CHAPTER

25 FORMAL METHODS

Software engineering methods can be categorized on a "formality" spectrum that is loosely tied to the degree of mathematical rigor applied during analysis and design. For this reason, the analysis and design methods discussed earlier in this book fall at the informal end of the spectrum. A combination of diagrams, text, tables, and simple notation is used to create analysis and design models, but little mathematical rigor has been applied.

We now consider the other end of the formality spectrum. Here, a specification and design are described using a formal syntax and semantics that specify system function and behavior. The specification is mathematical in form (e.g., predicate calculus can be used as the basis for a formal specification language).

In his introductory discussion of formal methods, Anthony Hall [HAL90] states:

> Formal methods are controversial. Their advocates claim that they can revolutionize [software] development. Their detractors think they are impossibly difficult. Meanwhile, for most people, formal methods are so unfamiliar that it is difficult to judge the competing claims.

In this chapter, we explore formal methods and examine their potential impact on software engineering in the years to come.

QUICK LOOK

What is it? Formal methods allow a software engineer to create a specification that is more complete, consistent, and unambiguous than those produced using conventional or object-oriented methods. Set theory and logic notation are used to create a clear statement of facts (requirements). This mathematical specification can then be analyzed to prove correctness and consistency. Because the specification is created using mathematical notation, it is inherently less ambiguous than informal modes of representation.

Who does it? A specially trained software engineer creates a formal specification.

Why is it important? In safety-critical or mission-critical systems, failure can have a high price. Lives may be lost or severe economic consequences can arise when computer software fails. In such situations, it is essential that errors are uncovered before software is put into operation. Formal methods reduce specification errors dramatically and, as a consequence, serve as the basis for software that has very few errors once the customer begins using it.

What are the steps? The first step in the application of formal methods is to define the data invariant, state, and operations for a system function. The data invariant is a condition that is true throughout the execution of a function that contains a collection of data, The state is the stored data that a

▶

QUICK LOOK

function accesses and alters; and operations are actions that take place in a system as it reads or writes data to a state. An operation is associated with two conditions: a precondition and a postcondition. The notation and heuristics of sets and constructive specification—set operators, logic operators, and sequences—form the basis of formal methods.

What is the work product? A specification represented in a formal language such as Z or VDM is produced when formal methods are applied.

How do I ensure that I've done it right? Because formal methods use discrete mathematics as the specification mechanism, logic proofs can be applied to each system function to demonstrate that the specification is correct.

25.1 BASIC CONCEPTS

The *Encyclopedia of Software Engineering* [MAR94] defines *formal methods* in the following manner:

Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner.

A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and more relevantly, specification, implementation and correctness.

The desired properties of a formal specification—consistency, completeness, and lack of ambiguity—are the objectives of all specification methods. However, the use of formal methods results in a much higher likelihood of achieving these ideals. The formal syntax of a specification language (Section 25.4) enables requirements or design to be interpreted in only one way, eliminating ambiguity that often occurs when a natural language (e.g., English) or a graphical notation must be interpreted by a reader. The descriptive facilities of set theory and logic notation (Section 25.2) enable clear statement of facts (requirements). To be consistent, facts stated in one place in a specification should not be contradicted in another place. Consistency is ensured by mathematically proving that initial facts can be formally mapped (using inference rules) into later statements within the specification.

Quote:

"[F]ormal methods have tremendous potential for improving the clarity and precision of requirements specifications, and in finding important and subtle errors.

Steve Easterbrook et al.

Completeness is difficult to achieve, even when formal methods are used. Some aspects of a system may be left undefined as the specification is being created; other characteristics may be purposely omitted to allow designers some freedom in choosing an implementation approach; and finally, it is impossible to consider every operational scenario in a large, complex system. Things may simply be omitted by mistake.

Although the formalism provided by mathematics has an appeal to some software engineers, others (some would say, the majority) look askance at a mathematical view of software development. To understand why a formal approach has merit, we must first consider the deficiencies associated with less formal approaches.

25.1.1 Deficiencies of Less Formal Approaches[1]

The methods discussed for analysis and design in Parts Three and Four of this book made heavy use of natural language and a variety of graphical notations. Although careful application of analysis and design methods, coupled with thorough review can and does lead to high-quality software, sloppiness in the application of these methods can create a variety of problems. A system specification can contain contradictions, ambiguities, vagueness, incomplete statements, and mixed levels of abstraction.

ADVICE

Although a good document index cannot eliminate contradictions, it can help to uncover them. Spend the time to create a comprehensive index for specifications and other documents.

Contradictions are sets of statements that are at variance with each other. For example, one part of a system specification may state that the system must monitor all the temperatures in a chemical reactor while another part, perhaps written by another member of staff, may state that only temperatures occurring within a certain range are to be monitored. Normally, contradictions that occur on the same page of a system specification can be detected easily. However, contradictions are often separated by a large number of pages.

Ambiguities are statements that can be interpreted in a number of ways. For example, the following statement is ambiguous:

The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited in the login file when an operator logs into the system.

In this extract, does the word *it* refer to the password or the operator identity?

Quote:

"Making mistakes is human, Repeating 'em is too."

Malcolm Forbes

Vagueness often occurs because a system specification is a very bulky document. Achieving a high level of precision consistently is an almost impossible task. It can lead to statements such as "The interface to the system used by radar operators should be user-friendly" or "The virtual interface shall be based on simple overall concepts that are straightforward to understand and use and few in number." A casual perusal of these statements might not detect the underlying lack of any useful information.

Incompleteness is probably one of the most frequently occurring problems with system specifications. For example, consider the functional requirement:

The system should maintain the hourly level of the reservoir from depth sensors situated in the reservoir. These values should be stored for the past six months.

1 This section and others in the first part of this chapter have been adapted from work contributed by Darrel Ince for the European version of the fourth edition of *Software Engineering: A Practitioner's Approach.*

ADVICE

Effective use of formal technical reviews during specification can eliminate many of these problems. However, some will not be uncovered. Therefore, be on the lookout for deficiencies during design, code, and test.

This describes the main data storage part of a system. If one of the commands for the system was

The function of the AVERAGE command is to display on a PC the average water level for a particular sensor between two times.

Assuming that no more detail was presented for this command, the details of the command would be seriously incomplete. For example, the description of the command does not include what should happen if a user of a system specifies a time that was more than six months before the current hour.

Mixed levels of abstraction occur when very abstract statements are intermixed randomly with statements that are at a much lower level of detail. For example, statements such as

The purpose of the system is to track the stock in a warehouse.

might be intermixed with

When the loading clerk types in the *withdraw* command he or she will communicate the order number, the identity of the item to be removed, and the quantity removed. The system will respond with a confirmation that the removal is allowable.

While such statements are important in a system specification, specifiers often manage to intermix them to such an extent that it becomes very difficult to see the overall functional architecture of a system.

Each of these problems is more common than we would like to believe. And each represents a potential deficiency in conventional and object-oriented methods for specification.

25.1.2 Mathematics in Software Development

Mathematics has many useful properties for the developers of large systems. One of its most useful properties is that it is capable of succinctly and exactly describing a physical situation, an object, or the outcome of an action. Ideally, the software engineer should be in the same position as the applied mathematician. A mathematical specification of a system should be presented, and a solution developed in terms of a software architecture that implements the specification should be produced.[2]

Another advantage of using mathematics in the software process is that it provides a smooth transition between software engineering activities. Not only functional specifications but also system designs can be expressed in mathematics, and of course, the program code is a mathematical notation—albeit a rather long-winded one.

2 A word of caution is appropriate at this point. The mathematical system specifications that are presented in this chapter are not as succinct as a simple mathematical expression. Software systems are notoriously complex, and it would be unrealistic to expect that they could be specified in one line of mathematics.

The major property of mathematics is that it supports abstraction and is an excellent medium for modeling. Because it is an exact medium there is little possibility of ambiguity: Specifications can be mathematically validated for contradictions and incompleteness, and vagueness disappears completely. In addition, mathematics can be used to represent levels of abstraction in a system specification in an organized way.

Mathematics is an ideal tool for modeling. It enables the bare bones of a specification to be exhibited and helps the analyst and system specifier to validate a specification for functionality without intrusion of such issues as response time, design directives, implementation directives, and project constraints. It also helps the designer, because the system design specification exhibits the properties of a model, providing only sufficient details to enable the task in hand to be carried out.

Finally, mathematics provides a high level of validation when it is used as a software development medium. It is possible to use a mathematical proof to demonstrate that a design matches a specification and that some program code is a correct reflection of a design. This is preferable to current practice, where often little effort is put into early validation and where much of the checking of a software system occurs during system and acceptance testing.

25.1.3 Formal Methods Concepts

The aim of this section is to present the main concepts involved in the mathematical specification of software systems, without encumbering the reader with too much mathematical detail. To accomplish these, we use a few simple examples.

Example 1: A Symbol Table. A program is used to maintain a symbol table. Such a table is used frequently in many different types of applications. It consists of a collection of items without any duplication. An example of a typical symbol table is shown in Figure 25.1. It represents the table used by an operating system to hold the names of the users of the system. Other examples of tables include the collection of names of staff in a payroll system, the collection of names of computers in a network communications system, and the collection of destinations in a system for producing railway timetables.

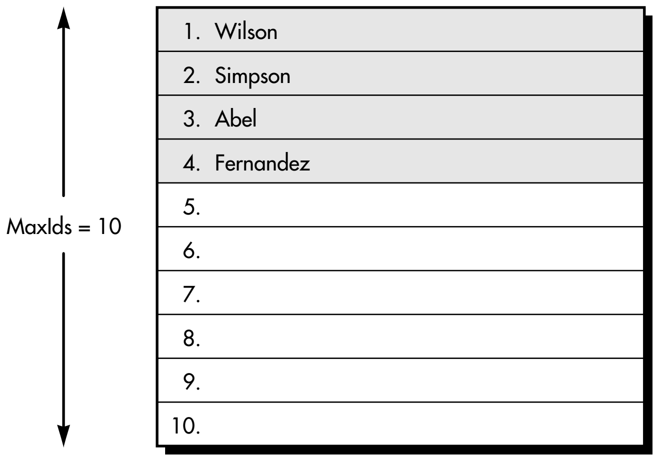
KEY POINT

A data invariant is a set of conditions that are true throughout the execution of the system that contains a collection of data.

Assume that the table presented in this example consists of no more than *MaxIds* members of staff. This statement, which places a constraint on the table, is a component of a condition known as a *data invariant*—an important idea that we shall return to throughout this chapter.

A data invariant is a condition that is true throughout the execution of the system that contains a collection of data. The data invariant that holds for the symbol table just discussed has two components: (1) that the table will contain no more than *MaxIds* names and (2) that there will be no duplicate names in the table. In the case of the symbol table program, this means that, no matter when the symbol table

FIGURE 25.1
A symbol table used for an operating system



KEY POINT
In formal methods, a "state" is stored data that the system accesses and alters. An "operation" is an action that reads or writes data to a state.

is examined during execution of the system, it will always contain no more than *MaxIds* staff identifiers and will contain no duplicates.

Another important concept is that of a *state.* In the context of formal methods,[3] a state is the stored data that a system accesses and alters. In the example of the symbol table program, the state is the symbol table.

The final concept is that of an *operation*. This is an action that takes place in a system and reads or writes data to a state. If the symbol table program is concerned with adding and removing staff names from the symbol table, then it will be associated with two operations: an operation to add a specified name to the symbol table and an operation to remove an existing name from the table. If the program provides the facility to check whether a specific name is contained in the table, then there would be an operation that would return some indication of whether the name is in the table.

KEY POINT
A "precondition" defines the circumstances in which a particular operation is valid. A "postcondition" defines what happens when an operation has completed its action.

An operation is associated with two conditions: a precondition and a postcondition. A *precondition* defines the circumstances in which a particular operation is valid. For example, the precondition for an operation that adds a name to the staff identifier symbol table is valid only if the name that is to be added is not contained in the table and also if there are fewer than MaxIds staff identifiers in the table. The *postcondition* of an operation defines what happens when an operation has completed its action. This is defined by its effect on the state. In the example of an operation that adds an identifier to the staff identifier symbol table, the postcondition would specify mathematically that the table has been augmented with the new identifier.

3 Recall that the term *state* has also been used in Chapters 12 and 21 as a representation of the behavior of a system or objects.

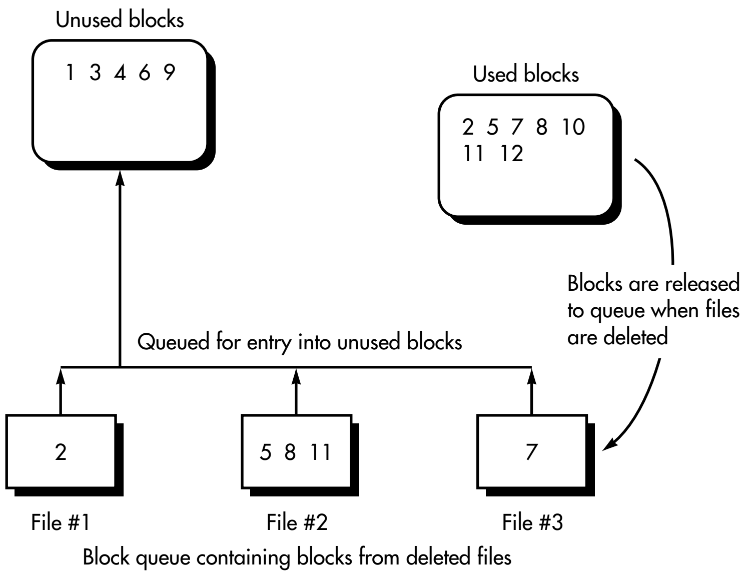

Unused blocks
1 3 4 6 9
Used blocks
2 5 7 8 10
11 12
Blocks are released to queue when files are deleted
Queued for entry into unused blocks
2
5 8 11
7
File #1
File #2
File #3
Block queue containing blocks from deleted files

FIGURE 25.2
A block handler

Example 2: A Block Handler. One of the more important parts of a computer's operating system is the subsystem that maintains files created by users. Part of the filing subsystem is the *block handler.* Files in the file store are composed of blocks of storage that are held on a file storage device. During the operation of the computer, files will be created and deleted, requiring the acquisition and release of blocks of storage. In order to cope with this, the filing subsystem will maintain a reservoir of unused (free) blocks and keep track of blocks that are currently in use. When blocks are released from a deleted file they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks. This is shown in Figure 25.2. In this figure, a number of components are shown: the reservoir of unused blocks, the blocks that currently make up the files administered by the operating system, and those blocks that are waiting to be added to the reservoir. The waiting blocks are held in a queue, with each element of the queue containing a set of blocks from a deleted file.

ADVICE

Brainstorming techniques can work well when you need to develop a data invariant for a reasonably complex function. Have a number of people write down bounds, restrictions, and limitations for the function and then combine and edit.

For this subsystem the state is the collection of free blocks, the collection of used blocks, and the queue of returned blocks. The data invariant, expressed in natural language, is

- No block will be marked as both unused and used.
- All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.
- No elements of the queue will contain the same block numbers.
- The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.

- The collection of unused blocks will have no duplicate block numbers.
- The collection of used blocks will have no duplicate block numbers.

Some of the operations associated with these data are

- An operation that adds a collection of blocks to the end of the queue.
- An operation that removes a collection of used blocks from the front of the queue and places them in the collection of unused blocks.
- An operation that checks whether the queue of blocks is empty.

The precondition of the first operation is that the blocks to be added must be in the collection of used blocks. The postcondition is that the collection of blocks must be added to the end of the queue.

The precondition of the second operation is that the queue must have at least one item in it. The postcondition is that the blocks must be added to the collection of unused blocks.

The final operation—checking whether the queue of returned blocks is empty—has no precondition. This means that the operation is always defined, regardless of what value the state has. The postcondition delivers the value *true* if the queue is empty and *false* otherwise.

