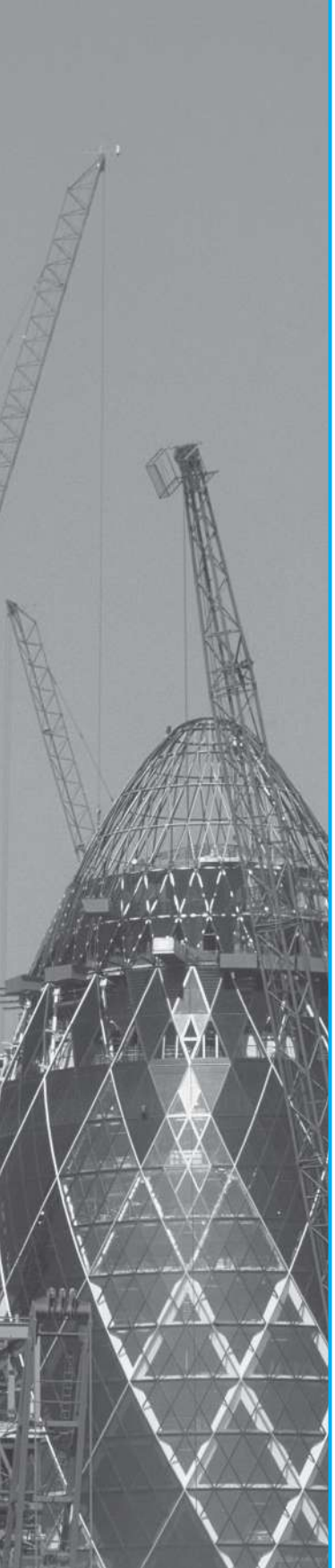


7 Design and implementation

Objectives

The objectives of this chapter are to introduce object-oriented software design using the UML and highlight important implementation concerns. When you have read this chapter, you will:

- understand the most important activities in a general, object-oriented design process;
- understand some of the different models that may be used to document an object-oriented design;
- know about the idea of design patterns and how these are a way of reusing design knowledge and experience;
- have been introduced to key issues that have to be considered when implementing software, including software reuse and open-source development.

Contents

Software design and implementation is the stage in the software engineering process at which an executable software system is developed. For some simple systems, software engineering means software design and implementation and all other software engineering activities are merged with this process. However, for large systems, software design and implementation is only one of a number of software engineering processes (requirements engineering, verification and validation, etc.).

Software design and implementation activities are invariably interleaved. Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements. Implementation is the process of realizing the design as a program. Sometimes there is a separate design stage, and this design is modeled and documented. At other times, a design is in the programmer's head or roughly sketched on a whiteboard or sheets of paper. Design is about how to solve a problem, so there is always a design process. However, it isn't always necessary or appropriate to describe the design in detail using the UML or other design description language.

Design and implementation are closely linked, and you should normally take implementation issues into account when developing a design. For example, using the UML to document a design may be the right thing to do if you are programming in an object-oriented language such as Java or C#. It is less useful, I think, if you are developing using a dynamically typed language like Python. There is no point in using the UML if you are implementing your system by configuring an off-the-shelf package. As I discussed in Chapter 3, agile methods usually work from informal sketches of the design and leave design decisions to programmers.

One of the most important implementation decisions that has to be made at an early stage of a software project is whether to build or to buy the application software. For many types of application, it is now possible to buy off-the-shelf application systems that can be adapted and tailored to the users' requirements. For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It is usually cheaper and faster to use this approach rather than developing a new system in a conventional programming language.

When you develop an application system by reusing an off-the-shelf product, the design process focuses on how to configure the system product to meet the application requirements. You don't develop design models of the system, such as models of the system objects and their interactions. I discuss this reuse-based approach to development in Chapter 15.

I assume that most readers of this book have had experience of program design and implementation. This is something that you acquire as you learn to program and master the elements of a programming language like Java or Python. You will have probably learned about good programming practice in the programming languages that you have studied, as well as how to debug programs that you have developed. Therefore, I don't cover programming topics here. Instead, this chapter has two aims:

1. To show how system modeling and architectural design (covered in Chapters 5 and 6) are put into practice in developing an object-oriented software design.

2. To introduce important implementation issues that are not usually covered in programming books. These include software reuse, configuration management and open-source development.

As there are a vast number of different development platforms, the chapter is not biased toward any particular programming language or implementation technology. Therefore, I have presented all examples using the UML rather than a programming language such as Java or Python.

7.1 Object-oriented design using the UML

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

To develop a system design from concept to detailed, object-oriented design, you need to:

1. Understand and define the context and the external interactions with the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify interfaces.

Like all creative activities, design is not a clear-cut, sequential process. You develop a design by getting ideas, proposing solutions, and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process. Sometimes you use notations, such as the UML, precisely to clarify aspects of the design; at other times, notations are used informally to stimulate discussions.

I explain object-oriented software design by developing a design for part of the embedded software for the wilderness weather station that I introduced in Chapter 1. Wilderness weather stations are deployed in remote areas. Each weather station

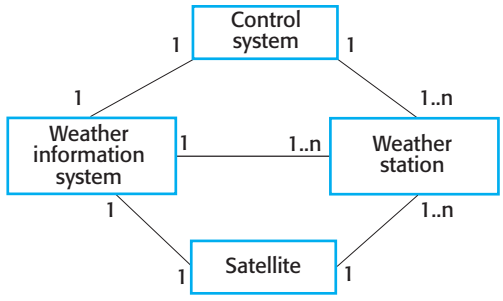


Figure 7.1 System context for the weather station

records local weather information and periodically transfers this to a weather information system, using a satellite link.

7.1.1 System context and interactions

The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. As I discussed in Chapter 5, understanding the context also lets you establish the boundaries of the system.

Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems. In this case, you need to decide how functionality is distributed between the control system for all of the weather stations and the embedded software in the weather station itself.

System context models and interaction models present complementary views of the relationships between a system and its environment:

1. A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
2. An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations. Figure 7.1 shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system.

When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail. One way to do this is to use a use case model. As I discussed in Chapters 4 and 5, each use case represents

Weather station use cases

Report weather–send weather data to the weather information system

Report status–send status information to the weather information system

Restart–if the weather station is shut down, restart the system

Shutdown–shut down the weather station

Reconfigure–reconfigure the weather station software

Powersave–put the weather station into power-saving mode

Remote control–send control commands to any weather station subsystem

http://software-engineering-book.com/web/ws-use-cases/

an interaction with the system. Each possible interaction is named in an ellipse, and the external entity involved in the interaction is represented by a stick figure.

The use case model for the weather station is shown in Figure 7.2. This shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware. Other interactions are with a control system that can issue specific weather station control commands. The stick figure is used in the UML to represent other systems as well as human users.

Each of these use cases should be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do. I use a standard format for this description that clearly identifies what information is exchanged, how the interaction is initiated, and so on. As I explain in Chapter 21, embedded systems are often modeled by describing

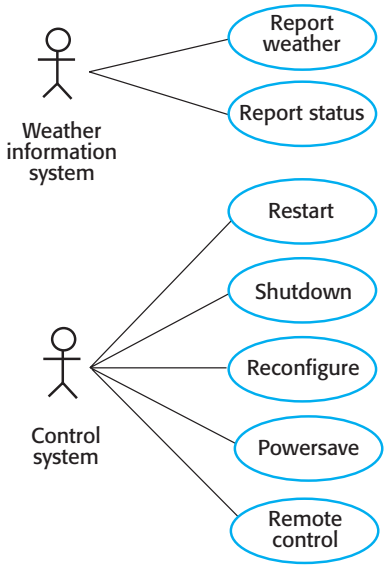


Figure 7.2 Weather station use cases

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

Figure 7.3 Use case description—Report weather

how they respond to internal or external stimuli. Therefore, the stimuli and associated responses should be listed in the description. Figure 7.3 shows the description of the Report weather use case from Figure 7.2 that is based on this approach.

7.1.2 Architectural design

Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture. Of course, you need to combine this knowledge with your general knowledge of the principles of architectural design and with more detailed domain knowledge. You identify the major components that make up the system and their interactions. You may then design the system organization using an architectural pattern such as a layered or client–server model.

The high-level architectural design for the weather station software is shown in Figure 7.4. The weather station is composed of independent subsystems that communicate

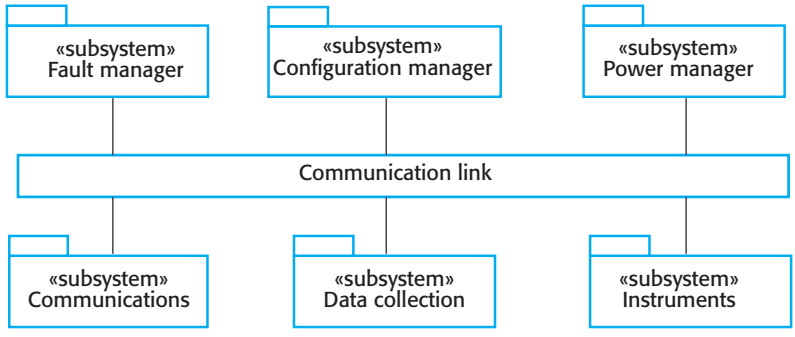


Figure 7.4 High-level architecture of weather station

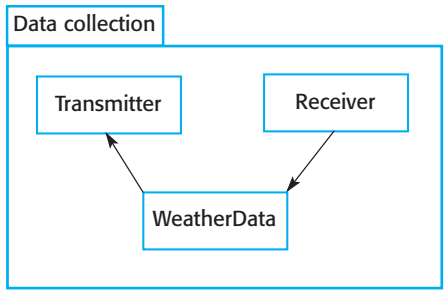


Figure 7.5 Architecture of data collection system

by broadcasting messages on a common infrastructure, shown as Communication link in Figure 7.4. Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them. This "listener model" is a commonly used architectural style for distributed systems.

When the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way. The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

Figure 7.5 shows the architecture of the data collection subsystem, which is included in Figure 7.4. The Transmitter and Receiver objects are concerned with managing communications, and the WeatherData object encapsulates the information that is collected from the instruments and transmitted to the weather information system. This arrangement follows the producer–consumer pattern, discussed in Chapter 21.

7.1.3 Object class identification

By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing. As your understanding of the design develops, you refine these ideas about the system objects. The use case description helps to identify objects and operations in the system. From the description of the Report weather use case, it is obvious that you will need to implement objects representing the instruments that collect weather data and an object representing the summary of the weather data. You also usually need a high-level system object or objects that encapsulate the system interactions defined in the use cases. With these objects in mind, you can start to identify the general object classes in the system.

As object-oriented design evolved in the 1980s, various ways of identifying object classes in object-oriented systems were suggested:

1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs (Abbott 1983).

