Argila, C.A., Jones, C., Martin, J.J. "Software Engineering"
*The Electrical Engineering Handbook*
Ed. Richard C. Dorf
Boca Raton: CRC Press LLC, 2000

# 90

# Software Engineering



Carl A. Argila
*Software Engineering Consultant*

Capers Jones
*Software Productivity Research, Inc.*

Johannes J. Martin
*University of New Orleans*

## 90.1 Tools and Techniques[1]

*Carl A. Argila*

The last decade has seen a revolution in software engineering tools and techniques. This revolution has been fueled by the ever-increasing complexity of the software component of delivered systems. Although the software component of delivered systems may not be the most expensive component, it is usually, however, "in series" with the hardware component; if the software doesn't work, the hardware is useless.

Traditionally, software engineering has focused primarily on computer programming with ad hoc analysis and design techniques. Each software system was a unique piece of intellectual work; little emphasis was placed on architecture, interchangeability of parts, reusability, etc. These ad hoc software engineering methods resulted in the production of software systems which did not meet user requirements, were usually delivered over budget and beyond schedule, and were extraordinarily difficult to maintain and enhance.

In an attempt to find some solutions to the "software crisis," large governmental and private organizations motivated the development of so-called "waterfall" methods. These methods defined formal requirement definition and analysis phases, which had to be completed before commencing a formal design stage, which in turn had to be completed before beginning a formal implementation phase, etc. Although waterfall methods were usually superior to ad hoc methods, large and complex software systems were still being delivered over budget and beyond schedule, which did not meet user requirements. There were several reasons for this. First, waterfall methods focus on the generation of *work products* rather than "engineering." Simply put, writing documents is not the same as doing good engineering. Second, the waterfall methods do not support the *evolution* of system requirements throughout the development life cycle. Also, the prose English specifications produced within the waterfall methods are not well suited to describing the complex behaviors of software systems.

[1]The material in this article was originally published by CRC Press in *The Electrical Engineering Handbook,* Richard C. Dorf, Editor, 1993.

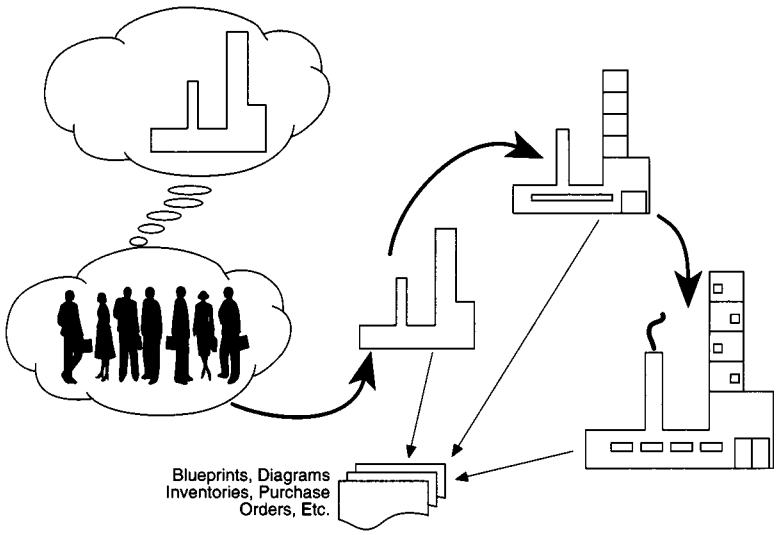



**FIGURE 90.1** Model-based software engineering.

The basic, underlying philosophy of how software systems should be developed changed dramatically in 1978 when Tom DeMarco published his truly seminal book, *Structured Analysis and System Specification* [DeMarco, 1979]. DeMarco proposed that software systems should be developed like any large, complex engineering systems—by first building scale models of proposed systems so as to investigate their behavior. This *model-based software engineering* approach is analogous to that used by architects to specify and design large complex buildings (see Fig. 90.1). We build scale models of software systems for the same reason that architects build scale models of houses, so that users can visualize living with the systems of the future. These models serve as vehicles for communication and negotiation between users, developers, sponsors, builders, etc. Model-based software engineering holds considerable promise for enabling large, complex software systems to be developed on budget, within schedule, while meeting user requirements [see Harel, 1992].

As shown in Fig. 90.2, a number of specific software development models may be built as part of the software development process. These models may be built by different communities of users, developers, customers, etc. Most importantly, however, these models are built in an *iterative* fashion. Although work products (documents, milestone reviews, code releases, etc.) may be delivered chronologically, models are built iteratively throughout the software system's development life cycle.

In Fig. 90.3 we illustrate the distinction between *methodology*, *tool*, and *work product*. A number of differing software development methods have evolved, all based on the underlying model-based philosophy. Different methods may in fact be used for the requirements and analysis phases of project development than for design and implementation. These differing methods may or may not integrate well. Tools such as **CASE** may support all, or only a part, of a given method. Work products, such as document production or code generation, may be generated manually or by means of CASE tools.

This article will present a synopsis of various practical software engineering techniques which can be used to construct software development models; these techniques are illustrated within the context of a simple case study system.

## Approach

One of the most widely accepted approaches in the software engineering industry is to build two software development models. An **essential model** captures the behavior of a proposed software system, independent of implementation specifics. An essential model of a software system is analogous to the scale model of a house built by an architect; this model is used to negotiate the *essential* requirements of a system between customers and developers. A second model, an **implementation model**, of a software system describes the technical aspects of a proposed system within a particular implementation environment. This model is analogous to the detailed

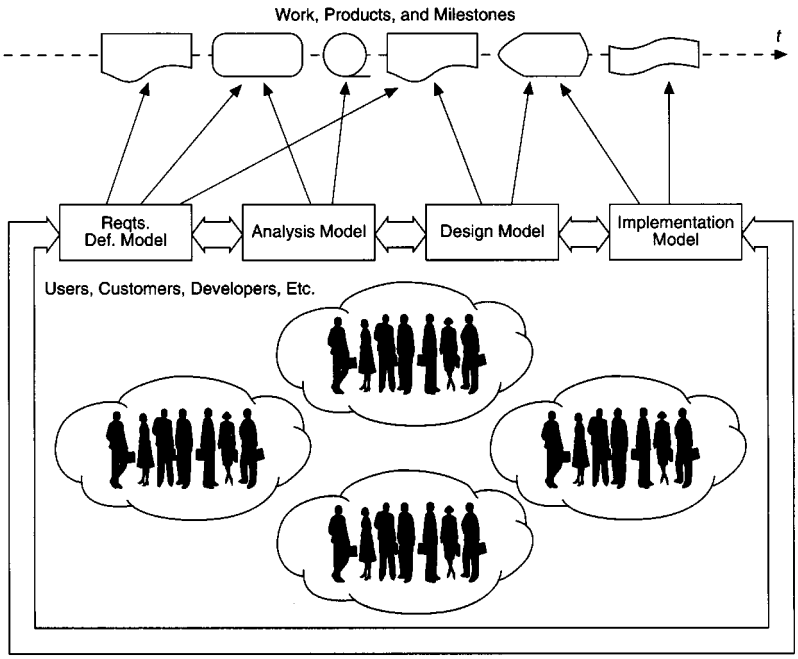



**FIGURE 90.2** Modeling life cycle.

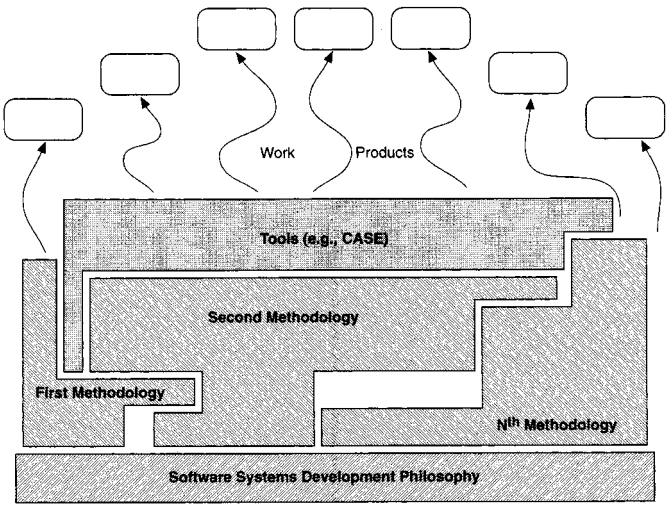



**FIGURE 90.3** Methods, tools and work products.

blueprints created by an architect; it specifies the *implementation* aspects of a system to those who will do the construction. These models [described in Argila, 1992] are shown in Fig. 90.4. The essential and implementation models of a proposed software system are built in an iterative fashion.

## Methods

The techniques used to build the essential and implementation models of a proposed software system are illustrated by means of a simple case study. The Radio Button System (RBS) is a component of a fully automated, digital automobile sound system. The RBS monitors a set of front-panel *station selection buttons* and performs station selection functions.

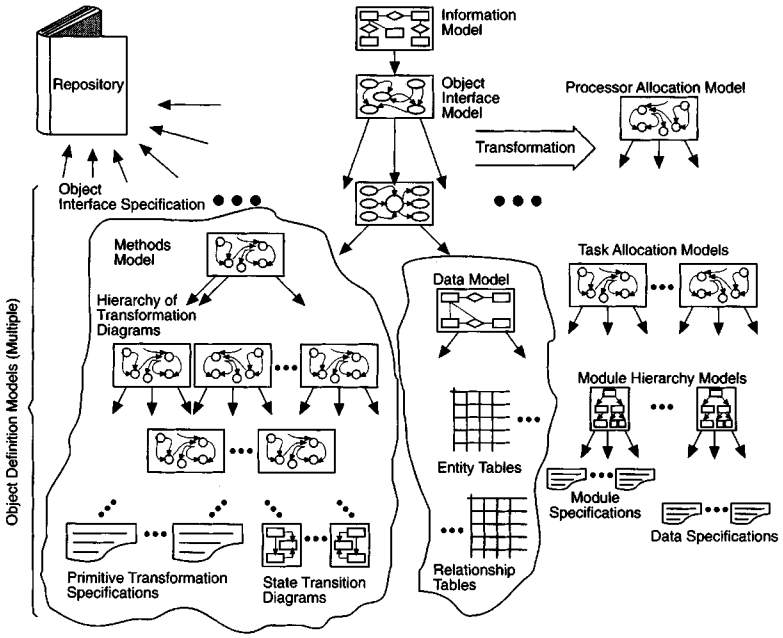



**FIGURE 90.4** Software engineering methods overview.

When a station selection button is momentarily depressed, the RBS causes a new station to be selected. This selection is made on the basis of station-setting information stored within the RBS. The RBS can "memorize" new station selections in the following manner: When a given station selection button is depressed longer than "momentarily" (say, for more than 2 seconds), the currently selected station will be "memorized." Future momentary depressions of this button will result in this "memorized" station being selected.

