Facts and Fallacies of Software Engineering

Robert L. Glass

Foreword by Alan M. Davis

Facts and Fallacies of Software Engineering

➥ Reifer, Donald J. 1997. *Practical Software Reuse.* New York: John Wiley and Sons.

➥ Tracz, Will. 1995. *Confessions of a Used Program Salesman: Institutionalizing Reuse.* Reading, MA: Addison-Wesley.

Fact 17 **Reuse-in-the-large works best in families of related systems and thus is domain-dependent. This narrows the potential applicability of reuse-in-the-large.**

Discussion

OK, so reuse-in-the-large is a difficult, if not intractable, problem. Is there any way in which we can increase the odds of making it work?

The answer is "yes." It may be nearly impossible to find components of consequence that can be reused across application domains, but within a domain, the picture improves dramatically. The SEL experience in building software for the flight dynamics domain is a particularly encouraging example.

Software people speak of "families" of applications and "product lines" and "family-specific architectures." Those are the people who are realistic enough to believe that reuse-in-the-large, if it is ever to succeed, must be done in a collection of programs that attacks the same kinds of problems. Payroll programs, perhaps even human resource programs. Data reduction programs for radar data. Inventory control programs. Trajectory programs for space missions. Notice the number of adjectives that it takes to specify a meaningful domain, one for which reuse-in-the-large might work.

Reuse-in-the-large, when applied to a narrowly defined application domain, has a good chance of being successful. Cross-project and cross-domain reuse, on the other hand, does not (McBreen 2002).

Controversy

The controversy surrounding this particular fact is among people who don't want to give up on the notion of fully generalized reuse-in-the-large. Some of those people are vendors selling reuse-in-the-large support products. Others are academics who understand very little about application domains and want to believe that domain-specific approaches aren't necessary. There is a philosophical connection between these latter people and the one-size-fits-all tools and methodologists. They would like to believe that the construction of software is the same no matter what domain is being addressed. And they are wrong.

Sources

The genre of books on software product families and product architectures is growing rapidly. This is, in another words, a fact that many are just beginning to grasp, and a bandwagon of supporters of the fact is now being established. A couple of very recent books that address this topic in a domain-focused way are

- Bosch, Jan. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Boston: Addison-Wesley.
- Jazayeri, Mehdi, Alexander Ran, and Frank van der Linden. 2000. *Software Architecture for Product Families: Principles and Practice.* Boston: Addison-Wesley.

Reference

- McBreen, Pete. 2002. *Software Craftsmanship.* Boston: Addison-Wesley. Says "cross-project reuse is very hard to achieve."

Fact 18

There are two "rules of three" in reuse: (a) It is three times as difficult to build reusable components as single use components, and (b) a reusable component should be tried out in three different applications before it will be sufficiently general to accept into a reuse library.

Discussion

There is nothing magic about the number three in reuse circles. In the two rules of three, those threes are rules of thumb, nothing more. But they are nice, memorable, realistic rules of thumb.

The first is about the effort needed to build reusable components. As we have seen, to construct reusable components is a complex task. Often, someone building a reusable component is thinking of a particular problem to be solved and trying to determine whether there is some more general problem analogous to this specific one. A reusable component, of course, must solve this more general problem in such a way that it solves the specific one as well.

Not only must the component itself be generalized, but the testing approach for the component must address the generalized problem. Thus the complexity of building a reusable component arises in the requirements—"what is the generalized problem?"—design, "how can I solve this generalized problem? in coding, and in testing portions of the life cycle. In other words, from start to finish.

It is no wonder that knowledgeable reuse experts say it takes three times as long. It is also worth pointing out that, although most people are capable of thinking about problems in a generalized way, it still requires a different mindset from simply solving the problem at hand. Many advocate the use of particularly skilled, expert generalizers.

The second rule of thumb is about being sure that your reusable component really is generalized. It is not enough to show that it solves your problem at hand. It must solve some related problems, problems that may not have been so clearly in mind when the component was being developed. Once again, the number three—try your component out in three different settings—is arbitrary. My guess is that it represents a minimum constraint. That is, I would recommend trying out your generalized component in *at least* three different applications before concluding that it truly is generalized.

Controversy

This fact represents a couple of rules of thumb, rules that few have reason to doubt. Everyone would acknowledge that reusable components are harder to develop and require more verification than their single-task brethren. The numbers three might be argued by some, but there is hardly anyone who is likely to defend them to the death, since they are rules of thumb and nothing more.

Sources

This fact has come to be known over the years as "Biggerstaff's Rules of Three." There is a very early paper by Ted Biggerstaff, published in the 1960s or 1970s, that first mentions reuse rules of three. Unfortunately, the passage of time has eroded by ability to recall the specific reference, and my many attempts to use the Internet to overcome that memory loss have not helped. However, in the References section, I mention studies of Biggerstaff's role.

I have a particular reason for remembering the rules of thumb and Biggerstaff, however. At the time Biggerstaff's material was published, I was working on a generalized report generator program for business applications (I mentioned it earlier in passing). I had been given three report generators (for very specific tasks) to program, and—since I had never written a report generator program before—I gave more than the usual amount of thought to the problem.

The development of the first of the three generators went quite slowly, as I thought about all the problems that, to me, were unique. Summing up columns of figures. Summing sums. Summing sums of sums. There were some interesting

problems, very different from the scientific domain that I was accustomed to, to be solved here.

The second program didn't go much faster. The reason was that I was beginning to realize how much these three programs were going to have in common, and it had occurred to me that a generalized solution might even work.

