of abstraction. 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 systemwide database.

Internal data structures. Reverse engineering techniques for internal program data focus on the definition of classes of objects.[4] 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.

---

3 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.

4 For a complete discussion of these object-oriented concepts, see Part Four of this book.

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

Breuer and Lano [BRE91] suggest the following approach for reverse engineering of classes:

1. Identify flags and local data structures within the program that record important information about global data structures (e.g., a file or database).
2. Define the relationship between flags and local data structures and the global data structures. For example, a flag may be set when a file is empty; a local data structure may serve as a buffer that contains the last 100 records acquired from a central database.
3. For every variable (within the program) that represents an array or file, list all other variables that have a logical connection to it.

These steps enable a software engineer to identify classes within the program that interact with the global data structures.

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:

What steps can be applied to reverse engineer an existing database structure?

1. **Build an initial object model.** The classes defined as part of the model may be acquired by reviewing records in a flat file database or tables in a relational schema. The items contained in records or tables become attributes of a class.
2. **Determine candidate keys.** The attributes are examined to determine whether they are used to point to another record or table. Those that serve as pointers become candidate keys.
3. **Refine the tentative classes.** Determine whether similar classes can be combined into a single class.
4. **Define generalizations.** Examine classes that have many similar attributes to determine whether a class hierarchy should be constructed with a generalization class at its head.
5. **Discover associations.** Use techniques that are analogous to the CRC approach (Chapter 21) to establish associations among classes.

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.

30.3.3 Reverse Engineering User Interfaces

Sophisticated GUIs have become 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 (UI), 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 12) 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 extracted from the code.

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 UI 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.

30.4 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 30.5).

Arnold [ARN89] defines a number of benefits that can be achieved when software is restructured:

What benefits are derived from restructuring?

- Programs have higher quality—better documentation, less complexity, and conformance to modern software engineering practices and standards.
- Frustration among software engineers who must work on the program is reduced, thereby improving productivity and making learning easier.
- Effort required to perform maintenance activities is reduced.
- Software is easier to test and debug.

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 modules and data need extensive modification.[5]

30.4.1 Code Restructuring

Code restructuring is performed to yield a design that produces the same function but with higher quality than the original program. 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 logic. The objective is to take "spaghetti-bowl" code and derive a procedural design that conforms to the structured programming philosophy (Chapter 16).

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.

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.

30.4.2 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.

30.5 FORWARD ENGINEERING

A program with control flow that is the graphic equivalent of a bowl of spaghetti, with "modules" that are 2,000 statements long, with few meaningful comment lines in

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

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 CASE (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 haunt you.

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. CASE tools for reengineering will automate 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 re-create 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.

30.5.1 Forward Engineering for Client/Server Architectures

XRef

Client/server software engineering is discussed in Chapter 28.

Over the past decade many mainframe applications have been reengineered to accommodate client/server architectures. 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, c/s or OO systems designed to replace a legacy application should be approached as a new development project. Reengineering enters the picture only when elements of an old system are to be integrated with the new architecture. In some cases, you may be better off rejecting the old and creating identical new functionality.

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

Reengineering for c/s applications begins with a thorough analysis of the business environment that encompasses the existing mainframe. Three layers of abstraction (Figure 30.4) can be identified. The database sits at the foundation of a client/server architecture and manages transactions and queries from server 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 foundation layer. In some cases a new data model (Chapter 12) is created. In every case, the c/s 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 com-

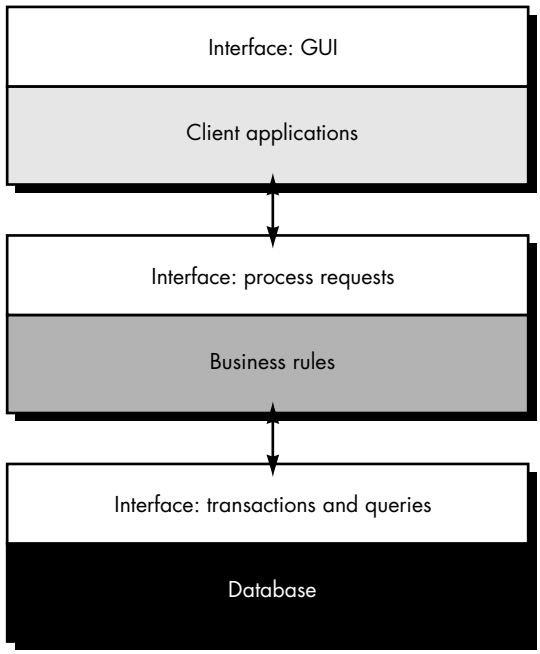

FIGURE 30.4
Reengineering mainframe applications to client/server

munications exist for that vendor), that queries can be accommodated efficiently, and that full archiving capability has been established.

The business rules layer represents software 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 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 [VAS93], [INM93], and [BER92].

30.5.2 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 Four 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 11 and 21) are created. The data models created during reverse engineering are then used in conjunction with CRC modeling (Chapter 21) 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 27) can be invoked. If the existing 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.

30.5.3 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.

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 fourth generation 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.

30.6 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:

Quote:

"You can pay us a little now, or pay us a lot more later."

Sign in auto dealership suggesting a tune-up

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

$$C_{\text{maint}} = [P_3 - (P_1 + P_2)] \times L \tag{30-1}$$

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

$$C_{\text{reeng}} = [P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9)] \tag{30-2}$$

Using the costs presented in equations (30-1) and (30-2), the overall benefit of reengineering can be computed as

$$\text{cost benefit} = C_{\text{reeng}} - C_{\text{maint}} \tag{30-3}$$

The cost/benefit analysis presented in the equations can be performed for all high-priority applications identified during inventory analysis (Section 30.2.2). Those applications that show the highest cost/benefit can be targeted for reengineering, while work on others can be postponed until resources are available.

30.7 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

[ARN89] Arnold, R.S., "Software Restructuring," *Proc. IEEE,* vol. 77, no. 4, April 1989, pp. 607–617.

[BER92] Berson, A., *Client/Server Architecture,* McGraw-Hill, 1992.

[BLE93] Bleakley, F.R., "The Best Laid Plans: Many Companies Try Management Fads, Only to See Them Flop," *The Wall Street Journal,* July 6, 1993, p. 1.

[BRE91] Breuer, P.T. and K. Lano, "Creating Specification From Code: Reverse-Engineering Techniques," *Journal of Software Maintenance: Research and Practice,* vol. 3, 1991, pp. 145–162.

[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.

[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.

[DIC95] Dickinson, B., *Strategic Business Reengineering,* LCI Press, 1995.

[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.

[INM93] Inmon, W.H., *Developing Client Server Applications,* QED Publishing, 1993.

[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.

[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.

[VAS93] Vaskevitch, D., *Client/Server Strategies,* IDG Books, 1993.

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

[WEI95] Weisz, M., "BPR Is Like Teenage Sex," *American Programmer,* vol. 8, no. 6, June 1995, pp. 9–15.

PROBLEMS AND POINTS TO PONDER

30.1. 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 30.1.3 to recommend changes to the process in an effort to make it more efficient.

