CHAPTER 1

Application Frameworks

Computing power and network bandwidth have increased dramatically over the past decade. However, the design and implementation of complex software remain expensive and error-prone. Much of the cost and effort stem from the continuous rediscovery and reinvention of core concepts and components across the software industry. In particular, the growing heterogeneity of hardware architectures and diversity of operating system and communication platforms make it hard to build correct, portable, efficient, and inexpensive applications from scratch.

Frameworks are an object-oriented reuse technique. They share a lot of characteristics with reuse techniques in general and object-oriented reuse techniques in particular. Although they have been used successfully for some time and are an important part of the culture of long-time object-oriented developers, most framework development projects are failures, and most object-oriented methodologies do not describe how to use frameworks. Moreover, there is a lot of confusion about whether frameworks are just large-scale patterns, or whether they are just another kind of component.

Even the definition of *framework* varies. A frequently used definition is "a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact." Another common definition is "a framework is the skeleton of an application that can be customized by an application developer." These are not conflicting definitions; the first describes the structure of a framework while the second describes its purpose. Nevertheless, they point out the difficulty of defining frameworks clearly.

Frameworks are important and are becoming more important. Systems like Object Linking and Embedding (OLE), OpenDoc, and Distributed System Object Model

(DSOM) are frameworks. The rise of Java is spreading new frameworks like Abstract Window Toolkit (AWT), which is part of the Java Foundation Classes (JFC) and Beans. Most commercially available frameworks seem to be for technical domains such as user interfaces or distribution, and most application-specific frameworks are proprietary. But the steady rise of frameworks means that every software developer should know what they are and how to deal with them.

This chapter compares and contrasts frameworks with other reuse techniques and describes how to use them, how to evaluate them, and how to develop them. It describes the trade-offs involved in using frameworks, including the costs and pitfalls, and when frameworks are appropriate.

1.1 What Is an Application Framework?

Object-oriented (OO) application frameworks are a promising technology for reifying proven software designs and implementations in order to reduce the cost and improve the quality of software. A framework is a reusable, semi-complete application that can be specialized to produce custom applications [Johnson-Foote 1988]. In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics). Frameworks like MacApp, ET++, Interviews, Advanced Computing Environment (ACE), Microsoft Foundation Classes (MFCs) and Microsoft's Distributed Common Object Model (DCOM), JavaSoft's Remote Method Invocation (RMI), and implementations of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) play an increasingly important role in contemporary software development.

A framework is a reusable design of a system that describes how the system is decomposed into a set of interacting objects. Sometimes the system is an entire application; sometimes it is just a subsystem. The framework describes both the component objects and how these objects interact. It describes the interface of each object and the flow of control between them. It describes how the system's responsibilities are mapped onto its objects [Johnson-Foote 1988; Wirfs-Brock 1990].

The most important part of a framework is the way that a system is divided into its components [Deutsch 1989]. Frameworks also reuse implementation, but that is less important than reuse of the internal interfaces of a system and the way that its functions are divided among its components. This high-level design is the main intellectual content of software, and frameworks are a way to reuse it.

Typically, a framework is implemented with an object-oriented language like C++, Smalltalk, or Eiffel. Each object in the framework is described by an *abstract class*. An abstract class is a class with no instances, so it is used only as a superclass [Wirfs-Brock 1990]. An abstract class usually has at least one unimplemented operation deferred to its subclasses. Since an abstract class has no instances, it is used as a template for creating subclasses rather than as a template for creating objects. Frameworks use them as designs of their components because they both define the interface of the components and provide a skeleton that can be extended to implement the components.

Some of the more recent object-oriented systems, such as Java, the Common Object Model (COM), and CORBA, separate interfaces from classes. In these systems, a framework can be described in terms of interfaces. However, these systems can specify only

the static aspects of an interface, but a framework is also the collaborative model or pattern of object interaction. Consequently, it is common for Java frameworks to have both an interface and an abstract class defined for a component.

In addition to providing an interface, an abstract class provides part of the implementation of its subclasses. For example, a *template method* defines the skeleton of an algorithm in an abstract class, deferring some of the steps to subclasses [Gamma 1995]. Each step is defined as a separate method that can be redefined by a subclass, so a subclass can redefine individual steps of the algorithm without changing its structure. The abstract class can either leave the individual steps unimplemented (in other words, they are abstract methods) or provide a default implementation (in other words, they are hook methods) [Pree 1995]. A concrete class must implement all the abstract methods of its abstract superclass and may implement any of the hook methods. It will then be able to use all the methods it inherits from its abstract superclass.

Frameworks take advantage of all three of the distinguishing characteristics of object-oriented programming languages: data abstraction, polymorphism, and inheritance. Like an abstract data type, an abstract class represents an interface behind which implementations can change. *Polymorphism* is the ability for a single variable or procedure parameter to take on values of several types. Object-oriented polymorphism lets a developer mix and match components, lets an object change its collaborators at runtime, and makes it possible to build generic objects that can work with a wide range of components. Inheritance makes it easy to make a new component.

A framework describes the architecture of an object-oriented system; the kinds of objects in it, and how they interact. It describes how a particular kind of program, such as a user interface or network communication software, is decomposed into objects. It is represented by a set of classes (usually abstract), one for each kind of object, but the interaction patterns between objects are just as much a part of the framework as the classes.

One of the characteristics of frameworks is *inversion of control*. Traditionally, a developer reused components from a library by writing a main program that called the components whenever necessary. The developer decided when to call the components and was responsible for the overall structure and flow of control of the program. In a framework, the main program is reused, and the developer decides what is plugged into it and might even make some new components that are plugged in. The developer's code gets called by the framework code. The framework determines the overall structure and flow of control of the program.

The first widely used framework, developed in the late 1970s, was the Smalltalk-80 user interface framework called Model/View/Controller (MVC) [Goldberg 1984; Krasner 1988; LaLonde 1991]. MVC showed that object-oriented programming was well suited for implementing graphical user interfaces (GUIs). It divides a user interface into three kinds of components: models, views, and controllers. These objects work in trios consisting of a view and a controller interacting with a model. A model is an application object and is supposed to be independent of the user interface. A view manages a region of the display and keeps it consistent with the state of the model. A controller converts user events (mouse movements and key presses) into operations on its model and view. For example, controllers implement scrolling and menus. Views can be nested to form complex user interfaces. Nested views are called *subviews*.

Figure 1.1 shows the user interface of one of the standard tools in the Smalltalk-80 environment, the file tool. The file tool has three subviews. The top subview holds a string that is a pattern that matches a set of files, the middle subview displays the list

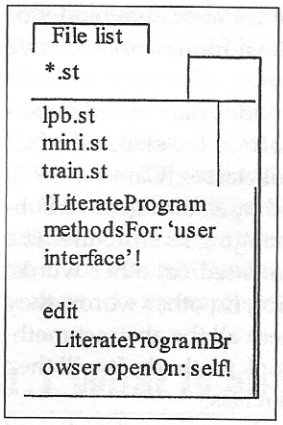

Figure 1.1 The Smalltalk-80 file tool.

of files that match the pattern, and the bottom subview displays the selected file. All three subviews have the same model—a FileBrowser. The top and bottom subviews are instances of TextView, while the middle subview is an instance of SelectionInListView. As shown by Figure 1.2, all three views are subviews of a StandardSystemView. Each of the four views has its own controller.

