Thinking about "reuse"

Call me a heretic if you like, but it seems to me that “reuse” – that Holy Grail of software coders – seems to be, well, not to put to fine a point on it, a false idol. I’m not alone in this, Kent Beck and the Extreme Programming crowd have been denying reuse for several years in their efforts to “do the simplest thing that works” and Kevlin Henney[1] has questioned some of our assumptions too.

What is happening here? Well for me the serious doubts set in last year when reviewing Generative Programming by Czarnecki and Eisenecker[2], for me, their cure where worse than the disease!

Further, times are changing and so are the economics of software. We are no longer confined to our traditional models of “reuse.” The development of web services means there are new models appearing. Got a good library of chemical simulators? Why not SOAP enable it and change per use from your web server? And why not experiment with new ways of funding development? Chris Rasch has suggested using the bond market to fund software development[3].

How do we get here?

The appeal of “reuse” is obvious – especially if you have to pick up the bill for software development. If we could just reuse what we have already done....

This was one of the early promises of object-orientation. The idea was that objects represented some kind of entity, which would be applicable in more than one program. As Czarnecki and Eisenecker point out, the idea that reusable objects would simply “drop out” of a system development proved false. It transpires that producing an object usable in more than one system is actually much more difficult, in some cases more than double the effort. Czarnecki and Eisenecker introduce the idea of “designing for reuse” and “designing with reuse”. (Now I agree with them on everything up to now. Beyond this, my problem with *Generative Programming* is that I don’t see how their solutions can be used in a real world environment. Much of the code they produce comes dangerously close to being unmaintainable, but I digress.)

This division seems to recall ideas of the “problem domain” and “solution domain”: in analysing and problem domain we would identify and create objects which could be reused, when working in the solution domain we would take these prefabricated modules to create out solution.

It appears that finding the right granularity for “reuse” is somewhat difficult: function level reuse gave way to object reuse and object reuse has given way to component reuse. (The word module is so malleable that I’m deliberately avoiding it here.)

All the time “reuse” seems almost within our grasp. Truth is, as an industry we reuse more code than ever before. We make extensive use of code libraries, code-generating wizards (which are basically a form of meta programming), cut-and-paste’ed code and packaged applications with macro languages or automation interfaces. So maybe we are not giving ourselves credit where credit is due.

So, why do I say reuse is a false idol?

There are two things wrong with “reuse” as we generally talk about it. First, I think we need to look at the costs of reuse. Second, “reuse” is vague a term, not only does it depend on who is using it but it covers a multitude of ideas.

The cost of reuse

Sometimes costs are obvious: Joe will be tasked with creating a reusable library of widgets, so the cost is Joe’s salary for the six months of development. More often these costs are hidden. As a developer you will decide to make your object slightly more generalised than it needs to be, or you will allow for some obvious enhancement which is not yet required. In effect you are doing more work than you need to, so the cost is say 20% of your salary.

In either case you should have some idea of when the payback will come - a few weeks? months? Are you sure the payback will come at all? Even if it does come who will see the benefit?

In addition to the financial cost there is the cost of work forgone. If a less generalised object had been produced in half the time you would be free to move onto the next work item. Economists call this “opportunity cost” it is the cost of not doing something else.

Building more generalised code frequently (but not always) leads to more complex code. By its nature this is more prone to errors and more difficult to maintain.

Further, can we be sure that the additional functionality, added but not yet used, is tested as well as the functionality that is to be used? If I create a Widget class with three methods, and then I decide to add an extra three methods (which I’m sure we are going to need in future) do I test the second three as completely as the first three?

There are other issues with this scenario:

? If before we get to use the three extra methods we change the class do we need to change them?

? If Fred takes over the class from me won’t he be wondering what is going on?

? If we need to port the class, or rewrite it, or refactor it, we have twice as many methods to work with than we actually need.

If these methods aren’t used we have wasted our time writing them. Even if they are used, between creating them and using them we have to support them, they represent the “cost of carry” – to borrow another term from economists.

Finally, consider those three methods again. When they are finally used they aren’t even being “reused” – they are being used for the first time! Sure, the Widget object is being “reused” but are we putting the cart before the horse here?