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

30.3. 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.

30.4. 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 30.6.

30.5. 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.)

30.6. 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.

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

30.8. Why must interactivity increase if completeness is to increase?

30.9. Get product literature on three reverse engineering tools and present their characteristics in class.

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

30.11. Research the literature to find one or more papers that discuss case studies of mainframe to client/server reengineering. Present a summary.

30.12. How would you determine P_4 through P_7 in the cost-benefit model presented in Section 30.6?

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,* HarperCollins, 1993) 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 Andersen (*Business Process Improvement Toolbox,* American Society for Quality, 1999), Harrington et al. (*Business Process Improvement Workbook,* McGraw-Hill, 1997), Hunt (*Process Mapping: How to Reengineer Your Business Processes,* Wiley, 1996), and Carr and Johansson (*Best Practices in Reengineering: What Works and What Doesn't in the Reengineering Process,* McGraw-Hill, 1995) 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.

Relatively few books have been dedicated to software reengineering. Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software,* Fitzroy Dearborn Publishers, 1999) focuses on 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 business process reengineering and 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/engcs/compsci/pressman/resources/reengineering.mhtml

CHAPTER

31 COMPUTER-AIDED SOFTWARE ENGINEERING

KEY CONCEPTS

Everyone has heard the old saying about the shoemaker's children: The shoemaker is so busy making shoes for others that his children don't have shoes of their own. Prior to the 1990s, many software developers were the "shoemaker's children." Although these technical professionals built complex systems and products that automated the work of others, they used very little automation themselves.

Today, software engineers have their first new pair of shoes—computer-aided software engineering (CASE). The shoes don't come in as many varieties as they would like, haven't lived up to the many exaggerated promises made by their manufacturers, are often a bit stiff and sometimes uncomfortable, don't provide enough sophistication for those who are stylish, and don't always match other garments that software developers use, but they provide an absolutely essential piece of apparel for the software engineer's wardrobe and will, over time, become more comfortable, more useable, and more adaptable to the needs of individual practitioners.

In earlier chapters of this book we have attempted to provide a reasonable understanding of the underpinnings of software engineering technology. In this chapter, the focus shifts to the tools and environments that will help to automate the software process.

QUICK LOOK

What is it? Computer-aided software engineering (CASE) tools assist software engineering managers and practitioners in every activity associated with the software process. They automate project management activities, manage all work products produced throughout the process, and assist engineers in their analysis, design, coding and test work. CASE tools can be integrated within a sophisticated environment.

Who does it? Project managers and software engineers use CASE.

Why is it important? Software engineering is difficult. Tools that reduce the amount of effort required to produce a work product or accomplish some project milestone have substantial benefit. But there's something that's even more important. Tools can provide new ways of looking at software engineering information—ways that improve the insight of the engineer doing the work. This leads to better decisions and higher software quality.

What are the steps? CASE is used in conjunction with the process model that is chosen. If a full tool set is available, CASE will be used during virtually every step of the software process.

What is the work product? CASE tools assist a software engineer in producing high-quality work products. In addition, the availability of automation allows the CASE user to produce additional customized work products that could not be

QUICK LOOK

easily or practically produced without tool support.

How do I ensure that I've done it right? Use tools to complement solid software engineering practices—not to replace them. Before tools can be used effectively, a software process framework must be established, software engineering concepts and methods must be learned, and software quality must be emphasized. Only then will CASE provide benefit.

31.1 WHAT IS CASE?

A good workshop for any craftsperson—a mechanic, a carpenter, or a software engineer—has three primary characteristics: (1) a collection of useful tools that will help in every step of building a product, (2) an organized layout that enables tools to be found quickly and used efficiently, and (3) a skilled artisan who understands how to use the tools in an effective manner. Software engineers now recognize that they need more and varied tools along with an organized and efficient workshop in which to place the tools.

The workshop for software engineering has been called an *integrated project support environment* (discussed later in this chapter) and the tools that fill the workshop are collectively called *computer-aided software engineering*.

CASE provides the software engineer with the ability to automate manual activities and to improve engineering insight. Like computer-aided engineering and design tools that are used by engineers in other disciplines, CASE tools help to ensure that quality is designed in before the product is built.

31.2 BUILDING BLOCKS FOR CASE

Quote:

"The most valuable CASE tools are those that contribute information to the development process."

Robert Dixon

Computer aided software engineering can be as simple as a single tool that supports a specific software engineering activity or as complex as a complete "environment" that encompasses tools, a database, people, hardware, a network, operating systems, standards, and myriad other components. The building blocks for CASE are illustrated in Figure 31.1. Each building block forms a foundation for the next, with tools sitting at the top of the heap. It is interesting to note that the foundation for effective CASE environments has relatively little to do with software engineering tools themselves. Rather, successful environments for software engineering are built on an environment architecture that encompasses appropriate hardware and systems software. In addition, the environment architecture must consider the human work patterns that are applied during the software engineering process.

The environment architecture, composed of the hardware platform and system support (including networking software, database management, and object management services), lays the ground work for CASE. But the CASE environment itself

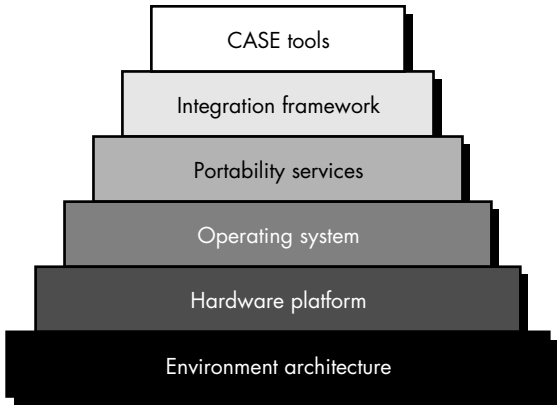

FIGURE 31.1
CASE building blocks

demands other building blocks. A set of *portability services* provides a bridge between CASE tools and their integration framework and the environment architecture. The *integration framework* is a collection of specialized programs that enables individual CASE tools to communicate with one another, to create a project database, and to exhibit the same look and feel to the end-user (the software engineer). Portability services allow CASE tools and their integration framework to migrate across different hardware platforms and operating systems without significant adaptive maintenance.

The building blocks depicted in Figure 31.1 represent a comprehensive foundation for the integration of CASE tools. However, most CASE tools in use today have not been constructed using all these building blocks. In fact, some CASE tools remain "point solutions." That is, a tool is used to assist in a particular software engineering activity (e.g., analysis modeling) but does not directly communicate with other tools, is not tied into a project database, is not part of an *integrated CASE environment* (I-CASE). Although this situation is not ideal, a CASE tool can be used quite effectively, even if it is a point solution.

Point-solution CASE tools can provide substantial individual benefit, but a software team needs tools that talk to one another. Integrated tools help the team develop, organize, and control work products. Use them.