Class View is an abstract class with base operations for creating and accessing the subview hierarchy, transforming from view coordinates to screen coordinates, and keeping track of its region on the display. It has abstract and template operations for displaying, since different kinds of views require different display algorithms. TextView, SelectionInListView, and StandardSystemView are concrete subclasses of View, each of which has a unique display algorithm.

As a user moves the mouse from one subview to another, controllers are activated and deactivated so that the active controller is always the controller of the view managing the region of the display that contains the cursor. Class Controller implements

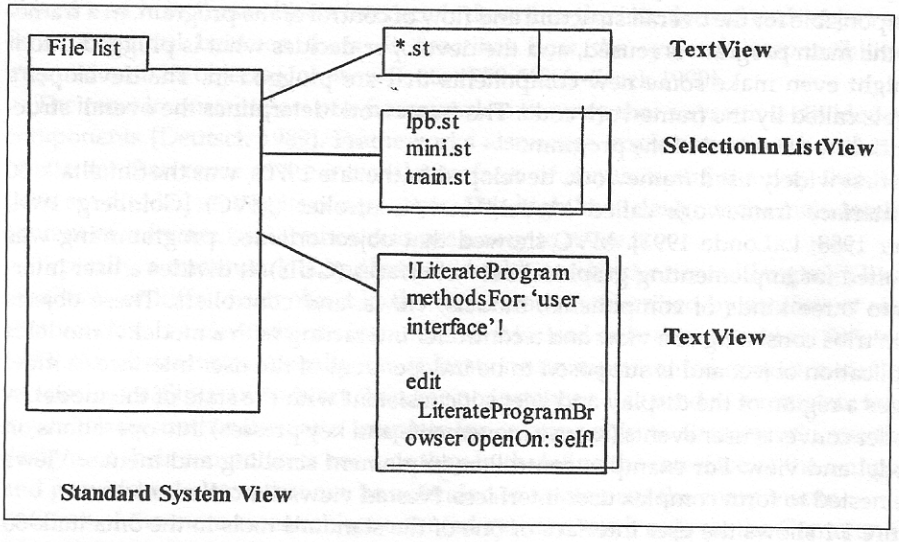

Figure 1.2 Subview hierarchy in file tool.

the protocol that ensures this, so a subclass of Controller automatically inherits the ability to cooperate with other controllers.

Class Object provides a dependency mechanism that views can use to detect when the model's state changes. Thus, any object can be a model. There is also a Model class that provides a faster version of the dependency mechanism at the cost of an extra variable [ParcPlace 1988].

The file tool is a typical Model/View/Controller application that does not need new subclasses of View or Controller. Its user interface consists entirely of objects from classes that are a standard part of the Smalltalk-80 class library. The Smalltalk-80 class library contains several dozen concrete subclasses of View and Controller. However, when these are not sufficient, new subclasses can be built to extend the user interface.

Successful frameworks evolve and spawn other frameworks. One of the first user interface frameworks influenced by Model/View/Controller was MacApp, which was designed specifically for implementing Macintosh applications [Schmucker 1986]. It was followed by user interface frameworks from universities, such as the Andrew Toolkit from Carnegie Mellon University [Palay 1988], InterViews from Stanford [Linton 1989], and ET++ from the University of Zurich [Weinand 1988, 1989]. There are now a large number of commercial user interface frameworks, such as zAPP, OpenStep, and MFCs. Some, like OpenStep's, are a small part of a much more comprehensive system. Some, like zAPP, are designed for developing portable software and shield the developer from the peculiarities of an operating system. Each of these frameworks borrows ideas from earlier systems. Although the differences between the frameworks are due partly to different requirements, sometimes newer systems incorporate better design techniques, and so the state of the art gradually improves.

Frameworks are not limited to user interfaces, but can be applied to any area of software design. They have been applied to very large scale integration (VLSI) routing algorithms [Gossain 1990], to hypermedia systems [Meyrowitz 1986], to structured drawing editors [Beck 1994; Vlissides 1989], to operating systems [Russo 1990], to psychophysiological experiments [Foote 1988], to network protocol software [Hueni 1995], and to manufacturing control [Schmid 1995], to mention a few.

Frameworks do not even require an object-oriented programming language. For example, the Genesis database system compiler is a framework for database management systems (DBMSs) [Batory 1989] as well as a tool for specifying how DBMSs are built from the framework. Genesis does not use an object-oriented language but rather a macroprocessor and conditional compilation to implement an object-oriented design in C.

The important classes in a framework, such as Model, View, and Controller of Model/View/Controller, are usually abstract. Like MVC, a framework usually comes with a *component library* that contains concrete subclasses of the classes in the framework. Although a good component library is a crucial companion to a framework, the essence of a framework is not the component library, but the model of interaction and control flow among its objects.

A framework reuses code because it makes it easy to build an application from a library of existing components. These components can be easily used with each other because they all use the interfaces of the framework. A framework also reuses code because a new component can inherit most of its implementation from an abstract superclass. But reuse is best when you don't have to understand the component you

are reusing, and inheritance requires a deeper understanding of a class that is using it as a component, so it is better to reuse existing components than to make a new one.

Of course, the main reason a framework enables code reuse is that it is a reusable design. It provides reusable abstract algorithms and a high-level design that decomposes a large system into smaller components and describes the internal interfaces between components. These standard interfaces make it possible to mix and match components and to build a wide variety of systems from a small number of existing components. New components that meet these interfaces will fit into the framework, so component designers also reuse the design of a framework.

Finally, a framework reuses analysis. It describes the kinds of objects that are important and provides a vocabulary for talking about a problem. An expert in a particular framework sees the world in terms of the framework and will naturally divide it into the same components. Two expert users of the same framework will find it easier to understand each other's designs, since they will come up with similar components and will describe the systems they want to build in similar ways.

Analysis, design, and code reuse are all important, though in the long run it is probably the analysis and design reuse that provide the biggest payoff [Biggerstaff 1996].

1.2 Benefits

The primary benefits of OO application frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to developers, as described in the following:

Modularity. Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.

Reusability. The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid recreating and revalidating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability, and interoperability of software.

Extensibility. A framework enhances extensibility by providing explicit hook methods [Pree 1995] that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.

Inversion of control. The runtime architecture of a framework is characterized by an *inversion of control*. This architecture enables canonical application processing steps

to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on preregistered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end users or packets arriving on communication ports).

1.3 An Overview of Widely Used Frameworks

Developers in certain domains have successfully applied OO application frameworks for many years. Early object-oriented frameworks (such as MacApp and InterViews) originated in the domain of graphical user interfaces. The Microsoft Foundation Classes constitute a contemporary GUI framework that has become the de facto industry standard for creating graphical applications on PC platforms. Although the MFCs have limitations (such as lack of portability to non-PC platforms), their widespread adoption demonstrates the productivity benefits of reusing common frameworks to develop graphical business applications.

Application developers in more complex domains (such as telecommunications, distributed medical imaging, and real-time avionics) have traditionally lacked standard off-the-shelf frameworks. As a result, developers in these domains largely build, validate, and maintain software systems from scratch. In an era of deregulation and stiff global competition, however, it has become prohibitively costly and time consuming to develop applications entirely in-house from the ground up.

Fortunately, the next generation of OO application frameworks is targeting complex business and application domains. At the heart of this effort are *Object Request Broker* (ORB) frameworks, which facilitate communication between local and remote objects. ORB frameworks eliminate many tedious, error-prone, and nonportable aspects of creating and managing distributed applications and reusable service components. This enables programmers to develop and deploy complex applications rapidly and robustly, rather than wrestling endlessly with low-level infrastructure concerns.