2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations

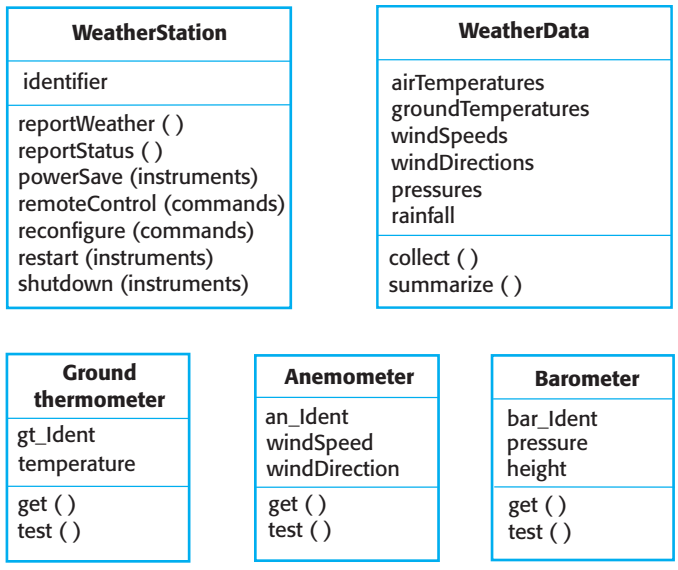


Figure 7.6 Weather station objects

such as offices, organizational units such as companies, and so on (Wirfs-Brock, Wilkerson, and Weiner 1990).

3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations (Beck and Cunningham 1989).

In practice, you have to use several knowledge sources to discover object classes. Object classes, attributes, and operations that are initially identified from the informal system description can be a starting point for the design. Information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information can be collected from requirements documents, discussions with users, or analyses of existing systems. As well as the objects representing entities external to the system, you may also have to design "implementation objects" that are used to provide general services such as searching and validity checking.

In the wilderness weather station, object identification is based on the tangible hardware in the system. I don't have space to include all the system objects here, but I have shown five object classes in Figure 7.6. The **Ground thermometer**, **Anemometer**, and **Barometer** objects are application domain objects, and the **WeatherStation** and **WeatherData** objects have been identified from the system description and the scenario (use case) description:

1. The **WeatherStation** object class provides the basic interface of the weather station with its environment. Its operations are based on the interactions shown in Figure 7.3. I use a single object class, and it includes all of these interactions. Alternatively, you could design the system interface as several different classes, with one class per interaction.

2. The WeatherData object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.

3. The Ground thermometer, Anemometer, and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the WeatherData object on request.

You use knowledge of the application domain to identify other objects, attributes. and services:

1. Weather stations are often located in remote places and include various instruments that sometimes go wrong. Instrument failures should be reported automatically. This implies that you need attributes and operations to check the correct functioning of the instruments.

2. There are many remote weather stations, so each weather station should have its own identifier so that it can be uniquely identified in communications.

3. As weather stations are installed at different times, the types of instrument may be different. Therefore, each instrument should also be uniquely identified, and a database of instrument information should be maintained.

At this stage in the design process, you should focus on the objects themselves, without thinking about how these objects might be implemented. Once you have identified the objects, you then refine the object design. You look for common features and then design the inheritance hierarchy for the system. For example, you may identify an Instrument superclass, which defines the common features of all instruments, such as an identifier, and get and test operations. You may also add new attributes and operations to the superclass, such as an attribute that records how often data should be collected.

7.1.4 Design models

Design or system models, as I discussed in Chapter 5, show the objects or object classes in a system. They also show the associations and relationships between these entities. These models are the bridge between the system requirements and the implementation of a system. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.

The level of detail that you need in a design model depends on the design process used. Where there are close links between requirements engineers, designers and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. Similarly, if agile development is used, outline design models on a whiteboard may be all that is required.

However, if a plan-based development process is used, you may need more detailed models. When the links between requirements engineers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then precise design descriptions are needed for communication. Detailed models, derived from the high-level abstract models, are used so that all team members have a common understanding of the design.

An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. A sequential data-processing system is quite different from an embedded real-time system, so you need to use different types of design models. The UML supports 13 different types of models, but, as I discussed in Chapter 5, many of these models are not widely used. Minimizing the number of models that are produced reduces the costs of the design and the time required to complete the design process.

When you use the UML to develop a design, you should develop two kinds of design model:

1. *Structural models,* which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
2. *Dynamic models,* which describe the dynamic structure of the system and show the expected runtime interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions.

I think three UML model types are particularly useful for adding detail to use case and architectural models:

1. *Subsystem models,* which show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are structural models.
2. *Sequence models,* which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. *State machine models,* which show how objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

A subsystem model is a useful static model that shows how a design is organized into logically related groups of objects. I have already shown this type of model in Figure 7.4 to present the subsystems in the weather mapping system. As well as subsystem models, you may also design detailed object models, showing the objects in the systems and their associations (inheritance, generalization, aggregation, etc.). However, there is a danger

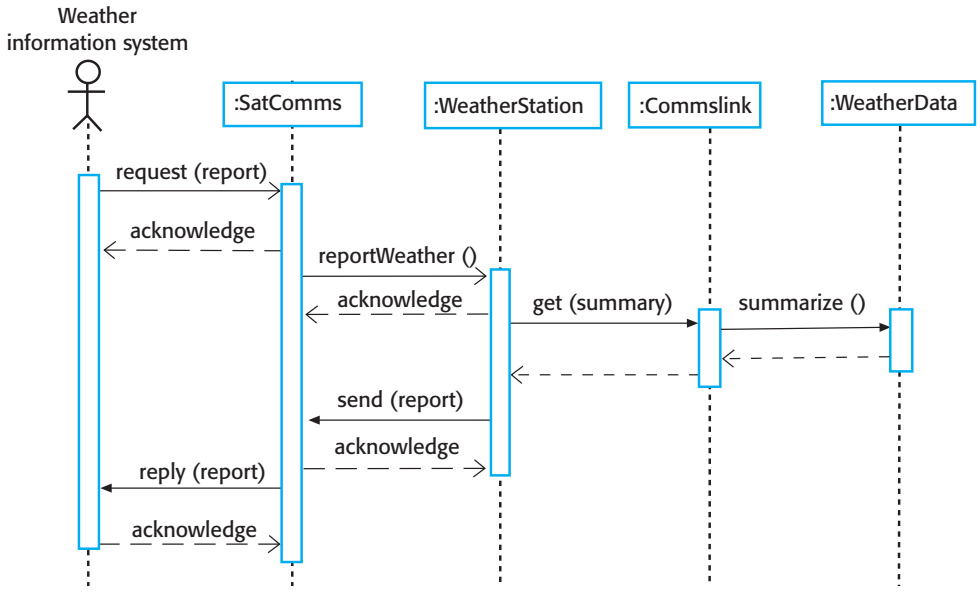


Figure 7.7 Sequence diagram describing data collection

in doing too much modeling. You should not make detailed decisions about the implementation that are really best left until the system is implemented.

Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place. When documenting a design, you should produce a sequence model for each significant interaction. If you have developed a use case model, then there should be a sequence model for each use case that you have identified.

Figure 7.7 is an example of a sequence model, shown as a UML sequence diagram. This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:

1. The SatComms object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.
2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.
3. WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.
4. Commslink calls the summarize method in the object WeatherData and waits for a reply.

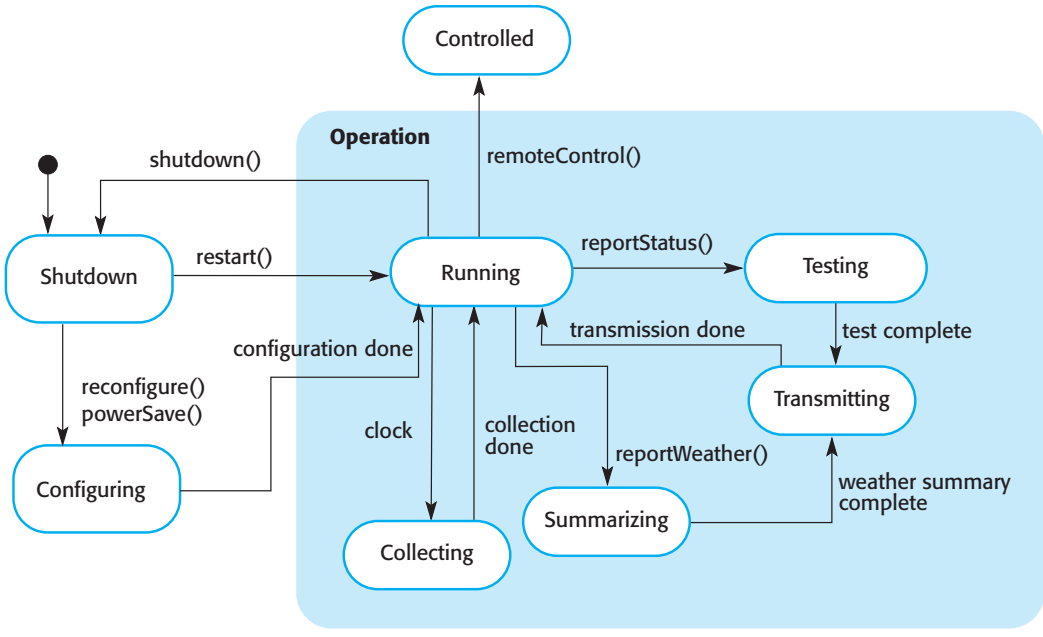


Figure 7.8 Weather station state diagram

5. The weather data summary is computed and returned to **WeatherStation** via the **Commslink** object.

6. **WeatherStation** then calls the **SatComms** object to transmit the summarized data to the weather information system, through the satellite communications system.

The **SatComms** and **WeatherStation** objects may be implemented as concurrent processes, whose execution can be suspended and resumed. The **SatComms** object instance listens for messages from the external system, decodes these messages, and initiates weather station operations.

Sequence diagrams are used to model the combined behavior of a group of objects, but you may also want to summarize the behavior of an object or a subsystem in response to messages and events. To do this, you can use a state machine model that shows how the object instance changes state depending on the messages that it receives. As I discuss in Chapter 5, the UML includes state diagrams to describe state machine models.

Figure 7.8 is a state diagram for the weather station system that shows how it responds to requests for various services.

You can read this diagram as follows:

1. If the system state is **Shutdown**, then it can respond to a **restart**(), a **reconfigure**() or a **powerSave**() message. The unlabeled arrow with the black blob indicates that the **Shutdown** state is the initial state. A **restart**() message causes a transition to normal operation. Both the **powerSave**() and **reconfigure**() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is allowed only if the system has been shut down.

2. In the Running state, the system expects further messages. If a shutdown() message is received, the object returns to the shutdown state.
3. If a reportWeather() message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted to the remote system. It then returns to the Running state.
4. If a signal from the clock is received, the system moves to the Collecting state, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.
5. If a remoteControl() message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

State diagrams are useful high-level models of a system or an object's operation. However, you don't need a state diagram for all of the objects in the system. Many system objects in a system are simple, and their operation can be easily described without a state model.

7.1.5 Interface specification

An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and subsystems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.

Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section, and the UML stereotype «interface» should be included in the name part. The semantics of the interface may be defined using the object constraint language (OCL). I discuss the use of the OCL in Chapter 16, where I explain how it can be used to describe the semantics of components.

You should not include details of the data representation in an interface design, as attributes are not defined in an interface specification. However, you should include operations to access and update data. As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack. By contrast, you should normally expose the attributes in an object model, as this is the clearest way of describing the essential characteristics of the objects.

There is not a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where interfaces are declared separately from objects and objects "implement" interfaces. Equally, a group of objects may all be accessed through a single interface.