The relative levels of CASE integration are shown in Figure 31.2. At the low end of the integration spectrum is the individual (point solution) tool. When individual tools provide facilities for data exchange (most do), the integration level is improved slightly. Such tools produce output in a standard format that should be compatible with other tools that can read the format. In some cases, the builders of complementary CASE tools work together to form a bridge between the tools (e.g., an analysis and design tool that is coupled with a code generator). Using this approach, the synergy between the tools can produce end products that would be difficult to create using either tool separately. *Single-source integration* occurs when a single CASE tools vendor integrates a number of different tools and sells them as a package. Although this approach is quite effective, the closed architecture of most single-source environments precludes easy addition of tools from other vendors.

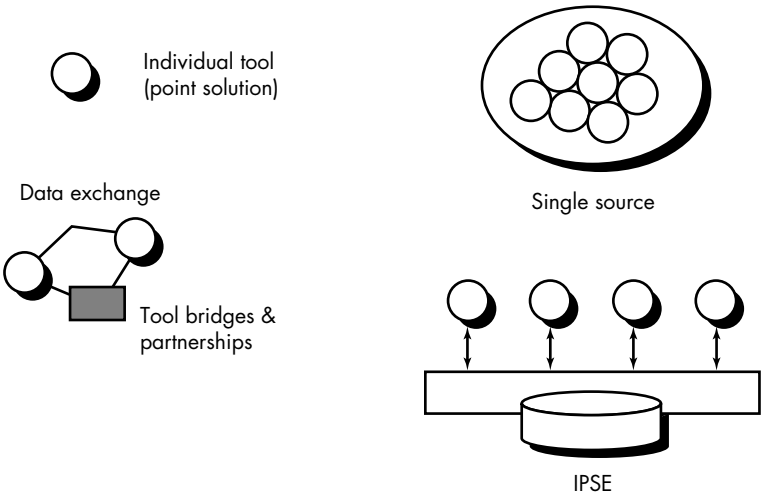

FIGURE 31.2 Integration options

At the high end of the integration spectrum is the *integrated project support environment* (IPSE). Standards for each of the building blocks described previously have been created. CASE tool vendors use IPSE standards to build tools that will be compatible with the IPSE and therefore compatible with one another.

31.3 A TAXONOMY OF CASE TOOLS

CASE Tools

A number of risks are inherent whenever we attempt to categorize CASE tools. There is a subtle implication that to create an effective CASE environment, one must implement all categories of tools—this is simply not true. Confusion (or antagonism) can be created by placing a specific tool within one category when others might believe it belongs in another category. Some readers may feel that an entire category has been omitted—thereby eliminating an entire set of tools for inclusion in the overall CASE environment. In addition, simple categorization tends to be flat—that is, we do not show the hierarchical interaction of tools or the relationships among them. But even with these risks, it is necessary to create a taxonomy of CASE tools—to better understand the breadth of CASE and to better appreciate where such tools can be applied in the software engineering process.

CASE tools can be classified by function, by their role as instruments for managers or technical people, by their use in the various steps of the software engineering process, by the environment architecture (hardware and software) that supports them, or even by their origin or cost [QED89]. The taxonomy presented here uses function as a primary criterion.

Business process engineering tools. By modeling the strategic information requirements of an organization, business process engineering tools provide a "meta-model" from which specific information systems are derived. Rather than focusing on the requirements of a specific application, business information is modeled as it moves between various organizational entities within a company. The primary objective for tools in this category is to represent business data objects, their relationships, and how these data objects flow between different business areas within a company.

XRef
Business process engineering is discussed in Chapter 10.

Process modeling and management tools. If an organization works to improve a business (or software) process, it must first understand it. Process modeling tools (also called *process technology* tools) are used to represent the key elements of a process so that it can be better understood. Such tools can also provide links to process descriptions that help those involved in the process to understand the work tasks that are required to perform it. Process management tools provide links to other tools that provide support to defined process activities.

XRef
The elements of the software process are discussed in Chapter 2.

Project planning tools. Tools in this category focus on two primary areas: software project effort and cost estimation and project scheduling. Estimation tools compute estimated effort, project duration, and recommended number of people for a project. Project scheduling tools enable the manager to define all project tasks (the work breakdown structure), create a task network (usually using graphical input), represent task interdependencies, and model the amount of parallelism possible for the project.

XRef
Estimation techniques are presented in Chapter 5. Scheduling methods are discussed in Chapter 7.

Risk analysis tools. Identifying potential risks and developing a plan to mitigate, monitor, and manage them is of paramount importance in large projects. Risk analysis tools enable a project manager to build a risk table by providing detailed guidance in the identification and analysis of risks.

XRef
Risk analysis and management are discussed in Chapter 6.

Project management tools. The project schedule and project plan must be tracked and monitored on a continuing basis. In addition, a manager should use tools to collect metrics that will ultimately provide an indication of software product quality. Tools in the category are often extensions to project planning tools.

XRef
Tracking and monitoring are discussed in Chapter 7.

Requirements tracing tools. When large systems are developed, things "fall into the cracks." That is, the delivered system does not fully meet customer specified requirements. The objective of requirements tracing tools is to provide a systematic approach to the isolation of requirements, beginning with the customer request for proposal or specification. The typical requirements tracing tool combines human-interactive text evaluation with a database management system that stores and categorizes each system requirement that is "parsed" from the original RFP or specification.

XRef
Requirements engineering methods are discussed in Chapter 10.

Metrics and management tools. Software metrics improve a manager's ability to control and coordinate the software engineering process and a practitioner's ability

XRef
Metrics are presented in Chapters 4, 19, and 24.

to improve the quality of the software that is produced. Today's metrics or measurement tools focus on process and product characteristics. Management-oriented tools capture project specific metrics (e.g., LOC/person-month, defects per function point) that provide an overall indication of productivity or quality. Technically oriented tools determine technical metrics that provide greater insight into the quality of design or code.

XRef
Documentation is discussed throughout the book. More detail is presented at the SEPA Web site.

Documentation tools. Document production and desktop publishing tools support nearly every aspect of software engineering and represent a substantial "leverage" opportunity for all software developers. Most software development organizations spend a substantial amount of time developing documents, and in many cases the documentation process itself is quite inefficient. It is not unusual for a software development organization to spend as much as 20 or 30 percent of all software development effort on documentation. For this reason, documentation tools provide an important opportunity to improve productivity.

XRef
See Chapters 27, 28, and 29 for limited discussion of these topics.

System software tools. CASE is a workstation technology. Therefore, the CASE environment must accommodate high-quality network system software, object management services, distributed component support, electronic mail, bulletin boards, and other communication capabilities.

XRef
SQA is presented in Chapter 8.

Quality assurance tools. The majority of CASE tools that claim to focus on quality assurance are actually metrics tools that audit source code to determine compliance with language standards. Other tools extract technical metrics (Chapters 19 and 24) in an effort to project the quality of the software that is being built.

XRef
The software repository is discussed in Chapter 9.

Database management tools. Database management software serves as a foundation for the establishment of a CASE database (repository) that we have called the *project database.* Given the emphasis on configuration objects, database management tools for CASE are evolving from relational database management systems to object-oriented database management systems.

XRef
SCM activities, including identification, version control, change control, auditing, and status accounting, are discussed in Chapter 9.