The RBS also performs a muting function. While a station is being selected, the RBS will cause the *audio system* to mute the audio output signal. The RBS will also cause the audio output signal to be muted until a new station selection has been successfully memorized.

The RBS interfaces with the front-panel station selection buttons by "reading" a single-byte memory location. Each bit position of this memory location is associated with a particular front-panel station selection button. The value of 0 in a given bit position indicates that the corresponding button is *not* depressed. The value of 1 in that bit position indicates that the corresponding button *is* depressed. (For example, 0000 0000 indicates no station selection buttons are currently depressed; 0000 0010 indicates that the second button is currently depressed, etc.)

The RBS interfaces with the *tuning system* by means of a common memory location. This single-byte memory location contains a non-negative integer value which represents a station selection. (For example, 0000 0000 might represent 87.9 MHz, 0000 0001 might represent 88.1 MHz, etc.) The RBS may "read" this memory location to "memorize" a current station selection. The RBS may also "write" to this memory location to cause the tuning system to select another station.

Finally, the RBS interfaces with the audio system by sending two signals. The RBS may send a MUTE-ON signal to the audio system causing the audio system to disable the audio output. A MUTE-OFF signal would cause the audio system to enable the audio output.

## Information Modeling

The construction of an **information model** is fundamental to so-called object-oriented approaches. An information model captures a "view" of an application domain within which a software system will be built. Information models are based on entity-relationship diagrams and underlying textual information. A sample

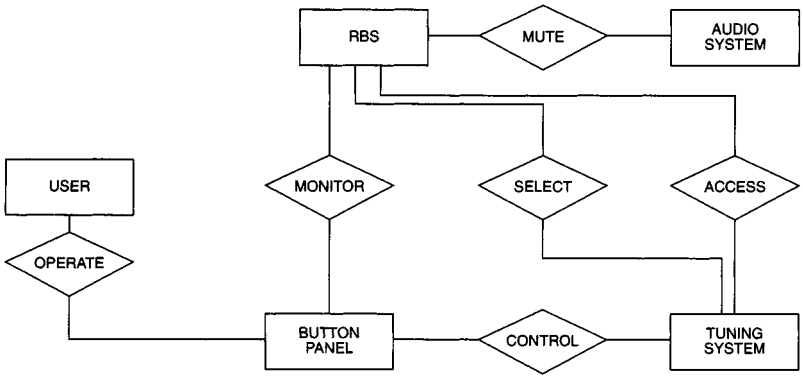
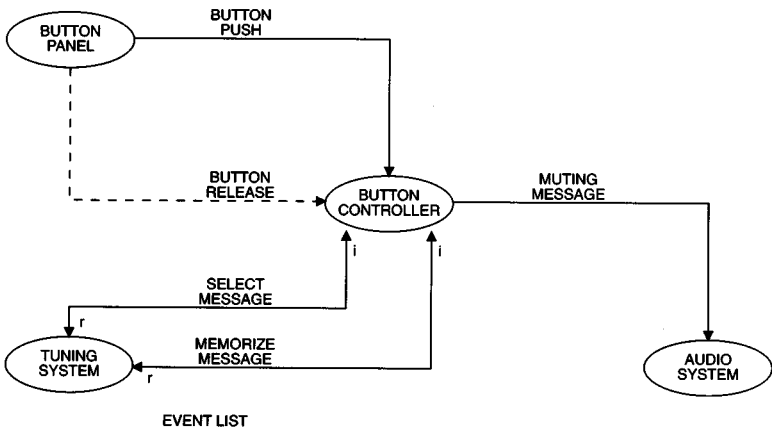



**FIGURE 90.5** RBS information model.

information model for the RBS is shown in Fig. 90.5. Entities (shown as rectangles) represent "things" or "**objects**" in the application domain. Entities may be established by considering principal nouns or noun phrases in the application domain. Entities have *attributes* associated with them which express the qualities of the entity. Entities participate in *relationships;* these are shown as diamonds in the entity-relationship diagram. Relationships may be determined by considering principal verbs or verb phrases in the application domain. Relationships have *cardinality* associated with them and entities may participate *conditionally* in relationships. Finally, there are special kinds of relationships which show *hierarchical relationships* between objects.

## Essential Modeling

The essential model consists of a number of graphical components with integrated textual information. Figure 90.6 shows the **object collaboration model** for the RBS. This model depicts how a collection of objects or entities can communicate (by exchanging messages) to perform the proposed system functions. An *event list* is part of this model; it shows what responses must be produced for a given external stimulus.




EVENT LIST

| EVENT | RESPONSE |
|---|---|
| 1. Selection button depressed. | 1. Mute audio system. |
| 2. Button released within two seconds after it was depressed. | 2. Select a memorized station.<br>Un-mute audio system. |
| 3. Two seconds elapses after button was depressed while button remained depressed. | 3. Memorize current station.<br>Un-mute audio system. |

**FIGURE 90.6** RBS object collaboration model.

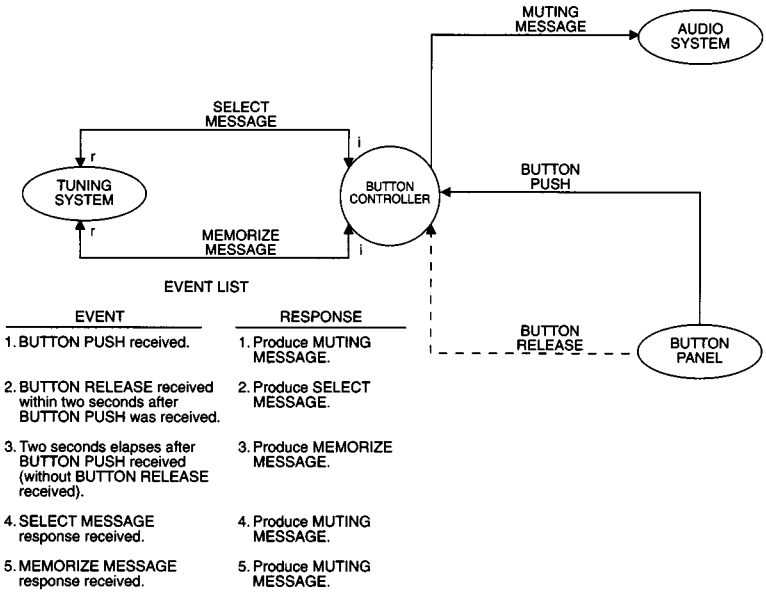



**FIGURE 90.7** RBS object interface specification.

For each object there is an **object interface specification** (as shown in Fig. 90.7) which shows the public and private interfaces to an object. An event list is also associated with this specification; it shows how the object will respond to external stimuli. A hierarchy of **transformation diagrams** is associated with each object specification (as shown in Fig. 90.8 for the RBS). This diagram defines all of the functions or "methods" which the object performs. Some behavior may be expressed by means of a **state transition diagram** (Fig. 90.9).

## Implementation Modeling

Two principal activities must be accomplished in transitioning from the essential to the implementation model. First, all of the methods and data encapsulated by each object must be mapped to the implementation environment. This process is illustrated in Fig. 90.10. Second, all of the details which were ignored in the essential model (such as user interfaces, communication protocols, hardware limitations, etc.) must now be accounted for.

Each component of the essential model must be allocated to hardware processors. Within each hardware processor, allocation must be continued to the *task* level. Within each task, the computer program controlling that task must be described. This latter description is accomplished by means of a **module structure chart.** As illustrated in Fig. 90.11 for one component of the RBS, the module structure chart is a formal description of each of the computer program units and their interfaces.

## CASE Tools

The term *computer-aided software engineering* (CASE) is used to describe a collection of tools which automate all or some of various of the software engineering life cycle phases. These tools may facilitate the capturing, tracking and tracing of requirements, the construction and verification of essential and implementation models and the automatic generation of computer programs. Most CASE tools have an underlying *project repository* which stores project-related information, both textual and graphical, and uses this information for producing reports and work products.

CASE tool features may include:

- Requirements capture, tracing, and tracking
- Maintenance of all project-related information
- Model verification
- Facilitation of model validation

