Chapter 8

Reusability and Portability

Learning Objectives

After studying this chapter, you should be able to

- Explain why reuse is so important.
- Appreciate the obstacles to reuse.
- Describe techniques for achieving reuse during the various workflows.
- Appreciate the importance of design patterns.
- Discuss the impact of reuse on maintainability.
- Explain why portability is essential.
- Understand the obstacles to achieving portability.
- Develop portable software.

If reinventing the wheel were a criminal offense, many software professionals would today be languishing in jail. For example, there are tens of thousands (if not hundreds of thousands) of different COBOL payroll programs, all doing essentially the same thing. Surely, the world needs just one payroll program that can run on a variety of hardware and be tailored, if necessary, to cater to the specific needs of an individual organization. However, instead of utilizing previously developed payroll programs, myriad organizations all over the world have built their own payroll program from scratch. In this chapter, we investigate why software engineers delight in continually reinventing the wheel, and what can be done to achieve portable software built using reusable components. We begin by distinguishing between portability and reusability.

As explained in the Preface, the material of this chapter may be taught in parallel with that of Part 2

Just in Case You Wanted to Know Box 8.1

Reuse is not restricted to software. For example, lawyers nowadays rarely draft wills from scratch. Instead, they use a word processor to store wills they have previously drafted, and then make appropriate changes to an existing will. Other legal documents, like contracts, are usually drafted in the same way from existing documents.

Classical composers frequently reused their own music. For example, in 1823 Franz Schubert wrote an entr'acte for Helmina von Chezy's play, *Rosamunde, Fürstin von Zypern (Rosamunde, Princess of Cyprus)* and the following year he reused that material in the slow movement of his String Quartet No. 13. Ludwig van Beethoven's Opus 66, "Variations for Cello on Mozart's *Ein Mädchen oder Weibchen*," is a good example of one great composer reusing the music of another great composer; Beethoven simply took the aria "A Girlfriend or Little Wife" from Scene 22 of Wolfgang Amadeus Mozart's opera *Der Zauberflöte (The Magic Flute)* and wrote a series of seven variations on that aria for the cello with piano accompaniment.

In my opinion, the greatest reuser of all time was William Shakespeare. His genius lay in reusing the plots of others—I cannot think of a single story line he made up himself. For example, his historical plays heavily reused parts of Raphael Holinshed's 1577 work, *Chronicles of England, Scotland and Ireland*. Then, Shakespeare's *Romeo and Juliet* (1594) is borrowed, on an almost line-for-line basis, from Arthur Brooke's lengthy poem *The Tragicall Historye of Romeus and Iuliet* published in 1562, two years before Shakespeare was born.

But this reuse saga didn't begin there. In fact, the earliest known version appeared around 200 CE in *Ephesiaka (Ephesian Tale)* by the Greek novelist Xenophon of Ephesus. In 1476, Tommaso Guardati (more commonly known as Masuccio Salernitano) reused Xenophon's tale in novella 33 in his collection of 50 novellas, *Il Novellino*. In 1530, Luigi da Porto reused that story in *Historia Novellamente Ritrovata di Due Nobili Amanti (A Newly Found Story of Two Noble Lovers)*, for the first time setting it in Verona, Italy. Brooke's poem reuses parts of *Giulietta e Romeo* (1554) by Matteo Bandello, a reuse of da Porto's version.

And this reuse saga didn't end with *Romeo and Juliet*, either. In 1957, *West Side Story* opened on Broadway. The musical, with book by Arthur Laurents, lyrics by Stephen Sondheim, and score by Leonard Bernstein, reused Shakespeare's version of the story. The Broadway musical was then reused in a Hollywood movie, which won 10 Academy Awards in 1961.

8.1 Reuse Concepts

A product is ***portable*** if it is significantly easier to modify the product as a whole to run it on another compiler–hardware–operating system configuration than to recode it from scratch. In contrast, ***reuse*** refers to using components of one product to facilitate the development of a different product with a different functionality. A reusable component need not necessarily be a module or a code fragment—it could be a design, a part of a manual, a set of test data, or a duration and cost estimate. (For a different view on reuse, see Just in Case You Wanted to Know Box 8.1.)

There are two types of reuse, opportunistic reuse and deliberate reuse. If the developers of a new product realize that a component of a previously developed product can be reused in the new product, then this is ***opportunistic reuse***, sometimes referred to as ***accidental reuse***. On the other hand, utilization of software components constructed specifically for possible future reuse is ***systematic reuse*** or ***deliberate reuse***. A potential advantage

of systematic reuse over opportunistic reuse is that artifacts specially constructed for use in future products are more likely to be easy and safe to reuse; such artifacts generally are robust, well documented, and thoroughly tested. In addition, they usually display a uniformity of style that makes maintenance easier. The other side of the coin is that implementing systematic reuse within a company can be expensive. It takes time to specify, design, implement, test, and document a software artifact. However, there can be no guarantee that such an artifact will be reused and thereby recoup the money invested in developing the potentially reusable artifact.

When computers were first constructed, nothing was reused. Every time a product was developed, items such as multiplication routines, input–output routines, or routines for computing sines and cosines were constructed from scratch. Quite soon, however, it was realized that this was a considerable waste of effort, and subroutine libraries were constructed. Programmers then simply could invoke square root or sine functions whenever they wished. These subroutine libraries have become more and more sophisticated and developed into run-time support routines. Therefore, when a programmer calls a C++ or Java method, there is no need to write code to manage the stack or pass the arguments explicitly; it is handled automatically by calling the appropriate run-time support routines. The concept of subroutine libraries has been extended to large-scale statistical libraries such as SPSS [Norušis, 2005] and numerical analysis libraries like NAG [2003]. Class libraries also play a major role in assisting users of object-oriented languages. For example, the success of Smalltalk is due at least partly to the wide variety of items in the Smalltalk library together with the presence of a browser, a CASE tool that assists the user to scan a class library. With regard to C++, a large number of different libraries are available, many in the public domain. One example is the C++ Standard Template Library (STL) [Musser and Saini, 1996].

An application programming interface (API) generally is a set of operating system calls that facilitate programming. For example, Win32 is an API for Microsoft operating systems such as Windows XP; and Cocoa is an API for Mac OS X, a Macintosh operating system. Although an API usually is implemented as a set of operating system calls, to the programmer the routines constituting the API can be viewed as a subroutine library. For example, the Java Application Programming Interface consists of a number of packages (libraries).

No matter how high the quality of a software product may be, it will not sell if it takes 2 years to get it onto the market when a competitive product can be delivered in only 1 year. The length of the development process is critical in a market economy. All other criteria as to what constitutes a "good" product are irrelevant if the product cannot compete timewise. For a corporation that has repeatedly failed to get a product to market first, software reuse offers a tempting technique. After all, if an existing component is reused, then there is no need to specify, design, implement, test, and document that component. The key point is that, on average, only about 15 percent of any software product serves a truly original purpose [Jones, 1984]. The other 85 percent of the product in theory could be standardized and reused in future products.

The figure of 85 percent is essentially a theoretical upper limit for the reuse rate; nevertheless, reuse rates on the order of 40 percent can be achieved in practice. This leads to an obvious question: If such reuse rates are attainable in practice and reuse is by no means a new idea, why do so few organizations employ reuse to shorten the development process?

8.2 Impediments to Reuse

There are a number of impediments to reuse:

- All too many software professionals would rather rewrite a component from scratch than reuse a component written by someone else, the implication being that a component cannot be any good unless they wrote it themselves, otherwise known as the ***not invented here (NIH) syndrome*** [Griss, 1993]. NIH is a management issue, and, if management is aware of the problem, it can be solved, usually by offering financial incentives to promote reuse.
- Many developers would be willing to reuse a component provided they could be sure that the component in question would not introduce faults into the product. This attitude toward software quality is perfectly easy to understand. After all, every software professional has seen faulty software written by others. The solution here is to subject potentially reusable components to exhaustive testing before making them available for reuse.
- A large organization may have hundreds of thousands of potentially useful components. How should these components be stored for effective later retrieval? For example, a reusable components database might consist of 20,000 items, 125 of which are sort routines. The database must be organized so that the designer of a new product can quickly determine which (if any) of those 125 sort routines is appropriate for the new product. Solving the storage/retrieval problem is a technical issue for which a wide variety of solutions have been proposed (e.g., [Meyer, 1987] or [Prieto-Díaz, 1991]).
- Reuse can be expensive. Tracz [1994] has stated that three costs are involved: the cost of making a component reusable, the cost of reusing a component, and the cost of defining and implementing a reuse process. He estimates that just making a component reusable increases its cost by at least 60 percent. Some organizations have reported cost increases of 200 percent and even up to 480 percent, whereas the cost of making a component reusable was only 11 percent in one Hewlett-Packard reuse project [Lim, 1994].
- Legal issues can arise with contract software. In terms of the type of contract usually drawn up between a client and a software development organization, the software product belongs to the client. Therefore, if the software developer reuses a component of one client's product in a new product for a different client, this essentially constitutes a violation of the first client's copyright. For internal software, that is, when the developers and client are members of the same organization, this problem does not arise.
- Another impediment arises when commercial off-the-shelf (COTS) components are reused. Rarely are developers given the source code of a COTS component, so software that reuses COTS components has limited extensibility and modifiability.

The first four impediments can be overcome, at least in principle. So, other than certain legal issues and problems with COTS components, essentially no major impediments prevent implementing reuse within a software organization (but see Just in Case You Wanted to Know Box 8.2).

Just in Case You Wanted to Know Box 8.2

The World Wide Web is a great source of "urban myths," that is, apparently true stories that somehow just do not stand up under scrutiny when they are investigated closely. One such urban myth concerns code reuse.

The story is told that the Australian Air Force set up a virtual reality training simulator for helicopter combat training. To make the scenarios as realistic as possible, programmers included detailed landscapes and (in the Northern Territory) herds of kangaroos. After all, the dust from a herd disturbed by a helicopter might reveal the position of that helicopter to the enemy.

The programmers were instructed to model both the movements of the kangaroos and their reaction to helicopters. To save time, the programmers reused code originally used to simulate the reaction of infantry to attack by a helicopter. Only two changes were made: They changed the icon from a soldier to a kangaroo, and they increased the speed of movement of the figures.

One fine day, a group of Australian pilots wanted to demonstrate their prowess with the flight simulator to some visiting American pilots. They "buzzed" (flew extremely low over) the virtual kangaroos. As expected, the kangaroos scattered, and then reappeared from behind a hill and launched Stinger missiles at the helicopter. The programmers had forgotten to remove that part of the code when they reused the virtual infantry implementation.

However, as reported in *The Risks Digest*, it appears that the story is not totally an urban myth—much of it actually happened [Green, 2000]. Dr. Anne-Marie Grisogono, head of the Simulation Land Operations Division at the Australian Defence Science and Technology Organisation, told the story at a meeting in Canberra, Australia, on May 6, 1999. Although the simulator was designed to be as realistic as possible (it even included over 2 million virtual trees, as indicated on aerial photographs), the kangaroos were included for fun. The programmers indeed reused Stinger missile detachments so that the kangaroos could detect the arrival of helicopters, but the behavior of the kangaroos was set to "retreat" so that the kangaroos, correctly, would flee if a helicopter approached. However, when the software team tested their simulator in their laboratory (not in front of visitors), they discovered that they had forgotten to remove both the weapons and "fire" behavior. Also, they had not specified what weapons were to be used by the simulated figures, so when the kangaroos fired on the helicopters, they fired the default weapon, which happened to be large multicolored beachballs.

Grisogono confirmed that the kangaroos were immediately disarmed and therefore it is now safe to fly over Australia. But notwithstanding this happy ending, software professionals still must take care when reusing code not to reuse too much of it.

8.3 Reuse Case Studies

Many published case studies show how reuse has been successfully achieved in practice; reuse case studies that have had a major impact include [Matsumoto, 1984, 1987; Selby, 1989; Prieto-Díaz, 1991; and Lim, 1994]. Here, we analyze two case studies. The first, which describes a reuse project that took place between 1976 and 1982, is important because the reuse mechanism used then for COBOL designs is the same as the reuse mechanism used today in object-oriented application frameworks (Section 8.5.2). This case study therefore serves to clarify modern reuse practices.

8.3.1 Raytheon Missile Systems Division

In 1976, a study was undertaken at Raytheon's Missile Systems Division to determine whether systematic reuse of designs and code was feasible [Lanergan and Grasso, 1984]. Over 5000 COBOL products in use were analyzed and classified. The researchers determined that only six basic operations are performed in a business application product. As a result, between 40 and 60 percent of business application designs and modules could be standardized and reused. The basic operations were found to be sort data, edit or manipulate data, combine data, explode data, update data, and report on data. For the next 6 years, a concerted attempt was made to reuse both design and code wherever possible.

The Raytheon approach employed reuse in two ways, what the researchers termed *functional modules* and *COBOL program logic structures*. In Raytheon's terminology a ***functional module*** is a COBOL code fragment designed and coded for a specific purpose, such as an edit routine, database procedure division call, tax computation routine, or date aging routine for accounts receivable. Use of the 3200 reusable modules resulted in applications that, on average, consisted of 60 percent reused code. Functional modules were carefully designed, tested, and documented. Products that used these functional modules were found to be more reliable, and less testing of the product as a whole was needed.

The modules were stored in a standard copy library and obtained with the **copy** verb. That is, the code was not physically present within the application product but was included by the COBOL compiler at compilation time, a mechanism similar to #**include** in C++. The resulting source code therefore was shorter than if the copied code were physically present. As a consequence, maintenance was easier.

The Raytheon researchers also used what they termed a ***COBOL program logic structure***. This is a framework that has to be fleshed out into a complete product. One example of a logic structure is the update logic structure. This is used to perform a sequential update, such as the mini case study in Section 5.1.1. Error handling is built in, as is sequence checking. The logic structure is 22 paragraphs (units of a COBOL program) in length. Many of the paragraphs can be filled in by using functional modules such as get-transaction, print-page-headings, and print-control-totals. Figure 8.1 is a symbolic depiction of the framework of a COBOL program logic structure with the paragraphs filled in by functional modules.

