Components, Frameworks, Patterns

Ralph E. Johnson

February 21, 1997

Abstract

Frameworks are an object-oriented reuse technique that are widely used in industry but not discussed much by the software engineering research community. They are a way of reusing design that is part of the reason that some object-oriented developers are so productive. This paper compares and contrasts frameworks with other reuse techniques, and describes how to use them, how to evaluate them, and how to develop them. It describe the tradeoffs involved in using frameworks, including the costs and pitfalls, and when frameworks are appropriate.

1 Introduction

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 definitions of frameworks vary. The definition we use most 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

Authors' address: Ralph Johnson, Department of Computer Science, University of Illinois, 1304 West Springfield Ave., Urbana IL 61801, Telephone: (217) 244-0093, e-mail: johnson@cs.uiuc.edu

purpose. Nevertheless, they point out the difficulty of defining frameworks clearly.

Frameworks are important, and becoming more important. Systems like OLE, OpenDoc, and DSOM are frameworks. The rise of Java is spreading new frameworks like AWT 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 paper compares and contrasts frameworks with other reuse techniques, and describes how to use them, how to evaluate them, and how to develop them. It describe the tradeoffs involved in using frameworks, including the costs and pitfalls, and when frameworks are appropriate.

2 Components vs. Designs

Frameworks are just one of many reuse techniques[kreuger92]. 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 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 only compile one language. On the other hand, you could build a compiler for your own language by reusing parts of gcc[Sta95], which has a parser generator and a reusable backend 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[BR87]. 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[McI68]. In the beginning, commercial interest in object-oriented technology also focused on code reuse, as indicated by Brad Cox's Software ICs[Cox86]. More recently, pure design reuse has become popular, as seen in the form of patterns [GHJV95, BMR+96, CS95, VCK96, Fow97]. 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) has many motivations. The main one is to save time and money during development. Time to market is increasingly important, and many companies build frameworks to decrease time to market. But they find that the uniformity caused by frameworks is just as important. Graphical user interface frameworks give a set of applications a similar look and feel, and a reusable network interface mean 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.

3 What is a Framework?

One of the key ideas that underlies frameworks is that of the *abstract class*. An abstract class is a class with no instances, so it is used only as a superclass [GR83, WBJ90]. 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 a template for creating objects. Frameworks use them as designs of their components.

One way that an abstract class acts as a design is that it specifies the interface of its subclasses. When a client indicates that it wants to use an instance of an abstract class, it always means an instance of a class that meets the interface of the abstract class.

An abstract class usually provides part of the implementation of its subclasses, too. For example, a *template method* defines the skeleton of an algorithm in an abstract class, deferring some of the steps to subclasses [GHJV95]. 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 (i.e. they are abstract methods) or can provide a default implementation (i.e. they are hook methods) [Pre95]. 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.

A framework is a larger-scale design that describes how a program is decomposed into a set of interacting objects. It is usually represented as a set of abstract classes and the way their instances interact [JF88, WBJ90]. Since it includes the way that instances of those classes interact, it is the collaborative model or pattern of object interaction as much as it is the kinds of classes in the design. The most important part of a framework is the way that a system is divided into its components[Deu87, Deu89]. 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.

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 run-time, 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 calls the components whenever necessary. The developer decides when to call the components, and is 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 developers 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 70's, was the Smalltalk-80 user interface framework called Model/View/Controller (MVC) [Gol84, KP88, LP91]. MVC showed that object-oriented programming was well-suited for implementing graphical user interfaces. It divides a user interface into three kinds of components; models, views and controllers. These objects work in trios consisting of a view and 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 shows a picture of 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 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 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

