Frameworks Versus Libraries: A Dichotomy of Reuse Strategies

Mitchell D. Lubars

Electronic Data Systems, Research and Development
1601 Rio Grande, Suite 500
Austin, TX 78701
Tel: (512) 477-1892
Email: lubars@austin.eds.com

Neil Iscoe

Electronic Data Systems, Research and Development
1601 Rio Grande, Suite 500
Austin, TX 78701
Tel: (512) 477-1892
Email: iscoe@austin.eds.com

Abstract

In general, setting up reuse libraries for random collections of software components has not been a very effective reuse strategy for software organizations. Rather, most examples of successful reuse libraries actually consists of reusable components that are structured and organized according to well-defined frameworks for their use. Often these frameworks correspond to reusable architectures in a particular application domain, a common set of experiences in the software community, a set of standards or protocols, or a well-structured system layer. One situation where reuse with a random collections of components does seem to payoff, however, is in the reuse of large scale components in a very large community, such as that supported by the Internet through FTP sites and shareware. This position paper discusses the problems with random reuse libraries and the advantages offered by focusing on frameworks and domain analysis for reusability.

Keywords: Reuse, framework, library, component

Workshop Goals: Networking, Advance state of theory.

Working Groups: Domain analysis/engineering, Design guidelines for reuse, Reuse and OO methods, Reuse and formal methods, Tools and environments, Reuse handbook, Education.

1 Background

The majority of our involvement in software reuse has been concerned with constructing systems to support software design reuse and researching domain analysis techniques and representations to develop reusable software. The design reuse systems, IDeA, ROSE-1, and ROSE-2, contain collections of *design schemas*, each of which is an abstract reusable design, similar in principle to reusable software architectures and frameworks. The user interface system CRTForm is based on a framework of language-independent functions that provide a firewall between users and application programs.

We are currently investigating domain modeling representations to express commonality in business-related application areas and exploring the use of formal techniques to check the consistency of the domain models and to use the models for generating domain-specific programs.

2 Position

2.1 Introduction

For many years now, a popular approach for starting a reuse program at an organization has been to set up a reuse library mechanism and encourage developers to contribute components to the library and search the library for components that they need in new situations. In some cases, organizations have set up elaborate incentive programs to encourage people to contribute components (usually functions or classes) to and to look for components in those libraries. Anecdotal evidence suggests that where these efforts have focused on random and spontaneous contributions of small components, the efforts have not generally been successful. The major reasons for this lack of success are two fold.

1. A random collection of reusable components cannot be expected to seamlessly fit together with the other components that someone is developing.
2. The effort to search a large collection of random components is frequently not worth the expected reward (partly because of the first point).

Where reuse programs have been most successful, the reusable components were generally constructed and organized according to a well-defined framework, either through a careful domain analysis of an application domain, or through evolution of a large base of experience over the years.

There is one intermediate reuse situation, however, where reuse libraries of randomly contributed components do seem to have high success rates. This is where the granularity of the reusable components is very large, such as in the size of a subsystem, supporting application domain, or a complete mini-framework. There is significant anecdotal evidence to support the importance of this kind of reuse. In particular, the browsing of FTP sites, distribution of shareware, and simple cataloging schemes like *Archie* show that this type of reuse occurs at significant levels.

This position paper discusses some of the properties and advantages of the reuse situations, and its ramifications for library approaches.

2.2 The Problems with Most Random Reuse Libraries

When starting a reuse program in an organization, a very tempting technique is to create a library, provide some mechanisms for classifying the components and browsing the library, declare that the library exists, and encourage developers to contribute components. These components are usually subroutines, functions, or classes. One theory is that as the library grows, the chances will increase that components that a programmer needs will be available in the library. That will increase the likelihood that the programmer will look in the library for desirable components, instead of writing them himself.

Often there is little intrinsic incentive to contribute to the library, so the organization may create extrinsic incentives to encourage programmers to contribute their code, such as providing cash rewards. One of the problems that can result from this is that the library can fill up with lots of random, even poorly written, code that most programmers will never reuse. Techniques to deal with these problems include library administrators to regulate what goes in the library, librarians to provide assistance, review boards to review the quality of reusable components, and elaborate classification schemes to organize the library material.

Although these techniques help somewhat, they don't address the real problem which is that any random piece of code, no matter how reusable and well-written, is unlikely to be reusable by itself in a new program. To be reused, the code often requires many other pieces of code to be reused at the same time or require that a lot of additional code be written to make the reused code fit into the program.

