chically. As decomposition begins, the system is represented as a collection of architectural frameworks, each composed of one or more design patterns (Chapter 10). Further refinement identifies the components that are required to create each design pattern. In an ideal context, all of these components would be acquired from a repository (*component qualification, adaptation,* and *composition* activities apply). When specialized components are required, *component engineering* is applied.

30.3 Domain Engineering

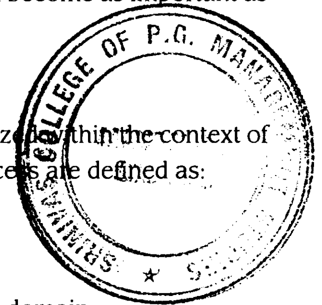
The intent of *domain engineering* is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems. Domain engineering includes three major activities—analysis, construction, and dissemination.

> **"Domain engineering is about finding commonalities among systems to identify components that can be applied to many systems, and to identify program families that are positioned to take fullest advantage of those components."**
>
> **Paul Clements**

It can be argued that "reuse will disappear, not by elimination, but by integration" into the fabric of software engineering practice [TRA95]. As greater emphasis is placed on reuse, some believe that domain engineering will become as important as software engineering over the next decade.

The analysis process we discuss in this section focuses on reusable components. However, the analysis of complete COTS systems (e.g., e-commerce apps, sales force automation apps) can also be a part of domain analysis.

30.3.1 The Domain Analysis Process



The overall approach to domain analysis is often characterized within the context of object-oriented software engineering. The steps in the process are defined as:

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample and define analysis classes.
5. Develop an analysis model for the classes.

It is important to note that domain analysis is applicable to any software engineering paradigm and may be applied for conventional as well as object-oriented development.

What components identified during domain analysis will be candidates for reuse?

Although the steps just noted provide a useful model for domain analysis, they provide no guidance for deciding which software components are candidates for reuse. Hutchinson and Hindley [HUT88] suggest the following set of pragmatic questions as a guide for identifying reusable software components:

- Is component functionality required on future implementations?

- How common is the component's function within the domain?
- Is there duplication of the component's function within the domain?
- Is the component hardware dependent? If so, does the hardware remain unchanged between implementations or can the hardware specifics be removed to another component?
- Is the design optimized enough for the next implementation?
- Can we parameterize a nonreusable component so that it becomes reusable?
- Is the component reusable in many implementations with only minor changes?
- Is reuse through modification feasible?
- Can a nonreusable component be decomposed to yield reusable components?
- How valid is component decomposition for reuse?

For additional information on domain analysis, see [ATK01], [HEI01], and [PRI93].

30.3.2 Characterization Functions

It is sometimes difficult to determine whether a potentially reusable component is in fact applicable in a particular situation. To make this determination, it is necessary to define a set of domain characteristics that are shared by all software within a domain. A domain characteristic defines some generic attribute of all products that exist within the domain. For example, generic characteristics might include the importance of safety/reliability, programming language, concurrency in processing, and many others.

A set of domain characteristics for a reusable component can be represented as $\{D_p\}$, where each item, D_{pi}, in the set represents a specific domain characteristic. The value assigned to D_{pi} represents an ordinal scale that is an indication of the relevance of the characteristic for component *p*. A typical scale [BAS94] might be

1: Not relevant to whether reuse is appropriate.

2: Relevant only under unusual circumstances.

3: Relevant—the component can be modified so that it can be used, despite differences.

4: Clearly relevant, and if the new software does not have this characteristic, reuse will be inefficient but may still be possible.

5: Clearly relevant, and if the new software does not have this characteristic, reuse will be ineffective and reuse without the characteristic is not recommended.

WebRef
Useful information on domain analysis can be found at www.sei.cmu.edu/str/descriptions/doda.html.

When new software, *w*, is to be built within the application domain, a set of domain characteristics is derived for it. A comparison is then made between D_{pi} and D_{wi} to determine whether the existing component *p* can be effectively reused in application *w*.

Even when software to be engineered clearly exists within an application domain, the reusable components within that domain must be analyzed to determine their applicability. In some cases (hopefully, a limited number), "reinventing the wheel" may still be the most cost-effective choice.

30.3.3 Structural Modeling and Structure Points

When domain analysis is applied, the analyst looks for repeating patterns in the applications that reside within a domain. Structural modeling is a pattern-based domain engineering approach that works under the assumption that every application domain has repeating patterns (of function, data, and behavior) that have reuse potential.

Each application domain can be characterized by a structural model (e.g., aircraft avionics systems differ greatly in specifics, but all modern software in this domain has the same structural model). Therefore, the structural model is an architectural style (Chapter 10) that can and should be reused across applications within the domain.

McMahon [MCM95] describes a *structure point* as "a distinct construct within a structural model." Structure points have three distinct characteristics:

What is a structure point, and what are its characteristics?

1. A structure point is an abstraction that should have a limited number of instances. In addition, the abstraction should recur throughout applications in the domain. Otherwise, the cost to verify, document, and disseminate the structure point cannot be justified.
2. The rules that govern the use of the structure point should be easily understood. In addition, the interface to the structure point should be relatively simple.
3. The structure point should implement information hiding by isolating all complexity contained within the structure point itself. This reduces the perceived complexity of the overall system.

KEY POINT
A structure point is analogous to a design pattern that can be found repeatedly in applications with a specific domain.

As an example of structure points as architectural patterns for a system, consider the domain of software for alarm systems. This domain might encompass systems as simple as *SafeHome* (discussed in earlier chapters) or as complex as the alarm system for an industrial process. In every case, however, a set of predictable structural patterns are encountered: an *interface* that enables the user to interact with the system, a *bounds-setting mechanism* that allows the user to establish bounds on the parameters to be measured, a *sensor management mechanism* that communicates with all monitoring sensors, a *response mechanism* that reacts to the input provided by the sensor management system, and a *control mechanism* that enables the user to control the manner in which monitoring is carried out. Each of these structure points is integrated into a domain architecture.