Example 3: A Print Spooler. In multitasking operating systems, a number of tasks make requests to print files. Often, there are not enough printing devices to satisfy all current print requests simultaneously. Any print request that cannot be immediately satisfied is placed in a queue awaiting printing. The part of an operating system that deals with the administration of such queues is known as a *print spooler.*

In this example we assume that the operating system can employ no more than *MaxDevs* output devices and that each device has a queue associated with it. We will also assume that each device is associated with a limit of lines in a file which it will print. For example, an output device that has a limit of 1000 lines of printing will be associated with a queue that contains only files having no more than 1000 lines of text. Print spoolers sometimes impose this constraint in order to forbid large print jobs that may occupy slow printing devices for exceptionally long periods. A schematic representation of a print spooler is shown in Figure 25.3.

Referring to the figure, spooler state consists of four components: the queues of files waiting to be printed, each queue being associated with a particular output device; the collection of output devices controlled by the spooler; the relationship between the output devices and the maximum file size that each can print; and the relationship between the files awaiting printing and their size in lines. For example, Figure 25.3 shows that the output device LP1 which has a print limit of 750 lines has two files **ftax** and **persons** awaiting printing, and that the size of the files are 650 lines and 700 lines, respectively.

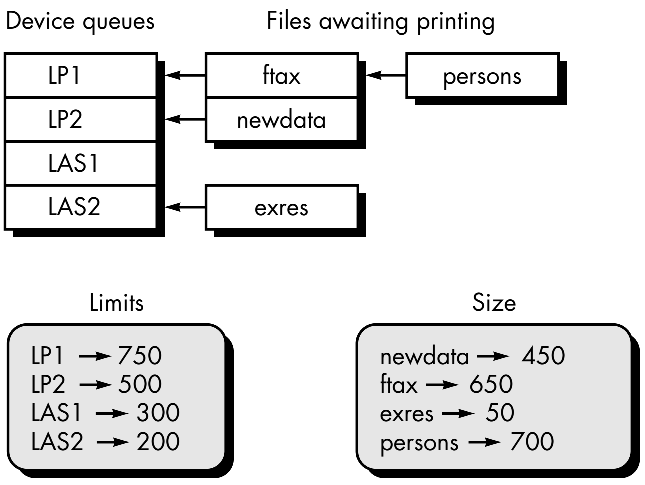


FIGURE 25.3
A print spooler

The state of the spooler is represented by the four components: queues, output devices, limits, and sizes. The data invariant has five components:

- Each output device is associated with an upper limit on print lines.
- Each output device is associated with a possibly nonempty queue of files awaiting printing.
- Each file is associated with a size.
- Each queue associated with an output device contains files that have a size less than the upper limit of the output device.
- There will be no more than *MaxDevs* output devices administered by the spooler.

KEY POINT

States and operations are analogous in many ways to the class definition for OO systems. States represents the data domain (attributes) and operations are the processes (methods) that manipulate the data.

A number of operations can be associated with the spooler. For example,

- An operation that adds a new output device to the spooler together with its associated print limit.
- An operation that removes a file from the queue associated with a particular output device.
- An operation that adds a file to the queue associated with a particular output device.
- An operation that alters the upper limit of print lines for a particular output device.
- An operation that moves a file from a queue associated with an output device to another queue associated with a second output device.

Each of these operations corresponds to a function of the spooler. For example, the first operation would correspond to the spooler being notified of a new device being attached to the computer containing the operating system that administers the spooler.

As before, each operation is associated with a precondition and a postcondition. For example, the precondition for the first operation is that the output device name does not already exist and that there are currently less than *MaxDevs* output devices known to the spooler. The postcondition is that the name of the new device is added to the collection of existing device names, a new entry is formed for the device with no files being associated with its queue, and the device is associated with its print limit.

The precondition for the second operation (removing a file from a queue associated with a particular output device) is that the device is known to the spooler and that at least one entry in the queue is associated with the device. The postcondition is that the head of the queue associated with the output device is removed and its entry in the part of the spooler that keeps tracks of file sizes is deleted.

The precondition for the fifth operation described (moving a file from a queue associated with an output device to another queue associated with a second output device) is

- The first output device is known to the spooler.
- The second output device is known to the spooler.
- The queue associated with the first device contains the file to be moved.
- The size of the file is less than or equal to the print limit associated with the second output device.

The postcondition is that the file is removed from one queue and added to another queue.

In each of the examples noted in this section, we introduce the key concepts of formal specification. But we do so without emphasizing the mathematics that are required to make the specification formal. In the next section, we consider these mathematics.

25.2 MATHEMATICAL PRELIMINARIES

To apply formal methods effectively, a software engineer must have a working knowledge of the mathematical notation associated with sets and sequences and the logical notation used in predicate calculus. The intent of the section is to provide a brief introduction. For a more detailed discussion the reader is urged to examine books dedicated to these subjects (e.g., [WIL87], [GRI93], and [ROS95]).

25.2.1 Sets and Constructive Specification

A set is a collection of objects or elements and is used as a cornerstone of formal methods. The elements contained within a set are unique (i.e., no duplicates are allowed). Sets with a small number of elements are written within curly brackets (braces) with the elements separated by commas. For example, the set

{C++, Pascal, Ada, COBOL, Java}

contains the names of five programming languages.

The order in which the elements appear within a set is immaterial. The number of items in a set is known as its *cardinality*. The # operator returns a set's cardinality. For example, the expression

$\#\{A, B, C, D\} = 4$

implies that the cardinality operator has been applied to the set shown with a result indicating the number of items in the set.

? What is constructive set specification?

There are two ways of defining a set. A set may be defined by enumerating its elements (this is the way in which the sets just noted have been defined). The second approach is to create a *constructive set specification.* The general form of the members of a set is specified using a Boolean expression. Constructive set specification is preferable to enumeration because it enables a succinct definition of large sets. It also explicitly defines the rule that was used in constructing the set.

Consider the following constructive specification example:

$\{n : \mathbb{N} \mid n < 3 \cdot n\}$

This specification has three components, a signature, $n : \mathbb{N}$, a predicate $n < 3$, and a term, *n.* The *signature* specifies the range of values that will be considered when forming the set, the *predicate* (a Boolean expression) defines how the set is to be constricted, and, finally, the *term* gives the general form of the item of the set. In the example above, $\mathbb{N}$ stands for the natural numbers; therefore, natural numbers are to be considered. The predicate indicates that only natural numbers less than 3 are to be included; and the term specifies that each element of the set will be of the form *n.* Therefore, this specification defines the set

{0, 1, 2}

When the form of the elements of a set is obvious, the term can be omitted. For example, the preceding set could be specified as

$(n : \mathbb{N} \mid n < 3\}$

All the sets that have been described here have elements that are single items. Sets can also be made from elements that are pairs, triples, and so on. For example, the set specification

$\{x, y : \mathbb{N} \mid x + y = 10 \bullet (x, y^2)\}$

describes the set of pairs of natural numbers that have the form (x, y^2) and where the sum of x and y is 10. This is the set

$\{ (1, 81), (2, 64), (3, 49), \ldots\}$

Obviously, a constructive set specification required to represent some component of computer software can be considerably more complex than those noted here. However, the basic form and structure remain the same.

25.2.2 Set Operators

A specialized set of symbology is used to represent set and logic operations. These symbols must be understood by the software engineer who intends to apply formal methods.

ADVICE

Knowledge of set operations is indispensible when formal specifications are developed. Spend the time to familiarize yourself with each, if you intend to apply formal methods.

The $\in$ operator is used to indicate membership of a set. For example, the expression

$x \in X$

has the value *true* if x is a member of the set X and the value *false* otherwise. For example, the predicate

$12 \in \{6, 1, 12, 22\}$

has the value *true* since 12 is a member of the set.

The opposite of the $\in$ operator is the $\notin$ operator. The expression

$x \notin X$

has the value *true* if x is not a member of the set X and *false* otherwise. For example, the predicate

$13 \notin \{13, 1, 124, 22\}$

has the value *false*.

The operators $\subset$ and $\subseteq$ take sets as their operands. The predicate

$A \subset B$

has the value *true* if the members of the set A are contained in the set B and has the value *false* otherwise. Thus, the predicate

$\{1, 2\} \subset \{4, 3, 1, 2\}$

has the value *true.* However, the predicate

$\{\text{HD1, LP4, RC5}\} \subset \{\text{HD1, RC2, HD3, LP1, LP4, LP6}\}$

has a value of *false* because the element RC5 is not contained in the set to the right of the operator.

The operator $\subseteq$ is similar to $\subset$. However, if its operands are equal, it has the value *true.* Thus, the value of the predicate

{HD1, LP4, RC5} $\subseteq$ {HD1, RC2, HD3, LP1, LP4, LP6}

is *false,* and the predicate

{HD1, LP4, RC5} $\subseteq$ {HD1, LP4, RC5}

is *true.*

A special set is the empty set $\varnothing$. This corresponds to zero in normal mathematics. The empty set has the property that it is a subset of every other set. Two useful identities involving the empty set are

$\varnothing \cup A = A$ and $\varnothing \cap A = \varnothing$

for any set *A,* where $\cup$ is known as the *union operator,* sometimes known as *cup*; $\cap$ is the *intersection operator,* sometimes known as *cap.*

Quote:
"Mathematical structures are among the most beautiful discoveries made by the human mind."
Douglas Hofstadter

The union operator takes two sets and forms a set that contains all the elements in the set with duplicates eliminated. Thus, the result of the expression

{File1, File2, Tax, Compiler} $\cup$ {NewTax, D2, D3, File2}

is the set

{Filel, File2, Tax, Compiler, NewTax, D2, D3}

The intersection operator takes two sets and forms a set consisting of the common elements in each set. Thus, the expression

{12, 4, 99, 1} $\cap$ {1, 13, 12, 77}

results in the set {12, 1}.

The *set difference* operator, \, as the name suggests, forms a set by removing the elements of its second operand from the elements of its first operand. Thus, the value of the expression

{New, Old, TaxFile, Sysparam} \ {Old, SysParam}

results in the set {New, TaxFile}.

The value of the expression

{a, b, c, d} $\cap$ {x, y}

will be the empty set $\varnothing$. The operator always delivers a set; however, in this case there are no common elements between its operands so the resulting set will have no elements.

The final operator is the *cross product,* $\times$, sometimes known as the *Cartesian product.* This has two operands which are sets of pairs. The result is a set of pairs where each pair consists of an element taken from the first operand combined with an element from the second operand. An example of an expression involving the cross product is

$$\{1, 2\} \times \{4, 5, 6\}$$

The result of this expression is

$$\{(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6)\}$$

Notice that every element of the first operand is combined with every element of the second operand.

A concept that is important for formal methods is that of a *powerset.* A powerset of a set is the collection of subsets of that set. The symbol used for the powerset operator in this chapter is $\mathbb{P}$. It is a unary operator that, when applied to a set, returns the set of subsets of its operand. For example,

$$\mathbb{P}\,\{1, 2, 3\} = \{\varnothing, \{1\}, (2\}\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

since all the sets are subsets of {1, 2, 3}.

25.2.3 Logic Operators

Another important component of a formal method is logic: the algebra of true and false expressions. The meaning of common logical operators is well understood by every software engineer. However, the logic operators that are associated with common programming languages are written using readily available keyboard symbols. The equivalent mathematical operators to these are

$\wedge$	and
$\vee$	or
$\neg$	not
$\Rightarrow$	implies

Universal quantification is a way of making a statement about the elements of a set that is true for every member of the set. Universal quantification uses the symbol, ∀. An example of its use is

$$\forall\, i, j : \mathbb{N} \bullet i > j \Rightarrow i^2 > j^2$$

which states that for every pair of values in the set of natural numbers, if i is greater than j, then i^2 is greater than j^2.

25.2.4 Sequences

A sequence is a mathematical structure that models the fact that its elements are ordered. A sequence s is a set of pairs whose elements range from 1 to the highest-number element. For example,

{(1, Jones), (2, Wilson), (3, Shapiro), (4, Estavez)} is a sequence. The items that form the first elements of the pairs are collectively known as the *domain* of the sequence and the collection of second elements is known as the *range* of the sequence. In this book, sequences are designated using angle brackets. For example, the preceding sequence would normally be written as

$\langle$ Jones, Wilson, Shapiro, Estavez $\rangle$

Unlike sets, duplication in a sequence is allowed and the ordering of a sequence is important. Therefore,

$\langle$ Jones, Wilson, Shapiro $\rangle \neq \langle$ Jones, Shapiro, Wilson $\rangle$

The empty sequence is represented as $\langle\ \rangle$.

A number of sequence operators are used in formal specifications. Catenation, $\frown$, is a binary operator that forms a sequence constructed by adding its second operand to the end of its first operand. For example,

$\langle 2, 3, 34, 1 \rangle \frown \langle 12, 33, 34, 200 \rangle$

results in the sequence $\langle 2, 3, 34, 1, 12, 33, 34, 200 \rangle$.

Other operators that can be applied to sequences are *head, tail, front,* and *last.* The operator *head* extracts the first element of a sequence; *tail* returns with the last $n - 1$ elements in a sequence of length n; *last* extracts the final element in a sequence; and *front* returns with the first $n - 1$ elements in a sequence of length n. For example,

$head\langle 2, 3, 34, 1, 99, 101 \rangle = 2$
$tail\langle 2, 3, 34, 1, 99, 101 \rangle = \langle 3, 34, 1, 99, 101 \rangle$
$last\langle 2, 3, 34, 1, 99, 101 \rangle = 101$
$front\langle 2, 3, 34, 1, 99, 101 \rangle = \langle 2, 3, 34, 1, 99 \rangle$

Since a sequence is set of pairs, all set operators described in Section 25.2.2 are applicable. When a sequence is used in a state, it should be designated as such by using the keyword *seq.* For example,

FileList : *seq* FILES
NoUsers : $\mathbb{N}$

describes a state with two components: a sequence of files and a natural number.

25.3 APPLYING MATHEMATICAL NOTATION FOR FORMAL SPECIFICATION

To illustrate the use of mathematical notation in the formal specification of a software component, we revisit the block handler example presented in Section 25.1.3. To review, an important component of a computer's operating system maintains files that have been created by users. The block handler maintains a reservoir of unused blocks and will also keep track of blocks that are currently in use. When blocks are released from a deleted file they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks. This has been depicted schematically in Figure 25.2.[4]

How can I represent states and data invariants using the set and logic operators that have already been introduced?

A set named *BLOCKS* will consist of every block number. *AllBlocks* is a set of blocks that lie between 1 and *MaxBlocks*. The state will be modeled by two sets and a sequence. The two sets are *used* and *free.* Both contain blocks—the *used* set contains the blocks that are currently used in files and the *free* set contains blocks that are available for new files. The sequence will contain sets of blocks that are ready to be released from files that have been deleted. The state can be described as

$$used, free: \mathbb{P}\ BLOCKS$$
$$BlockQueue: \mathrm{seq}\ \mathbb{P}\ BLOCKS$$

This is very much like the declaration of program variables. It states that *used* and *free* will be sets of blocks and that *BlockQueue* will be a sequence, each element of which will be a set of blocks. The data invariant can be written as

$$used \cap free = \varnothing \land$$
$$used \cup free = AllBlocks \land$$
$$\forall\, i: \mathrm{dom}\ BlockQueue \bullet BlockQueue\ i \subseteq used \land$$
$$\forall\, i, j: \mathrm{dom}\ BlockQueue \bullet i \neq j \Rightarrow BlockQueue\ i \cap BlockQueue\ j = \varnothing$$

WebRef
Extensive information on formal methods can be found at **archive.comlab.ox.ac.uk/formal-methods.html**

The mathematical components of the data invariant match four of the bulleted, natural-language components described earlier. The first line of the data invariant states that there will be no common blocks in the used collection and free collections of blocks. The second line states that the collection of used blocks and free blocks will always be equal to the whole collection of blocks in the system. The third line indicates the *i*th element in the block queue will always be a subset of the used blocks. The final line states that, for any two elements of the block queue that are not the same, there will be no common blocks in these two elements. The final two natural language components of the data invariant are implemented by virtue of the fact that used and free are sets and therefore will not contain duplicates.