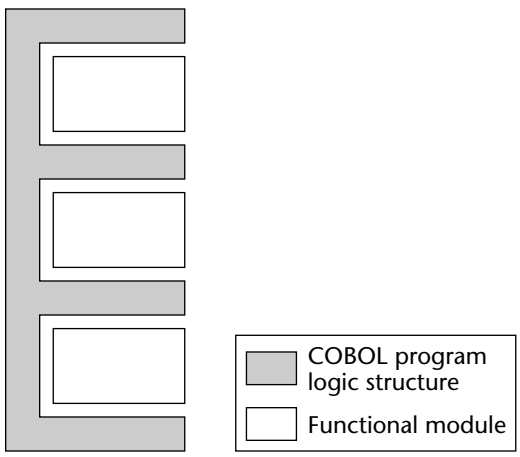


Figure 8.1
A symbolic representation of the Raytheon Missile Systems Division reuse mechanism.

The use of such templates has many advantages. It makes the design and coding of a product quicker and easier, because the framework of the product already is present; all that is needed is to fill in the details. Fault-prone areas such as end-of-file conditions already have been tested. In fact, testing as a whole is easier. But Raytheon believed that the major advantage would occur when the users requested modifications or enhancements. Once a maintenance programmer was familiar with the relevant logic structure, it was almost as if he or she had been a member of the original development team.

By 1983, logic structures had been used over 5500 times in developing new products. About 60 percent of the code consisted of functional modules, that is, reusable code. This meant that design, coding, module testing, and documentation time also was reduced by 60 percent, leading to an estimated 50 percent increase in productivity in software product development. But, for Raytheon, the real benefit of the technique lay in the hope that the readability and understandability resulting from the consistent style would reduce the cost of maintenance by between 60 and 80 percent. Unfortunately, Raytheon closed the division before the necessary maintenance data could be obtained.

The second reuse case study is a cautionary tale, rather than a success story.

8.3.2 European Space Agency

On June 4, 1996, the European Space Agency launched the Ariane 5 rocket for the first time. As a consequence of a software fault, the rocket crashed about 37 seconds after liftoff. The cost of the rocket and payload was about $500 million [Jézéquel and Meyer, 1997].

The primary cause of the failure was an attempt to convert a 64-bit integer into a 16-bit unsigned integer. The number being converted was larger than 2^{16}, so an Ada **exception** (run-time failure) occurred. Unfortunately, there was no explicit exception handler in the code to deal with this exception, so the software crashed. This caused the onboard computers to crash which, in turn, caused the Ariane 5 rocket to crash.

Ironically, the conversion that caused the failure was unnecessary. Certain computations are performed before liftoff to align the inertial reference system. These computations should stop 9 seconds before liftoff. However, if there is a subsequent hold in the countdown, resetting the inertial reference system after the countdown has recommenced can take several hours. To prevent that happening, the computations continue for 50 seconds after the start of flight mode, that is, well into the flight (notwithstanding that, once liftoff has occurred, there is no way to align the inertial reference system). This futile continuation of the alignment process caused the failure.

The European Space Agency uses a careful software development process that incorporates an effective software quality assurance component. Then, why was there no exception handler in the Ada code to handle the possibility of such an overflow? To prevent overloading the computer, conversions that could not possibly result in overflow were left unprotected. The code in question was 10 years old. It had been reused, unchanged and without any further testing, from the software controlling the Ariane 4 rocket (the precursor of the Ariane 5). Mathematical analysis had proven that the computation in question was totally safe for the Ariane 4. However, the analysis was performed on the basis of certain assumptions that were true for the Ariane 4 but not for the Ariane 5. Therefore, the analysis no longer was valid, and the code needed the protection of an exception handler to cater to the possibility of an overflow. Were it not for the performance constraint, there surely would

have been exception handlers throughout the Ariane 5 Ada code. Alternatively, the use of the **assert pragma** both during testing and after the product had been installed (Section 6.5.3) could have prevented the Ariane 5 crash if the relevant module had included an assertion that the number to be converted was smaller than 2^{16} [Jézéquel and Meyer, 1997].

The major lesson of this reuse experience is that software developed in one context must be retested when reused in another context. That is, a reused software module does not need to be retested by itself, but it must be retested after it has been integrated into the product in which it is reused. Another lesson is that it is unwise to rely exclusively on the results of mathematical proofs, as discussed in Section 6.5.2.

We now examine the impact of the object-oriented paradigm on reuse.

8.4 Objects and Reuse

When the theory of composite/structured design first was put forward about 30 years ago, the claim was made that an ideal module has functional cohesion (Section 7.2.6). That is, if a module performed only one operation, it was thought to be an exemplary candidate for reuse, and maintenance of such a module was expected to be easy. The flaw in this reasoning is that a module with functional cohesion is not self-contained and independent. Instead, it has to operate on data. If such a module is reused, then the data on which it is to operate must be reused, too. If the data in the new product are not identical to those in the original, then either the data have to be changed or the module with functional cohesion has to be changed. Therefore, contrary to what we used to believe, functional cohesion is not ideal for reuse.

According to C/SD as originally put forward in 1974, the next best type of module is one with informational cohesion (Section 7.2.7). Nowadays, we appreciate that such a module essentially is an object, that is, an instance of a class. A well-designed object is the fundamental building block of software because it models all aspects of a particular real-world entity (conceptual independence, or encapsulation) but conceals the implementation of both its data and the operations that operate on the data (physical independence, or information hiding). Therefore, when the object-oriented paradigm is utilized correctly, the resulting modules (objects) have informational cohesion, and this promotes reuse.

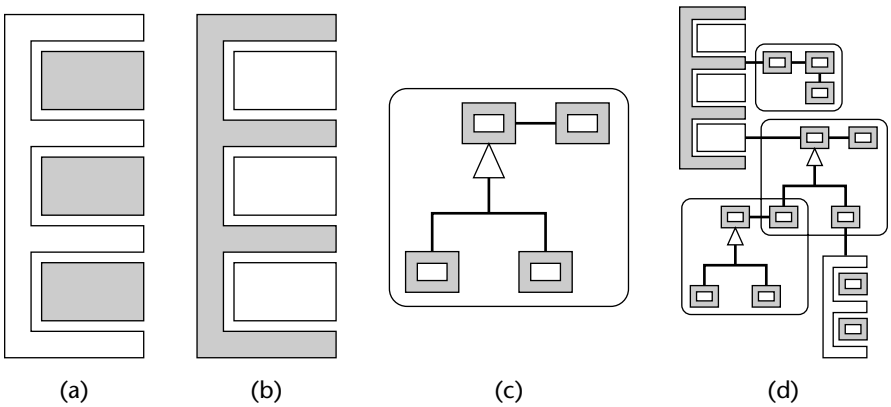
8.5 Reuse during Design and Implementation

Dramatically different types of reuse are possible during design. The reused material can vary from just one or two artifacts to the architecture of the complete software product. We now examine various types of design reuse, some of which carry over into implementation.

8.5.1 Design Reuse

When designing a product, a member of the design team may realize that a class from an earlier design can be reused in the current project, with or without minor modifications. This type of reuse is particularly common in an organization that develops software in one specific application domain, such as banking or air traffic control systems. The organization can

Figure 8.2 A symbolic representation of four types of design reuse. Shading denotes design reuse within (a) a library or a toolkit, (b) a framework, (c) a design pattern, and (d) a software architecture comprising a framework, a toolkit, and three design patterns.



promote this type of reuse by setting up a repository of design components likely to be reused in the future and encouraging designers to reuse them, perhaps by a cash bonus for each such reuse. This type of reuse, limited though it may be, has two advantages. First, tested designs are incorporated into the product. The overall design therefore can be produced more quickly and is likely to have a higher quality than when the entire design is produced from scratch. Second, if the design of a class can be reused, then it is likely that the implementation of that class also can be reused, if not the actual code then at least conceptually.

This approach can be extended to library reuse, depicted in Figure 8.2(a). A library is a set of related reusable routines. For example, developers of scientific software rarely write the methods to perform such common tasks as matrix inversion or finding eigenvalues. Instead, a scientific class library such as LAPACK++ [2000] is purchased. Then, whenever possible, the classes in the scientific library are utilized in future software.

Another example is a library for a graphical user interface. Instead of writing the GUI methods from scratch, it is far more convenient to use a GUI class library or ***toolkit***, that is, a set of classes that can handle every aspect of the GUI. Many GUI toolkits of this kind are available, including the Java Abstract Windowing Toolkit [Flanagan, 2005].

A problem with library reuse is that libraries frequently are presented in the format of a set of reusable code artifacts rather than reusable designs. Toolkits, too, generally promote code reuse rather than design reuse. This problem can be alleviated with the help of a browser, that is, a CASE tool for displaying the inheritance tree. The designer then can traverse the inheritance tree of the library, examine the fields of the various classes, and determine which class is applicable to the current design.

A key aspect of library and toolkit reuse is that, as depicted in Figure 8.2(a), the designer is responsible for the control logic of the product as a whole. The library or toolkit contributes to the software development process by supplying parts of the design that incorporate the specific operations of the product.

On the other hand, an application framework is the converse of a library or toolkit in that it supplies the control logic; the developers are responsible for the design of the specific operations. This is described in Section 8.5.2.

8.5.2 Application Frameworks

As shown in Figure 8.2(b), an ***application framework*** incorporates the control logic of a design. When a framework is reused, the developers have to design the application-specific operations of the product being built. The places where the application-specific operations are inserted frequently are referred to as ***hot spots***.

The term ***framework*** nowadays usually refers to an object-oriented application framework. For example, in [Gamma, Helm, Johnson, and Vlissides, 1995], a *framework* is defined as a "set of cooperating classes that make up a reusable design for a specific class of software." However, consider the Raytheon Missiles Systems Division case study of Section 8.3.1. Figure 8.1 is identical to Figure 8.2(b). In other words, the Raytheon COBOL program logic structure of the 1970s is a classical precursor of today's object-oriented application framework.

An example of an application framework is a set of classes for the design of a compiler. The design team merely has to provide classes specific to the language and desired target machine. These classes then are inserted into the framework, as depicted by the white boxes in Figure 8.2(b). Another example of a framework is a set of classes for the software controlling an ATM. Here, the designers need to provide the classes for the specific banking services offered by the ATMs of that banking network.

Reusing a framework results in faster product development than reusing a toolkit, for two reasons. First, more of the design is reused with a framework, so there is less to design from scratch. Second, the portion of the design that is reused with a framework (the control logic) generally is harder to design than the operations, so the quality of the resulting design also is likely to be higher than when a toolkit is reused. As with library or toolkit reuse, often the implementation of the framework can be reused as well. The developers probably have to use the names and calling conventions of the framework, but that is a small price to pay. Also, the resulting product is likely to be maintained easily because the control logic has been tested in other products that reuse the application framework and the maintainer previously may have maintained another product that reused that same framework.

IBM's WebSphere (formerly known as *e-Components*, and originally as *San Francisco*) is a framework for building online information systems in Java. It utilizes Enterprise JavaBeans, that is, classes that provide services for clients distributed throughout a network.

In addition to application frameworks, many code frameworks are available. One of the first commercially successful code frameworks was MacApp, a framework for writing application software on the Macintosh. Borland's Visual Component Library (VCL) is an object-oriented set of frameworks for building GUIs in Windows-based applications. VCL applications can perform standard windowing operations, such as moving and resizing windows, processing input via dialog boxes, and handling events like mouse clicks or menu selections.

We now consider design patterns.

8.5.3 Design Patterns

Christopher Alexander (see Just in Case You Wanted to Know Box 8.3) said, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander et al., 1977]. Although he was writing within the context of patterns in buildings and other architectural objects, his remarks are equally applicable to design patterns.

Just in Case You Wanted to Know Box 8.3

One of the most influential individuals in the field of object-oriented software engineering is Christopher Alexander, a world-famous architect who freely admits to knowing little or nothing about objects or software engineering. In his books, and especially in [Alexander et al., 1977], he describes a pattern language for architecture, that is, for describing towns, buildings, rooms, gardens, and so on. His ideas were adopted and adapted by object-oriented software engineers, especially the so-called Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides). Their best-selling book on design patterns [Gamma, Helm, Johnson, and Vlissides, 1995] resulted in Alexander's ideas being widely accepted by the object-oriented community.

Patterns occur in other contexts as well. For example, when approaching an airport, pilots have to know the appropriate landing pattern, that is, the sequence of directions, altitudes, and turns needed to land the plane on the correct runway. Also, a dressmaking pattern is a series of shapes that can be used repeatedly to create a particular dress. The concept of a pattern itself is by no means novel. What is new is the application of patterns to software development and especially design.

A design pattern is a solution to a general design problem in the form of a set of interacting classes that have to be customized to create a specific design. This is depicted in Figure 8.2(c). The shaded boxes connected by lines denote the interacting classes. The white boxes inside the shaded boxes denote that these classes must be customized for a specific design.

To understand how patterns can assist with software development, consider the following example. Suppose that a software engineer wishes to reuse two existing classes, **P** and **Q**, say, but that their interfaces are incompatible. For example, when **P** sends a message to **Q**, it passes four parameters, but **Q**'s interface is such that it expects only three parameters. Changing the interface of **P** or **Q** would create a whole host of incompatibility problems in all the applications that currently incorporate **P** or **Q**. Instead, a class **A** needs to be constructed that accepts a message from **P** with four parameters, and sends a message to **Q** with only three parameters. (A class of this kind is sometimes called a ***wrapper***.)

What we have described is a specific solution to a more general problem, namely, enabling any two incompatible classes to work together. Instead of designing this one solution, we need a design pattern, the *Adapter* pattern. Just as an instance of a class is an object, an instance of the *Adapter* pattern is a solution to the incompatibility problem tailored to the two classes involved. This pattern is described in more detail in Section 8.6.

