CHAPTER 10
Software Engineering

Objectives

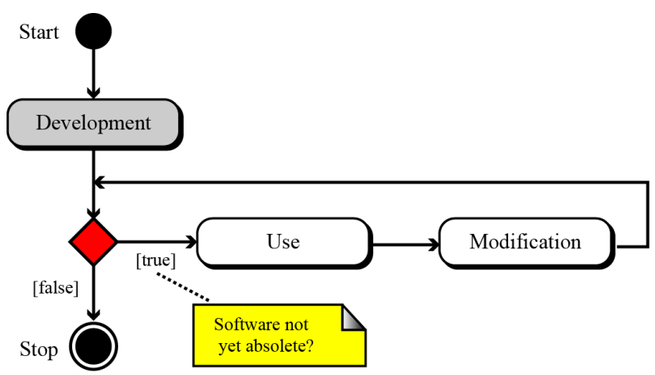
After studying this chapter, the student should be able to:

- Understand the concept of the software life cycle in software engineering.
- Describe two major types of development process, the waterfall and incremental models.
- Understand the analysis phase and describe two separate approaches in the analysis phase: procedure-oriented analysis and object-oriented analysis.
- Understand the design phase and describe two separate approaches in the design phase: procedure-oriented design and object-oriented design.
- Describe the implementation phase and recognize the quality issues in this phase.
- Describe the testing phase and distinguish between glass-box testing and black-box testing.
- Recognize the importance of documentation in software engineering and distinguish between user documentation, system documentation, and technical documentation.

10-1 THE SOFTWARE LIFECYCLE

A fundamental concept in software engineering is the *software lifecycle*. Software, like many other products, goes through a cycle of repeating phases (Figure 10.1).

Figure 10.1 The software lifecycle

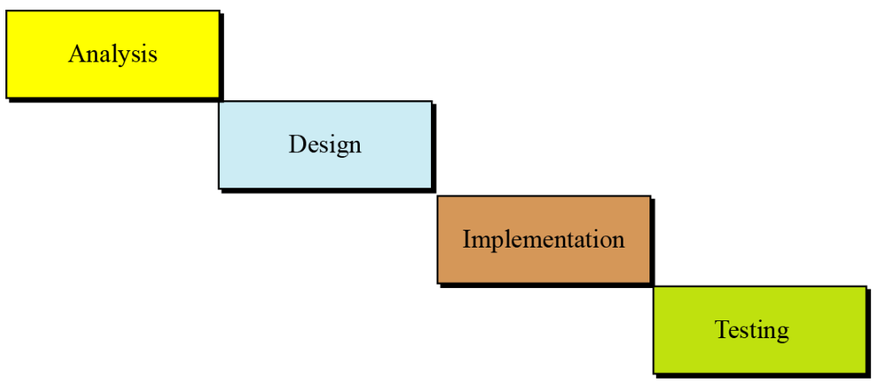


Although software engineering involves all three processes in Figure 10.1, in this chapter we discuss only the development process, which is shown outside the cycle in Figure 10.1. The development process in the software lifecycle involves four phases: analysis, design, implementation, and testing. There are several models for the development process. We discuss the two most common here: the waterfall model and the incremental model.

The waterfall model

In this model, the development process flows in only one direction. This means that a phase cannot be started until the previous phase is completed.