The first operation we shall define is one that removes an element from the head of the block queue. The precondition is that there must be at least one item in the queue:

$$\#BlockQueue > 0,$$

The postcondition is that the head of the queue must be removed and placed in the collection of free blocks and the queue adjusted to show the removal:

4 If your recollection of the block handler example is hazy, please return to Section 25.1.3 to review the data invariant, operations, preconditions, and postconditions associated with the block handler.

$$used' = used \setminus head\ BlockQueue \wedge$$
$$free' = free \cup head\ BlockQueue \wedge$$
$$BlockQueue' = tail\ BlockQueue$$

A convention used in many formal methods is that the value of a variable after an operation is primed. Hence, the first component of the preceding expression states that the new used blocks (*used'*) will be equal to the old used blocks minus the blocks that have been removed. The second component states that the new free blocks (*free'*) will be the old free blocks with the head of the block queue added to it. The third component states that the new block queue will be equal to the tail of the old value of the block queue; that is, all elements in the queue apart from the first one. A second operation adds a collection of blocks, *Ablocks,* to the block queue. The precondition is that *Ablocks* is currently a set of used blocks:

? How do I represent pre- and post-conditions?

$$Ablocks \subseteq used$$

The postcondition is that the set of blocks is added to the end of the block queue and the set of used and free blocks remains unchanged:

$$BlockQueue' = BlockQueue \frown \langle Ablocks \rangle \wedge$$
$$used' = used \wedge$$
$$free' = free$$

There is no question that the mathematical specification of the block queue is considerably more rigorous that a natural language narrative or a graphical model. The additional rigor requires effort, but the benefits gained from improved consistency and completeness can be justified for many types of applications.

25.4 FORMAL SPECIFICATION LANGUAGES

A formal specification language is usually composed of three primary components: (1) a syntax that defines the specific notation with which the specification is represented, (2) semantics to help define a "universe of objects" [WIN90] that will be used to describe the system, and (3) a set of relations that define the rules that indicate which objects properly satisfy the specification.

The *syntactic domain* of a formal specification language is often based on a syntax that is derived from standard set theory notation and predicate calculus. For example, variables such as *x, y,* and *z* describe a set of objects that relate to a problem (sometimes called the *domain of discourse*) and are used in conjunction with the operators described in Section 25.2. Although the syntax is usually symbolic, icons (e.g., graphical symbols such as boxes, arrows, and circles) can also be used, if they are unambiguous.

The *semantic domain* of a specification language indicates how the language represents system requirements. For example, a programming language has a set of formal semantics that enables the software developer to specify algorithms that transform input to output. A formal grammar (such as BNF) can be used to describe the syntax of the programming language. However, a programming language does not make a good specification language because it can represent only computable functions. A specification language must have a semantic domain that is broader; that is, the semantic domain of a specification language must be capable of expressing ideas such as, "For all *x* in an infinite set *A*, there exists a *y* in an infinite set *B* such that the property *P* holds for *x* and *y*" [WIN90]. Other specification languages apply semantics that enable the specification of system behavior. For example, a syntax and semantics can be developed to specify states and state transition, events and their effect on state transition, synchronization and timing.

It is possible to use different semantic abstractions to describe the same system in different ways. We did this in a less formal fashion in Chapters 12 and 21. Data flow and corresponding processing were described using the data flow diagram, and system behavior was depicted with the state transition diagram. Analogous notation was used to describe object-oriented systems. Different modeling notation can be used to represent the same system. The semantics of each representation provides complementary views of the system. To illustrate this approach when formal methods are used, assume that a formal specification language is used to describe the set of events that cause a particular state to occur in a system. Another formal relation depicts all functions that occur within a given state. The intersection of these two relations provides an indication of the events that will cause specific functions to occur.

A variety of formal specification languages are in use today. CSP ([HIN95], [HOR85]), LARCH [GUT93], VDM [JON91], and Z ([SPI88], [SPI92]) are representative formal specification languages that exhibit the characteristics noted previously. In this chapter, the Z specification language is used for illustrative purposes. Z is coupled with an automated tool that stores axioms, rules of inference, and application-oriented theorems that lead to mathematical proof of correctness of the specification.

25.5 USING Z TO REPRESENT AN EXAMPLE SOFTWARE COMPONENT

Z specifications are structured as a set of schemas—a boxlike structure that introduces variables and specifies the relationship between these variables. A schema is essentially the formal specification analog of the programming language subroutine or procedure. In the same way that procedures and subroutines are used to structure a system, schemas are used to structure a formal specification.

In this section, we use the Z specification language to model the block handler example, introduced in Section 25.1.3 and discussed further in Section 25.3. A summary of Z language notation is presented in Table 25.1. The following example of a schema describes the state of the block handler and the data invariant:

TABLE 25.1 Summary of Z Notation

Z notation is based on typed set theory and first-order logic. Z provides a construct, called a *schema,* to describe a specification's state space and operations. A schema groups variable declarations with a list of predicates that constrain the possible value of a variable. In Z, the schema X is defined by the form

```
———X———————————————————————————
    declarations
———————————————————————————————
    predicates
———————————————————————————————
```

Global functions and constants are defined by the form

```
    declarations
———————————————————————————————
    predicates
```

The declaration gives the type of the function or constant, while the predicate gives it value. Only an abbreviated set of Z symbols is presented in this table.

Sets:

$S : \mathbb{P}\, X$	*S* is declared as a set of *X*s.
$x \in S$	*x* is a member of *S*.
$x \notin S$	*x* is not a member of *S*.
$S \subseteq T$	*S* is a subset of *T*: Every member of *S* is also in *T*.
$S \cup T$	The union of *S* and *T*: It contains every member of *S* or *T* or both.
$S \cap T$	The intersection of *S* and *T:* It contains every member of both *S* and *T*.
$S \setminus T$	The difference of *S* and *T*: It contains every member of *S* except those also in *T*.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just *x*.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S : \mathbb{F}\, X$	*S* is declared as a finite set of *X*s.
max (*S*)	The maximum of the nonempty set of numbers *S*.

Functions:

$f{:}X ⤔ Y$	*f* is declared as a partial injection from *X* to *Y*
dom *f*	The domain of *f:* the set of values *x* for which *f*(*x*) is defined.
ran *f*	The range of *f*: the set of values taken by *f*(*x*) as *x* varies over the domain of *f*.
$f \oplus \{x \mapsto y\}$	A function that agrees with *f* except that *x* is mapped to *y*.
$\{x\} \trianglelefteq f$	A function like *f*, except that *x* is removed from its domain.

Logic:

$P \wedge Q$	*P* and *Q:* It is true if both *P* and *Q* are true.
$P \Rightarrow Q$	*P* implies *Q:* It is true if either *Q* is true or *P* is false.
$\theta\, S' = \theta\, S$	No components of schema *S* change in an operation.

———*BlockHandler*———————————

$used, free : \mathbb{P}\ BLOCKS$

$BlockQueue : \mathrm{seq}\ \mathbb{P}\ BLOCKS$

————————————————————

$used \cap free = \varnothing \wedge$

$used \cup free = AllBlocks \wedge$

$\forall i : \mathrm{dom}\ BlockQueue \bullet BlockQueue\ i \subseteq \mathrm{used} \wedge$

$\forall i, j : \mathrm{dom}\ BlockQueue \bullet i \neq j \Rightarrow$

$BlockQueue\ i \cap BlockQueue\ j = \varnothing$

————————————————————

WebRef
Detailed information on the Z language, including FAQ, can be found at **archive.comlab.ox.ac.uk/z.html**

The schema consists of two parts. The part above the central line represents the variables of the state, while the part below the central line describes the data invariant. Whenever the schema representing the data invariant and state is used in another schema it is preceded by the **Δ** symbol. Therefore, if the preceding schema is used in a schema that, for example, describes an operation, then it would be written as **Δ***BlockHandler.* As the last sentence implies, schemas can be used to describe operations. The following example of a schema describes the operation that removes an element from the block queue:

———*RemoveBlock*———————————

$\Delta BlockHandler$

————————————————————

$\#BlockQueue > 0,$

$used' = used \setminus head\ BlockQueue \wedge$

$free' = free \cup head\ BlockQueue \wedge$

$BlockQueue' = tail\ BlockQueue$

————————————————————

The inclusion of **Δ***BlockHandler* results in all variables that make up the state being available for the *RemoveBlock* schema and ensures that the data invariant will hold before and after the operation has been executed.

The second operation, which adds a collection of blocks to the end of the queue, is represented as

———*AddBlock*———————————

$\Delta BlockHandler$

$Ablocks? : \mathrm{BLOCKS}$

————————————————————

$Ablocks? \subseteq used$

$BlockQueue' = BlockQueue \frown \langle Ablocks? \rangle$

$used' = used \wedge$

$free' = free$

————————————————————

By convention in Z, an input variable that is read from and does not form part of the state is terminated by a question mark. Thus, *Ablocks*?, which acts as an input parameter, is terminated by a question mark.

25.6 THE TEN COMMANDMENTS OF FORMAL METHODS

The decision to use of formal methods in the real world is not one that is taken lightly. Bowan and Hinchley [BOW95] have coined "the ten commandments of formal methods" as a guide for those who are about to apply this important software engineering approach.[5]

ADVICE

The decision to use formal methods should not be taken lightly. Follow these "commandments" and be sure that everyone has received proper training.

1. **Thou shalt choose the appropriate notation.** In order to choose effectively from the wide array of formal specification languages, a software engineer should consider language vocabulary, application type to be specified, and breadth of usage of the language.
2. **Thou shalt formalize but not overformalize.** It is generally not necessary to apply formal methods to every aspect of a major system. Those components that are safety critical are first choices, followed by components whose failure cannot be tolerated (for business reasons).
3. **Thou shalt estimate costs.** Formal methods have high startup costs. Training staff, acquisition of support tools, and use of contract consultants result in high first-time costs. These costs must be considered when examining the return on investment associated with formal methods.
4. **Thou shalt have a formal methods guru on call.** Expert training and ongoing consulting is essential for success when formal methods are used for the first time.

WebRef

Useful information on formal methods can be obtained at **www.cl.cam.uk/users/mgh1001**

5. **Thou shalt not abandon thy traditional development methods.** It is possible, and in many cases desirable, to integrate formal methods with conventional or object-oriented methods (Chapters 12 and 21). Each has strengths and weakness. A combination, if properly applied, can produce excellent results.[6]
6. **Thou shalt document sufficiently.** Formal methods provide a concise, unambiguous, and consistent method for documenting system requirements. However, it is recommended that a natural language commentary accompany the formal specification to serve as a mechanism for reinforcing the reader's understanding of the system.

5 This treatment is a much abbreviated version of [BOW95].
6 Cleanroom software engineering (Chapter 26) is an example of an integrated approach that uses formal methods and more conventional development methods.

7. **Thou shalt not compromise thy quality standards.** "There is nothing magical about formal methods" [BOW95] and for this reason, other SQA activities (Chapter 8) must continue to be applied as systems are developed.
8. **Thou shalt not be dogmatic.** A software engineer must recognize that formal methods are not a guarantee of correctness. It is possible (some would say, likely) that the final system, even when developed using formal methods, may have small omissions, minor bugs, and other attributes that do not meet expectations.
9. **Thou shalt test, test, and test again.** The importance of software testing has been discussed in Chapters 17, 18, and 23. Formal methods do not absolve the software engineer from the need to conduct well-planned, thorough tests.
10. **Thou shalt reuse.** Over the long term, the only rational way to reduce software costs and increase software quality is through reuse (Chapter 27). Formal methods do not change this reality. In fact, it may be that formal methods are an appropriate approach when components for reuse libraries are to be created.

25.7 FORMAL METHODS—THE ROAD AHEAD

Although formal, mathematically based specification techniques are not as yet used widely in the industry, they do offer substantial advantages over less formal techniques. Liskov and Bersins [LIS86] summarize these in the following way:

> Formal specifications can be studied mathematically while informal specifications cannot. For example, a correct program can be proved to meet its specifications, or two alternative sets of specifications can be proved equivalent . . . Certain forms of incompleteness or inconsistency can be detected automatically.

In addition, formal specification removes ambiguity and encourages greater rigor in the early stages of the software engineering process.

But problems remain. Formal specification focuses primarily on function and data. Timing, control, and behavioral aspects of a problem are more difficult to represent. In addition, some elements of a problem (e.g., human/machine interfaces) are better specified using graphical techniques or prototypes. Finally, specification using formal methods is more difficult to learn than methods such as structured analysis and represents a significant "culture shock" for some software practitioners. For this reason, it is likely that formal, mathematical specification techniques will form the foundation for a future generation of CASE tools. When and if this occurs, mathematically based specification may be adopted by a wider segment of the software engineering community.[7]

7 It is important to note that others disagree. See [YOU94].

25.8 SUMMARY

Formal methods provide a foundation for specification environments leading to analysis models that are more complete, consistent, and unambiguous than those produced using conventional or object-oriented methods. The descriptive facilities of set theory and logic notation enable a software engineer to create a clear statement of facts (requirements).

The underlying concepts that govern formal methods are, (1) the data invariant, a condition true throughout the execution of the system that contains a collection of data; (2) the state, the stored data that a system accesses and alters; and (3) the operation, an action that takes place in a system and reads or writes data to a state. An operation is associated with two conditions: a precondition and a postcondition.

Discrete mathematics—the notation and heuristics associated with sets and constructive specification, set operators, logic operators, and sequences—forms the basis of formal methods. Discrete mathematics is implemented in the context of a formal specification language, such as Z.

Z, like all formal specification languages, has both syntactic and semantic domains. The syntactic domain uses a symbology that is closely aligned with the notation of sets and predicate calculus. The semantic domain enables the language to express requirements in a concise manner. The structure of Z incorporates schemas—box-like structures that introduce variables and specify the relationship between these variables.

A decision to use formal methods must consider startup costs as well as the cultural changes associated with a radically different technology. In most instances, formal methods have highest payoff for safety-critical and business-critical systems.

REFERENCES

[BOW95] Bowan, J.P. and M.G. Hinchley, "Ten Commandments of Formal Methods," *Computer,* vol. 28, no. 4, April 1995.

[GRI93] Gries, D. and F.B. Schneider, *A Logical Approach to Discrete Math,* Springer-Verlag, 1993.

[GUT93] Guttag, J.V., and J.J. Horning, *Larch: Languages and Tools for Formal Specification,* Springer-Verlag, 1993.

[HAL90] Hall, A., "Seven Myths of Formal Methods," *IEEE Software,* September 1990, pp. 11–20.

[HIN95] Hinchley, M.G. and S.A. Jarvis, *Concurrent Systems: Formal Development in CSP,* McGraw-Hill, 1995.

[HOR85] Hoare, C.A.R., *Communicating Sequential Processes,* Prentice-Hall International, 1985.

[JON91] Jones, C.B., *Systematic Software Development Using VDM,* 2nd ed., Prentice-Hall, 1991.

[LIS86] Liskov, B.H., and V. Berzins, "An Appraisal of Program Specifications," in *Software Specification Techniques,* N. Gehani and A.T. McKittrick (eds.), Addison-Wesley, 1986, p. 3.

[MAR94] Marciniak, J.J. (ed.), *Encyclopedia of Software Engineering,* Wiley, 1994.

[ROS95] Rosen, K.H., *Discrete Mathematics and Its Applications,* 3rd ed., McGraw-Hill, 1995.

[SPI88] Spivey, J.M., *Understanding Z: A Specification Language and Its Formal Semantics,* Cambridge University Press, 1988.

[SPI92] Spivey, J.M., *The Z Notation: A Reference Manual,* Prentice-Hall, 1992.

[WIL87] Wiltala, S.A., *Discrete Mathematics: A Unified Approach,* McGraw-Hill, 1987.

[WIN90] Wing, J.M., "A Specifier's Introduction to Formal Methods," *Computer,* vol. 23, no. 9, September 1990, pp. 8–24.

[YOU94] Yourdon, E., "Formal Methods," *Guerrilla Programmer,* Cutter Information Corp., October 1994.

PROBLEMS AND POINTS TO PONDER

25.1. Review the types of deficiencies associated with less formal approaches to software engineering in Section 25.1.1. Provide three examples of each from your own experience.

