6

SOFTWARE DESIGN APPROACHES

Software design is a salient part of the entire software development process, which can be an individual subprocess of the high-level product development process. Furthermore, in embedded applications, the product development process may include concurrent hardware and subsystem development subprocesses, too. Software designers translate the problem-domain requirements document discussed in the previous chapter into physical models of the solution that are sufficient for straightforward implementation or programming. The resultant design document should be such that even an external programming consultant could implement the code with minimal interaction with the design team. Completeness of the design document is particularly important in globally distributed projects, where the requirements document might be created in the United States, the design document in Finland, and the implementation in India, for instance. Integrated CASE environments, which provide smooth transitions from the requirements engineering phase to the design phase and further to the implementation phase, should be used throughout the software development process. Besides, it is essential to use standardized/widespread modeling techniques, such as the SA/SD methods or the UML, to make the core documentation understandable for various collaborating teams.

The Institute of Electrical and Electronics Engineers (IEEE) Standard Dictionary of Electrical and Electronics Terms (IEEE Std 100–2000) describes the term "design" as follows: The process of defining the architecture, components, interfaces, and other characteristics of a system. Hence, during the software design phase, numerous decisions are made concerning responsibility assignment and fulfillment, system architecture and deployment, separation of concerns, as well as layering and modularization (Bernstein and Yuhas, 2005). Moreover, the computational algorithms and their numerical precision are specified in the design document; such important decisions are often supported by simulations or prototyping. The opportunity of design reuse also should be carefully considered. In our experience with typical real-time applications, the software design phase takes roughly the same amount of resources (in person months) as the requirements-engineering and programming phases together.

The desired qualities of real-time software as well as advantageous software engineering principles will be discussed in Sections 6.1 and 6.2, respectively. In addition, a mapping from these principles to qualities is sketched with a pragmatic discussion. As the procedural and object-oriented approaches exist in requirements engineering and programming phases, they both are naturally available in the design phase, as well. Therefore, we discuss the procedural design approach in Section 6.3 and the alternative object-oriented approach in Section 6.4. Both of these design sections are composed on real-world examples. Section 6.5 gives an evaluative overview on a sample of life cycle models that are currently used for the development of real-time software. The preceding sections on software design approaches are summarized in Section 6.6 with some suggestions. A carefully selected collection of stimulating exercises is provided in Section 6.7. Lastly, Section 6.8 contains a comprehensive case study on designing real-time software (the corresponding requirements document of this traffic-light control system is available in Section 5.7).

Some parts of this chapter have been adapted from Laplante (2003).

6.1 QUALITIES OF REAL-TIME SOFTWARE

Software systems and individual components can be characterized by a number of diverse qualities. *External* qualities are those that are observable by the user, such as performance and usability, and are of explicit interest to the end user. *Internal* qualities, on the other hand, are not observable by the user, but aid the software developers to achieve certain improvement in external qualities. For example, although the requirements and design documentation might never be seen by a typical user, their adequate quality is essential in achieving satisfactory external qualities. Such an external–internal distinction is a function of the software itself and the type of user involved.

While it is beneficial to know the software qualities and the motivations behind them, it is equally desirable to measure them objectively. Measuring of these characteristics of software is necessary in enabling end users and designers to talk succinctly about the product, and for effective software process control and project management. More importantly, however, it is these qualities that shall be embodied in the real-time design.

6.1.1 Eight Qualities from Reliability to Verifiability

Reliability is a measure of whether a user can depend on the software (Teng and Pham, 2006). This quality can be informally defined in a number of ways. For instance, one definition might be simply "a system that a user can depend on." Other common characterizations of a reliable software system include:

- The system "stands the test of time."
- There is an absence of known errors that render the system useless.
- The system recovers "gracefully" from errors.
- The software is robust.

In particular, for real-time systems, other informal characterizations of reliability might include:

- Downtime is below a specified threshold.
- The accuracy of the system remains within a certain tolerance.
- Real-time performance requirements are met consistently.

While all of these informal characteristics are certainly desirable in real-time systems, they are difficult to measure or predict. Moreover, they are not true measures of reliability, but of various attributes of the software instead.

There is specialized literature on software reliability that defines this quality in terms of statistical behavior, that is, the probability that the software product will operate as expected over a specified time interval (Pham, 2000). These characterizations generally take the following approach. Let S be a software system, and let T be the time instant of system failure. Then the reliability of S at time t, denoted $r_s(t)$, or when there can be no confusion with other systems, $r(t)$, is the probability that T is greater than t; that is,

$$r(t) = P(T > t). \tag{6.1}$$

This is the probability that a software system will operate without failure for a specified period of time. In addition to the actual operating phase, also the testing phase may be included in the considered period.

A system with reliability function $r(t) = 1$ would never fail. However, it is unrealistic to have such an expectation with any real-world system. Instead, some reasonable goal, $r(t) < 1$, should be specified.

Example: Failure Probability Increases as a Function of Time

Consider the monitoring system of a nuclear power plant with the specified failure probability of no more than 10^{-9} per hour. This represents a reliability function of $r(t) = (0.999999999)^t$, where t is in hours. Note that as $t \to \infty$, $r(t) \to 0$. To illustrate, the failure probability, $q(t) = 1 - r(t)$, for various values of t is given in Table 6.1. Moreover, after 35 years of operation (306,600 hours)—still a reasonable time for nuclear power plants—the failure probability is approximately 0.0003.

Another way to characterize software reliability is in terms of a failure function or model. One failure function uses an exponential distribution where the abscissa is time and the ordinate represents the expected failure intensity at that time:

$$f(t) = \lambda / e^{\lambda t}, \quad t \geq 0. \tag{6.2}$$

Here the failure intensity is initially high, as would be expected in new software, since failures are detected more frequently during the testing phase. However, the number of failures would be expected to decrease with time during the operating phase, presumably as failures are uncovered and repaired (see Fig. 6.1). The factor λ is a system-dependent parameter that must be determined empirically.

TABLE 6.1. Failure probability as a Function of Operating Hours

t	10^0	10^1	10^2	10^3	10^4	10^5	10^6
$q(t)$	10^{-9}	$\approx 10^{-8}$	$\approx 10^{-7}$	$\approx 10^{-6}$	$\approx 10^{-5}$	$\approx 10^{-4}$	$\approx 10^{-3}$

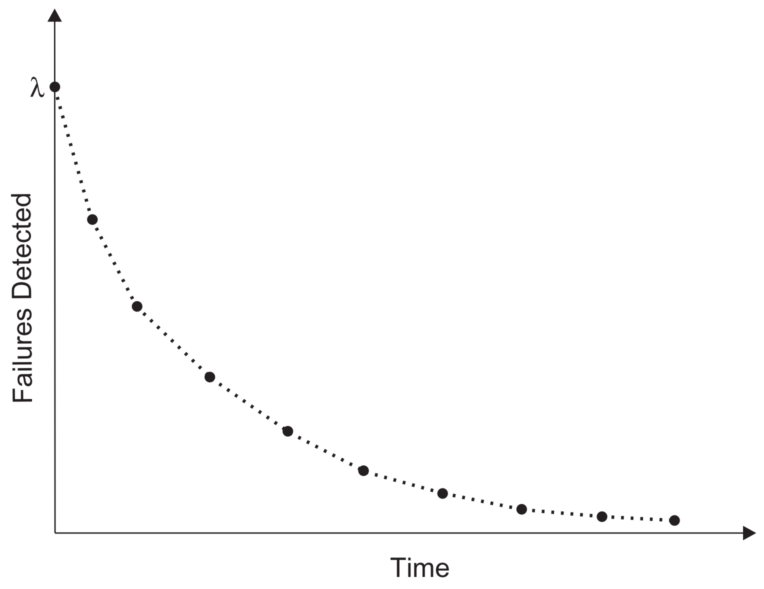

Figure 6.1. A model of failure represented by the exponential failure function (Laplante, 2003).

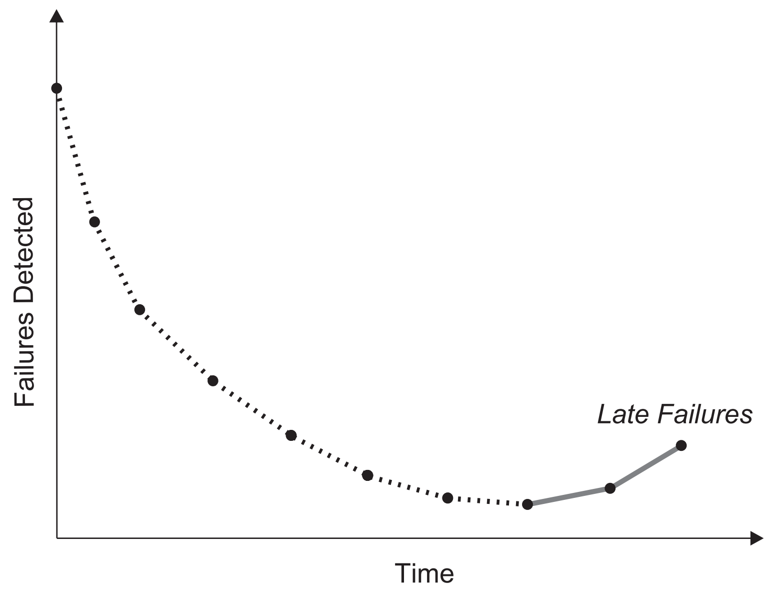

Figure 6.2. A software failure function represented by the bathtub curve (Laplante, 2003).

Another common failure model is given by the "bathtub curve" shown in Figure 6.2. Brooks notes that while this curve is widely used to describe the failure function of hardware and mechanical components, it might also be useful in describing the number of errors found in a software product (Brooks, 1995). This is particularly valid with embedded systems that have a long lifetime (even 10–30 years), and the software is updated (repaired/enhanced) numerous times during the lengthy period.

The interpretation of this failure function is apparent for hardware and mechanics: a certain number of product units will fail early due to manufacturing defects. Later, the failure intensity will increase again as the hardware/mechanics ages and wears out. But software does not wear out. Therefore, if software systems really seem to fail according to the bathtub curve, then there has to be some plausible explanation.

It is understandable that the largest number of errors will be found early in a software product's life cycle, just as the exponential failure model indicates. But why would the failure intensity increase much later? There are at least three possible explanations:

1. The failures are due to the effects of patching the software (making quick corrections to the code without designing them properly) for various reasons.
2. Late software failures are actually due to wearing of the underlying hardware or possible sensors/actuators.
3. As users master the basic software functions and begin to expose and strain advanced features, it is possible that certain inadequately tested functionality is eventually beginning to be used.

Empirical failure models are used commonly to make rough predictions of software failures during the entire operating phase. As the operating environments for the software may vary drastically in embedded applications, the randomness of a practical environment will affect the failure rate in an unpredictable way (Teng and Pham, 2006). Hence, the λ factor of Equation 6.2 should be a random variable.

Often, the traditional quality measures of mean time to first failure (MTFF) or mean time between failures (MTBF) are used to stipulate reliability in the software requirements specification. This approach to failure definition places great importance on the effective elicitation and specification of functional requirements, because the requirements also define the possible software failures.

Furthermore, real-time software execution is very sensitive to initial conditions and the external data driving it. What appear to be random failures are actually repeatable. The problem in finding and fixing these problems before a design is released, or even if the problem emerges once the embedded software is in use, is the difficulty of doing the detective work needed to discover first the particular conditions and second the data sequences that triggered the fault to become a failure. The longer a software system runs, the more likely it becomes that such a fault will be executed.

Correctness of software (Mills, 1992) is closely related to software reliability, and the terms may sometimes be used interchangeably. The fundamental difference is that even a minor deviation from the requirements is strictly considered a failure and hence means the software is incorrect. However, a system may still be deemed reliable if only minor deviations from the requirements are experienced. As widely known, such minor deviations are rather common in many software products, because typical software can only be tested partially, and often just a small proportion of the actual input space is explored statistically. In real-time systems, correctness incorporates both correctness of outputs, as well as deadline satisfaction, as discussed in Chapter 1.

Performance of software (Caprihan, 2006) is an explicit measure of some required behavior. A general methodology for measuring algorithmic performance is based on computational complexity theory (Goldreich, 2008). Alternatively, a simulation model of the real-time system might be built with the actual purpose of estimating performance. The most accurate approach, though, involves directly timing the behavior of the completed system with a logic analyzer or specific performance analysis tools.

Usability, which is often referred to as ease of use or user friendliness, is a measure of how easy and comfortable the software is for humans to use (Nielsen, 1993). This software quality is an elusive one. Properties that make an application user friendly to novice users are often very different from those desired by expert users or the software designers themselves. Demonstrative prototyping can increase the usability of a software system because, for instance, user interfaces can be evaluated and fine-tuned by a group of end users of the final product.

Usability is often difficult to quantify, although it may be easy to argue that some system is not usable. However, qualitative feedback from users and individual problem reports can be used in most cases for evaluating usability. Such general issues as user training time and readability of user documentation are possible measures of usability (Bernstein and Yuhas, 2005).

Interoperability refers to the ability of the software to coexist and cooperate with other relevant software. It is especially important in component-based software development, software reuse, and network-based software systems (Wileden and Kaplan, 1999). For example, in real-time applications, the software must be able to communicate with various devices using standard bus structures and protocols. Interoperability is usually straightforward to achieve if the decision to communicate is made before the software is designed—it is much more laborious to attain afterwards.

A concept related to interoperability is that of an open system (Dargan, 2005). An open system is an extensible collection of independently written applications that cooperate to function as an integrated system. Open systems differ from open source code, which is source code that is made available to the global user community for evolutionary improvement, extension, and correction, provided that the terms of the associated license are honored. An open system allows the addition of new functionality by independent parties through the use of standard interfaces whose detailed characteristics are published. Any applications developer can then take advantage of these interfaces, and thereby create software that can communicate using the interface. Open systems let different applications written by different organizations interoperate. For example, there are open standards for automotive (AUTOSAR, Automotive Open System Architecture), building automation (BAS, Building Automation System), and railway vehicle (IEEE Std 1473-L) systems. Interoperability can be measured in terms of compliance with relevant open system standards.

