Software Reuse Myths Revisited

Will Tracz
Loral Federal Systems Company
Owego, NY
tracz@vnet.ibm.com

It has been six years since the author published the paper *Software Reuse Myths* in ACM Software Engineering Notices [tracz87h]. This short paper comments on these "myths" in light of recent technology advances.

Myth #1: Software Reuse is a Technical Problem

Three major non-technical impediments to reuse have diminished somewhat over the last six years. There has been some consensus building efforts focused on coming up with standard definitions for reuse and reusability. The RIG (Reuse Interoperability Group), for example, is attempting to come up with standards for reusable components and their interchange through common library access methods. Furthermore, recent efforts within the DoD to change government acquisition policies will further stimulate the industry. Finally, with copyright law now being re-interpreted by the courts to allow for more flexibility in developing plug compatible solutions (that may have some common design structure but no common code), the software marketplace is poised to better be served by competitive COTS (Commercial Off-The-Shelf) offerings for reuse.

On the technical side, research in process programming, persistent object bases, and knowledge representation and manipulation show potential for facilitating the creation of fertile reuse environments. The success of ARPA's Domain-Specific Software Architecture program also provides valuable credibility to reuse in general.

Myth #2: Special Tools are Needed for Software Reuse

Integrated CASE tools, when they get here, have the potential to enhance software reuse. While they are not necessary for reuse, they do go a long ways toward making reuse fun. They will facilitate the traceability of requirements to design and code, along with the other artifacts associated with software development. Reusable software can be thought of as a CASE tool in itself in that reusable software components are just another tool that should be leveraged in solving problems.

Myth #3: Reusing Code Results in Huge Increases in Productivity

There has been a lack of good empirical data related to reuse success stories. There really should be more data out in the literature, but there isn't. While it is true that in some instances, reuse has been shown to result in a 20 to 1 reduction in effort for "certain" stages in the development life cycle, this should be placed in perspective with the "cost" of reuse.

There are three costs associated with reuse: the cost of making something reusable, the cost of reusing it, and the cost of defining and implementing a reuse process. Focusing on the cost of making software reusable, a conservative breakdown is a follows:

25%	for additional generalization
15%	for additional documentation
10%	for additional testing
5%	for library support and maintenance
60%	additional cost of making something reusable

The subtle/sad thing is, reusability is quite subjective. Who can say that spending x% on documentation will make it y% more reusable, or an additional z% spent generalizing will make it q% more reusable? Obviously, you don't just "spend the money", you focus it on making certain attributes of the software "better".

Myth #4: Artificial Intelligence will Solve the Reuse Problem

I have always been skeptical of "imitation intelligence." Recently, through my involvement in the ARPA DSSA community, I have become convinced that

0270-5257/94 $3.00 © 1994 IEEE

software reuse is the common ground where AI and software engineering will meet. It is becoming more and more apparent that one needs to reuse more than just code to gain increased benefits from reuse. AI, with its knowledge acquisition and representation experience, has a lot to offer to support this kind of reuse. In fact, machine learning and knowledge acquisition, closely resemble application generators and domain analysis -- both fundamental to software reuse.

Myth #5: The Japanese Have Solved the Reuse Problem

The Japanese continue to do good things by taking an evolutionary rather than revolutionary approach. They have recognized the non-technical inhibitors to software reuse and have invested in creating the assets and processes to support reuse. As I have previously stated "Before you can reuse software, you need software to reuse." They have addressed this issue head on, though they haven't declared victory yet.

Myth #6: Ada has Solved the Reuse Problem

Ada has not, nor will not go away. Ada9X has made claims to have increased support for reusability in the form of an anemic inheritance mechanism and object types (but it still lends itself for improvement as well as abuse). As far as a language that promotes reuse goes, C++, as baroque as it is, has almost become the defacto standard, but again, it is missing some parameterization capabilities that would enhance its applicability.

Myth #7: Designing Software from Reusable Parts is like Designing Hardware using Integrated Circuits

I have wavered on the position I took on this point five years ago, as the analogy of component-based software design and component-based hardware design is conveniently seductive. (Yes Brad Cox, I hate to admit you may be right with your software-IC analogy.) The big "if" that makes me want to agree with the analogy is "if software interfaces were defined as crisply as hardware interfaces" then I would believe the myth is true, especially in light of all the object-oriented religious fervor that has been so popular of late. But, of course, there are no catalogs of components, outside of user interface widgets. The reason may still lay in the immaturity of the science, or in that we do not understand the complexity of integration. Said another way, we don't understand the types of glue/solder that are needed to connect the components together with, nor do we have standard bus protocols to let us hook up larger components.

Myth #8: Reused Software is the Same as Reusable software

This point needs no discussion. With several reuse standards for "designing software for reuse" available, it is testimony that reusability, like quality, is an attribute of software that must be planned for. It is refreshing to see that the government is finally recognizing the difference between ``unplanned/opportunistic" reuse, which it (and others in industry call ``salvaging") and "planned" reuse.

Myth #9: Software Reuse Will Just Happen

In five years time, reuse has been given a lot of press, but it really has not blossomed. I am encouraged by the DoD initiatives (e.g., STARS, ARPA's DSSA, and the DISA CIM effort). I am also pleased by the progress HP and IBM have made at institutionalizing software reuse as part of the corporate programming strategy.

The bottom line is that software reuse is maturing. It has learned to crawl and in the next 5 years, may even walk upright without dragging its knuckles.

References

[Tracz87h] W.J. Tracz, Software Reuse Myths. ACM Software Engineering Notes, volume 13, Number1, January, 1988, Pages 17-21.