25.2. The benefits of mathematics as a specification mechanism have been discussed at length in this chapter. Is there a downside?

25.3. You have been assigned to a team that is developing software for a fax modem. Your job is to develop the "phone book" portion of the application. The phone book function enables up to *MaxNames* people to be stored along with associated company names, fax numbers, and other related information. Using natural language, define

a. The data invariant.

b. The state.

c. The operations that are likely.

25.4. You have been assigned to a software team that is developing software, called *MemoryDoubler,* that provides greater apparent memory for a PC than physical memory. This is accomplished by identifying, collecting, and reassigning blocks of memory that have been assigned to an existing application but are not being used. The unused blocks are reassigned to applications that require additional memory. Making appropriate assumptions and using natural language, define

a. The data invariant.

b. The state.

c. The operations that are likely.

25.5. Develop a constructive specification for a set that contains tuples of natural numbers of the form (x, y, z^2) such that the sum of x and y equals z.

25.6. The installer for a PC-based application first determines whether an acceptable set of hardware and systems resources is present. It checks the hardware configuration to determine whether various devices (of many possible devices) are present, and determines whether specific versions of system software and drivers are already installed. What set operator could be used to accomplish this? Provide an example in this context.

25.7. Attempt to develop a expression using logic and set operators for the following statement: "For all x and y, if x is the parent of y and y is the parent of z, then x is the grandparent of z. Everyone has a parent." Hint: Use the function $P(x, y)$ and $G(x, z)$ to represent parent and grandparent functions, respectively.

25.8. Develop a constructive set specification of the set of pairs where the first element of each pair is the sum of two nonzero natural numbers and the second element is the difference between the same numbers. Both numbers should be between 100 and 200 inclusive.

25.9. Develop a mathematical description for the state and data invariant for Problem 25.3. Refine this description in the Z specification language.

25.10. Develop a mathematical description for the state and data invariant for Problem 25.4. Refine this description in the Z specification language.

25.11. Using the Z notation presented in Table 25.1, select some part of the *SafeHome* security system described earlier in this book and attempt to specify it with Z.

25.12. Using one or more of the information sources noted in the references to this chapter or Further Readings and Information Sources, develop a half-hour presentation on the basic syntax and semantics of a formal specification language other than Z.

FURTHER READINGS AND INFORMATION SOURCES

In addition to the books used as references in this chapter, a fairly large number of books on formal methods topics have been published over the past decade. A listing of some of the more useful offerings follows:

Bowan, J., *Formal Specification and Documentation using Z: A Case Study Approach,* International Thomson Computer Press, 1996.

Casey, C., *A Programming Approach to Formal Methods,* McGraw-Hill, 2000.

Cooper, D. and R. Barden, *Z in Practice,* Prentice-Hall, 1995.

Craigen, D., S. Gerhart, and T. Ralston, *Industrial Application of Formal Methods to Model, Design and Analyze Computer Systems,* Noyes Data Corp., 1995.

Diller, A., *Z: An Introduction to Formal Methods,* 2nd ed., Wiley, 1994.

Harry, A., *Formal Methods Fact File: VDM and Z,* Wiley, 1997.

Hinchley, M. and J. Bowan, *Applications of Formal Methods,* Prentice-Hall, 1995.

Hinchley, M. and J. Bowan, *Industrial Strength Formal Methods,* Academic Press, 1997.

Hussmann, H., *Formal Foundations for Software Engineering Methods,* Springer-Verlag, 1997.

Jacky, J., *The Way of Z: Practical Programming with Formal Methods,* Cambridge University Press, 1997.

Lano, J. and H. Haughton (eds.), *Object-Oriented Specification Case Studies,* Prentice-Hall, 1993.

Rann, D., J. Turner, and J. Whitworth, *Z: A Beginner's Guide,* Chapman and Hall, 1994.

Ratcliff, B., *Introducing Specification Using Z: A Practical Case Study Approach,* McGraw-Hill, 1994.

D. Sheppard, *An Introduction to Formal Specification with Z and VDM,* McGraw-Hill, 1995.

The September 1990, issues of *IEEE Transactions on Software Engineering, IEEE Software,* and *IEEE Computer* were dedicated to formal methods. They remain an excellent source of useful information.

Schuman (*Formal Object-Oriented Development,* Springer-Verlag, 1996) has edited a book that addresses formal methods and object technologies, providing guidelines on the selective use of formal methods, and showing how such methods can be used in conjunction with OO approaches. Bowman and Derrick (*Formal Methods for Open Object-Based Distributed Systems,* Kluwer Academic Publishers, 1997) address the use of formal methods when coupled with OO applications in a distributed environment.

A wide variety of information sources on formal methods and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to formal methods can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/formal-methods.mhtml

CHAPTER

26 CLEANROOM SOFTWARE ENGINEERING

KEY CONCEPTS

black-box spec	705
box structure	704
certification	714
cleanroom strategy	701
clear-box spec	706
design refinement	706
functional spec	703
proof of correctness	707
state-box spec	705
statistical use testing	712
stimulus	713
test probability distribution	713
verification	707

The integrated use of conventional software engineering modeling (and possibly formal methods), program verification (correctness proofs), and statistical SQA have been combined into a technique that can lead to extremely high-quality software. *Cleanroom software engineering* is an approach that emphasizes the need to build correctness into software as it is being developed. Instead of the classic analysis, design, code, test, and debug cycle, the cleanroom approach suggests a different point of view [LIN94]:

> The philosophy behind cleanroom software engineering is to avoid dependence on costly defect removal processes by writing code increments right the first time and verifying their correctness before testing. Its process model incorporates the statistical quality certification of code increments as they accumulate into a system.

In many ways, the cleanroom approach elevates software engineering to another level. Like the formal methods presented in Chapter 25, the cleanroom process emphasizes rigor in specification and design, and formal verification of each design element using correctness proofs that are mathematically based. Extending the approach taken in formal methods, the cleanroom approach also emphasizes techniques for statistical quality control, including testing that is based on the anticipated use of the software by customers.

QUICK LOOK

What is it? How many times have you heard someone say "Do it right the first time"? That's the overriding philosophy of cleanroom software engineering—a process that emphasizes mathematical verification of correctness before program construction commences and certification of software reliability as part of the testing activity. The bottom line is extremely low failure rates that would be difficult or impossible to achieve using less formal methods.

Who does it? A specially trained software engineer.

Why is it important? Mistakes create rework. Rework takes time and increases costs. Wouldn't it be nice if we could dramatically reduce the number of mistakes (bugs) introduced as the software is designed and built? That's the premise of cleanroom software engineering.

What are the steps? Analysis and design models are created using box structure representation. A "box" encapsulates the system (or some aspect of the system) at a specific level of abstraction. Correctness verification is applied once the box structure design is complete. Once correctness has been verified for each box structure, statistical usage testing commences. The software is tested by defining a set of usage scenarios, determining the probability of use for each scenario, and then defining random tests that conform to the probabilities. The

QUICK LOOK

error records that result are analyzed to enable mathematical computation of projected reliability for the software component.

What is the work product? Black-box, state-box, and clear-box specifications are developed. The results of formal correctness proofs and statistical use tests are recorded.

How do I ensure that I've done it right? Formal proof of correctness is applied to the box structure specification. Statistical use testing exercises usage scenarios to ensure that errors in user functionality are uncovered and corrected. Test data are used to provide an indication of software reliability.

When software fails in the real world, immediate and long-term hazards abound. The hazards can be related to human safety, economic loss, or effective operation of business and societal infrastructure. Cleanroom software engineering is a process model that removes defects before they can precipitate serious hazards.

26.1 THE CLEANROOM APPROACH

The philosophy of the "cleanroom" in hardware fabrication technologies is really quite simple: It is cost-effective and time-effective to establish a fabrication approach that precludes the introduction of product defects. Rather than fabricating a product and then working to remove defects, the cleanroom approach demands the discipline required to eliminate defects in specification and design and then fabricate in a "clean" manner.

Quote:

"Cleanroom engineering achieves statistical quality control over software development by strictly separating the design process from the testing process in a pipeline of incremental software development."

Harlan Mills

The cleanroom philosophy was first proposed for software engineering by Mills, Dyer, and Linger [MIL87] during the 1980s. Although early experiences with this disciplined approach to software work showed significant promise [HAU94], it has not gained widespread usage. Henderson [HEN95] suggests three possible reasons:

1. A belief that the cleanroom methodology is too theoretical, too mathematical, and too radical for use in real software development.
2. It advocates no unit testing by developers but instead replaces it with correctness verification and statistical quality control—concepts that represent a major departure from the way most software is developed today.
3. The maturity of the software development industry. The use of cleanroom processes requires rigorous application of defined processes in all life cycle phases. Since most of the industry is still operating at the ad hoc level (as defined by the Software Engineering Institute Capability Maturity Model), the industry has not been ready to apply those techniques.

Despite elements of truth in each of these concerns, the potential benefits of cleanroom software engineering far outweigh the investment required to overcome the cultural resistance that is at the core of these concerns.

26.1.1 The Cleanroom Strategy

The cleanroom approach makes use of a specialized version of the incremental software model (Chapter 2). A "pipeline of software increments" [LIN94] is developed by small independent software engineering teams. As each increment is certified, it is integrated in the whole. Hence, functionality of the system grows with time.

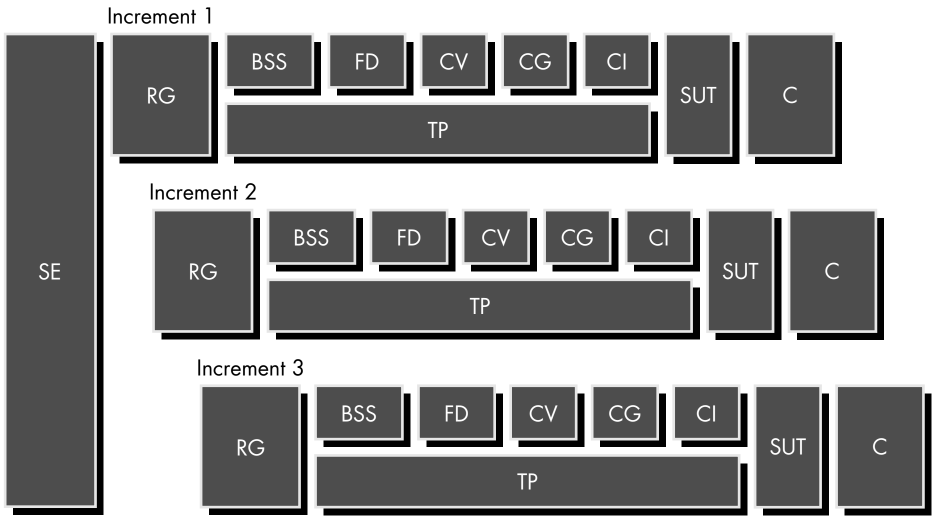
The sequence of cleanroom tasks for each increment is illustrated in Figure 26.1. Overall system or product requirements are developed using the system engineering methods discussed in Chapter 10. Once functionality has been assigned to the software element of the system, the pipeline of cleanroom increments is initiated. The following tasks occur:

? What are the major tasks conducted as part of cleanroom software engineering?

Increment planning. A project plan that adopts the incremental strategy is developed. The functionality of each increment, its projected size, and a cleanroom development schedule are created. Special care must be taken to ensure that certified increments will be integrated in a timely manner.

Requirements gathering. Using techniques similar to those introduced in Chapter 11, a more-detailed description of customer-level requirements (for each increment) is developed.

Box structure specification. A specification method that makes use of box structures [HEV93] is used to describe the functional specification. Conforming



SE — system engineering
RG — requirements gathering
BSS — box structure specification
FD — formal design
CV — correctness verification
CG — code generation
CI — code inspection
SUT — statistical use testing
C — certification
TP — test planning

FIGURE 26.1 The cleanroom process model

to the operational analysis principles discussed in Chapter 11, box structures "isolate and separate the creative definition of behavior, data, and procedures at each level of refinement."

WebRef

An excellent source of information and resources for cleanroom software engineering can be found at **www.cleansoft.com**

Formal design. Using the box structure approach, cleanroom design is a natural and seamless extension of specification. Although it is possible to make a clear distinction between the two activities, specifications (called *black boxes*) are iteratively refined (within an increment) to become analogous to architectural and component-level designs (called *state boxes* and *clear boxes,* respectively).

Correctness verification. The cleanroom team conducts a series of rigorous correctness verification activities on the design and then the code. Verification (Sections 26.3 and 26.4) begins with the highest-level box structure (specification) and moves toward design detail and code. The first level of correctness verification occurs by applying a set of "correctness questions" [LIN88]. If these do not demonstrate that the specification is correct, more formal (mathematical) methods for verification are used.

Quote:

"Quality is not an act. It is a habit."
Aristotle

Code generation, inspection, and verification. The box structure specifications, represented in a specialized language, are translated into the appropriate programming language. Standard walkthrough or inspection techniques (Chapter 8) are then used to ensure semantic conformance of the code and box structures and syntactic correctness of the code. Then correctness verification is conducted for the source code.

Statistical test planning. The projected usage of the software is analyzed and a suite of test cases that exercise a "probability distribution" of usage are planned and designed (Section 26.4). Referring to Figure 26.1, this cleanroom activity is conducted in parallel with specification, verification, and code generation.

ADVICE

Cleanroom emphasizes tests that exercise the way the software is really used. Use-cases provide excellent input to the statistical test planning process.

Statistical use testing. Recalling that exhaustive testing of computer software is impossible (Chapter 17), it is always necessary to design a finite number of test cases. Statistical use techniques [POO88] execute a series of tests derived from a statistical sample (the probability distribution noted earlier) of all possible program executions by all users from a targeted population (Section 26.4).

Certification. Once verification, inspection, and usage testing have been completed (and all errors are corrected), the increment is certified as ready for integration.

Like other software process models discussed elsewhere in this book, the cleanroom process relies heavily on the need to produce high-quality analysis and design models. As we will see later in this chapter, box structure notation is simply another way for a software engineer to represent requirements and design. The real distinction of the cleanroom approach is that formal verification is applied to engineering models.

26.1.2 What Makes Cleanroom Different?

Dyer [DYE92] alludes to the differences of the cleanroom approach when he defines the process:

Cleanroom represents the first practical attempt at putting the software development process under statistical quality control with a well-defined strategy for continuous process improvement. To reach this goal, a cleanroom unique life cycle was defined which focused on mathematics-based software engineering for correct software designs and on statistics-based software testing for certification of software reliability.

KEY POINT

The most important distinguishing characteristics of cleanroom are proof of correctness and statistical use testing.

Cleanroom software engineering differs from the conventional and object-oriented views presented in Parts Three and Four of this book because

1. It makes explicit use of statistical quality control.
2. It verifies design specification using a mathematically based proof of correctness.
3. It relies heavily on statistical use testing to uncover high-impact errors.

Obviously, the cleanroom approach applies most, if not all, of the basic software engineering principles and concepts presented throughout this book. Good analysis and design procedures are essential if high quality is to result. But cleanroom engineering diverges from conventional software practices by deemphasizing (some would say, eliminating) the role of unit testing and debugging and dramatically reducing (or eliminating) the amount of testing performed by the developer of the software.[1]

Quote:

"It's a funny thing about life: if you refuse to accept anything but the best, you very often get it."

W. Somerset Maugham

In conventional software development, errors are accepted as a fact of life. Because errors are deemed to be inevitable, each program module should be unit tested (to uncover errors) and then debugged (to remove errors). When the software is finally released, field use uncovers still more defects and another test and debug cycle begins. The rework associated with these activities is costly and time consuming. Worse, it can be degenerative—error correction can (inadvertently) lead to the introduction of still more errors.

In cleanroom software engineering, unit testing and debugging are replaced by correctness verification and statistically based testing. These activities, coupled with the record keeping necessary for continuous improvement, make the cleanroom approach unique.