Maintainability is related to the anticipation of change that should guide the software engineer throughout the development project. A software system in which changes are relatively easy to make has a high level of maintainability; this is connected directly to the readability and understandability of the program code and associated documentation (Aggarwal et al., 2002). In the long run, design for change will significantly lower software life cycle costs and lead to an enhanced reputation for the software engineer, the software product, and the corresponding organization. Some embedded software products are maintained even for a few decades, and, therefore, the issue of maintainability is of particular importance in such cases.

Maintainability can be broken down into two contributing properties: evolvability and repairability. Evolvability is a measure of how easily the system can be changed to accommodate new features or modification of existing features. Furthermore, software is repairable if it allows for the fixing of all defects with a reasonable effort.

Measuring these qualities of software is not always easy or even possible, and often is based on anecdotal observation only. This means that changes and the cost of making them should be tracked over time. Collecting such history data has a twofold purpose. First, the costs of maintenance can be compared with other similar systems for benchmarking and project management purposes. Second, the information can provide experiential learning that will help to improve the overall software development process, as well as the skills of software engineers.

Portability of software is a measure how easily the software can be made to run in different environments. Here, the term "environment" refers to the hardware platform on which the software runs, the real-time operating system used, or other system/application software with which the particular software is expected to interact. Because of the I/O-intensive hardware with which the software closely interacts, special care must be taken in making embedded software portable.

Hardware portability is achieved through a deliberate design strategy in which hardware-dependent code is confined to the fewest code units as possible (such as device drivers). This strategy can be achieved using either procedural or object-oriented programming languages and through structured or object-oriented design approaches. Both of these are discussed throughout the text.

On the other hand, portability of real-time operating systems or other system programs means usually the adoption of some standard application program interface (API) (Shinjo and Pu, 2005). This is commonly associated with potential overhead caused by the standards-prescribed interface. In this sense, portability may degrade the achievable real-time performance.

Also, portability is difficult to measure, other than through anecdotal observation. Person-months required to move the software to a new environment is a usual measure of this property. But this cannot be known before the actual moving effort.

Verifiability of software qualities refers to the degree to which various qualities, including all of those previously introduced, can be verified. In real-time systems, verifiability of deadline satisfaction (a form of performance) is of the utmost importance. This topic is discussed further in Chapter 7.

One common technique for increasing verifiability is through the insertion of special program code that is intended to monitor certain qualities, such as performance or correctness. Rigorous software engineering practices and the effective use of an appropriate programming language can also contribute to verifiability.

Measurement or prediction of software qualities is essential throughout the whole software life cycle. Therefore, this activity should be integrated seamlessly into the software development process. A summary of the software qualities just discussed and possible ways to measure them is given in Table 6.2.

Today, in the "embedded systems era," the emphasis on desirable software qualities has shifted gradually from *correctness* to *reliability* and *maintainability* (Aggarwal et al., 2002). A further emphasis is on the need to increase the

TABLE 6.2. Software Qualities and Possible Means for Measuring Them

Software Quality	Possible Measurement Approach
Reliability	Probabilistic measures, MTFF, MTBF, heuristic measures
Correctness	Probabilistic measures, MTFF, MTBF
Performance	Algorithmic complexity analysis, simulation, direct measurement
Usability	User feedback from surveys and problem reports
Interoperability	Compliance with relevant open standards
Maintainability	Anecdotal observation of resources spent
Portability	Anecdotal observation of resources spent
Verifiability	Insertion of special monitoring code

productivity of software developers, due to the growing complexity of software products and need for shorter time-to-market. The object-oriented design approach, to be discussed in Section 6.4, may help address this productivity challenge (Siok and Tian, 2008).

6.2 SOFTWARE ENGINEERING PRINCIPLES

Software engineering has been criticized for not having the same kind of theoretical foundation as older engineering disciplines, such as electrical, mechanical, or civil engineering. While it is true that only a few formulaic principles exist, there are several fundamental rules that form the basis of sound software engineering practice. The following subsection describes the most general and prevalent principles that are particularly applicable in the design and implementation phases of real-time software.

6.2.1 Seven Principles from Rigor and Formality to Traceability

Because software development is a creative human activity related to problem solving, there is an inherent tendency toward using informal *ad hoc* techniques in software specification, design, and coding. Nevertheless, a purely informal approach is contrary to "best software engineering practices." It should be pointed out, however, that the best practices are actually dependent on the application size as well as application type (Jones, 2010), and also on the size of the development organization (Jantunen, 2010).

Rigor in software engineering requires the use of mathematical techniques. *Formality*, on the other hand, is a higher form of rigor in which precise and unambiguous engineering approaches are used. In the case of real-time systems, strict formality would further require that there be an underlying algorithmic approach to the specification, design, coding, and documentation of the software. Due to insuperable difficulties in creating a pure algorithmic approach, semiformal and informal approaches are needed to complement individual formal approaches. For instance, certain parts of the design document can be formal while most others are semiformal.

Separation of concerns is an effective divide-and-conquer strategy practiced by software engineers to manage miscellaneous problems related to complexity. There are various ways in which separation of concerns can be achieved. In terms of software design and coding, it is used in object-oriented design and in modularization of procedural code. Moreover, there may be separation in time, for example, developing an appropriate schedule for a collection of periodic computing tasks with different execution periods.

Yet another way of separating concerns is in dealing with individual software qualities. For instance, it may be helpful to address the fault tolerance of a system only while ignoring other qualities for some time. However, it must be remembered that many of the software qualities are actually interrelated, and it is often impossible to improve one without deteriorating another. Hence, a project-specific compromise is typically needed.

Modularity is commonly achieved by grouping together logically related elements, such as statements, procedures, variable declarations, and object attributes, in an increasingly fine-grained level of detail (see Fig. 6.3). Modular design involves the decomposition of software behavior in encapsulated software units, and can be achieved with both procedural and object-oriented

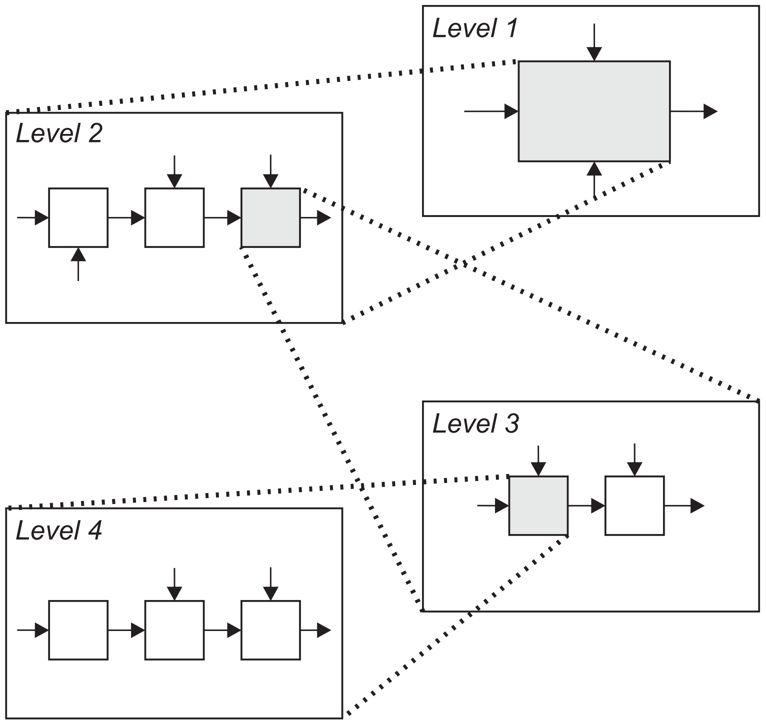

Figure 6.3. Modular decomposition of code units. The arrows represent inputs and outputs in the procedural paradigm. In the object-oriented paradigm, they represent associations. The boxes represent encapsulated data and procedures in the procedural paradigm. In the object-oriented paradigm, they represent classes (Laplante, 2003).

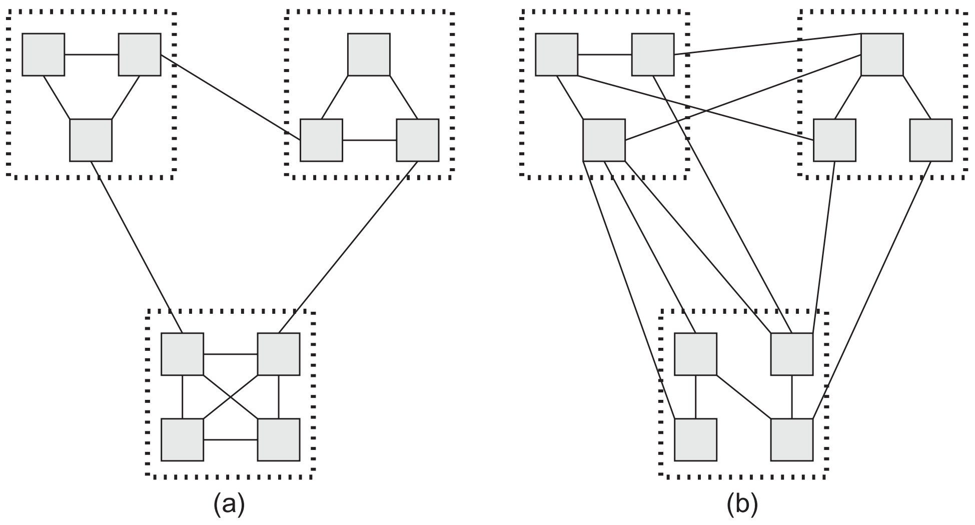

Figure 6.4. Software structures with (a) high cohesion and low coupling, and (b) low cohesion and high coupling. The inside squares represent statements or data; connecting lines indicate functional dependency.

programming languages. The main goal of modularity is high cohesion and low coupling of the software structure. With respect to the code units, cohesion represents intramodule connectivity and coupling represents intermodule connectivity. Cohesion and coupling can be illustrated as in Figure 6.4, which depicts software structures with high cohesion and low coupling (left), as well as low cohesion and high coupling (right). Cohesion relates to the relationship of the elements within a module. High cohesion implies that each module represents a single part of the problem solution. Therefore, if the system ever needs modification, then the part that needs to be modified exists in a single place, being easier and less error prone to change.

Constantine and Yourdon identified seven levels of cohesion in the order of increasing strength (Pressman, 2009):

1. ***Coincidental.*** Parts of a module are not related at all, but simply bundled into a single module.
2. ***Logical.*** Parts that perform similar tasks are put together in a joint module.
3. ***Temporal.*** Tasks that execute within the same time span are brought together.
4. ***Procedural.*** The elements of a module make up a single control sequence.
5. ***Communicational.*** All elements of a module act on the same area of a data structure.
6. ***Sequential.*** The output of one part in a module serves as input for another part.
7. ***Functional.*** Each part of a module is necessary for the execution of a single function.

This above list could be used when designing the contents of specific software modules; it brings valuable insight to the heuristic module-creation process. Modules should not be created solely by "grouping together logically related elements"—as is usually done. But there are multiple reasons to group individual elements together.

Coupling relates to the relationships between the modules themselves. There is a great benefit in reducing coupling so that changes made to one code unit do not propagate to others; they are said to be hidden. This principle of "information hiding," also known as Parnas partitioning, is the cornerstone of all software design and will be discussed in Section 6.3.1 (Parnas, 1979). Low coupling limits the effects of errors in a specific module (lower "ripple effect") and reduces the likelihood of data-integrity problems. In some cases, however, high coupling due to time-critical control structures may be necessary. For example, in most graphical user interfaces, control coupling is unavoidable, and indeed desirable.

Coupling has been characterized by six levels in the order of increasing strength:

1. ***None.*** All modules are completely unrelated.
2. ***Data.*** Every argument is either a simple argument or data structure in which all elements are used by the called module.
3. ***Stamp.*** When a data structure is passed from one module to another but that module operates on only some of the data elements of the whole structure.
4. ***Control.*** One module explicitly controls the logic of the other by passing an element of control to it.
5. ***Common.*** If two modules both have access to the same global data.
6. ***Content.*** One module directly references the contents of another.

To further illustrate both coupling and cohesion, consider the class structure diagram (object-oriented design approach) shown in Figure 6.5; the figure illustrates two interesting points. The first is the clear difference between the same system embodying low coupling and high cohesion versus high coupling and low cohesion. The second point is that the proper use of visual design techniques can positively influence the eventual design outcome.

Anticipation of change is another important principle in software design. As has been mentioned, software products are subject to frequent change either to support new hardware or software requirements or to repair defects. A high maintainability level of the software product is one of the hallmarks of outstanding commercial software.

Developers of embedded software know that their systems are subject to changes in hardware, algorithms, and even application. Therefore, these systems must be designed in such a way as to facilitate changes without degrading considerably the other desirable properties of the software. Anticipation of

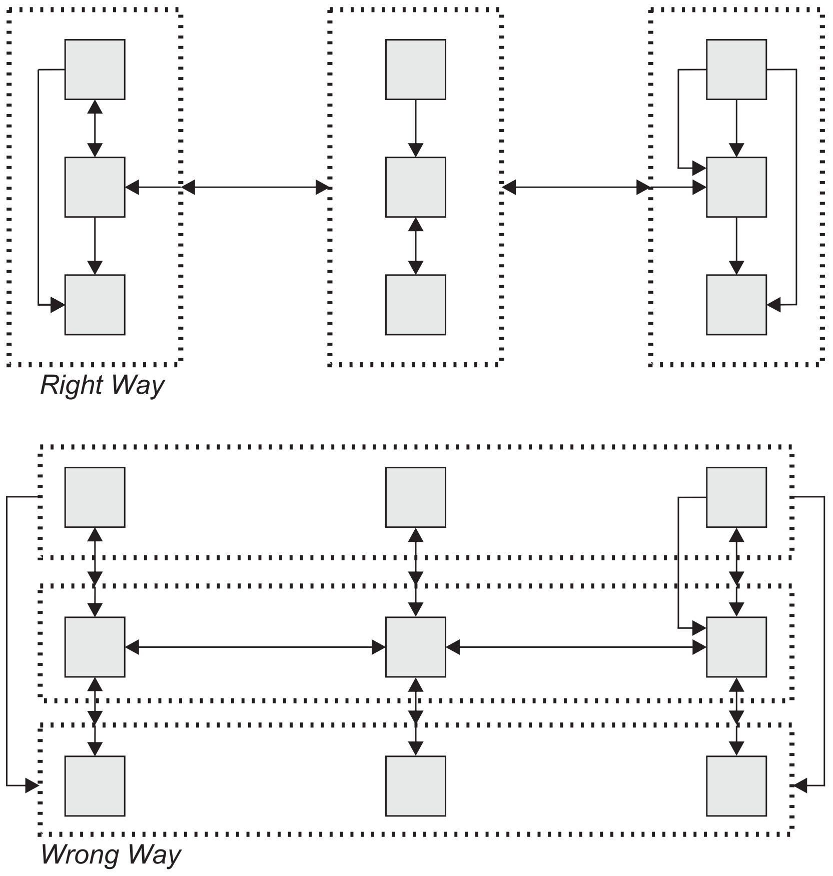

Figure 6.5. Coupling and cohesion. The right way: low coupling and high cohesion. The wrong way: high coupling and low cohesion.

change can be achieved in the software design through the adoption of an appropriate software life cycle model and corresponding design methodologies, as well as through appropriate project-management practices and associated training efforts.