Patterns can interact with other patterns. This is represented symbolically in Figure 8.2(d) where the bottom-left block of the middle pattern again is a pattern. A case study of a document editor in [Gamma, Helm, Johnson, and Vlissides, 1995] contains eight interacting patterns. That is what happens in practice; it is unusual for the design of a product to contain only one pattern.

As with toolkits and frameworks, if a design pattern is reused, then an implementation of that pattern probably also can be reused. In addition, analysis patterns can assist with the analysis workflow [Fowler, 1997]. Finally, in addition to patterns, there are antipatterns; these are described in Just in Case You Wanted to Know Box 8.4.

Because of the importance of design patterns, we return to this topic in Section 8.6, after we have concluded our overview of reuse in design and implementation.

Just in Case You Wanted to Know Box 8.4

An antipattern is a practice that can cause a project to fail, such as "analysis paralysis" (spending far too much time and effort on the analysis workflow) or designing an object-oriented product in which just one object does almost all the work. A major motivation for writing the first antipattern book was that nearly one-third of all software projects are canceled, two-thirds of all software projects encounter cost overruns in excess of 200 percent, and over 80 percent of all software projects are deemed failures [Brown et al., 1998].

8.5.4 Software Architecture

The architecture of a cathedral might be described as Romanesque, Gothic, or Baroque. Similarly, the architecture of a software product might be described as object-oriented, pipes and filters (UNIX components), or client–server (with a central server providing file storage and computing facilities for a network of client computers). Figure 8.2(d) symbolically depicts an architecture composed of a toolkit, a framework, and three design patterns.

Because it applies to the design of a product as a whole, the field of ***software architecture*** encompasses a variety of design issues, including the organization of the product in terms of its components; product-level control structures; issues of communication and synchronization; databases and data access; the physical distribution of the components; performance; and choice of design alternatives [Shaw and Garlan, 1996]. Accordingly, software architecture is a considerably more wide-ranging concept than design patterns.

In fact, Shaw and Garlan [1996] state, "Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, *patterns that guide their composition, and constraints on those patterns*" [emphasis added]. Consequently, in addition to the many items listed in the previous paragraph, software architecture includes patterns as a subfield. This is one reason why Figure 8.2(d) shows three design patterns as components of a software architecture.

The many strengths of design reuse are even greater when a software architecture is reused. One way that reuse of architectures is achieved in practice is with ***software product lines*** [Lai, Weiss, and Parnas, 1999; Jazayeri, Ran, and van der Linden, 2000]. The idea is to develop a software architecture common to a number of software products and instantiate this architecture when developing a new product. For example, Hewlett-Packard manufactures a broad variety of printers, and new models constantly are being developed. Hewlett-Packard now has a firmware architecture that is instantiated for each new printer model. The results have been impressive. For example, between 1995 and 1998, the number of person-hours to develop the firmware for a new printer model decreased by a factor of 4 and the time to develop the firmware decreased by a factor of 3. Also, reuse has increased. For more recent printers, over 70 percent of the components of the firmware are reused, almost unchanged, from earlier products [Toft, Coleman, and Ohta, 2000].

Architecture patterns are another way of achieving architectural reuse. One popular architecture pattern is the ***model-view-controller (MVC) architecture pattern***. As shown in Section 5.1, a traditional way of designing software is to decompose it into three pieces: input, processing, and output. The MVC pattern can be viewed as an extension of the input–processing–output architecture to the GUI domain. The

Figure 8.3 The correspondence between the components of the MVC model and the input–processing–output model.

MVC component	Description	Corresponds to
Model	Core functionality, data	Processing
View	Displays information	Output
Controller	Handles user input	Input

correspondence is shown in Figure 8.3. The view(s) and the controller provide the GUI. The decomposition of the architecture into model, view, and controller allows each of the components to be changed independently of the other two, thereby enhancing the reusability.

Another popular architectural pattern is the three-tier architecture. The ***presentation logic tier*** accepts user input and generates user output—this tier corresponds to the GUI. The ***business logic tier*** incorporates the processing of the business rules. The ***data access logic tier*** communicates with the underlying database. Again, this architectural pattern permits each of the three components to be changed independently of the other two. This independence is a major reason why the three-tier architecture promotes reuse.

8.5.5 Component-Based Software Engineering

The goal of ***component-based software engineering*** is to construct a standard collection of reusable components. Then, instead of reinventing the wheel each time, in the future all software will be constructed by choosing a standard architecture and standard reusable frameworks and inserting standard reusable code artifacts into the hot spots of the frameworks. That is, software products will be built by composing reusable components. Ideally, this will be done using an automated tool.

In this chapter, we describe the many advantages that accrue through the reuse of code artifacts, design patterns, and software architectures. Hence, achieving component-based software engineering would solve numerous problems in software development. In particular, it would lead to order-of-magnitude increases in software productivity and quality and decreases in time to market and maintenance effort.

Unfortunately, the state of the art with regard to reuse is currently far from this ambitious target. In addition, component-based software construction has many challenges, including the definition, standardization, and retrieval of components. However, researchers in many centers are actively engaged in trying to achieve the goal of component-based software engineering [Heineman and Councill, 2001].

8.6 More on Design Patterns

Because of the importance of design patterns in object-oriented software engineering, we now examine design patterns in greater detail. We begin with a mini case study that illustrates the *Adapter* design pattern (Section 8.5.3).

Mini Case Study

8.6.1 FLIC Mini Case Study

Until recently, premiums at Flintstock Life Insurance Company (FLIC) depended on both the age and the gender of the person applying for insurance. FLIC has recently decided that certain policies will now be gender-neutral; that is, the premium for those policies will depend solely on the age of the applicant.

Up to now, premiums have been computed by sending a message to method computePremium of class **Applicant**, passing the age and gender of the applicant. Now, however, a different computation has to be made, based solely on the applicant's age. A new class is written, **Neutral Applicant**, and premiums are computed by sending a message to method computeNeutralPremium in that class. However, there has not been enough time to change the whole system. The situation is therefore as shown in Figure 8.4.

There are serious interfacing problems. First, an **Insurance** object passes a message to an object of type **Applicant**, instead of **Neutral Applicant**. Second, the message is sent to method computePremium instead of method computeNeutralPremium. Third, parameters age and gender are passed, instead of just age. The three question marks on the lower arrow in Figure 8.4 represent these three interfacing problems.

To solve these problems, we need to interpose class **Wrapper**, as shown in Figure 8.5. An object of class **Insurance** sends the same message computePremium passing the same two parameters (age and gender), but now the message is sent to an object of type **Wrapper**. This object then sends message computeNeutralPremium to an object of class **NeutralApplicant**, passing only age as the parameter. The three interfacing problems have been solved.

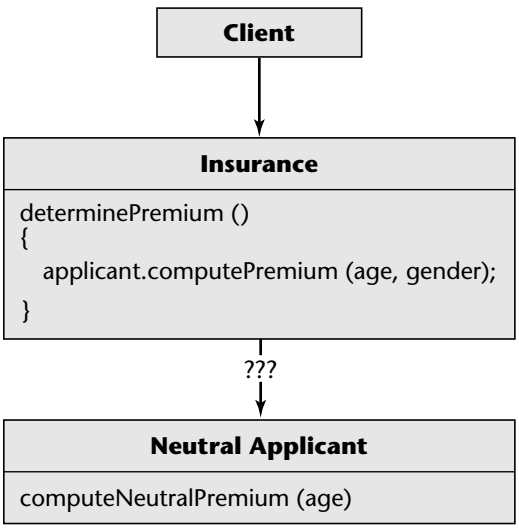


Figure 8.4 A UML diagram showing interfacing problems between classes.

Figure 8.5 Wrapper solution to the interfacing problems of Figure 8.4.

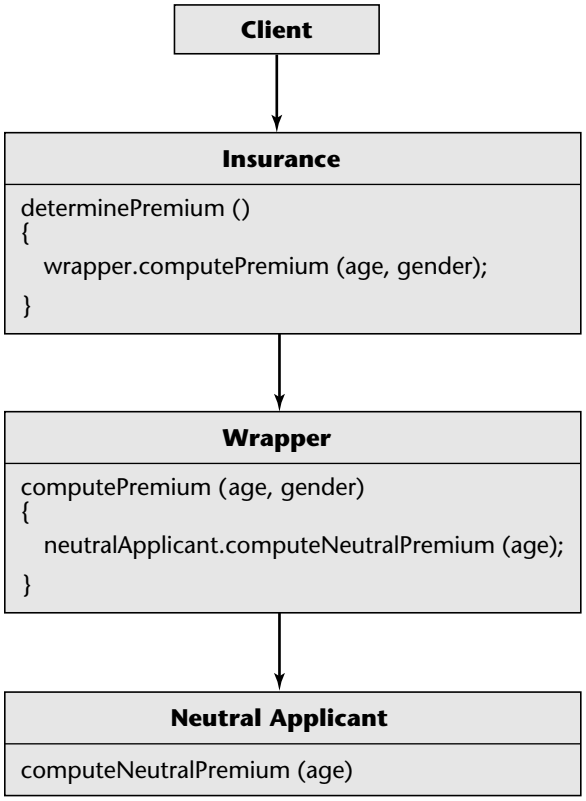
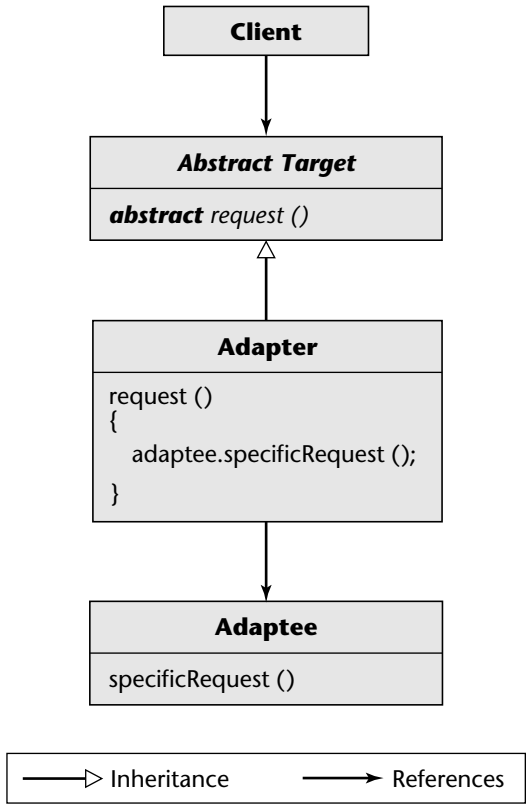


Figure 8.6 The *Adapter* design pattern.



8.6.2 *Adapter* Design Pattern

Generalizing the solution of Figure 8.5 leads to the ***Adapter design pattern*** shown in Figure 8.6 [Gamma, Helm, Johnson, and Vlissides, 1995]. In this figure, the names of abstract classes and their abstract (virtual) methods are in *sans serif italics*. (An ***abstract class*** is a class that cannot be instantiated, although it can be used as a base class. An abstract class usually contains at least one ***abstract method***, that is, a method with an interface but without an implementation.) Method *request* is defined as an abstract method of class ***Abstract Target***. It is then implemented in (concrete) class **Adapter** to send message specificRequest to an object of class **Adaptee**. This solves the implementation incompatibilities. Class **Adapter** is a concrete subclass of abstract class ***Abstract Target***, as reflected by the open arrow denoting inheritance in Figure 8.6.

Figure 8.6 depicts a general solution to the problem of permitting communication between two objects with incompatible interfaces. In fact, the *Adapter* design pattern is even more powerful than that. It provides a way for an object to permit access to its internal implementation in such a way that clients are not coupled to the structure of that internal

implementation. That is, it provides all the advantages of information hiding (Section 7.6) without having to actually hide the implementation details.

We now turn to the *Bridge* design pattern.

8.6.3 *Bridge* Design Pattern

The aim of the ***Bridge design pattern*** is to decouple an abstraction from its implementation so that the two can be changed independently of one another. The *Bridge* pattern is sometimes called a ***driver*** (for example, a printer driver or video driver).

Suppose that part of a design is hardware-dependent, but the rest is not. The design then consists of two pieces. Those parts of the design that are hardware-dependent are put on one side of the bridge, the hardware-independent pieces on the other side. In this way, the abstract operations are uncoupled from the hardware-dependent parts; there is a "bridge" between the two parts. Now, if the hardware changes, the modifications to the design and the code are localized to only one side of the bridge. The *Bridge* design pattern can therefore be viewed as a way of achieving information hiding via encapsulation.

This is shown in Figure 8.7. The implementation-independent piece is in classes ***Abstract Conceptualization*** and **Refined Conceptualization**, and the implementation-dependent piece is in classes ***Abstract Implementation*** and **Concrete Implementation**.

The *Bridge* design pattern is also useful for decoupling operating system–dependent pieces or compiler-dependent pieces, thereby supporting multiple implementations. This is shown in Figure 8.8.

Figure 8.7 The *Bridge* design pattern.