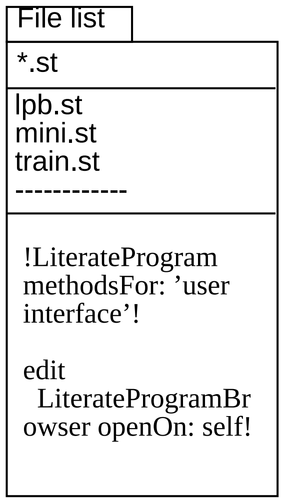

Figure 1: The Smalltalk-80 File Tool

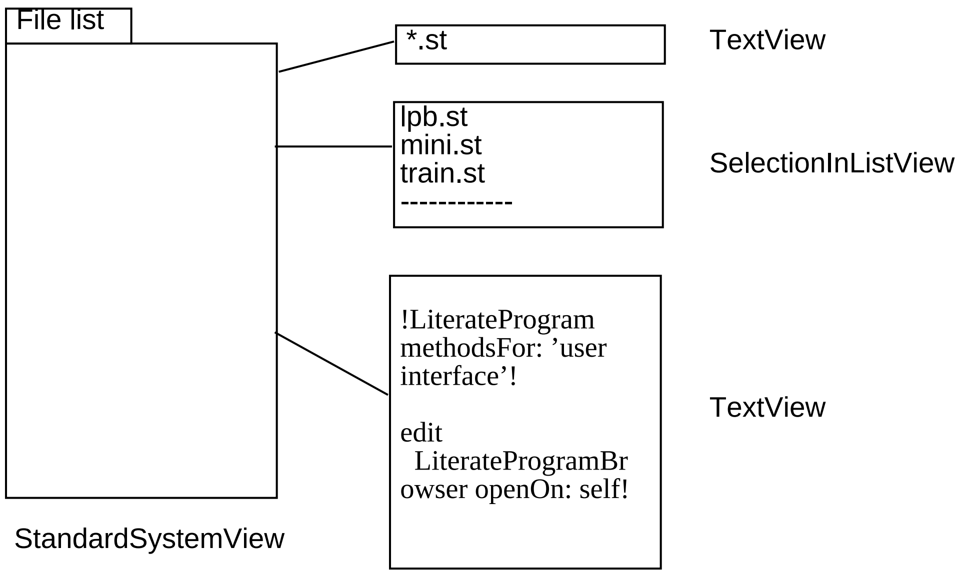

Figure 2: Subview Hierarchy in File Tool

template operations for displaying, since different kinds of views require different display algorithms. TextView, SelectionInListView, and StandardSystemView are concrete subclasses of View that each have 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 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. Later versions of Smalltalk-80 have added a Model class that provides a more efficient version of the dependency mechanism[PS88].

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 [Sch86]. It was followed by user interface frameworks from universities, such as the Andrew Toolkit from Carnegie Mellon University [PHK+88], InterViews from Stanford [LVC89], and ET++ from the University of Zurich [WGM88, WGM89]. There are now a large number of commercial user interface frameworks, such as OWL, zAPP, OpenStep, and MFC. 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 VLSI routing algorithms [Gos90], to hypermedia systems [Mey86], to structured drawing editors [VL89, Vli90, BJ94], operating systems [RC89, Rus90], psychophysiological experiments [Foo88], network protocol software [HJE95], and manufacturing control [Sch95], 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 [Bat88, BBR+89] as well as a tool for specifying how DBMSs are built from the framework [BB91]. Genesis does not use an object-oriented language but rather a macro processor 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 Model/View/Controller, 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 using it as a component, so it is better to reuse existing components than to make a new one.

Of course, the main reason that a framework enables code reuse is because 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 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 [BR87].

4 Frameworks vs. Other Things

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 venders sell them as products, and an application might use several frameworks bought from various venders. But frameworks are much more customizable than most components. As a consequence, using a framework takes work even when you know 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 the amount of effort to develop customized applications by an order of magnitude.

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[Ber90]. 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 (i.e. a user interface) out of smaller components (i.e. 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 [Spe88, VK89] or schemas [KRT89, LH87]. 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 [Cle88]. 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[GHJV95]).

Frameworks are a kind of domain-specific architecture [Tra94]. 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 [GHJV95, Cop96, CS95, VCK96]. 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 tradeoffs 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 in Bushmann et. al. [BMR+96]. 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"[GHJV95] 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 than 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 [GHJV95]. 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 programming language.

5 How to use a Framework

There are several ways to use a framework. Some of them 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 the most like building a circuit board by connecting integrated circuits, or building a toy house from Legos. The application programmer only has to know that objects of type A are connected to objects of type B, he does 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 to 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 together without having to look at their implementation then the framework is a "black-box" framework. Frameworks that rely on inheritance usually require more knowledge on the part of developers, and so are called "white-box" frameworks. Black-box frameworks are easier to learn to use, but white-box frameworks are often more powerful in the hands of experts. But black-box and white-box frameworks are a spectrum, not a dichotomy. It is common for a framework to be used in a black-box way most of the time, and to be extended when the occasion demands. One of the big advantages of a black-box framework over an application specific language is that it can be extended by making new concrete subclasses.

