28.9 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, a representation of a system's externally observable mode of behavior, or (in Z and related languages) 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 formal specification languages, such as OCL and Z. These formal specification languages have 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.

A decision to use formal methods should 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.

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

[ISO02] *Z Formal Specification Notation—Syntax, Type System and Semantics,* ISO/IEC 13568:2002, Intl. Standards Organization, 2002.

[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.

[OMG03] "Object Constraint Language Specification," in *Unified Modeling Language,* v2.0, Object Management Group, September 2003, download from www.omg.org.

[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

28.1. Using the OCL or Z notation presented in Tables 28.1 or 28.2, select some part of the *SafeHome* security system described earlier in this book and attempt to specify it with OCL or Z.

28.2. Develop a mathematical description for the state and data invariant for Problem 28.4. Refine this description in the OCL or Z specification language.

28.3. 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.

28.4. Develop a mathematical description for the state and data invariant for Problem 28.3. Refine this description in the OCL or Z specification language.

28.5. You have been assigned to a software team that is developing software, called *Memory Doubler,* 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.

28.6. 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.

28.7. Attempt to develop an 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.

28.8. 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 inclusively.

28.9. The installer for a PC-based application first determines whether an acceptable set of hardware and system 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.

28.10. Review the types of deficiencies associated with less formal approaches to software engineering in Section 28.1.1. Provide three examples of each from your own experience.

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

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

Clark. T., et al. (eds.), *Object Modeling with OCL,* Springer-Verlag, 2002.

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.

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.

Monin, F., and M. Hinchley, *Understanding Formal Methods,* Springer-Verlag, 2003.

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.

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

Warmer, J., and A. Kleppe, *Object Constraint Language,* Addison-Wesley, 1998.

Dean (*Essence of Discrete Mathematics,* Prentice-Hall, 1996), Gries and Schneider [GRI93], and Lipschultz and Lipson (*2000 Solved Problems in Discrete Mathematics,* McGraw-Hill, 1991) present useful information for those who must learn more about the mathematical underpinnings of formal methods.

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

CHAPTER

29 CLEANROOM SOFTWARE ENGINEERING

KEY CONCEPTS

black-box spec
box structure
cleanroom strategy
certification
clear-box spec
design refinement
functional specification
proof of correctness
state-box spec
statistical use testing
subproofs
verification

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 28, 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.

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.

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 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.

29.1 THE CLEANROOM APPROACH

The philosophy of the "cleanroom" in hardware fabrication technologies is really quite simple: It is cost- 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 errors in specification and design and then fabricate in a "clean" manner.

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 much of the industry continues operating at relatively low levels of process maturity, software engineers have not been ready to apply cleanroom 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.

> **"The only way for errors to occur in a program is by being put there by the author. No other mechanisms are known . . . Right practice aims at preventing insertion of errors and, failing that, removing them before testing or any other running of the program."**
>
> **Harlan Mills**

29.1.1 The Cleanroom Strategy

The cleanroom approach makes use of a specialized version of the incremental process model (Chapter 3). A "pipeline of software increments" [LIN94] is developed by small independent software teams. As each increment is certified, it is integrated into the whole. Hence, functionality of the system grows with time.

The sequence of cleanroom tasks for each increment is illustrated in Figure 29.1. Overall system or product requirements are developed using the system engineering methods discussed in Chapter 6. Once functionality has been assigned to the software element of the system, the pipeline of cleanroom increments is initiated. The following tasks occur:

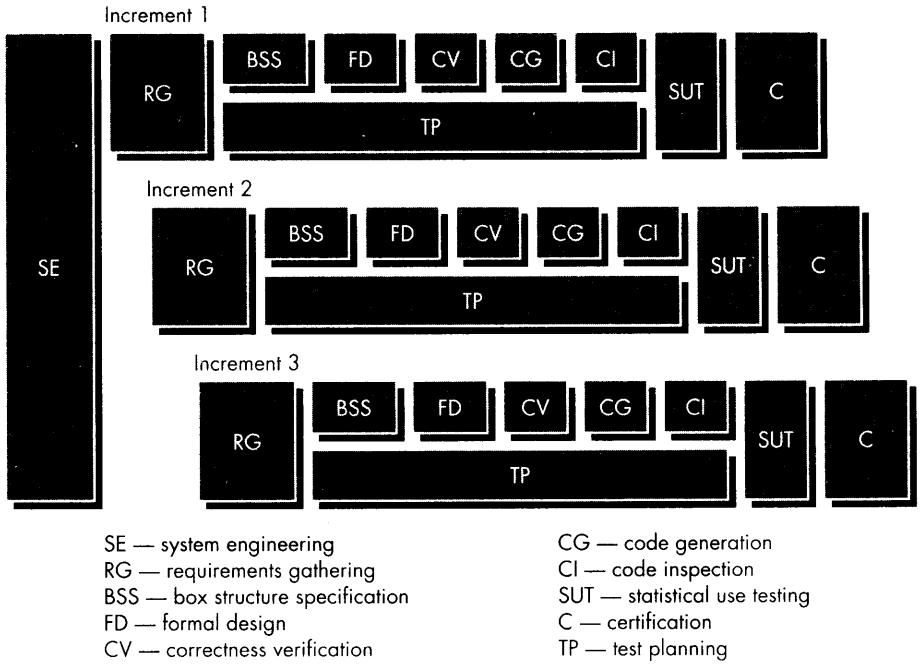
? What are the major tasks conducted as part of cleanroom software engineering?

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 7, 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 to

FIGURE 29.1
The cleanroom process model



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

the operational analysis principles discussed in Chapters 5 and 7, 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.deansoft.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 29.3 and 29.4) 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.

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 26) 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.

> **"Cleanroom software 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**

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

Statistical test planning. The projected usage of the software is analyzed and a suite of test cases that exercise a "probability distribution" of usage is planned and designed (Section 29.4). Referring to Figure 29.1, this cleanroom activity is conducted in parallel with specification, verification, and code generation.

Statistical use testing. Recalling that exhaustive testing of computer software is impossible (Chapter 14), 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 29.4).

Certification. Once verification, inspection, and use 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.

29.1.2 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.

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 software engineering methods because:

1. It makes explicit use of statistical quality control.
2. It verifies design specifications using a mathematically based proof of correctness.
3. It implements testing techniques that have a high likelihood of uncovering 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]

In conventional software development, errors are accepted as a fact of life. Because errors are deemed to be inevitable, each program component 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.

> **"It's a funny thing about life: If you refuse to accept anything but the best, you may very often get it."**
>
> **W. Somerset Maugham**

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.

---

1 Testing is conducted by an independent testing team.

29.2 FUNCTIONAL SPECIFICATON

Regardless of the analysis method that is chosen, the operational analysis principles presented in Chapter 7 apply. Data, function, and behavior are modeled. The resultant models must be partitioned (refined) to provide increasingly greater detail. The overall objective is to move from a specification (or model) 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 elaboration or 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 a 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 29.2 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 correctness verification 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.

FIGURE 29.2
Box structure refinement

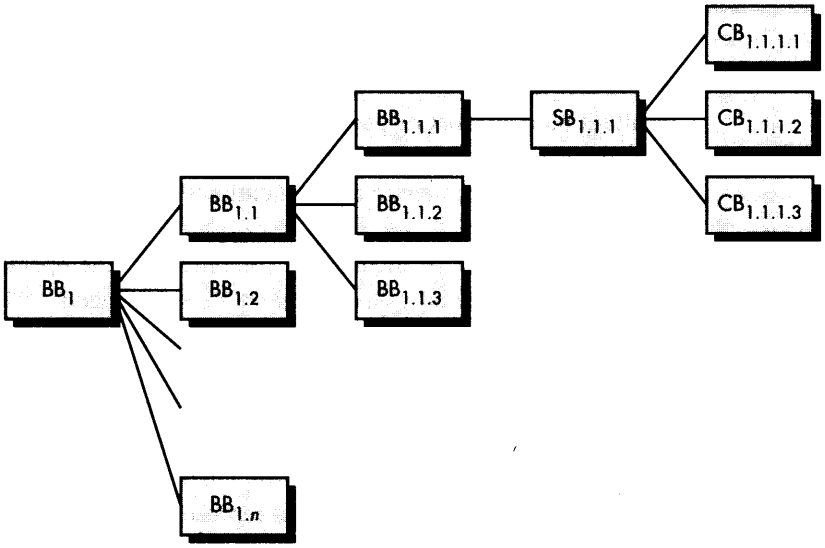
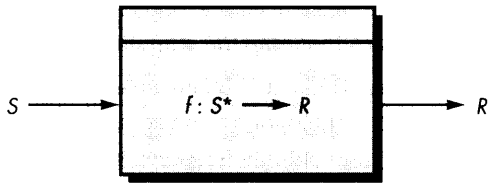