What does “reuse” actually mean?

The next problem with “reuse” is the very word itself. In part this comes from your perspective. If I create my Widget class for program Foo, and I use it again in program Bar then I have *reused* it because I have *used* it before. But if you where to write program Bar are you *reusing* Widget or just *using* it?

Adding “re” to the front of the word “use” doesn’t actually buy us very much. When you got on the number 65 bus this morning, where you using the bus to get to work? Or where you reusing the bus to get to work?

OK, maybe I’m playing semantic games here but when we talk about code “reuse” we cover a lot of ground:

? using a third party library, maybe even the STL

? using code salvaged from the system we are replacing

? using code from a different project within the company

? developing two programs in parallel and share some code

This list could go one but I hope you get my point. What is *use*? And what is *reuse*? The term “reuse” is not just overloaded but is actually pretty devoid of any real meaning!

So where does this leave me?

So far I’ve pointed out that “reuse” is a “good thing”. I’ve also claimed that much of where we do in the name of "“reuse” is wasteful. And finally, so far, I’ve attacked the word itself. So, where am I going with this?

“Reuse” has become too generalised, too much of a “good thing”. I think we need some new words, some new terms, new ideas to describe what we are actually talking about here.

OK, so we will end up sounding like management consultants, but I don’t think one word, can really cover all the things we use it for. (And maybe, just maybe, we will see our salaries get a bit closer to those of management consultants!)

What else can I say?

Now what I’m about to suggest may come across as jargon-making, for which I apologise. However, jargon can serve a useful purpose: to specifically identify an idea or concept. So here goes, here are some terms and ideas.

Commodity

This is a word I like. I was already playing around with it in this context before I saw Kevlin Henney’s use of the word[4]. If we are developing an object, a module, a component, or whatever, which we are intending to use in several places commodity fits: we are developing a piece of software which can but used in various ways.

Commodity also has nice over tones of trading, we buy and sell commodities. This highlights the way commercial nature of most software development.

Another benefit of *commodity* is that it implies certain properties, interfaces and standards – some commonality. A television is a commodity, we can be sure that in the USA it will use one standard, and in England another – inconvenient yes, but a known issue. We can develop DVD players to work with this interface for the TV, we don’t need specialised sets.

"Product" has similar commercial overtones to commodity but must be rejected for our purposes. The word carries too much other baggage. It implies a complete, or completed, item. Commodity is much more granular.

Technology Transfer

Although this term smells a lot like management-consultant-speak I think it is much more actuate than "reuse." It we clearly stating what we are talking about and is free of the baggage that "reuse" carries. As the side bar outlines the term is already used elsewhere and it can be expanded to describe the temporal characteristics of the transfer.

There are two distinct forms software transfer that can be identified:

? Horizontal transfer: this is the sharing of common commodities between diverse projects, e.g. when we use a strings library, or a threading library we are transferring technology horizontally. The projects may have little business commonality but they have a great deal of raw technology in common.

? Vertical transfer: is the transfer of technology between similar products, typically within a family of software. For example, a bank that uses the same evaluation model for equities options, foreign exchange trading and risk management is transferring technology vertically.

Vertical transfer usually relates to the application domain so it is more common within organisations, while horizontal transfer, which frequently relates to the system domain, is more common between organisations. Program families, as discussed by Parnas[5], are akin to vertical technology transfer.

If we look at most of well known libraries available to developer (e.g. Boost, Rogue Wave Source Pro, Microsoft MFC, etc.) these are horizontal libraries. They deal well known themes (threads, database connectivy, GUIs, etc.) which developers are actually rather good at solving because we deal with the same problems all the time.

A vertical library inhabits a niche, as such it is less well known. For example NAG's libraries are less well known than Rogue Wave because fewer people need advanced mathematical algorithms. Even so, more developers know about NAG's libraries than know about, say, libraries for computing fuel consumption in cars because many developers have a mathematical background.

Leverage

Love it or loathe it, this seems to be a word you can't ignore any more. To my mind it always conjures up visions of very big levers forcing squares into round holes. There is something forced about the word, which doesn't fit into a smooth running organisation.