In solving a problem, the principle of *generality* can be stated as the intent to look for a more general problem that may be hidden behind it. As an obvious example, designing an elevator control system for a low-end apartment building is less general than designing it to be adaptable to various hotels, offices, shopping centers, and apartment buildings.

Generality can be achieved through a diverse number of approaches associated with procedural and object-oriented paradigms. For example, Parnas' information hiding can be used with procedural languages. Extensive parameterization is another commonly used approach for providing generality to software. In object-oriented software, generalization is achieved by applying certain design principles and through the use of architectural and design patterns. Although generalized solutions may be more costly in terms of the problem at hand, in the long run, the extra costs of a generalized solution may be worthwhile. Nonetheless, these extra costs might affect real-time performance, which is always a difficult issue to handle. Moreover, a manager of a specific development project might ask a relevant question: "Why should *this* project pay some costs of possible future projects in advance?" This is, indeed, a good question and should be addressed by the steering group of that particular project; there may be a conflict between short-term and long-term goals.

Incrementality involves a software-engineering approach in which progressively larger increments of the desired product are developed. Each increment provides additional functionality, which brings the unfinished product closer to the final one. Each increment also offers an opportunity for demonstration of the product to the customer for the purposes of gathering supplementary requirements and refining the look and feel of the product or its user interface, for example. In reality, however, some advanced sets of increments have even been delivered to the customer as "the product" due to sizeable delays in the development project. This usually leads to serious problems and shall be strictly avoided.

Traceability is concerned with the relationships between requirements, their sources, and the system design. Regardless of the life cycle model used, documentation and code traceability are truly important. A high level of traceability ensures that the software requirements flow down through the design and program code, and then can be traced back up at every stage of the development process. This would ensure, for instance, that a coding decision can be traced back to a design decision to satisfy a corresponding requirement.

Traceability is particularly important in embedded systems, because specific design and coding decisions are often made to satisfy rather unique hardware constraints that may not be directly associated with any higher-level requirement. Failure to provide a traceable path from such decisions through the requirements can lead to substantial difficulties in extending and maintaining the system.

Generally, traceability can be obtained by providing consistent links between all documentation and the software code. In particular, there should be links:

- From requirements to stakeholders who proposed these requirements.
- Between dependent requirements.
- From the requirements to the design.
- From the design to associated code segments.
- From requirements to the test plan.
- From the test plan to individual test cases.

One way to create these links is through the use of an appropriate numbering system throughout the documentation. For instance, a requirement numbered 3.1.1.2 would be linked to a design element with a similar number (the numbers do not have to be the same as long as the annotation in the document guarantees traceability). In practice, a traceability matrix is constructed to help cross reference the documentation and associated code elements (Table 6.3).

TABLE 6.3. A Traceability Matrix Sorted by Requirement Number

Requirement Number	Design Document Reference Number(s)	Test Plan Reference Number(s)	Code Unit Name(s)	Test Case Number(s)
3.1.1.1	3.1.1	3.1.1.1	Task_A	3.1.1.A
	3.2.4	3.2.4.1		3.1.1.B
		3.2.4.3		3.1.1.C
3.1.1.2	3.1.1	3.1.1.2	Task_B	3.1.1.A
				3.1.1.D
3.1.1.3	3.1.1.3	3.1.1.3	Task_C	3.1.1.B
				3.1.1.E

The matrix is constructed by listing the relevant software documents and the code units as columns, and then each software requirement in the rows. Traceability to the stakeholders related to certain requirements or to relevant standards and regulations could also be added as columns in Table 6.3.

Constructing the traceability matrix in a spreadsheet software package allows for providing multiple matrices sorted and cross-referenced by each column as needed. For example, a matrix sorted by test case numbers would be an appropriate appendix to the test plan. The traceability matrices are updated at each step in the software life cycle. For instance, the column for the code unit names (e.g., procedure names or object classes) would not be added until after the code is developed. A way to foster traceability between code units is through the use of data dictionaries, which are described later.

Finally, a mapping (*positive effect*) from the individual software-engineering principles, just discussed, to the desired software qualities of Table 6.2 is sketched in Table 6.4. Some of these mappings are explicit, while others are more implicit. Interestingly, the software quality of maintainability appears to be improvable by all the seven principles. Of the software engineering principles, modularity seems to be a particularly strong one, since it can improve all the software qualities except "usability" and "verifiability."

6.2.2 The Design Activity

The design activity is involved in identifying the components of the software design and their interfaces from the software requirements specification. The principal artifact of this activity is the Software Design Description (SDD). In the same way as the IEEE Std 830–1998 (discussed in Section 5.1.2) provides a sound framework for requirements engineering documents, a recently revised standard, IEEE Std 1016–2009, specifies requirements on the information content and organization for software design descriptions (IEEE, 2009). According to the standard, "SDD is a representation of a software design that is to be used for recording design information, addressing various design concerns, and communicating that information to the design's stakeholders."

TABLE 6.4. A Mapping Matrix from Software Engineering Principles to Software Qualities in Real-Time Applications

Principles/Qualities	Reliability	Correctness	Performance	Usability	Interoperability	Maintainability	Portability	Verifiability
Rigor and Formality	×	×	×			×	×	×
Separation of Concerns	×	×	×	×	×	×	×	×
Modularity	×	×	×		×	×	×	
Anticipation of Change					×	×	×	
Generality					×	×	×	
Incrementality	×	×		×		×		
Traceability	×	×				×	×	

During the design phase, a team of real-time systems engineers creates a detailed software design and acquires a formal acceptance for it. That involves the following tasks from the initial Architecture Design (Taylor et al., 2010) to the Final Design Review (Hadar and Hadar, 2007):

1. *Architecture Design*
   - Performing hardware/software trade-off analysis leading to hardware–software partitioning.
   - Making the determination between centralized or distributed processing schemes.
   - Designing interfaces to external components.
   - Designing interfaces between internal components.
2. *Control Design*
   - Determining concurrency of execution.
   - Designing principal control strategies.
3. *Data Design*
   - Determining storage, maintenance, and allocation strategy for data.
   - Designing database structures and handling routines.
4. *Functional Design*
   - Designing the start-up and shutdown processing.
   - Designing algorithms and functional processing.
   - Designing error processing and error-message handling.
   - Conducting performance analyses of critical functions.
5. *Physical Design*
   - Determining physical locations of software components and data.
6. *Test Design*
   - Designing any test software identified in test planning.
7. *Documentation Design*
   - Creating possible support documentation, such as the Operator's Manual, User's Manual, Programmer's Manual, and Application Notes.
8. *Intermediate Design Reviews* (→ internal acceptances)
   - Conducting internal design reviews.
9. *Detailed Design*
   - Developing the detailed design for all software components.
   - Developing the test cases and procedures to be used in the formal acceptance testing.
10. *Final Design Review* (→ organizational acceptance)
    - Documenting the software design in the form of the SDD.
    - Presenting the SDD at a formal design review for examination and criticism.

This is an intimidating set of substantial tasks that is further complicated by the fact that many of them must occur in parallel or be iterated several times. There is obviously no algorithm, per se, for conducting these tasks. Instead, it takes many years of practicing, learning from the experience of others, and good judgment to guide the software engineer heuristically through this maze of individual design tasks. In such effort, collective knowledge of a matured development organization would be of significant aid.

Two alternative methodologies, procedural and object-oriented design, which are related to structured analysis and object-oriented analysis, respectively, can be used to perform the design activities based on the software requirements specification. Both methodologies seek to arrive at a physical software model containing small, detailed components.

6.3 PROCEDURAL DESIGN APPROACH

Procedural design methodologies, like structured design, involve top-down and bottom-up approaches centered on procedural programming languages, such as the popular C language. The most common of these approaches utilize effective design decomposition via Parnas partitioning (Parnas, 1979).

6.3.1 Parnas Partitioning

Software partitioning into multiple software units with low external coupling and high internal cohesion can be achieved through the principle of *information hiding*. In this technique, a list of difficult design decisions or things that are likely to change is first prepared. Individual modules are then designated to hide the eventual implementation of each design decision or a specific feature from the rest of the system. Thus, only the functionality of each module is visible to other modules, not the method of implementation. Changes in these modules are therefore not likely to affect the rest of the system.

This form of functional decomposition is based on the notion that some aspects of a system are fundamental and remain constant, whereas others are somewhat arbitrary and likely to change. Moreover, it is those arbitrary aspects that often contain the most valuable design information. Arbitrary facts are hard to remember and usually require lengthy descriptions; hence, they are typical sources of documentation complexity.

The following five steps can be used to implement a good design that embodies information hiding:

1. Begin by characterizing the likely changes (consider different time horizons of the life cycle) and their effects.
2. Estimate the probabilities of each type of change.
3. Organize the software to confine likely and significant changes to a minimum amount of code.

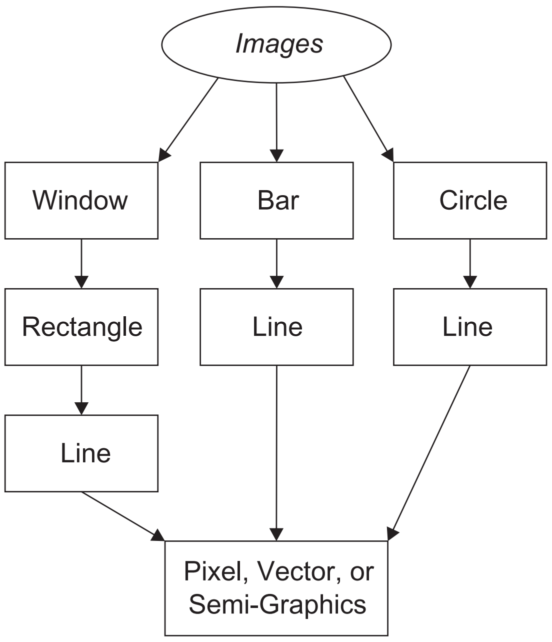

Figure 6.6. Parnas partitioning of graphics rendering software.

4. Provide an "abstract interface" that abstracts from the potential differences.
5. Implement "objects," that is, abstract data types and modules that hide changeable data and other structures.

These steps reduce intermodule coupling and increase intramodule cohesion. Parnas also indicated that although module design is easy to describe in textbooks, it is difficult to achieve in practice. He suggested that extensive real-world examples are needed to illustrate the point correctly (Parnas, 1979).

As an example, consider a portion of the display function of a graphics subsystem associated with an elevator monitoring system and depicted in hierarchical form in Figure 6.6. Such monitoring systems are used in supervision centers and can also be available in large lobbies for displaying the elevator traffic. It consists of color graphics that must be displayed (e.g., a representation of multiple elevator shafts, animated elevator cars, and registered calls) and are essentially composed from bars, rectangles, and circles. Different objects can naturally reside in different display windows. The actual implementation of bars, rectangles, and circles is based on the composition of line-drawing calls. Thus, line drawing is the most basic (hardware-dependent) function in this application. Whether the actual graphics controller is based on pixel, vector, or even semi-graphics does not matter; only the line-drawing routine with standard software interfaces needs to be changed. Hence, the hardware dependencies have been isolated to a single code unit.

Parnas partitioning hides the implementation details of software features, design decisions, low-level hardware drivers, and so forth, in order to limit the scope of impact of future changes or corrections. Such a technique is especially applicable and useful in embedded systems; since they are so directly tied to hardware, it is important to partition and localize each implementation detail with a particular hardware interface. This approach allows easier modifications due to possible hardware interface changes, and minimizes the amount of code affected.

If in designing the software modules, increasing levels of detail are deferred until later (subordinate code units), then the software design approach is called *top-down*. If, on the other hand, the design detail is dealt with first and then increasing levels of abstraction are used to encapsulate those details, the approach is obviously *bottom-up*.

In Figure 6.6, it would be possible to design the software by first describing the characteristics of various components of the system and the functions that are to be performed on them, such as opening, sizing, and closing windows. Then the window functionality could be broken down into its constituent parts, such as rectangles and text. These could be subdivided even further, that is, all rectangles consist of lines, and so on. The top-down refinement continues until the lowest level of detail needed for code development has been reached.

Alternatively, it is possible to begin by encapsulating the details of the most volatile part of the system, the hardware implementation of a line or pixel, into a single code unit. Then working upward, increasing levels of abstraction are created until the system requirements are satisfied. This is a bottom-up approach to software design. In many real-world applications, however, the software design process contains both top-down and bottom-up sections.

6.3.2 Structured Design

Structured design (SD) is the companion methodology to structured analysis. It is a systematic approach concerned with the specification of the software architecture and involves a number of strategies, techniques, and tools. SD supports a comprehensive but easy-to-learn design process that is intended to provide high-quality software and minimized life cycle expenses, as well as to improve reliability, maintainability, portability, and overall performance of software products. Structured analysis (SA) is related to SD in the same way as a requirements representation is related to the software architecture, that is, the former is functional and flat, but the latter is modular and hierarchical.

