Object-Oriented Software Reuse in Business Systems

Daniel Brandon, Jr.
Christian Brothers University, USA

INTRODUCTION

"Reuse [software] engineering is a process where a technology asset is designed and developed following architectural principles, and with the intent of being reused in the future" (Bean, 1999). "If programming has a Holy Grail, widespread code reuse is it with a silver bullet. While IT has made and continues to make laudable progress in our reuse, we never seem to make great strides in this area" (Grinzo, 1998). "The quest for that Holy Grail has taken many developers over many years down unproductive paths" (Bowen, 1997). This article is an overview of software reuse methods, particularly object oriented, that have been found effective in business systems over the years.

BACKGROUND

Traditional software development is characterized by many disturbing but well documented facts, including:

- Most software development projects "fail" (60%) (Williamson, 1999).
- The supply of qualified IT professionals is much less than the demand (www.bls.gov).
- The complexity of software is constantly increasing.
- IT needs "better," "cheaper," "faster" software development methods.

Over the years, IT theorists and practitioners have come up with a number of business and technical methods to address these problems and improve the software development process and results thereof. Most notable in this sequence of techniques are CASE (computer-aided software engineering), JAD (joint application development), prototyping, 4GL (fourth generation languages), and Pair/Xtreme programming. While these methods have often provided some gains, none have provided the improvements necessary to become that "silver bullet." CASE methods have allowed development organizations to build the wrong system even faster, "wrong" in the sense that requirements are not met and/or the resulting system is not maintainable or adaptable. JAD methods tend to waste more of everyone's time in meetings. While prototypes can help better define user requirements, the tendency (or expectation) that the prototype can be easily extended into the real system is very problematic. The use of 4GL languages only speeds up the development of the parts of the system that were easy to make anyway, while unable to address the more difficult and time consuming portions. Pair programming has some merits but stifles creativity and often requires more time and money.

The only true "solution" has been effective software reuse. Reuse of existing proven components can result in the faster development of software with higher quality. Improved quality results from both the use of previous "tried and true" components and the fact that standards (technical and business) can be built into the reusable components (Brandon, 2000). This improved quality results in lower lifecycle maintenance costs, and since two thirds of software product lifecycle costs are in post-delivery maintenance, this cost savings aspect of reusability is the most rewarding (Schach, 2005). There are several types of reusable components that can address both the design and implementation process. These come in different levels of "granularity" and in both object oriented and non-object oriented flavors.

Software reuse received much attention in the 1980s but did not catch on in a big way until the advent of object oriented languages and tools" (Anthes, 2003). In Charles Darwin's theory of species survival, it was the most adaptable species that would survive (not the smartest, strongest, or fastest). In today's fast moving business and technical world, software must be adaptable to survive and be of continuing benefit. Object oriented software offers a very high degree of adaptability. "Object technology promises a way to deliver cost-effective, high quality and flexible systems on time to the customer" (McClure, 1996). "IS shops that institute component-based software development reduce failure, embrace efficiency and augment the bottom line" (Williamson, 1999). "The bottom line is this: while it takes time for reuse to settle into an organization—and for an organization to settle on reuse—you can add increasing value throughout the process" (Barrett & Schmuller, 1999). We say "object technology" not just adopting an object oriented language (such as C++, Java, or PHP), since one can still build poor, non-object oriented, and non-reusable software, even using a fully object oriented language.

TYPES AND APPLICATIONS OF REUSE

Radding (1998) defines several different types of reusable components, which form a type of "granularity scale":

- **GUI Widgets:** Effective, but only provide modest payback.
- **Server-Side Components:** Provide significant payback but require extensive up-front design and an architectural foundation.
- **Infrastructure Components:** Generic services for transactions, messaging, and database … require extensive design and complex programming.
- **High-Level Patterns:** Identify components with high reuse potential.
- **Packaged Applications:** Only guaranteed reuse—may not offer the exact functionality required. This includes COTS (commercial off the shelf software).

An even lower level of granularity is often defined to include simple text files that may be used in a number of code locations such as "read-me" and documentation files, "help" files, Web content, business rules, XML schemas, test cases, and so forth. Among the most important recent developments of object oriented technologies is the emergence of design patterns and frameworks, which are intended to address the reuse of software design and architectures (Xiaoping, 2003). The reuse of "patterns" can have a higher level of effectiveness over just source code reuse. Current pattern level reuse includes such entities as a J2EE Session Façade or the .Net Model-View-Controller pattern.

Reuse has two types. The first is called opportunistic (or accidental) reuse, where developers realize that a component from a previous project could be used in the current project. The second is systematic (or deliberate) reuse, where components are built to be reused (Schach, 2005). Reusing code also has several key implementation areas: application evolution, multiple implementations, standards, and new applications. The reuse of code from prior applications in new applications has received the most attention. However, just as important is the reuse of code (and the technology embedded therein) within the same application.

Application Evolution

Applications must evolve even before they are completely developed, since the environment under which they operate (business, regulatory, social, political, etc.) changes during the time the software is designed and implemented. This is the traditional "requirements creep." Then after the application is successfully deployed, there is a constant need for change.

Multiple Implementations

Another key need for reusability within the same application is for multiple implementations. The most common need for multiple implementations involves customizations, internationalization, and multiple platform support. Organizations whose software must be utilized globally may have a need to present an interface to customers in the native language and socially acceptable look and feel ("localization"). The multiple platform dimension of reuse today involves an architectural choice in languages and delivery platforms.

Corporate Software Development Standards

Corporate software development standards concern both maintaining standards in all parts of an application and maintaining standards across all applications. "For a computer system to have lasting value it must exist compatibly with users and other systems in an ever-changing information technology (IT) world" (Brandon, 2000). As stated by Weinschenk and Yeo, "Interface designers, project managers, developers, and business units need a common set of look-and-feel guidelines to design and develop by" (Weinschenk & Yeo, 1995). In the area of user interface standards alone, Appendix A of Weinschenk's book presents a list of these standards; there are over 300 items (Weinschenk, Jamar, & Yeo, 1997). Many companies today still rely on some type of printed "Standards Manuals."

EFFECTIVE SOFTWARE REUSE

Only about 15% of any information system serves a truly original purpose; the other 85% could be theoretically reused in future information systems. However, reuse rates over 40% are rare (Schach, 2004). "Programmers have been swapping code for as long as software has existed" (Anthes, 2003). Formal implementation of reuse in various forms of software reuse has been a part of IT since the early refinements to 3GLs (Third Generation Languages). COBOL had the "copy book" concept, where common code could be kept in a separate file and used in multiple programs. Most all modern 3GL's have this same capability, even today's Web-based languages like HTML and JavaScript on the client side, and PHP (on the server side). HTML has "server side includes"; JavaScript has ".js" and ".css" files; and PHP has "require" files (".inc"). Often used in conjunction with these "include" files is the procedure capability where some code is compartmentalized to perform a particular task, and that code can be sent arguments and possibly also return arguments. In different 3GLs this might be called "subroutines"