If a framework is black-box enough, it is used just by instantiating existing classes and connecting them together. Many of the graphical user interface frameworks are like this, most user interfaces only contain objects 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 [Ree96].

6 How to learn a framework

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 there is good training, and complex frameworks require training. But what are the characteristics of good documention 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, and so 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 seems to be 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 why 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 [Com86, Joh92, PS94]. 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 users, but they are very important for beginners.

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 more clear. 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 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.

7 How to evaluate a framework

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 MFC to OWL or 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 tradeoffs 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 check list of features that the software must support. Since frameworks are extensible, and since they probably are only supposed to handle 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 then 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 only know one framework well, they might be biased by business connections with the frameworks 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 only going to learn a framework once then the time to learn it might dominate the time actually using it, so simplicity is more important than power. If you are going to use a framework over and over 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 turn-over and a small training budget then frameworks must be simple and easy to use. Unless it repeatedly builds the same kind of applications then it should 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.

8 How to Develop a Framework

One of the most common observations about framework design is that it takes iteration [JF88, WB90, Ros95]. 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[Kru92, Tra95]. 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 white-box framework [JF88, RJ97]. 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 black-box. This process can continue forever, though 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 [BBR+89]. 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 still changing. The more an immature framework is used, the more it changes. Changing a framework causes the applications that use it to change, too[Ros95]. On the other hand, the only way to find out what is wrong with a framework is to use it. It is better to 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 the 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 design applications, but they divide their time into phases when they extend the framework by applying it and phases when they revise the framework by consolidating earlier extensions [Foo91]. 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.

9 Problems with Frameworks

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 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 languge interoperate with programs in another. Further, some frameworks have been implemented twice so that users of two different languages can apply them [EFH+96, Con].

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 [HHG90]. 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.

10 Conclusion

Frameworks are a practical way to express reusable designs. They deserves 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.

References

[Bat88] D.S. Batory. Concepts for a database system compiler. *Principles of Database Systems*, 1988.

[BB91] D.S. Batory and J.R. Barnett. Date: The Genesis DBMS software layout editor. Technical report, Department of Computer Sciences, University of Texas at Austin, 1991.

[BBR+89] D.S. Batory, J.R. Barnett, J. Roy, B.C. Twichell, and J. Garza. Construction of file management systems from software components. In *Proceedings of COMPSAC 1989*, 1989.

[Ber90] Lucy Berlin. When objects collide: Experiences with using multiple class hierarchies. In *Proceedings of OOPSLA '90*, pages 181–193, October 1990. printed as SIGPLAN Notices, 25(10).

[BJ94] Kent Beck and Ralph Johnson. Patterns generate architectures. In *European Conference on Object-Oriented Programming*, pages 139–149, Bologna, Italy, July 1994. Springer-Verlag.

[BMR+96] Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, Chichester, West Sussex, England, 1996.

[BR87] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–49, March 1987.

[Cle88] J. C. Cleaveland. Building application generators. *IEEE Software*, 4(5):25–33, July 1988.

[Com86] Apple Computer. *MacApp Programmer's Guide*. 1986.

[Con] Andersen Consulting. Eagle architecture specification. http://www.ac.com/eagle/spec/.

[Cop96] James O. Coplien. *Patterns*. SIGS, New York, NY, 1996.

[Cox86] Brad J. Cox. *Object Oriented Programming*. Addison-Wesley, Reading, Massachusetts, 1986.

[CS95] James O. Coplien and Doug Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1995.

[Deu87] L. Peter Deutsch. Levels of reuse in the Smalltalk-80 programming system. In Peter Freeman, editor, *Tutorial: Software Reusability*. IEEE Computer Society Press, 1987.

[Deu89] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 programming system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Vol II*, pages 55–71. ACM Press, 1989.

[EFH+96] Lawrence Eng, 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.