The underlying situation is that the original code was written to work in some framework that the programmer had in mind; an architecture for the program he was developing or some understanding about how many pieces of code work together to accomplish an objective. While it is true that high cohesion and low coupling increase the reusability of a piece of software, it is unusual that any small piece of code accomplishes something useful completely on its own. The way that the piece is used, and how it interfaces to the other components, is embedded in the framework for that piece.

Often object-oriented advocates say that objects are inherently reusable and object-oriented programming techniques promote reusability. There is some truth to this in that objects package a set of related methods together so that each object embodies a mini-framework. However, this usually isn't enough, since an object-oriented program consists of a large set of collaborating objects to accomplish shared responsibilities. In this view, individual objects are often not reusable unless their object-oriented framework and related objects are also reused. A single object is still rarely reusable all by itself.

2.3 What is a Software Framework?

A framework is a set of principles that describe how a set of software components interact with one another to accomplish a set of shared responsibilities. Often frameworks are considered to be synonymous with architectures. There is a subtle distinction, however. An architecture is concerned more with the physical layout, such as allocation to modules, processes and processors, the aggregation and scoping of components, and the physical nature of communication between the pieces. Frameworks are more concerned with the conceptual nature and conventions of the communication between the pieces and their inter-dependencies. Both frameworks and architectures are related to one another and both affect reusability. For the purpose of this discussion, framework will be used to generally refer to all communication and dependency issues (both conceptual and physical).

There are also several varieties of frameworks, which focus on different kinds of dependency and communication issues. Some are more concerned with constructing components that work together as units to accomplish higher-level responsibilities. These often correspond to modules, or program subsystems, or supporting domains. For example, frameworks may be available for constructing databases, device drivers, or file systems. These frameworks normally prescribe ways that the components must be interconnected to solve problems. They also tend to be rather application-specific.

Most object-oriented frameworks [1, 2] are of this type. They define the abstract classes that must be in the system and the protocol that must be supported between those classes. Each concrete class must support the protocol and a set of such concrete classes must be reused together to support the framework. A given concrete class is not reusable by itself, because it only provides part of the framework's protocol.

Another type of framework defines a set of components that may be used together to solve problems, but it is up to the user to compose the components together. Often these frameworks are constructed to provide an orthogonal, but composable, set of behaviors. Typical examples are well-engineered subroutine or class libraries, such as math libraries, window-system libraries, and statistical packages. Usually these frameworks can be thought of as providing a *layer*, abstract machine, or supporting domain in a complex system.

Another type of framework defines and enforces a set of standards in some domain. Toolkits, such as Motif, which supports a common look and feel, or protocol-supporting libraries are typical examples of these frameworks. These frameworks may also be thought of as providing a *layer* or supporting domain in a complex system, with additional supporting policy.

2.4 Most Successful Reuse Libraries are Frameworks in Disguise

Most examples of successful reuse libraries are actually well-engineered frameworks, rather than random reuse libraries. Math libraries and statistical packages are based on a mature understanding of subject matter, which has evolved over many years. Well before the packages were written, there was general agreement in the math and engineering communities of what the useful functions would be. Providing consistent typing and nomenclature further makes these libraries into a coherent framework. Had the functions been provided in a library, but with random interfaces, the functions would have been considerably less reusable.

Similarly, data structure function and class libraries support frameworks. The set of useful data structure and common operations on them (such as searching and sorting) have been well-known and taught for many years. In addition, providing some unifying principles, such as viewing them as different kinds of collections with some common attributes and operations, further strengthens the framework. When someone reuses a data structure, he is not reusing a single class or a small set of functions, but he is reusing the knowledge of the data structure and the set of principles for using data structures that he shares with all other computer scientists.

Many other examples of reuse success stories, such as the Toshiba software factory [3] and similar examples in Japan and the U.S. [4], are also really framework successes. These successes occur in application domains in which the organizations have considerable experience in developing similar applications. Over the years, a framework (or architecture) is developed for the application domain, which is embodied in the expertise of the developers. When these developers contribute components to the reuse library, they are not contributing random components. Instead they are contributing components that they know will fit within the framework for their domain.

Similarly, when developers look for reusable components, they are looking for something that fits into a specific portion of the framework. They are not browsing through a large collection of randomly contributed components. Thus, cataloging components and searching through the framework-supporting library are not serious problems, as they might be for a random reuse library.

2.5 Reuse of Random Large Scale Components

The general thesis of this position paper has been that reuse of a random collection of software components does not work well. There is, however, a significant counter-example. That example has to do with reuse of random software components across the Internet and through shareware.