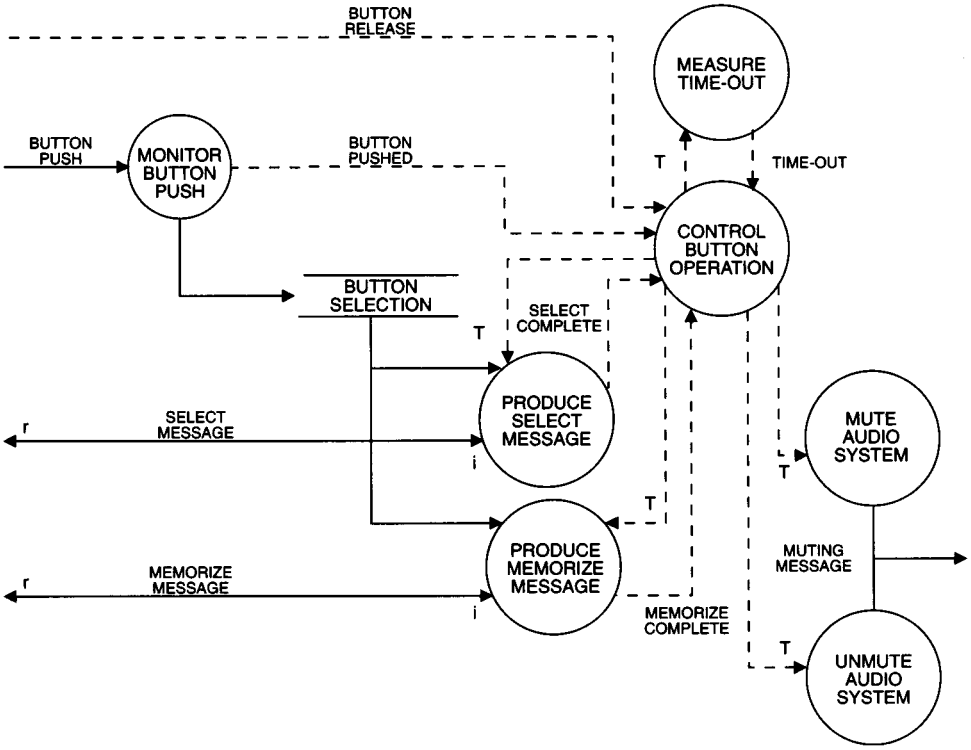



**FIGURE 90.8** RBS transformation diagram.

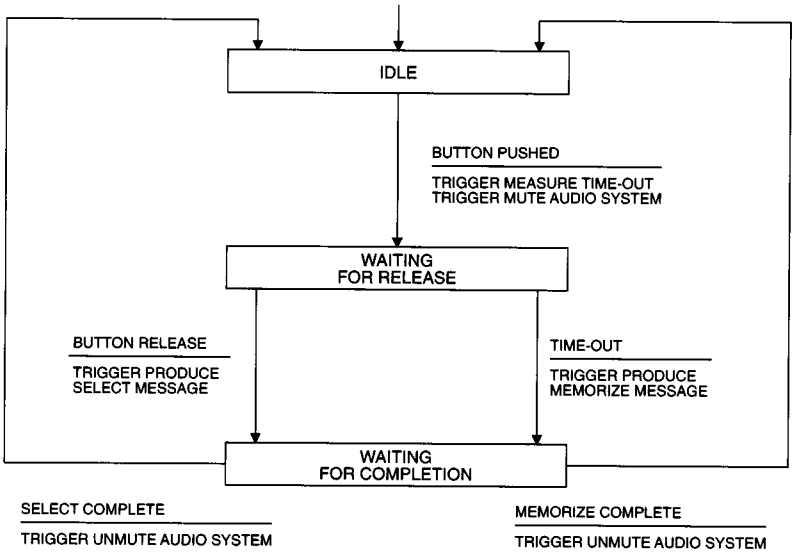



**FIGURE 90.9** RBS state transition diagram.

- Document production
- Configuration management
- Collection and reporting of project management data
- CASE data exchange

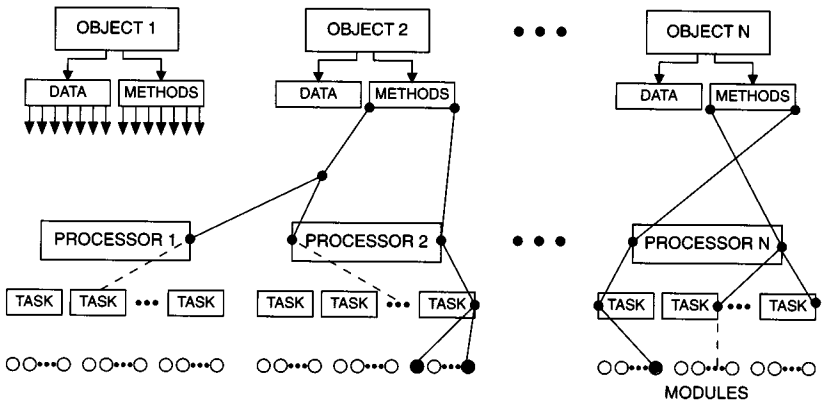



**FIGURE 90.10** Implementation modeling.

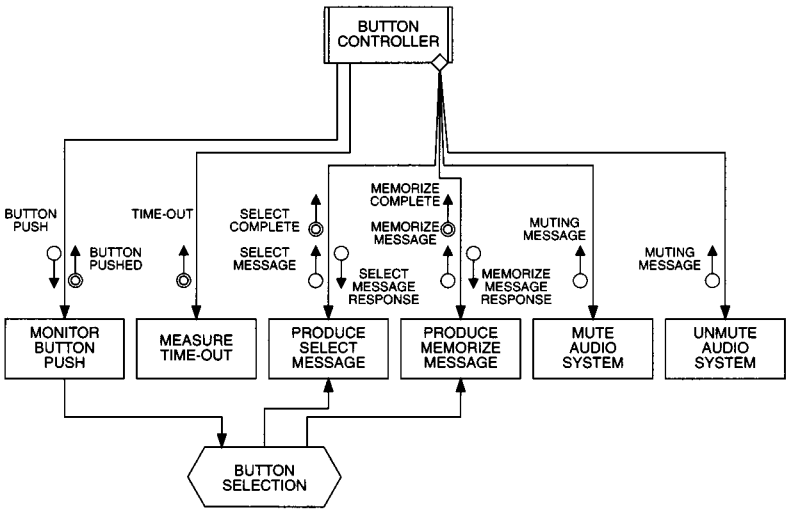



**FIGURE 90.11** RBS module structure chart.

## Defining Terms

**CASE:** Computer-aided software engineering. A general term for tools which automate various of the software engineering life cycle phases.

**Essential model:** A software engineering model which describes the behavior of a proposed software system independent of implementation aspects.

**Implementation model:** A software engineering model which describes the technical aspects of a proposed system within a particular implementation environment.

**Information model:** A software engineering model which describes an application domain as a collection of objects and relationships between those objects.

**Module structure chart:** A component of the implementation model; it describes the architecture of a single computer program.

**Object:** An "entity" or "thing" within the application domain of a proposed software system.

**Object collaboration model:** A component of the essential model; it describes how objects exchange messages in order to perform the work specified for a proposed system.

**Object interface specification:** A component of the essential model; it describes all of the public and private interfaces to an object.
**State transition diagram:** A component of the essential model; it describes event-response behaviors.
**Transformation diagram:** A component of the essential model; it describes system functions or "methods."

## Related Topic

90.3 Programming Methodology

## References


C. Argila, "Object-oriented real-time systems development" (video course notes), Los Angeles: University of Southern California IITV, June 11, 1992.
G. Booch, *Object-Oriented Design with Applications*, Redwood City, Calif.: Benjamin/Cummings, 1991.
P. Coad and E. Yourdon, *Object-Oriented Analysis*, 2nd ed., New York: Prentice-Hall, 1991.
P. Coad and E. Yourdon, *Object-Oriented Design*, New York: Prentice-Hall, 1991.
T. DeMarco, *Structured Analysis and System Specification*, New York: Prentice-Hall, 1979.
D. Harel, "Biting the silver bullet," *Computer*, January 1992.
J. Rumbaugh et al., *Object-Oriented Modeling and Design*, New York: Prentice-Hall, 1991.
S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, New York: Prentice-Hall, 1988.
S. Shlaer and S. Mellor, *Object Life-Cycles: Modeling the World in States*, New York: Prentice-Hall, 1992.
P. Ward and S. Mellor, *Structured Development for Real-Time Systems*, New York: Prentice-Hall, vol. 1, 1985; vol. 2, 1985; vol. 3, 1986.
E. Yourdon and L. Constantine, *Structured Design*, 2nd ed., New York: Prentice-Hall, 1975, 1978.


## Further Information

A video course presenting the software engineering techniques described here is available [see Argila, 1992]. The author may be contacted for additional information and comments at (800) 347-6903.

# 90.2 Testing, Debugging, and Verification


*Capers Jones*


Achieving acceptable levels of software quality has been one of the most troublesome areas of software engineering since the industry began. It has been so difficult to achieve error-free software that, historically, the cost of finding and fixing **"bugs"** has been the most time-consuming and expensive aspect of large-scale software development.

Software quality control is difficult to achieve, but the results of the careful application of **defect prevention** and **defect removal** techniques are quite good. The software producers who are most effective in quality control discover several derivative benefits as well: software with the highest quality levels also tends to be the least expensive to produce, tends to have minimum schedules during development, and also tends to have the highest levels of post-release user satisfaction.

The topic of defect prevention is outside the primary scope of this article. However, it is important to note that the set of technologies associated with defect prevention must be utilized concurrently with the technologies of defect removal in order to achieve high levels of final quality. The technologies which prevent defects are those concerned with optimizing both clarity and structure and with minimizing ambiguity. Joint application design (JAD) for preventing requirements defects; prototyping; reuse of certified material; clean-room development; any of the standard structured analysis, design, and coding techniques; and quality function deployment (QFD) for evaluating end-user quality demands are examples of the technologies associated with defect prevention. Many aspects of total quality management (TQM) programs are also associated with defect prevention.

**TABLE 90.1** Origins and Causes of Software Defects

| | Defect Causes | | |
|---|---|---|---|
| Defect Origins | Errors of Omission | Errors of Commission | Errors of Clarity or Ambiguity |
| Requirements defects | Frequent | Seldom | Frequent |
| Design defects | Frequent | Frequent | Frequent |
| Coding defects | Frequent | Frequent | Frequent |
| Document defects | Frequent | Seldom | Frequent |
| Bad fix defects | Seldom | Frequent | Seldom |

## The Origins and Causes of Software Defects

Before software defects can be prevented or removed effectively, it is necessary to know where defects originate and what causes them. There are five primary origin points for software defects and three primary causes. The five primary origin points are (1) requirements, (2) design, (3) code, (4) user documentation, and (5) bad fixes, or secondary defects that occur while attempting to repair a primary defect. The three primary causes of software defects are (1) errors of omission, (2) errors of commission, and (3) errors of clarity or ambiguity. Table 90.1 shows the interaction of software defect origins and defect types.

The phrase *errors of omission* refers to problems caused by a failure to include vital information. Such errors are frequently encountered in requirements, specifications, source code, and user documents. An example of such an error became highly visible on February 29, 1992, when it was discovered that the calendar routine for a particular application omitted leap year calculations, thus shutting down the system at midnight on the 28th. From 15% to more than 30% of the post-deployment problems encountered in software can be traced to errors of omission. The probability and frequency of such errors rises with the size of the system.

The phrase *errors of commission* refers to problems caused by an action that is not correct, such as looping through a counter routine one time too often or including conditions in a specification that are mutually contradictory. Such errors are frequently encountered in design, code, and in "bad fixes" or secondary defects created as a by-product of repairing prior defects. From 40 to 65% of the post-deployment problems encountered in software can be traced to errors of commission, thus making it the most common source of software problems.

The phrase *errors of clarity or ambiguity* refers to problems caused by two or more interpretations of the same information. For example, a requirement may contain a phrase that an application should provide "rapid response time" without actually defining what "rapid" means. To the client, sub-second response time may have been intended, but to the development team one-minute response time may have been their interpretation. Such errors are frequently encountered in all software deliverables based on natural language such as English. They are also frequently encountered in source code itself, especially so if the code were not well structured. Also, certain languages such as APL and those using nested parentheses are notorious for being ambiguous. From less than 5% to more than 10% of the post-deployment problems encountered in software can be traced to errors of clarity or ambiguity.