1.4 Classifying Application Frameworks

Although the benefits and design principles underlying frameworks are largely independent of the domain to which they are applied, we've found it useful to classify frameworks by their scope, as follows:

System infrastructure frameworks. These frameworks simplify the development of portable and efficient system infrastructure such as operating system [Campbell-Islam 1993] and communication frameworks [Schmidt 1997], and frameworks for user interfaces and language processing tools. System infrastructure frameworks are primarily used internally within a software organization and are not sold to customers directly.

Middleware integration frameworks. These frameworks are commonly used to integrate distributed applications and components. Middleware integration frameworks are designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment. There is a thriving market for middleware integration frameworks, which are rapidly becoming commodities. Common examples include ORB frameworks, message-oriented middleware, and transactional databases.

Enterprise application frameworks. These frameworks address broad application domains (such as telecommunications, avionics, manufacturing, and financial engineering [Birrer 1993]) and are the cornerstone of enterprise business activities [Fayad-Hamu 1999; Hamu-Fayad 1998]. Relative to system infrastructure and middleware integration frameworks, enterprise frameworks are expensive to develop and/or purchase. However, enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly. In contrast, system infrastructure and middleware integration frameworks focus largely on internal software development concerns. Although these frameworks are essential to rapidly create high-quality software, they typically do not generate substantial revenue for large enterprises. As a result, it is often more cost-effective to buy system infrastructure and middleware integration frameworks rather than build them in-house [Fayad-Hamu 1999; Hamu-Fayad 1998].

Regardless of their scope, frameworks can also be classified by the techniques used to extend them, which range along a continuum from *whitebox frameworks* to *graybox frameworks* to *blackbox frameworks*. Whitebox frameworks rely heavily on OO language features like inheritance and dynamic binding in order to achieve extensibility. Existing functionality is reused and extended by (1) inheriting from framework base classes and (2) overriding predefined hook methods using patterns like the Template Method [Gamma 1995]. Blackbox frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by (1) defining components that conform to a particular interface and (2) integrating these components into the framework using patterns like Strategy [Gamma 1995] and Functor. Graybox frameworks are designed to avoid the disadvantages presented by whitebox and blackbox frameworks. In other words, a good graybox framework has enough flexibility and extendibility, and also has the ability to hide unnecessary information from the application developers [Yassin-Fayad 1999].

Whitebox frameworks require application developers to have intimate knowledge of the frameworks' internal structure. Although whitebox frameworks are widely used, they tend to produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies. In contrast, blackbox frameworks are structured using object composition and delegation rather than inheritance. As a result, blackbox frameworks are generally easier to use and extend than whitebox frameworks. However, blackbox frameworks are more difficult to develop since they require framework developers to define interfaces and hooks that anticipate a wider range of potential use-cases [Hueni 1995].

1.5 The Strengths and Weaknesses of Application Frameworks

When used in conjunction with patterns, class libraries, and components, OO application frameworks can significantly increase software quality and reduce development effort. However, a number of challenges must be addressed in order to employ frameworks effectively. Companies attempting to build or use large-scale reusable frameworks often fail unless they recognize and resolve challenges such as *development effort, learning curve, integratability, maintainability, validation and defect removal, efficiency,* and *lack of standards,* which are outlined as follows:

Development effort. While developing complex software is hard enough, developing high-quality, extensible, and reusable frameworks for complex application domains is even harder. The skills required to produce frameworks successfully often remain locked in the heads of expert developers. One of the goals of this theme issue is to demystify the software process and design principles associated with developing and using frameworks.

Learning curve. Learning to use an OO application framework effectively requires considerable investment of effort. For instance, it often takes 6 to 12 months to become highly productive with a GUI framework like MFCs or MacApp, depending on the experience of the developers. Typically, hands-on mentoring and training courses are required to teach application developers how to use the framework effectively. Unless the effort required to learn the framework can be amortized over many projects, this investment may not be cost-effective. Moreover, the suitability of a framework for a particular application may not be apparent until the learning curve has flattened.

Integratability. Application development will be increasingly based on the integration of multiple frameworks (GUIs, communication systems, databases, and so on), together with class libraries, legacy systems, and existing components. However, many earlier-generation frameworks were designed for internal extension rather than for integration with other frameworks developed externally. Integration problems arise at several levels of abstraction, ranging from documentation issues [Fayad-Hamu 1999; Hamu-Fayad 1998], to the concurrency/distribution architecture, to the event dispatching model. For instance, while inversion of control is an essential feature of a framework, integrating frameworks whose event loops are not designed to interoperate with other frameworks is hard.

Maintainability. Application requirements change frequently. Therefore, the requirements of frameworks often change, as well. As frameworks invariably evolve, the applications that use them must evolve with them.

Framework maintenance activities include modification and adaptation of the framework. Both modification and adaptation may occur on the *functional level* (in other words, certain framework functionality does not fully meet developers' requirements), as well as on the *nonfunctional level* (which includes more qualitative aspects such as portability or reusability).

Framework maintenance may take different forms, such as adding functionality, removing functionality, and generalization. A deep understanding of the framework components and their interrelationships is essential to perform this task successfully. In some cases, the application developers and/or the end users must rely entirely on framework developers to maintain the framework.

Validation and defect removal. Although a well-designed, modular framework can localize the impact of software defects, validating and debugging applications built using frameworks can be tricky for the following reasons:

Generic components are harder to validate in the abstract. A well-designed framework component typically abstracts away from application-specific details, which are provided via subclassing, object composition, or template parameterization. While this improves the flexibility and extensibility of the framework, it greatly complicates module testing since the components cannot be validated in isolation from their specific instantiations.

Moreover, it is usually hard to distinguish bugs in the framework from bugs in the application code. As with any software development, bugs are introduced into a framework from many possible sources, such as failure to understand the requirements, overly coupled design, or an incorrect implementation. When customizing the components in a framework to a particular application, the number of possible error sources will increase.

Inversion of control and lack of explicit control flow. Applications written with frameworks can be hard to debug since the framework's *inverted* flow of control oscillates between the application-independent framework infrastructure and the application-specific method callbacks. This increases the difficulty of single-stepping through the runtime behavior of a framework within a debugger since the control flow of the application is driven implicitly by callbacks, and developers may not understand or have access to the framework code. This is similar to the problems encountered in trying to debug a compiler lexical analyzer and parser written with LEX and YACC. In these applications, debugging is straightforward when the thread of control is in the user-defined action routines. Once the thread of control returns to the generated DFA skeleton, however, it is hard to trace the program's logic.

Efficiency. Frameworks enhance extensibility by employing additional levels of indirection. For instance, dynamic binding is commonly used to allow developers to subclass and customize existing interfaces. However, the resulting generality and flexibility often reduce efficiency. For instance, in languages like C++ and Java, the use of dynamic binding makes it impractical to support Concrete Data Types (CDTs), which are often required for time-critical software. The lack of CDTs yields (1) an increase in storage layout (for example, due to embedded pointers to virtual tables), (2) performance degradation (for example, due to the additional overhead of invoking a dynamically bound method and the inability to inline small methods), and (3) a lack of flexibility (for example, due to the inability to place objects in shared memory).