Many reusable pieces of software are available at a variety of FTP sites across the U.S. and even other countries. Many people regularly browse these FTP sites looking for reusable software. In some cases, they are looking for software to meet a specific need, and in some cases they are simply looking for interesting software that they might experiment with and maybe find a use for later on. There are some primitive means for cataloging and organizing this information, such as local hierarchical file system structures at each FTP site and *Archie* for cataloging and querying.

Other reusable software is available through computer news groups, distribution tapes, and other shareware sources. Much of this software is poorly cataloged, if cataloged at all. Yet people go to the trouble to make it available and search it for useful components.

The big question is why is this completely ad hoc and voluntary reuse process successful when so many attempts to set up random reuse libraries in organizations have failed? There are three primary reasons for its success.

1. Each component embodies a framework so that it can usually be reused by itself. The components do not require reusing a significant set of other components at the same time.
2. The components are rather large. It would take a substantial amount of work and time to reconstruct the components from scratch.
3. The availability of the reusable components is almost unlimited, but the actual reuse rates are small. However, taken together these variables make the absolute amount of reuse significant.

The fact that the components are large-scale is significant. People are much more willing to look for a reusable piece of software in a random collection that will save them a substantial amount of development effort, than they are for a small function or object. But this by itself is not sufficient. In addition, the component must stand-alone. It must, by itself, embody a useful framework that people want to reuse, such as a graphic editor, a paint program, a spreadsheet program, a protocol library, a language interpretor, a compiler, or a window toolkit. Frequently, these large scale reusable components correspond to supporting domains, modules, or subsystems. In many cases, people are even more interested in reusing the frameworks than the actual code. They are interested in how the editor works or what the protocol is, rather than (or in addition to) the code. These issues together provide a large granularity of reusability.

The wide availability of the code is also significant. First, because the code is so widely distributed, there is often some increased confidence (even though this is often misguided) that the code is widely used and perhaps more reliable than code in a more localized random library. This increases the chance that people are willing to try it out. Even if it doesn't work right, there is a belief that someone will soon post a fix or an improved version of it. A corollary to this principle is that if someone is willing to distribute their software that widely, they must have a significant level of pride in it, and they must have spent a lot of time to engineer it for reusability.

Another consequence of the wide availability is that if only one out of 1000 programmers who have access to the software uses it, that would amount to very significant reuse. That is simply not true for reuse libraries in any single organization. For most organizations, if one out of 1000 programmers reuse any software component in the library, the library would be considered a dismal failure. Indeed it is likely that reuse rates for most reusable components in the Internet and shareware sources are quite low; much lower than would be cost effective within any one organization.

2.6 Conclusions

The overall conclusion of this position paper is that organizations that are interested in starting a reuse program should not attempt to construct a library and populate it with a random collection of reusable components. Rather, they should conduct a domain analysis of their application domain (or domains) to determine a framework or architecture for the domain, and then only construct reusable components that fit within that framework. If they take this approach, then the problems of cataloging, searching, and retrieving components in a library can be reduced or even obviated.

These conclusions are not particularly novel, as is evidenced by the increasing interest in domain analysis in recent years [5, 6, 7, 8]. Unfortunately, there still tend to be many examples of organizations and researchers heading back down the path of building better tools and techniques to aid in constructing yet more random reuse libraries. This is witnessed by a continuing stream of new papers in the literature and questions on harvesting reusable components from poor legacy code, new variations of software cataloging and retrieval techniques, and new incentive strategies to make programmers contribute to reuse libraries. Many of these efforts could be better directed to helping organizations discover the frameworks underlying their application domains and engineering the components to work together within the framework, or to constructing and organizing large-scale reusable components in the software community at large.

3 Comparison

The framework concepts presented in this position paper are similar to the object-oriented frameworks advocated by Peter Deutsch [1], Johnson and Foot [2], and others. Another similar area is that of domain-specific software architectures (DSSA), which provide reusable architectures for specific application domains.

The design schemas that are used in the IDeA [9], ROSE-1 [10], and ROSE-2 [11] design reuse systems are also similar to frameworks. In addition, they contain constraints and inference rules to choose consistent design components and semi-automatically customize abstract designs. Similar techniques to reusable design schemas include the plan calculus techniques used in the Programmer's Apprentice project [12] and subsequent work including the Requirement's Apprentice [13].

The user interface system CRTForm [14] is also based on a framework, supplying language-independent functions that provide a firewall between users and application programs. Another framework-based software reuse system that is similar to IDeA, ROSE-1, ROSE-2, and CRTForm is GENESIS [15], which generates different kinds of databases systems based on a reusable database architecture.