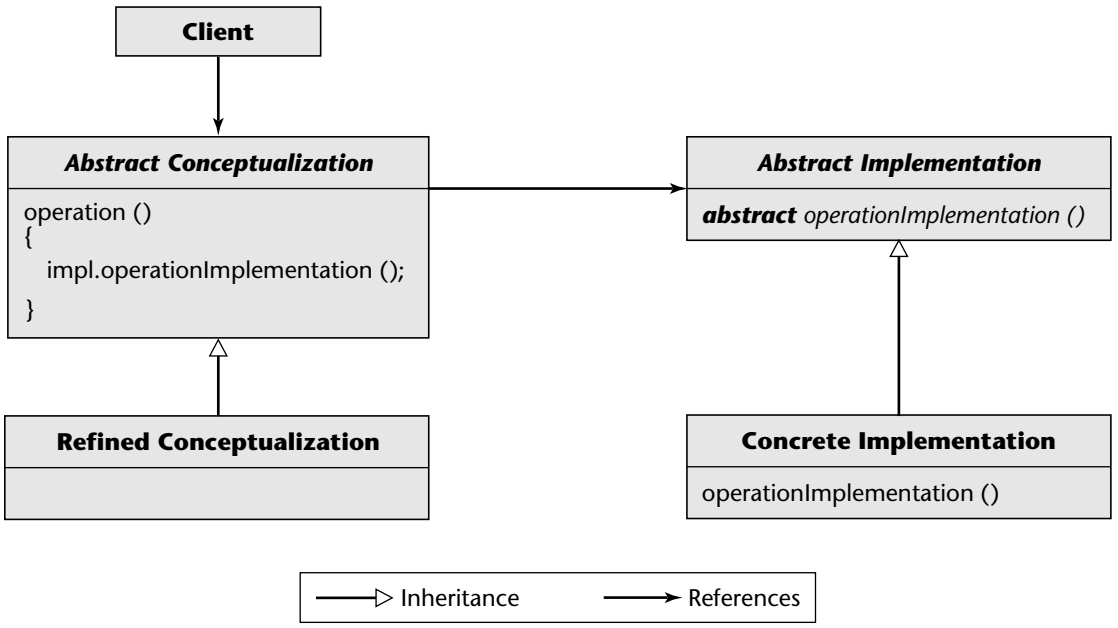


Figure 8.8 Using the *Bridge* design pattern to support multiple implementations.

Figure 8.9 The *Iterator* design pattern.

8.6.4 *Iterator* Design Pattern

An ***aggregate*** object (or ***container*** or ***collection***) is an object that contains other objects grouped together as a unit. Examples include a linked list and a hash table. An ***iterator*** is a programming construct that allows a programmer to traverse the elements of an aggregate object without exposing the implementation of that aggregate. An iterator is frequently referred to as a ***cursor***, especially within a database context.

An iterator may be viewed as a pointer with two main operations: ***element access***, or referencing a specific element in the collection; and ***element traversal***, or modifying itself so it points to the next element in the collection.

A well-known example of an iterator is a television remote control. Every remote control has a key (often labeled Up or ▲) that increases the channel number by 1 and a key (often labeled Down or ▼) that decreases the channel number by 1. The remote control increases or decreases the channel number without the viewer having to specify (or even having to know) the current channel number, let alone the program that is being carried on that channel. That is, the device implements element traversal without exposing the implementation of the aggregate.

The ***Iterator design pattern*** is shown in Figure 8.9. A Client object deals with only the ***Abstract Aggregate*** and ***Abstract Iterator*** (essentially an interface). The Client object asks the ***Abstract Aggregate*** object to create an iterator for the **Concrete Aggregate** object and then utilizes the returned **Concrete Iterator** to traverse the contents of the aggregate. The ***Abstract Aggregate*** object has to have an abstract method, *createIterator*, as a way of returning an iterator to the Client object within the application program, whereas the ***Abstract Iterator*** interface needs to define only the basic four traversal operations, abstract methods *first*, *next*, *isDone*, and *currentItem*. Implementation of these five methods is achieved at the next level of abstraction, in **Concrete Aggregate** (createIterator) and **Concrete Iterator** (first, next, isDone, and currentItem).

The key aspect of the *Iterator* design pattern is that implementation details of the elements are hidden from the iterator itself. Accordingly, we can use an iterator to process every element in a collection, independently of the implementation of the container of the elements.

Furthermore, the pattern allows different traversal methods. It even allows multiple traversals to be in progress concurrently, and these traversals can be achieved without having the specific operations listed in the interface. Instead, we have one uniform interface, namely, the four abstract operations *first*, *next*, *isDone*, and *currentItem* in ***Abstract Iterator***, with the specific traversal method(s) implemented in **Concrete Iterator**.

8.6.5 *Abstract Factory* Design Pattern

Suppose that a software organization wishes to build a widget generator, a tool that assists developers in constructing a graphical user interface. Instead of having to develop the various ***widgets*** (such as windows, buttons, menus, sliders, and scroll bars) from scratch, a developer can use the set of classes created by the widget generator that define the widgets to be utilized within the application program.

The problem is that the application program (and, therefore, the widgets) may have to run under many different operating systems, including Linux, Mac OS, and Windows. The widget generator is to support all three operating systems. However, if the widget generator hard-codes routines that run under one specific system into an application program,

it will be difficult to modify that application program in the future, replacing the generated routines with different routines that run under a different operating system. For example, suppose that the application program is to run under Linux. Then, every time a menu is to be generated, message create Linux menu is sent. However, if that application program now needs to run under Mac OS, every instance of create Linux menu must be replaced by create Mac OS menu. For a large application program, such a conversion from Linux to Mac OS is laborious and fault prone.

The solution is to design the widget generator in such a way that the application program is uncoupled from the specific operating system. This can be achieved using the ***Abstract Factory design pattern*** [Gamma, Helm, Johnson, and Vlissides, 1995]. Figure 8.10 shows the resulting design of the graphical user interface toolkit. Again, the names of abstract classes and their abstract (virtual) methods are in *sans serif italics*. At the top of Figure 8.10 is abstract class ***Abstract Widget Factory***. This abstract class contains numerous abstract methods; for simplicity, only two are shown here: *create menu* and *create window*. Moving down in the figure, **Linux Widget Factory**, **Mac OS Widget Factory**, and **Windows Widget Factory** are concrete subclasses of ***Abstract Widget Factory***. Each class contains the specific methods for creating widgets that run under a given operating system. For example, create menu within **Linux Widget Factory** causes a menu object to be created that will run under Linux.

There are also abstract classes for each widget. Two are shown here, ***Abstract Menu*** and ***Abstract Window***. Each has concrete subclasses, one for each of the three operating systems. For example, **Linux Menu** is one concrete subclass of ***Abstract Menu***. Method create menu within concrete subclass **Linux Widget Factory** causes an object of type **Linux Menu** to be created.

To create a window, a **Client** object within the application program need only send a message to abstract method *create window* of ***Abstract Widget Factory*** and polymorphism ensures that the correct widget is created. Suppose that the application program has to run under Linux. First, an object Widget Factory of type (class) **Linux Widget Factory** is created. Then a message to virtual (abstract) method *create window* of ***Abstract Widget Factory*** passing Linux as a parameter is interpreted as a message to method create window within concrete subclass **Linux Widget Factory**. Method create window in turn sends a message to create a **Linux Window**; this is indicated by the leftmost vertical dashed line in Figure 8.10.

The critical aspect of this figure is that the three interfaces between the **Client** within the application program and the widget generator, classes ***Abstract Widget Factory***, ***Abstract Menu***, and ***Abstract Window***, all are abstract classes. None of these interfaces is specific to any one operating system because the methods of the abstract classes are **abstract** (**virtual** in C++). Consequently, the design of Figure 8.10 indeed has uncoupled the application program from the operating system.

The design of Figure 8.10 is an instance of the *Abstract Factory* design pattern shown in Figure 8.11. To use this pattern, specific classes replace the generic names like **Concrete Factory 2** and **Product B3**. That is why Figure 8.2(c), the symbolic representation of a design pattern, contains white rectangles within the shaded rectangles; the white rectangles represent the details that have to be supplied to reuse this pattern in a design.

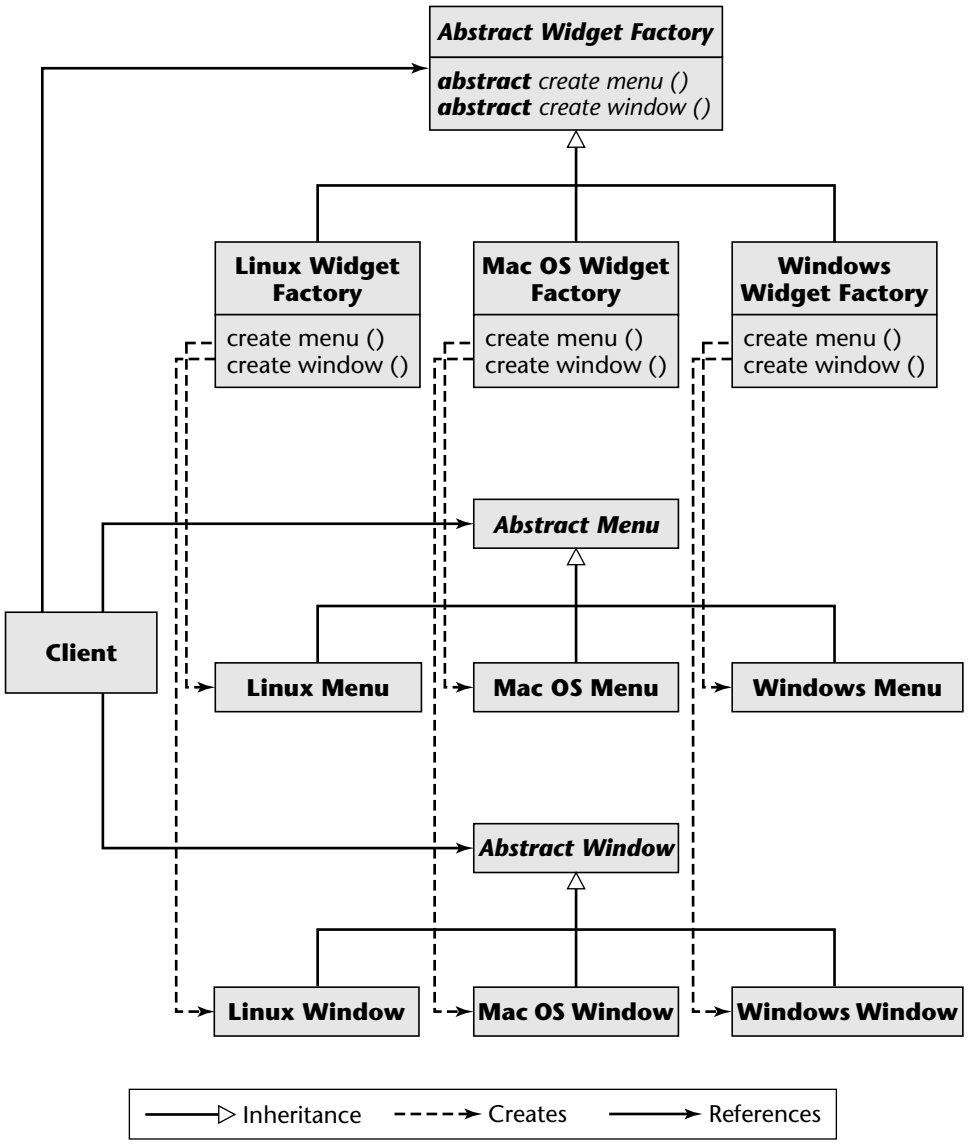


Figure 8.10 Design of graphical user interface toolkit. The names of abstract classes and their virtual functions are italicized.

8.7 Categories of Design Patterns

The definitive list of 23 design patterns given in [Gamma, Helm, Johnson, and Vlissides, 1995] is presented in Figure 8.12. The patterns are divided into three categories: creational patterns, structural patterns, and behavioral patterns. ***Creational design patterns*** solve design problems by creating objects; the *Abstract Factory* pattern (Section 8.6.5) is an example. ***Structural design patterns*** solve design problems by identifying a simple

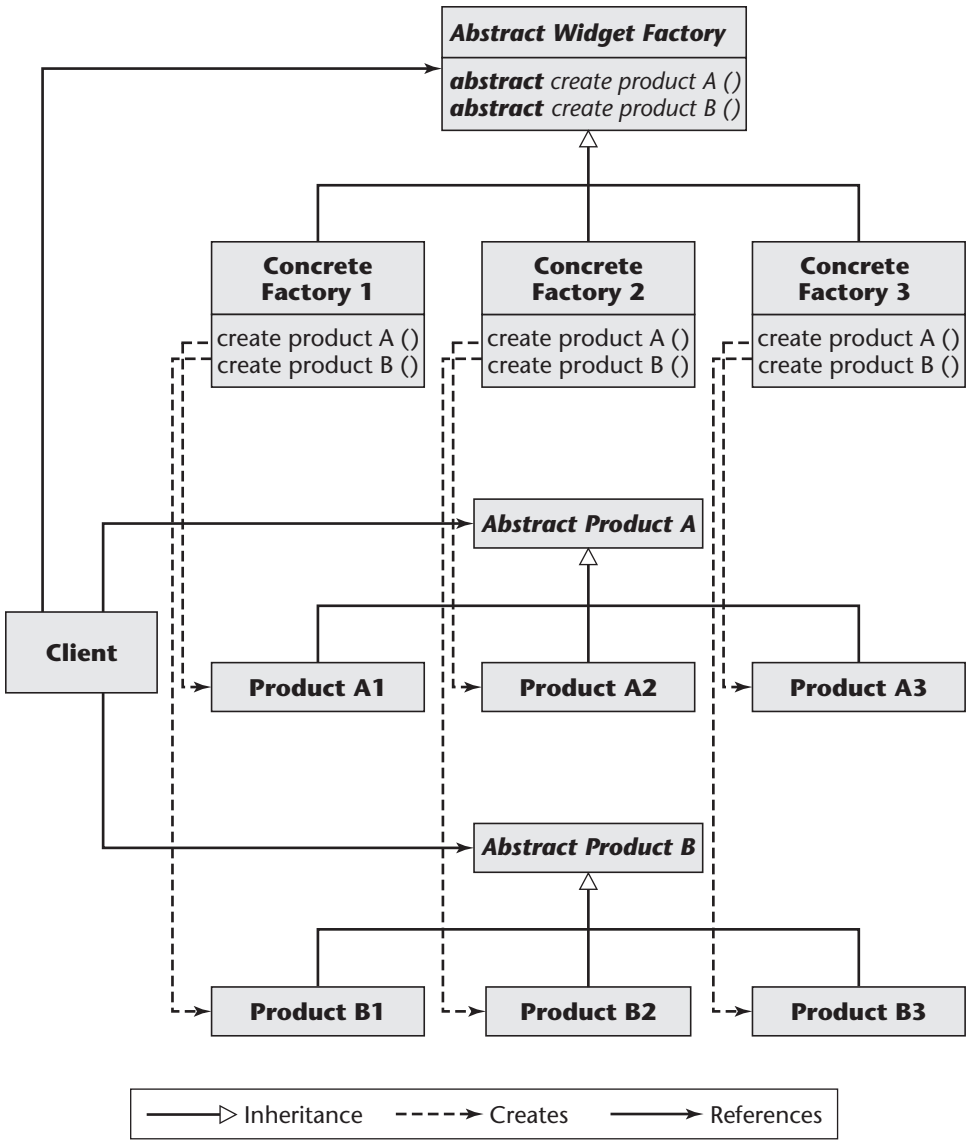


Figure 8.11
Abstract Factory design pattern. The names of abstract classes and their virtual functions are italicized.

way to realize relationships between entities. Examples include the *Adapter* pattern (Section 8.6.2) and the *Bridge* pattern (Section 8.6.3). Finally, ***behavioral design patterns*** solve design problems by identifying common communication patterns between objects. An example of this type of design pattern is the *Iterator* pattern (Section 8.6.4).