When considering the origins of software defects, it is significant that the requirements and specifications themselves may be the source of as many as 40% of the defects later encountered after deployment. This fact implies that some forms of verification and validation, which assume the requirements and specifications are complete and correct, have hidden logical flaws.

The distribution of defects among the common origin points varies with the size and complexity of the application. Table 90.2 shows the probable defect distributions for four size ranges of software projects, using the language C as the nominal coding language.

The sum of the five defect categories is termed the *defect potential* of a software program or system [Jones, 1986]. The defect potential constitutes the universe of all errors and bugs that might cause an application to either fail to operate or to produce erratic and unacceptable results while operating.

The defect potential of software tends to be alarmingly high. When expressed with the traditional metric "KLOC" (where K stands for 1000 and LOC stands for lines of code) the observed defect potentials have ranged from about 10 bugs per KLOC to more than 100 bugs per KLOC, assuming procedural languages such as C,

**TABLE 90.2** Software Defect Origins and Project Size

| | Software Project Size (Statements in C Language) | | | |
|---|---|---|---|---|
| | 1000 | 10,000 | 100,000 | 1,000,000 |
| Requirements | 5% | 7% | 10% | 15% |
| Design | 10% | 15% | 20% | 25% |
| Code | 70% | 60% | 50% | 40% |
| Documents | 5% | 8% | 10% | 10% |
| Bad fixes | 10% | 10% | 10% | 10% |
| Total | 100% | 100% | 100% | 100% |

Fortran, or Jovial. When expressed with the newer Function Point metric, the range of software defect potentials is from less than 2 to more than 10 bugs per Function Point.

When historical defect data is analyzed, it can easily be seen why defect prevention and defect removal are complementary and synergistic technologies. The defect prevention approaches are used to lower defect potentials, and the defect removal technologies are then used to eliminate residual defects.

Another dimension of concern is the *severity level* of software defects. The severity level scheme used by IBM, and widely adopted by other software producers, is based on a four-point scale. Severity 1 defects are those which stop the software completely or prevent it from being used at all. Severity 2 defects are those where major functions are disabled or unusable. Severity 3 defects are those that can be bypassed or which have a minor impact. Severity 4 defects are very minor problems which do not affect the operation of the software, for example, a spelling error in a text message.

Software **debugging,** testing, and verification methods should be effective against the entire defect potential of software, and not just against coding defects. Defect removal methods should approach 100% in efficiency against severity 1 and severity 2 defects. Further, defect removal methods should deal with errors of omission as well as errors of commission. Software operates at extremely high speed, so defect removal operations must deal with timing and performance problems as well as with structural and logical problems. Finally, and most difficult, defect removal methods should deal with errors of clarity or ambiguity. These are challenging tasks.

## The Taxonomy and Efficiency of Software Defect Removal

There is no standard taxonomy that encompasses all of the many forms of defect removal that can be applied to software [Dunn, 1984]. For convenience, we will divide defect removal operations into three broad categories: pre-test defect removal, testing, and post-release defect removal.

Since the goal of defect removal is the elimination of bugs or defects, the primary figure of merit for a defect removal operation is its *efficiency* [Jones, 1991]. **Defect removal efficiency** is defined as the percent of latent defects which a given removal operation will detect. Cumulative defect removal efficiency is the overall efficiency of a complete series of pre-test removal activities combined with all test stages.

Calculating the efficiency of a defect removal operation, or a series, is necessarily a long-range operation. All defects encountered prior to release are enumerated and running totals are kept. After the first year of production, the client-reported defects and the pre-release defects are aggregated as a set, and the efficiencies of pre-release operations are then calibrated. Thus for large systems with multi-year development cycles, it may take up to five years before defect removal efficiency rates are fully known.

Major computer companies, telecommunications manufacturers, defense contractors, and some software vendors have been measuring defect removal efficiencies for more than 20 years. The body of empirical data is large enough to make general observations about the efficiencies of many different kinds of defect removal operation.

Empirical observations and long-range studies of commercial software have demonstrated that most forms of testing are less than 30% efficient. That is, any given stage of testing such as system testing or acceptance testing is likely to find less than 30% of the latent defects that are actually present. However, a carefully planned series of defect removal operations can achieve very respectable levels of cumulative defect removal efficiency. Computer manufacturers, commercial software producers, and defense software producers may utilize as many as 9 to 20 consecutive defect removal operations, and sometimes achieve cumulative defect removal efficiencies that approach the six sigma quality level, i.e., defect removal efficiency rates approaching 99.999999%.

## Pre-Test Defect Removal

The set of defect removal activities for software carried out prior to the commencement of testing can be further subdivided into *manual defect removal* and *automated defect removal.*

A fundamental form of manual defect removal is termed *desk checking*, or the private review of a specification, code document, or document by the author. The efficiency of desk checking varies widely in response to both individual talents and to the clarity and structure of the materials being analyzed. However, most humans are not particularly efficient in finding their own mistakes, so the overall efficiency of desk checking is normally less than 35%.

The most widely utilized forms of manual defect removal are reviews, inspections, and walkthroughs. Unfortunately common usage blurs the distinction among these three forms of defect removal. All three are group activities where software deliverables such as the requirements, preliminary or detailed design, or a source code listing are discussed and evaluated by technical team members.

Casual reviews or informal walkthroughs seldom keep adequate records, and so their defect removal efficiencies are not precisely known. A few controlled studies carried out by large companies such as IBM indicate that informal reviews or walkthroughs may average less than 45% in defect removal efficiency.

The term *inspection* is often used to define a very rigorous form of manual analysis [Fagan, 1976]. Ideally, the inspection team will have received formal training before participating in their first inspection. Also, the inspection team will follow formal protocols in terms of preparation, execution, recording of defects encountered, and follow-up of the inspection session. The normal complement for a formal inspection includes a moderator, a recorder, a reader to paraphrase the material being inspected, the developer whose work is undergoing inspection, and often one or more additional inspectors.

Formal inspections of the kind just discussed have the highest measured efficiency of any kind of defect removal operation yet observed. For inspections of plans, specifications, and documents the defect removal efficiency of formal inspections can exceed 65%. Formal inspections of source code listings can exceed 60%.

Formal inspections are rather expensive, but extremely valuable and cost-effective. Not only do inspections achieve high rates of defect removal efficiency, but they are also effective against both errors of omission and errors of clarity or ambiguity, which are extremely difficult to find via testing. Formal inspections also operate as a defect prevention mechanism. Those who participate in the inspection process obviously learn a great deal about the kinds of defects encountered. These observations are kept in memory and usually lead to spontaneous reductions in potential defects within a few months of the adoption of formal inspections.

Military software projects in the United States are governed by various military standards such as DOD 2167A and DOD 1614. These standards call for an extensive series of reviews or inspections that are given the generic name of *verification and validation.* The word *verification* is generally defined as ensuring that each stage in the software process follows the logical intent of its predecessor. The term *validation* is generally defined as ensuring that each delivered feature or function can be traced back to a formal requirement.

U.S. military verification and validation has developed its own argot, and those interested in the full set of U.S. military verification and validation steps should refer to the military standards themselves or to some of the specialized books that deal with this topic. Examples of the forms of verification and validation used on U.S. military projects, and the three-letter abbreviations by which they are commonly known, include system requirements review (SRR), system design review (SDR), preliminary design review (PDR), critical design review (CDR).

The defect removal efficiencies of the various forms of military review have not been published, but there is no reason to doubt that they are equivalent or even superior to similar reviews in the civilian domain. The cumulative defect removal efficiency of the full series of U.S. military defect removal operations is rather good: from about 94% to well over 99% for critical weapons systems.

Large military software projects use a special kind of external review, termed *independent verification and validation* (IV&V). The IV&V activities are often performed by contracting organizations which specialize in this kind of work. The efficiency ranges of IV&V reviews have not been published, but there is no reason to doubt that they would achieve levels similar to those of civilian reviews and inspections, i.e., in the 40 to 60% range.

The most elaborate and formal type of manual defect removal methodology is that of *correctness proofs.* The technique of correctness proofs calls for using various mathematical and logical procedures to validate the assertions of software algorithms. Large-scale studies and data on the efficiency of correctness proofs have not

been published, and anecdotal results indicate efficiency levels of less than 30%. To date there is no empirical evidence that software where correctness proofs were used actually achieves lower defect rates in the field than software not using such proofs. Also, the number of correctness proofs which are themselves in error appears to constitute an alarming percentage, perhaps more than half, of all proofs attempted.

Among commercial and military software producers, a very wide spectrum of manual defect removal activities are performed by *software quality assurance* (SQA) teams [Dunn and Ullman, 1982]. The full set of activities associated with formal software quality assurance is outside the primary scope of this article. However, defect prediction, defect measurement, moderating and participating in formal inspections, test planning, test case execution, and providing training in quality-related topics are all aspects of software quality assurance. Organizations that have formal software quality assurance teams will typically average 10 to 15% higher in cumulative defect removal efficiency than organizations which lack such teams.

The suite of automated pre-test tools that facilitate software defect removal has improved significantly in recent years, but still has a number of gaps that may be filled in the future.

For requirements, specifications, user documents, and other software deliverables based on natural language, the use of word processors and their built-in spelling checkers has minimized the presence of minor spelling errors. Also available, although not used as widely as spelling checkers, are a variety of automated grammar and syntax checkers and even textual complexity analyzers. Errors of clarity and ambiguity, long a bane of software, can be reduced significantly by judicious usage of such tools.

Several categories of specialized tools have recently become available for software projects. Many CASE (computer-aided software engineering) tool suites have integral support for both producing and verifying structural descriptions of software applications. There are also tools that can automatically generate test cases from specifications and tools that can trace test cases back to the original requirements. While such tools do not fully eliminate errors of commission, they do provide welcome assistance.

Errors of omission in requirements, specifications, and user documents are the most resistant to automatic detection and elimination. Studies carried out on operating systems and large switching systems have revealed that in spite of the enormous volume of software specifications (sometimes more than 100 English words per source statement, with many diagrams and tables also occurring) more than 50% of the functionality in the source code could not be found described in the specifications or requirements. Indeed, one of the unsolved challenges of software engineering is to determine if it is even theoretically possible to fully specify large and complex software systems. Assuming that full specification is possible, several practical issues are also unknown: (1) What combination of text, graphics, and other symbols is optimal for software specifications? (2) What is the optimum volume or size of the specifications for an application of any given size? (3) What will be the impact of multi-media software extensions on specifications, debugging, and testing?

