

On Issues with Component-Based Software Reuse

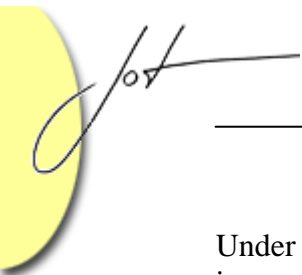
Won Kim, Samsung Electronics, SuWon, Korea

Abstract

As the size and complexity of software products, and the compression of software product development cycle seem ready to go out of control, large software development organizations are taking a hard look at the allure of the substantial productivity enhancements envisioned by early proponents of component-based software reuse. In this article, I will summarize practical difficulties in realizing the benefits of component-based software reuse and discuss key pre-conditions to meet before attempting software reuse on a wide scale in a formalized manner.

1 INTRODUCTION

The notion of software reuse has been around for the past thirty years as a way to drastically reduce the cost of developing software. This includes component-based software design and even service-oriented architecture. The need for software reuse has become urgent as the size and complexity of software have started to escalate steeply and as the product development cycle has been compressed. Enterprise software, such as database systems, ERP systems, CRM systems, SCM systems, data warehousing systems, etc. each consist of millions of lines of code, and require expensive technical consulting for proper use. Even software embedded in consumer electronics products such as cell phones and digital television now consist of millions of lines of code. The competitive pressures have forced vendors to reduce product development cycles from 18-24 months perhaps a decade ago to 3-9 months today. The significant worldwide reduction in students choosing to major in computer science during the past several years simply means a shallow pool of software developers with basic training today and into the foreseeable future. Today the number of software engineers that some of the large global corporations in such countries as Japan and Korea anticipate needing in the next several years is far greater than the total number of students anticipated to graduate with degrees in computer science and related areas. This, not necessarily the presumed cost savings, is the primary reason these companies have been outsourcing software development jobs to countries such as India, China, Russia, etc.



Under these circumstances the idea of stockpiling a large number of software components in a software component database and simply selecting some of them and dropping them into new software being developed is enticing indeed. However, this is a rather naïve notion that flies in the face of harsh realities of software development, and as such large software development organizations must exercise extreme caution before pursuing organization-wide efforts to create a large database of software components for reuse. In the rest of this article, I will examine various impediments to component-based software reuse, and also outline areas that require standardization and trained discipline as pre-conditions to component-based software reuse.

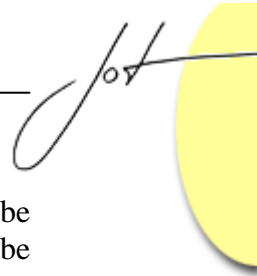
2 DIFFICULTIES WITH COMPONENT-BASED SOFTWARE REUSE

Broadly, there are two types of component-based reuse: with no change to an existing component, and with change. Let us examine reuse without change first. Reuse without change means simply selecting a component from a software component database, and dropping it into new software being developed. The cost of developing the component anew is zero! In fact, there are at least three types of component-based reuse in wide use. One is reuse of most of existing software when developing the next version of the software. Typically, some 60-80 percent of the existing software gets to be reused in this situation. However, developers do not go through the formality of “registering” components in a common software component database in this case. Another is “reuse” of thirty-party software, such as a sorting package, a database loader, etc. on the market or on the Internet as open source code. Again, such software is not “registered” in an organization’s common software component database. A third type of reuse is common functions available in programming language libraries, such as the math functions in the C Programming Library.

There are a large variety of technical reasons that make reusing existing components without change not exactly a picnic in the park. Let us examine them. (I do not believe the following list is exhaustive, but the types and number of reasons should make the point clear.)

1. functional differences

This is possibly the most serious reason an existing component cannot be reused without change. It is rarely the case that an existing component and a component to be newly developed match precisely in functions. The new component to be developed may require some changes to corresponding functions in the existing component, or may require additional functions. The existing component may be excessively rich in functions and, for, say, performance reasons, most of the excess functions may need to be eliminated. It is entirely possible, especially in this age of digital convergence where such devices as cell phones and MP3 players are continually taking on additional capabilities, software developed to provide one particular major capability for one product line may be reused



in its entirety for another product line; for example, software for digital cameras may be reused on cell phones. However, it is unlikely that smaller granule components may be reused without change across product lines (i.e., business units).

2. programming language difference

If it has been decided that new software must be developed in, say, Java, an existing component written in, say C, cannot be reused without change, even if it satisfies all other requirements, such as the functionality, operating environment, system architecture, etc. Although cross compilation may sometimes allow reuse of components, in general it does not apply.

3. target environment differences

Target environments include platforms (chipset for embedded systems, and operating system), computer system architecture (single CPU, parallel computer, hub and spoke distributed architecture, peer to peer distributed architecture, networking protocol, etc.), computing environment (disk, CD ROM, USB, flash memory, etc.) and operating constraints (main memory size, secondary storage size, power consumption, screen size, paper size for a printer, etc.). Software components developed for one particular target environment often cannot be reused without change for another target environment.

4. operating environment differences

Operating environments include the number of simultaneous users, read-only or read-write, volumes of data to manage, data input and output rate, etc. Software developed for a single user cannot in general be reused to support a multiple simultaneous-user environment. Software designed to only read data cannot in general be reused to support a read-write environment. Software designed for slow input and output of data cannot in general be reused to support an environment where the input and output rate is very high.