Software configuration management tools. Software configuration management lies at the kernel of every CASE environment. Tools can assist in all five major SCM tasks—identification, version control, change control, auditing, and status accounting. The CASE database provides a mechanism for identifying each configuration item and relating it to other items; the change control process can be implemented with the aid of specialized tools; easy access to individual configuration items facilitates the auditing process; and CASE communication tools can greatly improve status accounting (reporting information about changes to all who need to know).

Analysis and design tools. Analysis and design tools enable a software engineer to create models of the system to be built. The models contain a representation of data, function, and behavior (at the analysis level) and characterizations of data, archi-

tectural, component-level, and interface design.[1] By performing consistency and validity checking on the models, analysis and design tools provide a software engineer with some degree of insight into the analysis representation and help to eliminate errors before they propagate into the design, or worse, into implementation itself.

XRef
Analysis and design are discussed throughout Parts Three and Four of this book.

PRO/SIM tools. PRO/SIM (prototyping and simulation) tools [NIC90] provide the software engineer with the ability to predict the behavior of a real-time system prior to the time that it is built. In addition, these tools enable the software engineer to develop mock-ups of the real-time system, allowing the customer to gain insight into the function, operation and response prior to actual implementation.

XRef
Prototyping and simulation are discussed briefly in Chapter 10.

Interface design and development tools. Interface design and development tools are actually a tool kit of software components (classes) such as menus, buttons, window structures, icons, scrolling mechanisms, device drivers, and so forth. However, these tool kits are being replaced by interface prototyping tools that enable rapid on-screen creation of sophisticated user interfaces that conform to the interfacing standard that has been adopted for the software.

XRef
The elements of user interface design are presented in Chapter 15.

Prototyping tools. A variety of different prototyping tools can be used. *Screen painters* enable a software engineer to define screen layout rapidly for interactive applications. More sophisticated CASE prototyping tools enable the creation of a data design, coupled with both screen and report layouts. Many analysis and design tools have extensions that provide a prototyping option. PRO/SIM tools generate skeleton Ada and C source code for engineering (real-time) applications. Finally, a variety of fourth generation tools have prototyping features.

XRef
Prototyping is discussed in Chapters 2 and 11.

Programming tools. The programming tools category encompasses the compilers, editors, and debuggers that are available to support most conventional programming languages. In addition, object-oriented programming environments, fourth generation languages, graphical programming environments, application generators, and database query languages also reside within this category.

Web development tools. The activities associated with Web engineering are supported by a variety of tools for WebApp development. These include tools that assist in the generation of text, graphics, forms, scripts, applets, and other elements of a Web page.

XRef
WebE is discussed in Chapter 29.

Integration and testing tools. In their directory of software testing tools, Software Quality Engineering [SQE95] defines the following testing tools categories:

- *Data acquisition*—tools that acquire data to be used during testing.
- *Static measurement*—tools that analyze source code without executing test cases.

XRef
Software testing is discussed in Chapters 17, 18, and 23 as well as 28 and 29.

1 Analogous representations are provided by object-oriented analysis and design tools.

- *Dynamic measurement*—tools that analyze source code during execution.
- *Simulation*—tools that simulate function of hardware or other externals.
- *Test management*—tools that assist in the planning, development, and control of testing.
- *Cross-functional tools*—tools that cross the bounds of the preceding categories.

It should be noted that many testing tools have features that span two or more of the categories.

Static analysis tools. Static testing tools assist the software engineer in deriving test cases. Three different types of static testing tools are used in the industry: code-based testing tools, specialized testing languages, and requirements-based testing tools. *Code-based testing tools* accept source code (or PDL) as input and perform a number of analyses that result in the generation of test cases. *Specialized testing languages* (e.g., ATLAS) enable a software engineer to write detailed test specifications that describe each test case and the logistics for its execution. *Requirements-based testing tools* isolate specific user requirements and suggest test cases (or classes of tests) that will exercise the requirements.

XRef
White-box testing methods are discussed in Chapter 17.

Dynamic analysis tools. Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise instrumenting the execution flow of the program. Dynamic tools can be either intrusive or nonintrusive. An *intrusive tool* changes the software to be tested by inserting probes (extra instructions) that perform the activities just mentioned. *Nonintrusive testing tools* use a separate hardware processor that runs in parallel with the processor containing the program that is being tested.

Test management tools. Test management tools are used to control and coordinate software testing for each of the major testing steps. Tools in this category manage and coordinate regression testing, perform comparisons that ascertain differences between actual and expected output, and conduct batch testing of programs with interactive human/computer interfaces. In addition to the functions noted, many test management tools also serve as generic test drivers. A test driver reads one or more test cases from a testing file, formats the test data to conform to the needs of the software under test, and then invokes the software to be tested.

XRef
Test strategies are discussed in Chapter 18.

Client/server testing tools. The c/s environment demands specialized testing tools that exercise the graphical user interface and the network communications requirements for client and server.

XRef
c/s testing is discussed in Chapter 28.

Reengineering tools. Tools for legacy software address a set of maintenance activities that currently absorb a significant percentage of all software-related effort. The reengineering tools category can be subdivided into the following functions:

XRef
Reengineering methods are discussed in Chapter 30.

- *Reverse engineering to specification tools* take source code as input and generate graphical structured analysis and design models, where-used lists, and other design information.
- *Code restructuring and analysis tools* analyze program syntax, generate a control flow graph, and automatically generate a structured program.
- *On-line system reengineering tools* are used to modify on-line database systems (e.g., convert IDMS or DB2 files into entity-relationship format).

These tools are limited to specific programming languages (although most major languages are addressed) and require some degree of interaction with the software engineer.

31.4 INTEGRATED CASE ENVIRONMENTS

? What are the benefits of integrated CASE?

Although benefits can be derived from individual CASE tools that address separate software engineering activities, the real power of CASE can be achieved only through integration. The benefits of integrated CASE (I-CASE) include (1) smooth transfer of information (models, programs, documents, data) from one tool to another and one software engineering step to the next; (2) a reduction in the effort required to perform umbrella activities such as software configuration management, quality assurance, and document production; (3) an increase in project control that is achieved through better planning, monitoring, and communication; and (4) improved coordination among staff members who are working on a large software project.

KEY POINT
CASE tools integration demands a database that contains consistent representations of software engineering information.

But I-CASE also poses significant challenges. Integration demands consistent representations of software engineering information, standardized interfaces between tools, a homogeneous mechanism for communication between the software engineer and each tool, and an effective approach that will enable I-CASE to move among various hardware platforms and operating systems. Comprehensive I-CASE environments have emerged more slowly than originally expected. However, integrated environments do exist and are becoming more powerful as the years pass.

The term *integration* implies both *combination* and *closure*. I-CASE combines a variety of different tools and a spectrum of information in a way that enables closure of communication among tools, between people, and across the software process. Tools are integrated so that software engineering information is available to each tool that needs it; usage is integrated so that a common look and feel is provided for all tools; a development philosophy is integrated, implying a standardized software engineering approach that applies modern practice and proven methods.