Our domain modeling work at the EDS Research Lab is also similar to other work in object-oriented frameworks, but focuses more on the data modeling aspects of frameworks, such as the data constraints and rules that apply to all programs within an application domain.

References

[1] L. Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System," in *Software Reusability Volume II: Applications and Experience* (T. J. Biggerstaff and A. J. Perlis, eds.), pp. 57–71, Addison Wesley, 1989.

[2] R. Johnson and B. Foot, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 2, pp. 22–35, June/July 1988.

[3] Y. Matsumoto, "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," in *Software Reusability Volume II: Applications and Experience* (T. J. Biggerstaff and A. J. Perlis, eds.), pp. 157–185, Addison Wesley, 1989.

[4] R. Lanergan and C. Grasso, "Software Engineering with Reusable Design and Code," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 498–501, September 1984.

[5] R. Prieto-Diaz, "Domain Analysis for Reusability," in *Proceedings of COMPSAC '87*, pp. 23–29, 1987.

[6] G. Arango, "Domain Analysis: From Art to Engineering Discipline," in *Proceedings Fifth International Workshop on Software Specification and Design*, pp. 152–159, May 19-20 1989.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[8] M. Lubars, "Domain Analysis and Domain Engineering in IDeA," in *Domain Analysis and Software systems Modeling* (R. Prieto-Diaz and G. Arango, eds.), IEEE Computer Society Press, 1991.

[9] M. Lubars, "A Knowledge-Based Design Aid for the Construction of Software Systems," Tech. Rep. UIUCDCS-R-86-1304, Department of Computer Science, University of Illinois, Urbana, IL., November 1986.

[10] M. Lubars, "Wide-Spectrum Support for Software Reusability," in *IEEE Tutorial, Software Reuse: Emerging Technology* (W. Tracz, ed.), pp. 275–281, The Computer Society of the IEEE, 1988.

[11] M. Lubars and M. Harandi, "Addressing Software Reuse Through Knowledge-Based Design," in *Software Reusability Volume II: Applications and Experience* (T. J. Biggerstaff and A. J. Perlis, eds.), Addison Wesley, 1989.

[12] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, (Vancouver, Canada), August 1981.

[13] C. Rich, R. Waters, and H. Reubenstein, "Toward a Requirements Apprentice," in *Proceedings of the Fourth International Workshop on Software Specification and Design*, (Monterey, CA), April 1987.

[14] N. Iscoe, "CRTForm: An Object-Oriented Application Development System Using Type and Type-Type Managers," Tech. Rep. Masters Report, Department of Computer Sciences, University of Texas, Austin, TX., August 1986.

[15] D. Batory and A. Buchmann, "Molecular Objects, Abstract Data Types, and Data Models: A Framework," in *Proceedings of the Tenth International Conference on Very Large Databases*, (Singapore), pp. 172–184, August 1984.

4 Biography

Mitchell D. Lubars is a research scientist at Electronic Data Systems, Research and Development in Austin Texas. He is part of a team developing a domain modeling language and tools to support the application-specific domain modeling needs within EDS. He also provides part-time consulting in the areas of software reusability, domain analysis, and object-oriented analysis and design.

Prior to coming to EDS, Dr. Lubars was a member of the technical staff at MCC for six years, working in the areas of software design reusability and requirements analysis. While at MCC, he developed the ROSE-1 and ROSE-2 design reuse systems.

Dr. Lubars received the A.B. degree in biology from Cornell University, in 1977. He received the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign, in 1980 and 1986 respectively. He is a member of the Association for Computing Machinery, the IEEE Computer Society, and AAAI.

Neil Iscoe is director of the EDS Austin Laboratory for Software Engineering and Computer Science, which is part of the Electronic Data Systems Research and Development organization. Prior to EDS, Iscoe had extensive experience with large industrial applications at MCC (Microelectronics and Computer Consortium). He conducted field studies and performed analyses of large software projects in application domains such as telephony, command and control, manufacturing, avionics, and other real time distributed application areas. Prior to MCC, he served as president of Statcom Corporation, a company that produced and marketed a line of CASE tools and provided consulting services for programming companies. As a founder of the company, Iscoe designed the initial product and also served as the company's technical director.

Dr. Iscoe received his Ph.D. and M.S. in Computer Sciences from the University of Texas at Austin, in 1990 and 1988 respectively, and his undergraduate engineering degree from the University of Wisconsin in Madison in 1977. He serves as an adjunct assistant professor at the University of Texas.