It is possible to define generic structure points that transcend a number of different application domains [STA94]:

- *Application front end*—the GUI including all menus, panels, and input and command editing facilities.
- *Database*—the repository for all objects relevant to the application domain.
- *Computational engine*—the numerical and nonnumerical models that manipulate data.
- *Reporting facility*—the function that produces output of all kinds.
- *Application editor*—the mechanism for customizing the application to the needs of specific users.

Structure points have been suggested as an alternative to lines of code and function points for software cost estimation [MCM95]. A brief discussion of costing using structure points is presented in Section 30.6.2.

30.4 Component-Based Development

Component-based development (CBD) is a CBSE activity that occurs in parallel with domain engineering. Using analysis and architectural design methods discussed earlier in this book, the software team refines an architectural style that is appropriate for the analysis model created for the application to be built.[2]

Once the architecture has been established, it must be populated by components that (1) are available from reuse libraries and/or (2) are engineered to meet custom needs. Hence, the task flow for component-based development has two parallel paths (Figure 30.1). When reusable components are available for potential integration into the architecture, they must be qualified and adapted. When new components are required, they must be engineered. The resultant components are then "composed" (integrated) into the architecture template and tested thoroughly.

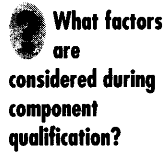
30.4.1 Component Qualification, Adaptation, and Composition

As we have already seen, domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in-house, others can be extracted from existing applications, and still others may be acquired from third parties.

Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development activities is applied when a component is proposed for use.

2 It should be noted that the architectural style is often influenced by the generic structural model created during domain engineering (see Figure 30.1).

Component qualification. Component qualification ensures that a candidate component will perform the function required, will properly "fit" into the architectural style specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application.



The interface description provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can, in fact, be reused effectively in a new application. Among the many factors considered during component qualification are [BRO96]: application programming interface (API); development and integration tools required by the component; run-time requirements, including resource usage (e.g., memory or storage), timing or speed, and network protocol; service requirements, including operating system interfaces and support from other components; security features, including access controls and authentication protocol; embedded design assumptions, including the use of specific numerical or non-numerical algorithms; and exception handling.

Each of these factors is relatively easy to assess when reusable components that have been developed in-house are proposed. However, it is much more difficult to determine the internal workings of COTS or third-party components because the only available information may be the interface specification itself.

Component adaptation. In an ideal setting, domain engineering creates a library of components that can be easily integrated into an application architecture. The implication of "easy integration" is that (1) consistent methods of resource management have been implemented for all components in the library, (2) common activities such as data management exist for all components, and (3) interfaces within the architecture and with the external environment have been implemented in a consistent manner.

In addition to assessing whether the cost of adaptation for reuse is justified, the software team also assesses whether achieving required functionality and performance can be done cost-effectively.

In reality, even after a component has been qualified for use within an application architecture, conflicts may occur in one or more of the areas just noted. To avoid these conflicts, an adaptation technique called *component wrapping* [BRO96] is often used. When a software team has full access to the internal design and code for a component (often not the case when COTS components are used) *white-box wrapping* is applied. Like its counterpart in software testing (Chapter 14), white-box wrapping examines the internal processing details of the component and makes code-level modifications to remove any conflict. *Gray-box wrapping* is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. *Black-box wrapping* requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts. The software team must determine whether the effort required to adequately wrap a component is justified or whether a custom component (designed to eliminate the conflicts encountered) should be engineered instead.

Component composition. The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure must be established to bind the components into an operational system. The infrastructure (usually a library of specialized components) provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

Among the many mechanisms for creating an effective infrastructure is a set of four "architectural ingredients" [ADL95] that should be present to achieve component composition:

What ingredients are necessary to achieve component composition?

Data exchange model. Mechanisms that enable users and applications to interact and transfer data (e.g., drag and drop, cut and paste) should be defined for all reusable components. The data exchange mechanisms not only allow human-to-software and component-to-component data transfer but also transfer among system resources (e.g., dragging a file to a printer icon for output).

Automation. A variety of tools, macros, and scripts should be implemented to facilitate interaction between reusable components.

Structured storage. Heterogeneous data (e.g., graphical data, voice/video, text, and numerical data) contained in a "compound document" should be organized and accessed as a single data structure, rather than a collection of separate files. "Structured data maintains a descriptive index of nesting structures that applications can freely navigate to locate, create, or edit individual data contents as directed by the end user" [ADL95].

Underlying object model. The object model ensures that components developed in different programming languages that reside on different platforms can be interoperable. That is, objects must be capable of communicating across a network. To achieve this, the object model defines a standard for component interoperability.

Because the potential impact of reuse and CBSE on the software industry is enormous, a number of major companies and industry consortia have proposed standards for component software:

WebRef The latest information on CORBA can be obtained at **www.omg.org.**

OMG/CORBA. The Object Management Group has published a *common object request broker architecture* (OMG/CORBA). An *object request broker* (ORB) provides a variety of services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.

WebRef The latest information on COM can be obtained at **www.microsoft.com/COM.**

Microsoft COM. Microsoft has developed a *component object model* (COM) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. COM encompasses two elements: COM interfaces (implemented as COM objects) and a set of mechanisms for registering and passing messages between COM interfaces.

WebRef
The latest information on Javabeans can be obtained at **java.sun.com/products/javabeans/docs/.**

Sun JavaBeans Components. The JavaBeans component system is a portable, platform independent CBSE infrastructure developed using the Java programming language. The JavaBeans component system encompasses a set of tools, called the *Bean Development Kit* (BDK), that allows developers to (1) analyze how existing Beans (components) work, (2) customize their behavior and appearance, (3) establish mechanisms for coordination and communication, (4) develop custom Beans for use in a specific application, and (5) test and evaluate Bean behavior.