Many other lists of design patterns, organized into a variety of different categories, have been put forward. These categories are either for design patterns in general, or for specific domains, such as design patterns for Web pages or computer games. However, these alternative lists of patterns have not been widely accepted.

Creational patterns	
Abstract factory	Creates an instance of several families of classes (Section 8.6.5)
Builder	Allows the same construction process to create different representations
Factory method	Creates an instance of several possible derived classes
Prototype	A class to be cloned
Singleton	Restricts instantiation of a class to a single instance
Structural patterns	
Adapter	Matches interfaces of different classes (Section 8.6.2)
Bridge	Decouples an abstaction from its implementation (Section 8.6.3)
Composite	A class that is a composition of similar classes
Decorator	Allows additional behavior to be dynamically added to a class
Façade	A single class that provides a simplified interface
Flyweight	Uses sharing to support large numbers of fine-grained classes efficiently
Proxy	A class functioning as an interface
Behavioral patterns	
Chain-of-responsibility	A way of processing a request by a chain of classes
Command	Encapsulates an action within a class
Interpreter	A way to implement specialized language elements
Iterator	Sequentially accesses the elements of a collection (Section 8.6.4)
Mediator	Provides a unified interface to a set of interfaces
Memento	Captures and restores an object's internal state
Observer	Allows the observation of the state of an object at run time
State	Allows an object to partially change its type at run time
Strategy	Allows an algorithm to be dynamically selected at run time
Template method	Defers implementations of an algorithm to its subclasses
Visitor	Adds new operations to a class without changing it

Figure 8.12 The 23 design patterns listed in [Gamma, Helm, Johnson, and Vlissides, 1995].

8.8 Strengths and Weaknesses of Design Patterns

Design patterns have many strengths:

1. As pointed out in Section 8.5.3, design patterns promote reuse by solving a general design problem. The reusability of a design pattern can be enhanced by careful incorporation of features that can be used to further enhance reuse, such as inheritance.
2. A design pattern provides high-level documentation of the design, because patterns specify design abstractions.
3. Implementations of many design patterns exist. In such cases, there is no need to code or document those parts of a program that implement design patterns. (Testing of those parts of the program is still essential, of course.)
4. If a maintenance programmer is familiar with design patterns, it will be easier to comprehend a program that incorporates design patterns, even if he or she has never seen that specific program before.

However, design patterns have a number of weaknesses, too:

1. The use of the 23 standard design patterns in [Gamma, Helm, Johnson, and Vlissides, 1995] in a software product may be an indication that the language we are using is not powerful enough. Norwig [1996] examined the C++ implementations of those patterns and found that 16 out of the 23 have simpler implementations in Lisp or Dylan than in C++, for at least some uses of each pattern.

2. A major problem is that there is as yet no systematic way to determine when and how to apply design patterns. Design patterns are still described informally, using natural language text. Accordingly, we have to decide manually when to apply a pattern; a CASE tool (Chapter 5) cannot be used.
3. To obtain maximal benefit from design patterns, multiple interacting patterns are employed. For example, as stated in Section 8.5.3, a case study of a document editor in [Gamma, Helm, Johnson, and Vlissides, 1995] contains eight interacting patterns. As already pointed out, we do not yet have a systematic way of knowing when and how to use one pattern, let alone multiple interacting patterns.
4. When performing maintenance on a software product built using the classical paradigm, it is essentially impossible to retrofit classes and objects. It is similarly all but impossible to retrofit patterns to an existing software product, whether classical or object-oriented.

However, the weaknesses of design patterns are outweighed by their strengths. Furthermore, once current research efforts to formalize and hence automate design patterns have succeeded, patterns will be much easier to use than at present.

8.9 Reuse and Postdelivery Maintenance

The traditional reason for promoting reuse is that it can shorten the development process. For example, a number of major software organizations are trying to halve the time needed to develop a new product, and reuse is a primary strategy in these endeavors. However, as reflected in Figure 1.3, for every $1 spent on developing a product, $2 or more are spent on maintaining that product. Therefore, a second important reason for reuse is to reduce the time and cost of maintaining a product. In fact, reuse has a greater impact on postdelivery maintenance than on development.

Suppose now that 40 percent of a product consists of components reused from earlier products and this reuse is evenly distributed across the entire product. That is, 40 percent of the specification artifacts consist of reused components, 40 percent of the design artifacts, 40 percent of the code artifacts, 40 percent of the manuals, and so on. Unfortunately, this does not mean that the time to develop the product as a whole will be 40 percent less than it would have been without reuse. First, some of the components have to be tailored to the new product. Suppose that one-quarter of the reused components are changed. If a component has to be changed, then the documentation for that component also has to be changed. Furthermore, the changed component has to be tested. Second, if a code artifact is reused unchanged, then unit testing of that code artifact is not required. However, integration testing of that code artifact still is needed. So, even if 30 percent of a product consists of components reused unchanged and a further 10 percent are reused changed, the time needed to develop the complete product at best is only about 27 percent less [Schach, 1992]. Suppose that, as in Figure 1.3(a), 33 percent of a software budget is devoted to development. Then, if reuse reduces development costs by 27 percent, the overall cost of that product over its 12- to 15-year lifetime is reduced by only about 9 percent as a consequence of reuse; this is reflected in Figure 8.13.

Similar but lengthier arguments can be applied to the postdelivery maintenance component of the software process [Schach, 1994]. Under the assumptions of the previous paragraph, the effect of reuse on postdelivery maintenance is an overall cost saving of about

Figure 8.13 Average percentage cost savings under the assumption that 40 percent of a new product consists of reused components, three-quarters of which are reused unchanged.

Activity	Percentage of Total Cost over Product Lifetime	Percentage Savings over Product Lifetime due to Reuse
Development	33%	9.3%
Postdelivery maintenance	67	17.9

18 percent, as shown in Figure 8.13. Clearly, the major impact of reuse is on postdelivery maintenance rather than development. The underlying reason is that reused components generally are well designed, thoroughly tested, and comprehensively documented, thereby simplifying all three types of postdelivery maintenance.

If the actual reuse rates in a given product are lower (or higher) than assumed in this section, then the benefits of reuse are different. But the overall result is still the same: Reuse affects postdelivery maintenance more than it does development.

We turn now to portability.

8.10 Portability

The ever-rising cost of software makes it imperative that some means be found to contain costs. One way is to ensure that the product as a whole can be adapted easily to run on a variety of different hardware–operating system combinations. Some of the cost of writing the product may then be recouped by selling versions that run on other computers. But the most important reason for writing software that can be implemented easily on other computers is that, every 4 years or so, the client organization purchases new hardware, and all its software then must be converted to run on the new hardware. A product is considered portable if it is significantly less expensive to adapt the product to run on the new computer than to write a new product from scratch [Mooney, 1990].

More precisely, *portability* may be defined as follows: Suppose a product P is compiled by compiler C and then runs on the ***source computer***, namely, hardware configuration H under operating system O. A product P' is needed that functionally is equivalent to P but must be compiled by compiler C' and run on the ***target computer***, namely, hardware configuration H' under operating system O'. If the cost of converting P into P' is significantly less than the cost of coding P' from scratch, then P is said to be *portable*.

Overall, the problem of porting software is nontrivial because of incompatibilities among different hardware configurations, operating systems, and compilers. Each of these aspects is examined in turn.

8.10.1 Hardware Incompatibilities

Product P currently running on hardware configuration H is to be installed on hardware configuration H'. Superficially, this is simple; copy P from the hard drive of H onto DAT tape and transfer it to H'. However, this will not work if H' uses a Zip drive for backup; DAT tape cannot be read on a Zip drive.

Suppose now that the problem of physically copying the source code of product *P* to computer *H′* has been solved. There is no guarantee that *H′* can interpret the bit patterns created by *H*. A number of different character codes exist, the most popular of which are Extended Binary Coded Decimal Interchange Code (EBCDIC) and American Standard Code for Information Interchange (ASCII), the American version of the 7-bit ISO code [Mackenzie, 1980]. If *H* uses EBCDIC but *H′* uses ASCII, then *H′* will treat *P* as so much garbage.

Although the original reason for these differences is historical (that is, researchers working independently for different manufacturers developed different ways of doing the same thing), there are definite economic reasons for perpetuating them. To see this, consider the following imaginary situation. MCM Computer Manufacturers has sold thousands of its MCM-1 computer. MCM now wishes to design, manufacture, and market a new computer, the MCM-2, which is more powerful in every way than the MCM-1 but costs considerably less. Suppose further that the MCM-1 uses ASCII code and has 36-bit words consisting of four 9-bit bytes. Now, the chief computer architect of MCM decides that the MCM-2 should employ EBCDIC and have 16-bit words consisting of two 8-bit bytes. The sales force then has to tell current MCM-1 owners that the MCM-2 is going to cost them $35,000 less than any competitor's equivalent machine but will cost them up to $200,000 to convert existing software and data from MCM-1 format to MCM-2 format. No matter how good the scientific reasons for redesigning the MCM-2, marketing considerations will ensure that the new computer is compatible with the old one. A salesperson then can point out to an existing MCM-1 owner that, not only is the MCM-2 computer $35,000 less expensive than any competitor's machine, but any customer ill-advised enough to buy from a different manufacturer will be spending $35,000 too much and also will have to pay some $200,000 to convert existing software and data to the format of the non-MCM machine.

Moving from the preceding imaginary situation to the real world, the most successful line of computers to date has been the IBM System/360–370 series [Gifford and Spector, 1987]. The success of this line of computers is due largely to full compatibility between machines; a product that runs on an IBM System/360 Model 30 built in 1964 runs unchanged on an IBM eServer zSeries 990 built in 2007. However, the product that runs on the IBM System/360 Model 30 under OS/360 may require considerable modification before it can run on a totally different 2007 machine, such as a Sun Fire E25K server under Solaris. Part of the difficulty may be due to hardware incompatibilities. But part may be caused by operating system incompatibilities.

8.10.2 Operating System Incompatibilities

The job control languages (JCLs) of any two computers usually are vastly different. Some of the difference is syntactic—the command for executing an executable load image might be `@xeq` on one computer, `//xqt` on another, and `.exc` on a third. When porting a product to a different operating system, syntactic differences are relatively straightforward to handle by simply translating commands from the one JCL into the other. But other differences can be more serious. For example, some operating systems support virtual memory. Suppose that a certain operating system allows products to be up to 1024 MB in size, but the actual area of main memory allocated to a particular product may be only 64 MB. What happens is that the user's product is partitioned into pages 2048 KB in size, and only 32 of these

pages can be in main memory at any one time. The rest of the pages are stored on disk and swapped in and out as needed by the virtual memory operating system. As a result, products can be written with no effective constraints as to size. But, if a product that has been successfully implemented under a virtual memory operating system is to be ported to an operating system with physical constraints on product size, the entire product may have to be rewritten and then linked using overlay techniques to ensure that the size limit is not exceeded.

8.10.3 Numerical Software Incompatibilities

When a product is ported from one machine to another or even compiled using a different compiler, the results of performing arithmetic may differ. On a 16-bit machine, that is, a computer with a word size of 16 bits, an integer ordinarily is represented by one word (16 bits) and a double-precision integer by two adjacent words (32 bits). Unfortunately, some language implementations do not include double-precision integers. For example, standard Pascal does not include double-precision integers. Therefore, a product that functions perfectly on a compiler–hardware–operating system configuration in which Pascal integers are represented using 32 bits may fail to run correctly when ported to a computer in which integers are represented by only 16 bits. The obvious solution—representing integers larger than 2^{16} by floating-point numbers (type **real**)—does not work because integers are represented exactly whereas floating-point numbers in general are only approximated using a mantissa (fraction) and exponent.

This problem can be solved in Java, because each of the eight primitive data types has been carefully specified. For example, type **int** always is implemented as a signed 32-bit two's complement integer, and type **float** always occupies 32 bits and satisfies ANSI/IEEE (Standard) 754 [1985] for floating-point numbers. The problem of ensuring that a numerical computation is performed correctly on every target hardware–operating system therefore cannot arise in Java. (For more insights into the design of Java, see Just in Case You Wanted to Know Box 8.5.) However, where a numerical computation is performed in a language other than Java, it is important, but often difficult, to ensure that numerical computations are performed correctly on the target hardware–operating system.

8.10.4 Compiler Incompatibilities

Portability is difficult to achieve if a product is implemented in a language for which few compilers exist. If the product has been implemented in a specialized language such as CLU [Liskov, Snyder, Atkinson, and Schaffert, 1977], it may be necessary to rewrite it in a different language if the target computer has no compiler for that language. On the other hand, if a product is implemented in a popular object-oriented language such as C++ or Java, the chances are good that a compiler or interpreter for that language can be found for a target computer.