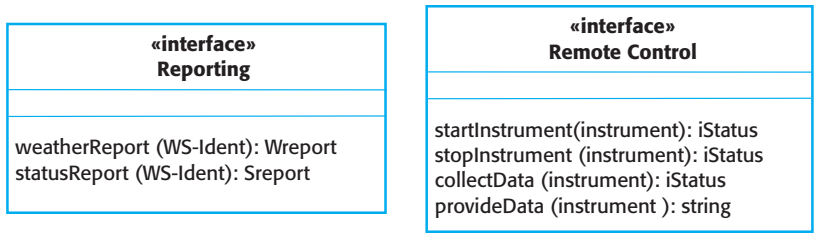


Figure 7.9 Weather station interfaces

Figure 7.9 shows two interfaces that may be defined for the weather station. The left-hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports. These map directly to operations in the WeatherStation object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object. In this case, the individual operations are encoded in the command string associated with the remoteControl method, shown in Figure 7.6.

7.2 Design patterns

Design patterns were derived from ideas put forward by Christopher Alexander (Alexander 1979), who suggested that there were certain common patterns of building design that were inherently pleasing and effective. The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.

A quote from the Hillside Group website (hillside.net/patterns/), which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

> *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience*[†].

Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used. This is particularly true for the best known design patterns that were originally described by the "Gang of Four" in their patterns book, published in 1995 (Gamma et al. 1995). Other important pattern descriptions are those published in a series of books by authors from Siemens, a large European technology company (Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; Buschmann, Henney, and Schmidt 2007a, 2007b).

Patterns are a way of reusing the knowledge and experience of other designers. Design patterns are usually associated with object-oriented design. Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in a pattern is

[†]The HIllside Group: hillside.net/patterns

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.

The UML model of the pattern is shown in Figure 7.12.

Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Figure 7.10 The Observer pattern

one that is equally applicable to any kind of software design. For instance, you could have configuration patterns for instantiating reusable application systems.

The Gang of Four defined the four essential elements of design patterns in their book on patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

Gamma and his co-authors break down the problem description into motivation (a description of why the pattern is useful) and applicability (a description of situations in which the pattern may be used). Under the description of the solution, they describe the pattern structure, participants, collaborations, and implementation.

To illustrate pattern description, I use the Observer pattern, taken from the Gang of Four's patterns book. This is shown in Figure 7.10. In my description, I use the

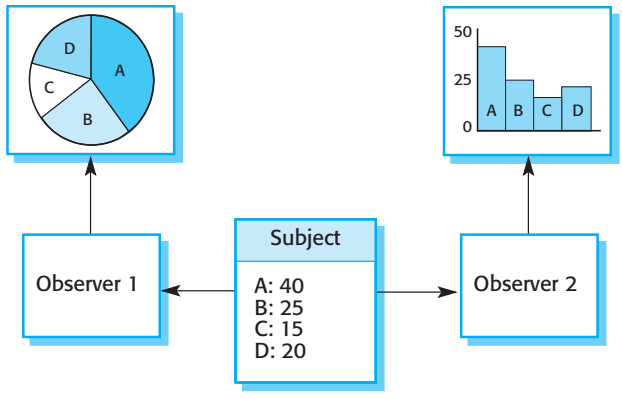


Figure 7.11 Multiple displays

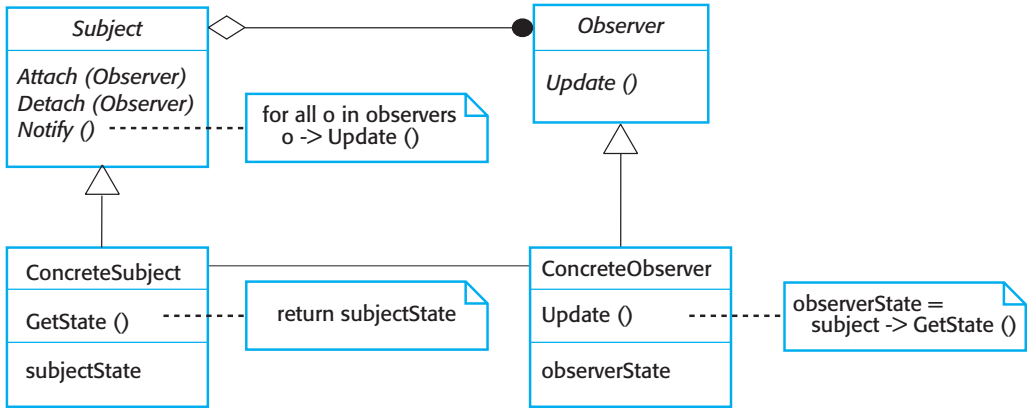
four essential description elements and also include a brief statement of what the pattern can do. This pattern can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. This is illustrated in Figure 7.11, which shows two different graphical presentations of the same dataset.

Graphical representations are normally used to illustrate the object classes in patterns and their relationships. These supplement the pattern description and add detail to the solution description. Figure 7.12 is the representation in UML of the Observer pattern.

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied. Examples of such problems, documented in the Gang of Four's original patterns book, include:

1. Tell several objects that the state of some other object has changed (Observer pattern).
2. Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).

Figure 7.12 A UML model of the Observer pattern



3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
4. Allow for the possibility of extending the functionality of an existing class at runtime (Decorator pattern).

Patterns support high-level, concept reuse. When you try to reuse executable components you are inevitably constrained by detailed design decisions that have been made by the implementers of these components. These range from the particular algorithms that have been used to implement the components to the objects and types in the component interfaces. When these design decisions conflict with your requirements, reusing the component is either impossible or introduces inefficiencies into your system. Using patterns means that you reuse the ideas but can adapt the implementation to suit the system you are developing.

When you start designing a system, it can be difficult to know, in advance, if you will need a particular pattern. Therefore, using patterns in a design process often involves developing a design, experiencing a problem, and then recognizing that a pattern can be used. This is certainly possible if you focus on the 23 general-purpose patterns documented in the original patterns book. However, if your problem is a different one, you may find it difficult to find an appropriate pattern among the hundreds of different patterns that have been proposed.

Patterns are a great idea, but you need experience of software design to use them effectively. You have to recognize situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

7.3 Implementation issues

Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system. A critical stage of this process is, of course, system implementation, where you create an executable version of the software. Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

I assume that most readers of this book will understand programming principles and will have some programming experience. As this chapter is intended to offer a language-independent approach, I haven't focused on issues of good programming practice as language-specific examples need to be used. Instead, I introduce some aspects of implementation that are particularly important to software engineering and that are often not covered in programming texts. These are:

1. *Reuse* Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.

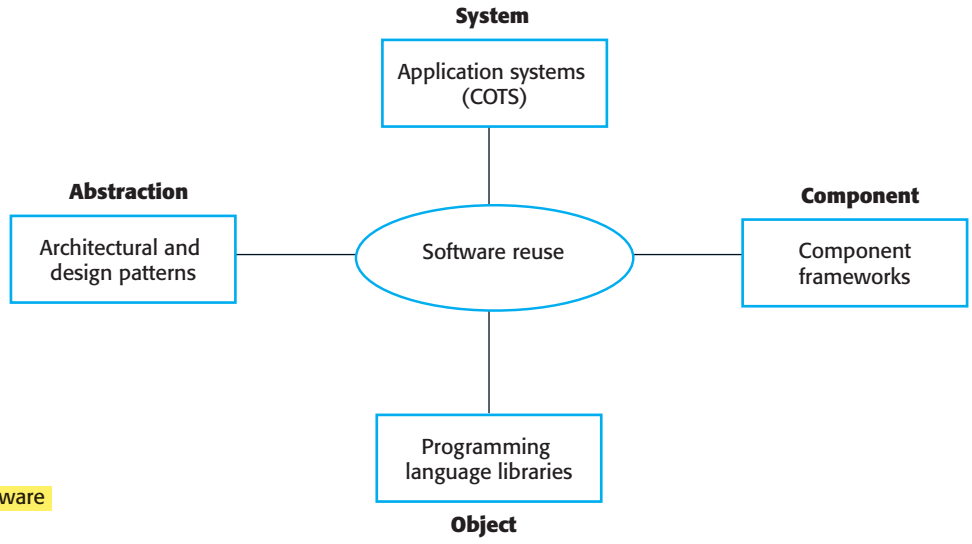


Figure 7.13 Software reuse

2. *Configuration management* During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.

3. *Host-target development* Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type, but often they are completely different.

7.3.1 Reuse

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries. However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Consequently, an approach to development based on the reuse of existing software is now the norm for many types of system development. A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels, as shown in Figure 7.13:

1. *The abstraction level* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns (covered in Chapter 6) are ways of representing abstract knowledge for reuse.

2. *The object level* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process email messages in a Java program, you may use objects and methods from a JavaMail library.

3. *The component level* Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.

4. *The system level* At this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic application systems systems are adapted and reused. Sometimes this approach may involve integrating several application systems to create a new system.

By reusing existing software, you can develop new systems more quickly, with fewer development risks and at lower cost. As the reused software has been tested in other applications, it should be more reliable than new software. However, there are costs associated with reuse:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment.

2. Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.

3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.

4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

How to reuse existing knowledge and software should be the first thing you should think about when starting a software development project. You should consider the

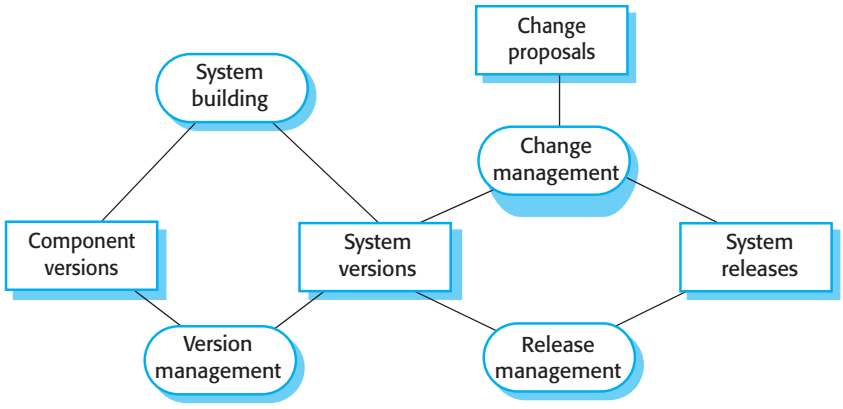


Figure 7.14 Configuration management

possibilities of reuse before designing the software in detail, as you may wish to adapt your design to reuse existing software assets. As I discussed in Chapter 2, in a reuse-oriented development process, you search for reusable elements, then modify your requirements and design to make the best use of these.

Because of the importance of reuse in modern software engineering, I devote several chapters in Part 3 of this book to this topic (Chapters 15, 16, and 18).

7.3.2 Configuration management

In software development, change happens all the time, so change management is absolutely essential. When several people are involved in developing a software system, you have to make sure that team members don't interfere with each other's work. That is, if two people are working on a component, their changes have to be coordinated. Otherwise, one programmer may make changes and overwrite the other's work. You also have to ensure that everyone can access the most up-to-date versions of software components; otherwise developers may redo work that has already been done. When something goes wrong with a new version of a system, you have to be able to go back to a working version of the system or component.

Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. As shown in Figure 7.14, there are four fundamental configuration management activities:

1. *Version management,* where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer from overwriting code that has been submitted to the system by someone else.
2. *System integration*, where support is provided to help developers define what versions of components are used to create each version of a system. This

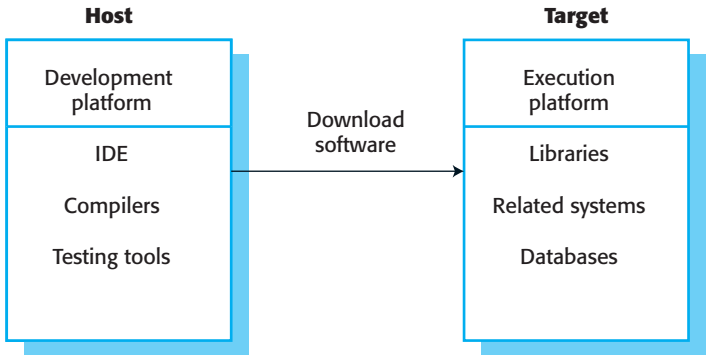


Figure 7.15 Host-target development

description is then used to build a system automatically by compiling and linking the required components.