Which of these standards will dominate the industry? There is no easy answer at this time. Although many developers have adopted one of the standards, it is likely that large software organizations may choose to use all three standards, depending on the application categories and platforms that are chosen.

INFO

Object Request Broker Architecture

Client/server systems are implemented using software components (objects) that must be capable of interacting with one another within a single machine (either client or server) or across the network. An *object request broker* (ORB) is "middleware" that enables an object residing on a client to send a message to a method that is encapsulated by an object residing on a server. In essence, the ORB intercepts the message and handles all communication and coordination activities required to find the object to which the message was addressed, invoke its method, pass appropriate data to the object, and transfer the resulting data back to the object that generated the message in the first place.

CORBA, COM, and JavaBeans implement an object request broker philosophy. In this sidebar CORBA will be used to illustrate ORB middleware.

The basic structure of a CORBA architecture is illustrated in Figure 30.2. When CORBA is implemented in a client/server system, objects on both the client and the server are defined using an *interface description language* (IDL), a declarative language that allows a software engineer to define objects, attributes, methods, and the messages required to invoke them. To accommodate a request for a server-resident method by a client-resident object, client and server IDL stubs are created. The stubs provide the gateway through which requests for objects across the c/s system are accommodated.

Because requests for objects across the network occur at run time, a mechanism for storing the object description must be established so that pertinent information about the object and its location are available when needed. The interface repository accomplishes this.

When a client application must invoke a method contained within an object elsewhere in the system, CORBA uses dynamic invocation to (1) obtain pertinent information about the desired method from the interface repository, (2) create a data structure with parameters to be passed to the object, (3) create a request for the object, and (4) invoke the request. The request is then passed to the ORB core—an implementation-specific part of the network operating system that manages requests—and the request is fulfilled.

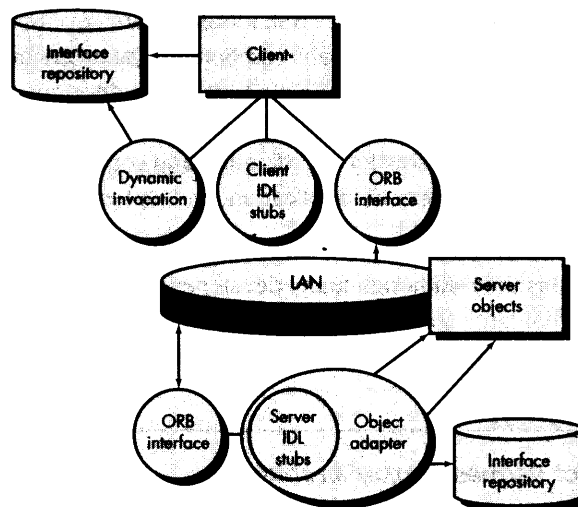
The request is passed through the core and is processed by the server. At the server site, an object adapter stores class and object information in a server-resident interface repository, accepts and manages incoming requests from the client, and performs a variety of other object management functions. At the server, IDL stubs that are similar to those defined at the client machine are used as the interface to the actual object implementation resident at the server site.

30.4.2 Component Engineering

As we noted earlier in this chapter, the CBSE process encourages the use of existing software components. However, there are times when components must be engineered. That is, new software components must be developed and integrated with

FIGURE 30.2

The basic CORBA architecture



existing COTS and in-house components. Because these new components become members of the in-house library of reusable components, they should be engineered for reuse.

Nothing is magical about creating software components that can be reused. Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, SQA, and correctness verification methods, all contribute to the creation of software components that are reusable.[3] In this section we will not revisit these topics. Rather, we consider the reuse-specific issues that are complementary to solid software engineering practices.

30.4.3 Analysis and Design for Reuse

The analysis model is analyzed to determine those elements of the model that point to existing reusable components. The problem is extracting information from the requirements model in a form that can lead to "specification matching."

If specification matching yields components that fit the needs of the current application, the designer can extract these components from a reuse library (repository) and use them in the design of new systems. If design components cannot be found, the software engineer must apply conventional or OO design methods to create them. It is at this point—when the designer begins to create a new component—that *design for reuse* (DFR) should be considered.

As we have already noted, DFR requires the software engineer to apply solid software design concepts and principles (Chapter 9). But the characteristics of the ap-

3 To learn more about these concepts, see Parts 2 and 5 of this book.

DFR can be quite difficult when components must be interfaced or integrated with legacy systems or with multiple systems whose architecture and interfacing protocols are inconsistent.

plication domain must also be considered. Binder [BIN93] suggests a number of key issues[4] that form a basis for design for reuse:

Standard data. The application domain should be investigated and standard global data structures (e.g., file structures or a complete database) should be identified. All design components can then be characterized to make use of these standard data structures.

Standard interface protocols. Three levels of interface protocol should be established: the nature of intramodular interfaces, the design of external technical (nonhuman) interfaces, and the human/machine interface.

Program templates. The structure model (Section 30.3.3) can serve as a template for the architectural design of a new program.

Once standard data, interfaces, and program templates have been established, the designer has a framework in which to create the design. New components that conform to this framework have a higher probability for subsequent reuse.

30.5 Classifying and Retrieving Components

Consider a university library. Tens of thousands of books, periodicals, and other information resources are available for use. But to access these resources, a categorization scheme must be developed. To navigate this large volume of information, librarians have defined a classification scheme that includes a Library of Congress classification code, keywords, author names, and other index entries. All enable the user to find the needed resource quickly and easily.

Now, consider a large component repository. Tens of thousands of reusable software components reside in it. But how does a software engineer find the one she needs? To answer this question, another question arises: How do we describe software components in unambiguous, classifiable terms? These are difficult questions, and no definitive answer has yet been developed. In this section we explore current directions that will enable future software engineers to navigate reuse libraries.