Suppose that a product is written in an object-oriented language such as standard Fortran, Fortran 2003 (see Just in Case You Wanted to Know Box 8.6 for more on the name "Fortran 2003"). In theory, there should be no problem in porting the product from one machine to another—after all, standard Fortran is standard Fortran. Regrettably, that is not the case; in practice, there is no such thing as standard Fortran. Even though there is an ISO/

Just in Case You Wanted to Know Box 8.5

In 1991, James Gosling of Sun Microsystems developed Java. While developing the language, he frequently stared out the window at a large oak tree outside his office. In fact, he did this so often that he decided to name his new language *Oak*. However, his choice of name was unacceptable to Sun because it could not be trademarked, and without a trademark Sun would lose control of the language.

After an intensive search for a name that could be trademarked and was easy to remember, Gosling's group came up with *Java*. During the 18th century, much of the coffee imported into England was grown in Java, the most populous island in the Dutch East Indies (now Indonesia). As a result, *Java* now is a slang word for coffee, the third most popular beverage among software engineers. Unfortunately, the names of the Big Two carbonated cola beverages are already trademarked.

To understand why Gosling designed Java, it is necessary to appreciate the source of the weaknesses he perceived in C++. And, to do that, we have to go back to C, the parent language of C++.

In 1972, the programming language C was developed by Dennis Ritchie at AT&T Bell Laboratories (now Lucent Technologies) for use in systems software. The language was designed to be extremely flexible. For example, it permits arithmetic on pointer variables, that is, on variables used to store memory addresses. From the viewpoint of the average programmer, this poses a distinct danger; the resulting programs can be extremely insecure because control can be passed to anywhere in the computer. Also, C does not embody arrays as such. Instead, a pointer to the address of the beginning of the array is used. As a result, the concept of an out-of-range array subscript is not intrinsic to C. This is a further source of possible insecurity.

These and other insecurities were no problem at Bell Labs. After all, C was designed by an experienced software engineer for use by other experienced software engineers at Bell Labs. These professionals could be relied on to use the powerful and flexible features of C in a secure way. A basic philosophy in the design of C was that the person using C knows exactly what he or she is doing. Software failures that occurred when C is used by less competent or inexperienced programmers should not be blamed on Bell Labs; there never was any intent that C should be widely employed as a general-purpose programming language, as it is today.

IEC standard for Fortran 2003 [ISO/IEC 1539–1, 2004], there is no reason for a compiler writer to adhere to it. For example, a decision may be taken to support additional features not usually found in Fortran 2003 so that the marketing division can tout a "new, extended Fortran compiler." Conversely, a compiler for a small embedded microprocessor may not be a full Fortran implementation. Also, with a deadline to produce a compiler, management may decide to bring out a less-than-complete implementation, intending to support the full standard in a later revision. Suppose that the compiler on the source computer supports a superset of Fortran 2003. Suppose further that the compiler on the target computer is an implementation of standard Fortran 2003. When a product implemented on that source computer is ported to the target, any portions of the product that use nonstandard Fortran 2003 constructs from the superset have to be recoded. Therefore, to ensure portability, programmers should use only standard Fortran language features.

Early COBOL standards were developed by the COnference on DAta SYstems Languages (CODASYL), a committee of American computer manufacturers and government and private users. Joint Technical Committee 1 of Subcommittee 22 of the International Organization for Standardization (ISO) and the International Electrotechnical

With the rise of the object-oriented paradigm, a number of object-oriented programming languages based on C were developed, including Object C, Objective C, and C++. The idea behind these languages was to embed object-oriented constructs within C, which by then was a popular programming language. It was argued that it would be easier for programmers to learn a language based on a familiar language than to learn a totally new syntax. However, only one of the many C-based object-oriented languages became widely accepted, C++, developed by Bjarne Stroustrup, also of AT&T Bell Laboratories.

It has been suggested that the reason behind the success of C++ was the enormous financial clout of AT&T (now part of SBC Communications). However, if corporate size and financial strength were relevant features in promoting a programming language, today we would all be using PL/I, a language developed and strongly promoted by IBM. The reality is that PL/I, notwithstanding the prestige of IBM, has retreated into obscurity. The real reason for the success of C++ is that it is a true superset of C. That is, unlike any of the other C-based object-oriented programming languages, virtually any C program is also valid C++. Therefore, organizations realized that they could switch from C to C++ without changing any of their existing C software. They could advance from the classical paradigm to the object-oriented paradigm without disruption. A remark frequently encountered in the Java literature is, "Java is what C++ should have been." The implication is that, if only Stroustrup had been as smart as Gosling, C++ would have turned out to be Java. On the contrary, if C++ had not been a true superset of C, it would have gone the way of all other C-based object-oriented programming languages; that is, it essentially would have disappeared. Only after C++ had taken hold as a popular language was Java designed in reaction to perceived weaknesses in C++. Java is not a superset of C; for example, Java has no pointer variables. Therefore, it would be more accurate to say that "Java is what C++ could not possibly have been."

Finally, it is important to realize that Java, like every other programming language, has weaknesses of its own. In addition, in some areas (such as access rules), C++ is superior to Java [Schach, 1997]. It will be interesting to see, in the coming years, whether C++ continues to be the predominant object-oriented programming language or whether it is supplanted by Java or some other language.

Commission (IEC) now are responsible for COBOL standards [Schricker, 2000]. Unfortunately, COBOL standards do not promote portability. A COBOL standard has an official life of 5 years, but each successive standard is not necessarily a superset of its predecessor. Equally worrisome is that many features are left to the individual implementer, subsets may be termed *standard COBOL*, and there is no restriction on extending the language to form a superset. OO-COBOL [ISO/IEC 1989, 2002], the language of the current COBOL standard, is object oriented, as is Fortran 2003 [ISO/IEC 1539–1, 2004].

The standard for C++ [ISO/IEC 14882, 1998] was unanimously approved by the various national standards committees (including ANSI) in November 1997. The standard received final ratification in 1998.

The only truly successful language standard so far has been the Ada 83 standard, embodied in the Ada Reference Manual [ANSI/MIL-STD-1815A, 1983]. (For background information on Ada, see Just in Case You Wanted to Know Box 8.6.) Until the end of 1987, the name Ada was a registered trademark of the U.S. government, Ada Joint Program Office (AJPO). As owner of the trademark, the AJPO stipulated that the name Ada legally could be used only for language implementations that complied exactly with the standard; subsets and supersets

Just in Case You Wanted to Know Box 8.6

Names of programming languages are spelled in uppercase when the name is an acronym. Examples include ALGOL (ALGOrithmic Language), COBOL (COmmon Business Oriented Language), and FORTRAN (FORmula TRANslator). Conversely, all other programming languages begin with an uppercase letter and the remaining letters in the name (if any) are in lowercase. Examples include Ada, C, C++, Java, and Pascal. *Ada* is not an acronym; the language was named after Ada, Countess of Lovelace (1815–1852). Daughter of the poet Lord Alfred Byron, Ada was the world's first programmer by virtue of her work on Charles Babbage's difference engine. *Pascal* is not an acronym either—this language was named after the French mathematician and philosopher, Blaise Pascal (1623–1662). And I am sure that you have read all about the name *Java* in Just in Case You Wanted to Know Box 8.5.

There is one exception: Fortran. The FORTRAN Standards Committee decided that, effective with the 1990 version, the name of the language would thenceforth be written *Fortran*.

were expressly forbidden. A mechanism was set up for validating Ada compilers, and only a compiler that successfully passed the validation process could be called an Ada compiler. Consequently, the trademark was used as a means of enforcing standards and hence portability.

Now that the name Ada no longer is a trademark, enforcement of the standard is being achieved via a different mechanism. There is little or no market for an Ada compiler that has not been validated. Therefore, strong economic forces encourage Ada compiler developers to have their compilers validated and hence certified as conforming to the Ada standard. This has applied to compilers for both Ada 83 [ANSI/MIL-STD-1815A, 1983] and Ada 95 [ISO/IEC 8652, now 1995]; the latter is object oriented.

For Java to be a totally portable language, it is essential for the language to be standardized and to ensure that the standard is strictly obeyed. Sun Microsystems, like the Ada Joint Program Office, uses the legal system to achieve standardization. As mentioned in Just in Case You Wanted to Know Box 8.5, Sun chose a name for its new language that could be copyrighted so that Sun could enforce its copyright and bring legal action against alleged violators (which happened when Microsoft developed nonstandard Java classes). After all, portability is one of the most powerful features of Java. If multiple versions of Java are permitted, the portability of Java suffers; Java can be truly portable only if every Java program is handled identically by every Java compiler. To try to influence public opinion, in 1997 Sun ran a "Pure Java" advertising campaign.

Version 1.0 of Java was released early in 1997. A series of revised versions followed in response to comments and criticisms. The latest version at the time of writing is Java J2SE (Java 2 Platform, Standard Edition), version 5.0. This process of stepwise refinement of Java will continue. When the language eventually stabilizes, it is likely that a standards organization such as ANSI or ISO will publish a draft standard and elicit comments from all over the world. These comments will be used to put together the official Java standard.

8.11 Why Portability?

In the light of the many barriers to porting software, the reader might well wonder if it is worthwhile to port software at all. An argument in favor of portability stated in Section 8.10 is that the cost of software may be partially recouped by porting the product to a different hardware–operating system configuration. However, selling multiple variants of the

software may not be possible. The application may be highly specialized, and no other client may need the software. For instance, a management information system written for one major car rental corporation may simply be inapplicable to the operations of other car rental corporations. Alternatively, the software itself may give the client a competitive advantage, and selling copies of the product would be tantamount to economic suicide. In the light of all this, is it not a waste of time and money to engineer portability into a product when it is designed?

The answer to this question is an emphatic *No*. The major reason why portability is essential is that the life of a software product generally is longer than the life of the hardware for which it was first written. Good software products can have a life of 15 years or more, whereas hardware frequently is changed every 4 years. Therefore, good software can be implemented, over its lifetime, on three or more different hardware configurations.

One way to solve this problem is to buy upwardly compatible hardware. The only expense is the cost of the hardware; the software need not be changed. Nevertheless, in some cases it may be economically more sound to port the product to different hardware entirely. For example, the first version of a product may have been implemented 7 years ago on a mainframe. Although it may be possible to buy a new mainframe on which the product can run with no changes, it may be considerably less expensive to implement multiple copies of the product on a network of personal computers, one on the desk of each user. In this instance, if the software has been written in a way that would promote portability, then porting the product to the personal computer network makes good financial sense.

But there are other kinds of software. For example, many organizations that write software for personal computers make their money by selling multiple copies of COTS software. For instance, the profit on a spreadsheet package is small and cannot possibly cover the cost of development. To make a profit, 50,000 (or even 500,000) copies may have to be sold. After this point, additional sales are pure profit. So, if the product can be ported to additional types of hardware with ease, even more money can be made.

Of course, as with all software, the product is not just the code but also the documentation, including the manuals. Porting the spreadsheet package to other hardware means changing the documentation as well. Therefore, portability also means being able to change the documentation easily to reflect the target configuration, instead of having to write new documentation from scratch. Considerably less training is needed if a familiar, existing product is ported to a new computer than if a completely new product were to be written. For this reason, too, portability is to be encouraged.

Techniques to facilitate portability now are described.

8.12 Techniques for Achieving Portability

One way to try to achieve portability is to forbid programmers to use constructs that might cause problems when ported to another computer. For example, an obvious principle would seem to be this: Write all software in a standard version of a high-level programming language. But how is a portable operating system to be written? After all, it is inconceivable that an operating system could be written without at least some assembler code. Similarly, a compiler has to generate object code for a specific computer. Here, too, it is impossible to avoid all implementation-dependent components.

8.12.1 Portable System Software

Instead of forbidding all implementation-dependent aspects, which would prevent almost all system software from being written, a better technique is to isolate any necessary implementation-dependent pieces. An example of this technique is the way the original UNIX operating system was constructed [Johnson and Ritchie, 1978]. About 9000 lines of the operating system were written in C. The remaining 1000 lines constituted the kernel. The kernel was written in assembler and had to be rewritten for each implementation. About 1000 lines of the C code consisted of device drivers; this code, too, had to be rewritten each time. However, the remaining 8000 lines of C code remained largely unchanged from implementation to implementation.

Another useful technique for increasing the portability of system software is to use levels of abstraction (Section 7.4.1). Consider, for example, graphical display routines for a workstation. A user inserts a command such as drawLine into his or her source code. The source code is compiled and then linked with graphical display routines. At run time, drawLine causes the workstation to draw a line on the screen as specified by the user. This can be implemented using two levels of abstraction. The upper level, written in a high-level language, interprets the user's command and calls the appropriate lower-level code artifact to execute that command. If the graphical display routines are ported to a new type of workstation, then no changes need be made to the user's code or the upper level of the graphical display routines. However, the lower-level code artifacts of the routines have to be rewritten, because they interface with the actual hardware, and the hardware of the new workstation is different from that of the workstation on which the package was previously implemented. This technique also has been used successfully for porting communications software that conforms to the seven levels of abstraction of the ISO-OSI model [Tanenbaum, 2002].

8.12.2 Portable Application Software

With regard to application software, rather than system software such as operating systems and compilers, it generally is possible to write the product in a high-level language. Section 13.1 points out that frequently no choice can be made with regard to implementation language, but that when it is possible to select a language, the choice should be made on the basis of cost–benefit analysis (Section 5.2). One factor that must enter into the cost–benefit analysis is the impact on portability.

At every stage in the development of a product, decisions can be made that result in a more portable product. For example, some compilers distinguish between uppercase and lowercase letters. For such a compiler, thisIsAName and thisisaname are different variables. But other compilers treat the two names the same. A product that relies on differences between uppercase letters and lowercase letters can lead to hard-to-discover faults when the product is ported.

Just as frequently no choice can be made of programming language, no choice may be allowed in the operating system. However, if at all possible, the operating system under which the product runs should be a popular one. This is an argument in favor of the UNIX operating system. UNIX has been implemented on a wide range of hardware. In addition, UNIX, or more precisely, UNIX-like operating systems, have been implemented on top of many mainframe operating systems. For personal computers, it remains to be seen whether

Linux will overtake Windows as the most widely used operating system. Just as use of a widely implemented programming language promotes portability, so too does use of a widely implemented operating system.