Lack of standards. Currently, there are no widely accepted standards for designing, implementing, documenting, and adapting frameworks. Moreover, emerging

industry standard frameworks (such as CORBA, DCOM, and Java RMI) currently lack the semantics, features, and interoperability to be truly effective across multiple application domains. Often, vendors use industry standards to sell proprietary software under the guise of open systems. Therefore, it is essential for companies and developers to work with standards organizations and middleware vendors to ensure that the emerging specifications support true interoperability and define features that meet their software needs.

Some of the problems with frameworks have been described already. In particular, because they are powerful and complex, they are hard to learn. This means that they require better documentation than other systems, and longer training. Moreover, they are hard to develop. This means that they cost more to develop and require different kinds of programmers than normal application development. These are some of the reasons that frameworks are not used more widely than they have been, in spite of the fact that the technology is so old. But these problems are shared with other reuse techniques. Although reuse is valuable, it is not free. Companies that are going to take advantage of reuse must pay its price.

One of the strengths of frameworks is that they are represented by normal object-oriented programming languages. This is also a weakness of frameworks. Since this feature is unique to frameworks, it is also a unique weakness.

One of the problems with using a particular language is that it restricts frameworks to systems using that language. In general, different object-oriented programming languages don't work well together, so it is not cost-effective to build an application in one language with a framework written in another. COM and CORBA address this problem, since they let programs in one language interoperate with programs in another. OLE is essentially a framework based on COM that lets applications be developed in any programming language. Moreover, some frameworks have been implemented twice so that users of two different languages can apply them [Andersen 1998; Eng 1996].

Current programming languages are good at describing the static interface of an object, but not its dynamic interface. Because frameworks are described with programming languages, it is hard for developers to learn the collaborative patterns of a framework by reading it. Instead, they depend on other documentation and talking to experts. This adds to the difficulty of learning a framework. One approach to this problem is to improve the documentation, such as with patterns. Another approach is to describe the constraints and interactions between components formally, such as with contracts [Helm 1990]. But since part of the strength of frameworks is the fact that the framework is expressed in code, it might be better to improve object-oriented languages so that they can express patterns of collaboration more clearly.

1.6 Reuse: Components versus Designs

Frameworks are just one of many reuse techniques [Kreuger 1992]. The ideal reuse technology provides components that can be easily connected to make a new system. The software developer does not have to know how the component is implemented, and the specification of the component is easy to understand. The resulting system will be efficient, easy to maintain, and reliable. The electric power system is like that; you

can buy a toaster from one store and a television from another, and they will both work at either your home or your office. Most people do not know Ohm's law, yet they have no trouble connecting a new toaster to the power system. Unfortunately, software is not nearly as composable as the electric power system.

When we design a software component, we always have to trade simplicity for power. A component that does one thing well is easy to use, but can be used in fewer cases. A component with many parameters and options can be used more often, but will be harder to learn to use. Reuse techniques range from the simple and inflexible to the complex and powerful. Those that let the developer make choices are usually more complicated and require more training on the part of the developer.

For example, the easiest way to get a compiler is to buy one. Most compilers compile only one language. On the other hand, you could build a compiler for your own language by reusing parts of the GUN C Compiler (gcc) [Stallman 1995], which has a parser generator and a reusable back end for code generation. It takes more work and expertise to build a compiler with gcc than it does just to use a compiler, but this approach lets you compile your own language. Finally, you might decide that gcc is not flexible enough, since your language might be concurrent or depend on garbage collection, so you write your compiler from scratch. Even though you don't reuse any code, you will probably still use many of the same design ideas as gcc, such as having a separate parser. You can learn these ideas from any good textbook on compilers.

A component represents code reuse. A textbook represents design reuse. The source for gcc lies somewhere in between. Reuse experts often claim that design reuse is more important than code reuse, mostly because it can be applied in more contexts and so is more common. Also, it is applied earlier in the development process and so can have a larger impact on a project. A developer's expertise is partly due to knowing designs that can be customized to fit a new problem. But most design reuse is informal. One of the main problems with reusing design information is capturing and expressing it [Biggerstaff 1996]. There is no standard design notation and there are no standard catalogs of designs to reuse. A single company can standardize, and some do. But this will not lead to industry-wide reuse.

The original vision of software reuse was based on components [McIlroy 1968]. In the beginning, commercial interest in object-oriented technology also focused on code reuse. More recently, pure design reuse has become popular, as seen in the form of patterns [Bushmann 1996; Coplien 1996; Fowler 1997; Gamma 1995; Vlissides 1996]. But frameworks are an intermediate form, part code reuse and part design reuse. Frameworks eliminate the need of a new design notation by using an object-oriented programming language as the design notation. Although programming languages suffer several defects as design notations, it is not necessary (though it might be desirable) to make specialized tools to use frameworks. Most programmers using a framework have no tools other than their compilers.

Reuse (and frameworks) may be motivated by many factors. One is to save time and money during development. The main purpose for many companies is to decrease time to market. But they find that the uniformity brought about by frameworks is also important. Graphical user interface frameworks give a set of applications a similar look and feel, and a reusable network interface means that all the applications that use it follow the same protocols. Uniformity reduces the cost of maintenance, too, since now maintenance programmers can move from one application to the next without

having to learn a new design. A final reason for frameworks is to enable customers to build open systems, so they can mix and match components from different vendors.

In spite of all these motivations, the predictions that were made when software reuse was first discussed 30 years ago still have not come true. Reuse is still a small part of most development projects. The one exception is in the world of object-oriented programming, where most environments have at least a user interface framework.

1.7 Application Frameworks versus Other Reuse Techniques

The ideal reuse technique is a component that exactly fits your needs and can be used without being customized or forcing you to learn how to use it. However, a component that fits today's needs perfectly might not fit tomorrow's. The more customizable a component is, the more likely it is to work in a particular situation, but the more work it takes to use it and to learn to use it.

Frameworks are a component in the sense that vendors sell them as products, and an application might use several frameworks bought from various vendors. But frameworks are much more customizable than most components. As a consequence, using a framework takes work even when you are familiar with it, and learning a new framework is hard. In return, frameworks are powerful; they can be used for just about any kind of application, and a good framework can reduce by an order of magnitude the amount of effort needed to develop customized applications.

It is probably better to think of frameworks and components as different, but cooperating, technologies. First, frameworks provide a reusable context for components. Each component makes assumptions about its environment. If components make different assumptions, then it is hard to use them together [Berlin 1990].

A framework will provide a standard way for components to handle errors, to exchange data, and to invoke operations on each other. The so-called component systems such as OLE, OpenDoc, and Beans, are really frameworks that solve standard problems that arise in building compound documents and other composite objects. But any kind of framework provides the standards that enable existing components to be reused.

A second way in which frameworks and components work together is that frameworks make it easier to develop new components. Applications seem infinitely variable, and no matter how good a component library is, it will eventually need new components. Frameworks let us make a new component (a user interface) out of smaller components (a widget). They also provide the specifications for new components and a template for their implementation.

Frameworks are similar to other techniques for reusing high-level design, such as templates [Spencer 1988] or schemas [Katz 1989]. The main difference is that frameworks are expressed in a programming language, but these other ways of reusing high-level design usually depend on a special-purpose design notation and require special software tools. The fact that frameworks are programs makes them easier for programmers to learn and to apply, but it also causes some problems that we will discuss later.