The number and utility of debugging tools for programming and programmers have made enormous strides over the past few years, and continued progress can be expected indefinitely. Software development in the 1990s often takes place with the aid of what is termed a *programming environment*. The environment constitutes the set of tools and aids which support various programming activities. For debugging purposes, software syntax checkers, trace routines, trap routines, static analyzers, complexity analyzers, cross-reference analyzers, comparators, and various data recording capabilities are fairly standard. Not every language and not every vendor provides the same level of debugging support, but the best are very good indeed.

## Testing Software

There is no standard taxonomy for discussing testing. For convenience, it is useful to consider test planning, test case construction, test case execution, test coverage analysis, and test library control as the major topics encompassing software testing.

For such a mainstream activity as test planning, both the literature and tool suites are surprisingly sparse. The standard reference is Myers [1979]. The basic aspects of test planning are to consider which of the many varieties of testing will be carried out and to specify the number and kind of test cases that will be performed at each step. U.S. military specifications are fairly thorough in defining the contents of test plans. There are also commercial books and courses available, but on the whole the topic of test planning is underreported in the software literature.

A new method for estimating the number of test cases required for software was developed in 1991 and is starting to produce excellent results. A metric termed *Function Points* was invented by A. J. Albrecht of IBM and placed in the public domain in 1978 [Albrecht, 1979]. This metric is derived from the weighted sums of five parameters: the numbers of inputs, outputs, inquiries, logical files, and interfaces that constitute a software application [Garmus, 1991]. The Function Point total of an application can be enumerated during the requirements and design phases.

It is obvious that testing must be carried out for each of the factors utilized by the Function Point metric. Empirical observations indicate that from two to four test cases per Function Point are the normal quantities created by commercial software vendors and computer manufacturers. Thus for an application of 1000 Function Points in size, from 2000 to 4000 test cases may be required to test the user-defined functionality of the application.

Testing can be approached from several directions. Testing which attempts to validate user requirements or the functionality of software, without regard for its inner structure of the application, is termed *black-box* testing.

Black-box test case construction for software has long been a largely manual operation that is both labor-intensive and unreliable. (Empirical observations of operating system test libraries revealed more errors in the test cases than in the product being tested.) Test case generators have been used experimentally since the 1970s and are starting to appear as both stand-alone products and as parts of CASE tool suites. However, in order for test case generation to work effectively, the specifications or written description of the software must be fairly complete, rigorous, and valid in its own right. It is to no purpose to generate automatic test cases for incorrect specifications.

Testing which attempts to exercise the structure, branching, and control flows of software is termed *white-box* or sometimes *glass-box* testing. In this domain, a fairly rich variety of tools has come into existence since 1985 that can analyze the structure and complexity of software. For certain languages such as COBOL and C, tools are available that not only analyze complexity but can restructure or simplify it. In addition, there are tools that can either create or aid in the creation of test cases. Quite a number of tools are available that can monitor the execution of test cases and identify portions of code which have or have not been reached by any given test run.

Although not full test case generators, many supplemental testing tools are available that provide services such as creating matrices of how test cases interact with functions and modules. Also widely used are *record and playback* tools which capture all events during testing. One of the more widely used testing tool classes is that of *test coverage analyzers*. Test coverage analyzers dynamically monitor the code that is being executed while test cases are being run, and then report on any code or paths that may have been missed. Test coverage is not the same as removal efficiency: it is possible to execute 100% of the instructions in a software application without finding 100% of the bugs. However, for areas of code that are not executed at all, the removal efficiency may be zero. Software *defect tracking systems* are exceptionally useful.

Once created, effective test cases will have residual value long after the first release of a software product. Therefore, formal test case libraries are highly desirable, to ensure that future changes to software do not cause regression or damage to existing functionality. Test library tools occur in several CASE tool suites, are available as stand-alone products, and are also often constructed as custom in-house tools by the larger software vendors and computer manufacturers.

Test case execution can be carried out either singly for individual test cases or for entire sets of related test cases using *test scripts*. It is obvious that manually running one test case at a time is too expensive for large software projects. Tools for multi-test execution support, sometimes called a *test harness*, are available in either stand-alone form or as part of several CASE tool suites.

There are no rigorous definitions or standard naming conventions for the kinds and varieties of testing that occur for software. Some of the more common forms of testing include the following.

The testing of an individual module or program by the programmer who created it is normally called *unit test*. Unit testing can include both black-box and white-box test cases. The efficiency of unit test varies with the skill of the programmer and the size and complexity of the unit being tested. However, the observed defect removal efficiency of unit testing seldom exceeds 50% and the average efficiency hovers around 25%. For small projects developed by a single programmer, unit test may be the only test step performed.

For large software projects involving multiple programmers, modules, or components a number of test stages will normally occur that deal with testing multiple facets of the application.

The phrase *new function test* is used to define the testing of capabilities being created or added to an evolving software project. New function testing may consist of testing the aggregated work of several programmers. New function testing may be carried out by the developers themselves, or it may be carried out by a team of testing specialists. The observed efficiency of new function testing is in the 30% range when carried out by developers and in the 35% range when carried out by testing specialists.

The phrase *regression test* is used to define the testing of an evolving software product to ensure that existing capabilities have not been damaged or degraded as a by-product of adding new capabilities. Regression testing is normally carried out using test cases created for earlier releases, or at least created earlier in the development cycle of the current release. The observed removal efficiency of regression testing against the specific class of errors that it targets may exceed 50% when carried out by sophisticated organizations such as computer manufacturers or defense contractors. However, efficiencies in the 20 to 25% range are more common.

The phrase *stress test* is used to define a special kind of testing often carried out for real-time or embedded software. With stress testing, the software is executed at maximum load and under maximum performance levels, to ensure that it can meet critical timing requirements. The observed removal efficiency of stress testing often exceeds 50% against timing and performance-related problems. Stress testing is normally carried out by testing and performance specialists who are supported by fairly sophisticated test tool suites.

The phrase *integration test* is used to define a recurring series of tests that occur when components or modules of an evolving software product are added to the fundamental system. The purpose of integration testing is to ensure that the newer portions of the product can interact and operate safely with the existing portions. The observed removal efficiency of integration testing is normally in the 20 to 35% range. When performed by commercial software producers, integration test is normally carried out by testing specialists supported by fairly powerful tools.

The phrase *system test* is used to define the final, internal testing stage of a complete product before it is released to customers. The test suites that are executed during system test will include both special tests created for system test and also regression tests drawn from the product's test library. The observed removal efficiency of system testing is in the 25 to 35% range when such testing is done by the developers themselves. When system testing is carried out by testing specialists for commercial software, its defect removal efficiency may achieve levels approaching 50%. However, high defect removal efficiency this late in a development cycle often leads to alarming schedule slippages.

The phrase *independent test* is used to define a form of testing by an independent testing contractor or an outside organization. Independent testing is standard for U.S. military software and sometimes used for commercial software as well. Defect removal efficiency of independent test may approach 50%.

The phrase *field test* is used to define testing by early customers of a software product, often under a special agreement with the software vendor. Field testing often uses live data and actual customer usage as the vehicle for finding bugs, although prepared test cases may also occur. The defect removal efficiency of field test fluctuates wildly from customer to customer and product to product, but seldom exceeds 30%.

The phrase *acceptance test* is used to define testing by a specific client, as a precursor to determining whether to accept a software product or not. Acceptance testing may be covered by an actual contract, and if so is a major business factor. The defect removal efficiency of acceptance testing fluctuates from client to client and product to product, but seldom exceeds 30%.

Sometimes the phrases *alpha test* and *beta test* are used as a linked pair of related terms. Alpha testing defines the set of tests run by a development organization prior to release to early customers. Beta testing defines the set of tests and actual usage experiences of a small group of early customers. Unfortunately the term alpha test is so ambiguous that it is not possible to assign a defect removal efficiency rating. Beta testing is better understood, and the observed removal efficiency is usually in the 25% range.

## Selecting an Optimal Series of Defect Prevention and Removal Operations

Since most forms of defect removal are less than 60% efficient, it is obvious that more than a single defect removal activity will be necessary for any software project. For mission-critical software systems where it is imperative to achieve a cumulative defect removal efficiency higher than 99%, then at least 10 discrete defect removal activities should be utilized, and careful defect prevention activities should also be planned (i.e., prototyping, use of structured techniques, etc.).

An effective series of defect removal operations utilized by computer manufacturers, telecommunication manufacturers, and defense contractors who wish to approach the six-sigma quality level (99.999999% efficiency) will include from 16 to 20 steps and resemble the following, although there are variances from company to company and project to project:

### Pre-Test Defect Removal Activities

1. Requirements inspection
2. Functional design inspection
3. Logical design inspection
4. Test plan inspection
5. User documentation inspection
6. Desk checking and debugging by developers
7. Code inspection
8. Software quality assurance review
9. Independent verification and validation (military projects)

### Testing Activities

10. Unit testing by developers
11. New function testing
12. Regression testing
13. Integration testing
14. Stress testing
15. Independent testing (military projects)
16. System testing
17. Field testing
18. Acceptance testing

### Post-Release Defect Removal Activities

19. Incoming defect report analysis
20. Defect removal efficiency calibration (after one year of deployment)

The series of defect removal operations just illustrated can achieve cumulative defect removal efficiencies well in excess of 99%. However, these levels of efficiency are normally encountered only for large mission-critical or important commercial-grade software systems, such as operating systems, defense systems, and telecommunication systems.

A rough rule of thumb can predict that number of defect removal operations that are normally carried out for software projects. Using the Function Point total of the application as the starting point, calculate the 0.3rd power, and express the result as an integer. Thus an application of 100 Function Points in size would normally employ a series of 4 defect removal operations. An application of 1000 Function Points in size would normally employ a series of 8 defect removal operations. An application of 10,000 Function Points in size would normally employ 16 defect removal operations.

## Post-Release Defect Removal

Defect removal operations do not cease when a software project is released. Indeed, one of the embarrassing facts about software is that post-deployment defect removal must sometimes continue indefinitely.

One of the critical tasks of post-release defect removal is the retroactive calculation of defect removal efficiency levels after software has been in use for one year. For a software project of some nominal size, such as 1000 Function Points, assume that 2000 bugs or defects were found via inspections and tests prior to release. The first year total of user-reported defects might be 50. Dividing the pre-release defect total (2000) by the sum of all defects (2050) indicates a provisional defect removal efficiency of 97.56%.

The defect potential of the application can be retroactively calculated at the same time, and in this case is 2.05 defects per Function Point. The annual rate of incoming user-reported defects for the first year is 0.05 defects per Function Point per year.

