Strategies for a Implementing a Successful Reuse Library

By Dr. Jeffrey S. Poulin
Systems Architect and author of Measuring Software Reuse, *published by Addison-Wesley*
11 July 2001

Years of research and experience have gone into building reuse libraries, populating them with components, and managing them as part of a software reuse program. However, much of this work goes unnoticed as organizations make haste to reap the significant business benefits available through reuse. Before beginning a strategic initiative such as the launch of a new reuse library, it makes sense to map out how you plan to integrate reuse into your application development process and ultimately your business.

This article will explain the reuse strategy that I have found essential to implementing a successful reuse library.

What to avoid: The traditional life of a reuse library

The funny thing about most reuse libraries is that, at first, they usually don't contain anything! This is like a store with nothing on its shelves — how can it possibly attract customers? If your organization has made the commitment to a software reuse program and has provided the tools to make it happen, you must now assemble a critical mass of reusable assets to justify your investment.

Most reuse libraries have traditionally gone through a natural three-phase progression of ineffectiveness [Poulin94]. As mentioned above, the first phase begins when an organization initially installs a new library system. Most out-of-the-box reuse libraries come without any reusable components. It goes without saying that if your library contains few or no components, your developers will hardly find it worth their time!

The second phase results from the natural reaction to having no reusable components: load the library as quickly as you can. In this phase, organizations go hunting for anything "reusable." However, while their mining expeditions will cause the libraries to quickly fill up, the organization soon learns that it has done little more than collect a lot of junk. Developers still will not use the library because they know that it mostly contains useless software of what I will politely call "questionable quality and heritage."

Finally, in the third phase, the organization attempts to fix the junk collected in phase two. The organization does this by establishing formal rules and certification criteria in an attempt to assure developers that the library contains quality components. While this strategy will indeed improve the quality of the contents of the library, it also has a number of adverse side effects. First, it increases the complexity of the process of submitting components. Second, with complexity comes increased cost. Third, tedious procedures scare off potential contributors as well as potential reusers. In the end, the organization still has not fixed the essential problem: the library does not contain software that the developers need!

To avoid this self-defeating progression, you must establish a strategy for your reuse library by which you supply your developers with components that they can use.

How to succeed: Set your sights on useful components

It's an obvious point, but successful reuse libraries contain *useful* components [Poulin95]. Most developers have prior experience using general-purpose, low-level routines because they can easily use this class of component in almost any situation. These routines, therefore, serve as the low-hanging fruit in the reuse world and provide the component foundation that belongs in any reuse library. However, because of the low-level generality of this class of software, it will rarely make up more than 20% of any application. This is certainly a respectable benefit, but we would like to do better.

To achieve the high return on investment possible with higher levels of software reuse, we must identify components that have more reuse potential than low-level utility routines. This begs the question: what software repeatedly forms the core of our applications?

Enter the idea of *domain-specific components*. Most organizations carve a niche in their market by specializing in a specific technical area, or *domain*, and then build a business by developing a family of related products within that domain. These related products usually require a core set of domain-specific components that, if engineered and managed for reuse, can make up as much as 65% of an application.

Table 1 summarizes the three classes of software that contribute to a well-engineered application within a domain or product line. The final 15% of an application normally comes from *application-specific components* that implement custom functionality for a customer and provide little opportunity for reuse. But this means that an organization can achieve reuse levels of up to 85% by directing its efforts to building and managing the software most used by their developers.

Class	Type of Component	Advantage	Disadvantage	Examples
1	Domain-independent, "General Purpose"	Reusable by anyone	Limited to about 20% of an application	Math/GUI routines, Abstract data types, Communications libraries
2	Domain-specific, "Business Objects"	Can lead to reuse levels of up to 65% within a product line	Requires up-front planning	Financial functions for banks, Inventory functions for warehouses
3	Application-specific	Customizes applications	Very limited reuse potential	User preferences, Custom logic

Table 1- How the different classes of components contribute to reuse levels

Reuse libraries succeed when they contain and help to distribute the domain-specific components that form the core of an organization's applications. Therefore, a strategy for populating a new reuse library must focus on identifying these components and managing them as the crown jewels of the reuse program.

Domains and Product Lines: Domains and Product Lines refer to highly related functional areas and applications in business or technology. The scope of a domain can vary, but examples include the domain of financial services in banking and the domain of flight control software in airplanes. Whereas a domain refers to a functional area, a product line refers to the series of related software applications that we build within a functional area [Poulin97]. For example, in the domain of computer printer control software we may have a product line of device drivers for a family of ink-jet printers. For more information, see the Software Engineering Institute (SEI) web site on Product Line Practice (PLP) Initiative at: http://www.sei.cmu.edu/plp/plp_init.html

How do you successfully implement such a strategy? Start by applying the same principles that you used when developing your business strategy. First, clearly define the market niche or scope of the applications that you build. This niche should map to your organization's core competency or to an area of technology that you clearly understand. Next, select the core functions that you repeatedly use when building applications for that niche. Stabilize the software that implements these functions (if you have not done so already) using standard configuration management techniques. Finally, package the functions, load them into your reuse library, and promote their use in future applications by recruiting reuse advocates among your developers.

Conclusion

Traditional approaches to reuse libraries tried to succeed with a shotgun approach to software reuse. But we now know that a targeted strategy works best, and that the most successful reuse libraries contain as little as 30 to perhaps 250 highly used components. The high level of reusability of these components results from an organization's adherence to a strategy that exploits the "domain-specific" functions at the core of its business. Organizations that position their reuse libraries to take advantage of the high levels of reuse possible within their product lines have the potential to see enormous returns on their reuse investment.

References

[Poulin94] Poulin, Jeffrey S., "Balancing the Need for Large Corporate and Small Domain-Specific Reuse Libraries," *Reusability Track of the 1994 ACM Symposium on Applied Computing (SAC'94),* Phoenix, Arizona, 6-8 March 1994, pp. 88-93.

[Poulin95] Poulin, Jeffrey S., "Domain Analysis and Engineering: How Domain-Specific Frameworks Increase Software Reuse," *Proceedings of CASE JAPAN'95,* Tokyo, Japan, 12-15 July 1995, pp. B-3/1-3.

[Poulin97] Poulin, Jeffrey S. "Domains, Product Lines, and Software Architectures: Choosing the Appropriate Level of Abstraction," *Proceedings of the 8th International Workshop on Software Reuse (WISR'97),* Columbus, OH, 23-26 March 1997.