[Foo88] Brian Foote. Designing to facilitate change with object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[Foo91] Brian Foote. The lifecycle of object-oriented frameworks: A fractal perspective. University of Illinois at Urbana-Champaign, 1991.

[Fow97] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Massachusetts, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[Gos90] Sanjiv Gossain. *Object-Oriented Development and Reuse*. PhD thesis, University of Essex, UK, June 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA '90*, pages 169–180, October 1990. printed as SIGPLAN Notices, 25(10).

[HJE95] Hermann Huni, Ralph Johnson, and Robert Engel. A framework for network protocol software. In *Proceedings of OOPSLA '95*, pages 358–369, Austin, Texas, July 1995. ACM.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of OOPSLA '92*, pages 63–76, Vancouver, British Columbia, October 1992. ACM.

[KP88] Glenn E. Krasner 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/Spetember 1988.

[KRT89] Shmuel Katz, Charles A. Richter, and Khe-Sing The. Paris: A system for reusing partially interpreted schemas. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Vol I*, pages 257–273. ACM Press, 1989.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[LH87] Mitchell D. Lubars and Mehdi T. Harandi. Knowledge-based software design using design schemas. In *Proc. 9th Intl. Conf. on Software Engineering*, pages 253–262, March 1987.

[LP91] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk, Volume II.* Prentice Hall, 1991.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[McI68] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randall, editors, *Software Engineering: Report on a conference by the NATO Science Committee*, pages 138–150. NATO Scientific Affairs Division, 1968.

[Mey86] Norman Meyrowitz. Intermedia: The architecture and construction of an object-oriented hypermedia system and application framework. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 186–201, November 1986. printed as SIGPLAN Notices, 21(11).

[PHK+88] A. J. Palay, 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. In *USENIX Association Winter Conference*, Dallas, 1988.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, Reading, Massachusetts, 1995.

[PS88] Inc. ParcPlace Systems. *Smalltalk-80 Reference Manual.* 1988.

[PS94] Inc. ParcPlace Systems. *VisualWorks Cookbook*. 1994.

[RC89] Vincent Russo and Roy H. Campbell. Virtual memory and backing storage management in multiprocessor operating systems using class hierarchical design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[Ree96] Trygve Reenskaug. *Working with Objects: The OORam Software Engineering Method*. Manning, Greenwich, CT, 1996.

[RJ97] Don Roberts and Ralph Johnson. *Evolving Frameworks: A Pattern Language for Developing Frameworks*. Addison-Wesley, Reading, Massachusetts, 1997.

[Ros95] Larry Rosenstein. *MacApp: First Commercially Successful Framework*, chapter 5. Manning, Greenwich, CT, 1995.

[Rus90] Vincent F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[Sch86] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh*. Hayden Book Company, 1986.

[Sch95] Hans Albrecht Schmidt. Creating the architecture of a manufacturing framework by design patterns. In *Proceedings of OOPSLA '95*, pages 370–384, Austin, Texas, July 1995. ACM.

[Spe88] Henry Spencer. How to steal code. In *Proceedings of the Winter 1988 Usenix Technical Conference*, 1988.

[Sta95] Richard Stallman. *Using and Porting GNU CC*. Free Software Foundation, Boston, MA, 1995.

[Tra94] Will Tracz. Dssa frequently asked questions. *ACM Software Engineering Notes*, 19(2):52–56, April 1994.

[Tra95] Will Tracz. *Domain Specific Software Architecture Engineering Process Guidelines*, chapter Appendix A. Addison-Wesley, 1995.

[VCK96] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, Massachusetts, 1996.

[VK89] Dennis M. Volpano and Richard B. Kieburtz. The templates approach to software reuse. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Vol I*, pages 247–255. ACM Press, 1989.

[VL89] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM User Interface Software and Technologies '89 Conference*, pages 81–94, November 1989.

[Vli90] John M. Vlissides. *Generalized Graphical Object Editing*. PhD thesis, Stanford University, June 1990.

[WB90] Allan Wirfs-Brock. Ecoop/oopsla'90 panel on designing reusable frameworks, October 1990.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++: An object-oriented application framework in C++. In *Proceedings of OOPSLA '88*, pages 46–57, November 1988. printed as SIGPLAN Notices, 23(11).

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.