3. *Problem tracking,* where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

4. *Release management,* where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Software configuration management tools support each of the above activities. These tools are usually installed in an integrated development environment, such as Eclipse. Version management may be supported using a version management system such as Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) or Git (Loeliger and McCullough 2012), which can support multi-site, multi-team development. System integration support may be built into the language or rely on a separate toolset such as the GNU build system. Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed. A comprehensive set of tools built around the Git system is available at Github (http://github.com).

Because of its importance in professional software engineering, I discuss change and configuration management in more detail in Chapter 25.

7.3.3 Host-target development

Most professional software development is based on a host-target model (Figure 7.15). Software is developed on one computer (the host) but runs on a separate machine (the target). More generally, we can talk about a development platform (host) and an execution platform (target). A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.

Sometimes, the development platform and execution platform are the same, making it possible to develop the software and test it on the same machine. Therefore, if you develop in Java, the target environment is the Java Virtual Machine. In principle, this is the same on every computer, so programs should be portable from one machine to another. However, particularly for embedded systems and mobile systems, the development and the execution platforms are different. You need to either move your developed software to the execution platform for testing or run a simulator on your development machine.

Simulators are often used when developing embedded systems. You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed. Simulators speed up the development process for embedded systems as each developer can have his or her own execution platform with no need to download the software to the target hardware. However, simulators are expensive to develop and so are usually available only for the most popular hardware architectures.

If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software. It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions. If this is the case, you need to transfer your developed code to the execution platform to test the system.

A software development platform should provide a range of tools to support software engineering processes. These may include:

1. An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.
2. A language debugging system.
3. Graphical editing tools, such as tools to edit UML models.
4. Testing tools, such as JUnit, that can automatically run a set of tests on a new version of a program.
5. Tools to support refactoring and program visualization.
6. Configuration management tools to manage source code versions and to integrate and build systems.

In addition to these standard tools, your development system may include more specialized tools such as static analyzers (discussed in Chapter 12). Normally, development environments for teams also include a shared server that runs a change and configuration management system and, perhaps, a system to support requirements management.

Software development tools are now usually installed within an integrated development environment (IDE). An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface. Generally, IDEs are created to support development in a specific programming

UML deployment diagrams

UML deployment diagrams show how software components are physically deployed on processors. That is, the deployment diagram shows the hardware and software in the system and the middleware used to connect the different components in the system. Essentially, you can think of deployment diagrams as a way of defining and documenting the target environment.

http://software-engineering-book.com/web/deployment/

language such as Java. The language IDE may be developed specially or may be an instantiation of a general-purpose IDE, with specific language-support tools.

A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together. The best-known general-purpose IDE is the Eclipse environment (http://www.eclipse.org). This environment is based on a plug-in architecture so that it can be specialized for different languages, such as Java, and application domains. Therefore, you can install Eclipse and tailor it for your specific needs by adding plug-ins. For example, you may add a set of plug-ins to support networked systems development in Java (Vogel 2013) or embedded systems engineering using C.

As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform. This is straightforward for embedded systems, where the target is usually a single computer. However, for distributed systems, you need to decide on the specific platforms where the components will be deployed. Issues that you have to consider in making this decision are:

1. *The hardware and software requirements of a component* If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

2. *The availability requirements of the system* High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

3. *Component communications* If there is a lot of intercomponent communication, it is usually best to deploy them on the same platform or on platforms that are physically close to one another. This reduces communications latency—the delay between the time that a message is sent by one component and received by another.

You can document your decisions on hardware and software deployment using UML deployment diagrams, which show how software components are distributed across hardware platforms.

If you are developing an embedded system, you may have to take into account target characteristics, such as its physical size, power capabilities, the need for real-time responses to sensor events, the physical characteristics of actuators and its real-time operating system. I discuss embedded systems engineering in Chapter 21.

7.4 Open-source development

Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process (Raymond 2001). Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish. There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code.

Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code. In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality. However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.

Open-source software is the backbone of the Internet and software engineering. The Linux operating system is the most widely used server system, as is the open-source Apache web server. Other important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system. The Android operating system is installed on millions of mobile devices. Major players in the computer industry such as IBM and Oracle, support the open-source movement and base their software on open-source products. Thousands of other, lesser-known open-source systems and components may also be used.

It is usually cheap or even free to acquire open-source software. You can normally download open-source software without charge. However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low. The other key benefit of using open-source products is that widely used open-source systems are very reliable. They have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system. Bugs are discovered and repaired more quickly than is usually possible with proprietary software.

For a company involved in software development, there are two open-source issues that have to be considered:

1. Should the product that is being developed make use of open-source components?
2. Should an open-source approach be used for its own software development?

The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

If you are developing a software product for sale, then time to market and reduced costs are critical. If you are developing software in a domain in which there are high-quality open-source systems available, you can save time and money by using these systems. However, if you are developing software to a specific set of organizational requirements, then using open-source components may not be an option. You may have to integrate your software with existing systems that are incompatible with available

open-source systems. Even then, however, it could be quicker and cheaper to modify the open-source system rather than redevelop the functionality that you need.

Many software product companies are now using an open-source approach to development, especially for specialized systems. Their business model is not reliant on selling a software product but rather on selling support for that product. They believe that involving the open-source community will allow software to be developed more cheaply and more quickly and will create a community of users for the software.

Some companies believe that adopting an open-source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model. However, if you are working in a small company and you open source your software, this may reassure customers that they will be able to support the software if your company goes out of business.

Publishing the source code of a system does not mean that people from the wider community will necessarily help with its development. Most successful open-source products have been platform products rather than application systems. There are a limited number of developers who might be interested in specialized application systems. Making a software system open source does not guarantee community involvement. There are thousands of open-source projects on Sourceforge and GitHub that have only a handful of downloads. However, if users of your software have concerns about its availability in future, making the software open source means that they can take their own copy and so be reassured that they will not lose access to it.

7.4.1 Open-source licensing

Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code. Legally, the developer of the code (either a company or an individual) owns the code. They can place restrictions on how it is used by including legally binding conditions in an open-source software license (St. Laurent 2004). Some open-source developers believe that if an open-source component is used to develop a new system, then that system should also be open source. Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed-source systems.

Most open-source licenses (Chapman 2010) are variants of one of three general models:

1. The GNU General Public License (GPL). This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.
2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open-source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.
3. The Berkley Standard Distribution (BSD) License. This is a nonreciprocal license, which means you are not obliged to re-publish any changes or modifications made to

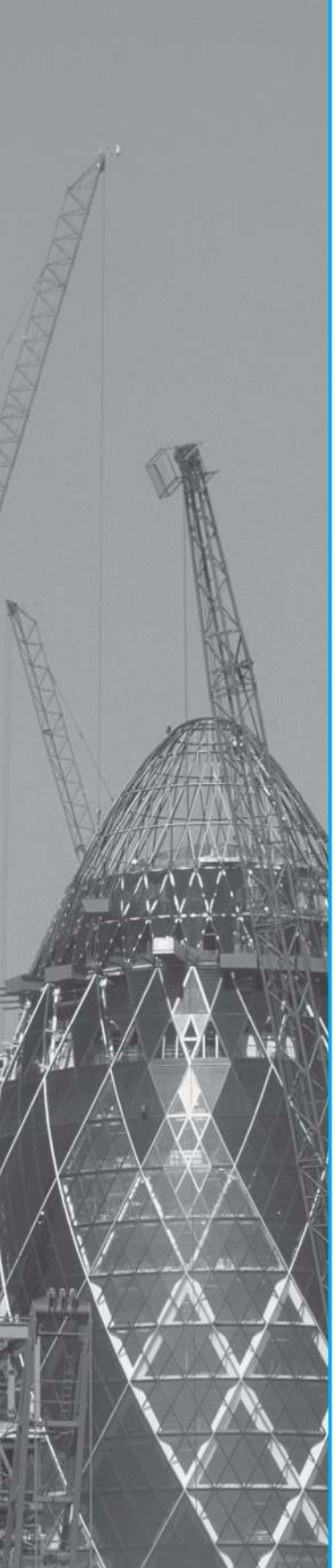
open-source code. You can include the code in proprietary systems that are sold. If you use open-source components, you must acknowledge the original creator of the code. The MIT license is a variant of the BSD license with similar conditions.

Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source. If you are trying to sell your software, you may wish to keep it secret. This means that you may wish to avoid using GPL-licensed open-source software in its development.

If you are building software that runs on an open-source platform but that does not reuse open-source components, then licenses are not a problem. However, if you embed open-source software in your software, you need processes and databases to keep track of what's been used and their license conditions. Bayersdorfer (Bayersdorfer 2007) suggests that companies managing projects that use open source should:

1. Establish a system for maintaining information about open-source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change, so you need to know the conditions that you have agreed to.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.
3. Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.
4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open-source licensing.
5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.
6. Participate in the open-source community. If you rely on open-source products, you should participate in the community and help support their development.

The open-source approach is one of several business models for software. In this model, companies release the source of their software and sell add-on services and advice in association with this. They may also sell cloud-based software services—an attractive option for users who do not have the expertise to manage their own open-source system and also specialized versions of their system for particular clients. Open-source is therefore likely to increase in importance as a way of developing and distributing software.



9 Software evolution

Objectives

The objectives of this chapter are to explain why software evolution is such an important part of software engineering and to describe the challenges of maintaining a large base of software systems, developed over many years. When you have read this chapter, you will:

- understand that software systems have to adapt and evolve if they are to remain useful and that software change and evolution should be considered as an integral part of software engineering;
- understand what is meant by legacy systems and why these systems are important to businesses;
- understand how legacy systems can be assessed to decide whether they should be scrapped, maintained, reengineered, or replaced;
- have learned about different types of software maintenance and the factors that affect the costs of making changes to legacy software systems.

Contents

Large software systems usually have a long lifetime. For example, military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more. Business systems are often more than 10 years old. Enterprise software costs a lot of money, so a company has to use a software system for many years to get a return on its investment. Successful software products and apps may have been introduced many years ago with new versions released every few years. For example, the first version of Microsoft Word was introduced in 1983, so it has been around for more than 30 years.

During their lifetime, operational software systems have to change if they are to emain useful. Business changes and changes to user expectations generate new requirements for the software. Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for changes to its hardware and software platform, and to improve its performance or other non-functional characteristics. Software products and apps have to evolve to cope with platform changes and new features introduced by their competitors. Software systems, therefore, adapt and evolve during their lifetime from initial deployment to final retirement.

Businesses have to change their software to ensure that they continue to get value from it. Their systems are critical business assets, and they have to invest in change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development. Historical data suggests that somewhere between 60% and 90% of software costs are evolution costs (Lientz and Swanson 1980; Erlikh 2000). Jones (Jones 2006) found that about 75% of development staff in the United States in 2006 were involved in software evolution and suggested that this percentage was unlikely to fall in the foreseeable future.