Frameworks are similar to application generators [Cleaveland 1988]. Application generators are based on a high-level, domain-specific language that is compiled to a

standard architecture. Designing a reusable class library is a lot like designing a programming language, except that the only concrete syntax is that of the language it is implemented in. A framework is already a standard architecture. Thus, except for syntax and the fact that the translator of an application generator can perform optimizations, the two techniques are similar. Although problem domain experts usually prefer their own syntax, expert programmers usually prefer frameworks because they are easier to extend and combine than application generators. In fact, it is common to combine frameworks and a domain-specific language. Programs in the language are translated into a set of objects in the framework. (See the Interpreter pattern [Gamma 1995].)

Frameworks are a kind of domain-specific architecture [Tracz 1994]. The main difference is that a framework is ultimately an object-oriented design, while a domain-specific architecture might not be.

Patterns have recently become a popular way to reuse design information in the object-oriented community [Coplien 1996; Gamma 1995; Vlissides 1996]. A pattern is an essay that describes a problem to be solved, a solution, and the context in which that solution works. It names a technique and describes its costs and benefits. Developers who share a set of patterns have a common vocabulary for describing their designs and also a way of making design trade-offs explicit. Patterns are supposed to describe recurring solutions that have stood the test of time.

Since some frameworks have been implemented several times, they represent a kind of pattern, too. See, for example, the definition of Model/View/Controller [Bushmann 1996]. However, frameworks are more than just ideas—they are also code. This code provides a way of testing whether a developer understands the framework, examples for learning it, and an oracle for answering questions about it. In addition, code reuse often makes it possible to build a simple application quickly, and that application can then grow into the final application as the developer learns the framework.

The patterns in the book *Design Patterns* [Gamma 1995] are closely related to frameworks in another way. These patterns were discovered by examining a number of frameworks and were chosen as being representative of reusable, object-oriented software. In general, a single framework will contain many of the patterns, so these patterns are smaller than frameworks. Moreover, the design patterns cannot be expressed as C++ or Smalltalk classes and then just reused by inheritance or composition. So those patterns are more abstract than frameworks. Frameworks are at a different level of abstraction from the patterns in *Design Patterns*. Design patterns are the architectural elements of frameworks.

For example, Model/View/Controller can be decomposed into three major design patterns and several less important ones [Gamma 1995]. It uses the Observer pattern to ensure that the view's picture of the model is up to date. It uses the Composite pattern to nest views. It uses the Strategy pattern to have views delegate responsibility for handling user events to their controller.

Frameworks are firmly in the middle of reuse techniques. They are more abstract and flexible (and harder to learn) than components, but more concrete and easier to reuse than a raw design (but less flexible and less likely to be applicable). They are most comparable to reuse techniques that reuse both design and code, such as application generators and templates. Their major advantage is also their major liability; they can be implemented using any object-oriented programming environment, since they are represented by a program.

1.8 How to Use Application Frameworks

There are several ways to use a framework. Some ways require a deeper knowledge of the framework than others. All of them are different from the usual way of developing software using object-oriented technology, since all of them force an application to fit the framework. Thus, the design of the application must start with the design of the framework.

An application developed using a framework has three parts: the framework, the concrete subclasses of the framework classes, and everything else. *Everything else* usually includes a script that specifies which concrete classes will be used and how they will be interconnected. It might also include objects that have no relationship to the framework or that use one or more framework objects, but that are not called by framework objects. Objects that are called by framework objects will have to participate in the collaborative model of the framework and so are part of the framework.

The easiest way to use a framework is to connect existing components. This does not change the framework or make any new concrete subclasses. It reuses the framework's interfaces and rules for connecting components and is most like building a circuit board by connecting integrated circuits or building a toy house from Legos. Application programmers only have to know that objects of type A are connected to objects of type B; they do not have to know the exact specification of A and B.

Not all frameworks can work this way. Sometimes every new use of a framework requires new subclasses of the framework. That leads to the next easiest way to use a framework, which is to define new concrete subclasses and use them to implement an application. Subclasses are tightly coupled to their superclasses, so this way of using a framework requires more knowledge about the abstract classes than the first way. The subclasses must meet the specification implied by the superclasses, so the programmer must understand the framework's interfaces in detail.

The way of using a framework that requires the most knowledge is to extend it by changing the abstract classes that form the core of the framework, usually by adding new operations or variables to them. This way is the most like fleshing out a skeleton of an application. It usually requires the source code of a framework. Although it is the hardest way to use a framework, it is also the most powerful. On the other hand, changes to the abstract classes can break existing concrete classes, and this way will not work when the main purpose of the framework is to build open systems.

If application programmers can use a framework by connecting components without having to look at their implementation, then the framework is a *blackbox framework*. Frameworks that rely on inheritance usually require more knowledge on the part of developers and so are called *whitebox frameworks*. Blackbox frameworks are easier to learn to use, but whitebox frameworks are often more powerful in the hands of experts. However, blackbox and whitebox frameworks are a spectrum, not a dichotomy. It is common for a framework to be used in a blackbox way most of the time and to be extended when the occasion demands. One of the big advantages of a blackbox framework over an application-specific language is that it can be treated like a whitebox framework and extended by making new concrete subclasses.

If a framework is blackbox enough, it is used just by instantiating existing classes and connecting them. Many of the graphical user interface frameworks are like this; most

user interfaces contain objects only of existing classes. When this is true, it is usually easy to make an application builder for the framework that can instantiate the components and connect them. This makes the framework even easier to use by novices.

All these ways of using a framework require mapping the structure of the problem to be solved onto the structure of the framework. A framework forces the application to reuse its design. The existing object-oriented design methods usually start with an analysis model and derive the design from it, but this will not work with frameworks unless the framework design informs the analysis model. In spite of the importance of frameworks, most object-oriented methods do not support them very well. One exception is the OORam method [Reenskaug 1996].

1.9 How to Learn Application Frameworks

Learning a framework is harder than learning a regular class library, because you can't learn just one class at a time. The classes in a framework are designed to work together, so you have to learn them all at once. Moreover, the important classes are abstract, which makes them even harder to learn. These classes do not implement all the behavior of framework components, but leave some to the concrete subclasses, so you have to learn what never changes in the framework and what is left to the components.

Frameworks are easier to learn if they have good documentation. Even fairly simple frameworks are easier to learn if good training is available, and complex frameworks require training. But what are the characteristics of good documentation and good training for a framework?

The best way to start learning a framework is by example. Most frameworks come with a set of examples that you can study, and those that don't are nearly impossible to learn. Examples are concrete, thus easier to understand than the framework as a whole. They solve a particular problem and you can study their execution to learn the flow of control inside the framework. They demonstrate both how objects in the framework behave and how programmers use the framework. Ideally, a framework should come with a set of examples that range from the trivial to the advanced, and these examples will exercise the full range of features of the framework.

Although some frameworks have little documentation other than the source code and a set of examples, ideally a framework will have a complete set of documentation. The documentation should explain:

- The purpose of the framework
- How to use the framework
- How the framework works

It is hard to explain the purpose of a framework. Framework documentation often devolves to jargon or marketing pitches. Until people had seen the Macintosh, the claim that Model/View/Controller implemented graphical user interfaces did not make sense, and until people tried to implement one, they didn't understand why they needed help. Often the best way to learn the range of applicability of a framework is by

example, which is another reason that it helps if a framework comes with a rich set of examples.

