How frameworks compare to other object-oriented reuse techniques.

FRAMEWORKS = (COMPONENTS + PATTERNS)

Frameworks are an object-oriented reuse technique. They share many characteristics with reuse techniques in general [8], 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, they are not well understood outside the object-oriented community and are often misused. Moreover, there is confusion about whether frameworks are large-scale patterns, or whether they are just another kind of component.

Ralph E. Johnson

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

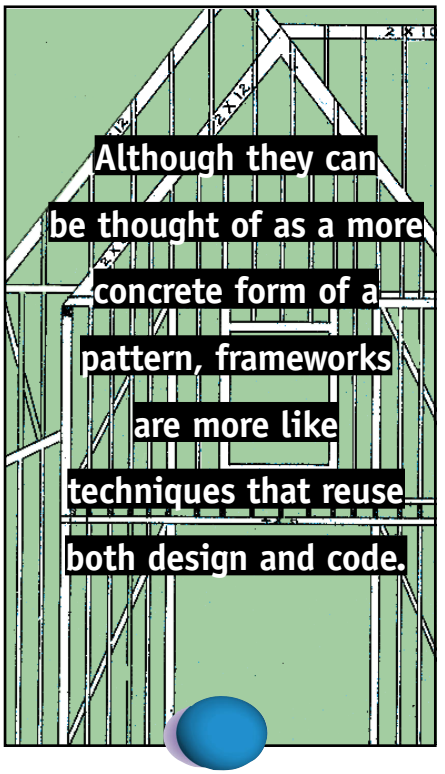
Frameworks are important, and continually become more important. Systems like OLE, OpenDoc, and DSOM are frameworks; Java is spreading new frameworks like AWT and Beans. Most commercially available frameworks seem to be for technical domains such as user interfaces or distribution, and most application-specific frameworks are proprietary. But the steady rise of frameworks means every software developer should know what they are and how to deal with them.

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 it is easy for the developer to learn how to use it. 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.

The original vision of software reuse was based on components. In the beginning, commercial interest in object-oriented technology also focused on reusable

components, as illustrated by Brad Cox's Software ICs [5]. However, object-oriented technology has not created a market in reusable components. This has happened in the Visual Basic market, in part because of the dominance of Microsoft, in part because Visual Basic is simple and easy to use. But frameworks are not software components as they were originally foreseen.

Frameworks are a component in the sense that venders sell them as products, and an application might use several frameworks. But frameworks are more customizable than most components, and have more complex interfaces. Programmers must learn these interfaces before they can use the framework, and, consequentially, 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 best 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. 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 standard interfaces 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 (such as a user interface) out of smaller components (such as a widget) and they also provide the specifications for new components and a template for implementing them.

Frameworks as Reusable Design

Designers often trade simplicity for power. A simple asset will be easy to use, but can be used in fewer cases. A generic asset with many parameters and options can be used more often, but will be harder to learn to use.

For example, it is cheaper to buy a compiler than to build one. Most compilers only compile one language. On the other hand, you could build a compiler for your own language by reusing parts of the GNU C compiler, gcc, which has a parser generator and a reusable backend for code generation. It takes more work and expertise to build a compiler by reusing parts of gcc than it does just to use a compiler, but then you can 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. Design reuse has advantages over code reuse [1]. 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. But most design reuse is informal, and happens through using experienced developers. 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.

Frameworks are a form of design reuse. They are similar to other techniques for reusing high-level design, such as templates [12] or schemas [9]. 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. They don't need any tools except their compilers, and they can gradually change an application into a framework. On the other hand, it means that frameworks

tend to be specific to a programming language. Moreover, some design ideas, such as behavioral constraints, cannot be expressed well in current languages.

Frameworks are similar to application generators [3], which usually compile a high-level, domain-specific language to a standard architecture. Designing a framework is like designing a programming language, except that the only concrete syntax is the one used to implement the framework. Also, the translator of an application generator can (but usually doesn't) perform optimizations. Problem domain experts usually prefer their own syntax, but expert programmers usually prefer frameworks because they are easier to extend and combine than special-purpose languages. It is possible to combine frameworks and a domain-specific language by translating programs in the language into a set of objects in the framework [10]. The window builders associated with GUI frameworks are examples of domain-specific visual programming languages.

Frameworks are a kind of domain-specific architecture [11]. The main difference between them is that a framework is ultimately an object-oriented design, while a domain-specific architecture might not be.

Because these techniques are similar, they share similar benefits. They all can save time and money during development. Time to market is increasingly important, and is the main reason many companies build frameworks. But they find that the uniformity caused by reuse is just as important. Graphical user interface frameworks give a set of applications a similar look and feel, and using a distributed object framework ensures that all applications can communicate with each other. 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.