Software evolution is particularly expensive in enterprise systems when individual software systems are part of a broader "system of systems." In such cases, you cannot just consider the changes to one system; you also need to examine how these changes affect the broader system of systems. Changing one system may mean that other systems in its environment may also have to evolve to cope with that change.

Therefore, as well as understanding and analyzing the impact of a proposed change on the system itself, you also have to assess how this change may affect other systems in the operational environment. Hopkins and Jenkins (Hopkins and Jenkins 2008) have coined the term *brownfield software development* to describe situations in which software systems have to be developed and managed in an environment where they are dependent on other software systems.

The requirements of installed software systems change as the business and its environment change, so new releases of the systems that incorporate changes and updates are usually created at regular intervals. Software engineering is therefore a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system (Figure 9.1). You start by creating release 1 of the system. Once delivered, changes are proposed, and the development of release 2 starts almost immediately. In fact, the need for evolution may become obvious even before the system is deployed, so later releases of the software may start development before the current version has even been released.

In the last 10 years, the time between iterations of the spiral has reduced dramatically. Before the widespread use of the Internet, new versions of a software system

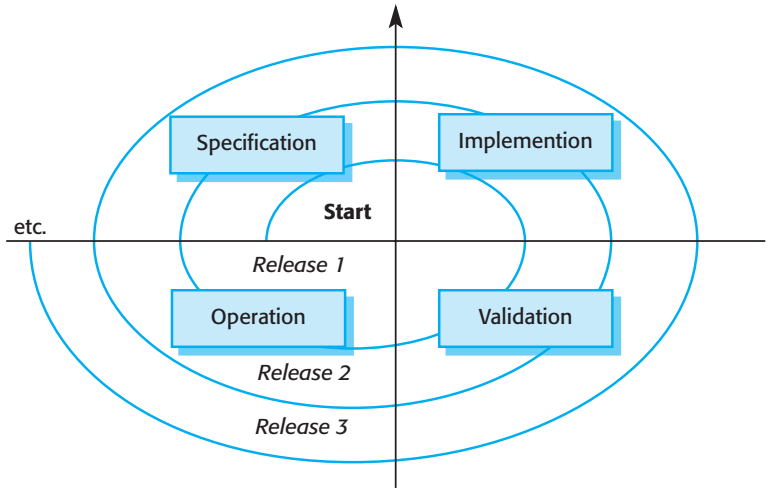


Figure 9.1 A spiral model of development and evolution

may only have been released every 2 or 3 years. Now, because of competitive pressures and the need to respond quickly to user feedback, the gap between releases of some apps and web-based systems may be weeks rather than years.

This model of software evolution is applicable when the same company is responsible for the software throughout its lifetime. There is a seamless transition from development to evolution, and the same software development methods and processes are applied throughout the lifetime of the software. Software products and apps are developed using this approach.

The evolution of custom software, however, usually follows a different model. The system customer may pay a software company to develop the software and then take over responsibility for support and evolution using its own staff. Alternatively, the software customer might issue a separate contract to a different software company for system support and evolution.

In this situation, there are likely to be discontinuities in the evolution process. Requirements and design documents may not be passed from one company to another. Companies may merge or reorganize, inherit software from other companies, and then find that this has to be changed. When the transition from development to evolution is not seamless, the process of changing the software after delivery is called software maintenance. As I discuss later in this chapter, maintenance involves extra process activities, such as program understanding, in addition to the normal activities of software development.

Rajlich and Bennett (Rajlich and Bennett 2000) propose an alternative view of the software evolution life cycle for business systems. In this model, they distinguish between evolution and servicing. Evolution is the phase in which significant changes to the software architecture and functionality are made. During servicing, the only changes that are made are relatively small but essential changes. These phases overlap with each other, as shown in Figure 9.2.

According to Rajlich and Bennett, when software is first used successfully, many changes to the requirements by stakeholders are proposed and implemented. This is

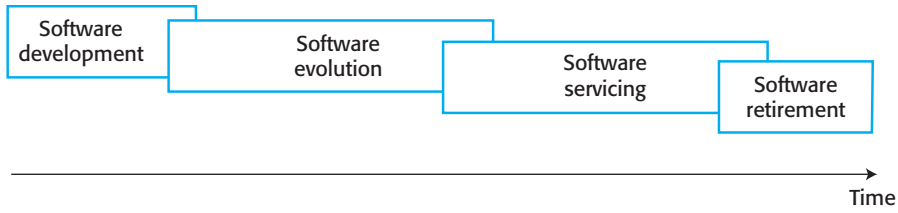


Figure 9.2 Evolution and servicing

the evolution phase. However, as the software is modified, its structure tends to degrade, and system changes become more and more expensive. This often happens after a few years of use when other environmental changes, such as hardware and operating systems, are also required. At some stage in the life cycle, the software reaches a transition point where significant changes and the implementation of new requirements become less and less cost-effective. At this stage, the software moves from evolution to servicing.

During the servicing phase, the software is still useful, but only small tactical changes are made to it. During this stage, the company is usually considering how the software can be replaced. In the final stage, the software may still be used, but only essential changes are made. Users have to work around problems that they discover. Eventually, the software is retired and taken out of use. This often incurs further costs as data is transferred from an old system to a newer replacement system.

9.1 Evolution processes

As with all software processes, there is no such thing as a standard software change or evolution process. The most appropriate evolution process for a software system depends on the type of software being maintained, the software development processes used in an organization, and the skills of the people involved. For some types of system, such as mobile apps, evolution may be an informal process, where change requests mostly come from conversations between system users and developers. For other types of systems, such as embedded critical systems, software evolution may be formalized, with structured documentation produced at each stage in the process.

Formal or informal system change proposals are the driver for system evolution in all organizations. In a change proposal, an individual or group suggests changes and updates to an existing software system. These proposals may be based on existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team. The processes of change identification and system evolution are cyclical and continue throughout the lifetime of a system (Figure 9.3).

Before a change proposal is accepted, there needs to be an analysis of the software to work out which components need to be changed. This analysis allows the cost and the impact of the change to be assessed. This is part of the general process of change management, which should also ensure that the correct versions of

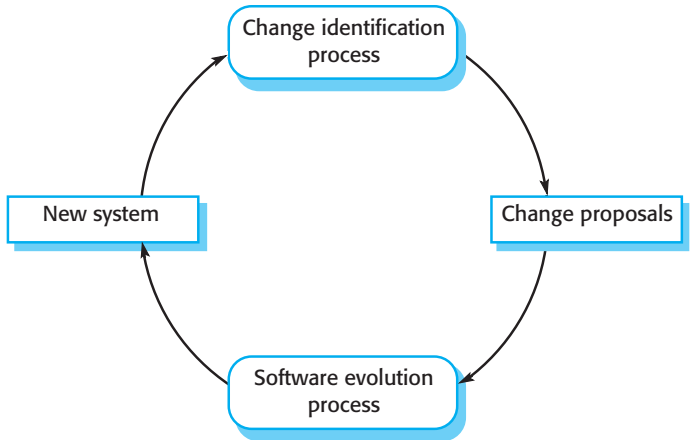


Figure 9.3 Change identification and evolution processes

components are included in each system release. I discuss change and configuration management in Chapter 25.

Figure 9.4 shows some of the activities involved in software evolution. The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.

If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

In situations where development and evolution are integrated, change implementation is simply an iteration of the development process. Revisions to the system are designed, implemented, and tested. The only difference between initial development and evolution is that customer feedback after delivery has to be considered when planning new releases of an application.

Where different teams are involved, a critical difference between development and evolution is that the first stage of change implementation requires program understanding.

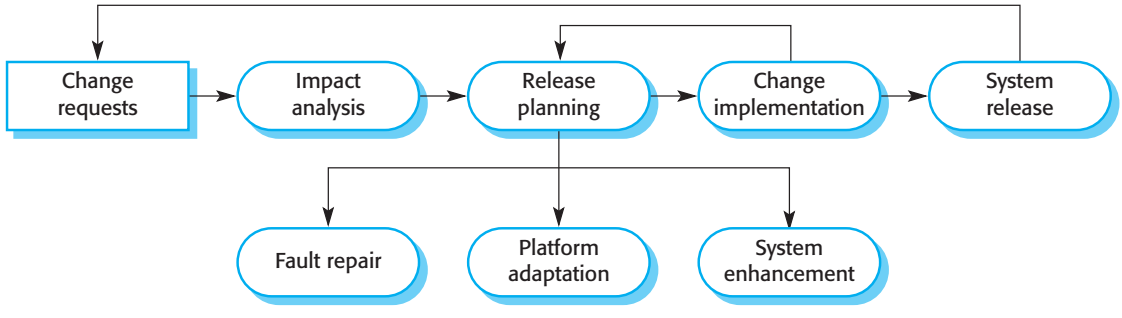


Figure 9.4 A general model of the software evolution process

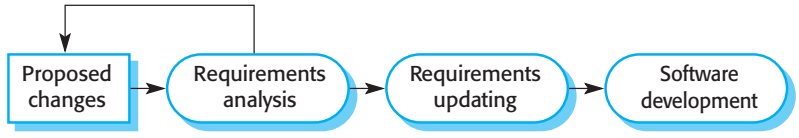


Figure 9.5 Change implementation

During the program understanding phase, new developers have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. They need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

If requirements specification and design documents are available, these should be updated during the evolution process to reflect the changes that are required (Figure 9.5). New software requirements should be written, and these should be analyzed and validated. If the design has been documented using UML models, these models should be updated. The proposed changes may be prototyped as part of the change analysis process, where you assess the implications and costs of making the change.

However, change requests sometimes relate to problems in operational systems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault is detected that has to be repaired to allow normal operation to continue or to address a serious security vulnerability.
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation.
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.

In these cases, the need to make the change quickly means that you may not be able to update all of the software documentation. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 9.6). The danger here is that the requirements, the software design, and the code can become inconsistent. While you may intend to document the change in the requirements and design, additional emergency fixes to the software may then be needed. These take priority over documentation. Eventually, the original change is forgotten, and the system documentation and code are never realigned. This problem of maintaining multiple representations of a system is one of the arguments for minimal documentation, which is fundamental to agile development processes.

Emergency system repairs have to be completed as quickly as possible. You choose a quick and workable solution rather than the best solution as far as system structure is concerned. This tends to accelerate the process of software ageing so that future changes become progressively more difficult and maintenance costs increase. Ideally, after emergency code repairs are made, the new code should be refactored

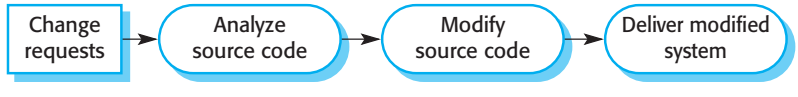


Figure 9.6 The emergency repair process

and improved to avoid program degradation. Of course, the code of the repair may be reused if possible. However, an alternative, better solution to the problem may be discovered when more time is available for analysis.

Agile methods and processes, discussed in Chapter 3, may be used for program evolution as well as program development. Because these methods are based on incremental development, making the transition from agile development to postdelivery evolution should be seamless.

However, problems may arise during the handover from a development team to a separate team responsible for system evolution. There are two potentially problematic situations:

1. Where the development team has used an agile approach but the evolution team prefers a plan-based approach. The evolution team may expect detailed documentation to support evolution, and this is rarely produced in agile processes. There may be no definitive statement of the system requirements that can be modified as changes are made to the system.

2. Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests. The code in the system may not have been refactored and simplified, as is expected in agile development. In this case, some program reengineering may be required to improve the code before it can be used in an agile development process.