The main danger with this word is that like "reuse" it means different things to different people at different times, for this reason alone I don't think it is useful.

Integration

Integration is the process of embedding the transferred technology into another development. We may want to distinguish between a donor project (where the technology comes from) and a recipient project (where the technology is used.)

Intellectual Property - "IP"

IP is the stuff that technology businesses are based on. Increasingly this is becoming a commodity as patent databases are set up and companies are encouraged to place their property on line for others to buy. In the hardware world this is established practice – think of the way ARM has built an entire business on the licensing of it's CPU technology.

Software can travel the same road. Our intellectual property is expressed in terms of source code, UML diagrams and documentation – although a significant part is still locked up inside developers heads.

When we use a third party library we are not just avoiding the need to write some code ourselves – we are actively seeking to use others work. For example, rather than try and understand the mathematics behind optimisation theory I just go and buy the work NAG have done, I use their intellectual property to save myself time in development, testing and verification.

A word of warning though: some people have taken a dislike to the idea of IP expressed in software. The problem is not with the idea of IP itself but with some of the patents granted, and enforced, which are giving it a bad name.

Publishing

Once we start to think of source code as a commodity with the ideas and work contained therein as intellectual property it becomes natural to think of publishing the work. Once our IP is expressed in a commodity, which may be bought and sold, we need to publish it.

Publishing may be external, we may place an advert in Dr Dobbs and request payments, or it may be internal, we may announce to our company that we now have a library that performs some business function, or may be just publishing at the level of our development group.

Commissioning

To continue the publishing metaphor we may like to think of the initial request as a commission. One could imagine a large company, say a bank, identifying vertical application families with common features, e.g. credit appraisal, and commissioning one development group to develop a commodity solution which other groups could also use.

Packaging

Once we have our intellectual property, we must ensure it is packaged for delivery as a commodity for publication. We may choose to package it as source code, a shared library, or a service accessible over the internet. How we package it will in part be determined by our revenue model and by how we wish to divide the market.

For example, look at the way Cola-Cola subdivides its market. Although it is nominally selling the same product (sugared water with flavouring) the pricing depends on the packaging. A 330ml can (sold to people on the move) costs more litre-for-litre than the 2 litre bottle (sold in the supermarket to families at home) which in turn costs much more, litre-for-litre, than the cola concentrate sold to restaurants for sale with food – and where it is mixed with water and marked up.

We may make our IP available as a SOAP enabled web service and charge per use. We may repackage this as a DLL and sell it for a fix price to a large corporation. Finally, we may enter into revenue sharing deals with other producers who compile our commodity into their own.

Conclusion

That several people in the software industry are questioning "The Holy Grail of Reuse" is probably a sign of a maturing industry. "Reuse" is a worthy goal, and has its uses, but it should not be all consuming. We must understand its costs and implications.

"Reuse" will probably remain a general, all embracing, term, it is too entrenched to go away, but we may be better served by coming up with new labels for some of the ideas it covers. I'm not saying my labels are the right labels or that they will stick but I'm starting the debate.

Nor am I claiming that we can simply replace the word "reuse". We may try to do a mental "global replace 'reuse' with 'technology transfer' " but this wouldn't change the real situation. We need to change the way we think about our code and the way we use it.

Even without our own debate on what constitutes software "reuse" the economics of software are changing. For many applications we can now assume an Internet link which enables us to think about pay-as-you-go and leasing models.

---

[1] Kevlin Henney, "minimalism: the imperial clothing crisis", Overload 47, February 2002

[2] Czarnecki & Eisenecker, Generative Programming, Addison-Wesley, 2000

[3] Chris Rasch, "The Wall Street Performer Protocol : using software completion bonds to fund open source development" http://www.firstmonday.dk/issues/issue6_6/rasch/index.html

[4] Overload 47

[5] David Parnas, "On the design and development of Program Families" (1976); reprinted in "Software Fundamentals: collected papers by David L. Parnas", edited by Hoffman and Weiss, Addison-Wesley 2001