30.5.1 Describing Reusable Components

A reusable software component can be described in many ways, but an ideal description encompasses what Tracz [TRA90] has called the *3C model*—concept, content, and context.

The *concept* of a software component is "a description of what the component does" [WHI95]. The interface to the component is fully described and the semantics—represented within the context of pre- and postconditions—are identified. The concept should communicate the intent of the component.

4 In general, DFR preparations should be undertaken as part of domain engineering (Section 30.3).

The *content* of a component describes how the concept is realized. In essence, the content is information that is hidden from casual users and need be known only to those who intend to modify or test the component.

The *context* places a reusable software component within its domain of applicability. That is, by specifying conceptual, operational, and implementation features, the context enables a software engineer to find the appropriate component to meet application requirements.

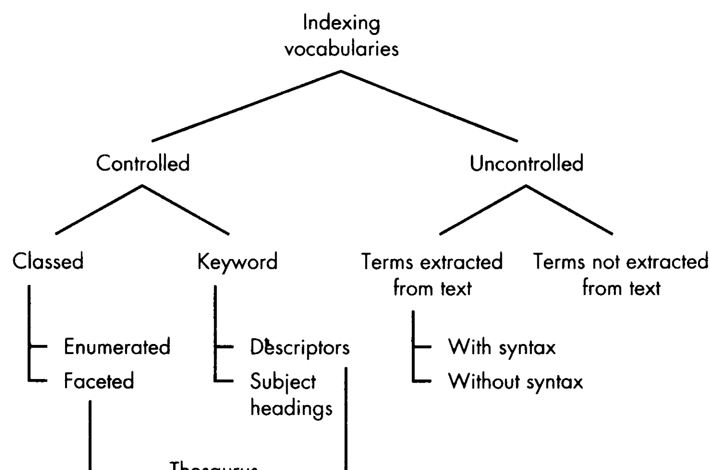
To be of use in a pragmatic setting, concept, content, and context must be translated into a concrete specification scheme. Dozens of papers and articles have been written about classification schemes for reusable software components (e.g., [LUC01] and [WHI95] contain extensive bibliographies). The methods proposed can be categorized into three major areas: library and information science methods, artificial intelligence methods, and hypertext systems. The vast majority of work done to date suggests the use of library science methods for component classification.

Figure 30.3 presents a taxonomy of library science indexing methods. *Controlled indexing vocabularies* limit the terms or syntax that can be used to classify an object (component). *Uncontrolled indexing vocabularies* place no restrictions on the nature of the description. The majority of classification schemes for software components fall into three categories:

Enumerated classification. Components are described by a hierarchical structure in which classes and varying levels of subclasses of software components are defined. The hierarchical structure of an enumerated classification scheme makes it easy to understand and to use. However, before a hierarchy can be built, domain engineering must be conducted so that sufficient knowledge of the proper entries in the hierarchy is available.

FIGURE 30.3

A taxonomy of indexing methods [FRA94]



Faceted classification. A domain area is analyzed and a set of basic descriptive features are identified. These features, called *facets,* are then ranked by importance and connected to a component. A facet can describe the function that the component performs, the data that are manipulated, the context in which they are applied, or any other feature. The set of facets that describe a component is called the *facet descriptor.* Generally, the facet description is limited to no more than seven or eight facets.

Attribute-value classification. A set of attributes is defined for all components in a domain area. Values are then assigned to these attributes in much the same way as faceted classification. In fact, attribute value classification is similar to faceted classification with the following exceptions: (1) no limit is placed on the number of attributes that can be used, (2) attributes are not assigned priorities, and (3) a thesaurus function is not used.

Based on an empirical study of each of these classification techniques, Frakes and Pole [FRA94] indicate that there is no clear "best" technique and that "no method did more than moderately well in search effectiveness. . . ." It would appear that further work remains to be done in the development of effective classification schemes for reuse libraries.

30.5.2 The Reuse Environment

Software component reuse must be supported by an environment that encompasses the following elements:

- A component database capable of storing software components and the classification information necessary to retrieve them.
- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation.

Each of these functions interact with or is embodied within the confines of a reuse library.

The reuse library is one element of a larger software repository (Chapter 27) and provides facilities for the storage of software components and a wide variety of reusable work products (e.g., specifications, designs, patterns, frameworks, code fragments, test cases, user guides). The library encompasses a database and the tools that are necessary to query the database and retrieve components from it. A component classification scheme (Section 30.5.1) serves as the basis for library queries.

Queries are often characterized using the context element of the 3C model described earlier in this section. If an initial query results in a voluminous list of

WebRef
A comprehensive collection of resources on CBSE can be found at **http:// www. cbd-hq.com/.**

candidate components, the query is refined to narrow the list. Concept and content information are then extracted (after candidate components are found) to assist the developer in selecting the proper component.

A detailed discussion of the structure of reuse libraries and the tools that manage them is best left to sources dedicated to the subject. The interested reader should see [FIS00] and [LIN95] for additional information.

SOFTWARE TOOLS

Component-Based Development

Objective: To aid in modeling, design, review, and integration of software components as part of a larger system.

Mechanics: Tools mechanics vary. In general, CBD tools assist in one or more of the following capabilities: specification and modeling of the software architecture; browsing and selection of available software components; integration of components.

Representative Tools[5]

ComponentSource (www.componentsource.com) provides a wide array of COTS software components (and tools) supported within many different component standards.

Component Manager, developed by Flashline (www.flashline.com), "is an application that enables, promotes, and measures software component reuse."

Select Component Factory, developed by Select Business Solutions (www.selectbs.com/products), "is an integrated set of products for software design, design review, service/component management, requirements management, and code generation."