Agile techniques such as test-driven development and automated regression testing are useful when system changes are made. System changes may be expressed as user stories, and customer involvement can help prioritize changes that are required in an operational system. The Scrum approach of focusing on a backlog of work to be done can help prioritize the most important system changes. In short, evolution simply involves continuing the agile development process.

Agile methods used in development may, however, have to be modified when they are used for program maintenance and evolution. It may be practically impossible to involve users in the development team as change proposals come from a wide range of stakeholders. Short development cycles may have to be interrupted to deal with emergency repairs, and the gap between releases may have to be lengthened to avoid disrupting operational processes.

9.2 Legacy systems

Large companies started computerizing their operations in the 1960s, so for the past 50 years or so, more and more software systems have been introduced. Many of these systems have been replaced (sometimes several times) as the business has changed and evolved. However, a lot of old systems are still in use and play a critical part in the running of the business. These older software systems are sometimes called legacy systems.

Legacy systems are older systems that rely on languages and technology that are no longer used for new systems development. Typically, they have been maintained over a long period, and their structure may have been degraded by the changes that have been made. Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures. It may be impossible to change to more effective business processes because the legacy software cannot be modified to support new processes.

Legacy systems are not just software systems but are broader sociotechnical systems that include hardware, software, libraries, and other supporting software and business processes. Figure 9.7 shows the logical parts of a legacy system and their relationships.

1. *System hardware* Legacy systems may have been written for hardware that is no longer available, that is expensive to maintain, and that may not be compatible with current organizational IT purchasing policies.
2. *Support software* The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.
3. *Application software* The application system that provides the business services is usually made up of a number of application programs that have been developed at different times. Some of these programs will also be part of other application software systems.
4. *Application data* These data are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent, may be duplicated in several files, and may be spread over a number of different databases.
5. *Business processes* These processes are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting an order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.
6. *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

An alternative way of looking at these components of a legacy system is as a series of layers, as shown in Figure 9.8.

Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then you should be able to make changes within a layer without affecting either of the adjacent layers. In practice, however, this simple encapsulation is an oversimplification, and changes to one layer of the system may

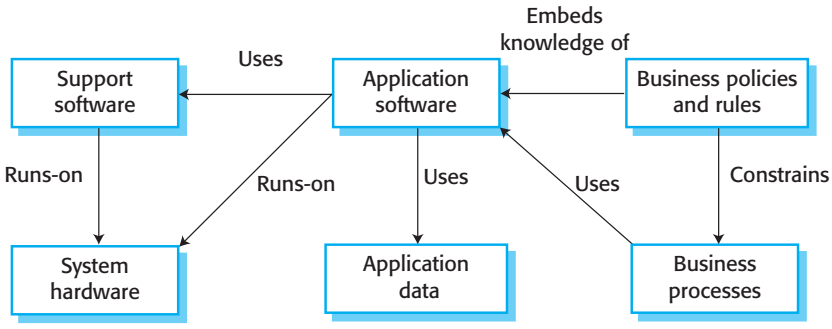


Figure 9.7 The elements of a legacy system

require consequent changes to layers that are both above and below the changed level. The reasons for this are as follows:

1. Changing one layer in the system may introduce new facilities, and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include facilities to access the data through a web browser, and business processes may be modified to take advantage of this facility.
2. Changing the software may slow the system down so that new hardware is needed to improve the system performance. The increase in performance from the new hardware may then mean that further software changes that were previously impractical become possible.
3. It is often impossible to maintain hardware interfaces, especially if new hardware is introduced. This is a particular problem in embedded systems where there is a tight coupling between software and hardware. Major changes to the application software may be required to make effective use of the new hardware.

It is difficult to know exactly how much legacy code is still in use, but, as an indicator, industry has estimated that there are more than 200 billion lines of COBOL code in current business systems. COBOL is a programming language designed for writing business systems, and it was the main business development language from the 1960s to the 1990s, particularly in the finance industry (Mitchell 2012). These programs still work effectively and efficiently, and the companies using them see no need to change them. A major problem that they face, however, is a shortage of COBOL programmers as the original developers of the system retire. Universities no longer teach COBOL, and younger software engineers are more interested in programming in modern languages.

Skill shortages are only one of the problems of maintaining business legacy systems. Other issues include security vulnerabilities because these systems were developed before the widespread use of the Internet and problems in interfacing with systems written in modern programming languages. The original software tool supplier may be out of business or may no longer maintain the support tools used to

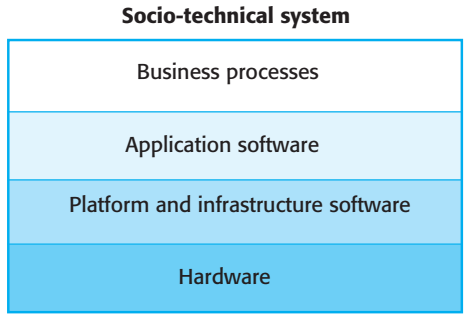


Figure 9.8 Legacy system layers

develop the system. The system hardware may be obsolete and so increasingly expensive to maintain.

Why then do businesses not simply replace these systems with more modern equivalents? The simple answer to this question is that it is too expensive and too risky to do so. If a legacy system works effectively, the costs of replacement may exceed the savings that come from the reduced support costs of a new system. Scrapping legacy systems and replacing them with more modern software open up the possibility of things going wrong and the new system failing to meet the needs of the business. Managers try to minimize those risks and therefore do not want to face the uncertainties of new software systems.

I discovered some of the problems of legacy system replacement when I was involved in analyzing a legacy system replacement project in a large organization. This enterprise used more than 150 legacy systems to run its business. It decided to replace all of these systems with a single, centrally maintained ERP system. For a number of business and technology reasons, the new system development was a failure, and it did not deliver the improvements promised. After spending more than £10 million, only a part of the new system was operational, and it worked less effectively than the systems it replaced. Users continued to use the older systems but could not integrate these with the part of the new system that had been implemented, so additional manual processing was required.

There are several reasons why it is expensive and risky to replace legacy systems with new systems:

1. There is rarely a complete specification of the legacy system. The original specification may have been lost. If a specification exists, it is unlikely that it has been updated with all of the system changes that have been made. Therefore, there is no straightforward way of specifying a new system that is functionally identical to the system that is in use.

2. Business processes and the ways in which legacy systems operate are often inextricably intertwined. These processes are likely to have evolved to take advantage of the software's services and to work around the software's shortcomings. If the system is replaced, these processes have to change with potentially unpredictable costs and consequences.

3. Important business rules may be embedded in the software and may not be documented elsewhere. A business rule is a constraint that applies to some business function, and breaking that constraint can have unpredictable consequences for the business. For example, an insurance company may have embedded its rules for assessing the risk of a policy application in its software. If these rules are not maintained, the company may accept high-risk policies that could result in expensive future claims.

4. New software development is inherently risky, so that there may be unexpected problems with a new system. It may not be delivered on time and for the price expected.

Keeping legacy systems in use avoids the risks of replacement, but making changes to existing software inevitably becomes more expensive as systems get older. Legacy software systems that are more than a few years old are particularly expensive to change:

1. The program style and usage conventions are inconsistent because different people have been responsible for system changes. This problem adds to the difficulty of understanding the system code.

2. Part or all of the system may be implemented using obsolete programming languages. It may be difficult to find people who have knowledge of these languages. Expensive outsourcing of system maintenance may therefore be required.

3. System documentation is often inadequate and out of date. In some cases, the only documentation is the system source code.

4. Many years of maintenance usually degrades the system structure, making it increasingly difficult to understand. New programs may have been added and interfaced with other parts of the system in an ad hoc way.

5. The system may have been optimized for space utilization or execution speed so that it runs effectively on older slower hardware. This normally involves using specific machine and language optimizations, and these usually lead to software that is hard to understand. This causes problems for programmers who have learned modern software engineering techniques and who don't understand the programming tricks that have been used to optimize the software.

6. The data processed by the system may be maintained in different files that have incompatible structures. There may be data duplication, and the data itself may be out of date, inaccurate, and incomplete. Several databases from different suppliers may be used.

At same stage, the costs of managing and maintaining the legacy system become so high that it has to be replaced with a new system. In the next section, I discuss a systematic decision-making approach to making such a replacement decision.

9.2.1 Legacy system management

For new software systems developed using modern software engineering processes, such as agile development and software product lines, it is possible to plan how to integrate system development and evolution. More and more companies understand that the system development process is a whole life-cycle process. Separating software development and software evolution is unhelpful and leads to higher costs. However, as I have discussed, there is still a huge number of legacy systems that are critical business systems. These have to be extended and adapted to changing e-business practices.

Most organizations have a limited budget for maintaining and upgrading their portfolio of legacy systems. They have to decide how to get the best return on their investment. This involves making a realistic assessment of their legacy systems and then deciding on the most appropriate strategy for evolving these systems. There are four strategic options:

1. *Scrap the system completely* This option should be chosen when the system is not making an effective contribution to business processes. This usually occurs when business processes have changed since the system was installed and are no longer reliant on the legacy system.
2. *Leave the system unchanged and continue with regular maintenance* This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests.
3. *Reengineer the system to improve its maintainability* This option should be chosen when the system quality has been degraded by change and where new change to the system is still being proposed. This process may include developing new interface components so that the original system can work with other, newer systems.
4. *Replace all or part of the system with a new system* This option should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation, or where off-the-shelf systems would allow the new system to be developed at a reasonable cost. In many cases, an evolutionary replacement strategy can be adopted where major system components are replaced by off-the-shelf systems with other components reused wherever possible.

When you are assessing a legacy system, you have to look at it from both a business perspective and a technical perspective (Warren 1998). From a business perspective, you have to decide whether or not the business really needs the system. From a technical perspective, you have to assess the quality of the application software and the system's support software and hardware. You then use a combination of the business value and the system quality to inform your decision on what to do with the legacy system.

For example, assume that an organization has 10 legacy systems. You should assess the quality and the business value of each of these systems. You may then create a chart showing relative business value and system quality. An example of

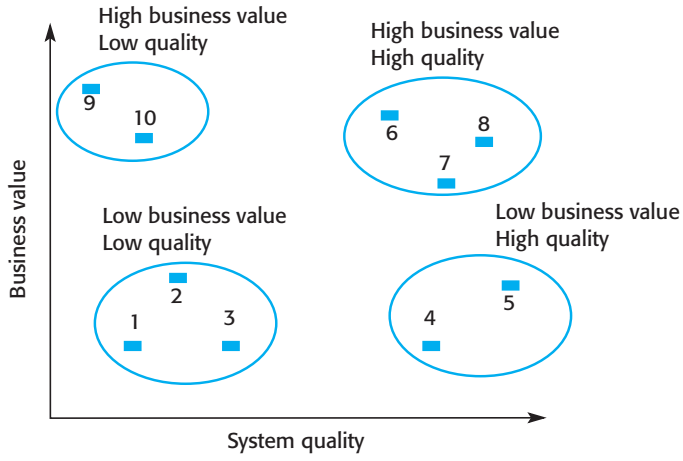


Figure 9.9 An example of a legacy system assessment

this is shown in Figure 9.9. From this diagram, you can see that there are four clusters of systems:

1. *Low quality, low business value* Keeping these systems in operation will be expensive, and the rate of the return to the business will be fairly small. These systems should be scrapped.
2. *Low quality, high business value* These systems are making an important business contribution, so they cannot be scrapped. However, their low quality means that they are expensive to maintain. These systems should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf systems are available.
3. *High quality, low business value* These systems don't contribute much to the business but may not be very expensive to maintain. It is not worth replacing these systems, so normal system maintenance may be continued if expensive changes are not required and the system hardware remains in use. If expensive changes become necessary, the software should be scrapped.
4. *High quality, high business value* These systems have to be kept in operation. However, their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

The business value of a system is a measure of how much time and effort the system saves compared to manual processes or the use of other systems. To assess the business value of a system, you have to identify system stakeholders, such as the end-users of a system and their managers, and ask a series of questions about the system. There are four basic issues that you have to discuss:

1. *The use of the system* If a system is only used occasionally or by a small number of people, this may mean that it has a low business value. A legacy system may have been developed to meet a business need that has either changed or can now be met

more effectively in other ways. You have to be careful, however, about occasional but important use of systems. For example, a university system for student registration may only be used at the beginning of each academic year. Although it is used infrequently, it is an essential system with a high business value.

