How to Solve the Reuse Problem: Find Out What Isn't Reusable and Don't Use It

James M. Neighbors
Bayfront Technologies, Inc.
1280 Bison B9-231
Newport Beach, CA 92660
neighbor@BayfrontTechnologies.com

1. What Isn't Reusable

The title of this position paper might seem a foolish answer but inverting the problem makes it easier to see the reuse constraints. Members of our panel represent the latest software technology applied to reuse. I am sure that not one of our panelists will claim to have found *the* answer. As with most problems, a solution will combine these technologies into a development process. The question is when and where to use each technology

In the discussion that follows, "It" pertains to all of the reusable artifacts and techniques that we discuss in software reuse. Each of the following sections discusses a constraint to reuse and suggests a panel member whose works focuses on eliminating that constraint.

1.1 It isn't reusable if it doesn't compile, link, and execute as it used to.

This is the untold secret of software reuse. Domain and code libraries decay and they must be maintained. The decay stems from changes in the version and configuration space. There are versions of the library (e.g., Numerical Recipes [1]), versions of the tools that compile the library (e.g., Microsoft C v5.0), and versions of the environment (e.g., Microsoft Windows® 98). There are configurations of the library (e.g., Numerical Recipes in C and FORTRAN), configurations of the tools that support the library (e.g., Microsoft C v5.0 compiler switches), and configurations of the environment (e.g., Microsoft Windows® 98 with TCP/IP networking). A new compiler and configuration might cause serious damage to an existing code library base. Consider that the cost of library maintenance should be spread across many projects. This makes the maintenance an organizational overhead expense. This type of expense is typically difficult to justify in organizations. The alternative of making the library a cost center can inhibit organizational reuse because each project using the library has to pay a fee.

On our panel, Jim Waldo's work on Java directly addresses the issue of code stability. Java promises a stable, compatible and safe programming environment with a controlled configuration and version space. In the past market forces have made this difficult.

1.2 It isn't reusable if you don't understand what it does.

The history of computing is driven by optimization. There is always the desire for more to be done with current computing resources. This will always be the case. However, currently it seems that software production is more a constraint than hardware speed. Today's emphasis seems to be on getting the software to work at all. Assuming we use algorithms whose complexity will scale, we know that Moore's Law will provide us with raw computing power. However, a legacy of the optimization era is the common use of side-effecting operations. As an example, CPUs set many condition bits on different operations. These side effects are not a problem by themselves, but when mixed with multiprocessing, caching, and pipelining they become a complex issue. Compiler builders and programmers spend quite a bit of time understanding these side effects. The same principle applies to application-oriented operations. In graphics, the idea that a drawline function updates the pen position is not a problem - until two processes draw using the same pen at the same time.

On our panel, Phillip Wadler advocates functional programming to aid program understanding. Functional programming abhors side effects and makes programs easier to understand because the context of each function is local to invocation.

1.3 It isn't reusable if it doesn't address the user's problem.

During development, the user's problem evolves from what to model in the problem domain to how to implement and ensure a long system lifetime. Thus, the

user's problem is not constrained to the user's problem domain. However, Software Engineering has found that a majority of the effort to build a new system is spent in the requirements, analysis, and design phases of system development. Thus, if implementation is only 15% of the effort of development, then a technique that completely automates implementation only improves the process by 15%. Domain Analysis attempts to reuse as many artifacts as possible from the early phases of system development. This, of course, only works for developing systems from problem domains that have been analyzed.

On our panel David Weiss' work on program families best typifies a solution to this constraint.

1.4 It isn't reusable if it can't interface to existing systems.

Most systems have to interface to an operating environment. Even stand-alone systems have to interface to ever more complex hardware subsystems. Presently most systems interface to some subset of graphics, networking and database subsystems. These interactions are evolving to standard complex protocols with hardware independent data. The benefit of such a standard to reuse is that large grain reusable subsystems may be made to encapsulate services.

On our panel, Tony Williams' work on COM/DCOM addresses the environment and development of these protocols. The work also addresses schemes for identifying versions and configurations.

2. Judging the technology

A "Magic Bullet" reuse technology is one that meets *all* of the above constraints in *all* system development cases. Clearly, we cannot expect such a technology to exist. There are always new problem domains, new versions, new configurations, new notations and new subsystems. The challenge of being a Software Architect is to minimize the effect of these changes through the lifetime of the system while still providing the system's functional requirements.

All of the technologies represented on our panel address the above constraints to some degree. They also all violate the constraints to some degree. *Each* new system development project requires the architect to determine which techniques are appropriate for that project.

3. References

[1] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 1986.