To define integration in the context of the software engineering process, it is necessary to establish a set of requirements [FOR89a] for I-CASE: An integrated CASE environment should

- Provide a mechanism for sharing software engineering information among all tools contained in the environment.
- Enable a change to one item of information to be tracked to other related information items.
- Provide version control and overall configuration management for all software engineering information.
- Allow direct, nonsequential access to any tool contained in the environment.
- Establish automated support for the software process model that has been chosen, integrating CASE tools and software configuration items (SCIs) into a standard work breakdown structure.
- Enable the users of each tool to experience a consistent look and feel at the human/computer interface.
- Support communication among software engineers.
- Collect both management and technical metrics that can be used to improve the process and the product.

XRef

Process-related issues are discussed in Chapters 2, 4, and 7. SCIs are presented in Chapter 9.

To achieve these requirements, each of the building blocks of a CASE architecture (Figure 31.1) must fit together in a seamless fashion. The foundation building blocks—environment architecture, hardware platform, and operating system—must be "joined" through a set of portability services to an integration framework that achieves these requirements.

31.5 THE INTEGRATION ARCHITECTURE

A list of all software engineering information items can be found under SCIs

A software engineering team uses CASE tools, corresponding methods, and a process framework to create a pool of software engineering information. The integration framework facilitates transfer of information into and out of the pool. To accomplish this, the following architectural components must exist: a database must be created (to store the information); an object management system must be built (to manage changes to the information); a tools control mechanism must be constructed (to coordinate the use of CASE tools); a user interface must provide a consistent pathway between actions made by the user and the tools contained in the environment. Most models (e.g., [FOR90], [SHA95]) of the integration framework represent these components as layers. A simple model of the framework, depicting only the components just noted is shown in Figure 31.3.

The *user interface layer* (Figure 31.3) incorporates a standardized interface tool kit with a common presentation protocol. The interface tool kit contains software for human/computer interface management and a library of display objects. Both provide a consistent mechanism for communication between the interface and individual CASE tools. The *presentation protocol* is the set of guidelines that gives all CASE

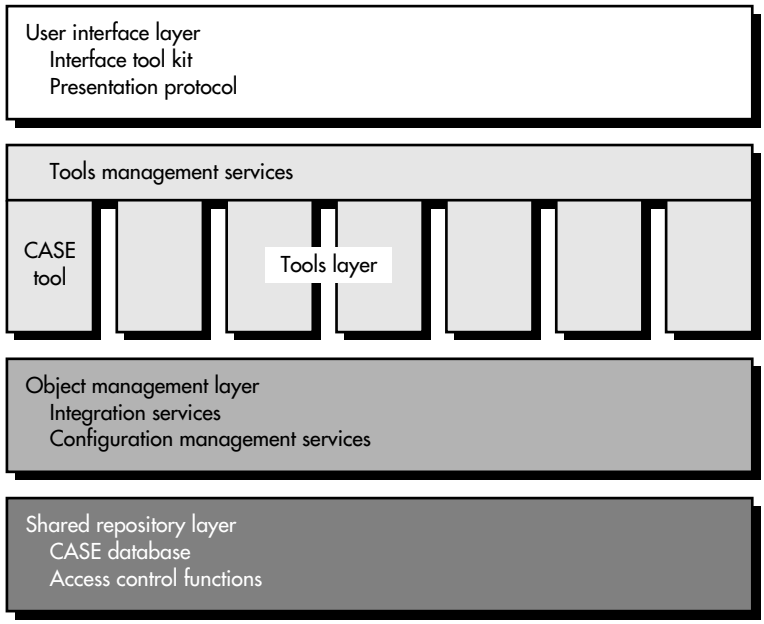

FIGURE 31.3 Architectural model for the integration framework

tools the same look and feel. Screen layout conventions, menu names and organization, icons, object names, the use of the keyboard and mouse, and the mechanism for tools access are all defined as part of the presentation protocol.

The *tools layer* incorporates a set of tools management services with the CASE tools themselves. *Tools management services* (TMS) control the behavior of tools within the environment. If multitasking is used during the execution of one or more tools, TMS performs multitask synchronization and communication, coordinates the flow of information from the repository and object management system into the tools, accomplishes security and auditing functions, and collects metrics on tool usage.

WebRef

Resources for CASE tool integration and integrated software engineering environments can be obtained at **see.cs.flinders.edu.au/seweb/ti/**

The *object management layer* (OML) performs the configuration management functions described in Chapter 9. In essence, software in this layer of the framework architecture provides the mechanism for tools integration. Every CASE tool is "plugged into" the object management layer. Working in conjunction with the CASE repository, the OML provides integration services—a set of standard modules that couple tools with the repository. In addition, the OML provides configuration management services by enabling the identification of all configuration objects, performing version control, and providing support for change control, audits, and status accounting.

The *shared repository layer* is the CASE database and the access control functions that enable the object management layer to interact with the database. Data integration is achieved by the object management and shared repository layers and is discussed in greater detail later in this chapter.

31.6 THE CASE REPOSITORY

Webster's Dictionary defines the word *repository* as "any thing or person thought of as a center of accumulation or storage." During the early history of software development, the repository was indeed a person—the programmer who had to remember the location of all information relevant to a software project, who had to recall information that was never written down and reconstruct information that had been lost. Sadly, using a person as "the center for accumulation and storage" (although it conforms to Webster's definition), does not work very well. Today, the repository is a "thing"—a database that acts as the center for both accumulation and storage of software engineering information. The role of the person (the software engineer) is to interact with the repository using CASE tools that are integrated with it.

In this book, a number of different terms have been used to refer to the storage place for software engineering information: *CASE database, project database, integrated project support environment (IPSE) database, requirements dictionary* (a limited database), and *repository.* Although there are subtle differences between some of these terms, all refer to the center for accumulation and storage.

31.6.1 The Role of the Repository in I-CASE

The repository for an I-CASE environment is the set of mechanisms and data structures that achieve data/tool and data/data integration. It provides the obvious functions of a database management system, but in addition, the repository performs or precipitates the following functions [FOR89b]:

? What functions are performed by the services that are coupled with the CASE repository?

- *Data integrity* includes functions to validate entries to the repository, ensure consistency among related objects, and automatically perform "cascading" modifications when a change to one object demands some change to objects related to it.
- *Information sharing* provides a mechanism for sharing information among multiple developers and between multiple tools, manages and controls multiuser access to data and locks or unlocks objects so that changes are not inadvertently overlaid on one another.
- *Data/tool integration* establishes a data model that can be accessed by all tools in the I-CASE environment, controls access to the data, and performs appropriate configuration management functions.
- *Data/data integration* is the database management system that relates data objects so that other functions can be achieved.
- *Methodology enforcement* defines an entity-relationship model stored in the repository that implies a specific paradigm for software engineering; at a minimum, the relationships and objects define a set of steps that must be conducted to build the contents of the repository.

- *Document standardization* is the definition of objects in the database that leads directly to a standard approach for the creation of software engineering documents.

To achieve these functions, the repository is defined in terms of a meta-model. The *meta-model* determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs [WEL89].