In addition to calculating defect removal efficiency, it is also useful to trace each incoming user-reported defect back to its origin, i.e., whether the defect originated in requirements, design, code, documentation, or as a result of a "bad fix." Defect origin analysis also requires a sophisticated tracking system, and full configuration control and traceability tools are useful adjuncts.

Note that the use of Function Points rather than the traditional KLOC metric is preferred for both defect potential and for defect origin analysis. KLOC metrics produce invalid and paradoxical results when applied to requirements, specification, and documentation error classes. Since defects outside the code itself often constitute more than 50% of the total bugs discovered, KLOC metrics are harmful to the long-range understanding of software quality.

If additional bugs are reported during the second or subsequent years, then defect removal efficiency levels should be recalculated as necessary. Removal efficiency analysis should also be performed for each specific review, inspection, and test step utilized on the project.

Calculating post-release defect removal efficiency requires an accurate quality measurement system throughout the life cycle. In addition, rules for counting post-release defects must be established. For example, if the same bug is reported more than once by different users, it should still count as only a single bug. If upon investigation user bug reports should turn out to be usage errors, hardware errors, or errors against some other software package, such bug reports should not be considered in calculating defect removal efficiency rates. If bugs are found by project personnel, rather than users, after release to customers, those bugs should still be considered to be in the post-release category.

The leading computer manufacturers, telecommunication companies, defense contractors, and software vendors utilize powerful defect tracking systems for monitoring both pre-release and post-release software defects. These systems record all incoming defects and the symptoms which were present when the defect was discovered. They offer a variety of supplemental facilities as well. Obviously statistical analysis on defect quantities and severities is a by-product of defect tracking systems. Less obvious, but quite important, is support for routing defects to the appropriate repair center and for notifying users of anticipated repair schedules. The most powerful defect tracking systems can show defect trends by product, by development laboratory, by time period, by country, by state, by city, by industry, and even by specific customer.

## Recent Industry Trends in Software Quality Control

Since the previous edition of this handbook, a number of changes have occurred which affect software quality control approaches. Three factors, in particular, are beginning to exert a major influence on the way software is built, inspected, and tested:

1. The explosion of software viruses
2. The on-rushing approach of the Year 2000 problem
3. The marked increase in lawsuits where poor quality is alleged

These three new and emerging topics are likely to stay important software considerations well into the 21st century.

### The Explosion of Software Viruses

Over the past 10 years, software viruses have grown from becoming a rare and minor annoyance to becoming major risks for software projects. The rampant increase in the number of viruses and the ever increasing sophistication of viruses has triggered a new subindustry of viral protection tools and software.

For the purpose of this handbook, viruses have also introduced new and "standard" testing phases aimed at eliminating all possibilities of viral contamination from the final delivered versions of software. As of 1996, every major software vendor in the world now includes viral protection screening as a standard activity, and many include multiple kinds of viral protection tools and approaches. This is a serious problem and it is not showing any sign of becoming less serious.

### The On-Rushing Year 2000 Software Problem

Another very significant change in recent years has been the sudden recognition that at the end of the century when the calendar changes from 1999 to 2000 AD, many of the world's software applications may stop completely

or begin to put out incorrect data. The reason for the Year 2000 problem (as it has come to be called) is because of the historical practice of storing all dates in two-digit form. Thus, calendar year "1997" would be stored as "97". The obvious failing of the two-digit approach will occur when "99" becomes "00". This will throw off normal collating sequences and cause major errors. This problem is an interesting one because, although obvious for more than 20 years, the problem was never found during testing. The reason for this situation is because the two-digit form of storing dates originated as a requirement and was also included in software designs. Therefore, test cases tended to conform to erroneous requirements rather than actually find the problem. The Year 2000 problem provides a strong lesson to the software community that "quality means conformance to user requirements" may sometimes be dangerous. The Year 2000 problem is going to be the most expensive software problem in history, and it was caused primarily by conforming to a requirement that was dangerous and flawed. Now that the end of the century is rapidly approaching, every corporation and government agency in the world is frantically racing to make software repairs before the problem surfaces at the end of 1999. Some hidden and obscure instances of this problem will almost certainly slip through, causing major errors and probable litigation.

## The Emergence of Software Quality as a Litigation Issue

On the topic of software litigation, another major change since the previous edition of this handbook is the increase in lawsuits and litigation involving allegations of poor quality. From observations on half a dozen such lawsuits, Table 90.3 shows the sequence of defect removal operations that are common when clients sue vendors for poor quality. It is an interesting observation that for outsource, military, and systems software that ends up in court for litigation which involves assertions of unacceptable or inadequate quality, the number of testing stages is much smaller, while formal design and code inspections were not utilized at all. Table 90.3 shows the typical patterns of defect removal activities for software projects larger than 1000 function points in size where the client sued the developing organization for producing software with inadequate quality levels. The table simply compares the pattern of defect removal operations observed for reliable software packages with high quality levels to the pattern noted during lawsuits where poor quality and low reliability was part of the litigation. It is interesting and ironic that the "reliable" projects actually had shorter development schedules than the projects that ended up in court with charges of poor quality. The reason for this is because finding and fixing software bugs late in the development cycle is the major cause for slipping delivery dates. High quality and short schedules usually are closely coupled, although few software project managers know this.

It is interesting that during the depositions and testimony of the litigation, the vendor often counter-charges that the short-cuts were made at the direct request of the client. Sometimes the vendors assert that the client

**TABLE 90.3** Defect Removal and Testing Stages Noted During Litigation for Poor Quality

| | Reliable Software | Software Involved in Litigation for Poor Quality |
|---|---|---|
| Formal design inspections | Used | Not used |
| Formal code inspections | Used | Not used |
| | | |
| Subroutine testing | Used | Used |
| Unit testing | Used | Used |
| New function testing | Used | Rushed or omitted |
| Regression testing | Used | Rushed or omitted |
| Integration testing | Used | Used |
| System testing | Used | Rushed or omitted |
| Performance testing | Used | Rushed or omitted |
| Capacity testing | Used | Rushed or omitted |

*Note:* The phrase "rushed or omitted" indicates that the vendor departed from best standard practices by eliminating a stage of defect removal or by rushing it in order to meet an arbitrary finish date or commitment to the client.

ordered the short-cuts even in the face of warnings that the results might be hazardous. As can be seen, software developed under contractual obligations is at some peril if quality control and testing approaches are not carefully performed.

## Summary and Conclusions

Software has been prone to such a wide variety of error conditions that much of the cost and time associated with developing software projects is devoted to defect removal. In spite of the high expense levels and long schedules historically associated with defect removal, deployed software is often unstable and requires continuous maintenance.

Synergistic combinations of defect prevention and defect removal operations, accompanied by careful measurements and the long-range calibration of inspection and test efficiencies, can give software-producing organizations the ability to consistently create stable and reliable software packages. As the state of the art advances, both six-sigma quality levels or even zero-defect quality levels may be achievable. Achieving high quality levels also reduces development schedules and lowers development costs.

## Defining Terms

**Bug:** The generic term for a defect or error in a software program. The term originated with Admiral Grace Hopper in the 1950s, who discovered an actual insect that was blocking a contact in an electromechanical device.

**Debugging:** The generic term for the process of eliminating bugs, i.e., errors, from software programs. The phrase is often used in a somewhat narrow sense to refer to the defect removal activities of individual programmers prior to the commencement of formal testing.

**Defect prevention:** The set of technologies which simplify complexity and reduce the probability of making errors when constructing software. Examples of defect prevention technologies include prototypes, structured methods, and reuse of certified components.

**Defect removal:** The set of activities concerned with finding and eliminating errors in software deliverables. This is a broad term which encompasses both manual and automated activities and which covers errors associated with requirement, design, documentation, code, and bad fixes or secondary defects created as a by-product of eliminating prior bugs.

**Defect removal efficiency:** The ratio of defects discovered and eliminated to defects present. Removal efficiency is normally expressed as a percent and is calculated based on all defects discovered prior to release and for the first year of deployment.

## Related Topic

90.3 Programming Methodology

## References


A.J. Albrecht, "Measuring application development productivity," *Proceedings of the Joint SHARE, GUIDE, IBM Application Development Conference,* October 1979, pp. 83–92.

R.H. Dunn, *Software Defect Removal,* New York: McGraw-Hill, 1984.

R.H. Dunn and R. Ullman, *Quality Assurance for Computer Software,* New York: McGraw-Hill, 1982.

M.E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal,* vol. 15, no. 3, pp. 182–211, 1976.

D. Garmus, Ed., *Function Point Counting Practices Manual,* Version 4.0. Westerville, Ohio: International Function Point Users Group (IFPUG), 1995.

C. Jones, *Programming Productivity,* New York: McGraw-Hill, 1986.

C. Jones, *Applied Software Measurement,* 2nd ed., New York: McGraw-Hill, 1996.

G. J. Myers, *The Art of Software Testing,* New York: John Wiley, 1979.

## Further Information

Both specialized and general-interest journals cover defect removal and software quality control. There are also frequent conferences, seminars, and workshops on these topics. Some of the journals and books include:

*ACM Transactions on Software Engineering and Methodology (TOSEM).* The Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036.

*American Programmer.* American Programmer, Inc., 161 West 86th Street, New York, NY 10024-3411.

*IEE Software Engineering Journal.* Institution of Electrical Engineers and the British Computer Society, Michael Faraday House, Six Hills Way, Stevenage, Herts, SG1 2AY, United Kingdom.

*IEEE Transactions on Software Engineering.* Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY 10017.

*ITEA Journal of Test and Evaluation.* International Test and Evaluation Association, 4400 Fair Lakes Court, Fairfax, VA 22033.

*Software Maintenance News.* Software Maintenance Society, 141 Saint Marks Place, Suite 5F, Staten Island, NY 10301.

T. Gilb and D. Graham, *Software Inspections,* Reading, Mass.: Addison Wesley, 1993.

B. Beizer, *Software Testing Techniques,* International Thomson Computer Press, 1995.

C. Jones, *Software Quality — Analysis and Guidelines for Success,* International Thomson Computer Press, 1996.

# 90.3 Programming Methodology

*Johannes J. Martin*

Programming methodology is concerned with the problem of producing and managing large software projects. It relates to programming as the theory of style relates to writing prose. Abiding by its rules does not, in itself, guarantee success, but ignoring them usually creates chaos. Many useful books have been written on the subject and there is a wealth of primary literature. Some of these books, that themselves contain numerous pointers to the primary literature, are listed at the end of this section.