26.2 FUNCTIONAL SPECIFICATION

Regardless of the analysis method that is chosen, the operational principles presented in Chapter 11 apply. Data, function, and behavior are modeled. The resultant

1 Testing is conducted but by an independent testing team.

models must be partitioned (refined) to provide increasingly greater detail. The overall objective is to move from a specification that captures the essence of a problem to a specification that provides substantial implementation detail.

Cleanroom software engineering complies with the operational analysis principles by using a method called *box structure specification.* A "box" encapsulates the system (or some aspect of the system) at some level of detail. Through a process of stepwise refinement, boxes are refined into a hierarchy where each box has *referential transparency.* That is, "the information content of each box specification is sufficient to define its refinement, without depending on the implementation of any other box" [LIN94]. This enables the analyst to partition a system hierarchically, moving from essential representation at the top to implementation-specific detail at the bottom. Three types of boxes are used:

? How is refinement accomplished as part of box structure specification?

Black box. The black box specifies the behavior of a system or a part of a system. The system (or part) responds to specific stimuli (events) by applying a set of transition rules that map the stimulus into a response.

State box. The state box encapsulates state data and services (operations) in a manner that is analogous to objects. In this specification view, inputs to the state box (stimuli) and outputs (responses) are represented. The state box also represents the "stimulus history" of the black box; that is, the data encapsulated in the state box that must be retained between the transitions implied.

Clear box. The transition functions that are implied by the state box are defined in the clear box. Stated simply, a clear box contains the procedural design for the state box.

Figure 26.2 illustrates the refinement approach using box structure specification. A black box (BB_1) defines responses for a complete set of stimuli. BB_1 can be refined into a set of black boxes, $BB_{1.1}$ to $BB_{1.n}$, each of which addresses a class of behavior. Refinement continues until a cohesive class of behavior is identified (e.g., $BB_{1.1.1}$). A state box ($SB_{1.1.1}$) is then defined for the black box ($BB_{1.1.1}$). In this case, $SB_{1.1.1}$ contains all data and services required to implement the behavior defined by $BB_{1.1.1}$. Finally, $SB_{1.1.1}$ is refined into clear boxes ($CB_{1.1.1.n}$) and procedural design details are specified.

KEY POINT

Box structure refinement and verification of correctness occur simultaneously.

As each of these refinement steps occurs, verification of correctness also occurs. State-box specifications are verified to ensure that each conforms to the behavior defined by the parent black-box specification. Similarly, clear-box specifications are verified against the parent state box.

It should be noted that specification methods based on formal methods (Chapter 25) can be used in lieu of the box structure specification approach. The only requirement is that each level of specification can be formally verified.

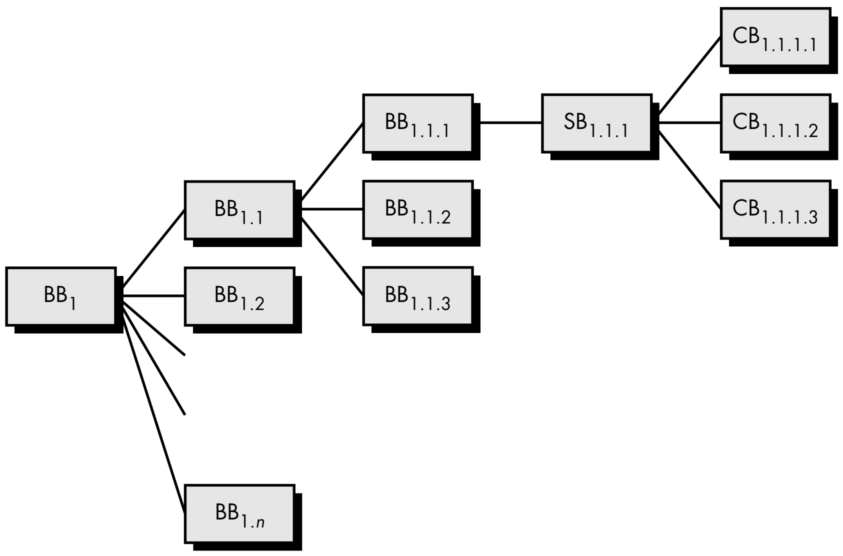


FIGURE 26.2
Box structure refinement

26.2.1 Black-Box Specification

A black-box specification describes an abstraction, stimuli, and response using the notation shown in Figure 26.3 [MIL88]. The function *f* is applied to a sequence, *S**, of inputs (stimuli), *S,* and transforms them into an output (response), *R.* For simple software components, *f* may be a mathematical function, but in general, *f* is described using natural language (or a formal specification language).

XRef
Object-oriented concepts are discussed in Chapter 20.

Many of the concepts introduced for object-oriented systems are also applicable for the black box. Data abstractions and the operations that manipulate those abstractions are encapsulated by the black box. Like a class hierarchy, the black box specification can exhibit usage hierarchies in which low-level boxes inherit the properties of those boxes higher in the tree structure.

26.2.2 State-Box Specification

The state box is "a simple generalization of a state machine" [MIL88]. Recalling the discussion of behavioral modeling and state transition diagrams in Chapter 12, a state is some observable mode of system behavior. As processing occurs, a system responds

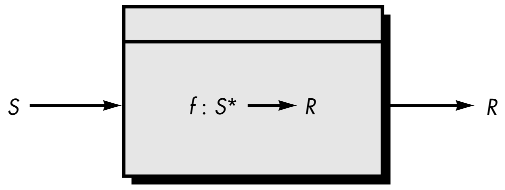
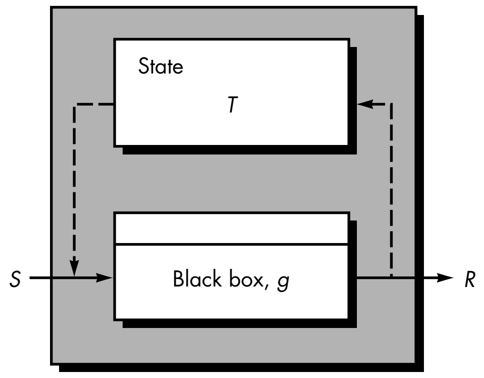


FIGURE 26.3
A black-box specification

FIGURE 26.4
A state box specification



to events (stimuli) by making a transition from the current state to some new state. As the transition is made, an action may occur. The state box uses a data abstraction to determine the transition to the next state and the action (response) that will occur as a consequence of the transition.

Referring to Figure 26.4, the state box incorporates a black box. The stimulus, *S,* that is input to the black box arrives from some external source and a set of internal system states, *T.* Mills [MIL88] provides a mathematical description of the function, *f,* of the black box contained within the state box:

$$g : S^* \times T^* \text{—>} R \times T$$

where *g* is a subfunction that is tied to a specific state, *t.* When considered collectively, the state-subfunction pairs (*t, g*) define the black box function *f.*

26.2.3 Clear-Box Specification

XRef
Procedural design and structured programming are discussed in Chapter 16.

The clear-box specification is closely aligned with procedural design and structured programming. In essence, the subfunction *g* within the state box is replaced by the structured programming constructs that implement *g.*

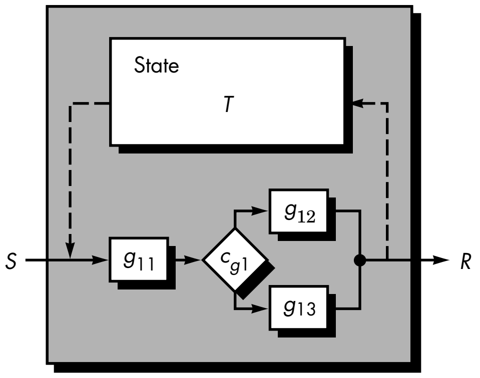
As an example, consider the clear box shown in Figure 26.5. The black box, *g,* shown in Figure 26.4, is replaced by a sequence construct that incorporates a conditional. These, in turn, can be refined into lower-level clear boxes as stepwise refinement proceeds.

It is important to note that the procedural specification described in the clear-box hierarchy can be proved to be correct. This topic is considered in the next section.

26.3 CLEANROOM DESIGN

The design approach used in cleanroom software engineering makes heavy use of the structured programming philosophy. But in this case, structured programming is applied far more rigorously.

FIGURE 26.5
A clear-box specification



WebRef
The DoD STARS program has developed a variety of cleanroom guides and documents:
ftp.cdrom.com/pub/ada/docs/cleanrm/

Basic processing functions (described during earlier refinements of the specification) are refined using a "stepwise expansion of mathematical functions into structures of logical connectives [e.g., *if-then-else*] and subfunctions, where the expansion [is] carried out until all identified subfunctions could be directly stated in the programming language used for implementation" [DYE92].

The structured programming approach can be used effectively to refine function, but what about data design? Here a number of fundamental design concepts (Chapter 13) come into play. Program data are encapsulated as a set of abstractions that are serviced by subfunctions. The concepts of data encapsulation, information hiding, and data typing are used to create the data design.

26.3.1 Design Refinement and Verification

Each clear-box specification represents the design of a procedure (subfunction) required to accomplish a state box transition. With the clear box, the structured programming constructs and stepwise refinement are used as illustrated in Figure 26.6. A program function, *f,* is refined into a sequence of subfunctions *g* and *h.* These in turn are refined into conditional constructs (*if-then-else* and *do-while*). Further refinement illustrates continuing logical refinement.

? What conditions are applied to prove structured constructs correct?

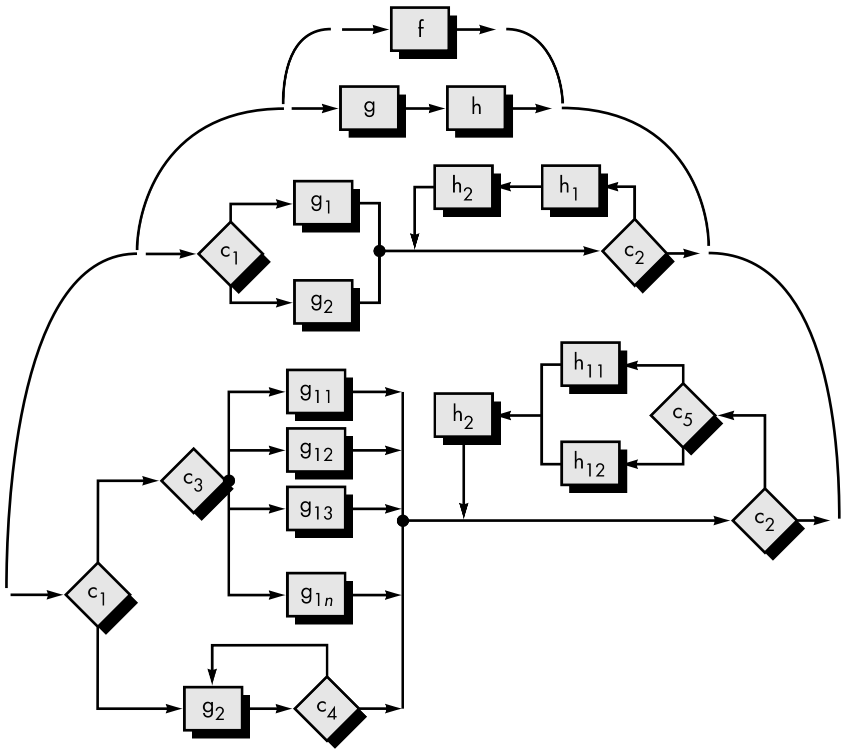
At each level of refinement, the cleanroom team[2] performs a formal correctness verification. To accomplish this, a set of generic *correctness conditions* are attached to the structured programming constructs. If a function *f* is expanded into a sequence *g* and *h*, the correctness condition for all input to *f* is

- **Does *g* followed by *h* do *f*?**

When a function *p* is refined into a conditional of the form, if ⟨ c ⟩ then *q,* else *r,* the correctness condition for all input to *p* is

2 Because the entire team is involved in the verification process, it is less likely that an error will be made in conducting the verification itself.

FIGURE 26.6
Stepwise refinement



- **Whenever condition ⟨c⟩ is true, does *q* do *p;* and whenever ⟨c⟩ is false, does *r* do *p*?**

ADVICE

If you limit yourself to just the structured constructs as you create a procedural design, proof of correctness is straightforward. If you "violate" the constructs, correctness proofs are difficult or impossible.

When function *m* is refined as a loop, the correctness conditions for all input to *m* are

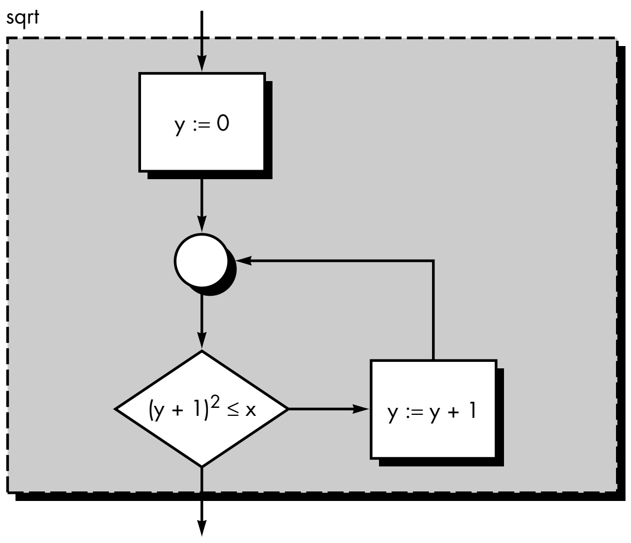
- **Is termination guaranteed?**
- **Whenever ⟨c⟩ is true, does *n* followed by *m* do *m*; and whenever ⟨c⟩ is false, does skipping the loop still do *m*?**

Each time a clear box is refined to the next level of detail, these correctness conditions are applied.

It is important to note that the use of the structured programming constructs constrains the number of correctness tests that must be conducted. A single condition is checked for sequences; two conditions are tested for if-then-else, and three conditions are verified for loops.

To illustrate correctness verification for a procedural design, we use a simple example first introduced by Linger, Mills, and Witt [LIN79]. The intent is to design and verify a small program that finds the integer part, *y,* of a square root of a given integer, *x.* The procedural design is represented using the flowchart in Figure 26.7.

FIGURE 26.7
Computing the integer part of a square root [LIN79]



To verify the correctness of this design, we must define entry and exit conditions as noted in Figure 26.8. The entry condition notes that *x* must be greater than or equal to 0. The exit condition requires that *x* remain unchanged and take on a value within the range noted in the figure. To prove the design to be correct, it is necessary to prove the conditions *init, loop, cont, yes,* and *exit* shown in Figure 26.8 are true in all cases. These are sometimes called *subproofs.*

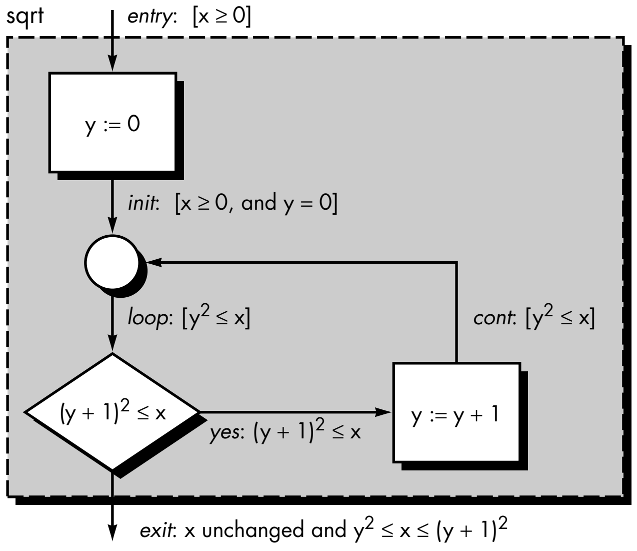


FIGURE 26.8
Proving the design correct [LIN79]

1. The condition *init* demands that [$x \geq 0$ and $y = 0$]. Based on the requirements of the problem, the entry condition is assumed correct.[3] Therefore, the first part of the *init* condition, $x \geq 0$, is satisfied. Referring to the flowchart, the statement immediately preceding the *init* condition, sets $y = 0$. Therefore, the second part of the *init* condition is also satisfied. Hence, *init* is true.