The meta-model is the template into which software engineering information is placed. A detailed discussion of these models is beyond the scope of this book. For further information, the interested reader should see [WEL89], [SHA95], and [GRI95].

31.6.2 Features and Content

The features and content of the repository are best understood by looking at it from two perspectives: what is to be stored in the repository and what specific services are provided by the repository. In general, the types of things to be stored in the repository include

- The problem to be solved.
- Information about the problem domain.
- The system solution as it emerges.
- Rules and instructions pertaining to the software process (methodology) being followed.
- The project plan, resources, and history.
- Information about the organizational context.

A detailed list of types of representations, documents and deliverables that are stored in the CASE repository is included in Table 31.1.

A robust CASE repository provides two different classes of services: (1) the same types of services that might be expected from any sophisticated database management system and (2) services that are specific to the CASE environment.

Many repository requirements are the same as those of typical applications built on a commercial database management system (DBMS). In fact, most of today's CASE repositories employ a DBMS (usually relational or object oriented) as the basic data management technology. The DBMS features that support the management of software development information include

? What typical DBMS features support CASE?

- *Nonredundant data storage.* Each object is stored only once, but is accessible by all CASE tools that need it.
- *High-level access.* A common data access mechanism is implemented so data handling facilities do not have to be duplicated in each CASE tool.

TABLE 31.1 Case Repository Contents [FOR89B]

Enterprise information
- Organizational structure
- Business area analyses
- Business functions
- Business rules
- Process models (scenarios)
- Information architecture

Application design
- Methodology rules
- Graphical representations
- System diagrams
- Naming standards
- Referential integrity rules
- Data structures
- Process definitions
- Class definitions
- Menu trees
- Performance criteria
- Timing constraints
- Screen definitions
- Report definitions
- Logic definitions
- Behavioral logic
- Algorithms
- Transformation rules

Construction
- Source code; Object code
- System build instructions
- Binary images
- Configuration dependencies
- Change information

Validation and verification
- Test plan; Test data cases
- Regression test scripts
- Test results
- Statistical analyses
- Software quality metrics

Project management information
- Project plans
- Work breakdown structure
- Estimates; Schedules
- Resource loading; Problem reports
- Change requests; Status reports
- Audit information

System documentation
- Requirements documents
- External/internal designs
- User manuals

- *Data independence.* CASE tools and the target applications are isolated from physical storage so they are not affected when the hardware configuration is changed.
- *Transaction control.* The repository implements record locking, two-stage commits, transaction logging, and recovery procedures to maintain the integrity of the data when there are concurrent users.
- *Security.* The repository provides mechanisms to control who can view and modify information contained within it.
- *Ad hoc data queries and reports.* The repository allows direct access to its contents through a convenient user interface such as SQL or a forms-oriented "browser," enabling user-defined analysis beyond the standard reports provided with the CASE tool set.
- *Openness.* Repositories usually provide a simple import/export mechanism to enable bulk loading or transfer.
- *Multiuser support.* A robust repository must permit multiple developers to work on an application at the same time. It must manage concurrent access to the database by multiple tools and users with access arbitration and lock-

ing at the file or record level. For environments based on networking, multi-user support also implies that the repository can interface with common networking protocols (object request brokers) and facilities.

The CASE environment also places special demands on the repository that go beyond what is directly available in a commercial DBMS. The special features of CASE repositories include

? What special features are exhibited by the CASE repository?

- *Storage of sophisticated data structures.* The repository must accommodate complex data types such as diagrams, documents, and files, as well as simple data elements. A repository also includes an information model (or meta-model) describing the structure, relationships and semantics of the data stored in it. The meta-model must be extensible so that new representations and unique organizational information can be accommodated. The repository not only stores models and descriptions of systems under development, but also associated meta-data (i.e., additional information describing the software engineering data itself, such as when a particular design component was created, what its current status is, and what other components it depends upon).
- *Integrity enforcement.* The repository information model also contains rules, or policies, describing valid business rules and other constraints and requirements on information being entered into the repository (directly or via a CASE tool). A facility called a *trigger* may be employed to activate the rules associated with an object whenever it is modified, making it possible to check the validity of design models in real time.
- *Semantics-rich tool interface.* The repository information model (meta-model) contains semantics that enable a variety of tools to interpret the meaning of the data stored in the repository. For example, a data flow diagram created by a CASE tool is stored in the repository in a form based on the information model and independent of any internal representations used by the tool itself. Another CASE tool can then interpret the contents of the repository and use the information as needed for its task. Thus, the semantics stored in the repository permit data sharing among a variety of tools, as opposed to specific tool-to-tool conversions or "bridges."
- *Process/project management.* A repository contains information not only about the software application itself, but also about the characteristics of each particular project and the organization's general process for software development (phases, tasks, and deliverables). This opens up possibilities for automated coordination of technical development activity with the project management activity. For example, updating the status of project tasks could be done automatically or as a by-product of using the CASE tools. Status updating can be made very easy for developers to perform without having to

A detailed tutorial and list of resources for OO repositories (which can be used for CASE environments) can be found at **mini.net/cetus/oo_db_systems_1.html**

leave the normal development environment. Task assignment and queries can also be handled by e-mail. Problem reports, maintenance tasks, change authorization, and repair status can be coordinated and monitored via tools accessing the repository.

The following repository features are all encompassed by software configuration management (Chapter 9). They are re-examined here to emphasize their interrelationship to I-CASE environments:

How does the repository assist in SCM?

Versioning. As a project progresses, many versions of individual work products will be created. The repository must be able to save all of these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.

The CASE repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents, and unique objects like screen and report definitions, object files, test data, and results. A mature repository tracks versions of objects with arbitrary levels of granularity, for example, a single data definition or a cluster of modules can be tracked.

To support parallel development, the version control mechanism should permit multiple derivatives (variants) from a single predecessor. Thus, a developer could be working on two possible solutions to a design problem at the same time, both generated from the same starting point.

Dependency tracking and change management. The repository manages a wide variety of relationships among the data elements stored in it. These include relationships between enterprise entities and processes, among the parts of an application design, between design components and the enterprise information architecture, between design elements and deliverables, and so on. Some of these relationships are merely associations, and some are dependencies or mandatory relationships. Maintaining these relationships among development objects is called *link management.*

KEY POINT

The repository's ability to track relationships among configuration objects is one of its most important features. The impact of change can be tracked if this feature is available.

The ability to keep track of all of these relationships is crucial to the integrity of the information stored in the repository and to the generation of deliverables based on it, and it is one of the most important contributions of the repository concept to the improvement of the software development process. Among the many functions that link management supports is the ability to identify and assess the effects of change. As designs evolve to meet new requirements, the ability to identify all objects that might be affected enables more accurate assessment of cost, downtime, and degree of difficulty. It also helps prevent unexpected side effects that would otherwise lead to defects and system failures.

Link management helps the repository mechanism ensure that design information is correct by keeping the various portions of a design synchro-

nized. For example, if a data flow diagram is modified, the repository can detect whether related data dictionaries, screen definitions, and code modules also require modification and can bring affected components to the developer's attention.