2. *The business processes that are supported* When a system is introduced, business processes are usually introduced to exploit the system's capabilities. If the system is inflexible, changing these business processes may be impossible. However, as the environment changes, the original business processes may become obsolete. Therefore, a system may have a low business value because it forces the use of inefficient business processes.

3. *System dependability* System dependability is not only a technical problem but also a business problem. If a system is not dependable and the problems directly affect business customers, or mean that people in the business are diverted from other tasks to solve these problems, the system has a low business value.

4. *The system outputs* The key issue here is the importance of the system outputs to the successful functioning of the business. If the business depends on these outputs, then the system has a high business value. Conversely, if these outputs can be cheaply generated in some other way, or if the system produces outputs that are rarely used, then the system has a low business value.

For example, assume that a company provides a travel ordering system that is used by staff responsible for arranging travel. They can place orders with an approved travel agent. Tickets are then delivered, and the company is invoiced for them. However, a business value assessment may reveal that this system is only used for a fairly small percentage of travel orders placed. People making travel arrangements find it cheaper and more convenient to deal directly with travel suppliers through their websites. This system may still be used, but there is no real point in keeping it—the same functionality is available from external systems.

Conversely, say a company has developed a system that keeps track of all previous customer orders and automatically generates reminders for customers to reorder goods. This results in a large number of repeat orders and keeps customers satisfied because they feel that their supplier is aware of their needs. The outputs from such a system are important to the business, so this system has a high business value.

To assess a software system from a technical perspective, you need to consider both the application system itself and the environment in which the system operates. The environment includes the hardware and all associated support software such as compilers, debuggers and development environments that are needed to maintain the system. The environment is important because many system changes, such as upgrades to the hardware or operating system, result from changes to the environment.

Factors that you should consider during the environment assessment are shown in Figure 9.10. Notice that these are not all technical characteristics of the environment. You also have to consider the reliability of the suppliers of the hardware and support software. If suppliers are no longer in business, their systems may not be supported, so you may have to replace these systems.

Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly, but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If high costs are associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system?

Figure 9.10 Factors used in environment assessment

In the process of environmental assessment, if possible, you should ideally collect data about the system and system changes. Examples of data that may be useful include the costs of maintaining the system hardware and support software, the number of hardware faults that occur over some time period and the frequency of patches and fixes applied to the system support software.

To assess the technical quality of an application system, you have to assess those factors (Figure 9.11) that are primarily related to the system dependability, the difficulties of maintaining the system, and the system documentation. You may also collect data that will help you judge the quality of the system such as:

1. *The number of system change requests* System changes usually corrupt the system structure and make further changes more difficult. The higher this accumulated value, the lower the quality of the system.
2. *The number of user interfaces* This is an important factor in forms-based systems where each form can be considered as a separate user interface. The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
3. *The volume of data used by the system* As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data. When data has been collected over a long period of time, errors and inconsistencies are inevitable. Cleaning up old data is a very expensive and time-consuming process.

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

Figure 9.11 Factors used in application assessment

Ideally, objective assessment should be used to inform decisions about what to do with a legacy system. However, in many cases, decisions are not really objective but are based on organizational or political considerations. For example, if two businesses merge, the most politically powerful partner will usually keep its systems and scrap the other company's systems. If senior management in an organization decides to move to a new hardware platform, then this may require applications to be replaced. If no budget is available for system transformation in a particular year, then system maintenance may be continued, even though this will result in higher long-term costs.

9.3 Software maintenance

Software maintenance is the general process of changing a system after it has been delivered. The term is usually applied to custom software, where separate development groups are involved before and after delivery. The changes made to the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or to accommodate new requirements. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system.

Program evolution dynamics

Program evolution dynamics is the study of evolving software systems, pioneered by Manny Lehman and Les Belady in the 1970s. This led to so-called Lehman's Laws, which are said to apply to all large-scale software systems. The most important of these laws are:

1. A program must continually change if it is to remain useful.
2. As an evolving program changes, its structure is degraded.
3. Over a program's lifetime, the rate of change is roughly constant and independent of the resources available.
4. The incremental change in each release of a system is roughly constant.
5. New functionality must be added to systems to increase user satisfaction.

http://software-engineering-book.com/web/program-evolution-dynamics/

There are three different types of software maintenance:

1. *Fault repairs to fix bugs and vulnerabilities*. Coding errors are usually relatively cheap to correct; design errors are more expensive because they may involve rewriting several program components. Requirements errors are the most expensive to repair because extensive system redesign may be necessary.
2. *Environmental adaptation to adapt the software to new platforms and environments*. This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes. Application systems may have to be modified to cope with these environmental changes.
3. *Functionality addition to add new features and to support new requirements.* This type of maintenance is necessary when system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

In practice, there is no clear-cut distinction between these types of maintenance. When you adapt a system to a new environment, you may add functionality to take advantage of new environmental features. Software faults are often exposed because users use the system in unanticipated ways. Changing the system to accommodate their way of working is the best way to fix these faults.

These types of maintenance are generally recognized, but different people sometimes give them different names. "Corrective maintenance" is universally used to refer to maintenance for fault repair. However, "adaptive maintenance" sometimes means adapting to a new environment and sometimes means adapting the software to new requirements. "Perfective maintenance" sometimes means perfecting the software by implementing new requirements; in other cases, it means maintaining the functionality of the system but improving its structure and its performance. Because of this naming uncertainty, I have avoided the use of these terms in this book.

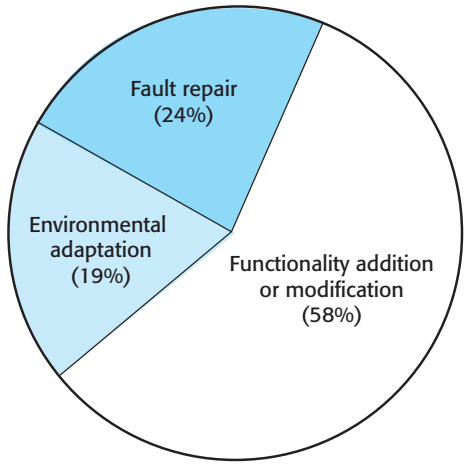


Figure 9.12 Maintenance effort distribution

Figure 9.12 shows an approximate distribution of maintenance costs, based on data from the most recent survey available (Davidsen and Krogstie 2010). This study compared maintenance cost distribution with a number of earlier studies from 1980 to 2005. The authors found that the distribution of maintenance costs had changed very little over 30 years. Although we don't have more recent data, this suggests that this distribution is still largely correct. Repairing system faults is not the most expensive maintenance activity. Evolving the system to cope with new environments and new or changed requirements generally consumes most maintenance effort.

Experience has shown that it is usually more expensive to add new features to a system during maintenance than it is to implement the same features during initial development. The reasons for this are:

1. *A new team has to understand the program being maintained.* After a system has been delivered, it is normal for the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before they can implement changes to it.

2. *Separating maintenance and development means there is no incentive for the development team to write maintainable software.* The contract to maintain a system is usually separate from the system development contract. A different company, rather than the original software developer, may be responsible for software maintenance. In those circumstances, a development team gets no benefit from investing effort to make the software maintainable. If a development team can cut corners to save effort during development it is worthwhile for them to do so, even if this means that the software is more difficult to change in future.

3. *Program maintenance work is unpopular*. Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development

Documentation

System documentation can help the maintenance process by providing maintainers with information about the structure and organization of the system and the features that it offers to system users. While proponents of agile approaches suggest that the code should be the principal documentation, higher level design models and information about dependencies and constraints can make it easier to understand and make changes to that code.

http://software-engineering-book.com/web/documentation/ (web chapter)

and is often allocated to the least experienced staff. Furthermore, old systems may be written in obsolete programming languages. The developers working on maintenance may not have much experience of these languages and must learn these languages to maintain the system.

4. *As programs age, their structure degrades and they become harder to change.* As changes are made to programs, their structure tends to degrade. Consequently, they become harder to understand and change. Some systems have been developed without modern software engineering techniques. They may never have been well structured and were perhaps optimized for efficiency rather than understandability. System documentation may be lost or inconsistent. Old systems may not have been subject to stringent configuration management, so developers have to spend time finding the right versions of system components to change.

The first three of these problems stem from the fact that many organizations still consider software development and maintenance to be separate activities. Maintenance is seen as a second-class activity, and there is no incentive to spend money during development to reduce the costs of system change. The only long-term solution to this problem is to think of systems as evolving throughout their lifetime through a continual development process. Maintenance should have as high a status as new software development.

The fourth issue, the problem of degraded system structure, is, in some ways, the easiest problem to address. Software reengineering techniques (described later in this chapter) may be applied to improve the system structure and understandability. Architectural transformations can adapt the system to new hardware. Refactoring can improve the quality of the system code and make it easier to change.

In principle, it is almost always cost-effective to invest effort in designing and implementing a system to reduce the costs of future changes. Adding new functionality after delivery is expensive because you have to spend time learning the system and analyzing the impact of the proposed changes. Work done during development to structure the software and to make it easier to understand and change will reduce evolution costs. Good software engineering techniques such as precise specification, test-first development, the use of object-oriented development, and configuration management all help reduce maintenance cost.

These principled arguments for lifetime cost savings by investing in making systems more maintainable are, unfortunately, impossible to substantiate with real

data. Collecting data is expensive, and the value of that data is difficult to judge; therefore, the vast majority of companies do not think it is worthwhile to gather and analyze software engineering data.

In reality, most businesses are reluctant to spend more on software development to reduce longer-term maintenance costs. There are two main reasons for their reluctance:

1. Companies set out quarterly or annual spending plans, and managers are incentivized to reduce short-term costs. Investing in maintainability leads to short-term cost increases, which are measurable. However, the long-term gains can't be measured at the same time, so companies are reluctant to spend money on something with an unknown future return.
2. Developers are not usually responsible for maintaining the system they have developed. Consequently, they don't see the point of doing additional work that might reduce maintenance costs, as they will not get any benefit from it.

The only way around this problem is to integrate development and maintenance so that the original development team remains responsible for software throughout its lifetime. This is possible for software products and for companies such as Amazon, which develop and maintain their own software (O'Hanlon 2006). However, for custom software developed by a software company for a client, this is unlikely to happen.