KEY POINT

To prove a design correct, you must first identify all conditions and then prove that each takes on the appropriate Boolean value. These are called *subproofs.*

2. The *loop* condition may be encountered in one of two ways: (1) directly from *init* (in this case, the *loop* condition is satisfied directly) or via control flow that passes through the condition *cont.* Since the *cont* condition is identical to the *loop* condition, *loop* is true regardless of the flow path that leads to it.
3. The *cont* condition is encountered only after the value of y is incremented by 1. In addition, the control flow path that leads to *cont* can be invoked only if the *yes* condition is also true. Hence, if $(y + 1)^2 \leq x$, it follows that $y^2 \leq x$. The *cont* condition is satisfied.
4. The *yes* condition is tested in the conditional logic shown. Hence, the *yes* condition must be true when control flow moves along the path shown.
5. The *exit* condition first demands that x remain unchanged. An examination of the design indicates that x appears nowhere to the left of an assignment operator. There are no function calls that use x. Hence, it is unchanged. Since the conditional test $(y + 1)^2 \leq x$ must fail to reach the *exit* condition, it follows that $(y + 1)^2 \leq x$. In addition, the loop condition must still be true (i.e., $y^2 \leq x$). Therefore, $(y + 1)^2 > x$ and $y^2 \leq x$ can be combined to satisfy the *exit* condition.

We must further ensure that the loop terminates. An examination of the loop condition indicates that, because y is incremented and $x \geq 0$, the loop must eventually terminate.

The five steps just noted are a proof of the correctness of the design of the algorithm noted in Figure 26.7. We are now certain that the design will, in fact, compute the integer part of a square root.

A more rigorous mathematical approach to design verification is possible. However, a discussion of this topic is beyond the scope of this book. Interested readers should refer to [LIN79].

26.3.2 Advantages of Design Verification[4]

Rigorous correctness verification of each refinement of the clear-box design has a number of distinct advantages. Linger [LIN94] describes these in the following manner:

? What do we gain by doing correctness proofs?

- **It reduces verification to a finite process.** The nested, sequential way that control structures are organized in a clear box naturally defines a hierar-

3 A negative value for a square root has no meaning in this context.
4 This section and Figures 26.7 through 26.9 have been adapted from [LIN94]. Used with permission.

FIGURE 26.9
A design with subproofs [LIN94]

```
[f1]
  DO
    g1
    g2
    [f2]
      WHILE
        p1
    DO [f3]
        g3
        [f4]
        IF
          p2
        THEN [f5]
          g4
          g5
        ELSE [f6]
          g6
          g7
        END
        g8
    END
END
```

```
Subproofs:

f1 = [DO g1; g2; [f2] END] ?

f2 = [WHILE p1 DO [f3] END] ?

f3 = [DO g3; [f4]; g8 END] ?

f4 = [IF p2; THEN [f5] ELSE [f6] END] ?

f5 = [DO g4; g5 END] ?

f6 = [DO g6; g7 END] ?
```

chy that reveals the correctness conditions that must be verified. An axiom of replacement [LIN79] lets us substitute intended functions with their control structure refinements in the hierarchy of subproofs. For example, the subproof for the intended function f1 in Figure 26.9 requires proving that the composition of the operations g1 and g2 with the intended function f2 has the same effect on data as f1. Note that f2 substitutes for all the details of its refinement in the proof. This substitution localizes the proof argument to the control structure at hand. In fact, it lets the software engineer carry out the proofs in any order.

KEY POINT

Despite the extremely large number of execution paths in a program, the number of steps to prove the program correct is quite small.

- **It is impossible to overemphasize the positive effect that reducing verification to a finite process has on quality.** Even though all but the most trivial programs exhibit an essentially infinite number of execution paths, they can be verified in a finite number of steps.
- **It lets cleanroom teams verify every line of design and code.** Teams can carry out the verification through group analysis and discussion on the basis of the correctness theorem, and they can produce written proofs when extra confidence in a life- or mission-critical system is required.
- **It results in a near zero defect level.** During a team review, every correctness condition of every control structure is verified in turn. Every team member must agree that each condition is correct, so an error is possible

only if every team member incorrectly verifies a condition. The requirement for unanimous agreement based on individual verification results in software that has few or no defects before first execution.

- **It scales up.** Every software system, no matter how large, has top-level, clear-box procedures composed of sequence, alternation, and iteration structures. Each of these typically invokes a large subsystem with thousands of lines of code—and each of those subsystems has its own top-level intended functions and procedures. So the correctness conditions for these high-level control structures are verified in the same way as are those of low-level structures. Verification at high levels may take, and well be worth, more time, but it does not take more theory.
- **It produces better code than unit testing.** Unit testing checks the effects of executing only selected test paths out of many possible paths. By basing verification on function theory, the cleanroom approach can verify every possible effect on all data, because while a program may have many execution paths, it has only one function. Verification is also more efficient than unit testing. Most verification conditions can be checked in a few minutes, but unit tests take substantial time to prepare, execute, and check.

It is important to note that design verification must ultimately be applied to the source code itself. In this context, it is often called *correctness verification.*

26.4 CLEANROOM TESTING

The strategy and tactics of cleanroom testing are fundamentally different from conventional testing approaches. Conventional methods derive a set of test cases to uncover design and coding errors. The goal of cleanroom testing is to validate software requirements by demonstrating that a statistical sample of use-cases (Chapter 11) have been executed successfully.

26.4.1 Statistical Use Testing

The user of a computer program rarely needs to understand the technical details of the design. The user-visible behavior of the program is driven by inputs and events that are often produced by the user. But in complex systems, the possible spectrum of input and events (i.e., the use-cases) can be extremely wide. What subset of use-cases will adequately verify the behavior of the program? This is the first question addressed by statistical use testing.

Statistical use testing "amounts to testing software the way users intend to use it" [LIN94]. To accomplish this, *cleanroom testing teams* (also called *certification teams*) must determine a usage probability distribution for the software. The specification (black box) for each increment of the software is analyzed to define a set of stimuli (inputs or events) that cause the software to change its behavior. Based on interviews with potential users, the creation of usage scenarios, and a general understanding of the application domain, a probability of use is assigned to each stimuli.

Test cases are generated for each stimuli[5] according to the usage probability distribution. To illustrate, consider the *SafeHome* security system discussed earlier in this book. Cleanroom software engineering is being used to develop a software increment that manages user interaction with the security system keypad. Five stimuli have been identified for this increment. Analysis indicates the percent probability distribution of each stimulus. To make selection of test cases easier, these probabilities are mapped into intervals numbered between 1 and 99 [LIN94] and illustrated in the following table:

Program Stimulus	**Probability**	**Interval**
Arm/disarm (AD)	50%	1–49
Zone set (ZS)	15%	50–63
Query (Q)	15%	64–78
Test (T)	15%	79–94
Panic alarm	5%	95–99

To generate a sequence of usage test cases that conform to the usage probability distribution, a series of random numbers between 1 and 99 is generated. The random number corresponds to an interval on the preceding probability distribution. Hence, the sequence of usage test cases is defined randomly but corresponds to the appropriate probability of stimuli occurrence. For example, assume the following random number sequences are generated:

13-94-22-24-45-56
81-19-31-69-45-9
38-21-52-84-86-4

Selecting the appropriate stimuli based on the distribution interval shown in the table, the following use-cases are derived:

AD–T–AD–AD–AD–ZS
T–AD–AD–AD–Q–AD–AD
AD–AD–ZS–T–T–AD

The testing team executes these use-cases and verifies software behavior against the specification for the system. Timing for tests is recorded so that interval times may be determined. Using interval times, the certification team can compute mean-time-to-failure. If a long sequence of tests is conducted without failure, the MTTF is low and software reliability may be assumed high.

5 Automated tools are used to accomplish this. For further information, see [DYE92].

26.4.2 Certification

The verification and testing techniques discussed earlier in this chapter lead to software components (and entire increments) that can be certified. Within the context of the cleanroom software engineering approach, *certification* implies that the reliability (measured by mean-time-to-failure, MTTF) can be specified for each component.

The potential impact of certifiable software components goes far beyond a single cleanroom project. Reusable software components can be stored along with their usage scenarios, program stimuli, and probability distributions. Each component would have a certified reliability under the usage scenario and testing regime described. This information is invaluable to others who intend to use the components.

The certification approach involves five steps [WOH94]:

? **How do we certify a software component?**

1. Usage scenarios must be created.
2. A usage profile is specified.
3. Test cases are generated from the profile.
4. Tests are executed and failure data are recorded and analyzed.
5. Reliability is computed and certified.

Steps 1 through 4 have been discussed in an earlier section. In this section, we concentrate on reliability certification.

Certification for cleanroom software engineering requires the creation of three models [POO93]:

Sampling model. Software testing executes *m* random test cases and is certified if no failures or a specified numbers of failures occur. The value of *m* is derived mathematically to ensure that required reliability is achieved.

Component model. A system composed of *n* components is to be certified. The component model enables the analyst to determine the probability that component *i* will fail prior to completion.

Certification model. The overall reliability of the system is projected and certified.

At the completion of statistical use testing, the certification team has the information required to deliver software that has a certified MTTF computed using each of these models.

A detailed discussion of the computation of the sampling, component, and certification models is beyond the scope of this book. The interested reader should see [MUS87], [CUR86], and [POO93] for additional detail.

26.5 SUMMARY

Cleanroom software engineering is a formal approach to software development that can lead to software that has remarkably high quality. It uses box structure specification (or formal methods) for analysis and design modeling and emphasizes correctness verification, rather than testing, as the primary mechanism for finding and removing errors. Statistical use testing is applied to develop the failure rate information necessary to certify the reliability of delivered software.

The cleanroom approach begins with analysis and design models that use a box structure representation. A "box" encapsulates the system (or some aspect of the system) at a specific level of abstraction. Black boxes are used to represent the externally observable behavior of a system. State boxes encapsulate state data and operations. A clear box is used to model the procedural design that is implied by the data and operations of a state box.

Correctness verification is applied once the box structure design is complete. The procedural design for a software component is partitioned into a series of subfunctions. To prove the correctness of the subfunctions, exit conditions are defined for each subfunction and a set of subproofs is applied. If each exit condition is satisfied, the design must be correct.

Once correctness verification is complete, statistical use testing commences. Unlike conventional testing, cleanroom software engineering does not emphasize unit or integration testing. Rather, the software is tested by defining a set of usage scenarios, determining the probability of use for each scenario, and then defining random tests that conform to the probabilities. The error records that result are combined with sampling, component, and certification models to enable mathematical computation of projected reliability for the software component.

The cleanroom philosophy is a rigorous approach to software engineering. It is a software process model that emphasizes mathematical verification of correctness and certification of software reliability. The bottom line is extremely low failure rates that would be difficult or impossible to achieve using less formal methods.

REFERENCES

[CUR86] Curritt, P.A., M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Trans, Software Engineering,* vol. SE-12, no. 1, January 1994.

[DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Development,* Wiley, 1992.

[HAU94] Hausler, P.A., R. Linger, and C. Trammel, "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems Journal,* vol. 33, no.1, January 1994, pp. 89–109.

[HEN95] Henderson, J., "Why Isn't Cleanroom the Universal Software Development Methodology?" *Crosstalk,* vol. 8, No. 5, May 1995, pp. 11–14.

[HEV93] Hevner, A.R. and H.D. Mills, "Box Structure Methods for System Development with Objects," *IBM Systems Journal,* vol. 31, no.2, February 1993, pp. 232–251.

{LIN79] Linger, R.M., H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley, 1979.

[LIN88] Linger, R.M. and H.D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proc. COMPSAC '88,* Chicago, October 1988.

[LIN94] Linger, R., "Cleanroom Process Model," *IEEE Software,* vol. 11, no. 2, March 1994, pp. 50–58.

[MIL87] Mills, H.D., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software,* vol. 4, no. 5, September 1987, pp. 19–24.

[MIL88] Mills, H.D., "Stepwise Refinement and Verification in Box Structured Systems," *Computer,* vol. 21, no. 6, June 1988, pp. 23–35.

[MUS87] Musa, J.D., A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures,* McGraw-Hill, 1987.

[POO88] Poore, J.H. and H.D. Mills, "Bringing Software Under Statistical Quality Control," *Quality Progress,* November 1988, pp. 52–55.

[POO93] Poore, J.H., H.D. Mills, and D. Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software,* vol. 10, no. 1, January 1993, pp. 88–99.

[WOH94] Wohlin, C. and P. Runeson, "Certification of Software Components," *IEEE Trans. Software Engineering,* vol. SE-20, no. 6, June 1994, pp. 494–499.

PROBLEMS AND POINTS TO PONDER

26.1. If you had to pick one aspect of cleanroom software engineering that makes it radically different from conventional or object-oriented software engineering approaches, what would it be?

26.2. How do an incremental process model and certification work together to produce high-quality software?

26.3. Using box structure specification, develop "first-pass" analysis and design models for the *SafeHome* system.

26.4. Develop a box structure specification for a portion of the PHTRS system introduced in Problem 12.13.

26.5. Develop a box structure specification for the e-mail system presented in Problem 21.15.

26.6. A bubble sort algorithm is defined in the following manner:

```
procedure bubblesort;
    var i, t, integer;
    begin
    repeat until t=a[1]
        t:=a[1];
        for j:= 2 to n do
            if a[j-1] > a[j] then begin
                t:=a[j-1];
            a[j-1]:=a[j];
            a[j]:=t;
            end
   endrep
end
```

Partition the design into subfunctions and define a set of conditions that would enable you to prove that this algorithm is correct.

26.7. Document a correctness verification proof for the bubble sort discussed in Problem 26.6.

26.8. Select a program component that you have designed in another context (or one assigned by your instructor) and develop a complete proof of correctness for it.

26.9. Select a program that you use regularly (e.g., an e-mail handler, a word processor, a spreadsheet program). Create a set of usage scenarios for the program. Define the probability of use for each scenario and then develop a program stimuli and probability distribution table similar to the one shown in Section 26.4.1.

26.10. For the program stimuli and probability distribution table developed in Problem 26.9, use a random number generator to develop a set of test cases for use in statistical use testing.

26.11. In your own words, describe the intent of certification in the cleanroom software engineering context.

26.12. Write a short paper that describes the mathematics used to define the certification models described briefly in Section 26.4.2. Use [MUS87], [CUR86], and [POO93] as a starting point.

FURTHER READINGS AND INFORMATION SOURCES

Prowell et al. (*Cleanroom Software Engineering: Technology and Process,* Addison-Wesley, 1999) provides an in-depth treatment of all important aspects of the cleanroom approach. Useful discussions of cleanroom topics have been edited by Poore and Trammell *(Cleanroom Software Engineering: A Reader,* Blackwell Publishing, 1996). Becker and Whittaker (*Cleanroom Software Engineering Practices,* Idea Group Publishing, 1996) present an excellent overview for those who are unfamiliar with cleanroom practices.

The Cleanroom Pamphlet (Software Technology Support Center, Hill AF Base, April 1995) contains reprints of a number of important articles. Linger [LIN94] produced one of the better introductions to the subject. *Asset Source for Software Engineering Technology,* ASSET, (United States Department of Defense) offers an excellent six volume set of *Cleanroom Engineering Handbooks.* ASSET can be contacted at info@source.asset.com. Lockheed Martin's *Guide to the Integration of Object-Oriented*

Methods and Cleanroom Software Engineering (1997) contains a generic cleanroom process for OO systems and is available at
http://www.asset.com/stars/loral/cleanroom/oo/guidhome.htm.

Linger and Trammell (*Cleanroom Software Engineering Reference Model,* SEI Technical Report CMU/SEI-96-TR-022, 1996) have defined a set of 14 cleanroom processes and 20 work products that form the basis for the SEI CMM for cleanroom software engineering (CMU/SEI-96-TR-023).

Michael Deck of Cleanroom Software Engineering has prepared a bibliography on cleanroom topics. Among the references are the following:

General and Introductory

Deck, M.D., "Cleanroom Software Engineering Myths and Realities," *Quality Week 1997,* May 1997.