Software Through Pictures-ACD, distributed by Aonix (www.aonix.com), enables comprehensive modeling using UML for the OMG model driven architecture—an open, vendor-neutral approach for CBSE.

30.6 ECONOMICS OF CBSE

WebRef
A variety of articles providing guidelines for CBD and COTS-based systems can be found at **www.sei.cmu.edu.**

Component-based software engineering has an intuitive appeal. In theory, it should provide a software organization with advantages in quality and timeliness. And these should translate into cost savings. But are there hard data that support our intuition?

To answer this question we must first understand what actually can be reused in a software engineering context and then what the costs associated with reuse really are. As a consequence, it is possible to develop a cost/benefit analysis for component reuse.

30.6.1 Impact on Quality, Productivity, and Cost

Considerable evidence from industry case studies (e.g., [ALL02], [HEN95], [MCM95]) indicates substantial business benefits can be derived from aggressive software reuse. Product quality, development productivity, and overall cost are all improved.

Quality. In an ideal setting, a software component that is developed for reuse would be verified to be correct (see Chapter 29) and would contain no defects. In

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

reality, formal verification is not carried out routinely, and defects can and do occur. However, with each reuse, defects are found and eliminated, and a component's quality improves as a result. Over time, the component becomes virtually defect free.

In a study conducted at Hewlett Packard, Lim [LIM94] reports that the defect rate for reused code is 0.9 defects per KLOC, while the rate for newly developed software is 4.1 defects per KLOC. For an application that was composed of 68 percent reused code, the defect rate was 2.0 defects per KLOC—a 51 percent improvement from the expected rate, had the application been developed without reuse. Henry and Faller [HEN95] report a 35 percent improvement in quality. Although anecdotal reports span a reasonably wide spectrum of quality improvement percentages, it is fair to state that reuse provides a nontrivial benefit in terms of the quality and reliability for delivered software.

Productivity. When reusable components are applied throughout the software process, less time is spent creating the plans, models, documents, code, and data that are required to create a deliverable system. It follows that the same level of functionality is delivered to the customer with less input effort. Hence, productivity is improved. Although percentage productivity improvement reports are notoriously difficult to interpret,[6] it appears that 30 to 50 percent reuse can result in productivity improvements in the 25–40 percent range.

The cost to develop a reusable component is often greater that the cost to develop a component that is specific to one application. Be sure that there will be a need for the reusable component in the future. That's where the payoff is realized.

Cost. The net cost savings for reuse are estimated by projecting the cost of the project if it were developed from scratch, C_s, and then subtracting the sum of the costs associated with reuse, C_r, and the actual cost of the software as delivered, C_d.

C_s can be determined by applying one or more of the estimation techniques discussed in Chapter 23. The costs associated with reuse, C_r, include [CHR95]: domain analysis and modeling, domain architecture development, increased documentation to facilitate reuse, support and enhancement of reuse components, royalties and licenses for externally acquired components, creation or acquisition and operation of a reuse repository, and training of personnel in design and construction for reuse. Although costs associated with domain analysis (Section 30.3) and the operation of a reuse repository can be substantial, many of the other costs noted here address issues that are part of good software engineering practice, whether or not reuse is a priority.

30.6.2 Cost Analysis Using Structure Points

In Section 30.3.3, we defined a structure point as an architectural pattern that recurs throughout a particular application domain. A software designer (or system engineer) can develop an architecture for a new application, system, or product by defining a domain architecture and then populating it with structure points. These structure points are either individual reusable components or packages of reusable components.

6 Many extenuating circumstances (e.g., application domain, problem complexity, team structure and size, project duration, technology applied) can have a profound impact on the productivity of the project team.

Even though structure points are reusable, their qualification, adaptation, integration, and maintenance costs are nontrivial. Before proceeding with reuse, the project manager should understand the costs associated with the use of structure points.

Since all structure points (and reusable components in general) have a past history, cost data can be collected for each. In an ideal setting, the qualification, adaptation, integration, and maintenance costs associated with each component in a reuse library is maintained for each instance of usage. These data can then be analyzed to develop projected costs for the next instance of reuse.

As an example, consider a new application, *X*, that requires 60 percent new code and the reuse of three structure points, SP_1, SP_2, and SP_3. Each of these reusable components has been used in a number of other applications, and average costs for qualification, adaptation, integration, and maintenance are available.

To estimate the effort required to deliver *X*, the following must be determined:

$$\text{overall effort} = E_{new} + E_{qual} + E_{adapt} + E_{int}$$

where

E_{new} = effort required to engineer and construct new software components (determined using techniques described in Chapter 23)

E_{qual} = effort required to qualify SP_1, SP_2, and SP_3

E_{adapt} = effort required to adapt SP_1, SP_2, and SP_3

E_{int} = effort required to integrate SP_1, SP_2, and SP_3

The effort required to qualify, adapt, and integrate SP_1, SP_2, and SP_3 is determined by taking the average of historical data collected for qualification, adaptation, and integration of the reusable components in other applications.

30.7 Summary

Component-based software engineering offers inherent benefits in software quality, developer productivity, and overall system cost. And yet, many roadblocks remain to be overcome before the CBSE process model is widely used throughout the industry.

In addition to software components, a variety of reusable artifacts can be acquired by a software engineer. These include technical representations of the software (e.g., specifications, architectural models, designs), documents, patterns, frameworks, test data, and even process-related tasks (e.g., inspection techniques).

The CBSE process encompasses two concurrent subprocesses—domain engineering and component-based development. The intent of domain engineering is to identify, construct, catalog, and disseminate a set of software components in a particular application domain. Component-based development then qualifies, adapts, and integrates these components for use in a new system. In addition, component-

based development engineers new components that are based on the custom requirements of a new system.

Analysis and design techniques for reusable components draw on the same principles and concepts that are part of good software engineering practice. Reusable components should be designed within an environment that establishes standard data structures, interface protocols, and program architectures for each application domain.