The transition mechanisms from SA to SD are purely manual and involve substantial problem-solving effort in the analysis and trade-offs of alternative approaches. Normally, SD proceeds from SA in the following manner. Once the context diagram (CD) is first created, a hierarchical set of data flow diagrams (DFDs) is developed. DFDs are used to partition system functions and document that partitioning inside the specification. The first DFD, the level 0 diagram, illustrates the highest level of system abstraction. Subdividing processes to lower and lower levels until they are ready for detailed design renders further DFDs with successive levels of increasing detail. This heuristic decomposition process is called downward leveling, and it corresponds to the top-down design approach. Nevertheless, the bottom-up approach is also used commonly when developing DFDs. In that case, the composition process is called upward leveling. A problem-driven mixture of downward and upward leveling is preferred by most software designers (Yourdon, 1989).

In the CD (see Fig. 5.13), rectangles represent terminators that model the environment boundary. They are labeled with a noun phrase that describes the agent, device, or system from which data enters or to which it exits. Each process (or data transformation) depicted by a circle in CD/DFDs is labeled as a verb phrase describing the operation to be performed on the data, although it may be labeled with the name of a system or specific operation that manipulates the data as well. Solid arrow lines are used to connect terminators to processes and between processes to indicate the flow of data through the system. Each arrow line is labeled with a noun phrase that describes the data it carries. Moreover, parallel lines indicate data stores, which are labeled by a noun phrase naming the database, file, or repository where the system stores data (either simple data elements or a more complex data structure). A data store is passed to lower levels of hierarchy by connecting it with the corresponding process.

Each DFD should preferably have between five and nine processes (Yourdon, 1989). The descriptions for the lowest level processes are called process specifications, or P-SPECs, and are expressed in either decision tables or trees, pseudocode, or structured English, and are used to describe the detailed algorithms and operational logic of the actual program code. Yourdon stated that the purpose of structured English is "to strike a reasonable balance between the precision of a formal programming language and the casual informality and readability of the English language" (Yourdon, 1989). Figure 6.7 illustrates a typical evolution path from the context diagram through data flow diagrams to process specifications.

Example: Highest-Level DFD of the Elevator Control System

Consider again the elevator control system discussed in Section 3.3.8 and refer to its context diagram given in Figure 5.13. The associated level 0 DFD is shown in Figure 6.8. It contains five individual processes and three shared data stores ("global memory"). To create such a DFD, a thorough view/understanding of the elevator control system to be designed is developed gradually; hence, the resulting DFD is a refined outcome of a longish iterative process consisting of both top-down and bottom-up stages.

It should be noted that this DFD includes also a few control flows (dashed arrow lines), which are used to activate individual processes. These

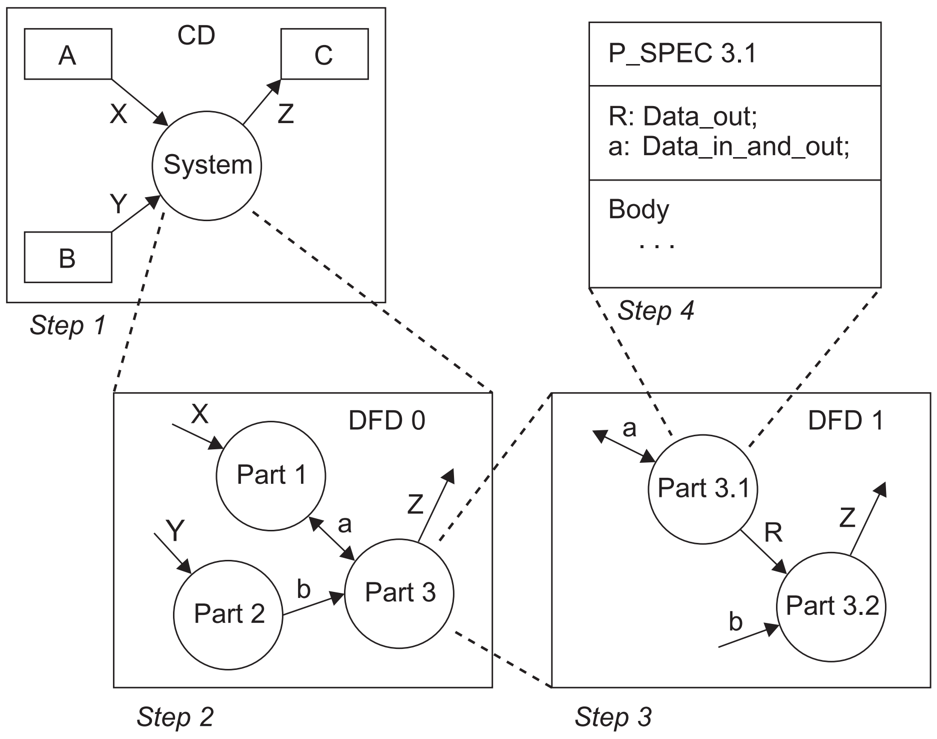

Figure 6.7. Evolution path from the context diagram to level 0 DFD to level 1 DFD, and finally to a P-SPEC.

activations are related to hardware interrupts and certain internal events, as outlined below:

1. ***Communications.*** Activated when the group dispatcher sends a request to communicate.
2. ***Update Destination.*** Periodic activation (75-ms timer interrupt).
3. ***Perform Runs.*** Activated primarily by Process 2 (also by the door and door zone interrupts) when there is a need to start a floor-to-floor run or to stop at the next possible floor (or perform some critical door control actions).
4. ***Supervise Operation.*** Periodic activation (500-ms timer interrupt).
5. ***Connect to Service Tool.*** Activated when an elevator technician presses some key of the service tool.

Notice that here the hardware interrupts were not included in the context diagram, but appear, for the first time, in this level 0 DFD.

To complement the DFDs, entity relationship diagrams (ERDs) are often used to define explicit relationships between stored data objects in the system. Hence, the entities of the ERD are modeling information concepts of the software application.

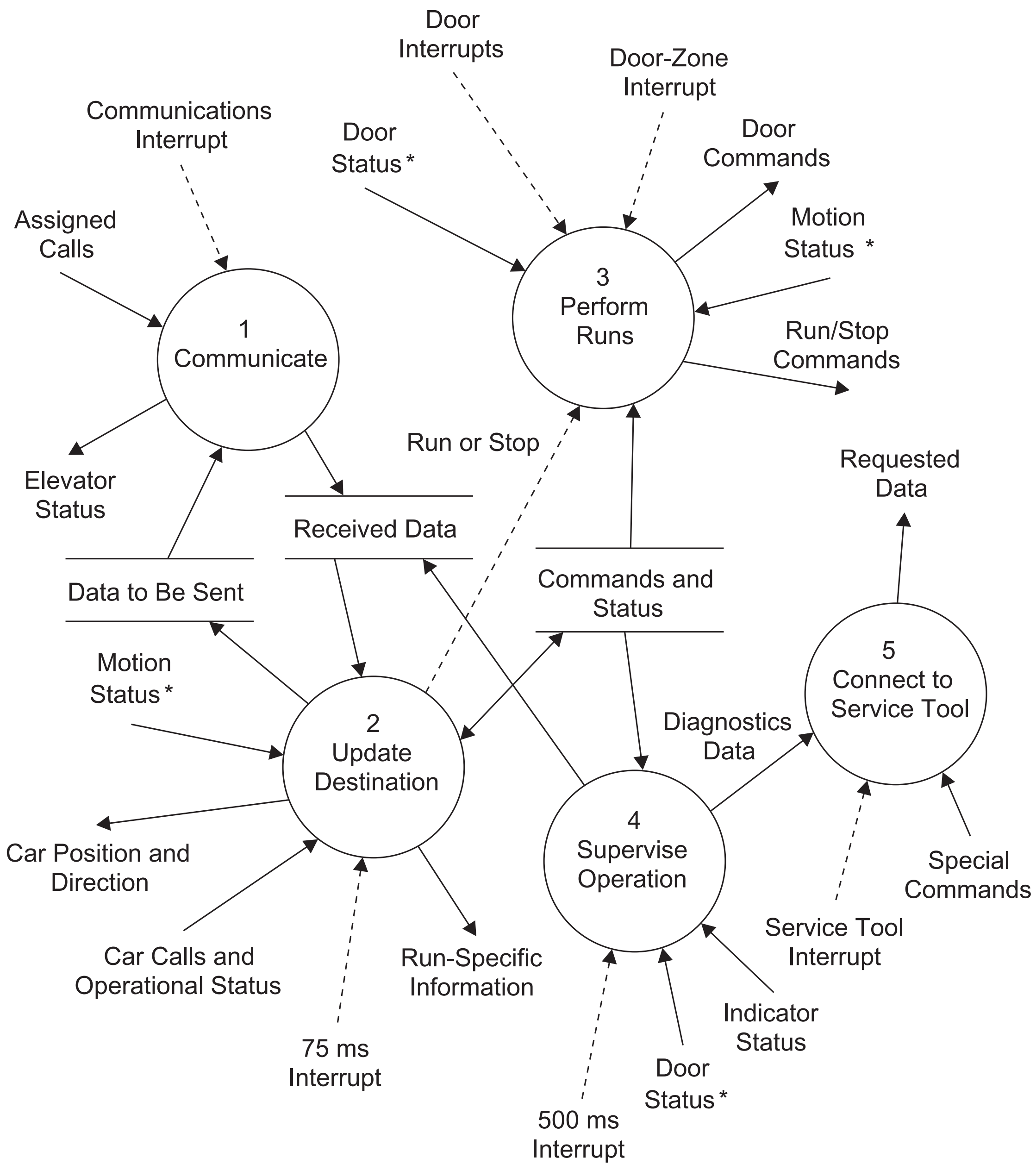

Figure 6.8. Level 0 DFD for the elevator control system. * This incoming data flow is connected to two processes.

Furthermore, a data dictionary (DD) is an essential component of the structured design, and includes entries for data flows, control flows, data stores, and buffers appearing in DFDs and control flow diagrams (to be discussed shortly). In addition, also the entries of ERDs should be included in the DD. Each entry is identified typically by its name, entry type, range, resolution, unit, location, and so forth. The data dictionary is organized alphabetically for ease of use. Other than that, there is no standard format, but every design element must have a descriptive entry in it. Most SA/SD CASE tools support the data-dictionary feature in addition to the diagrams mentioned above.

Example: A Sample Data-Dictionary Entry

For the elevator control system, one DD entry might appear as follows:

***Name*:**	Car call table
***Alias*:**	Car_calls
***Entry type*:**	Data store
***Description*:**	An integer vector containing the car call status for each possible destination floor
***Values*:**	"1" corresponds to "car call registered" and "0" represents "no car call," whereas other values are illegal
***Location*:**	Level 2.1 DFD

Additional "Location" information will be added as the program code is developed. In this way, data dictionaries help to provide traceability between design/code elements.

There are, however, apparent problems in using the standard structured analysis and structured design (SA/SD) to model *real-time* systems, including difficulty in modeling time dependencies and events. Consequently, concurrency is not adequately depictable using this form of SA/SD.

Another problem may arise already when creating the context diagram. Control flows are not easily translatable into code because they are hardware or operating-system dependent. In addition, such a control flow does not really make sense since there is no connectivity between portions of it, a condition known as "floating." As a representative example, the DFD of Figure 6.8 has altogether six floating control flows associated with hardware interrupts.

Details of the underlying hardware need to be known for further modeling of certain processes. For example, what happens if the communications hardware (interacts with Process 1) is changed? Or if another service tool with a different kind of keypad or display panel is taken in use (interacts with Process 5)? In such cases, the hardware-originated changes would need to propagate into the level 1 DFD for the corresponding process, any subsequent levels, and, ultimately, into the program code.

Making and tracking changes in structured design is fraught with danger, and hence requires special attention. Besides, a single change could mean that significant amounts of code would need to be rewritten, recompiled, and properly linked with the unchanged code to make the system work.

As expressed above, the standard SA/SD methodology is not well equipped for dealing with time, obviously, because it is a data-oriented and not a control-oriented approach. In order to address this shortcoming, the SA/SD method was extended by allowing for the addition of *control flow analysis*. This extension of SA/SD is called real-time SA/SD (SA/SD/RT). To accomplish this, the following artifacts were added to the standard approach: dashed arrow lines to indicate the flow of control messages and dashed parallel lines indicating message buffers. More specifically, dashed arrow lines can be either triggering events, such as hardware interrupts, or specific control flows between processes. A control flow can carry a single message (such as "activate" or "deactivate"), or it can form a structure of multiple messages. A message buffer, on the other hand, is a data store that contains explicit control characteristics, since it can behave autonomously as a stack or queue. Furthermore, a dashed circle represents a control transformation in Ward-Mellor SA/SD/RT (Ward and Mellor, 1985), and it can be used conveniently to sequence data flow diagrams. For that purpose, Mealy-type finite state machines are commonly used to define the encapsulated state sequence and corresponding process activations.

The addition of the control artifacts allows, in principle, for the creation of a diagram containing solely control artifacts called a control flow diagram (CFD). These CFDs can be further decomposed into C-SPECs (control specifications), which can then be described by finite state machines. However, the control and data flow diagrams are usually combined as shown in Figure 6.8. The important relationship between the control and process models is depicted in Figure 6.9.