To facilitate the moving of software from one UNIX-based system to another, the Portable Operating System Interface for Computer Environments (POSIX) was developed [NIST 151, 1988]. POSIX standardizes the interface between an application program and a UNIX operating system. POSIX has been implemented on a number of non-UNIX operating systems as well, broadening the number of computers to which application software can be ported with little or no problem.

Language standards can play their part in achieving portability. If the coding standards of a development organization stipulate that only standard constructs may be used, then the resulting product is more likely to be portable. To this end, programmers must be provided a list of nonstandard features supported by the compiler but whose use is forbidden without prior managerial approval. Like other sensible coding standards, this one can be checked by machine.

Graphical user interfaces similarly are becoming portable via the introduction of standard GUI languages. Examples of these include Motif and X11. The standardization of GUI languages is in reaction to the growing importance of GUIs, and the resulting need for portability of human–computer interfaces.

It is also necessary to plan for potential lack of compatibility between the operating system under which the product is being constructed and any future operating systems to which the product may be ported. If at all possible, operating system calls should be localized to one or two code artifacts. In any event, every operating system call must be carefully documented. The documentation standard for operating system calls should assume that the next programmer to read the code will have no familiarity with the current operating system, often a reasonable assumption.

Documentation in the form of an installation manual should be provided to assist with future porting. That manual points out what parts of the product have to be changed when porting the product and what parts may have to be changed. In both instances, a careful explanation must be provided of what has to be done and how to do it. Finally, lists of changes that have to be made in other manuals, such as the user manual or the operator manual, also must appear in the installation manual.

8.12.3 Portable Data

The problem of portability of data can be vexing. Problems of hardware incompatibilities were pointed out in Section 8.10.1. But even after such problems have been solved, software incompatibilities remain. For instance, the format of an indexed-sequential file is determined by the operating system; a different operating system generally implies a different format. Many files require headers containing information such as the format of the data in that file. The format of a header almost always is unique to the specific compiler and operating system under which that file was created. The situation can be even worse when database management systems are used.

The safest way of porting data is to construct an unstructured (sequential) file, which can then be ported with minimal difficulty to the target machine. From this unstructured file, the desired structured file can be reconstructed. Two special conversion routines have to be

written, one running on the source machine to convert the original structured file into sequential form and one running on the target machine to reconstruct the structured file from the ported sequential file. Although this solution seems simple enough, the two routines are nontrivial when conversions between complex database models have to be performed.

8.12.4 Web-Based Applications

One of the greatest strengths of the World Wide Web is that Web-based applications can achieve an extremely high level of portability. First, Web-based applications can be made portable by utilizing a language like HTML (Hypertext Markup Language) [HTML, 2006] or XML (Extensible Markup Language) [XML, 2003] that can be read by any Web browser, and by employing Java applets, which can be run on virtually every client. A further degree of portability can be achieved by separating the HTML or XML interface from the rest of the program (especially the application logic). The resulting application program will then run on a server, but can be accessed via virtually any client with a Web browser, including a personal digital assistant (PDA) or cell phone. Furthermore, such an application program can be ported to a new server without changing the clients that access it.

At the time of writing, not all applications can be run with every Web browser. For example, some applications that run under Internet Explorer will not work with Firefox because Firefox conforms to the World Wide Web Consortium (W3C) standards [W3C, 2006], but Internet Explorer does not [Computer Gripes, 2004]. However, as Web technology evolves in the future, it is likely that the highest levels of portability will be attained.

We conclude this chapter with a summary of the strengths of and impediments to reuse and portability (Figure 8.14); the section in which each item is discussed is stated.

Figure 8.14 Strengths of and impediments to reuse and portability, and the section in which the topic is discussed.

Strengths	Impediments
Reuse	
Shorter development time (Section 8.1) Lower development cost (Section 8.1) Higher-quality software (Section 8.1) Shorter maintenance time (Section 8.6) Lower maintenance cost (Section 8.6)	NIH syndrome (Section 8.2) Potential quality issues (Section 8.2) Retrieval issues (Section 8.2) Cost of making a component reusable (opportunistic reuse) (Section 8.2) Cost of making a component for future reuse (systematic reuse) (Section 8.2) Legal issues (contract software only) (Section 8.2) Lack of source code for COTS components (Section 8.2)
Portability	
Software has to be ported to new hardware every 4 years or so (Section 8.11) More copies of COTS software can be sold (Section 8.11)	Potential incompatibilities: Hardware (Section 8.7.1) Operating systems (Section 8.7.2) Numerical software (Section 8.7.3) Compilers (Section 8.7.4) Data formats (Section 8.9.3)

Chapter Review

Reuse is described in Section 8.1. Various impediments to reuse are described in Section 8.2. Two reuse case studies are presented in Section 8.3. The impact of the object-oriented paradigm on reuse is analyzed in Section 8.4. Reuse during design and implementation is the subject of Section 8.5; the topics covered include frameworks, patterns, software architecture, and component-based software engineering. Design patterns are discussed in greater detail in Section 8.6; after a mini case study (Section 8.6.1), the *Adapter*, *Bridge*, *Iterator*, and *Abstract Factory* design patterns are described in Sections 8.6.2, 8.6.3, 8.6.4, and 8.6.5, respectively. Categories of design patterns are discussed in Section 8.7. Section 8.8 contains a discussion of strengths and weaknesses of design patterns. The impact of reuse on postdelivery maintenance is discussed in Section 8.9.

Portability is discussed in Section 8.10. Portability can be hampered by incompatibilities caused by hardware (Section 8.10.1), operating systems (Section 8.10.2), numerical software (Section 8.10.3), or compilers (Section 8.10.4). Nevertheless, it is extremely important to try to make all products as portable as possible (Section 8.11). Ways of facilitating portability include using popular high-level languages, isolating the nonportable pieces of a product (Section 8.12.1), adhering to language standards (Section 8.12.2), and the use of unstructured data (Section 8.12.3). The chapter concludes with a discussion of Web-based applications (8.12.4).

For Further Reading

A variety of reuse case studies can be found in [Lanergan and Grasso, 1984; Matsumoto, 1984, 1987; Selby, 1989; Prieto-Díaz, 1991; Lim, 1994; Jézéquel and Meyer, 1997; and Toft, Coleman, and Ohta, 2000]. Successful reuse experiences at four European companies are described in [Morisio, Tully, and Ezran, 2000]. The management of reuse is described in [Lim, 1998]. A search scheme for object retrieval and reuse is described in [Isakowitz and Kauffman, 1996]. The cost-effectiveness of reuse is described in [Barnes and Bollinger, 1991] and ways of identifying components for future reuse in [Caldiera and Basili, 1991]. Meyer [1996a] analyzes the claim that the object-oriented paradigm promotes reuse; four case studies in reuse and object technology appear in [Fichman and Kemerer, 1997]. Reuse metrics are discussed in [Poulin, 1997]. Factors that affect the success of reuse programs are presented in [Morisio, Ezran, and Tully, 2002]. Reuse strategies are discussed in [Ravichandran and Rothenberger, 2003]. A comprehensive model for evaluating software reuse alternatives is presented in [Tomer et al., 2004]. Further papers on reuse are to be found in the May 2000 issue of *IEEE Transactions on Software Engineering*.

A good source of information on frameworks is [Lewis et al., 1995]. D'Souza and Wills [1999] present a development methodology based on object-oriented frameworks and components. A series of articles on frameworks can be found in [Fayad and Johnson, 1999; Fayad and Schmidt, 1999; and Fayad, Schmidt, and Johnson, 1999]. The October 2000 issue of *Communications of the ACM* includes articles on component-based frameworks, including [Fingar, 2000] and [Kobryn, 2000], which describes how to model components and frameworks using UML. Achieving reuse via frameworks and patterns is described in [Fach, 2001].

Design patterns were put forward by Alexander within the context of architecture, as described in [Alexander et al., 1977]. A first-hand account of the origins of pattern theory appears in [Alexander, 1999]. The primary work on software design patterns is [Gamma, Helm, Johnson, and Vlissides, 1995]; a newer book is [Vlissides, 1998]. Analysis patterns are described in [Fowler, 1997], and requirements patterns in [Hagge and Lappe, 2005].

Experiments to assess the impact of design pattern documentation on maintenance are described in [Prechelt, Unger-Lamprecht, Philippsen, and Tichy, 2002]. Antipatterns are described in [Brown et al., 1998]. Patterns for designing embedded systems are discussed in [Pont and Banner, 2004]. Vokac [2004] describes the impact of patterns on fault rates in a 500-KLOC product.

The primary source of information on software architectures is [Shaw and Garlan, 1996]. Newer works on software architectures include [Bosch, 2000] and [Bass, Clements, and Kazman, 2003]. Software product lines are described in [Jazayeri, Ran, and van der Linden, 2000; Knauber,

Muthig, Schmid, and Widen, 2000; Donohoe, 2000; and Clements and Northrop, 2002]. The state of the practice of software product lines is discussed in [Birk et al. 2003]. Cost–benefit analysis of software product lines is presented in [Bockle et al., 2004]. The July/August 2002 issue of *IEEE Software* contains a variety of articles on product lines.

Papers on component-based software engineering can be found in the September/October 1998 issue of *IEEE Software*, including [Weyuker, 1998], which discusses the testing of component-based software. Brereton and Budgen [2000] discuss the key issues in component-based software products. Articles on experiences with component-based software engineering include [Sparling, 2000] and [Baster, Konana, and Scott, 2001]. Strengths and weaknesses of component-based software engineering are discussed in [Vitharana, 2003]. [Heineman and Councill, 2001] is a highly recommended compendium of articles on component-based software engineering.

Strategies for achieving portability can be found in [Mooney, 1990]. Portability of UNIX is discussed in [Johnson and Ritchie, 1978].

Key Terms

abstract class 229
Abstract Factory design pattern 234
abstract method 229
accidental reuse 216
Adapter design pattern 229
aggregate 233
application framework 224
architecture pattern 226
Bridge design pattern 230
behavioral design patterns 236
business logic tier 227
COBOL program logic structure 220
collection 233
component-based software engineering 227
container 233
creational design patterns 235
cursor 233
data access logic tier 227
deliberate reuse 216
driver 230
element access 233
element traversal 233
framework 224
functional module 220
hot spot 224
iterator 233
Iterator design pattern 233
model-view-controller (MVC) architecture pattern 226
not invented here (NIH) syndrome 218
opportunistic reuse 216
portable 216
presentation logic tier 227
reuse 216
software architecture 226
software product line 226
source computer 239
structural design patterns 235
systematic reuse 216
target computer 239
toolkit 223
widget 233
wrapper 225

Problems

8.1 Explain in detail the differences between reusability and portability.

8.2 A code artifact is reused, unchanged, in a new product. In what ways does this reuse reduce the overall cost of the product? In what ways is the cost unchanged?

8.3 Suppose that a code artifact is reused with one change, an addition operation is changed to a subtraction. What impact does this minor change have on the savings of Problem 8.2?

8.4 What is the influence of cohesion on reusability?

8.5 What is the influence of coupling on reusability?

8.6 You have just joined a large organization that manufactures a variety of pollution control products. The organization has hundreds of software products consisting of some 8000 different Fortran 2003 classes. You have been hired to come up with a plan for reusing as many of these classes as possible in future products. What is your proposal?

8.7 Consider an automated library circulation system. Every book has a bar code, and every borrower has a card bearing a bar code. When a borrower wishes to check out a book, the librarian scans the bar codes on the book and the borrower's card, and enters C at the computer terminal. Similarly, when a book is returned, it is again scanned and the librarian enters R. Librarians can add books (+) to the library collection or remove them (–). Borrowers can go to a terminal and determine all the books in the library by a particular author (the borrower enters A= followed by the author's name), all the books with a specific title (T= followed by the title), or all the books in a particular subject area (S= followed by the subject area). Finally, if a borrower wants a book currently checked out, the librarian can place a hold on the book so that, when it is returned, it will be held for the borrower who requested it (H= followed by the number of the book). Explain how you would ensure a high percentage of reusable code artifacts.

8.8 You are required to build a product for determining whether a bank statement is correct. The data needed include the balance at the beginning of the month; the number, date, and amount of each check; the date and amount of each deposit; and the balance at the end of the month. Explain how you would ensure that as many code artifacts as possible of the product can be reused in future products.

8.9 Consider an automated teller machine (ATM). The user puts a card into a slot and enters a four-digit personal identification number (PIN). If the PIN is incorrect, the card is ejected. Otherwise, the user may perform the following operations on up to four different bank accounts:

(i) Deposit any amount. A receipt is printed showing the date, amount deposited, and account number.

(ii) Withdraw up to $200 in units of $20 (the account may not be overdrawn). In addition to the money, the user is given a receipt showing the date, amount withdrawn, account number, and account balance after the withdrawal.

(iii) Determine the account balance. This is displayed on the screen.

(iv) Transfer funds between two accounts. Again, the account from which the funds are transferred must not be overdrawn. The user is given a receipt showing the date, amount transferred, and the two account numbers.

(v) Quit. The card is ejected.

Explain how you would ensure that as many code artifacts as possible of the product can be reused in future products.

8.10 How early in the software life cycle could the developers have caught the fault in the Ariane 5 software (Section 8.3.2)?

8.11 Section 8.5.2 states that "the Raytheon COBOL program logic structure of the 1970s is a classical precursor of today's object-oriented application framework." What are the implications of this for technology transfer?

8.12 Explain the role played by abstract classes in the design pattern of Figure 8.10.

8.13 Explain how you would ensure that the automated library circulation system (Problem 8.7) is as portable as possible.

8.14 Explain how you would ensure that the product that checks whether a bank statement is correct (Problem 8.8) is as portable as possible.

8.15 Explain how you would ensure that the software for the automated teller machine (ATM) of Problem 8.9 is as portable as possible.

8.16 Your organization is developing a real-time control system for a new type of laser that will be used in cancer therapy. You are in charge of writing two assembler modules. How will you instruct your team to ensure that the resulting code will be as portable as possible?