It is also hard to explain how to use a framework. Understanding the inner workings of a framework does not tell an application programmer which subclasses to make. The best documentation seems to be a kind of cookbook [Apple 1986; Johnson 1992; ParcPlace 1994]. Beginning programmers can use a cookbook to make their first applications, and more advanced programmers can use it to look up solutions to particular problems. Cookbooks don't help the most advanced programmers, but most framework application programmers need them.

A lot of framework documentation simply lists the classes in the framework and the methods of each class. This is not useful to beginners, any more than a programming language standard is useful to a new programmer. If the programming environment has a good browser, it isn't much use to anybody. Programmers need to understand the framework's big picture. Documentation that describes the inner workings of a framework should focus on the interaction between objects and how responsibility is partitioned between them.

The first use of a framework is always a learning experience. Pick a small application and one that the framework is obviously well suited for. Study similar examples and copy them. Use the cookbook to see how to implement individual features. Single-step through the application to learn the flow of control, but don't expect to understand everything. The purpose of a framework is to reuse design, so if it is a good framework, then there will be large parts of its design that you can reuse without knowing about them.

Once you've built an actual application, the framework documentation will become clearer. Since the first use of a framework is usually the hardest, it is a good idea to use it under the direction of an expert. This is one of the main reasons that mentoring is so popular in the Smalltalk community; each version of Smalltalk comes with a set of frameworks, and learning to program in Smalltalk is largely a matter of learning the frameworks.

Frameworks are complex, and one of the biggest problems with most frameworks is just learning how to use them. Framework developers need to make sure they document their framework well and develop good training material for it. Framework users should plan to devote time and budget to learning the framework.

1.10 How to Evaluate Application Frameworks

Sometimes it is easy to choose a framework. A company that wants to develop distributed applications over the Internet that run in web browsers will want to use Java and will prefer standard frameworks. A company that is Microsoft-standard will prefer MFCs to zApp. Most application domains have no commercially available domain-specific frameworks, and many others have only one, so the choice is either to use that one or not to use one. But many times there are competing frameworks, and they must be evaluated.

Many aspects of a framework are easy to evaluate. A framework needs to run on the right platforms, use the right programming language, and support the right standards. Of course, each organization has its own definition of *right*, but these questions are easy to answer. If they aren't in the documentation, the vendor can answer them.

It is harder to tell how reliable a framework is, how well its vendor supports it, and whether the final applications are sufficiently efficient. Vendors will not give reliable answers to these questions, but usually the existing customers will.

The hardest questions are whether a framework is suited for your problem and whether it makes the right trade-offs between power and simplicity. Frameworks that solve technical problems such as distribution or user interface design are relatively easy to evaluate. Even so, if you are distributing a system to make it more concurrent and so handle greater throughput, you might find that a particular distribution framework is too inefficient, because it focuses on flexibility and ease of use instead of speed. No framework is good for everything, and it can be hard to tell whether a particular framework is well suited for a particular problem.

The standard approach for evaluating software is to make a checklist of features that the software must support. Since frameworks are extensible, and since they probably are supposed to handle only part of the application, it is more important that a framework be easy to extend than that it have all the features. However, it is easier to tell whether a framework has some feature than to tell whether it is extensible. As usual, we are more likely to measure things that are easy to measure than the things that are important.

An expert can usually tell how hard it is to add some missing features, but it is nearly impossible for novices to tell. So it is best to use some frameworks and develop some experience and expertise with them before choosing one as a corporate standard. If frameworks are large and expensive, then it is expensive to test them, and there is no choice except to rely on consultants. Of course, the consultants might know only one framework well, they might be biased by business connections with the framework's vendor, and they will tend to favor power over simplicity. Every framework balances simplicity with power. Simplicity makes a framework easier to learn, so simple frameworks are best for novices. Experts appreciate power and flexibility. If you are going to use a framework only once, then the time spent to learn it might exceed the time spent actually using it, so simplicity would be more important than power. If you are going to use a framework many times, then power is probably more important. Thus, experts are sometimes less able to make a good choice than novices.

In the end, the main value of a framework is whether it improves the way you develop software and the software you develop. If the software is too slow or impossible to maintain, it will not matter that it takes less time to develop. Many factors go into this: the quality of the framework, tools to support the framework, the quality of the documentation, and a community to provide training and mentoring. A framework must fit into the culture of a company. If a company has high turnover and a small training budget, then frameworks must be simple and easy to use. Unless it repeatedly builds the same kind of applications, it should not be building domain-specific frameworks, regardless of how attractive they might be. The value of a framework depends more on its context than on the framework itself, so there is no magic formula for evaluating them.

1.11 How to Develop Application Frameworks

One of the most common observations about framework design is that it takes iteration [Johnson-Foote 1988]. Why is iteration necessary? Clearly, a design is iterated only because its authors did not know how to do it right the first time. Perhaps this is the fault of the designers; they should have spent more time analyzing the problem domain, or they were not skilled enough. However, even skilled designers iterate when they are designing frameworks.

The design of a framework is like the design of most reusable software [Krueger 1992; Tracz 1995]. It starts with domain analysis, which (among other things) collects a number of examples. The first version of the framework is usually designed to be able to implement the examples and is usually a whitebox framework [Johnson 1997; Roberts 1997]. Then the framework is used to build applications. These applications point out weak points in the framework, which are parts of the framework that are hard to change. Experience leads to improvements in the framework, which often make it more blackbox. Eventually the framework is good enough that suggestions for improvement are rare. At some point, the developers have to decide that the framework is finished and release it.

One of the reasons for iteration is domain analysis [Batory 1989]. Unless the domain is mature, it is hard for experts to explain it. Mistakes in domain analysis are discovered when a system is built, which leads to iteration.

A second reason for iteration is that a framework makes explicit the parts of a design that are likely to change. Features that are likely to change are implemented by components so that they can be changed easily. Components are easy to change; interfaces and shared invariants are hard. In general, the only way to learn what changes is by experience.

A third reason for iterating is that frameworks are abstractions, so the design of the framework depends on the original examples. Each example that is considered makes the framework more general and reusable. Frameworks are large, so it is too expensive to look at many examples, and paper designs are not sufficiently detailed to evaluate the framework. A better notation for describing frameworks might let more of the iteration take place during framework design.

A common mistake is to start using a framework while its design is changing. The more an immature framework is used, the more it changes. Changing a framework causes the applications that use it to change, too. On the other hand, the only way to find out what is wrong with a framework is to use it. First use the framework for some small pilot projects to make sure that it is sufficiently flexible and general. If it is not, these projects will be good test cases for the framework developers. A framework should not be used widely until it has proven itself, because the more widely a framework is used, the more expensive it is to change it.

Because frameworks require iteration and deep understanding of an application domain, it is hard to create them on schedule. Thus, framework design should never be on the critical path of an important project. This suggests that they should be developed by advanced development or research groups, not by product groups. On the

other hand, framework design must be closely associated with application developers because framework design requires experience in the application domain.

This tension between framework design and application design leads to two models of the process of framework design. One model has the framework designers also designing applications, but they divide their time into phases in which they extend the framework by applying it and phases in which they revise the framework by consolidating earlier extensions [Foote 1991]. The other model is to have a separate group of framework designers. The framework designers test their framework by using it, but also rely on the main users of the framework for feedback.