Component-based software engineering uses a data exchange model, tools, structured storage, and an underlying object model to construct applications. The object model generally conforms to one or more component standards (e.g., OMG/CORBA) that define the manner in which an application can access reusable objects. Classification schemes enable a developer to find and retrieve reusable components and conform to a model that identifies concept, content, and context. Enumerated classification, faceted classification, and attribute-value classification are representative of many component classification schemes.

The economics of software reuse are addressed by a single question: Is it cost effective to build less and reuse more? In general, the answer is yes, but a software project planner must consider the nontrivial costs associated with the qualification, adaptation, and integration of reusable components.

REFERENCES

[ADL95] Adler, R.M., "Emerging Standards for Component Software, *Computer,* vol. 28, no. 3, March 1995, pp. 68–77.

[ALL02] Allen, P., "CBD Survey: The State of the Practice," *The Cutter Edge,* March, 2002, available at http://www.cutter.com/research/2002/edge020305.html.

[ATK01] Atkinson, C., et al; *Component-Based Product Line Engineering with UML,* Addison-Wesley, 2001.

[BAS94] Basili, V. R., L. C. Briand, and W. M. Thomas, "Domain Analysis for the Reuse of Software Development Experiences," *Proc. of the 19th Annual Software Engineering Workshop,* NASA/GSFC, Greenbelt, MD, December 1994.

[BIN93] Binder, R., "Design for Reuse Is for Real," *American Programmer,* vol. 6, no. 8, August 1993, pp. 30–37.

[BRO96] Brown, A. W., and K. C. Wallnau, "Engineering of Component-Based Systems," *Component-Based Software Engineering,* IEEE Computer Society Press, 1996, pp. 7–15.

[CHR95] Christensen, S. R., "Software Reuse Initiatives at Lockheed," *CrossTalk,* vol. 8, no. 5, May 1995, pp. 26–31.

[CLE95] Clements, P. C., "From Subroutines to Subsystems: Component-Based Software Development," *American Programmer,* vol. 8, No. 11, November 1995.

[DOG03] Dogru, A., and M. Tanik, "A Process Model for Component-Oriented Software Engineering, *IEEE Software,* vol. 20, no. 2, March/April 2003, pp. 34–41.

[FIS00] Fischer, B., "Specification-Based Browsing of Software Component Libraries," *J. Automated Software Engineering,* vol. 7, no. 2, 2000, pp. 179–200, available at http://ase.arc.nasa.gov/people/fischer/papers/ase-00.html.

[FRA94] Frakes, W. B., and T. P. Pole, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Trans. Software Engineering,* vol. SE-20, no. 8, August 1994, pp. 617–630.

[HEI01] Heineman, G., and W. Councill (eds.), *Component-Based Software Engineering,* Addison-Wesley, 2001.

[HEN95] Henry, E., and B. Faller, "Large Scale Industrial Reuse to Reduce Cost and Cycle Time," *IEEE Software,* September 1995, pp. 47–53.

[HUT88] Hutchinson, J. W., and P. G. Hindley, "A Preliminary Study of Large Scale Software Reuse," *Software Engineering Journal,* vol. 3, no. 5, 1988, pp. 208–212.

[LIA93] Liao, H., and Wang, F., "Software Reuse Based on a Large Object-Oriented Library," *ACM Software Engineering Notes,* vol. 18, no. 1, January 1993, pp. 74–80.

[LIM94] Lim, W. C., "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software,* September 1994, pp. 23–30.

[LIN95] Linthicum, D. S., "Component Development (a Special Feature)," *Application Development Trends,* June 1995, pp. 57–78.

[LUC01] deLucena, Jr., V., "Facet-Based Classification Scheme for Industrial Software Components," 2001, can be downloaded from http://research.microsoft.com/ users/cszypers/ events/WCOP2001/Lucena.pdf.

[MCM95] McMahon, P.E., "Pattern-Based Architecture: Bridging Software Reuse and Cost Management," *Crosstalk,* vol. 8, no. 3, March 1995, pp. 10–16.

[ORF96] Orfali, R., D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide,* Wiley, 1996.

[PRI93] Prieto-Diaz, R., "Issues and Experiences in Software Reuse," *American Programmer,* vol. 6, no. 8, August 1993, pp. 10–18.

[POL94] Pollak, W., and M. Rissman, "Structural Models and Patterned Architectures," *Computer,* vol. 27, no. 8, August 1994, pp. 67–68.

[STA94] Staringer, W., "Constructing Applications from Reusable Components," *IEEE Software,* September 1994, pp. 61–68.

[TRA90] Tracz, W., "Where Does Reuse Start?" *Proc. Realities of Reuse Workshop,* Syracuse University CASE Center, January 1990.

[TRA95] Tracz, W., "Third International Conference on Software Reuse—Summary," *ACM Software Engineering Notes,* vol. 20, no. 2, April 1995, pp. 21–22.

[WHI95] Whittle, B., "Models and Languages for Component Description and Reuse," *ACM Software Engineering Notes,* vol. 20, no. 2, April 1995, pp. 76–89.

[YOU98] Yourdon, E. (ed.), "Distributed Objects," *Cutter IT Journal,* vol. 11, no. 12, December 1998.

Problems and Points to Ponder

30.1. Develop a set of domain characteristics that are relevant for word-processing/desktop-publishing software.

30.2. How are characterization functions for application domains and component classification schemes the same? How are they different?

30.3. Do a bit of research on domain engineering and flesh out the process model outlined in Figure 30.1. Identify the tasks that are required for domain analysis and software architecture development.

30.4. Although software components are the most obvious reusable "artifact," many other work products produced as part of software engineering can be reused. Consider project plans and cost estimates. How can these be reused, and what is the benefit of doing so?