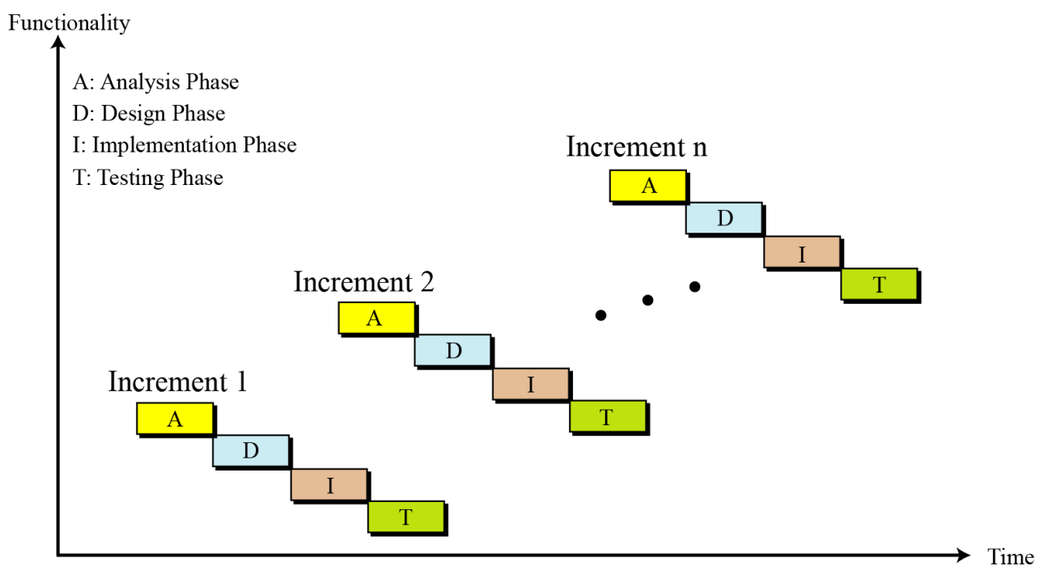
Figure 10.2 The waterfall model



The incremental model

In the incremental model, software is developed in a series of steps.

Figure 10.3 The incremental model



10-2 ANALYSIS PHASE

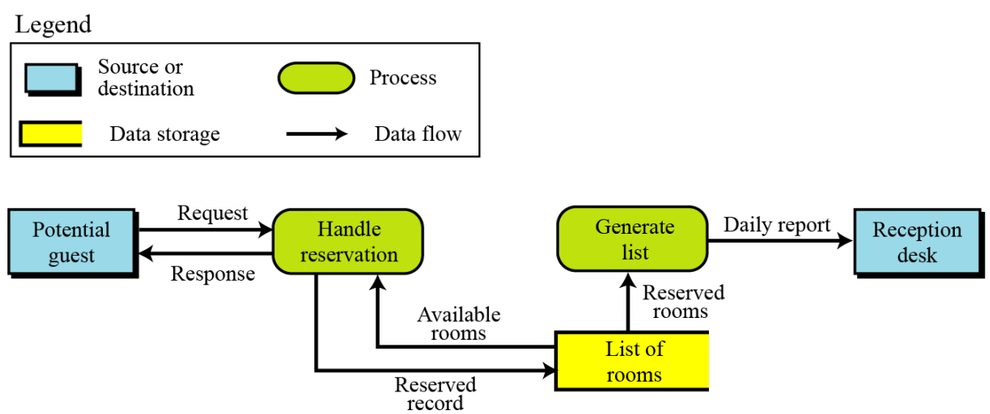
The development process starts with the analysis phase. This phase results in a specification document that shows what the software will do without specifying how it will be done. The analysis phase can use two separate approaches, depending on whether the implementation phase is done using a procedural programming language or an object-oriented language. We briefly discuss both in this section.

Procedure-oriented analysis—also called structured analysis or classical analysis—is the analysis process used if the system implementation phase will use a procedural language. The specification in this case may use several modeling tools, but we discuss only a few of them here.

Data flow diagrams

Data flow diagrams show the movement of data in the system.

Figure 10.4 An example of a data flow diagram



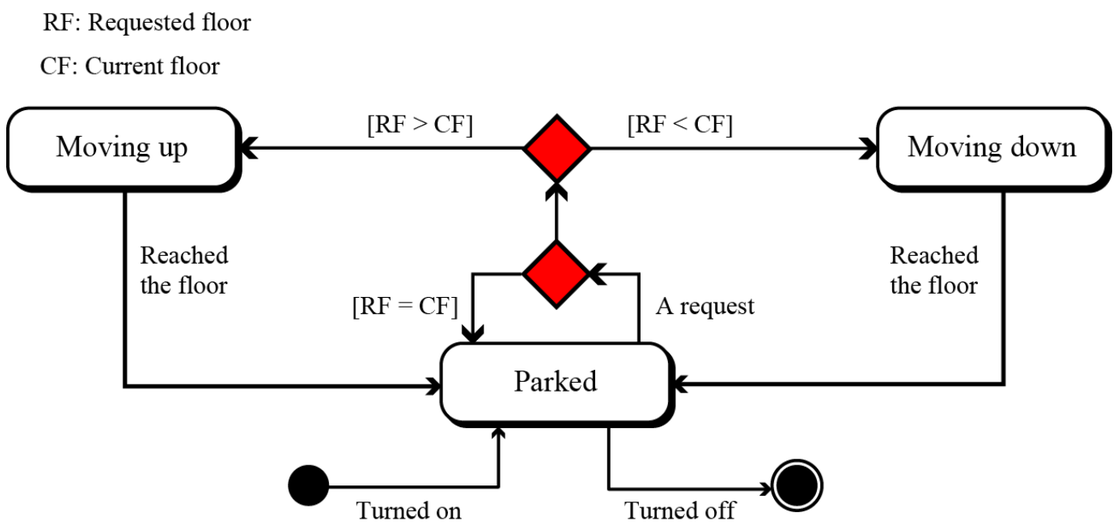
Entity-relationship diagrams

Another modeling tool used during the analysis phase is the entity-relationship diagram. Since this diagram is also used in database design, we discuss it in Chapter 12.