The rules and recommendations of programming methodology are rather independent of particular programming languages; however, significant progress in the understanding of programming methods has always led to the design of new languages or the enhancement of existing ones. Because of the necessity of upward compatibility, enhanced old languages are, in comparison to new designs, less likely to realize progressive programming concepts satisfactorily. Consequently, the year of its design usually determines how well a language supports state-of-the-art programming methods.

Programming methodology promotes program correctness, maintainability, portability, and efficiency.

***Program Correctness.*** For some relatively simple programs specifications and even verifications can be formalized. Unfortunately, however, formal specification methods are not available or not sufficiently developed for most programs of practical interest, and the specifications, given as narratives, are most likely less than complete. For the situations not explicitly covered by the specifications, the resulting program may exhibit bizarre behavior or simply terminate. Furthermore, informal methods of demonstrating correctness may also miss points that were, indeed, addressed by the specifications. The costs of these problems may range from time wasted in the workplace to injuries and loss of lives.

Short of formal methods for proving **program correctness,** simplicity of design gives informal methods a chance to succeed.

***Maintainability.*** Since the programmer who will do the **program maintenance** is usually not the one who also did the original coding (or if he did he cannot be expected to remember the details), proper documentation and straightforward design and implementation of the pertinent algorithms is mandatory.

***Portability.*** A program is called (easily) portable if it can be adapted to a different computer system without much change. One step toward **program portability** is using a high-level language common to both computer systems. The second step is avoiding the use of idiosyncratic features of a system if at all possible or, if such a

feature must be used, to isolate its use to small, well-designed program segments that can easily be rewritten for the new system.

***Efficiency.*** The costs of running a computer program are determined (1) by the time needed to compute the desired result and (2) by the amount of storage space the program requires to run. As the costs of computer hardware have declined dramatically over the past decade, the importance of program efficiency has similarly declined. Yet, there are different dimensions to efficiency. Choosing the right algorithm is still crucial, while local optimization should not be pursued, if it increases the complexity and decreases the clarity of a program and thereby jeopardizes its correctness and maintainability.

## Analysis of Algorithms

The solution of a problem as a sequence of computational steps, each taking finite time, is called an algorithm. As an example consider the algorithm that finds the greatest element in a random list of $n$ elements:

1. Give the name $A$ to the first element of the list.
2. For each element $x$ of the list beginning with the second and proceeding to the last, if $x > A$ then give the name $A$ to $x$.
3. The element named $A$ is the greatest element in the list.

Assume that examining the "next" element, comparing it with $A$ and possibly reassigning the name $A$ to the new element takes a fixed amount of time that does not depend on the number of elements in the list. Then the total time for finding the maximum increases proportional to the length $n$ of the list. We say the *time complexity* of the algorithm is of *order n*, denoted by $O(n)$.

While there are not many algorithms for finding the maximum (or minimum) in an unordered list, other tasks may have many different solutions with distinctly different performances. The task of putting a list in ascending or descending order (sorting), for example, has many different solutions, with time complexities of $O(n^2)$ and $O(n \cdot \log(n))$. Compared with the $O(n \cdot \log(n))$ algorithms, the $O(n^2)$ algorithms are simpler and their individual steps take less time, so that they are the appropriate choice, if $n$, the number of items to be sorted, is relatively small. However, as $n$ grows, the balance tips decidedly toward the $O(n \cdot \log(n))$ algorithms. In order to compute the break-even point suppose that the individual steps of an $O(n \cdot \log(n))$ and an $O(n^2)$ algorithm take $k \cdot t$ and $t$ units of time, respectively. Thus, for the break-even point we have

$$t \cdot n^2 = k \cdot t \cdot n \cdot \log(n)$$

hence

$$k = n/\log(n)$$

With $k = 5$, for example, the break-even point is $n = 13$. The difference in performance of the two algorithms for large numbers of $n$ is quite dramatic. Again with $k = 5$, the $O(n \cdot \log(n))$ algorithm is 217 times faster if $n =$ 10,000, and it is 14,476 times faster if $n =$ 1,000,000. If the faster algorithm would need, e.g., 10 minutes to sort a million items, the slower one would require about 100 days.

Consequently, while efficiency is no longer a predominant concern, analyzing algorithms and choosing the proper one is still a most important step in the process of program development.

## Flow of Control

The term *flow of control* refers to the order in which individual instructions of a program are performed. For the specification of the flow of control, assembly (machine) languages (and older high-level languages) provide conditional and unconditional branch instructions. Programmers discovered that the indiscriminate use of these instructions frequently leads to flow structures too complex to be analyzed at reasonable costs. On the other hand, keeping the flow of control reasonably simple is, indeed, possible since three elementary control

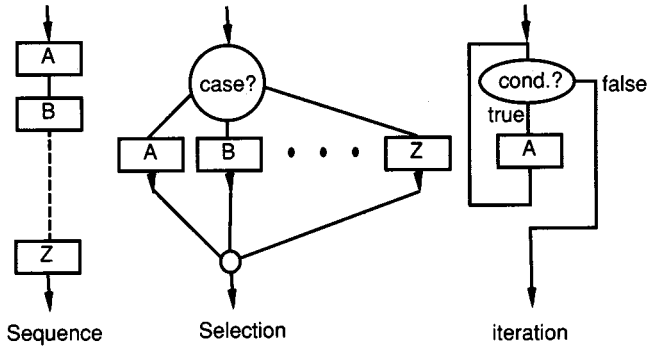



**FIGURE 90.12** Basic control constructors.

constructors are sufficient to express arbitrary algorithms. Using only these constructors contributes in an essential way to the manageability of programs. The basic rules are as follows:

1. A proper computational unit has exactly one entry point and one exit.
2. New computational units are formed from existing units A, B, . . . by a.sequencing (perform A, then B, and so forth) b.selection (perform A or B or . . ., depending on some condition) c.iteration (while some condition prevails, repeat A).

Figure 90.12 shows diagrams that illustrate these rules.

High-level languages support these three constructors by (1) juxtaposition for sequencing, (2) if-then-else and case (switch) constructs for selection, and (3) while-do and for loops for iteration. The somewhat rigid restriction on loops sometimes requires the duplication of code. Some languages ease this problem by providing a *break* or *leave* statement that permits leaving a loop prematurely (Fig. 90.13). A typical example for a program that profits from this mechanism is a loop that reads and processes records and terminates when a record with some given property is found. In Fig. 90.13 block B reads and block A processes a record. With a strict while-loop the code of B must be duplicated.

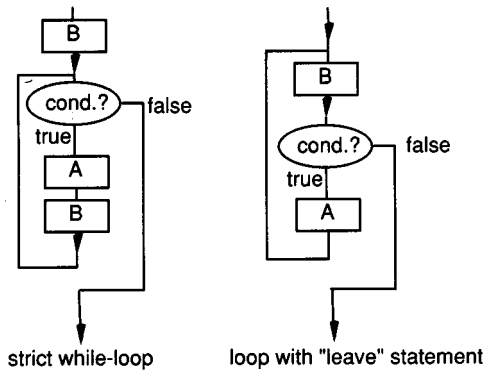



**FIGURE 90.13** The use of a leave statement.

## Abstraction

In programming, abstraction refers to the separation of what needs to be done from the details of how to do it. As it facilitates the separation of unrelated issues, it is a most powerful tool for making complex structures comprehensible. There are two varieties: *functional (procedural) abstraction* and *data abstraction.*

Almost all programming languages support *functional abstraction* by providing facilities for the definition of *subprograms.* The definition of a subprogram (procedure or function) associates a possibly complex sequence of instructions with an identifier and (usually) a list of parameters. The instruction sequence may then be performed by referring to its associated identifier and providing the required parameters. We could, for example, define a procedure *findRecord* that searches a given list of records for a record with a given key, e.g., an identification number. We could then use the statement

```
findRecord(theRecord, id_number, list);
```

instead of the instruction sequence that actually searches the list for the record. If this operation is needed more than once, then the obvious advantage is saving the replication of instructions. Much more important is a second advantage: When we study the text of a program that *uses findRecord*, we most likely wish to understand how this program processes and updates records or uses the information they contain. As this in itself may be

quite complex, we do not wish to be burdened simultaneously with the details of how to *find* a record, since this second problem has absolutely nothing to do with the process that we are trying to understand. Functional abstraction allows us to separate these different concerns and, thus, make each comprehensible.

The essential property of a data object is not its representation (is it an array, a record or something else?) but the operations that can be applied to it. The separation of these (abstract) operations from the subprograms that implement them, as well as from the representation of the object, is called *data abstraction*. This separation is possible because only the implementations of the operations need to know about the actual representation of the object. A system of data objects and operations defined by data abstraction is called an *abstract data type*. See Chapter 81.3 on *data types and data structures* for more details.

## Modularity

With the tool of abstraction programs are made modular, that is, they are broken into fairly small, manageable parts. Each part, called a module, consists of one or more procedures or functions and addresses a particular detail of a project. A module may consist of a sorting program, a package for matrix or time and date calculations, a team of programs that preprocess input or format output, and the like.

A program should be broken up into smaller parts for one of two reasons: (1) if it addresses more than one problem and (2) if it is physically too large.

First, a procedure (function) should address a single problem. If more than one problem is addressed by the same procedure, a reader of the program text experiences a distracting shift in subject, which makes the comprehension of the text unnecessarily difficult. For example, a program involved in interest calculations should not also solve calendar problems (such as how many days are between March 15 and June 27) but use a calendar module that provides the abstract data type *Date* to obtain the desired results. Similarly, a sorting procedure should not be burdened with the details of comparing records. Instead, a different function should be written that contains the details of the comparison, and this function should then be invoked by the sorting program. The program fragments in Example 1, written in the programming language C, illustrate the separation of these levels of concern.

Considering Example 2, the advantage of separating levels of concern may appear to be subtle. The reason is the small size of the original sort program. A quote by B. Stroustrup [1991] emphasizes this point, "You can make a small program (less than 1000 lines) work through brute force even when breaking every rule of good style. For a larger program, this is simply not so. . . ." Advantages that do not seem very significant for small programs become invaluable for large ones. Writing large programs is an inherently difficult activity that demands a high level of concentration, helped by separating and hindered by mixing unrelated levels of concern.

Second, as a rule of thumb, the text of a procedure should fit onto one page (or into a window). A single page of text can be studied exclusively by eye movement and, thus, comprehended more easily than text that spans over several pages. That is not to say that longer procedures are necessarily an indication of poor programming methods, but there should be compelling reasons if the limit of one page is to be exceeded.