Requirements tracing. This special function depends on link management and provides the ability to track all the design components and deliverables that result from a specific requirement specification (forward tracking). In addition, it provides the ability to identify which requirement generated any given deliverable (backward tracking).

Configuration management. A configuration management facility works closely with the link management and versioning facilities to keep track of a series of configurations representing specific project milestones or production releases. Version management provides the needed versions, and link management keeps track of interdependencies.

Audit trails. An audit trail establishes additional information about when, why, and by whom changes are made. Information about the source of changes can be entered as attributes of specific objects in the repository. A repository trigger mechanism is helpful for prompting the developer or the tool that is being used to initiate entry of audit information (such as the reason for a change) whenever a design element is modified.

31.7 SUMMARY

Computer-aided software engineering tools span every activity in the software process and those umbrella activities that are applied throughout the process. CASE combines a set of building blocks that begin at the hardware and operating system software level and end with individual tools.

In this chapter, we consider a taxonomy of CASE tools. Categories encompass both management and technical activities that span most software application areas. Each category of tool is considered a "point solution."

The I-CASE environment combines integration mechanisms for data, tools, and human/computer interaction. Data integration can be achieved through direct exchange of information, through common file structures, by data sharing or interoperability, or through the use of a full I-CASE repository. Tools integration can be custom designed by vendors who work together or achieved through management software provided as part of the repository. Human/computer integration is achieved through interface standards that have become commonplace throughout the industry. An integration architecture is designed to facilitate the integration of users with tools, tools with tools, tools with data, and data with data.

The CASE repository has been referred to as a "software bus." Information moves through it, passing from tool to tool as software engineering progresses. But

the repository is much more than a "bus." It is also a storage place that combines sophisticated mechanisms for integrating CASE tools and thereby improving the process through which software is developed. The repository is a relational or object-oriented database that is "the center of accumulation and storage" for software engineering information.

REFERENCES

[FOR89a] Forte, G., "In Search of the Integrated Environment," *CASE Outlook,* March–April 1989, pp. 5–12.

[FOR89b] Forte, G., "Rally Round the Repository," *CASE Outlook,* December 1989, pp. 5–27.

[FOR90] Forte, G., "Integrated CASE: A Definition," *Proc. 3rd Annual TEAMWORKERS Intl. User's Group Conference,* Cadre Technologies, Providence, RI, March 1990.

[GRI95] Griffen, J., "Repositories: Data Dictionary Descendant Can Extend Legacy Code Investment," *Application Development Trends,* April 1995, pp. 65–71.

[NIC90] Nichols, K.M., "Performance Tools," *IEEE Software,* May 1990, pp. 21–23.

[QED89] *CASE: The Potential and the Pitfalls,* QED Information Sciences,1989.

[SQE95] *Testing Tools Reference Guide,* Software Quality Engineering, 1995.

[SHA95] Sharon, D. and R. Bell, "Tools That Bind: Creating Integrated Environments," *IEEE Software,* March 1995, pp. 76–85.

[WEL89] Welke, R.J., "Meta Systems on Meta Models," *CASE Outlook,* December 1989, pp. 35–45.

PROBLEMS AND POINTS TO PONDER

31.1. Make a list of all software development tools that you use. Organize them according to the taxonomy presented in this chapter.

31.2. Using the ideas introduced in Chapters 13 through 16, how would you suggest that portability services be built?

31.3. Build a paper prototype for a project management tool that encompasses the categories noted in Section 31.3. Use Part Two of this book for additional guidance.

31.4. Do some research on object-oriented database management systems. Discuss why OODMS would be ideal for SCM tools.

31.5. Gather product information on at least three CASE tools in a category specified by your instructor. Develop a matrix that compares features.

31.6. Are there situations in which dynamic testing tools are "the only way to go"? If so, what are they?

31.7. Discuss other human activities in which the integration of a set of tools has provided substantially more benefit than the use of each of the tools individually. Do not use examples from computing.

31.8. Describe what is meant by data/tool integration in your own words.

31.9. In a number of places in this chapter, the terms meta-model and meta-data are used. Describe what these terms mean in your own words.

31.10. Can you think of additional configuration items that might be included in the repository contents shown in Table 31.1? Make a list.

FURTHER READINGS AND INFORMATION SOURCES

A number of books on CASE were published in the 1980s and early 1990s in an effort to capitalize on the high degree of interest in the industry at that time. Subsequently, few books on the subject have appeared. Among the early offerings that still have value are

Bergin, T. et al., *Computer-Aided Software Engineering: Issues and Trends for the 1990s and Beyond,* Idea Group Publishing, 1993.

Braithwaite, K.S., *Application Development Using CASE Tools,* Academic Press, 1990.

Brown, A.W., D.J. Carney, and E.J. Morris, *Principles of CASE Tool Integration,* Oxford University Press, 1994.

Clegg, D. and R. Barker, *CASE Method Fast-Track: A RAD Approach,* Addison-Wesley, 1994.

Lewis, T.G., *Computer-Aided Software Engineering,* Van Nostrand-Reinhold, 1990.

Mylls, R., *Information Engineering: CASE Practices and Techniques,* Wiley, 1993.

An anthology by Chikofsky (*Computer-Aided Software Engineering,* 2nd ed., IEEE Computer Society, 1992) contains a useful collection of early papers on CASE and software development environments. Muller and his colleagues (*Computer-Aided Software Engineering,* Kluwer Academic Publishers, 1996) have edited a collection of that describes CASE research in the mid-1990s. The best sources of current information on CASE tools are the Internet, technical periodicals, and industry newsletters.

IEEE Standard 1209 (*Evaluation and Selection of CASE Tools*) presents a set of guidelines for evaluating CASE tools for "project management processes, pre-development processes, development processes, post-development processes, and integral processes." A detailed report by Wallnau and Feiler (*Tool Integration and Environment Architectures, Software Engineering Institute,* CMU/SEI-91-TR-11, May 1991), although dated, remains one of the best discussions of CASE environments readily available.

A wide variety of information sources on CASE 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/engcs/compsci/pressman/resources/CASE.mhtml

CHAPTER

32 THE ROAD AHEAD

KEY CONCEPTS

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 CASE 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 begin our journey through 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 more than a decade ago, Max Hopper [HOP90] 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.

Changes in software engineering technology are indeed "rapid and unforgiving," but at the same time progress is often quite slow. By the time a decision

QUICK LOOK

What is it? The future is never easy to predict—pundits, talking heads, and industry experts notwithstanding. 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.

QUICK LOOK

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.

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 that 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.
>
> Software is pervasive, and yet, many people in positions of responsibility have little or no real understanding of what it really is, how it's built, or what it means to the institutions that they (and it) control. More importantly, they have little appreciation of the dangers and opportunities that software offers.

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."

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, engineering. During the next few decades, revolutionary advances in computing may well be driven by "soft sciences"—human psychology, biology, neurophysiology, sociology, philosophy, and others. The gestation period for the computing technologies that may be derived from these disciplines is very difficult to predict.

Quote:

"The best thing about the future is that it comes one day at a time."