State diagrams

State diagrams (see Appendix B) provide another useful tool that is normally used when the state of the entities in the system will change in response to events. As an example of a state diagram, we show the operation of a one-passenger elevator. When a floor button is pushed, the elevator moves in the requested direction. It does not respond to any other request until it reaches its destination.

Figure 10.5 An example of a state diagram

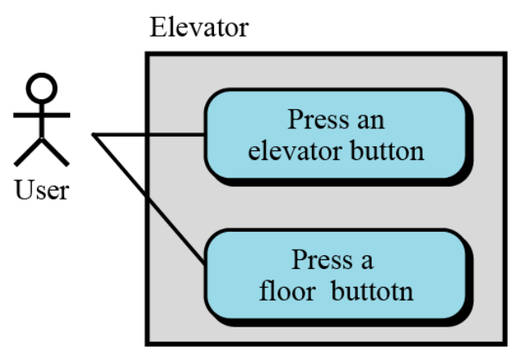


Object-oriented analysis is the analysis process used if the implementation uses an object-oriented language. The specification document in this case may use several tools, but we discuss only a few of them here.

Use case diagrams

A use-case diagram gives the user's view of a system: it shows how users communicate with the system. A use-case diagram uses four components: system, use cases, actors, and relationships. A system, shown by a rectangle, performs a function.

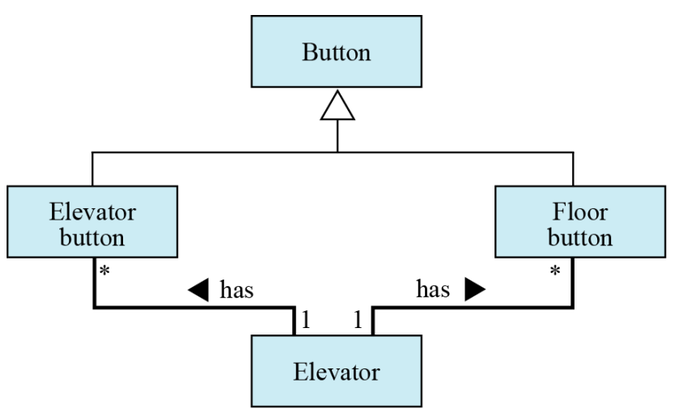
Figure 10.6 An example of use case diagram



Class diagrams

The next step in analysis is to create a class diagram for the system. For example, we can create a class diagram for our old-style elevator. To do so, we need to think about the entities involved in the system.

Figure 10.7 An example of a class diagram



State chart

After the class diagram is finalized, a state chart can be prepared for each class in the class diagram. A state chart in object-oriented analysis plays the same role as the state diagram in procedure-oriented analysis. This means that for the class diagram of Figure 10.7, we need to have a four-state chart.

10-3 DESIGN PHASE

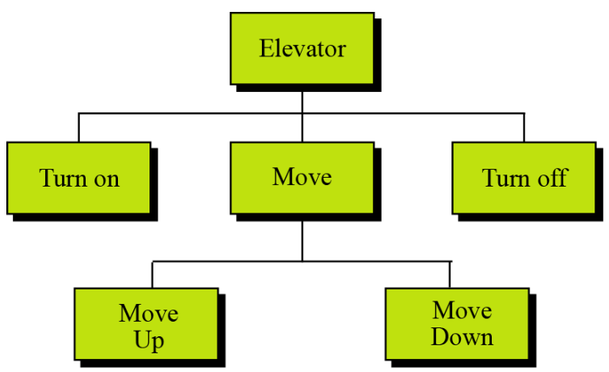
The design phase defines how the system will accomplish what was defined in the analysis phase. In the design phase, all components of the system are defined.

In procedure-oriented design we have both procedures and data to design. We discuss a category of design methods that concentrate on procedures. In procedure-oriented design, the whole system is divided into a set of procedures or modules.

Structure charts

A common tool for illustrating the relations between modules in procedure-oriented design is a structure chart. For example, the elevator system whose state diagram is shown in Figure 10.5 can be designed as a set of modules shown in the structure chart in Figure 10.8. Structure charts are discussed in Appendix D.

Figure 10.8 A structure chart



Modularity

Modularity means breaking a large project into smaller parts that can be understood and handled easily. In other words, modularity means dividing a large task into small tasks that can communicate with each other. The structure chart discussed in the previous section shows the modularity in the elevator system. There are two main concerns when a system is divided into modules: coupling and cohesion.