FIGURE 29.3
A black-box specification



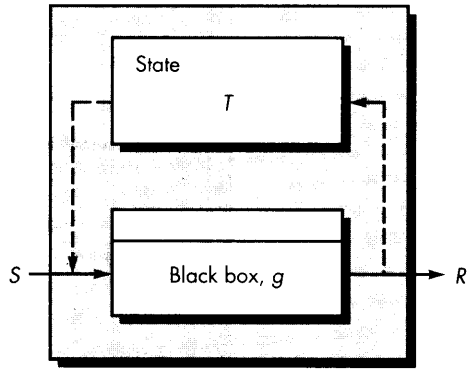
It should be noted that specification methods based on languages such as OCL or Z (Chapter 28) can be used in conjunction with the box structure specification approach. The only requirement is that each level of specification can be formally verified.

29.2.1 Black-Box Specification

A *black-box* specification describes an abstraction, stimuli, and response using the notation shown in Figure 29.3 [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).

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.

FIGURE 29.4
A state-box specification



29.2.2 State-Box Specification

The *state box* is "a simple generalization of a state machine" [MIL88]. Recalling the discussion of behavioral modeling and state diagrams in Chapter 8, a state is some observable mode of system behavior. As processing occurs, a system responds 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 29.4, 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:

$$g : S^* \times T^* \rightarrow R \times T$$

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*.

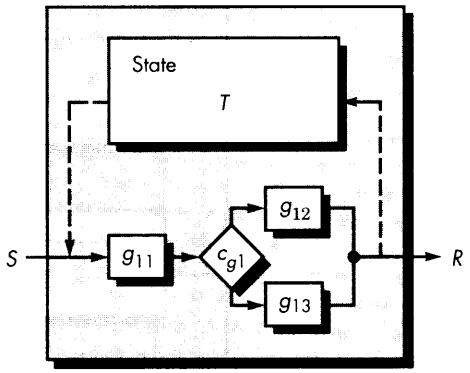
29.2.3 Clear-Box Specification

The clear-box specification is closely aligned with procedural design and structured programming (Chapter 11). 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 29.5. The black box, *g*, shown in Figure 29.4, is replaced by a sequence construct that incorporates a conditional. These constructs, 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 correct. This topic is considered in the next section.

FIGURE 29.5
A clear-box specification



29.3 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.

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 (Chapters 5 and 9) 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.