8.17 You are responsible for porting a 750,000-line OO-COBOL product to your company's new computer. You copy the source code to the new machine but discover when you try to compile it that every one of the over 15,000 input–output statements has been written in a nonstandard OO-COBOL syntax that the new compiler rejects. What do you do now?

8.18 In what ways does the object-oriented paradigm promote portability and reusability?

8.19 (Term Project) Suppose that the Osric's Office Appliances and Decor product of Appendix A has been developed using the object-oriented paradigm. What parts of the product could be reused in future products?

8.20 (Readings in Software Engineering) Your instructor will distribute copies of [Tomer et al., 2004]. What data would you need to accumulate in order to use the model?

References

[Alexander, 1999] C. Alexander, "The Origins of Pattern Theory," *IEEE Software* **16** (September/October 1999), pp. 71–82.

[Alexander et al., 1977] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*, Oxford University Press, New York, 1977.

[ANSI/IEEE 754, 1985] *Standard for Binary Floating Point Arithmetic*, ANSI/IEEE 754, American National Standards Institute, Institute of Electrical and Electronic Engineers, New York, 1985.

[ANSI/MIL-STD-1815A, 1983] *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, American National Standards Institute, United States Department of Defense, Washington, DC, 1983.

[Barnes and Bollinger, 1991] B. H. Barnes and T. B. Bollinger, "Making Reuse Cost-Effective," *IEEE Software* **8** (January 1991), pp. 13–24.

[Bass, Clements, and Kazman, 2003] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, Reading, MA, 2003.

[Baster, Konana, and Scott, 2001] G. Baster, P. Konana, and J. E. Scott, "Business Components: A Case Study of Bankers Trust Australia Limited," *Communications of the ACM* **44** (May 2001), pp. 92–98.

[Birk et al. 2003] A. Birk, G. Heller, I. John, K. Schmid, T. von der Massen, and K. Muller, "Product Line Engineering, the State of the Practice," *IEEE Software* **20** (November/December 2003), pp. 52–60.

[Bockle et al., 2004] G. Bockle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid, "Calculating ROI for Software Product Lines," *IEEE Software* **21** (May/June 2004), pp. 23–31.

[Bosch, 2000] J. Bosch, *Design and Use of Software Architectures*, Addison-Wesley, Reading, MA, 2000.

[Brereton and Budgen, 2000] P. Brereton and D. Budgen, "Component-Based Systems: A Classification of Issues," *IEEE Computer* **33** (November 2000), pp. 54–62.

[Brown et al., 1998] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick, III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,* John Wiley and Sons, New York, 1998.

[Caldiera and Basili, 1991] G. Caldiera and V. R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer* **24** (February 1991), pp. 61–70.

[Clements and Northrop, 2002] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, Reading, MA, 2002.

[Computer Gripes, 2004] "Gripes about Web Sites That Don't Work Well with Firefox," at: www.computergripes.com/firefoxsites.html, 2004.

[Donohoe, 2000] P. Donohoe (Editor), *Software Product Lines: Experience and Research Directions*, Kluwer Academic Publishers, Boston, 2000.

[D'Souza and Wills, 1999] D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, Reading, MA, 1999.

[Fach, 2001] P. W. Fach, "Design Reuse through Frameworks and Patterns," *IEEE Software* **18** (September/October 2001), pp. 71–76.

[Fayad and Johnson, 1999] M. Fayad and R. Johnson, *Domain-Specific Application Frameworks: Frameworks Experience by Industry,* John Wiley and Sons, New York, 1999.

[Fayad and Schmidt, 1999] M. Fayad and D. C. Schmidt, *Building Application Frameworks: Object-Oriented Foundations of Framework Design,* John Wiley and Sons, New York, 1999.

[Fayad, Schmidt, and Johnson, 1999] M. Fayad, D. C. Schmidt, and R. Johnson, *Implementing Application Frameworks: Object-Oriented Frameworks at Work,* John Wiley and Sons, New York, 1999.

[Fichman and Kemerer, 1997] R. G. Fichman and C. F. Kemerer, "Object Technology and Reuse: Lessons from Early Adopters," *IEEE Computer* **30** (July 1997), pp. 47–57.

[Fingar, 2000] P. Fingar, "Component-Based Frameworks for e-Commerce," *Communications of the ACM* **43** (October 2000), pp. 61–66.

[Flanagan, 2005] D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*, 5th ed., O'Reilly and Associates, Sebastopol, CA, 2005.

[Fowler, 1997] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Reading, MA, 1997.

[Gamma, Helm, Johnson, and Vlissides, 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[Gifford and Spector, 1987] D. Gifford and A. Spector, "Case Study: IBM's System/360–370 Architecture," *Communications of the ACM* **30** (April 1987), pp. 292–307.

[Green, 2000] P. Green, "FW: Here's an Update to the Simulated Kangaroo Story," *The Risks Digest* 20 (January 23, 2000), catless.ncl.ac.uk/Risks/20.76.html.

[Griss, 1993] M. L. Griss, "Software Reuse: From Library to Factory," *IBM Systems Journal* **32** (No. 4, 1993), pp. 548–66.

[Hagge and Lappe, 2005] L. Hagge and K. Lappe, "Sharing Requirements Engineering Experience Using Patterns," *IEEE Software* **22** (January/February 2005), pp. 24–31.

[Heineman and Councill, 2001] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Reading, MA, 2001.

[HTML, 2006] "W3C HTML Homepage," at www.w3.org/MarkUp, 2006.

[Isakowitz and Kauffman, 1996] T. Isakowitz and R. J. Kauffman, "Supporting Search for Reusable Software Objects," *IEEE Transactions on Software Engineering* **22** (June 1996), pp. 407–23.

[ISO/IEC 1539–1, 2004] *Information Technology—Programming Languages—Fortran—Part 1: Base Language*, ISO/IEC 1539–1, International Organization for Standardization, International Electrotechnical Commission, Geneva, 2004.

[ISO/IEC 1989, 2002] *Information Technology—Programming Language COBOL*, ISO 1989:2002, International Organization for Standardization, International Electrotechnical Commission, Geneva, 2002.

[ISO/IEC 8652, 1995] *Programming Language Ada: Language and Standard Libraries*, ISO/IEC 8652, International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[ISO/IEC 14882, 1998] *Programming Language C++*, ISO/IEC 14882, International Organization for Standardization, International Electrotechnical Commission, Geneva, 1998.

[Jazayeri, Ran, and van der Linden, 2000] M. Jazayeri, A. Ran, and F. van der Linden, *Software Architecture for Product Families: Principles and Practice,* Addison-Wesley, Reading, MA, 2000.

[Jézéquel and Meyer, 1997] J.-M. Jézéquel and B. Meyer, "Put It in the Contract: The Lessons of Ariane," *IEEE Computer* **30** (January 1997), pp. 129–30.

[Johnson and Ritchie, 1978] S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the UNIX System," *Bell System Technical Journal* **57** (No. 6, Part 2, 1978), pp. 2021–48.

[Jones, 1984] T. C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering* **SE-10** (September 1984), pp. 488–94.

[Knauber, Muthig, Schmid, and Widen, 2000] P. Knauber, D. Muthig, K. Schmid, and T. Widen, "Applying Product Line Concepts in Small and Medium-Sized Companies," *IEEE Software* **17** (September/October 2000), pp. 88–95.

[Kobryn, 2000] C. Kobryn, "Modeling Components and Frameworks with UML," *Communications of the ACM* **43** (October 2000), pp. 31–38.

[Lai, Weiss, and Parnas, 1999] C. T. R. Lai, D. M. Weiss, and D. L. Parnas, *Software Product-Line Engineering: A Family-Based Software Development Process,* Addison-Wesley, Reading, MA, 1999.

[Lanergan and Grasso, 1984] R. G. Lanergan and C. A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering* **SE-10** (September 1984), pp. 498–501.

[LAPACK++, 2000] "LAPACK++: Linear Algebra Package in C++," at math.nist.gov/lapack++, 2000.

[Lewis et al., 1995] T. Lewis, L. Rosenstein, W. Pree, A. Weinand, E. Gamma, P. Calder, G. Andert, J. Vlissides, and K. Schmucker, *Object-Oriented Application Frameworks*, Manning, Greenwich, CT, 1995.

[Lim, 1994] W. C. Lim, "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software* **11** (September 1994), pp. 23–30.

[Lim, 1998] W. C. Lim, *Managing Software Reuse*, Prentice Hall, Upper Saddle River, NJ, 1998.

[Liskov, Snyder, Atkinson, and Schaffert, 1977] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM* **20** (August 1977), pp. 564–76.

[Mackenzie, 1980] C. E. Mackenzie, *Coded Character Sets: History and Development*, Addison-Wesley, Reading, MA, 1980.

[Matsumoto, 1984] Y. Matsumoto, "Management of Industrial Software Production," *IEEE Computer* **17** (February 1984), pp. 59–72.

[Matsumoto, 1987] Y. Matsumoto, "A Software Factory: An Overall Approach to Software Production," in: *Tutorial: Software Reusability*, P. Freeman (Editor), Computer Society Press, Washington, DC, 1987, pp. 155–78.

[Meyer, 1987] B. Meyer, "Reusability: The Case for Object-Oriented Design," *IEEE Software* **4** (March 1987), pp. 50–64.

[Meyer, 1996a] B. Meyer, "The Reusability Challenge," *IEEE Computer* **29** (February 1996), pp. 76–78.

[Mooney, 1990] J. D. Mooney, "Strategies for Supporting Application Portability," *IEEE Computer* **23** (November 1990), pp. 59–70.

[Morisio, Ezran, and Tully, 2002] M. Morisio, M. Ezran, and C. Tully, "Success and Failure Factors in Software Reuse," *IEEE Transactions on Software Engineering* **28** (April 2002), pp. 340–57.

[Morisio, Tully, and Ezran, 2000] M. Morisio, C. Tully, and M. Ezran, "Diversity in Reuse Processes," *IEEE Software* **17** (July/August 2000), pp. 56–63.

[Musser and Saini, 1996] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley, Reading, MA, 1996.

[NAG, 2003] "NAG The Numerical Algorithms Group Ltd," at www.nag.co.uk, 2003.

[NIST 151, 1988] "POSIX: Portable Operating System Interface for Computer Environments," Federal Information Processing Standard 151, National Institute of Standards and Technology, Washington, DC, 1988.

[Norušis, 2005] M. J. Norušis, *SPSS 13.0 Guide to Data Analysis,* Prentice Hall, Upper Saddle Valley River, NJ, 2005.

[Norwig, 1996] P. Norwig, "Design Patterns in Dynamic Programming," norvig.com/design-patterns/ppframe.htm/, 1996.

[Pont and Banner, 2004] M. J. Pont and M. P. Banner, "Designing Embedded Systems Using Patterns: A Case Study," *Journal of Systems and Software* 71 (May 2004), pp. 201–13.

[Poulin, 1997] J. S. Poulin, *Measuring Software Reuse: Principles, Practice, and Economic Models*, Addison-Wesley, Reading, MA, 1997.

[Prechelt, Unger-Lamprecht, Philippsen, and Tichy, 2002] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, "Two Controlled Experiments in Assessing the Usefulness of Design Pattern Documentation in Program Maintenance," *IEEE Transactions on Software Engineering* **28** (June 2002), pp. 595–606.

[Prieto-Díaz, 1991] R. Prieto-Díaz, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM* **34** (May 1991), pp. 88–97.

[Ravichandran and Rothenberger, 2003] T. Ravichandran and M. A. Rothenberger, "Software Reuse Strategies and Component Markets," *Communications of the ACM* **46** (August 2003), pp. 109–14

[Schach, 1992] S. R. Schach, *Software Reuse: Past, Present, and Future*, videotape, 150 min, US-VHS format, IEEE Computer Society Press, Los Alamitos, CA, November 1992.

[Schach, 1994] S. R. Schach, "The Economic Impact of Software Reuse on Maintenance," *Journal of Software Maintenance—Research and Practice* **6** (July/August 1994), pp. 185–96.

[Schach, 1997] S. R. Schach, *Software Engineering with Java*, Richard D. Irwin, Chicago, 1997.

[Schricker, 2000] D. Schricker, "Cobol for the Next Millennium," *IEEE Software* **17** (March/April 2000), pp. 48–52.

[Selby, 1989] R. W. Selby, "Quantitative Studies of Software Reuse," in: *Software Reusability,* Vol. 2, *Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (Editors), ACM Press, New York, 1989, pp. 213–33.

[Shaw and Garlan, 1996] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, NJ, 1996.

[Sparling, 2000] M. Sparling, "Lessons Learned through Six Years of Component-Based Development," *Communications of the ACM* **43** (October 2000), pp. 47–53.

[Tanenbaum, 2002] A. S. Tanenbaum, *Computer Networks,* 4th ed., Prentice Hall, Upper Saddle River, NJ, 2002.

[Toft, Coleman, and Ohta, 2000] P. Toft, D. Coleman, and J. Ohta, "A Cooperative Model for Cross-Divisional Product Development for a Software Product Line," in: *Software Product Lines: Experience and Research Directions*, P. Donohoe (Editor), Kluwer Academic Publishers, Boston, 2000, pp. 111–32.

[Tomer et al., 2004] A. Tomer, L. Goldin, T. Kuflik, E. Kimchi, and S. R. Schach, "Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study," *IEEE Transactions on Software Engineering* **30** (September 2004), pp. 601–12.

[Tracz, 1994] W. Tracz, "Software Reuse Myths Revisited," *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 271–72.

[Vitharana, 2003] P. Vitharana, "Risks and Challenges of Component-Based Software Development," *Communications of the ACM* **46** (August 2003), pp. 67–72.

[Vlissides, 1998] J. Vlissides, *Pattern Hatching: Design Patterns Applied,* Addison-Wesley, Reading, MA, 1998.

[Vokac, 2004] M. Vokac, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code," *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 904–17.

[Weyuker, 1998] E. J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software* **15** (September/October 1998), pp. 54–59.

[XML, 2003] "Extensible Markup Language (XML)," at www.w3.org/XML/, 2003.