Coupling is a measure of how tightly two modules are bound to each other.

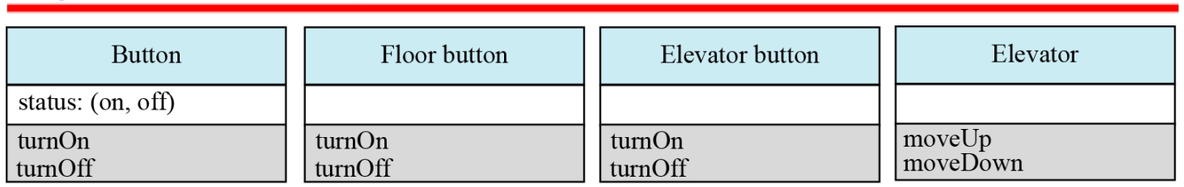
Coupling between modules in a software system must be minimized.

Another issue in modularity is cohesion. Cohesion is a measure of how closely the modules in a system are related. We need to have maximum possible cohesion between modules in a software system.

Cohesion between modules in a software system must be maximized.

In object-oriented design the design phase continues by elaborating the details of classes. As we mentioned in Chapter 9, a class is made of a set of variables (attributes) and a set of methods. The object-oriented design phase lists details of these attributes and methods. Figure 10.9 shows an example of the details of our four classes used in the design of the old-style elevator.

Figure 10.9 An example of classes with attributes and methods



10-4 IMPLEMENTATION PHASE

In the waterfall model, after the design phase is completed, the implementation phase can start. In this phase the programmers write the code for the modules in procedure-oriented design, or write the program units to implement classes in object-oriented design. There are several issues we need to mention in each case.

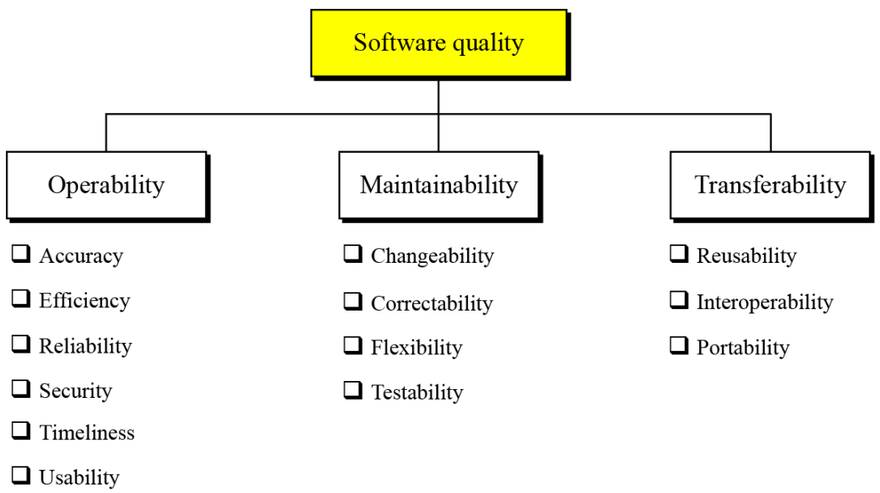
In a procedure-oriented development, the project team needs to choose a language or a set of languages from among the procedural languages discussed in Chapter 10. Although some languages like C++ are considered to be both a procedural and an object-oriented language—normally an implementation uses a purely procedural language such as C. In the object-oriented case, both C++ and Java are common.

The quality of software created at the implementation phase is a very important issue. A software system of high quality is one that satisfies the user's requirements, meets the operating standards of the organization, and runs efficiently on the hardware for which it was developed. However, if we want to achieve a software system of high quality, we must be able to define some attributes of quality.

Software quality factors

Software quality can be divided into three broad measures: operability, maintainability, and transferability. Each of these measures can be further broken down as shown in Figure 10.10.

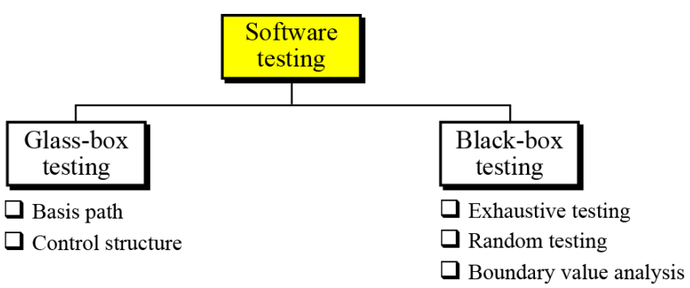
Figure 10.10 Quality factors



10-5 TESTING PHASE

The goal of the testing phase is to find errors, which means that a good testing strategy is the one that finds most errors. There are two types of testing: glass-box and black-box (Figure 10.11).