30.5. Develop a set of domain characteristics for information systems that are relevant to a university's student data processing.

30.6. One of the key roadblocks to reuse is getting software developers to consider reusing existing components, rather than reinventing new ones (after all, building things is fun!). Suggest three or four different ways that a software organization can provide incentives for software engineers to reuse. What technologies should be in place to support the reuse effort?

30.7. Develop a faceted classification scheme for an application domain assigned by your instructor or one with which you are familiar.

30.8. Acquire information on the most recent CORBA or COM or JavaBeans standard and prepare a three- to five-page paper that discusses its major highlights. Get information on an object request broker tool and illustrate how the tool achieves the standard.

30.9. What is a structure point?

30.10. Develop an enumerated classification for an application domain assigned by your instructor or one with which you are familiar.

30.11. Research the literature to acquire recent quality and productivity data that support the use of CBSE. Present the data to your class.

30.12. Develop a simple structural model for an application domain assigned by your instructor or one with which you are familiar.

Further Readings and Information Sources

Many books on component-based development and component reuse have been published in recent years. Heineman and Councill [HEI01], Brown (*Large Scale Component-Based Development,* Prentice-Hall, 2000), Allen (*Realizing e-Business with Components,* Addison-Wesley, 2000), Herzum and Sims (*Business Component Factory,* Wiley, 1999), and Allen, Frost, and Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective,* Cambridge University Press, 1998) cover all important aspects of the CBSE process. Apperly and his colleagues (*Service- and Component-Based Development,* Addison-Wesley, 2003), Atkinson [ATK01], and Cheesman and Daniels (*UML Components,* Addison-Wesley, 2000) discuss CBSE with a UML emphasis.

Leach (*Software Reuse: Methods, Models, and Costs,* McGraw-Hill, 1997) provides a detailed analysis of cost issues associated with CBSE and reuse. Poulin (*Measuring Software Reuse: Principles, Practices, and Economic Models,* Addison-Wesley, 1996) suggests a number of quantitative methods for assessing the benefits of software reuse.

Dozens of books describing the industry's component-based standards have been published in recent years. These address the inner workings of the standards themselves but also consider many important CBSE topics. A sampling for the three standards discussed in this chapter follows:

CORBA

Bolton, F., *Pure CORBA,* Sams Publishing, 2001.

Doss, G. M., *CORBA Networking With Java,* Wordware Publishing, 1999.

Hoque, R., *CORBA for Real Programmers,* Academic Press/Morgan Kaufmann, 1999.

Siegel, J., *CORBA Fundamentals and Programming,* Wiley, 1999.

Slama, D., J. Garbis, and P. Russell, *Enterprise CORBA,* Prentice-Hall, 1999.

COM

Box, D., K. Brown, T. Ewald, and C. Sells, *Effective COM: 50 Ways to Improve Your COM- and MTS-Based Applications,* Addison-Wesley, 1999.

Gordon, A., *The COM and COM+ Programming Primer,* Prentice-Hall, 2000.

Kirtland, M., *Designing Component-Based Applications,* Microsoft Press, 1999.

Tapadiya, P., *COM+ Programming,* Prentice-Hall, 2000.

Many organizations apply a combination of component standards. Books by Geraghty and his colleagues (*COM-CORBA Interoperability,* Prentice-Hall, 1999), Pritchard (*COM and CORBA Side by Side: Architectures, Strategies, and Implementations,* Addison-Wesley, 1999), and Rosen and his

colleagues (*Integrating CORBA and COM Applications,* Wiley, 1999) consider the issues associated with the use of both CORBA and COM as the basis for component-based development.

JavaBeans

Asbury, S., and S. R. Weiner, *Developing Java Enterprise Applications,* Wiley, 1999.

Anderson, G., and P. Anderson, *Enterprise Javabeans Component Architecture,* Prentice-Hall, 2002.

Monson-Haefel, R., *Enterprise Javabeans,* third edition, O'Reilly & Associates, 2001.

Roman, E., et al., *Mastering Enterprise Javabeans,* 2nd ed., Wiley, 2001.

A wide variety of information sources on component-based software engineering is available on the Internet. An up-to-date list of World Wide Web references can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER 31

REENGINEERING

KEY CONCEPTS

- BPR process model
- C/S architectures
- data structures
- economics
- forward engineering
- inventory analysis
- maintenance
- OO architectures
- reengineering process
- restructuring
- reverse engineering

In a seminal article written for the *Harvard Business Review,* Michael Hammer [HAM90] laid the foundation for a revolution in management thinking about business processes and computing:

> It is time to stop paving the cow paths. Instead of embedding outdated processes in silicon and software, we should obliterate them and start over. We should "reengineer" our businesses: use the power of modern information technology to radically redesign our business processes in order to achieve dramatic improvements in their performance.
>
> Every company operates according to a great many unarticulated rules . . . Reengineering strives to break away from the old rules about how we organize and conduct our business.

Like all revolutions, Hammer's call to arms resulted in both positive and negative changes. During the 1990s, some companies made a legitimate effort to reengineer, and the results led to improved competitiveness. Others relied solely on downsizing and outsourcing (instead of reengineering) to improve their bottom line. Organizations with little potential for future growth often resulted [DEM95].

During this first decade of the twenty-first century, the hype associated with reengineering has waned, but the process itself continues in companies large and small. The nexus between business reengineering and software engineering lies in a system view.

QUICK LOOK

What is it? Consider any technology product that has served you well. You use it regularly, but it's getting old. It breaks too often, takes longer to repair than you'd like, and no longer represents the newest technology. What to do? If the product is hardware, you'll likely throw it away and buy a newer model. But if it's custom-built software, that option may not be available. You'll need to rebuild it. You'll create a product with added functionality, better performance and reliability, and improved maintainability. That's what we call reengineering.