Deck, M.D, and J. A. Whittaker, "Lessons Learned from Fifteen Years of Cleanroom Testing," *Software Testing, Analysis, and Review (STAR) '97,* San Jose, CA, May 5–9, 1997.

Lokan, C.J., "The Cleanroom Process for Software Development," *The Australian Computer Journal,* vol. 25, no. 4, November 1993.

Linger, Richard C., "Cleanroom Software Engineering for Zero-Defect Software," *Proc. 15th International Conference on Software Engineering,* May 1993.

Keuffel, W., "Clean Your Room: Formal Methods for the '90s," *Computer Language,* July 1992, pp. 39–46.

Hevner, A.R., S.A. Becker, and L.B. Pedowitz, "Integrated CASE for Cleanroom Development," *IEEE Software,* March 1992, pp. 69–76.

Cobb, R.H. and H.D. Mills, "Engineering Software under Statistical Quality Control," *IEEE Software,* November 1990, pp. 44–54.

Management Practices

Becker, S.A., Deck, M.D., and Janzon, T., "Cleanroom and Organizational Change," *Proc. 14th Pacific Northwest Software Quality Conference,* Portland, OR, October 29–30, 1996.

Linger, R.C., "Cleanroom Process Model," *IEEE Software.* March 1994, pp. 50–58.

Linger, R.C. and R.A. Spangler, "The IBM Cleanroom Software Engineering Technology Transfer Program," *Sixth SEI Conference on Software Engineering Education,* San Diego, CA, October 1992.

Specification, Design, and Review

Deck, M.D., "Cleanroom and Object-Oriented Software Engineering: A Unique Synergy," *1996 Software Technology Conference,* Salt Lake City, UT, April 24, 1996.

Deck, M.D., "Using Box Structures to Link Cleanroom and Object-Oriented Software Engineering," Technical Report 94.01b, Cleanroom Software Engineering, 1994.

Dyer, M., "Designing Software for Provable Correctness: The Direction for Quality Software," *Information and Software Technology,* vol. 30 no. 6, July–August 1988, pp. 331–340.

Testing and Certification

Dyer, M., "An Approach to Software Reliability Measurement," *Information and Software Technology,* vol. 29 no. 8, October 1987, pp. 415–420.

Head, G.E., "Six-Sigma Software Using Cleanroom Software Engineering Techniques," *Hewlett-Packard Journal,* June 1994, pp. 40–50.

Oshana, R., "Quality Software via a Cleanroom Methodology," *Embedded Systems Programming,* September. 1996, pp. 36–52.

Whittaker, J.A. and M.G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Trans. Software Engineering,* vol. SE-20 October 1994, pp. 812–824.

Case Studies and Experience Reports

Head, G.E., "Six-Sigma Software Using Cleanroom Software Engineering Techniques," *Hewlett-Packard Journal,* June 1994, pp. 40–50.

Hevner, A.R. and H.D. Mills, "Box-Structured Methods for Systems Development with Objects," *IBM Systems Journal,* vol. 32, no. 2, 1993, p. 232–251.

Tann, L-G., "OS32 and Cleanroom," *Proc. First Annual European Industrial Symposium on Cleanroom Software Engineering,* Copenhagen, Denmark, 1993, pp. 1–40.

Hausler, P.A., "A Recent Cleanroom Success Story: The Redwing Project," *Proc. 17th Annual Software Engineering Workshop,* NASA Goddard Space Flight Center, December 1992.

Trammel, C.J., L.H. Binder, and C.E. Snyder, "The Automated Production Control Documentation System: A Case Study in Cleanroom Software Engineering," *ACM Trans. on Software Engineering and Methodology,* vol. 1, no. 1, January 1992, pp. 81–94.

Design verification via proof of correctness lies at the heart of the cleanroom approach. Books by Baber (*Error-Free Software,* Wiley, 1991) and Schulmeyer (*Zero Defect Software,* McGraw-Hill, 1990) discuss proof of correctness in considerable detail.

A wide variety of information sources on cleanroom software engineering and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to cleanroom software engineering can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/cleanroom.mhtml

CHAPTER

27 COMPONENT-BASED SOFTWARE ENGINEERING

In the software engineering context, reuse is an idea both old and new. Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc. Today, complex, high-quality computer-based systems must be built in very short time periods. This mitigates toward a more organized approach to reuse.

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software "components." Clements [CLE95] describes CBSE in the following way:

[CBSE] is changing the way large software systems are developed. [CBSE] embodies the "buy, don't build" philosophy espoused by Fred Brooks and others. In the same way that early subroutines liberated the programmer from thinking about details, [CBSE] shifts the emphasis from programming software to composing software systems. Implementation has given way to integration as the focus. At its foundation is the assumption that there is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality.

But a number of questions arise. Is it possible to construct complex systems by assembling them from a catalog of reusable software components? Can this

QUICK LOOK

What is it? You purchase a "stereo system" and bring it home. Each component has been designed to fit a specific architectural style—connections are standardized, communication protocol has be preestablished. Assembly is easy because you don't have to build the system from hundreds of discrete parts. Component-based software engineering strives to achieve the same thing. A set of prebuilt, standardized software components are made available to fit a specific architectural style for some application domain. The application is then assembled using these components, rather than the "discrete parts" of a conventional programming language.

Who does it? Software engineers apply the CBSE process.

Why is it important? It takes only a few minutes to assemble the stereo system because the components are designed to be integrated with ease. Although software is considerably more complex, it follows that component-based systems are easier to assemble and therefore less costly to build than systems constructed from discrete parts. In addition, CBSE encourages the use of predictable architectural patterns and standard software infrastructure, thereby leading to a higher-quality result.

What are the steps? CBSE encompasses two parallel engineering activities: domain engineering and

QUICK LOOK

component-based development. Domain engineering explores an application domain with the specific intent of finding functional, behavioral, and data components that are candidates for reuse. These components are placed in reuse libraries. Component-based development elicits requirements from the customer, selects an appropriate architectural style to meet the objectives of the system to be built, and then (1) selects potential components for reuse, (2) qualifies the components to be sure that they properly fit the architecture for the system, (3) adapts components if modifications must be made to properly integrate them, and (4) integrates the components to form subsystems and the application as a whole. In addition, custom components are engineered to address those aspects of the system that cannot be implemented using existing components.

What is the work product? Operational software, assembled using existing and newly developed software components, is the result of CBSE.

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 issues associated with acquired components, testing is applied to uncover errors in newly developed software and in reusable components that have been integrated into the architecture.

be accomplished in a cost- and time-effective manner? Can appropriate incentives be established to encourage software engineers to reuse rather than reinvent? Is management willing to incur the added expense associated with creating reusable software components? Can the library of components necessary to accomplish reuse be created in a way that makes it accessible to those who need it? Can components that do exist be found by those who need them?

These and many other questions continue to haunt the community of researchers and industry professionals who are striving to make software component reuse a mainstream approach to software engineering. We look at some of the answers in this chapter.

27.1 ENGINEERING OF COMPONENT-BASED SYSTEMS

On the surface, CBSE seems quite similar to conventional or object-oriented software engineering. The process begins when a software team establishes requirements for the system to be built using conventional requirements elicitation techniques (Chapters 10 and 11). An architectural design (Chapter 14) is established, but rather than moving immediately into more detailed design tasks, the team examines requirements to determine what subset is directly amenable to *composition,* rather than construction. That is, the team asks the following questions for each system requirement:

- Are commercial off-the-shelf (COTS) components available to implement the requirement?
- Are internally developed reusable components available to implement the requirement?

- Are the interfaces for available components compatible within the architecture of the system to be built?

The team attempts to modify or remove those system requirements that cannot be implemented with COTS or in-house components.[1] If the requirement(s) cannot be changed or deleted, conventional or object-oriented software engineering methods are applied to develop those new components that must be engineered to meet the requirement(s). But for those requirements that are addressed with available components, a different set of software engineering activities commences:

? What are the CBSE framework activities?

Component qualification. System requirements and architecture define the components that will be required. Reusable components (whether COTS or in-house) are normally identified by the characteristics of their interfaces. That is, "the services that are provided, and the means by which consumers access these services" [BRO96] are described as part of the component interface. But the interface does not provide a complete picture of the degree to which the component will fit the architecture and requirements. The software engineer must use a process of discovery and analysis to qualify each component's fit.

Component adaptation. In Chapter 14, we noted that software architecture represents design patterns that are composed of components (units of functionality), connections, and coordination. In essence the architecture defines the design rules for all components, identifying modes of connection and coordination. In some cases, existing reusable components may be mismatched to the architecture's design rules. These components must be adapted to meet the needs of the architecture or discarded and replaced by other, more suitable components.

Quote:

"[I]t seems clear that in the near future most software applications will be assembled from components rather than being constructed from scratch."

Paul Allen and Stuart Frost

Component composition. Architectural style again plays a key role in the way in which software components are integrated to form a working system. By identifying connection and coordination mechanisms (e.g., run-time properties of the design), the architecture dictates the composition of the end product.

Component update. When systems are implemented with COTS components, update is complicated by the imposition of a third party (i.e., the organization that developed the reusable component may be outside the immediate control of the software engineering organization).

Each of these CBSE activities is discussed in more detail in Section 27.4.

1 The implication is that the organization adjust its business or product requirements so that component-based implementation can be achieved without the need for custom engineering. This approach reduces system cost and improves time to market but is not always possible.

In the first part of this section, the term *component* has been used repeatedly, yet a definitive description of the term is elusive. Brown and Wallnau [BRO96] suggest the following possibilities:

- *Component*—a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture.
- *Run-time software component*—a dynamic bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered in run time.
- *Software component*—a unit of composition with contractually specified and explicit context dependencies only.
- *Business component*—the software implementation of an "autonomous" business concept or business process.

In addition to these descriptions, software components can also be characterized based on their use in the CBSE process. In addition to COTS components, the CBSE process yields:

XRef
Certification of software components can be accomplished using cleanroom methods. See Chapter 26 for details.

- *Qualified components*—assessed by software engineers to ensure that not only functionality, but performance, reliability, usability, and other quality factors (Chapter 19) conform to the requirements of the system or product to be built.
- *Adapted components*—adapted to modify (also called *mask* or *wrap*) [BRO96] unwanted or undesirable characteristics.
- *Assembled components*—integrated into an architectural style and interconnected with an appropriate infrastructure that allows the components to be coordinated and managed effectively.
- *Updated components*—replacing existing software as new versions of components become available.

Because CBSE is an evolving discipline, it is unlikely that a unifying definition will emerge in the near term.

27.2 THE CBSE PROCESS

In Chapter 2, a "component-based development model" (Figure 2.11) was used to illustrate how a library of reusable "candidate components" can be integrated into a typical evolutionary process model. The CBSE process, however, must be characterized in a manner that not only identifies candidate components but also qualifies each component's interface, adapts components to remove architectural mismatches, assembles components into a selected architectural style, and updates components as requirements for the system change [BRO96].

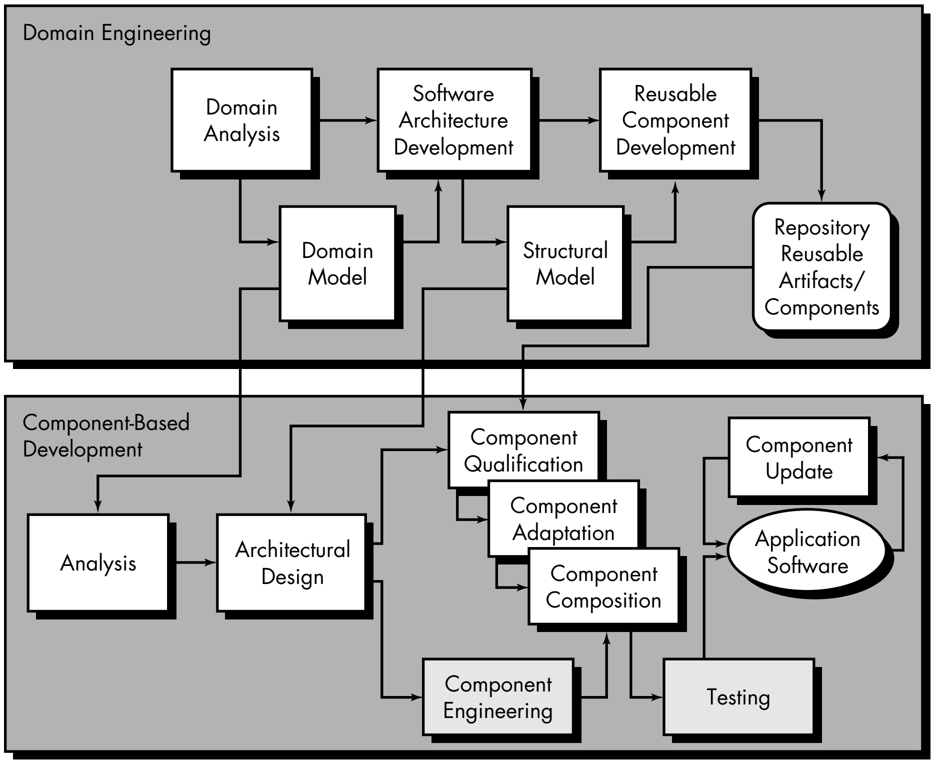


FIGURE 27.1
A process model that supports CBSE

The process model for component-based software engineering emphasizes parallel tracks in which domain engineering (Section 27.3) occurs concurrently with component-based development. *Domain engineering* performs the work required to establish a set of software components that can be reused by the software engineer. These components are then transported across a "boundary" that separates domain engineering from component-based development.

WebRef
The component technology homepage provides useful information for CBSE: **www.odateam.com /cop/**

Figure 27.1 illustrates a typical process model that explicitly accommodates CBSE [CHR95]. Domain engineering creates a model of the application domain that is used as a basis for analyzing user requirements in the software engineering flow. A generic software architecture (and corresponding structure points, see Section 27.3.3) provide input for the design of the application. Finally, after reusable components have been purchased, selected from existing libraries, or constructed (as part of domain engineering), they are made available to software engineers during component-based development.

27.3 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.

Quote:
"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

Domain engineering includes three major activities—analysis, construction, and dissemination. An overview of domain analysis was presented in Chapter 21. However, the topic is revisited in the sections that follow. Domain construction and dissemination are considered in later sections in this chapter.

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.

27.3.1 The Domain Analysis Process

In Chapter 21, we discussed the overall approach to domain analysis within the context of object-oriented software engineering. The steps in the process were 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.
5. Develop an analysis model for the objects.

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.

Prieto-Diaz [PRI87] expands the second domain analysis step and suggests an eight-step approach to the identification and categorization of reusable components:

? How can we identify and categorize reusable components?

1. Select specific functions or objects.
2. Abstract functions or objects.
3. Define a taxonomy.
4. Identify common features.
5. Identify specific relationships.
6. Abstract the relationships.
7. Derive a functional model.
8. Define a domain language.

A *domain language* enables the specification and later construction of applications within the domain.

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:

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

- 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?
- Does the hardware remain unchanged between implementations?
- 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?

An in-depth discussion of domain analysis methods is beyond the scope of this book. For additional information on domain analysis, see [PRI93].

27.3.2 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

TABLE 27.1 Domain Characteristics Affecting Reuse [BAS94]

Product	**Process**	**Personnel**
Requirements stability	Process model	Motivation
Concurrent software	Process conformance	Education
Memory constraints	Project environment	Experience/training
Application size	Schedule constraints	• application domain
User interface complexity	Budget constraints	• process
Programming language(s)	Productivity	• platform
Safety/reliability		• language
Lifetime requirements		Development team productivity
Product quality		
Product reliability		

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
A worthwhile report addressing object technology, architectures, and domain analysis can be found at **www.sei.cmu.edu/mbse/wisr_report.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*.

Table 27.1 [BAS94] lists typical domain characteristics that can have an impact of software reuse. These domain characteristics must be taken into account in order to reuse a component effectively.

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 (ideally, a limited number), "reinventing the wheel" may still be the most cost-effective choice.

27.3.3 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.