Figure 10.11 Software testing



10.5.1 Glass-box testing

Glass-box testing (or white-box testing) is based on knowing the internal structure of the software. The testing goal is to check to determine whether all components of the software do what they are designed for. Glass-box testing assumes that the tester knows everything about the software. In this case, the software is like a glass box in which everything inside the box is visible. Glass-box testing is done by the software engineer or a dedicated team.

- All independent paths in every module are tested at least once.
- All the decision constructs (two-way and multiway) are tested on each branch.
- Each loop construct is tested.
- All data structures are tested.

Several testing methodologies have been designed in the past. We briefly discuss two of them: basis path testing and control structure testing.

Basis path testing

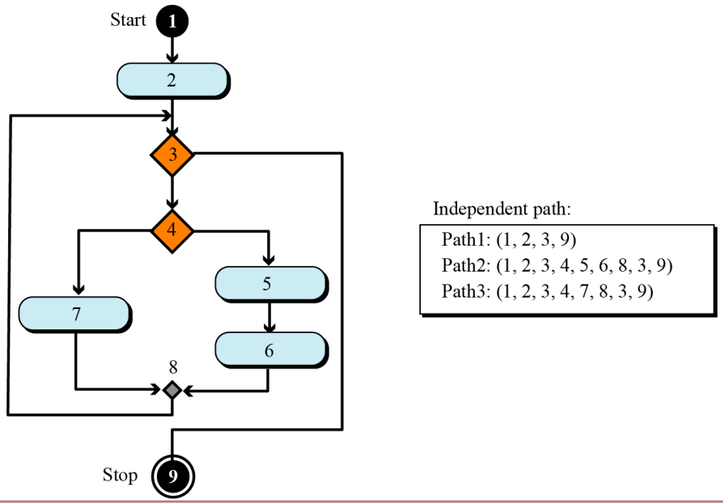
Basis path testing was proposed by Tom McCabe. This method creates a set of test cases that executes every statement in the software at least once.

Basis path testing is a method in which each statement in the software is executed at least once.

Example 10.1

To give the idea of basis path testing and finding the independent paths in part of a program, assume that a system is made up of only one program and that the program is only a single loop with the UML diagram shown in Figure 10.12.

Figure 10.12 An example of basis path testing



Independent path:

Path1: (1, 2, 3, 9)
Path2: (1, 2, 3, 4, 5, 6, 8, 3, 9)
Path3: (1, 2, 3, 4, 7, 8, 3, 9)

Control structure testing

Control structure testing is more comprehensive than basis path testing and includes it. This method uses different categories of tests that are listed below.

Black box testing gets its name from the concept of testing software without knowing what is inside it and without knowing how it works. In other words, the software is like a black box into which the tester cannot see. Black-box testing tests the functionality of the software in terms of what the software is supposed to accomplish, such as its inputs and outputs. Several methods are used in black-box testing, discussed below.

Exhaustive testing

The best black-box test method is to test the software for all possible values in the input domain. However, in complex software the input domain is so huge that it is often impractical to do so.

Random testing

In random testing, a subset of values in the input domain is selected for testing. It is very important that the subset be chosen in such a way that the values are distributed over the domain input. The use of random number generators can be very helpful in this case.

Boundary-value testing

Errors often happen when boundary values are encountered. For example, if a module defines that one of its inputs must be greater than or equal to 100, it is very important that module be tested for the boundary value 100. If the module fails at this boundary value, it is possible that some condition in the module's code such as $x \geq 100$ is written as $x > 100$.

10-6 DOCUMENTATION

For software to be used properly and maintained efficiently, documentation is needed. Usually, three separate sets of documentation are prepared for software: user documentation, system documentation, and technical documentation.

Documentation is an ongoing process.

To run the software system properly, the users need documentation, traditionally called a *user guide*, that shows how to use the software step by step. User guides usually contains a tutorial section to guide the user through each feature of the software.

A good user guide can be a very powerful marketing tool: the importance of user documentation in marketing cannot be overemphasized. User guides should be written for both the novice and the expert users, and a software system with good user documentation will definitely increase sales.

System documentation defines the software itself. It should be written so that the software can be maintained and modified by people other than the original developers. System documentation should exist for all four phases of system development.

Technical documentation describes the installation and the servicing of the software system. Installation documentation defines how the software should be installed on each computer, for example, servers and clients. Service documentation defines how the system should be maintained and updated if necessary.