The first model ensures that the framework designers understand the problems with their framework, but the second model ensures that framework designers are given enough time to revise the framework. The first model works well for small groups whose management understands the importance of framework design and so can budget enough time for revising the framework. The second model works well for larger groups or for groups developing a framework for users outside their organization, but requires the framework designer to work hard to communicate with the framework users. This seems to be the model most popular in industry.

A compromise is to develop a framework in parallel with developing several applications that use it. Although this will not benefit these first applications much, the framework developers usually help more than they hurt. The benefits usually do not start to show until the third or fourth application, but this approach minimizes the cost of developing a framework while providing the feedback that the framework developers need.

1.12 Organization of This Book

This book is organized into eight major parts: Part One, "Framework Overview," Part Two, "Framework Perspectives," Part Three, "Frameworks and Domain Analysis," Part Four, "Framework Development Concepts," Part Five, "Framework Development Approaches," Part Six, "Framework Testing and Integration," Part Seven, "Framework Documentation," and Part Eight, "Framework Management and Economics."

Part One includes Chapters 1, 2, and 3 and Sidebar 1, and provides complete coverage of application framework issues, defines application frameworks, classifies the application frameworks, describes the characteristics of application frameworks, discusses the pros and cons of application frameworks, and contrasts the frameworks with other reuse approaches. Part One also elaborates on the understanding of the term *software architecture* by further developing the concept of architectural abstractions, and describes the major problems with application frameworks.

Part Two includes Chapters 4, 5, and 6 and Sidebar 2, and discusses several perspectives of application frameworks related to some of the historical application frameworks, describes guidelines for constructing good classes and components for application frameworks, discusses general guidelines for application framework usability, and presents two types of viewpoints: inter- and intra-viewpoints.

Part Three contains Chapters 7 and 8 and Sidebar 3, and discusses the relationships between application frameworks and domain analysis, as well as how to drive application frameworks from domain knowledge.

Part Four consists of Chapters 9 through 14 and Sidebar 4, and discusses several new application framework concepts, such as the hooks approach and framework recipes.

Part Five consists of Chapters 15 through 19 and Sidebars 5 and 6, and discusses framework development approaches, such as systematic generalization, hot-spot-driven development, framework layering, framelets, understanding macroscopic behavior patterns in use-case maps, and composing modeling frameworks in Catalysis, as well as enduring business themes (EBTs).

Part Six is made up of Chapter 20 and Sidebar 7, and discusses issues related to framework testing and integration.

Part Seven comprises Chapters 21, 22, and 23 and Sidebar 8, and covers several topics related to framework documentation.

Part Eight contains Chapters 24, 25, and 26 and Sidebar 9, and covers framework investment analysis, framework structural and functional stability evaluation, framework management, and future research direction in the component-based framework technology.

1.13 Summary

Frameworks are a practical way to express reusable designs. They deserve the attention of both the software engineering research community and practicing software engineers. There are many open research problems associated with better ways to express and develop frameworks, but they have already shown themselves to be valuable.

The articles in this book reinforce the belief that object-oriented application frameworks will be at the core of leading-edge software technology in the twenty-first century. As software systems become increasingly complex, object-oriented application frameworks are becoming increasingly important for industry and academia. The extensive focus on application frameworks in the object-oriented community offers software developers an important vehicle for reuse and a means to capture the essence of successful patterns, architectures, components, and programming mechanisms.

The good news is that frameworks are becoming mainstream and developers at all levels are increasingly adopting and succeeding with framework technologies. However, OO application frameworks are ultimately only as good as the people who build and use them. Creating robust, efficient, and reusable application frameworks requires development teams with a wide range of skills. We need expert analysts and designers who have mastered patterns, software architectures, and protocols in order to alleviate the inherent and accidental complexities of complex software. Likewise, we need expert middleware developers who can implement these patterns, architectures, and protocols within reusable frameworks. In addition, we need application programmers who have the motivation, skills, and training to learn how to use these frameworks effectively. We encourage you to get involved with others working on frameworks by attending conferences, participating in online mailing lists and newsgroups, and contributing your insights and experience.

1.14 References

[Andersen 1998] Andersen Consulting. *Eagle Architecture Specification*. 1998.

[Apple 1986] Apple Computer. *MacApp Programmer's Guide*. 1986.

[Batory 1989] Batory, D.S., J.R. Barnett, J. Roy, B.C. Twichell, and J. Garza. Construction of file management systems from software components. *Proceedings of COMPSAC 1989*.

[Beck 1994] Beck, Kent, and Ralph Johnson. Patterns generate architectures. *European Conference on Object-Oriented Programming*, pp. 139–149, Bologna, Italy, July 1994.

[Berlin 1990] Berlin, Lucy. When objects collide: Experiences with using multiple class hierarchies. *Proceedings of OOPSLA 1990*, pp. 181–193, October 1990. Printed as SIGPLAN Notices, 25(10).

[Biggerstaff 1996] Biggerstaff, Ted J., and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software* 4(2):41–49, March 1987.

[Birrer 1993] Birrer, Eggenschwiler T. Frameworks in the financial engineering domain: An experience report. *ECOOP 1993 Proceedings, Lecture Notes in Computer Science no. 707*, 1993.

[Bushmann 1996] Bushmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, West Sussex, England: John Wiley & Sons. 1996.

[Campbell-Islam 1993] Campbell, Roy H., and Nayeem Islam. A technique for documenting the framework of an object-oriented system. *Computing Systems* 6(4), Fall 1993.

[Cleaveland 1988] Cleaveland, J.C. Building application generators. *IEEE Software* 4(5):25–33, July 1988.

[Coplien 1996] Coplien, James O. *Patterns*. New York: SIGS, 1996.

[Deutsch 1989] Deutsch, L. Peter. Design reuse and frameworks in the Smalltalk-80 programming system. In *Software Reusability, Volume II*, pp. 55–71, Ted J. Biggerstaff and Alan J. Perlis, editors. Reading, MA: ACM Press/Addison Wesley, 1989.

[Eng 1996] Eng, Lawrence, Ken Freed, Jim Hollister, Carla Jobe, Paul McGuire, Alan Moser, Vinayak Parikh, Margaret Pratt, Fred Waskiewicz, and Frank Yeager. *Computer Integrated Manufacturing (CIM) Application Framework Specification 1.3*. Technical Report Technology Transfer 93061697F-ENG, SEMATECH, 1996.

[Fayad 1996] Fayad, M.E., W.T. Tsai, and M. Fulghum. Transition to object-oriented software development. *Communications of the ACM* 39(2), February 1996.

[Fayad 1999] Fayad, M.E. Application frameworks. *ACM Computing Surveys Symposium*, March 1999.

[Fayad-Cline 1996a] Fayad, M.E., and M. Cline. Aspects of software adaptability. *Communications of the ACM, Theme Issue on Software Patterns* 39(10), October 1996. D. Schmidt, M.E. Fayad, and R. Johnson, guest editors.

[Fayad-Cline 1996b] Fayad, M.E., and M. Cline. Managing object-oriented software development. *IEEE Computer*, September 1996.

[Fayad-Hamu 1999] Fayad, Mohamed E., and David S. Hamu. Object-Oriented Enterprise Frameworks. Submitted for publication to *IEEE Computer*.

[Fayad-Laitinen 1998] Fayad, M.E., and M. Laitinen. *Transition to Object-Oriented Software Development*. New York: John Wiley & Sons, 1989.

[Fayad-Schmidt 1997] Fayad, M.E., and Douglas Schmidt. Object-oriented application frameworks. *Communications of the ACM* 40(10), October 1997.