One motivation of design reuse is to enable open systems, so developers can mix and match components from different vendors. This implies reusing interface design, and is one of the motivations for object-oriented systems like CORBA. So far, though, these systems have focused on lower-level interfaces and not reusable designs.

These reuse techniques share similar costs. In particular, they all require domain analysis and domain engineering, so there is a big expense before benefits can be realized. All reuse techniques require that developers be trained to use the artifact, they create dependences on the reused artifacts, and they often introduce inefficiencies. Before using any of them, the costs and benefits should be analyzed.

Because frameworks do not require any tools other than those needed for an object-oriented programming language, they tend to appear wherever object-oriented languages are used. In fact, developers often do not even know they are using a framework, but just talk about the "class library." Frameworks differ from other class libraries by reusing high-level design. This means that there is more to learn before a class can be reused, they can never be reused in isolation, and typically a set of classes must be learned at once. You can often tell that a class library is a framework if there are dependencies among its components and if programmers who are learning it complain about its complexity.

Frameworks and Patterns

Patterns have recently become a popular way to reuse design information in the object-oriented community [2, 4, 6]. A pattern 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. Model/View/Controller is a user-interface framework that is described as a pattern in Bushmann et al. [1]. Moreover, applications that use frameworks must conform to the frameworks' design and model of collaboration, so the framework causes patterns in the applications that use it. 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* [6] 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. A single framework usually contains many 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. Therefore, 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 micro-architectural elements of frameworks.

For example, Model/View/Controller can be decomposed into three major design patterns, and several less important ones [6]. It uses the Observer pattern to ensure the view's picture of the model is up-to-date, the Composite pattern to nest views, and the Strategy pattern to cause views to delegate responsibility for handling user events to their controller.

Frameworks are firmly in the middle of reuse techniques. They are more abstract and flexible than components, but more concrete and easier to reuse than a pure design (but less flexible and less likely to be applicable). Although they can be thought of as a more concrete form of a pattern, frameworks are more like techniques that reuse both design and code, such as application generators and templates. Patterns are illustrated by programs, but a framework is a program.

Problems with Frameworks

Some of the problems with frameworks have been described already. Because they are powerful and complex, they are difficult to learn. This means they require better documentation and longer training than other systems. They are hard to develop, therefore they cost more to develop and require better programmers than normal application development. These are some of the reasons frameworks are not used more widely, in spite of the fact that the technology is 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 traditional object-oriented programming languages. This is also a weakness of frameworks, however, and it is one that the other design-oriented reuse techniques do not share.

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 use them, such as the SEMATECH CIM framework described in this issue.

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. Patterns are one approach to improving the documentation. Another approach is to describe the constraints and interactions between components formally, such as with contracts [7]. 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.

Frameworks are a practical way to express reusable designs. They deserve the attention of both researchers and practitioners. Although we need better ways to express and develop frameworks, they have already shown themselves to be valuable. C

References

1. Biggerstaff, T.J., and Richter, C. Reusability framework, assessment, and directions. *IEEE Software 4*, 2 (Mar. 1987), 41–49.
2. Bushmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, West Sussex, England, 1996.
3. Cleaveland, J.C. Building application generators. *IEEE Software* 4, 5 (July 1988), 25–33.
4. Coplien, J.O. *Patterns*. SIGS Publications, NY, 1996.
5. Cox, B.J. *Object-Oriented Programming*. Addison-Wesley, Reading, Mass., 1986.
6. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
7. Helm, R., Holland, I.M., and Gangopadhyay, D. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA '90*, (Oct. 1990), pp. 169–180; printed as SIGPLAN Notices 25, 10.
8. Krueger, C.W. Software reuse. *ACM Comput. Surveys 24*, 2 (June 1992), 131–183.
9. Lubars, M.D. and Harandi, M.T. Knowledge-based software design using design schemas. In *Proceedings of the 9th International Conference on Software Engineering* (Mar. 1987), pp. 253–262.
10. Roberts, D. and Johnson, R. Evolving frameworks: A pattern language for developing frameworks. In D. Riehle, F. Buschmann, and R.C. Martin, Eds., *Pattern Languages of Program Design 3*, Addison-Wesley, Reading, Mass., 1997.
11. Tracz, W. Dssa frequently asked questions. *ACM Software Engineering Notes*, 19, 2 (Apr. 1994), 52–56.
12. Volpano, D.M. and Kieburtz, R.B. The templates approach to software reuse. In T.J. Biggerstaff and A.J. Perlis, Eds., *Software Reusability, Vol. I*, ACM Press, 1989.

Ralph E. Johnson (johnson@cs.uiuc.edu) is Coordinator of Project Design Activity in the Department of Computer Science at the University of Illinois-Urbana.