29.3.1 Design Refinement and Verification

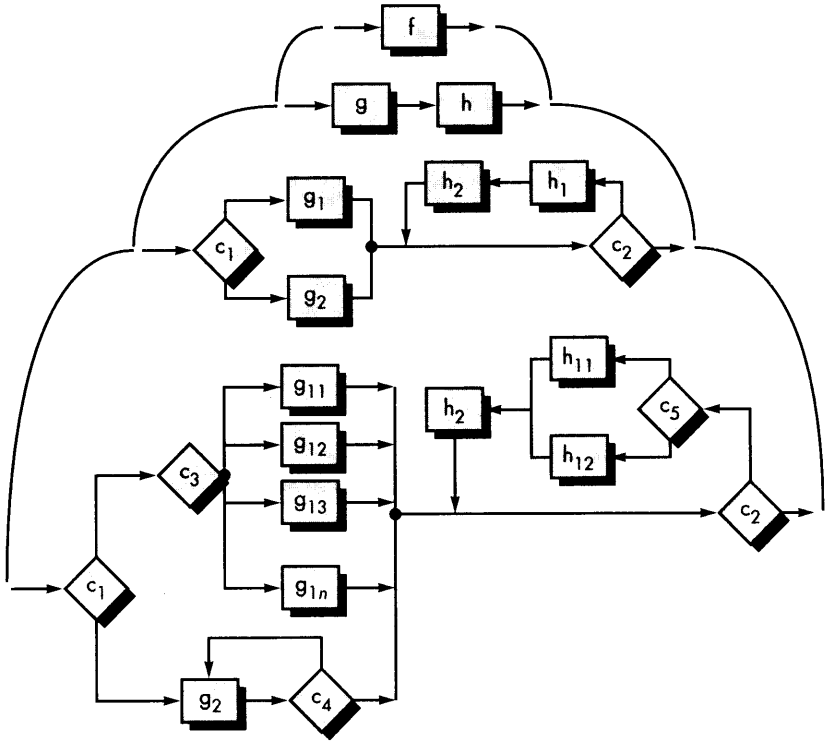
Each clear-box specification represents the design of a procedure (subfunction) required to accomplish a state box transition. With the clear box, the structured programming constructs and stepwise refinement are used as illustrated in Figure 29.6. A program function, *f*, is refined into a sequence 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

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 29.6
Stepwise refinement



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

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

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

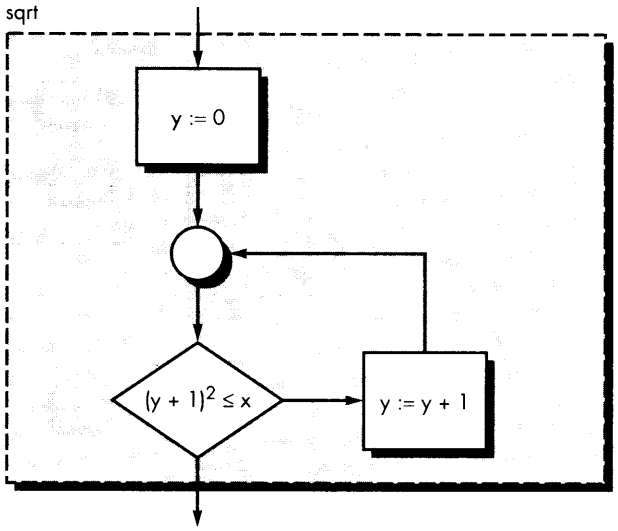
When function *m* is refined as a loop, the correctness conditions for all input to *m* are

- 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

FIGURE 29.7
Computing the integer part of a square root [LIN79]



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 29.7.

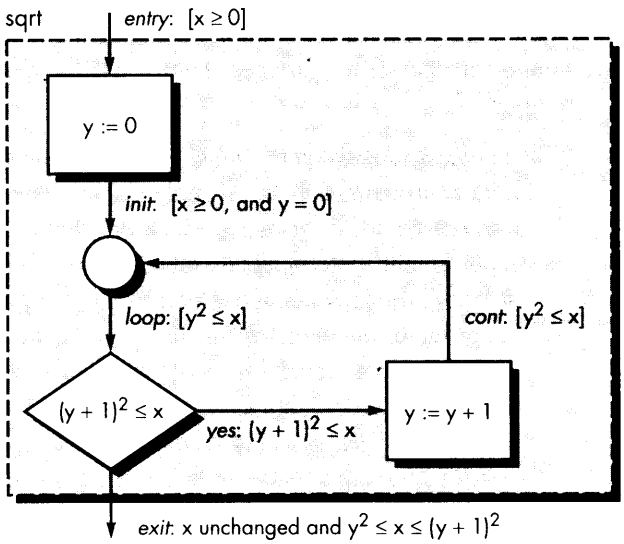
To verify the correctness of this design, we must define entry and exit conditions as noted in Figure 29.8. The entry condition notes that *x* must be greater than or equal to 0. The exit condition requires that *x* remain unchanged and that *y* satisfy the expression noted in the figure. To prove the design correct, it is necessary to prove the conditions *init, loop, cont, yes,* and *exit* shown in Figure 29.8 are true in all cases. These are sometimes called *subproofs.*

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.*

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.
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 A negative value for the square root has no meaning in this context.