5. industry standards differences

There are many industry-wide standards on a very wide range of aspects of computing and communications. Examples include Web document standards (HTML, XML, etc.), communications standards (CDMA, GSM, 3G, Bluetooth, WiFi, etc.), cable television standards (OCAP, ACAP), connectivity standards (DLNA, Marlin, OSGi, UPnP, etc.), multimedia data encoding standards (MPEG, JPEG, HDD vs. BluRay, etc.), national language encoding standards (Unicode, etc.), metadata management standards (MOF, XMI, CWM, Dublin Core), external database access standards (ODBC, JDBC), etc. Software components that explicitly reflect one standard cannot be reused in software supporting another standard.

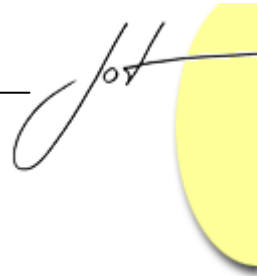
6. data format differences

A large volume of data is managed either by a file system or a database management system. Software components that interface with file systems are significantly different from those that interface with database management systems. Different countries use different formats to store such data as date, time, currency, etc., and use different measurement units. Some software encode data in ASCII, while others use EBCDIC. Software components that explicitly deal with such data formats and encoding schemes cannot be reused without change in software that use different formats or encoding scheme.

7. algorithm and data structure differences

A wide variety of algorithms implement key functions or capabilities supported in software systems. Examples include sorting, searching, data replication, database locking, database logging for recovery, security and encryption, message routing on a network, etc. For each such function or capability, typically there is more than one algorithm or technique, and accompanying data structures (e.g., linked list, hashing, binary tree, B+-tree, R*-tree, heap), to implement it, each with different tradeoff considerations. One technique may be simple and quick to implement, while it may result in low performance or low level of reliability or low level of security. One technique may be good to support the management of a small amount of data, while it may be totally inappropriate for a large volume of data or a very high input data rate. One technique may be good to support the routing of a large number of small messages, while it may be unacceptable for routing a mixture of a small number of large messages and a large number of small messages. Software components that implement algorithms and techniques, and accompanying data structures, under certain tradeoff considerations cannot in general be reused without change under different tradeoff considerations.

The difficulty with reuse with change to an existing component is obvious. It is difficult and time-consuming to fully understand an existing component that appears to closely match the requirements of the part of the new software for which the component may be reused with change. Then it takes efforts to determine those parts of the component that require changes, and to actually make those changes. Of course, once the changes are made, the modified component must be thoroughly tested and documented, and the entire software that includes the modified component must in turn be thoroughly tested. Faced with these practical difficulties, often software developers conclude that it would take them less time and manpower to develop the necessary parts of the new software from scratch than to select some seemingly closely matching components, analyze and modify them.



3 PRE-CONDITIONS TO COMPONENT-BASED REUSE

Formal reuse of components, with or without change, is not cost-free. First of all, developers must register the components in a software component database. There must first be a standard format for registering software components. Simply storing a directory of source code files is not enough. The source code must be properly documented, with block comments and inline comments. Requirements specification and design specification concerning the component to be reused should also be included. Before a component may be selected for reuse, it must be confirmed that the component closely matches the requirements of the part of the new software into which it is to be used. Even if the component comes with proper documentation, this task is difficult and time-consuming. Proper documentation is essential to make the task more accurate and less time-consuming. Further, the test suite for testing the code should be included. It must be made easy for other developers to confirm that the component selected will work properly in the possibly new environment of the new software being developed. Then the developers (or QA/test engineers) must develop test cases to confirm that the component in the context of the new software that now includes it behaves as expected.

Beyond these, programming guidelines, software development best practices, and documentation standards should be established organization-wide, and developers must follow them. Documentation standards include templates and samples for requirements specification, design specification, source code block comments, and software release notes. Developers and QA/testers must also be properly trained on all key architectural factors in the development of software (including software in embedded systems context), including performance, scalability, reliability, extensibility, security, and configurability.

4 CONCLUDING REMARKS

Component-based reuse should really be taken as one of several ways in which to enhance software development productivity. It is certainly not the only way to enhance productivity, and not necessarily even the best way. Other, more direct and immediate ways to enhance productivity is the use of a variety of software tools, including tools for analyzing and testing software, tools for automatically generating unit test cases, and simulators for automating the testing of embedded systems.

There are various pre-conditions that must be met before component-based software reuse should be attempted organization-wide and in a formal manner. Pre-conditions include proper training of all software developers and QA/testers on all facets of software development, including documentation, thorough testing, programming guidelines and programming and testing best practices, software design based on full consideration of all key architectural factors in the development of software. Of course, all developers and

QA/testers must follow all guidelines established and all best practices they are trained on.

About the Author



Won Kim is Senior Advisor at Samsung Electronics, Korea. He is Editor-in-Chief of ACM Transactions on Internet Technology (www.acm.org/toit). He is Global General Chair of the [Human.Society@Internet](#) International Conference. He is the recipient of the ACM 2001 Distinguished Services Award, and is an ACM Fellow.