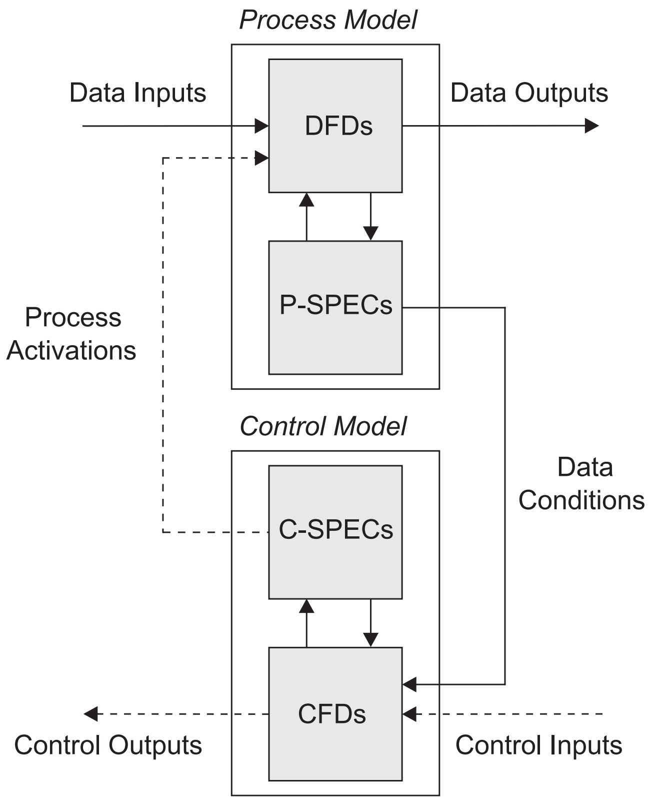

Figure 6.9. The relationship between control and process models (Laplante, 2003).

6.3.3 Design in Procedural Form Using Finite State Machines

One of the advantages of using finite state machines in the software requirements specification and later in the software design is that they can be easily (or even automatically) converted to code and test cases. For instance, consider the control of the elevator door. The tabular representation of the state transition function (see Table 5.2), which describes the system's high-level behavior rigorously, can be easily transformed into a design using the generic pseudocode shown in Figure 6.10. Each procedure associated with the possible door states (Open, Closing, Closed, Opening, Nudging, Fault C, and Fault O) will be structured code that can be viewed as executing in one of any number of possible states at every instant in time. This functionality can be described conveniently by the pseudocode shown in Figure 6.11.

```
typedef states:    (state 1,...,state n); {n is # of states}
        alphabet:  (input 1,...,input n);
        table_row: array [1..n] of states;

procedure move_forward; {advances FSM one state}

var
        state: states;
        input: alphabet;
        table: array [1..m] of table_row; {m is alphabet's size}

begin
        repeat
           get(input); {read one token from input stream}
           state := table[ord(input)] [state]; {next state}
           execute_process (state);
           until input = EOF;
end;
```

Figure 6.10. A generic pseudocode that can implement the behavior of finite state machines (Laplante, 2003).

```
procedure execute_process (state: states);

begin

    case state of
    state 1: process 1; {execute process 1}
    state 2: process 2; {execute process 2}
    ...
    state n: process n; {execute process n}

end;
```

Figure 6.11. Finite-state-machine code for executing a single operational process; each process can exist in multiple states, allowing partitioning of the program code into appropriate modules (Laplante, 2003).

Moreover, the pseudocodes given in Figures 6.10 and 6.11 can be easily translated to any procedural language or even to an object-oriented one. Alternatively, the system behavior can be described with a `case` statement or nested `if-then` statements such that, given the current state and receipt of a signal, a new state is assigned. The advantage of finite-state machine design over the `case` statement alternative is, of course, that the former is more flexible and compact.

6.4 OBJECT-ORIENTED DESIGN APPROACH

As discussed in Chapter 4, object-oriented programming languages are those characterized by data abstraction, inheritance, polymorphism, and messaging. Data abstraction through a variety of objects provides facilities for effective information hiding, or encapsulation and protected variation. Inheritance allows the software engineer to define new objects in terms of previously defined ones so that the new objects inherit properties. Function polymorphism allows the programmer to define operations that behave differently, depending on the type of object involved. Moreover, messaging allows objects to communicate and invoke the methods that they support.

Object-oriented languages provide a natural environment for information hiding through encapsulation. The state, data, and behavior of objects are encapsulated and accessed only via a published interface or certain private methods. For example, in the inertial measurement system (see Fig. 5.6), it would be appropriate to design a class called "accelerometer" with attributes describing its physical implementation and methods describing its output, compensation algorithm, and so forth.

Object-oriented design is a modern approach to systems design that views the system components as objects, as well as data processes, control processes, and data stores that are encapsulated within objects. Early forays into object-oriented design were led by aims to reuse some of the better features of structured methodologies, such as the data flow and entity relationship diagrams, by reinterpreting them in the context of object-oriented languages. This can be observed also in the popular unified modeling language (UML), which became standardized in the late nineties; the latest revision of the standard is UML 2.3 that was released in May 2010.

6.4.1 Advantages of Object Orientation

Over the last decade, the object-oriented framework has gained significant acceptance within the embedded-software community. The main advantages of applying object-oriented paradigms in real-time systems are the future extensibility and reuse that can be attained, and the relative ease of future changes. Also, the productivity of programmers is potentially improved through the use of object-oriented techniques. Most software systems are subject to near-continuous change: requirements change, merge, emerge, and mutate; target languages, platforms, and architectures change; and the way the software is employed in practice changes, too. Larman pointed out that after the initial release of a typical software product, at least half of the effort and cost is spent in modification (Larman, 2002a). This calls for flexibility and places a considerable burden on the software design: How can systems that must support such widespread change be built without compromising quality measures? There are four basic principles of object-oriented software engineering that address this question, and they have been recognized collectively as *supporting reuse*.

First recorded by Meyer, the *open-closed principle* (OCP) states that classes should be open to extension, but closed to modification (Meyer, 2000). That is, it should be possible to extend the behavior of a class in response to new or changing requirements, but modification to the source code is not allowed. While these expectations may seem at odds (particularly to those whose background is primarily in procedural languages), the obvious key is abstraction. In object-oriented systems, a superclass can be created that is fixed, but can represent unbounded variation by subclassing. This aspect is clearly superior to structured approaches in making changes, for instance, in accelerometer compensation algorithms, which would require new function parameter lists and wholesale recompilation of all modules calling that code in the structured design.

While not a new idea, Beck gave a name to the principle that any aspect of a software system—be it an algorithm, a set of constants, documentation, or logic—should exist in one and only one place (Beck, 1999). This so-called *once-and-only-once principle* (OAOOP) isolates future changes, makes the system easier to comprehend and maintain, and through the low coupling and high cohesion that the principle instills, the reuse potential increases significantly (Beck, 1999). The encapsulation of state and behavior in objects, and the ability to inherit properties between classes, allows for the rigorous application of these ideas in an object-oriented system, but is difficult to implement in structured approaches. More importantly, in structured approaches, OAOOP needs to be breeched frequently for reasons of performance, reliability, availability, and, often, for security as well.

Furthermore, the *dependency inversion principle* (DIP) states that high-level modules should not depend upon low-level modules; both should depend upon abstractions. This can be reformulated: Abstractions should not depend upon details—details should depend upon abstractions. Martin introduced this idea as an extension to the OCP with reference to the proliferation of dependencies that exist between high- and low-level modules (Martin, 1996). For example, in a structured decomposition approach, the high-level procedures reference the lower-level procedures, but changes often occur at the lowest levels. This infers that high-level modules or procedures that should be unaffected by such detailed modifications may be affected due to these dependencies. Again, consider the case where the accelerometer characteristics change and even though perhaps only one routine needs to be rewritten, the calling module(s) may need to be modified and recompiled, too. A preferable situation would be to reverse these dependencies, as is evident in the Liskov substitution principle (LSP). The intent here is to allow dynamic changes in the preprocessing scheme, which is achieved by ensuring that all the accelerometer objects conform to the same interface, and are therefore interchangeable.

Definition: Liskov Substitution Principle

> Liskov expressed the principle of the substitutivity of subclasses for their base classes as: If for each object o_1 of type S, there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T (Liskov, 1988).

This useful principle has led to the concept of type inheritance and is the basis of polymorphism in object-oriented systems, where instances of derived classes can be substituted for each other, provided they fulfill the obligations of a common superclass.

6.4.2 Design Patterns

Developing embedded software is hard, and developing truly reusable software is even harder. Competitive software designs should be specific to the current problem, but general enough to address potential future problems and requirements. Hence, there may arise a cost-related conflict between short-term and long-term goals. Experienced designers know not to solve every problem from first principles, but to reuse solutions encountered previously, that is, they find recurring patterns and use them as a basis for new designs. This is simply an embodiment of the principle of generality.

While object-oriented systems can be designed to be as rigid and resistant to extension and modification as in any other paradigm, object-orientation has the ability to include distinct design elements that can cater to future changes and extensions. These "design patterns" were first introduced to the mainstream of software engineering practice by Gamma, Helm, Johnson, and Vlissides, and are commonly referred to as the "Gang of Four" (GoF) patterns (Gamma et al., 1994).

The formal definition of a pattern varies throughout the literature. We will use the following informal definition throughout this text.

Definition: Pattern

> A pattern is a named problem–solution pair that can be applied in different contexts, with explicit advice on how to apply it in new situations.

Our presentation is concerned with three pattern types: *architectural patterns*, *design patterns*, and *idioms*. An architectural pattern occurs across subsystems; a design pattern occurs within a subsystem, but is independent of the programming language; and an idiom is a low-level pattern that is language specific (Horstmann, 2006).

In general, every pattern consists of four essential elements:

1. A name (such as "façade")
2. The problem to be solved (such as "provide a unified interface to a set of interfaces in a subsystem")
3. The solution to the problem
4. The consequences of the solution

More accurately, the problem describes when to apply the pattern in terms of specific design problems, such as how to represent algorithms as objects. The problem may describe class structures that are symptomatic of an inflexible design. Finally, the problem section might include conditions that must be met before it makes sense to apply the pattern.

The solution, on the other hand, describes the elements that make up the design, though it does not describe any concrete design or implementation. Rather, the solution provides how a general arrangement of objects and classes solves the problem. Consider, for instance, the previously mentioned GoF patterns. They describe 23 design patterns, each being either *creational*, *behavioral*, or *structural* in its intent (see Table 6.5). This table is provided for illustration only, and it is not our intention to describe any of these patterns in detail, since they are well documented elsewhere (Gamma et al., 1994). Some patterns have evolved specifically for real-time systems, and they provide various approaches to addressing the fundamental real-time scheduling, com-

TABLE 6.5. The Original Set of Design Patterns Popularized by the "Gang of Four" (Gamma et al., 1994)

Creational	Behavioral	Structural
Abstract factory	Chain of responsibility	Adapter
Builder	Command	Bridge
Factory method	Interpreter	Composite
Prototype	Iterator	Decorator
Singleton	Mediator	Façade
	Memento	Flyweight
	Observer	Proxy
	State	
	Strategy	
	Template method	
	Visitor	

munications, and synchronization problems, for example, Douglass (2003) and Schmidt et al. (2000).

Let us consider Douglass' real-time pattern set. Douglass groups his 48 patterns into six classes (Douglass, 2003):

1. Subsystem and component architecture
2. Concurrency
3. Memory
4. Resource
5. Distribution
6. Safety and reliability

As it turns out, we have discussed many of these patterns in Chapter 3 (but without mentioning the term "pattern"). The architecture patterns include the layered architecture that is so common to real-time operating systems (see Fig. 3.2), and the virtual machine which is the underlying architecture for Java. Of the concurrency patterns, many are based on solutions that we have already discussed, for instance, "round-robin," "static priority," "dynamic priority," and "cyclic executive." Various solutions for memory allocation, buffering, and garbage collection, are included in the memory patterns. The resource patterns, on the other hand, describe solutions to the critical-section problem through the use of semaphores, and the priority inheritance and priority ceiling protocols, among others. The distribution patterns deal with the problem of a synchronous control over a set of independent processes, and incorporates solutions found in other sets, such as the GoF's observer and proxy patterns. Finally, the safety and reliability patterns give solutions to improve fault tolerance and reliability through various types of redundancy, watchdog timers, and the like, many of which we will discuss in Chapter 8. Moreover, Douglass' pattern set includes many other solutions to real-time problems in a format that is quite accessible to the developer.

A comprehensive study on available pattern collections is provided by Henninger and Corrêa (2007). They pointed out: "As the number of patterns and diversity of pattern types continue to proliferate, pattern users and developers are faced with difficulties of understanding what patterns already exist and when, where, and how to use or reference them properly." This relevant concern is based on a careful survey, where altogether 170 software development-related pattern entities with more than 2200 patterns were identified and classified. A majority of those patterns is of architectural or design type. To avoid overlooking opportunities to utilize design patterns effectively, Briand et al. proposed a methodology for semiautomating the detection of areas within UML designs that are suitable candidates for the use of design patterns (Briand et al., 2006). Such methodologies, if just available in CASE environments with high usability, could advance the use of design patterns among practitioners.

6.4.3 Design Using the Unified Modeling Language

Today, the UML is widely accepted as the de facto standard for the specification and design of software-intensive systems using the object-oriented approach. By bringing together the "best-of-breed" in diverse specification techniques, the UML has become a sophisticated family of individual languages or diagram types, and users can choose which members of the family are suitable for their particular domain. Furthermore, complete UML models consist of a collection of diagrams, as well as accompanying textual and other documentation.

The UML is a graphical language based upon the premise that any system can be composed of communities of interacting entities. Various aspects of those entities and their interaction can be described using the original set of nine diagrams: activity, class, communication, component, deployment, sequence, state machine, object, and use case. Of these UML diagrams, five depict behavioral or dynamic views (activity, communication, sequence, state-machine, and use-case), while the remaining four are concerned with structural or static aspects. With respect to real-time systems, it is the behavioral diagrams that are of particular interest, since they define what must happen in the system under consideration. Many of those original diagrams are illustrated in the extensive design case study at the end of this chapter. The principal artifacts generated when using the UML as well as their relationships are depicted in Figure 6.12.