The third program went quite smoothly. The generalized approaches that had evolved in addressing the second problem (while remembering the first) worked nicely. Not only was the result of the third programming effort the third required report generator, but it also resulted in a general-purpose report generator. (I called it JARGON. The origin of the acronym is embarrassing and slightly complicated, but forgive me while I explain it. The company for which I worked at the time was Aerojet. The homegrown operating system we used there was called Nimble. And JARGON stood for Jeneralized (ouch!) Aerojet Report Generator on Nimble.)

Now, I had already formed the opinion that thinking through all three specific projects had been necessary to evolve the generalized solution. In fact, I had formed the opinion that the only reasonable way to create a generalized problem solution was to create three solutions to specific versions of that problem. And along came Biggerstaff's paper. You can see why I have remembered it all these years.

Unfortunately, I can't verify the first rule of three, the one about it taking three times as long. But I am absolutely certain that, in creating JARGON, it took me considerably longer than producing one very specific report generator. I find the number three quite credible in this context, also.

➥ Biggerstaff, Ted, and Alan J. Perlis, eds. 1989. *Software Reusability.* New York: ACM Press.

➥ Tracz, Will. 1995. *Confessions of a Used Program Salesman: Institutionalizing Reuse.* Reading, MA: Addison-Wesley.

Fact 19

Modification of reused code is particularly error-prone. If more than 20 to 25 percent of a component is to be revised, it is more efficient and effective to rewrite it from scratch.

Discussion

So reuse-in-the-large is very difficult (if not impossible), except for families of applications, primarily because of the diversity of the problems solved by software. So why not just change the notion of reuse-in-the-large a little bit? Instead of

reusing components as is, why not modify them to fit the problem at hand? Then, with appropriate modifications, we could get those components to work anywhere, even in totally unrelated families of applications.

As it turns out, that idea is a dead end also. Because of the complexity involved in building and maintaining significant software systems (we will return to this concept in future facts), modifying existing software can be quite difficult. Typically, a software system is built to a certain design envelope (the framework that enables but at the same time bounds the chosen solution) and with a design philosophy (different people will often choose very different approaches to building the same software solution). Unless the person trying to modify a piece of software understands that envelope and accepts that philosophy, it will be very difficult to complete a modification successfully.

Furthermore, often a design envelope fits the problem at hand very nicely but may completely constrain solving any problem not accommodated within the envelope, such as the one required to make a component reusable across domains. (Note that this is a problem inherent in the Extreme Programming approach, which opts for early and simple design solutions, making subsequent modification to fit an enhancement to the original solution potentially very difficult.)

There is another problem underlying the difficulties of modifying existing software. Those who have studied the tasks of software maintenance find that there is one task whose difficulties overwhelm all the other tasks of modifying software. That task is "comprehending the existing solution." It is a well-known phenomenon in software that even the programmer who originally built the solution may find it difficult to modify some months later.

To solve those problems, software people have invented the notion of maintenance documentation— documentation that describes how a program works and why it works that way. Often such documentation starts with the original software design document and builds on that. But here we run into another software phenomenon. Although everyone accepts the need for maintenance documentation, its creation is usually the first piece of baggage thrown overboard when a software project gets in cost or schedule trouble. As a result, the number of software systems with adequate maintenance documentation is nearly nil.

To make matters worse, during maintenance itself, as the software is modified (and modification is the dominant activity of the software field, as we see in Fact 42), whatever maintenance documentation exists is probably not modified to match. The result is that there may or may not be any maintenance documentation, but if there is, it is quite likely out-of-date and therefore unreliable. Given all of that, most software maintenance is done from reading the code.

And there we are back to square one. It is difficult to modify software. Things that might help are seldom employed or are employed improperly. And the reason for the lack of such support is often our old enemies, schedule and cost pressure. There is a Catch-22 here, and until we find another way of managing software projects, this collection of dilemmas is unlikely to change.

There is a corollary to this particular fact about revising software components:

It is almost always a mistake to modify packaged, vendor-produced software systems.

It is a mistake because such modification is quite difficult; that's what we have just finished discussing. But it is a mistake for another reason. With vendor-supplied software, there are typically rereleases of the product, wherein the vendor solves old problems, adds new functionality, or both. Usually, it is desirable for customers to employ such new releases (in fact, vendors often stop maintaining old releases after some period of time, at which point users may have no choice but to upgrade to a new release).

The problem with in-house package modifications is that they must be redone with every such new release. And if the vendor changes the solution approach sufficiently, the old modification may have to be redesigned totally to fit into the new version. Thus modifying packaged software is a never-ending proposition, one that continues to cost each time a new version is used. In addition to the unpleasant financial costs of doing that, there is probably no task that software people hate more than making the same old modification to a piece of software over and over again. Morale costs join dollar costs as the primary reason for accepting this corollary as fact.

There is nothing new about this corollary. I can remember back to the 1960s when, considering how to solve a particular problem, rejecting modifying vendor software on the grounds that it would be, long-term, the most disastrous solution approach. Unfortunately, as with many of the other frequently forgotten facts discussed in this book, we seem to have to keep learning that lesson over and over again.

In some research I did on the maintenance of Enterprise Resource Planning (ERP) systems (for example, SAPs), several users said that they had modified the ERP software in-house, only to back out of those changes when they realized to their horror what they had signed up for.

Note that this same problem has interesting ramifications for the open-source software movement. It is easy to access open-source code to modify it, but the wisdom of doing so is clearly questionable, unless the once-modified version