FIGURE 29.8
Proving the design correct [LIN79]



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 29.7. 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].

29.3.2 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 hierarchy 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 29.9 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.
- *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

FIGURE 29.9
A design with subproofs

```
                                Subproofs:

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

4 This section and Figures 29.7 through 29.9 have been adapted from [LIN94] and are used with permission.

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

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.*

29.4 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 7) have been executed successfully.

> **"Quality is not an act, it is a habit."**
>
> **Aristotle**

29.4.1 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.

Even if you decide against the cleanroom approach, it's worth considering statistical use testing as an integral part of your test strategy.

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 set of stimuli[5] according to the usage probability distribution. To illustrate, consider the *SafeHome* 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, random numbers between 1 and 99 are generated. Each 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

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

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 is likely to be high.

29.4.2 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.

How do we certify a software component?

The certification approach involves five steps [WOH94]:

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.

29.6 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

29.1. 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.

29.2. Develop a box structure specification for a portion of the PHTRS system introduced in Problem 8.10.

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

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

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

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

29.7. 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.

29.8. Select a program that you use regularly (e.g., an e-mail handler, a word processor, a spreadsheet program), and 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 29.4.1.

29.9. For the program stimuli and probability distribution table developed in Problem 29.8, use a random number generator to develop a set of test cases for use in statistical use testing.

29.10. Document a correctness verification proof for the bubble sort discussed in Problem 29.5.

Further Readings and Information Sources

Prowell et al. (*Cleanroom Software Engineering: Technology and Process,* Addison-Wesley, 1999) provide 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. The Data and Analysis Center for Software (DACS) (www.dacs.dtic.mil) provides many useful papers, guidebooks, and other information sources on cleanroom software engineering.

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 (www.cleansoft.com) has prepared a bibliography on cleanroom topics. Many are available in downloadable format.

Design verification via proof of correctness lies at the heart of the cleanroom approach. Books by Stavely (*Toward Zero-Defect Software,* Addison-Wesley, 1998), 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 is available on the Internet. An up-to-date list of World Wide Web references can be found at the SEPA Web site: **http://www.mhhe.com/pressman.**

CHAPTER

30

COMPONENT-BASED DEVELOPMENT

KEY CONCEPTS

adaptation
CBD
CBSE
  process
  economics
classification
component types
domain engineering
middleware
qualification
reuse environment
structure points

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 a very short time and demand 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 be accomplished in a cost- and time-effective manner? Can appropriate incentives be established to encourage software engineers to reuse rather than reinvent? Is

QUICK LOOK

What is it? You purchase an entertainment system and bring it home. Each component has been designed to fit a specific audio-video architecture—connections are standardized, and communication protocol has been predetermined. Assembly is easy because you don't have to build the system from hundreds of discrete parts. Component-based software engineering (CBSE) 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 home entertainment 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 helping to ensure [illegible]

What are the steps? [illegible] engineering and [illegible]

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.

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?

Even today, software engineers grapple with these and other questions about software component reuse. We look at some of the answers in this chapter.

30.1 Engineering of Component-Based Systems

WebRef

Useful information on CBSE for WebApps can be found at **www.cbd-hq.com.**

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 (Chapter 7). An architectural design (Chapter 10) 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 may attempt 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, software engineering methods are applied to build those new components that must be developed to meet the requirement(s). But for those requirements that are addressed with available components, a different set of software engineering activities commences: qualification, adaptation, composition, and update. Each of these CBSE activities is discussed in more detail in Section 30.4.

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:

- *Qualified components*—assessed by software engineers to ensure that not only functionality, but performance, reliability, usability, and other quality factors (Chapter 26) 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.

---

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

30.2 The CBSE Process

The *CBSE process* is 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]. The process model for component-based software engineering emphasizes parallel tracks in which domain engineering (Section 30.3) occurs concurrently with component-based development.

Figure 30.1 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 provides 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.

The *analysis* and *architectural design* steps defined as part of *component-based development* (Figure 30.1) can be implemented within the context of an *abstract design paradigm* (ADP) [DOG03]. An ADP implies that the overall model of the software—represented as data, function, and behavior (control)—can be decomposed hierar-

FIGURE 30.1
A process model that supports CBSE