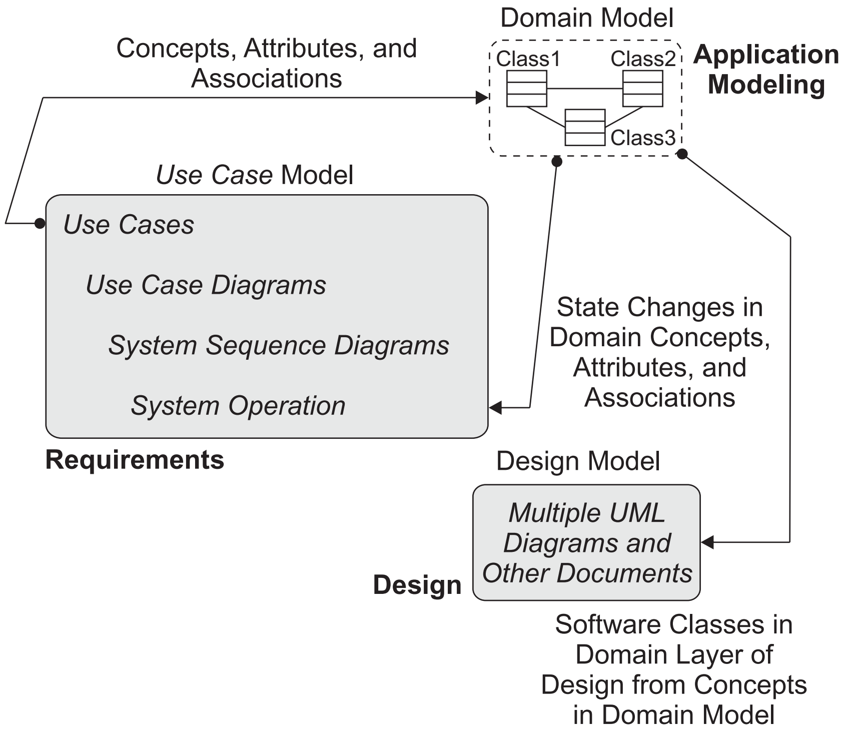
Domain Model
Concepts, Attributes, and Associations
Class1
Class2
Class3
Application Modeling
Use Case Model
Use Cases
Use Case Diagrams
System Sequence Diagrams
System Operation
Requirements
State Changes in Domain Concepts, Attributes, and Associations
Design Model
Multiple UML Diagrams and Other Documents
Design
Software Classes in Domain Layer of Design from Concepts in Domain Model

Figure 6.12. The role of UML in specification and design; adapted from Larman (2002b).

In addition to the nine diagrams mentioned above, the UML 2.2 (released in 2009) has five other diagrams (OMG Unified Modeling Language™ [OMG UML], 2009). Nonetheless, some of the numerous diagrams are partially redundant and used infrequently. All the 14 UML diagrams are introduced below in alphabetical order. For each of the diagrams, a suggested *Learning Priority* (*LP*) is given according to Ambler (2004); Ambler's *LP* has three possible levels: high, medium, and low. Although these suggestions are referring to the needs of "a business application developer," they give helpful guidelines also for real-time software developers.

Activity Diagram **(Behavioral/General; *LP* = High):** Activity diagrams are closely related to the classical flowchart and are used for the same purpose, that is, to specify the flow of control. However, unlike flowcharts, they can model concurrent computational steps and the flow of objects as they move from state to state at different points in the flow of control. In fact, in UML 2.0 and later, the activity diagram was refashioned to be more similar to the Petri net, which is widely used in digital hardware design to conduct synchronization analysis and to identify deadlocks, race conditions, and dead states. Thus, activity diagrams are useful in modeling dynamic aspects of a real-time system.

Class Diagram **(Structural; *LP* = High):** During system design, the class diagram defines the actual class attributes and methods implemented in an object-oriented programming language. Design pattern architectures are explored and physical requirements assessed during design. Design patterns provide guidance on how the defined class attributes, methods, and responsibilities should be assigned to objects. Physical requirements require the programmer to revisit the analysis class diagram, where new classes for the system requirements are defined. Figure 6.A10 in Appendix 1 at the end of this chapter is a design class diagram for the traffic-light control system.

Communication Diagram **(Behavioral/Interaction; *LP* = Low):** Communication diagrams show the messages passed between objects through the basic associations between classes. In essence, they depict the dynamic behavior on static class diagrams. Communication diagrams are the most emphasized of UML interaction diagrams because of their clarity and expression of more information. The communication diagram contains classes, associations, and message flows between classes. Figures 6.A4–6.A9 in Appendix 1 at the end of the chapter are communication diagrams for the traffic-light control system.

Component Diagram **(Structural; *LP* = Medium):** These diagrams are made up of components, interfaces, and relationships. Components represent preexisting entities. Interfaces represent the functionality of components that are directly available to the user, and relationships represent conceptual relationships between components (Holt, 2001).

***Composite Structure Diagram* (Structural; *LP* = Low):** Composite structure diagrams define the internal structure of a class and also the immediate collaborations that are enabled by this structure.

***Deployment Diagram* (Structural; *LP* = Medium):** Deployment diagrams consist of nodes representing real-world aspects (such as the hardware platform and execution environment) of a system, and links that show relationships between individual nodes.

***Interaction Overview Diagram* (Behavioral/Interaction; *LP* = Low):** These diagrams provide an interaction overview, where nodes represent individual interaction diagrams (a subset of behavioral diagrams).

***Object Diagram* (Structural; *LP* = Low):** Object diagrams realize part of the static model of a system and are closely related to class diagrams. They show the insides of things in the class diagrams, as well as their relationships. Moreover, they represent a model or "snapshot" of the partial or complete run-time system at a given point in time.

***Package Diagram* (Structural; *LP* = Low):** These diagrams show how the software system is partitioned into logical packages by depicting the interdependencies among these packages.

***Profile Diagram* (Structural; *LP* = low, Though Not Included in Ambler's Suggestions for UML 2.0):** A special kind of diagram that operates at the metamodel level (the metamodeling architecture is beyond the scope of this introduction).

***Sequence Diagram* (Behavioral/Interaction; *LP* = high):** sequence diagrams are composed of three basic elements: objects, links, and messages, which are exactly the same as for the communication diagram. However, the objects shown in a sequence diagram have a lifeline associated with them, which represents a logical timeline. The timeline is present whenever the object is active, and is illustrated graphically as a vertical line with logical time traveling down the line. The objects for the sequence diagram are shown going horizontally across the page and are shown staggered down the diagram depending on when they are created (Holt, 2001). Figure 6.A13 in Appendix 1 at the end of the chapter illustrates the sequence diagram for the traffic-light control system.

***State Machine Diagram* (Behavioral/General; *LP* = Medium):** These diagrams are versatile statecharts, which define the possible states and the allowed state transitions of the system.

***Timing Diagram* (Behavioral/Interaction; *LP* = Low):** Timing diagrams describe the critical timing constraints of the system.

***Use-Case Diagram* (Behavioral/General; *LP* = Medium):** Use-case diagrams represent the specific interactions of the software application with its external environment, as well as possible dependencies between individual use cases.

The UML, even in its current form, does not provide complete facilities for the specification and analysis needs of real-time systems. However, since the UML is an evolving family of languages, there is no compelling reason for not adding to the family if a suitable language is found. Unfortunately, the majority of appropriate candidates are formal methods—specification languages with a sound mathematical background—and these are traditionally shunned by practitioners.

As stated earlier, the domain model (see Fig. 6.12) is created based upon the use cases, and, through further exploration of system behavior via the interaction diagrams, the domain model evolves systematically into the design class diagram. The construction of the domain model is, therefore, analogous to the analysis stage in SA/SD described earlier. In domain modeling, the central objective is to represent the real-world entities involved in the domain as concepts in the domain model. This is a key aspect of object-oriented systems and is seen as a significant advantage of the paradigm, since the resultant model is closer to reality than in alternative modeling approaches, including the SA/SD.

While most development in object-oriented design was initially done with little or no provision for real-time requirements, the UML 2.0 (released in 2005) with significant extensions for real-time applications improved the situation greatly (Miles and Hamilton, 2006).

6.4.4 Object-Oriented versus Procedural Approaches

The preceding observations beg the question of whether object-oriented design is more suitable than structured design for embedded real-time systems. Structured design and object-oriented design are often compared and contrasted, and, indeed, they are similar in certain ways. This is no surprise, since both have their roots in the pioneering work of Parnas and his predecessors (Parnas, 1972, 1979). Table 6.6 provides a qualitative comparison of these methodologies.

TABLE 6.6. A Side-by-Side Comparison of SA/SD and OOAD (UML) Approaches

System Components	SA/SD Functions	OOAD Objects
Data processes Control processes Data stores	Separated through internal decomposition	All encapsulated within objects
Characteristics	Hierarchy of composition Classification of functions Encapsulation of knowledge within functions	Inheritance of properties Classification of objects Encapsulation of knowledge within objects
User's viewpoint	Rather easy to learn and use	Much more difficult to learn and use
CASE tools	Widely available	Widely available
Volume of usage	Shrinking	Growing

Both structured and object-oriented analysis and design (OOAD) are full life cycle methodologies and use some similar tools and techniques. However, there are major differences as well. SA/SD describes the system from a functional perspective and separates data flows from the functions that transform them, while OOAD describes the system from the perspective of encapsulated entities that possess both function and form.

Additionally, object-oriented models include inheritance, while structured ones do not have such a useful characteristic. Although SA/SD has a definite hierarchical structure, this is a hierarchy of decomposition rather than heredity. Such a shortcoming leads to difficulties in maintaining and extending both the specification and design.

From the user's viewpoint, UML is more difficult to learn and use than SA/SD methods, although they both are supported by matured CASE tools. On the other hand, we see the use of UML growing steadily, while the use of SA/SD is shrinking correspondingly in new products. Notably, these trends are slower in real-time applications than with other kind of software.

An experimental rule-based framework for transforming SA/SD artifacts to UML was proposed by Fries (2006). It is targeted for evolving legacy software that was initially designed using the structured approach. The original data flow and entity relationship diagrams of SA/SD are converted semiautomatically to a use case diagram, sequence diagrams, and a class diagram of UML.

Consider three distinct viewpoints of a system: data, events, and actions. Events represent stimuli, such as various measurements in control systems, as in the case study at the end of this chapter. Actions are precise rules that are followed in computational algorithms, such as "compensate" and "calibrate" in the case of the inertial measurement system. The majority of early computer systems were focused on one, or at most two, of these complementary viewpoints. For instance, nonreal-time image processing systems were certainly data and action intensive, but did not encounter much in the way of events.

Real-time systems are usually data intensive, and hence would seem well suited to structured analysis. Nevertheless, real-time systems also include control information, which is not particularly well suited to structured design. It is likely that a real-time system is as much event or action based as it is data based, which makes it quite suitable for object-oriented techniques, too.

The purpose of this discussion is not to dismiss SA/SD, or even to conclude that it is better than OOAD in all cases. An overriding indicator of suitability of OOAD versus SA/SD to real-time systems is the nature of the application. A similar conclusion was made—not surprisingly—when procedural and object-oriented programming languages were compared in Chapter 4.

6.5 LIFE CYCLE MODELS

A systematic engineering approach to the specification, design, programming, testing, and maintenance of software is essential for maximizing the reliability and maintainability of real-time systems, as well as for minimizing life cycle expenses. Therefore, software life cycle models form an integral part of any serious development and maintenance process for real-time systems; such models describe explicitly *what must be done* throughout the life cycle. The life cycle is considered to begin when the requirements engineering activities are commissioned and end when the particular software is no longer maintained by the responsible organization. This time period may vary from one year or so up to a few decades; and there are several life cycle models, which are practiced when developing and maintaining real-time software. These models include the classical waterfall model, the V-model, the spiral model, as well as a more recent collection of agile methodologies (Ruparelia, 2010). Nonetheless, most practiced life cycle models are actually hybrids; tailoring is commonly needed to create an appropriate compromise between strictly *sequential* and extensively *iterative* modeling approaches for a particular product and development organization.

Software life cycle models are intended to provide a solid and supportive framework leading to competitive software products within the available budget, personnel, and time frame. The word "competitive" refers here to an application- and environment-specific mixture of the desired software qualities discussed in Section 6.1. By using a well-defined life cycle model with thorough quality assurance procedures, it is possible to prevent the increasing and expensive late-failures period of the bathtub failure function (see Fig. 6.2) even in evolving embedded systems with a lengthy life span. In the following subsections, we will introduce a representative sample of sequential and iterative life cycle models and comment their strengths and weaknesses. All those models include at least a subset of the following fundamental activities:

- Requirements engineering
- Design
- Programming
- Testing
- Transfer to production
- Maintenance

6.5.1 Waterfall Model

The purely sequential waterfall (or cascade) model is the oldest software life cycle model, having its origins in the construction and manufacturing industries. It is based on the idealized assumptions that the requirements can be fixed on before starting the design phase, that the design can be fixed on before starting the programming phase, and so forth (see Fig. 6.13). Furthermore, there is typically a formal review between each phase, and one is allowed to advance to the following phase only when the preceding phase is finalized and

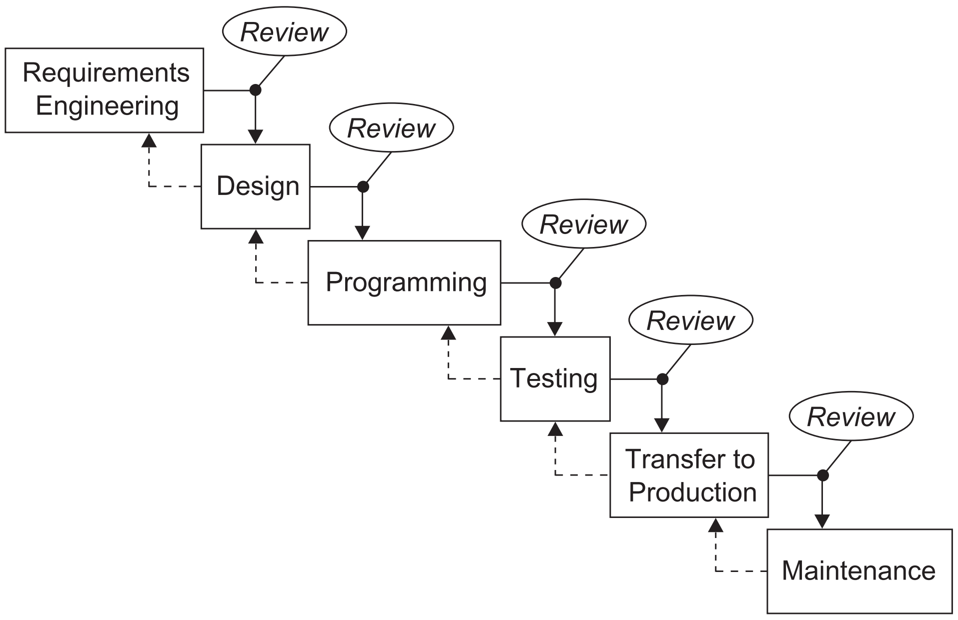

Figure 6.13. Sequential waterfall model with optional feedback enhancements.

approved. No feedback paths are provided for possible iteration in such an idealized scheme. In principle, it would be desirable to follow a feedforward model through the whole development process, but as there are multiple potential reasons why the requirements, design, or program code may need to be modified during a development project, the basic waterfall model has been enhanced to contain optional feedback paths (dashed lines in Fig. 6.13). These relaxing feedbacks make it possible to revisit preceding phases, for instance, to correct programming errors that were detected during tests. Although the enhanced waterfall model provides direct mechanisms for iteration, all the iterations are considered just as exceptions in the "waterfall philosophy." Moreover, quality assurance may be built seamlessly into the waterfall model; each phase can contain both the "*do* part" and the corresponding "*validate* part" (Ruparelia, 2010).