Who does it? At an organizational level, reengineering is performed by business specialists (often consulting companies). At the software level, reengineering is performed by software engineers.

Why is it important? We live in a rapidly changing world. The demands on business functions and the information technology that supports them are changing at a pace that puts enormous competitive pressure on every commercial organization. Both the business and the software that supports (or is) the business must be reengineered to keep pace.

What are the steps? Business process reengineering (BPR) defines business goals, identifies and evaluates existing business processes, and creates revised business processes that better meet current goals. The software reengineering process 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, and testing is applied to uncover errors in content, functionality, and interoperability.

KEY POINT
BPR often results in new software functionality, whereas software reengineering works to replace existing software functionality with better, more maintainable software.

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 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.

31.1 Business Process Reengineering

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?

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

1 The explosion of Web-based applications and systems discussed in Part 3 of this book is indicative of this trend.

31.1.1 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, material 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 work occurs at the bottom of this hierarchy. Be sure, however, that someone has given serious thought to the levels above. If this hasn't been done, your work is at risk.

In Chapter 6, we noted that every system is actually a hierarchy of subsystems. A business is no exception. Each business system (also called a *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 it grow dramatically. For this reason, most BPR efforts focus on individual processes or subprocesses.

> **"As soon as we are shown something old in a new thing, we are pacified."**
>
> **F. W. Nietzsche**

31.1.2 A BPR Model

WebRef
Extensive information on BPR can be found at **www.brint.com/BPR.htm.**

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 31.1. 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.

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.

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. . . .

Another reason for the software maintenance problem is the mobility of software people. It is likely that the software team (or person) that did the original work is no longer around. Worse, subsequent generations of software people have modified the system and moved on. Today, there may be no one left who has any direct knowledge of the legacy system.

As we noted in Chapter 27, 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.

> **"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**

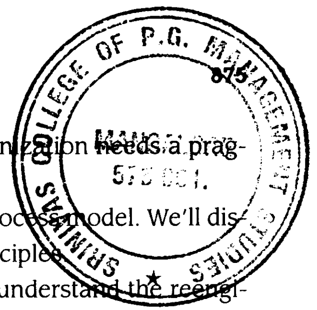
KEY POINT Software maintenance encompasses four activities: error correction, adaptation, enhancement, and reengineering.

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. *Software maintenance* can be defined by identifying four different 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.

WebRef An excellent source of information on software reengineering can be found at **www.reengineering.net.**

31.2.2 A Software Reengineering Process Model

Reengineering takes time, costs significant amounts of money, and 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, you would 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, you would be sure to 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'd gain would serve you well when you start construction.
- If you begin to rebuild, you would use only the most modern, long-lasting materials. This may cost a bit more now, but it would help you to avoid expensive and time-consuming maintenance later.
- If you decide to rebuild, you would be disciplined about it. Use practices that would result in high quality—today and in the future.

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 31.2. 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

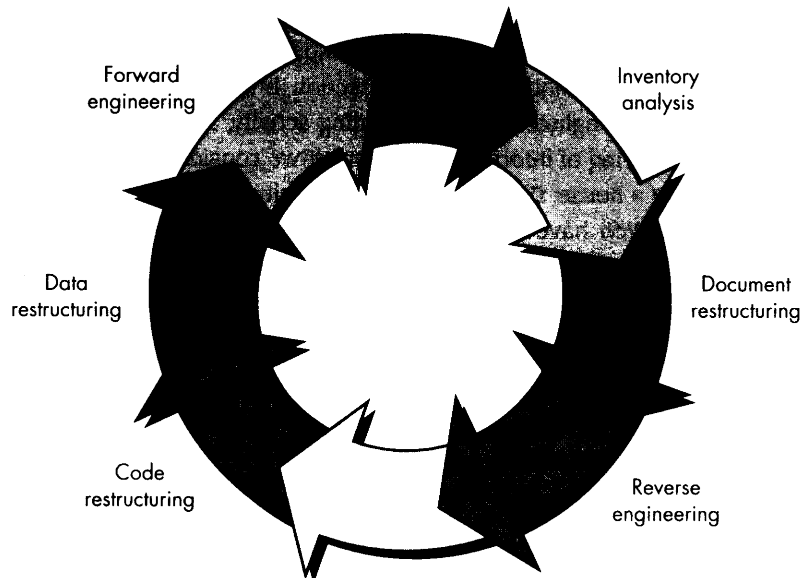


FIGURE 31.2 A software reengineering process model

If time and resources are in short supply, you might consider applying the Pareto principle to the software that is to be engineered. Apply the reengineering process to the 20 percent of the software that accounts for 80 percent of the problems.

Create only as much documentation as you need to understand the software, not one page more.

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.

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.

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 recreate 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

31.3 Reverse Engineering

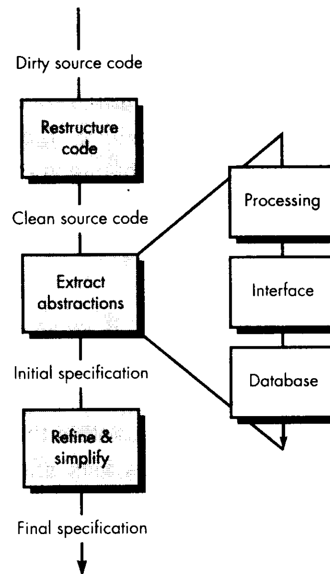
Reverse engineering conjures an image of the "magic slot." We feed a haphazardly designed, undocumented source listing into the slot and out the other end comes a complete design description (and full documentation) for the computer program. Unfortunately, the magic slot doesn't exist. Reverse engineering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

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

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

FIGURE 31.3

The reverse engineering process



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

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

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

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

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

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

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

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

31.3.1 Reverse Engineering to Understand Data

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

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

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

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

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

31.3.2 Reverse Engineering to Understand Processing

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

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