Abraham Lincoln

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.

The changes that will affect software engineering over the next decade will be influenced from 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."

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, bulletin boards, 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 bulletin board 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, bulletin boards 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.

Quote:

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

Alvin Toffler

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 coordinating 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.[1]

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 evolutionary models for software development.

Evolutionary process models recognize that uncertainty dominates most projects, that timelines are often impossibly short, and that iteration provides the ability to

1 The SEPA Web site can provide you with electronic links to most important subjects presented in this book.

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.

What activities must populate the evolutionary process? Over the past decade, the Capability Maturity Model developed by the Software Engineering Institute [PAU93] has had a substantial impact on efforts to improve software engineering practices. The CMM has generated much debate (e.g., [BOL91], [GIL96]), and yet, it provides a good indicator of the attributes that must exist when solid software engineering is practiced.

Quote:

"The best preparation for good work tomorrow is to do good work today."

Elbert Hubbard

Object technologies, coupled with component-based software engineering (Chapter 27), are a natural outgrowth of the trend toward evolutionary 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 Web-based applications (WebApps) is changing both the software engineering process and its participants. Again, we encounter an incremental, evolutionary paradigm. But in the case of WebApps, immediacy, security, and aesthetics become dominant concerns. A Web engineering team melds 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 Web engineering work 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 to accommodate the Web.

32.5 NEW MODES FOR REPRESENTING INFORMATION

Over the past two decades, a subtle transition has occurred in the terminology that is used to describe software development work performed for the business community. Thirty 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

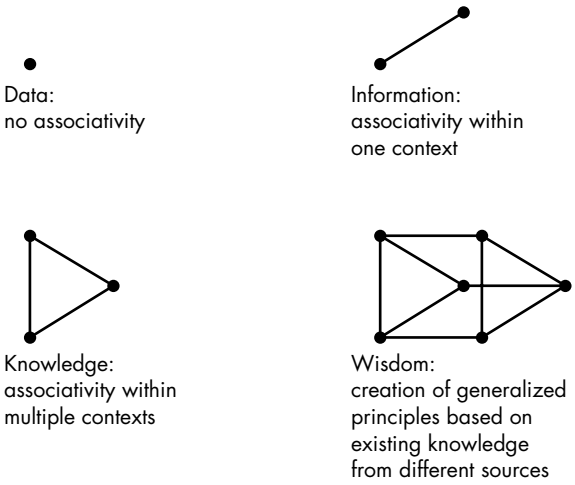

FIGURE 32.1 An "information" spectrum

applications, but its use is relatively rare. Virtually no one discusses *wisdom* in the context of computer 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.

Quote:

"Wisdom is the power that enables us to use knowledge for the benefit of ourselves and others."

Thomas J. Watson

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 knowledge.[2] 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.

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" have begun 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

2 The rapid growth in data mining and data warehousing technologies reflects this growing trend.

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 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 toward systems that process knowledge. We have been processing data for 50 years and extracting information for almost 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.

Quote:

"The new electronic independence recreates the world in the image of a global village."

Marshall McLuhan

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., massively parallel machines, optical processors, neural network machines) 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 not yet mature, it is difficult to determine which will survive 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 (technologies that are not yet mature) 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.

32.7 A CONCLUDING COMMENT

It has been 20 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,[3] and the rest, as they say, is history.

Over the past 20 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 degree of automation, and the most effective methods, the underlying principles of software engineering are now accepted throughout the industry. Why, then, are we seeing 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 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 begin a new millennium, software has become the most important product and the most important industry on the world stage. Its impact and importance

---

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

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

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

PROBLEMS AND POINTS TO PONDER

32.1. 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.

32.2. One of the hottest software application domains is Web-based systems and applications (Chapter 29). Discuss how people, communication, and process has to evolve to accommodate the development of "next generation" WebApps.

32.3. 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.

32.4. Review the discussion of the evolutionary process models in Chapter 2. Do some research and collect recent papers on the subject. Summarize the strengths and weaknesses of evolutionary paradigms based on experiences outlined in the papers.

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

32.6. Select a current "hot" technology (it need not be a software technology) that is being discussed in the popular media and describe how software enables its evolution and impact.

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. Robertson (*The New*

Renaissance: Computers and the Next Level of Civilization, Oxford University Press, 1998) argues that the computer revolution may be the single most significant advance in the history of civilization. Dertrouzos and Gates (*What Will Be: How the New World of Information Will Change Our Lives,* HarperBusiness, 1998) provide a thoughtful discussion of some of the directions that information technologies may take in the first few decades of this century. Barnatt (*Valueware: Technology, Humanity and Organization,* Praeger Publishing, 1999) presents an intriguing discussion of an "ideas economy" and how economic value will be created as cyber-business evolves. Negroponte's (*Being Digital,* Alfred A. Knopf, 1995) was a best seller in the mid-1990s and continues to provide an interesting view of computing and its overall impact.

Kroker and Kroker (*Digital Delirium,* New World Perspectives, 1997) have edited a controversial collection of essays, poems, and humor that examines the impact of digital technologies on people and society. Brin (*The Transparent Society: Will Technology Force Us to Choose Between Privacy and Freedom?* Perseus Books, 1999) revisits the continuing debate associated with the inevitable loss of personal privacy that accompanies the growth of information technologies. Shenk (*Data Smog: Surviving the Information Glut,* HarperCollins, 1998) discusses the problems associated with an "information-infested society" that is suffocating from the volume of information that information technologies produce.

Miller, Michalski, and Stevens (*21st Century Technologies: Promises and Perils of a Dynamic Future,* Brookings Institution Press, 1999) have edited a collection of papers and essays on the impact of technology on social, business, and economic structures. For those interested in technical issues, Luryi, Xu, and Zaslavsky *(Future Trends in Microelectronics,* Wiley, 1999) have edited a collection of papers on probable directions for computer hardware. Hayzelden and Bigham (*Software Agents for Future Communication Systems,* Springer-Verlag, 1999) have edited a collection that discusses trends in the development of intelligent software agents.

Kurzweil (*The Age of Spiritual Machines, When Computers Exceed Human Intelligence,* Viking/Penguin Books, 1999) argues that, within 20 years, hardware technology will have the capacity to fully model the human brain. Borgmann (*Holding on to Reality: The Nature of Information at the Turn of the Millennium,* University of Chicago Press, 1999) has written a intriguing history of information, tracing its role in the transformation of culture. Devlin (*InfoSense: Turning Information into Knowledge,* W. H. Freeman & Co., 1999) tries to make sense of the constant flow of information that bombards us on a daily basis. Gleick (*Faster: The Acceleration of Just About Everything,* Pantheon Books, 2000) discusses the ever-accelerating rate of technological change and its impact on every aspect of modern life. Jonscher (*The Evolution of Wired Life: From the Alphabet to the Soul-Catcher Chip—How Information Technologies Change Our World,* Wiley, 2000) argues that human thought and interaction transcend the importance of technology.

A wide variety of information sources on future trends in computing 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/engcs/compsci/pressman/resources/future.mhtml

INDEX