Since the introduction of the waterfall principles over five decades ago, various iterative models have appeared—and that seems to be the future trend, too. In iterative models, the development of all entities can continue throughout the life cycle. Nevertheless, according to a recent survey, a considerable majority (84%) of software projects were still developed according to the waterfall model (or one of its enhancements) and not using any of the modern iterative approaches (Gelbard et al., 2010).

As an important benefit, the cascade flow of the waterfall model makes it straightforward to even outsource individual development phases, since the firm documents, such as the approved Software Requirements Specification and the Software Design Description, are not expected to change later on. On the other hand, waterfall-type life cycle models do not support effectively such projects that have evolving or changing requirements specifications. This is an increasingly critical issue, for example, when developing a clear but versatile user interface to navigate through evolving smartphone features and functions.

6.5.2 V-Model

The waterfall model has been enhanced not only by introducing simple feedbacks between consecutive phases but in other ways, too. One widely used enhancement is the V-model, where "V" describes both the graphical shape of the development flow and the central objectives related to "validation." Figure 6.14 depicts the V-model for software development; the left fork contains the requirements engineering and design phases in the same way as they are included in the enhanced waterfall model of Figure 6.13; the programming effort with module testing is at the bottom; and the right fork is devoted to quality assurance actions. These quality assurance actions form the heart of V-model, and they are based on close interaction between the symmetrical left and right forks. This means, for instance, that strategies and plans for system validation and integration tests are created already during the requirements engineering and design phases, respectively. Hence, it is ensured that every requirement as well as the design itself are strictly verifiable (Ruparelia, 2010). The foremost aim of the V-model is to tackle two obvious risks appearing in any software development project:

1. Does the integrated software correspond *exactly* to the design?
2. Is the overall system fulfilling *all* the requirements?

In the traditional waterfall model, the testing phase contains similar activities as the right fork of the V-model, but the V-model emphasizes their role

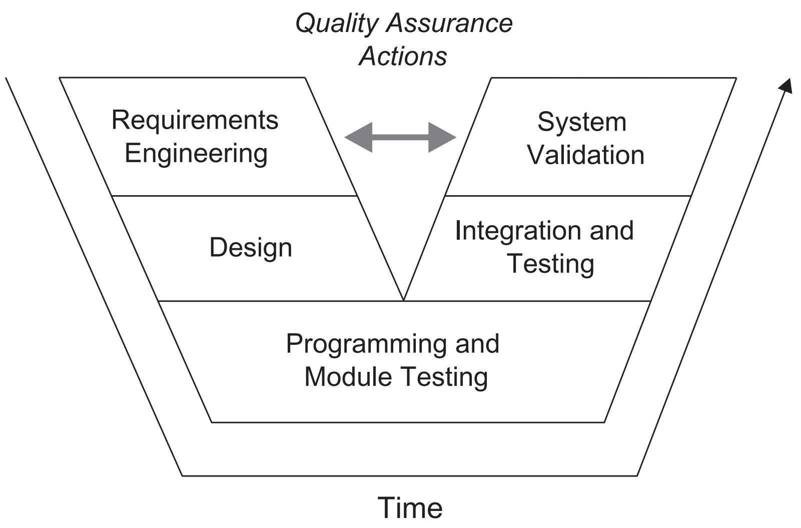

Figure 6.14. V-model with quality assurance activities.

throughout the development life cycle. It is a common practice to fine tune the presented model structure to correspond to the specific needs of a particular project; the five-phase structure of Figure 6.14 is just one example. The more complex a software system to be developed is, the longer the two forks become and hence contain more phases. In principle, the use of V-model is not dependent on the size of the software project. Furthermore, it can be used for hardware and mechanics development, as well.

6.5.3 Spiral Model

A particularly useful modification of the waterfall model, the spiral model (Boehm, 1986), has its orientation in risk analysis and intermediate prototyping. These are taking place cyclically until the phase of detailed design, which is then followed by a typical waterfall sequence (see Fig. 6.15). Each spiral cycle passes four quadrants, Q1–Q4, and ends up to a prototype that is validated, possibly with the stakeholders. The number of completed spiral cycles

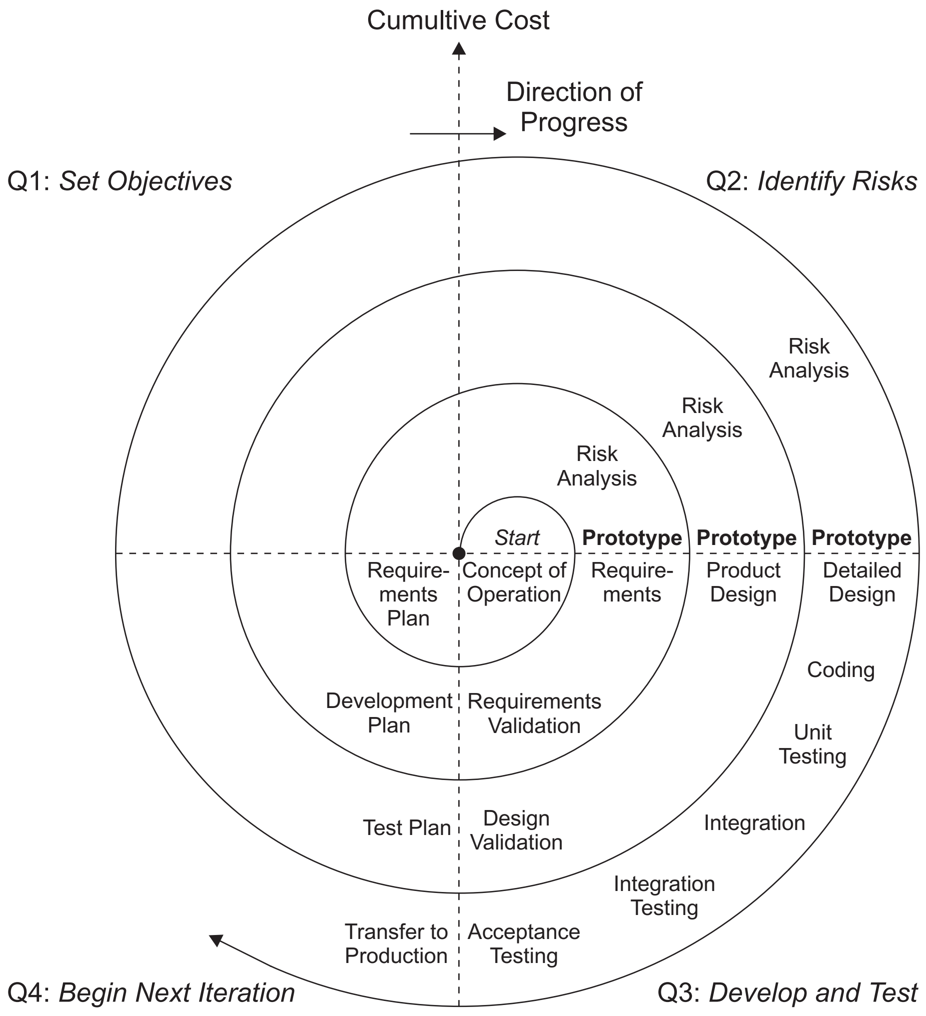

Figure 6.15. Spiral model with early prototyping efforts; adapted from Boehm (1986).

determines the cumulative cost of the development project. While the waterfall model is entirely specification driven, the spiral model is clearly a risk-driven approach.

Possible risks in a software development project are commonly related to the adequacy of available features, usability and user interface, real-time performance, externally furnished components (such as reused requirements specifications or partial designs), as well as various development issues (Boehm, 1991). Some of these risks may grow bigger if the software product is intended for a global market with significant cultural dynamics. Therefore, the importance of careful risk analysis is going to increase in the future. It should be noted, however, that the risk protection benefit of extensive prototyping can be costly. Besides, it is not always easy to identify critical risks, and hence development teams would benefit of risk identification and analysis training.

Also in the case of the spiral model, it is possible to adapt the model details to correspond to the needs of a particular project. Moreover, the use of the spiral model requires considerable effort by the project management. The philosophy of the spiral model can be stated as "start small, think big" (Ruparelia, 2010).

6.5.4 Agile Methodologies

Agile methodologies belong to a dynamic family of iterative and incremental software development strategies. While in the enhanced waterfall model of Figure 6.13, any iteration is considered as an undesired exception, agile methodologies are based on intentional iterations leading to incremental completion of the software under development. Hence, the underlying principle is far from that of the waterfall model, V-model, or spiral model, and it cannot be depicted with a static workflow diagram containing interconnected development activities. Figure 6.16 illustrates a flow of an imaginary software development project using an iterative agile strategy. At any iteration step, there are multiple merging development activities with varying effort volumes. Besides, a set of consecutive iterations can form a "mini project" that could provide a partial software release to the customer. Although agile methodologies are often deployed with a lack of rigid process, they can be, when correctly implemented, rigorous and thus suitable for embedded applications (Laplante, 2009).

There are several widely used agile methodologies, such as Crystal, Dynamic Systems Development Method, eXtreme Programming (XP), Feature-Driven Development, and Scrum, as well as a large number of *ad hoc* methodologies that claim to be agile (Laplante, 2009). Because they are relatively new, are light on documentation and formal process, and involve a high degree of experimentation early in the systems development process (when prototype hardware may be unavailable), agile methodologies are not frequently used in real-time and embedded systems development. Nevertheless, in many cases, where the true philosophy of agile development is embraced

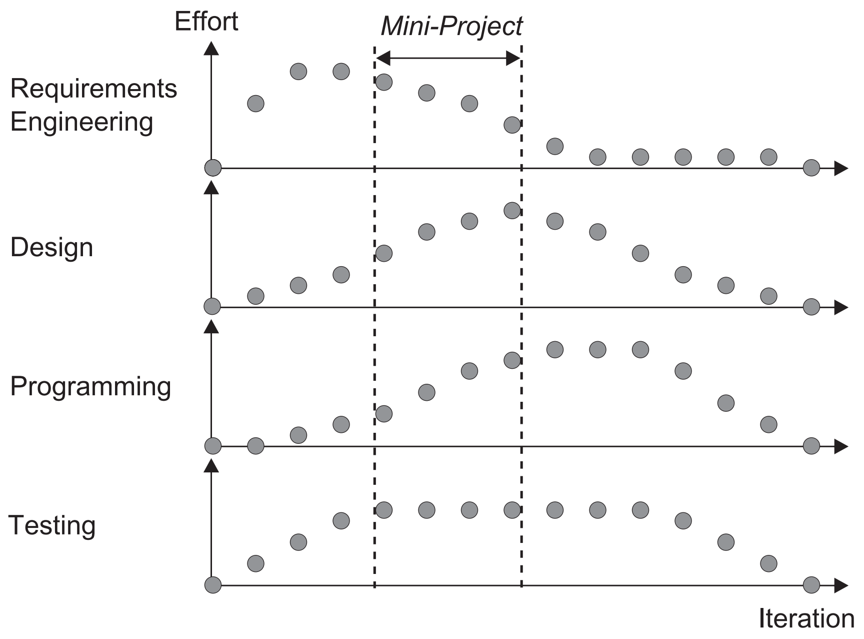

Figure 6.16. A sample flow of a software development project using iterative agile methodologies; adapted from Larman (2004).

and where the culture and application domain are appropriate, agile development can be the right development approach for real-time and embedded systems, as well.

It is beyond the scope of this book to describe any one agile methodology or to undertake a detailed analysis of when and how to use these approaches in real-time and embedded systems development. It is important, however, to understand and appreciate the philosophy of agile methodologies in order to see why they might be suitable for certain real-time applications. And to understand these approaches, it is essential to look at the Agile Manifesto and the explicit principles behind it. The following manifesto was introduced by a group of agility proponents in 2001 (Larman, 2004):

Definition: Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

That is, while there is value on the items on the right, we value items on the left more.

From the noble manifesto, a set of 12 principles were derived (Larman, 2004):

Definition: Agile Principles

P1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

P2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

P3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.

P4. Business people and developers must work together daily throughout the project.

P5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

P6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

P7. Working software is the primary measure of progress.

P8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

P9. Continuous attention to technical excellence and good design enhances agility.

P10. Simplicity—the art of maximizing the amount of work done—is essential.

P11. The best architectures, requirements, and designs emerge from self-organizing teams.

P12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These principles, if practiced consistently, end up to a flexible what-to-do plan for the project, instead of a firm one. Moreover, the agile approach is human oriented as opposed to task orientation. The principle P5 is particularly interesting, because it enables the team members to utilize their "free will" (a powerful characteristic, which distinguishes human intelligence from advanced machine intelligence [Martinez, 2006]) instead of being controlled merely by policies, procedures, superiors, and so on.