Pollak and Rissman [POL94] describe structural models in the following way:

Structural models consist of a small number of structural elements manifesting clear patterns of interaction. The architectures of systems using structural models are characterized by multiple ensembles that are composed from these model elements. Many architectural units emerge from simple patterns of interaction among this small number of elements.

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 14) 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. Restating this in object-oriented jargon (Chapter 20), the size of the class hierarchy should be small. 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.

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:

KEY POINT

A structure point can be viewed as a design pattern that can be found repeatedly in applications within a specific domain.

- 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.
- 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 27.6.2.

27.4 COMPONENT-BASED DEVELOPMENT

Component-based development 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 27.1). 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.

27.4.1 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 are applied when a component is proposed for use.

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.

2 It should be noted that the architectural style is often influenced by the generic structural model created during domain engineering (see Figure 27.1).

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]:

? What factors are considered during component qualification?

- 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 nonnumerical algorithms.
- Exception handling.

Each of these factors is relatively easy to assess when reusable components that have been developed in-house are proposed. If good software engineering practices were applied during their development, answers to the questions implied by the list can be developed. 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

Quote:

"Component integrators need to discover the function and form of software components."

Alan Brown and Kurt Wallnau

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 reality, even after a component has been qualified for use within an application architecture, it may exhibit conflict in one or more of the areas just noted. To mitigate against 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 17), *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 postprocessing 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[3] have proposed standards for component software:

3 An excellent discussion of the "distributed objects" standards is presented in [ORF96] and [YOU98].

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 on services that enable reusable components (objects) to communicate with other components, regardless of their location within a system. When components are built using the OMG/CORBA standard, integration of those components (without modification) within a system is assured if an *interface definition language* (IDL) interface is created for every component. Using a client/server metaphor, objects within the client application request one or more services from the ORB server. Requests are made via an IDL or dynamically at run time. An interface repository contains all necessary information about the service's request and response formats. CORBA is discussed further in Chapter 28.

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. From the point of view of the application, "the focus is not on how [COM objects are] implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects" [HAR98].

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

Sun JavaBean Components. The JavaBean component system is a portable, platform independent CBSE infrastructure developed using the Java programming language. The JavaBean system extends the Java applet[4] to accommodate the more sophisticated software components required for component-based development. The JavaBean 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 standardized on 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.

4 In this context, an applet may be viewed as a simple component.

27.4.2 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 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.[5] In this section we will not revisit these topics. Rather, we consider the reuse-specific issues that are complementary to solid software engineering practices.

27.4.3 Analysis and Design for Reuse

The components of the analysis model were discussed in detail in Parts Three and Four of this book. Data, functional, and behavioral models (represented in a variety of different notations) can be created to describe what a particular application must accomplish. Written specifications are then used to describe these models. A complete description of requirements is the result.

ADVICE

Even if your organization doesn't do domain engineering, do it informally as you work. As you build the analysis model ask yourself, "Is it likely that this object or function has been encountered in other applications of this type?" If the answer is, "Yes," a component may already exist.

Ideally, 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." Bellinzoni, Gugini, and Pernici [BEL95] describe one approach for object-oriented systems:

> Components are defined and stored as specification, design, and implementation classes at various levels of abstraction—with each class being an engineered description of a product from previous applications. The specification knowledge—development knowledge—is stored in the form of reuse-suggestion classes, which contain directions for retrieving reusable components on the basis of their description and for composing and tailoring them after retrieval.

Automated tools are used to browse a repository in an attempt to match the requirement noted in the current specification with those described for existing reusable components (classes). Characterization functions (Section 27.3.2) and keywords are used to help find potentially reusable components.

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.

5 To learn more about these topics, see Chapters 13 through 16 and 20 through 22.

It is at this point—when the designer begins to create a new component—that *design for reuse* (DFR) should be considered.

ADVICE

Although special issues must be considered when reuse is an objective, focus on the basic principles of good design. If you follow them, your chances of reuse increase significantly.

As we have already noted, DFR requires the software engineer to apply solid software design concepts and principles (Chapter 13). But the characteristics of the application domain must also be considered. Binder [BIN93] suggests a number of key issues[6] 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 27.3.3) 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.

Like design, the construction of reusable components draws on software engineering methods that have been discussed elsewhere in this book. Construction can be accomplished using conventional third generation programming languages, fourth generation languages and code generators, visual programming techniques, or more advanced tools.

27.5 CLASSIFYING AND RETRIEVING COMPONENTS

Quote:

"The next best thing to knowing something, is knowing where to find it."

Samuel Johnson

Consider a large 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.

6 In general, the design for reuse preparations should be undertaken as part of domain engineering (Section 27.3).

27.5.1 Describing Reusable Components

KEY POINT In order to describe a reusable component, the component's concept, content, and context should be described.

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.

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.

WebRef A detailed guide for component reuse can be downloaded from **web1.ssg.gunter.af.mil/sep/SEPver40/Main.html#GD**

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., [WHI95] contains an extensive bibliography). 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 27.2 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

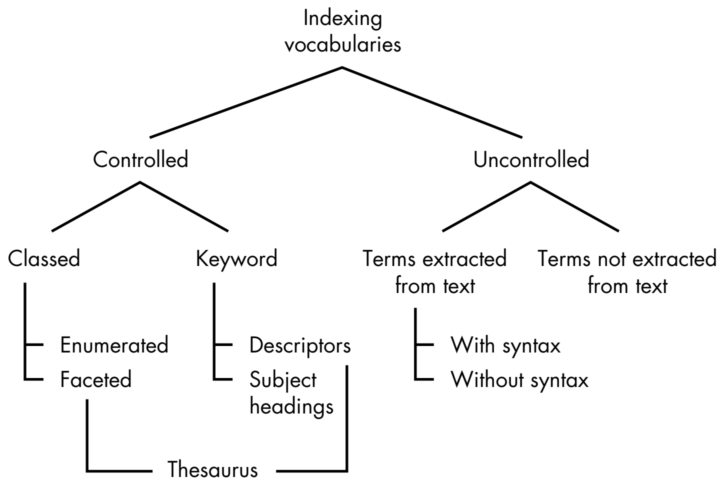


FIGURE 27.2 A taxonomy of indexing methods [FRA94]

of the description. The majority of classification schemes for software components fall into three categories:

? **What options are available for classifying components?**

Enumerated classification. Components are described by a hierarchical structure in which classes and varying levels of subclasses of software components are defined. Actual components are listed at the lowest level of any path in the enumerated hierarchy. For example, an enumerated hierarchy for window operations[7] might be

```
window operations
    display
        open
            menu-based
                openWindow
            system-based
                sysWindow
        close
            via pointer
                ...
    resize
        via command
            setWindowSize, stdResize, shrinkWindow
        via drag
            pullWindow, stretchWindow
    up/down shuffle
            ...
    move
            ...
    close
            ...
```

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.

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.

7 Only a small subset of all possible operations is noted.

As a simple illustration of the use of facets in component classification, consider a scheme [LIA93] that makes use of the following facet descriptor:

{function, object type, system type}

Each facet in the facet descriptor takes on one or more values that are generally descriptive keywords. For example, if *function* is a facet of a component, typical values assigned to this facet might be

function = (*copy, from*) or (*copy, replace, all*)

The use of multiple facet values enables the primitive function *copy* to be refined more fully. *Keywords* (values) are assigned to the set of facets for each component in a reuse library. When a software engineer wants to query the library for possible components for a design, a list of values is specified and the library is searched for matches. Automated tools can be used to incorporate a thesaurus function. This enables the search to encompass not only the keyword specified by the software engineer but also technical synonyms for those keywords. A faceted classification scheme gives the domain engineer greater flexibility in specifying complex descriptors for components [FRA94]. Because new facet values can be added easily, the faceted classification scheme is easier to extend and adapt than the enumeration approach.

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) the 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.

27.5.2 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 CASE repository (Chapter 31) and provides facilities for the storage of software components and a wide variety of reusable artifacts (e.g., specifications, designs, 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 27.5.1) 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 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 beyond the scope of this book. The interested reader should see [HOO91] and [LIN95] for additional information.

27.6 ECONOMICS OF CBSE

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.

27.6.1 Impact on Quality, Productivity, and Cost

ADVICE

CBSE is an economic "no brainer" if the components available are right for the job. If reuse demands customization, proceed with caution.

Considerable evidence from industry case studies (e.g., [HEN95], [MCM95], [LIM94]) 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 26) and would contain no defects. In 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.

KEY POINT

Although empirical data vary, industry evidence indicates that reuse does provide substantial cost benefit.

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,[8] it appears that 30 to 50 percent reuse can result in productivity improvements in the 25–40 percent range.

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 5. The costs associated with reuse, C_r, include [CHR95]

? What costs are associated with software reuse?

- 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.
- Training of personnel in design and construction for reuse.

Although costs associated with domain analysis (Section 27.4) 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.

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

27.6.2 Cost Analysis Using Structure Points

In Section 27.3.3, 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.

Even though structure points are reusable, their qualification, adaptation, integration, and maintenance costs are nontrivial. Before proceeding with reuse, the project manager must 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.

? Is there a quick calculation that allows us to estimate the cost benefit of component reuse?

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

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

where

E_{new} = effort required to engineer and construct new software components (determined using techniques described in Chapter 5).
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.

27.6.3 Reuse Metrics

A variety of software metrics have been developed in an attempt to measure the benefits of reuse within a computer-based system. The benefit associated with reuse within a system *S* can be expressed as a ratio

$$R_b(S) = [C_{\text{noreuse}} - C_{\text{reuse}}]/C_{\text{noreuse}} \tag{27-1}$$

where

C_{noreuse} is the cost of developing S with no reuse.
C_{reuse} is the cost of developing S with reuse.

It follows that $R_b(S)$ can be expressed as a nondimensional value in the range

$$0 \le R_b(S) \le 1 \tag{27-2}$$

Devanbu and his colleagues [DEV95] suggest that (1) R_b will be affected by the design of the system; (2) since R_b is affected by the design, it is important to make R_b a part of an assessment of design alternatives; and (3) the benefits associated with reuse are closely aligned to the cost benefit of each individual reusable component.

A general measure of reuse in object-oriented systems, termed *reuse leverage* [BAS94], is defined as

$$R_{\text{lev}} = \text{OBJ}_{\text{reused}}/\text{OBJ}_{\text{built}} \tag{27-3}$$

where

$\text{OBJ}_{\text{reused}}$ is the number of objects reused in a system.
$\text{OBJ}_{\text{built}}$ is the number of objects built for a system.

27.7 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, 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.

[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.

[BEL95] Bellinzona R., M.G. Gugini, and B. Pernici, "Reusing Specifications in OO Applications," *IEEE Software,* March 1995, pp. 65–75.

[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.

[DEV95] Devanbu, P., et al., "Analytical and Empirical Evaluation of Software Reuse Metrics," Technical Report, Computer Science Department, University of Maryland, August 1995.

[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.

[HAR98] Harmon, P., "Navigating the Distributed Components Landscape," *Cutter IT Journal,* vol. 11, no. 2, December 1998, pp. 4–11.

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

[HOO91] Hooper, J.W. and R.O. Chester, *Software Reuse: Guidelines and Methods,* Plenum Press, 1991.

[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.

[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.

[PRI87] Prieto-Diaz, R., "Domain Analysis for Reusability," *Proc. COMPSAC '87,* Tokyo, October 1987, pp. 23–29.

[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

27.1. 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?

27.2. Although software components are the most obvious reusable "artifact," many other entities 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?

27.3. Do a bit of research on domain engineering and flesh out the process model outlined in Figure 27.1. Identify the tasks that are required for domain analysis and software architecture development.

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

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

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

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

27.8. What is a structure point?

27.9. 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.

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

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

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

27.13. An object-oriented system is estimated to require 320 objects, when complete. It is further estimated that 190 objects can be acquired from an existing repository. What is the reuse leverage? Assume that new objects cost $1000 each and that the cost to adapt an object is $600 and to integrate each object is $400. What is the estimated cost of the system. What is the value for R_b?

FURTHER READINGS AND INFORMATION SOURCES

Many books on component-based development and component reuse have been published in recent years. Allen, Frost, and Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective,* Cambridge University Press, 1998) cover the entire CBSE process, using UML (Chapters 21 and 22) as the basis for their modeling approach. Books by Lim (*Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components,* Prentice-Hall, 1998); Coulange (*Software Reuse,* Springer-Verlag, 1998); Reifer (*Practical Software Reuse,* Wiley, 1997); and Jacobson, Griss, and Jonsson (*Software Reuse: Architecture Process and Organization for Business Success,* Addison-Wesley, 1997) address many CBSE topics. Fowler (*Analysis Patterns: Reusable Object Models,* Addison-Wesley, 1996) considers the application of architectural patterns within the CBSE process and provides many useful examples. Tracz (*Confessions of a Used Program Salesman: Institutionalizing Software Reuse,* Addison-Wesley, 1995) presents a sometimes lighthearted, but meaningful, discussion of the issues associated with creating a reuse culture.

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

Doss, G.M., *CORBA Networking With Java,* Wordware Publishing, 1999.
Hoque, R., *CORBA for Real Programmers,* Academic Press/Morgan Kaufmann, 1999.
Siegel, J., *CORBA 3 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.
Kirtland, M., *Designing Component-Based Applications,* Microsoft Press, 1999.

Many organizations apply a combination of component standards. Books by Geraghty et al. (*COM-CORBA Interoperability,* Prentice-Hall, 1999), Pritchard (*COM and CORBA Side by Side: Architectures, Strategies, and Implementations,* Addison-Wesley, 1999), and Rosen et al. (*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.
Valesky, T.C., *Enterprise Javabeans: Developing Component-Based Distributed Applications,* Addison-Wesley, 1999.
Vogel, A. and M. Rangarao, *Programming with Enterprise JavaBeans, JTS, and OTS,* Wiley, 1999.

A wide variety of information sources on component-based software engineering and component reuse is available on the Internet. An up-to-date list of World Wide Web references that are relevant to CBSE can be found at the SEPA Web site: **http://www.mhhe.com/engcs/compsci/pressman/resources/cbse.mhtml**

CHAPTER

28 CLIENT/SERVER SOFTWARE ENGINEERING

KEY CONCEPTS

analysis modeling 755
architectural design 756
architecture 748
components 750
configuration options 751
CORBA 753
database design. 758
data distribution 759
function distribution.....752
middleware 750
ORB...........753
testing.........761

At the turn of the twentieth century, the development of a new generation of machine tools capable of holding very tight tolerances empowered the engineers who designed a new factory process called *mass production.* Before the advent of this advanced machine tool technology, machines could not hold tight tolerances. But with it, easily assembled interchangeable parts—the cornerstone of mass production—could be built.

When a new computer-based system is to be developed, a software engineer is constrained by the limitations of existing computing technology and empowered when new technologies provide capabilities that were unavailable to earlier generations of engineers. The evolution of distributed computer architectures has enabled system and software engineers to develop new approaches to how work is structured and how information is processed within an organization.

New organization structures and new information processing approaches (e.g., intra- and Internet technologies, decision support systems, groupware, and imaging) represent a radical departure from the earlier mainframe- and minicomputer-based technologies. New computing architectures have provided the technology that has enabled organizations to reengineer their business processes (Chapter 30).

QUICK LOOK

What is it? Client/server (c/s) architectures dominate the landscape of computer-based systems. Everything from automatic teller networks to the Internet exist because software residing on one computer—the client—requests services and/or data from another computer—the server. Client/server software engineering blends conventional principles, concepts, and methods discussed earlier in this book with elements of object-oriented and component-based software engineering to create c/s systems.

Who does it? Software engineers perform the analysis, design, implementation, and testing of c/s systems.

Why is it important? The impact of c/s systems on business, commerce, government, and science is pervasive. As technological advances (e.g., component-based development, object request brokers, Java) change the way in which c/s systems are built, a solid software engineering process must be applied to their construction.

What are the steps? The steps involved in the engineering of c/s systems are similar to those applied during OO and component-based software engineering. The process model is evolutionary, beginning with requirements elicitation. Functionality is allocated to subsystems of components, which are then assigned to either the client or the server side of the c/s architecture.