## Simple Hierarchical Structuring

The relation among modules should be hierarchical, i.e., modules should be the vertices of a directed acyclic graph whose edges describe how modules refer to other modules. In such a structure, called a top-down design, each module can be understood in terms of the modules to which it refers directly. Yet, hierarchical structuring by itself does not yield simple and comprehensible programs. In addition, interfaces between modules must be kept narrow, that is, (1) the use of nonlocal variables must be avoided and (2) the number of parameters passed to a subprogram should be held to a minimum.

First, the scope rules of many programming languages permit subprograms to access entities of the program in which they are defined (static scoping) or of the calling program (dynamic scoping). These entities are called nonlocal in contrast to those defined within the subprogram itself. *Nonvariable* entities can and should be broadcast by means of nonlocal, preferably global definitions (i.e., definitions available to all modules and subprograms within an entire project). For these entities—types, procedures, and constant values—global definitions are very beneficial, since changes that may become necessary need to be made only at the place of

***Example 1.*** A program that addresses more than one problem:

```
void sort (recType *table, int size) /* recType is a structure with fields
                                        suitable for a personnel record */
{
   int i, j, result;
   recType temp;
  for (i = 0; i < size-1; i++)
    for (j = i+1; j<size; j++){
        if ((result = strcmp(table[i].last, table[j].last)) == 0)
           if ((result = strcmp(table[i].f irst, table[j].f irst)) == 0)
              if ((result = strcmp(table[i].midl, table[j].midl)) == 0)
                 if ((result = datecmp(table[i].birthdate, table[j].birthdate)) == 0)
                     result = addresscmp(table[i].address, table[j].address);
        if (result > 0){
             temp = table[i];
             table[i] = table[j];
             table[j] = temp;
         }
   }
}
```

***Example 2.*** Programs tackling one problem at a time:

```
BOOL isGreater (recType a, recType b);
void swap (recType *a, recType *b);

void sort (recType *table, int size)
{
   int i, j;
  for (i = 0; i < size-1; i++)
    for (j = i+1; j<size; j++)
        if (isGreater(table[i], table[j])) swap(&table[i], &table[j]);
}

BOOL isGreater (recType a, recType b)
{
   int result;
  if ((result = strcmp(a.last, b.last)) == 0)
     if ((result = strcmp(a.f irst, b.f irst)) == 0)
       if ((result = strcmp(a.midl, b.midl)) == 0)
          if ((result = datecmp(a.birthdate, b.birthdate)) == 0)
              result = addresscmp(a.address, b.address);
  return result > 0;
}

void swap (recType *a, recType *b)
{
   recType temp;
  temp = * a;
  *a = *b;
  *b = temp;
}
```

definition, not at every place of usage. Moreover, global definitions that replace numeric or alphanumeric constants by descriptive identifiers improve the readability of a program. Nonlocal *variables,* on the other hand, are to be avoided. For reasons of simplicity and clarity, the task performed by a subprogram should be determined *exclusively* by the subprogram itself and the values of its parameters. As a result, all information needed to understand why a subprogram may behave differently for different invocations is provided at each point of invocation. This rule, related to the concept of *referential transparency,* is clearly violated if a subprogram

uses nonlocal variables: in order to understand the behavior of the subprogram, information contained in the nonlocal variables must be consulted. These variables may appear nowhere near the point of invocation.

Similarly, because their return values are clearly identified, functions that do not change their parameters are preferred over those that do and over procedures. As functions are not always appropriate, a programmer choosing to use a procedure should aid the reader's comprehension by adopting some documented standard of consistently changing either the procedure's first or last parameter(s).

Second, the number of parameters should be kept to a minimum. If more than four or five parameters seem to be required, programmers should seriously consider packaging some of them into a record or an array. Of course, only values that in some logical sense belong together should be packaged, while parameters that serve different functions should be kept separate. For example, suppose the operation *updateRecord* is to find a personnel record with a given identification number in a file or a list and then change certain fields such as the job title, rate of pay, etc. A programmer may be tempted to define

```
updateRecord (Rectype myRecord);
```

yet the form

```
updateRecord (Idtype idNumber, Packtype attributes)
```

is better, since it suggests that the record with the key *idNumber* is the one to be updated by modifying the *attributes*.

The term *top-down design* (also called step-wise refinement) is frequently misunderstood as exclusively describing the *method* of design rather than the *structure* of the *finished product*. However, the creative process of designing software (or anything else) frequently proceeds in a rather erratic fashion, and there is no fault in this as long as the final product through analyses and reorganizations has a simple hierarchical structure. This is not to say that the process of design should not be guided by a top-down analysis; however, neither should a programmer's creativity be hampered by the straightjacket of a formalistic design principle, nor can a poorly designed program be defended with reference to the superior (top-down design) method used for its creation.

## Object-Oriented Programming

In languages that support object-oriented programming, *classes* (i.e., data types) of objects are defined by specifying (1) the variables that each object will own as *instance variables* and (2) operations, called *methods*, applicable to the objects of the class. As a difference in style, these methods are not invoked like functions or procedures, but are *sent* to an object as a *message*. The expression [*window moveTo* : *x* : *y*], for example, is a message in the programming language *Objective C*, a dialect of C. Here the object *window*, which may represent a window on the screen, is instructed to apply to itself the method *moveTo* using the parameters *x* and *y*.

New objects of a class are created—usually dynamically—by *factory methods* addressed to the *class* itself. These methods allocate the equivalent of a record whose fields are the instance variables of the object and return a reference to this record, which represents the new object. After its creation, an object can receive messages from other objects.

To data abstraction, object-oriented programming adds the concept of inheritance: From an existing class new (sub)classes can be derived by adding additional instance variables and/or methods. Each subclass inherits the instance variables and methods of its superclass. This encourages the use of existing code for new purposes.

With object-oriented programming, classes become the modules of a project. For the most part, the rules of hierarchical structuring apply, interfaces should be kept narrow, and nonlocal variables ought to be restricted to the instance variables of an object. There are, however, at least two situations, both occurring with modern graphical user interfaces, where the strict hierarchical structure is replaced by a structure of mutual referencing.

Objects—instances of classes—frequently represent visible objects on the screen such as windows, panels, menus, or parts thereof. It may now happen that actions (such as mouse clicks) applied to one object influence another object and vice versa. For example, the selection of an object, such as a button or a text field, may launch or modify an inspector panel that, in turn, allows the modification of the appearance or function of the button or the text field (mutual referencing).

In conventional programming, library routines are, with rare exceptions, at the bottom of the hierarchy, i.e., they are called by subprograms written by the user of the library (later referred to as the "user") but they do not call subprograms that the user has written.

With object-oriented systems, library programs may have user-written *delegates* whose methods are used by the library object. For example, a window—a library object—may send the message "windowWillClose" (written by the user) to its delegate in response to the operators clicking of the window's close button. In response, the delegate may now interrogate the window in order to determine whether its contents have been changed and possibly require saving (mutual referencing).

Furthermore, buttons, menus, and the like are library objects. Yet, when operated, they usually activate user methods. Again, the user program may then interrogate the button about its state or request that the button highlight itself, etc. (mutual referencing).

While in the cases discussed, mutual referencing seems to be natural and appropriate, it should, in general, be used very sparingly.

## Program Testing

Aside from keeping the structure of a program as simple as possible, a top-down design can be tested in a divide-and-conquer fashion. There are basically two possible strategies: bottom-up testing and top-down testing.

Bottom-up testing proceeds by first testing all those modules that do not depend on other modules (except library modules). One then proceeds always testing those modules that use only modules already tested earlier. Testing of each (sub)program must exercise all statements of the program, and it must ensure that the program handles all exceptional responses from supporting subprograms correctly. For each module a driver program must be written that invokes the module's functions and procedures in an appropriate way. If a program is developed in a top-down fashion, this method cannot be used until a substantial part of the project is completed. Thus design flaws may not be detected until corrections have become difficult and costly.

Top-down testing begins with the modules that are not used by any other module. Since the supporting modules have not been tested yet (or do not even exist yet), simple stand-in programs must be written that simulate the actual programs. This can be done by replacing the computation of the actual program with the programmer, who enters the required results from the keyboard in response to the parameter values displayed on the screen. This method is nontrivial, especially if the supporting program deals with complex objects.

In practice both methods are being used. Frequently, developing the procedure for top-down testing by itself leads to the discovery of errors and can prevent major design flaws.

## Defining Terms

**Program correctness:** A program's conformation with its specifications.
**Program maintenance:** Modifications and (late) corrections of programs already released for use.
**Program portability:** A program is called (easily) portable if it can be adapted to a different computer system without much change.

## Related Topic

87.3 Data Types and Data Structures

## References

B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1988.
J.J. Martin, *Data Types and Data Structures*, C.A.R. Hoare, Series Ed., New York: Prentice-Hall, 1986.
NeXT Step Concepts, *Objective C*, NeXT Developers' Library, NeXT, Inc., 1991, chap 3.
J.W. Rozenblit, *Codesign: Software/Hardware Engineering*, IEEE Press, 1995.
J.C. Reynolds, *The Craft of Programmings*, C.A.R. Hoare, Series Ed., New York: Prentice-Hall, 1983.

Proceedings of 1996 International Conference on Software Maintenance, Los Alamitos, Calif.: IEEE Computer Society, 1996.
B. Stroustrup, *The C++ Programming Language*, Reading, Mass.: Addison-Wesley, 1991.
J. Welsh et al., *Sequential Program Structures*, C.A.R. Hoare, Series Ed., New York: Prentice-Hall, 1984.
N. Wirth, *Algorithms + Data Structures = Programs,* Berlin: Springer-Verlag, 1984.

## Further Information

Textbooks on programming usually address the subject of good programming style in great detail. More information can be found in articles on object-oriented programming and design and on software engineering as published, for example, in the proceedings of the annual conferences on *Object Oriented Programming Systems, Languages and Applications (OOPSLA)* sponsored by the Association for Computing Machinery (ACM) and on *Computer Software and Applications (CompSac)* sponsored by the Institute of Electrical and Electronics Engineers, Inc. (IEEE). Articles on the subject are also found in periodicals such as *IEEE Transactions on Software Engineering*, *IEEE Transactions on Computers*, *ACM Transactions on Software Engineering and Methodology, Software Engineering Notes* published by the ACM Special Interest Group on Software Engineering, *ACM Transactions on Programming Languages and Systems*, the *Communications of the ACM*, or *Acta Informatica.*