Could agile methodologies even offer "the most suitable strategy" for all software development? We will answer this tempting question by referring synergistically to the "no free lunch" (NFL) theorems that Wolpert and Macready discussed in the context of optimization algorithms (Wolpert and Macready, 1997). They proved that improved performance for any optimization algorithm indicates a match between the structure of the algorithm and the structure of the problem at hand. Therefore, a general-purpose algorithm is never the most suitable one for a specific problem, and the most suitable algorithm for a specific problem is not a general-purpose one. Intuitively reasoning, it could be possible to develop similar NFL theorems for software development strategies, as well; such an effort is, however, beyond the scope of this text. Nevertheless, if we apply our intuition freely, we can say that any general-purpose strategy is never the best one for all software development. For this reason, case-dependent tuning is advantageous when creating an appropriate life cycle model for a particular application and development environment. For instance, applying agile methodologies strictly to large software projects is difficult, since they stress the face-to-face communication and self-organizing teams that may not be possible to achieve due to large development groups and multiple geographical locations where the work is carried out (Ruparelia, 2010). Nonetheless, agile methodologies are undoubtedly usable in smaller-scale applications with changing or evolving requirements specifications.

Vignette: Are We Witnessing a Second Agile Development Period?

When examining the Agile Manifesto and the associated principles, we noticed that they contain numerous similarities to those informal software development strategies, which were used in the beginning of the embedded systems era—some three decades ago. At that time, industry was still inconsistent in the use of microprocessors; requirements specifications changed frequently; the entire software was not more than a few tens of kilobytes; development teams consisted typically of 2–3 highly motivated software engineers; and nobody in the team was a "guru," but all members were fairly inexperienced. Such initial conditions formed a fruitful basis for agile-type behavior to emerge by itself.

Based on the authors' subjective observations in a few development organizations—rather than any real survey—it can be argued that the *most successful* teams practiced up to 10 of the 12 agile principles: P1, P2, P4–P7, and P9–P12. Furthermore, all items of the manifesto were, more or less, common practices.

Is this just a coincidence, or is there truly something similar in the advanced products of tomorrow and the early embedded systems? No, it is probably not a coincidence, but the *frequently changing requirements and new technological opportunities* form common points of contact, which are as concrete with the future smartphone user interfaces as they were with the pioneering forest harvesters, for example. However, most of the agile principles were put aside for a couple of decades; since embedded software was growing rapidly in size, development teams became bigger and geographically distributed, and it was impossible to find an adequate number of well motivated and self-organizing individuals to support the embedded systems boom. These and other changes in the operating environment pushed the development organizations toward rigid life cycle models and strict project management practices.

Lastly, it should be pointed out that there are also other iterative software development approaches than the popular agile methodologies outlined above. A comprehensive discussion on agile and iterative development, from the manager's viewpoint, is available in Larman (2004).

6.6 SUMMARY

The purpose of software design is to create a sound mapping from the requirements document to an implementable design document. In general, there exist an infinite number of possible mappings. But which one of those is the desired one? To achieve a desirable mapping, we first need to define the term "desirable," which is related directly to a set of weighted software quality measures. These specific qualities (such as performance and maintainability) and usual ways to achieve them (such as modularity and generality) were discussed in the beginning of this chapter. In addition to software qualities, the term "desirable" is related necessarily to the development organization and environment, as well as the type and market of the software product. Thus, every design process includes actually a multi-objective optimization problem with considerable uncertainties; for instance, how should the different software qualities be weighted with respect to each other? The ultimate success of the software is largely dependent on the experience and skills of the design team to solve such problems.

There are two principal approaches to generate and document software designs: the procedural design approach and the object-oriented approach. It is useless to debate which approach is better and should hence be preferred generally. Instead, the selected approach must be justified by the concrete application needs and future visions of the development organization. Currently, many industrial companies with a long history of embedded systems development either have just switched to object-oriented techniques or are in the process of such a major transition. In large procedural-oriented organizations, this transition can be a laborious educational effort, since, for example, the fluent use of UML is more demanding than the use of SA/SD methods. The situation is apparently very different in young and small development organizations.

Software development and maintenance life cycle models have a central role in every serious development process. The purpose of strict "life cycle thinking" is to minimize the total expenses during the entire life cycle—not just the development expenses, as is traditionally done in the first place. However, while the life cycle thinking makes a lot of sense from the corporation or company point of view, it may be challenging to put into practice in large organizations, where the development expenses are often paid from "a different pocket" than the maintenance expenses. The situation is even more difficult when the lifespan of the software product is lengthy. Thus, the adoption of a life cycle model to cover both the development and maintenance phases is actually an executive-level decision.

The classical waterfall model or one of its enhancements have an established position in most development organizations. Nevertheless, as there is no single kind of software or development environment, there is a need to tune and evolve the existing life cycle models, or even create new life cycle philosophies. In the past decade, agile methodologies have gained interest and acceptance in applications where the requirements are changing frequently during the design and implementation phases. Agile methodologies provide an iterative and incremental alternative to the primarily sequential and rigid development life cycles. While these methodologies are shown to be effective in certain situations, they are no "silver bullet" for all. On the other hand, their position is clearly emerging in smaller-scale real-time systems (or subsystems) involved with novel technological opportunities.

In the future, such productivity issues as design reuse and (semi-)automatic design from requirements will continue to be important but challenging areas of research and development.

We want to close this chapter by Norman Maclean's captivating words: "Eventually, all things merge into one, and a river runs through it" (Maclean, 2001). These words have an obvious analogy to the design of embedded software.

6.7 EXERCISES

6.1. For whom should you, as a designer, prepare the software design description?

6.2. What are the primary reasons behind the current (and seemingly continuing) situation that there is no single, universally accepted approach for software design?

6.3. Why is it that the actual program code, even though it is an exact model of system behavior, is insufficient in serving as either a software requirements document or a software design document? In any case, pseudocode is used widely for such purposes.

6.4. How would you, as a software project manager, handle the confusing situation in which the software requirements specification contains numerous design-level details as well?

6.5. Why is it of utmost importance that the program code be traceable to the software design specification, and, in turn, to the software requirements specification? What are the possible consequences if it is not traceable?

6.6. A mapping from advantageous software engineering principles to desired software qualities is sketched in Table 6.4. Give specific explanations why "Modularity" is mapping to "Reliability," "Correctness," "Performance," "Interoperability," "Maintainability," and "Portability." Or, do you disagree on some of those suggested mappings?

6.7. What are the principal differences between procedural design approaches and object-oriented ones?

6.8. Why are procedural design approaches still practiced with many embedded systems—even with completely new products? What could be hindering the adoption of object-oriented approaches in those cases?

6.9. Using a data flow diagram, capture the data and functional requirements for monitoring the entry, exit, and traversal of aircraft in a busy airspace. Aircraft entering the space are sensed by the *Radar* input; the *Comm* input identifies aircraft that leave the space. The current contents of the space are maintained in the data area *AirspaceStatus*. A detailed log or history of the space usage is kept in the *AirspaceLog* storage. Air-traffic control personnel can request the display of the status of a particular aircraft through the *Controller* input.

6.10. Take the procedural design approach and create first the context diagram, and then the highest-level data flow and control-flow diagrams for an electronic lock in the laboratory door having the following requirements specifications:

- The lock has an integrated RFID card reader, and every registered user has a unique identification code.
- An accepted card is acknowledged by a green LED and a rejected one by a red LED.
- The lock will open when an adequate current is flowing through its control solenoid; otherwise, it remains locked.
- Information about registered users and their permitted entrance times is stored on a database of a remote workstation that manages all locks within the whole college building.
- Every successful and unsuccessful opening attempt is recorded on the database with the corresponding identification code, date, and time.
- Embedded controllers of individual locks in the building communicate with the common workstation through a wireless communications network.

You may define additional requirements yourself, if needed.

6.11. Perform a web search and find the reasons why the Unified Modeling Language (UML) was originally developed. What were the primary reasons why UML 2.0 appeared?

6.12. UML's use-case diagrams (see Fig. 5.14) are usually complemented by textual descriptions; what kind of information do they contain?

REFERENCES

K. K. Aggarwal, Y. Singh, and J. K. Chhabra, “An integrated measure of software maintainability,” *Proceedings of the Annual Reliability and Maintainability Symposium*, Seattle, WA, 2002, pp. 235–241.

S. W. Ambler, *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3rd Edition. New York: Cambridge University Press, 2004.

K. Beck, *Extreme Programming Explained: Embrace Change*. New York: Addison-Wesley, 1999.

L. Bernstein and C. M. Yuhas, *Trustworthy Systems through Quantitative Software Engineering*. Hoboken, NJ: Wiley-Interscience, 2005.

B. W. Boehm, “A spiral model of software development and enhancement,” *ACM SIGSOFT Software Engineering Notes*, 11(4), pp. 14–24, 1986.

B. W. Boehm, “Software risk management: Principles and practices,” *IEEE Software*, 8(1), pp. 32–41, 1991.

L. C. Briand, Y. Labiche, and A. Sauve, “Guiding the application of design patterns based on UML models,” *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, Philadelphia, PA, 2006, pp. 234–243.

F. Brooks, *The Mythical Man-Month*, 2nd Edition. New York: Addison-Wesley, 1995.

G. Caprihan, “Managing software performance in the globally distributed software development paradigm,” *Proceedings of the IEEE International Conference on Global Software Engineering*, Florianopolis, Brazil, 2006, pp. 83–91.

P. A. Dargan, *Open Systems and Standards for Software Product Development*. Norwood, MA: Artech House, 2005.

B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley, 2003.

T. P. Fries, “A framework for transforming structured analysis and design artifacts to UML,” *Proceedings of the 24th Annual ACM International Conference on Design of Communication*, Myrtle Beach, SC, 2006, pp. 105–112.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. New York: Addison-Wesley, 1994.

R. Gelbard, D. Te’eni, and M. Sadeh, “Object-oriented analysis—Is it just theory?” *IEEE Software*, 27(1), pp. 64–71, 2010.

O. Goldreich, *Computational Complexity: A Conceptual Perspective*. New York: Cambridge University Press, 2008.

E. Hadar and I. Hadar, “Effective preparation for design review: Using UML arrow checklist leveraged on the gurus’ knowledge,” *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Montreal, Canada, 2007, pp. 955–962.

S. Henninger and V. Corrêa, “Software pattern communities: Current practices and challenges,” *Proceedings of the 14th Conference on Pattern Languages of Programs*, Monticello, IL, 2007, Article no. 14.

J. Holt, *UML for Systems Engineering*. London, UK: IEE, 2001.

C. Horstmann, *Object-Oriented Design and Patterns*, 2nd Edition. Hoboken, NJ: John Wiley & Sons, 2006.

Institute of Electrical and Electronics Engineers, *IEEE Std 1016–2009, IEEE Standard for Information Technology—Systems Design—Software Design Descriptions*. New York: IEEE Computer Society, 2009.

S. Jantunen, "Exploring software engineering practices in small and medium-sized organizations," *Proceedings of the ICSE Workshop on Cooperative Aspects of Software Engineering*, Cape Town, South Africa, 2010, pp. 96–101.

C. Jones, *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. New York: McGraw-Hill, 2010.

P. A. Laplante, *Software Engineering for Image Processing*. Boca Raton, FL: CRC Press, 2003.

P. A. Laplante, *Requirements Engineering for Software and Systems*. Boca Raton, FL: CRC Press, 2009.

C. Larman, "Tutorial: Mastering design patterns," *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, 2002a, p. 704.

C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd Edition. Englewood Cliffs, NJ: Prentice-Hall, 2002b.

C. Larman, *Agile and Iterative Development: A Manager's Guide*. Boston: Pearson Education, 2004.

B. Liskov, "Data abstraction and hierarchy," *SIGPLAN Notices*, 23(5), pp. 17–34, 1988.

N. Maclean, *A River Runs through It and Other Stories*, 25th Anniversary Edition. Chicago, IL: The University of Chicago Press, 2001, p. 104.

R. C. Martin, "The dependency inversion principle," *C++ Report*, 8(6) pp. 61–66, 1996.

T. Martinez, "Computer-based intelligence: Where is it going?" *Opening Talk in Panel Discussion*, *IEEE Mountain Workshop on Adaptive and Learning Systems*, Logan, UT, July 26, 2006.

B. Meyer, *Object-Oriented Software Construction*, 2nd Edition. Englewood Cliffs, NJ: Prentice-Hall, 2000.

R. Miles and K. Hamilton, *Learning UML 2.0*. Sebastopol, CA: O'Reilly Media, 2006.

H. D. Mills, "Certifying the correctness of software," *Proceedings of the 25th Hawaii International Conference on System Science*, Kauai, HI, 1992, vol. 2, pp. 373–381.

J. Nielsen, *Usability Engineering*. New York: Academic Press, 1993.

OMG Unified Modeling Language™ (OMG UML), "Superstructure, Version 2.2," 2009, p. 686. Available at http://www.omg.org/spec/UML/2.2/, last accessed August 17, 2011.

D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, 15(12), pp. 1053–1058, 1972.

D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Transactions on Software Engineering*, 5(2), pp. 128–138, 1979.

H. Pham, *Software Reliability*. New York: Springer, 2000.

R. S. Pressman, *Software Engineering: A Practitioners Approach*, 7th International Edition. New York: McGraw-Hill, 2009.

N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, 35(3), pp. 8–13, 2010.

D. C. Schmidt, M. Stal, H. Robert, and F. Bushmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons, 2000.

Y. Shinjo and C. Pu, "Achieving efficiency and portability in systems software: A case study on POSIX-compliant multithread programs," *IEEE Transactions on Software Engineering*, 31(9), pp. 785–800, 2005.

M. F. Siok and J. Tian, "Empirical study of embedded software quality and productivity," *Proceedings of the 10th High Assurance Systems Engineering Symposium*, Plano, TX, 2008, pp. 313–320.

R. N. Taylor, N. Medvidović, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ: John Wiley & Sons, 2010.

X. Teng and H. Pham, "A new methodology for predicting software reliability in the random field environment," *IEEE Transactions on Reliability*, 55(3), pp. 458–468, 2006.

P. Ward and S. Mellor, *Structured Development for Real-Time Systems*, vols. 1–3. New York: Yourdon Press, 1985.

J. C. Wileden and A. Kaplan, "Software interoperability: Principles and practice," *Proceedings of the International Conference on Software Engineering*, Los Angeles, CA, 1999, pp. 675–676.

D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, 1(1), pp. 67–82, 1997.

E. Yourdon, *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1989.