9.3.1 Maintenance prediction

Maintenance prediction is concerned with trying to assess the changes that may be required in a software system and with identifying those parts of the system that are likely to be the most expensive to change. If you understand this, you can design the software components that are most likely to change to make them more adaptable. You can also invest effort in improving those components to reduce their lifetime maintenance costs. By predicting changes, you can also assess the overall maintenance costs for a system in a given time period and so set a budget for maintaining the software. Figure 9.13 shows possible predictions and the questions that these predictions may answer.

Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. Some systems have a very complex relationship with their external environment, and changes to that environment inevitably result in changes to the system. To evaluate the relationships between a system and its environment, you should look at:

1. *The number and complexity of system interfaces* The larger the number of interfaces and the more complex these interfaces, the more likely it is that interface changes will be required as new requirements are proposed.

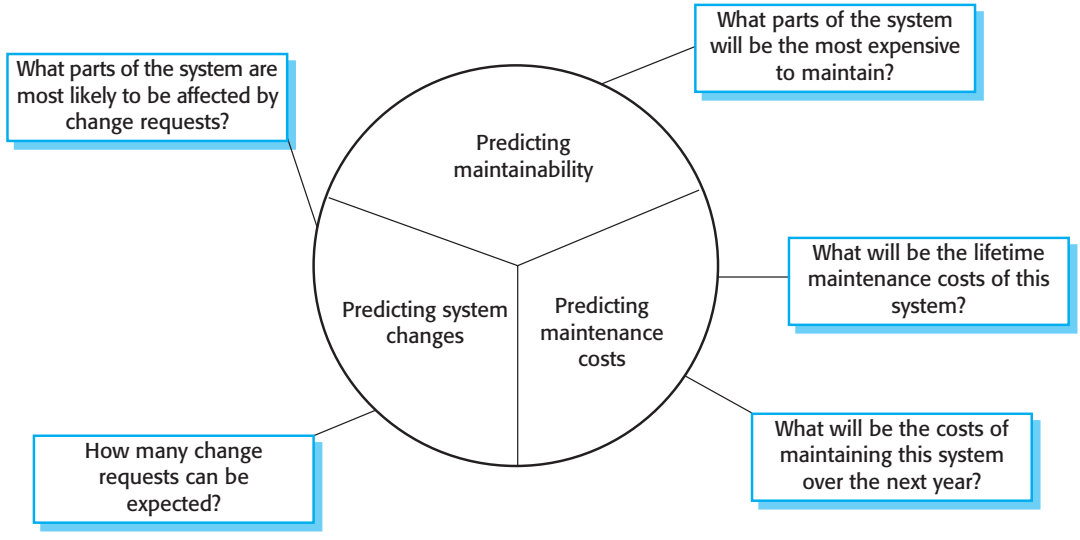


Figure 9.13 Maintenance prediction

2. *The number of inherently volatile system requirements* As I discussed in Chapter 4, requirements that reflect organizational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.

3. *The business processes in which the system is used* As business processes evolve, they generate system change requests. As a system is integrated with more and more business processes, there are increased demands for changes.

In early work on software maintenance, researchers looked at the relationships between program complexity and maintainability (Banker et al. 1993; Coleman et al. 1994; Kozlov et al. 2008). These studies found that the more complex a system or component, the more expensive it is to maintain. Complexity measurements are particularly useful in identifying program components that are likely to be expensive to maintain. Therefore, to reduce maintenance costs you should try to replace complex system components with simpler alternatives.

After a system has been put into service, you may be able to use process data to help predict maintainability. Examples of process metrics that can be used for assessing maintainability are:

1. *Number of requests for corrective maintenance* An increase in the number of bug and failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. This may indicate a decline in maintainability.

2. *Average time required for impact analysis* This is related to the number of program components that are affected by the change request. If the time required for impact analysis increases, it implies that more and more components are affected and maintainability is decreasing.

3. *Average time taken to implement a change request* This is not the same as the time for impact analysis although it may correlate with it. This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected. An increase in the time needed to implement a change may indicate a decline in maintainability.

4. *Number of outstanding change requests* An increase in this number over time may imply a decline in maintainability.

You use predicted information about change requests and predictions about system maintainability to predict maintenance costs. Most managers combine this information with intuition and experience to estimate costs. The COCOMO 2 model of cost estimation, discussed in Chapter 23, suggests that an estimate for software maintenance effort can be based on the effort to understand existing code and the effort to develop the new code.

9.3.2 Software reengineering

Software maintenance involves understanding the program that has to be changed and then implementing any required changes. However, many systems, especially older legacy systems, are difficult to understand and change. The programs may have been optimized for performance or space utilization at the expense of understandability, or, over time, the initial program structure may have been corrupted by a series of changes.

To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability. Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of the system's data. The functionality of the software is not changed, and, normally, you should try to avoid making major changes to the system architecture.

Reengineering has two important advantages over replacement:

1. *Reduced risk* There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.

2. *Reduced cost* The cost of reengineering may be significantly less than the cost of developing new software. Ulrich (Ulrich 1990) quotes an example of a commercial system for which the reimplementation costs were estimated at $50 million. The system was successfully reengineered for $12 million. I suspect that, with modern software technology, the relative cost of reimplementation is probably less than Ulrich's figure but will still be more than the costs of reengineering.

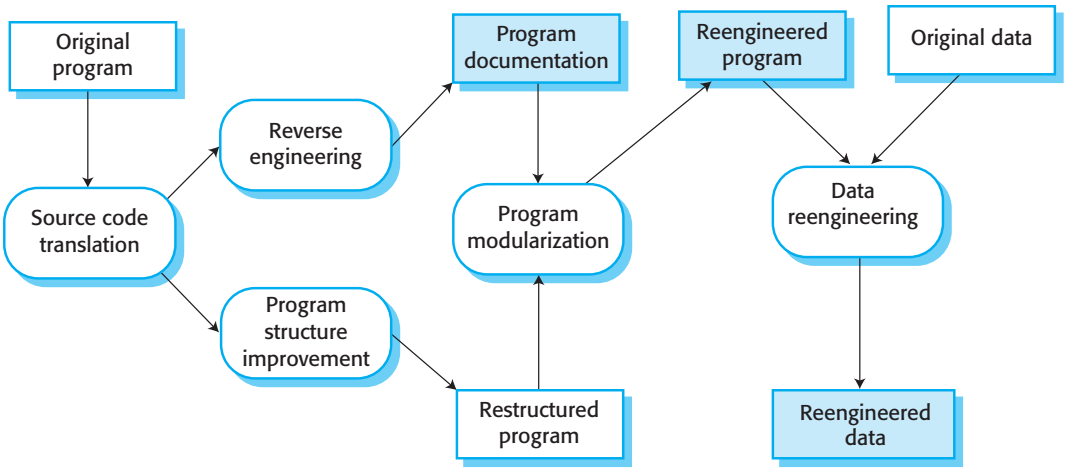


Figure 9.14 The reengineering process

Figure 9.14 is a general model of the reengineering process. The input to the process is a legacy program, and the output is an improved and restructured version of the same program. The activities in this reengineering process are:

1. *Source code translation* Using a translation tool, you can convert the program from an old programming language to a more modern version of the same language or to a different language.
2. *Reverse engineering* The program is analyzed and information extracted from it. This helps to document its organization and functionality. Again, this process is usually completely automated.
3. *Program structure improvement* The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated, but some manual intervention is usually required.
4. *Program modularization* Related parts of the program are grouped together, and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository). This is a manual process.
5. *Data reengineering* The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure. You should usually also clean up the data. This involves finding and correcting mistakes, removing duplicate records, and so on. This can be a very expensive and prolonged process.

Program reengineering may not necessarily require all of the steps in Figure 9.11. You don't need source code translation if you still use the application's programming language. If you can do all reengineering automatically, then recovering documentation through reverse engineering may be unnecessary. Data reengineering is required only if the data structures in the program change during system reengineering.

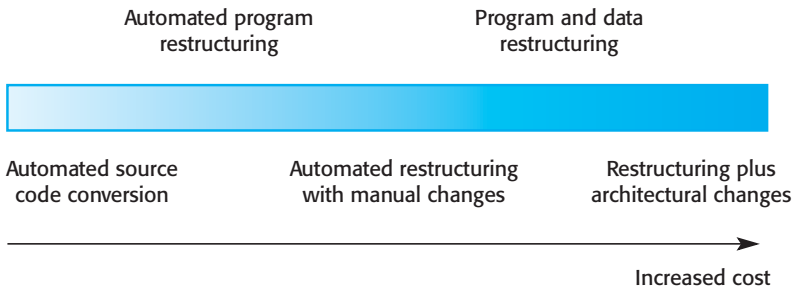


Figure 9.15 Reengineering approaches

To make the reengineered system interoperate with the new software, you may have to develop adaptor services, as discussed in Chapter 18. These hide the original interfaces of the software system and present new, better-structured interfaces that can be used by other components. This process of legacy system wrapping is an important technique for developing large-scale reusable services.

The costs of reengineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to reengineering, as shown in Figure 9.15. Costs increase from left to right so that source code translation is the cheapest option, and reengineering, as part of architectural migration, is the most expensive.

The problem with software reengineering is that there are practical limits to how much you can improve a system by reengineering. It isn't possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes or radical reorganizing of the system data management cannot be carried out automatically, so they are very expensive. Although reengineering can improve maintainability, the reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

9.3.3 Refactoring

Refactoring is the process of making improvements to a program to slow down degradation through change. It means modifying a program to improve its structure, reduce its complexity, or make it easier to understand. Refactoring is sometimes considered to be limited to object-oriented development, but the principles can in fact be applied to any development approach. When you refactor a program, you should not add functionality but rather should concentrate on program improvement. You can therefore think of refactoring as "preventative maintenance" that reduces the problems of future change.

Refactoring is an inherent part of agile methods because these methods are based on change. Program quality is liable to degrade quickly, so agile developers frequently refactor their programs to avoid this degradation. The emphasis on regression testing in agile methods lowers the risk of introducing new errors through refactoring. Any errors that are introduced should be detectable, as previously successful tests should then fail. However, refactoring is not dependent on other "agile activities."

Although reengineering and refactoring are both intended to make software easier to understand and change, they are not the same thing. Reengineering takes place after a system has been maintained for some time, and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Fowler et al. (Fowler et al. 1999) suggest that there are stereotypical situations (Fowler calls them "bad smells") where the code of a program can be improved. Examples of bad smells that can be improved through refactoring include:

1. *Duplicate code* The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
2. *Long methods* If a method is too long, it should be redesigned as a number of shorter methods.
3. *Switch (case) statements* These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
4. *Data clumping* Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program. These can often be replaced with an object that encapsulates all of the data.
5. *Speculative generality* This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Fowler, in both his book and website, also suggests some primitive refactoring transformations that can be used singly or together to deal with bad smells. Examples of these transformations include Extract method, where you remove duplication and create a new method; Consolidate conditional expression, where you replace a sequence of tests with a single test; and Pull up method, where you replace similar methods in subclasses with a single method in a superclass. Interactive development environments, such as Eclipse, usually include refactoring support in their editors. This makes it easier to find dependent parts of a program that have to be changed to implement the refactoring.

Refactoring, carried out during program development, is an effective way to reduce the long-term maintenance costs of a program. However, if you take over a program for maintenance whose structure has been significantly degraded, then it may be practically impossible to refactor the code alone. You may also have to think about design refactoring, which is likely to be a more expensive and difficult problem. Design refactoring involves identifying relevant design patterns (discussed in Chapter 7) and replacing existing code with code that implements these design patterns (Kerievsky 2004).