[Fayad-Tsai 1995] Fayad, M.E., and W.T. Tsai. Object-oriented experiences. *Communications of the ACM*, 38 (10), October 1995.

[Foote 1988] Foote, Brian. Designing to facilitate change with object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[Foote 1991] Foote, Brian. The lifecycle of object-oriented frameworks: A fractal perspective. Technical Report, University of Illinois at Urbana-Champaign, 1991.

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

[Gamma 1995] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[Goldberg 1984] Goldberg, Adele. *Smalltalk-80: The Interactive Programming Environment.* Reading, MA: Addison-Wesley, 1984.

[Gossain 1990] Gossain, Sanjiv. Object-oriented development and reuse. Ph.D. thesis, University of Essex, UK, June 1990.

[Hamu-Fayad 1998] Hamu, David S., and Mohamed E. Fayad. Achieve bottom-line improvements with enterprise frameworks. *Communications of the ACM,* 41 (8), August 1998.

[Helm 1990] Helm, Richard, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. *Proceedings of OOPSLA 1990*, pp. 169–180, October 1990. Printed as SIGPLAN Notices, 25(10).

[Hueni 1995] Hueni, Herman, Ralph Johnson, and Robert Engel. A framework for network protocol software. *Proceedings of OOPSLA 1995,* Austin, TX, October 1995.

[Johnson 1992] Johnson, Ralph E. Documenting frameworks using patterns. *Proceedings of OOPSLA 1992*, pp. 63–76, Vancouver, BC, October 1992.

[Johnson 1997] Johnson, Ralph E. Frameworks = (Components + Patterns). *Communications of the ACM,* in Object-Oriented Application Frameworks Theme Issue, M.E. Fayad and D.C. Schmidt, editors, 40 (10), October 1997, 39-42.

[Johnson-Foote 1988] Johnson, Ralph E., and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(5), June/July. 1988: 22–35.

[Katz 1989] Katz, Shmuel, Charles A. Richter, and Khe-Sing The. Paris: A system for reusing partially interpreted schemas. In *Software Reusability, Volume I*, pp. 257–273, Ted J. Biggerstaff and Alan J. Perlis, editors. Reading, MA: ACM Press/Addison Wesley, 1989.

[Krasner 1988] Krasner, Glenn E., and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1(3):26–49, August/September 1988.

[Krueger 1992] Krueger, Charles W. Software reuse. *ACM Computing Surveys* 24(2):131–183, June 1992.

[LaLonde 1991] LaLonde, W.R. and J.R. Pugh. *Inside Smalltalk Vol. II.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Linton 1989] Linton, Mark A., John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer* 22(2):8–22, February 1989.

[McIlroy 1968] McIlroy, M.D., Mass produced software components. In *Software Engi-*

neering: Report on a Conference by the NATO Science Committee, pp. 138–150, P. Naur and B. Randall, editors. NATO Scientific Affairs Division, 1968.

[Meyrowitz 1986] Meyrowitz, Norman. Intermedia: The architecture and construction of an object-oriented hypermedia system and application framework. *Proceedings of OOPSLA 1986*, pp. 186–201, November 1986. Printed as SIGPLAN Notices 21(11).

[Palay 1988] Palay, A.J., W.J. Hansen, M.L. Kazar, M. Sherman, M.G. Wadlow, T.P. Neuendorffer, Z. Stern, M. Bader, and T. Petre. The Andrew Toolkit—an overview. *Proceedings of the Winter 1988 USENIX Conference*, Dallas, TX, 1988.

[ParcPlace 1988] ParcPlace Systems, Inc. *Smalltalk-80 Reference Manual*. 1988.

[ParcPlace 1994] ParcPlace Systems, Inc. *VisualWorks Cookbook*. 1994.

[Pree 1995] Pree, Wolfgang. *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1995.

[Reenskaug 1996] Reenskaug, Trygve. *Working with Objects: The OORam Software Engineering Method*. Greenwich, CT: Manning, 1996.

[Roberts 1997] Roberts, Don, and Ralph Johnson. *Evolving Frameworks: A Pattern Language for Developing Frameworks*. Reading, MA: Addison-Wesley, 1997.

[Russo 1990] Russo, Vincent F. An object-oriented operating system. Ph.D. thesis, University of Illinois at Urbana-Champaign, October 1990.

[Schmid 1995] Schmid, Hans Albrecht. Creating the architecture of a manufacturing framework by design patterns. *Proceedings of OOPSLA 1995*, pp. 370–384, Austin, TX, July 1995.

[Schmidt 1997] Schmidt, Douglas C. *Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software* (Handbook of Programming Languages), vol. I. Edited by Peter Salus. New York: Macmillan Computer Publishing, 1997.

[Schmucker 1986] Schmucker, Kurt J. *Object-Oriented Programming for the Macintosh*. Hayden Book Company, 1986.

[Spencer 1988] Spencer, Henry. How to steal code. *Proceedings of the Winter 1988 USENIX Technical Conference*, Dallas, TX, 1988.

[Stallman 1995] Stallman, Richard. *Using and Porting GNU CC*. Boston, MA: Free Software Foundation, 1995.

[Tracz 1994] Tracz, Will. DSSA frequently asked questions. *ACM Software Engineering Notes* 19(2):52–56, April 1994.

[Tracz 1995] Tracz, Will. *Domain Specific Software Architecture Engineering Process Guidelines*, Appendix A. Reading, MA: Addison-Wesley, 1995.

[Vlissides 1989] Vlissides, John M., and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *Proceedings of the ACM User Interface Software and Technologies 1989 Conference*, pp. 81–94, November 1989.

[Vlissides 1996] Vlissides, John M., James O. Coplien, and Norman L. Kerth, eds. *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996.

[Weinand 1988] Weinand, A., E. Gamma, and R. Marty. ET++: An object-oriented application framework in C++. *Proceedings of OOPSLA '88*, pp. 46–57, November 1988. Printed as SIGPLAN Notices 23(11).

[Weinand 1989] Weinand, A., E. Gamma, and R. Marty. Design and implementation of {ET++}, a seamless object-oriented application framework. *Structured Programming* 10(2):63–87, 1989.

[Wirfs-Brock 1990] Wirfs-Brock, Rebecca J., and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM* 33(9):104–124, 1990.

[Yassin-Fayad 1999] Yassin, A.F., and M.E. Fayad. *Domain-Specific Application Frameworks*, Chapter 29, Mohamed E. Fayad and Ralph Johnson, editors, New York: John Wiley & Sons, 1999.

1.15 Review Questions

1. What is an application framework?
2. Describe briefly the benefits of application frameworks.
3. Give examples for whitebox, graybox, and blackbox frameworks.
4. Give examples for enterprise frameworks.
5. What are the differences between whitebox and blackbox frameworks?
6. What are the differences between system frameworks and enterprise frameworks?
7. What are the differences between application frameworks and design patterns?
8. What are the differences between application frameworks and software components?
9. What are the differences between application frameworks and class libraries?
10. What are the differences between application frameworks and application generators?
11. State the strengths of application frameworks.
12. State the weaknesses of application frameworks.
13. Explain how to use application frameworks.
14. Explain how to develop application frameworks.
15. Explain how to learn application frameworks.
16. What do the following terms stand for?

    MFC, GUI, ORB, CDT, CORBA, DCOM, EBT, and MVC