contents

credits & contacts

Editor: John Merrells
merrells@acm.org
241 Heartwood Lane,
Mountain View,
CA 94041-11836,
U.S.A

Readers:
Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Mike Woolley
mike@bulsara.com

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
Thaddaeus.frogley@creaturelabs.com

Membership and subscription enquires:
David Hodge
membership@accu.org
31 Egerton Road
Bexhill-on-Sea, East Sussex
TN39 3HJ, UK

Advertising:
Peter Goodliffe
ads@accu.org
4 Malvern Road
Cherry Hinton
Cambridge CB1 9LD, UK
01223 518579

Website: http://www.accu.org/

editorial Product Definition

The key to successful product definition is dissatisfied customers, for customers only complain about problems they really care about.

In defining the features of my current project I have tried to reduce them to a minimally useful set, leaving just enough substance to get the customer interested. The first version may make no sales, but if it draws some complaints then I will be happy, as every complaint is an opportunity to satisfy a need.

Product Requirements Document

The product requirements document, PRD, is central to the product definition activity. It lists the features that the product is to provide. It specifically does not prescribe any particular design or implementation strategy.

There are three sections to a PRD: the general goals of the project, the itemized feature list, and a detailed feature list offering evidence of the feature's worthiness.

The goals should be broad, and most features should fit within the category of a goal. Perhaps the goals of a PRD for a 2.0 product would be:

1. Quality: The 1.0 product was well received, but its quality was perceived to be low.
2. Performance: Key areas of the product do not perform well enough.
3. Ease of Use: Administration tools are too hard to use.

The feature list briefly describes each feature and orders them by priority. Each list item includes a snappy name for easy reference, a brief descriptive paragraph, and a list of any dependent features. Priority is denoted by order. Each feature on the list is deemed more important than the subsequent features on the list. This prioritization exercise can involve much soul searching, but pays off greatly towards the end of the project.

The detailed feature list includes a more expansive entry for each feature providing a complete description of the feature, and as much marketplace evidence as could be collected. The marketplace validates the worth of the feature, and so influences its priority. Evidence can be drawn from interviews with customers and salespeople, support and professional services staff, and also from industry research reports, competitor information, and plain gut feeling.

The framework provided by the PRD is designed to guide the project definition process to ensure that the most return will be realized from the resources being put into the project. Thinking of features is easy. Validating them against a marketplace is hard work.

PRD for an Established Product

A product manager usually owns the PRD, and collects and validates contributions from many sources. Engineering is consulted often to ensure that each feature definition is well understood.

Interpreting customer input can be hard work, as customers are mostly interested in what they want right now and not what they would like to have a year from now.

It is important to discover what they need, not what they say they want. A common mistake is to present customers with a questionnaire based on everything that the engineering team can think of adding to the product. Question: 'Dolby and Tweeters?' Answer: 'Yes please, we want that.' The thought has now been seeded in their minds. When subsequently asked what they need from the product they helpfully suggest 'Dolby and Tweeters?'[1]

It can be worse when the idea comes from the data sheet of a competing product. A product I worked on many years ago had some features in this category. 'Check box' features they were called. Two years after we implemented a particular feature a journalist tried it out for a product review article. He complained that he couldn't get it working. 'Simple-minded journo' we muttered to ourselves as we investigated the problem. It didn't work. We started hunting down the bugs. And, looking back through the source code repository, we found they'd always been there. The feature had never worked. Customers wanted it, but didn't need it, so never used it.

PRD for a 1.0 Product

Writing a PRD for a 1.0 product is even harder than writing one for an established product. There are no existing customers to consult. Prospective customers have to be identified, which involves both a marketing and sales activity.

[1] Oblique reference to a 'Not the Nine O'Clock News' sketch involving the humiliation of a naive gramophone purchaser by an ever so superior sales man.

Part of the 1.0 product release has to include feedback mechanisms for the customer to talk back to the development team. Communication channels are established from the customer to product management and engineering via sales, support and professional services. Companies are now also creating communities of customers around their products, typically through a mailing list or newsgroup. The product development team usually lurks in these places picking up on complaints and comments, and sometimes dipping in to get more detail, or to explain a solution to a newly discovered problem.

Project Process

The PRD has to be signed off by all senior management. The PRD serves as a contract between product management and engineering. Product management promises not to fiddle with the feature set, and engineering promises to implement no more and no less.

The danger of not signing off on the PRD is feature creep, or feature leak. People slowly add new features to the list to be implemented, or people decide that they don't believe in a feature and put it off until it's never implemented. Often this is not malicious intent. Poor documentation, faulty memories, long project cycles, and staff turnover all contribute.

The engineering and quality assurance schedules are drawn up from the PRD. Senior management is presented with the PRD and the schedules for a project review cycle. Decisions are made about the relative importance of quality, schedule, and features within the project, and how that project inter-relates with other projects, and the needs of the business as a whole. These three aspects; quality, schedule, and features, are each be traded off against the other to find a satisfactory balance.

Often time-to-market pressures will dictate that the schedule must be shortened. Management must decide if the priority for the release is quality or features. Often the feature set will be reduced to bring the schedule back to the desired release date. A line is drawn between those features deemed 'in' and those 'out'. The feature prioritization process ensures that all the 'in' features are more important than the 'out' features.

It is important that the 'out' features remain in the document. It must be clear to readers of the PRD that a favorite feature has not been forgotten, just deferred. And, as the project progresses scheduling feedback might allow 'out' features to be moved 'in', and more commonly 'in' features to be moved 'out'.

Of course requirements do change during a project, and a strong PRD actually facilitates the renegotiation process, as much of the determination of relative feature value has already been done. A scaled down project review cycle is undertaken to ensure that the schedules are features are properly balanced.

My PRD

My solution to the 1.0 product definition problem is to just get something out there and create a community around it. The community will serve as a marketplace from which I can learn the value of the features I am and could be offering.

New Reader

Thaddaeus Frogley has been programming professionally for eight years, seven of which in the games industry, five of which in (mostly) C++. He has written articles published online, here in Overload, and over-there in CVu. Despite being dyslexic, all this apparently qualifies him to be on the editorial team, a responsibility he has wisely taken on at the same time as becoming a father. His website of fun programming stuff is here: `http://thad.notagoth.og/`

John Merrells
`merrells@acm.org`

Copy Deadline

All articles intended for publication in *Overload 48* should be submitted to the editor by March 1st, and for *Overload 49* by May 1st.

minimalism • the imperial clothing crisis

By Kevlin Henney

Marx and Engels [Marx+1848] were ahead of the object crowd:

The history of all hitherto existing society is the history of class struggles.

A spectre is haunting software development — the spectre of design. There is a growing recognition that design is the essence of software development and, consequently, the missing link, or ingredient, in much software. Much of what has passed for design or design movement has just been theory without practice, rhetoric or fashion — design is the pictures that you draw, reuse is a principal foundation of object orientation, component-based development leads to flexible architectures. None of these views is particularly useful, and they can sometimes be considered harmful.

design considered useful

In software, design has often been seen as filler between a misunderstood concept of analysis and a view of implementation reminiscent of industrial manufacturing. Design has been seen as a phased activity divorced from code, associated with the power of pure thought and the production of never-to-be-quite-built blueprints. An air it retains today even in many iterative and incremental lifecycles — the old divisions are often still there, just more thinly sliced.

In the past, design has also suffered from an image problem: it has often been branded *formal* — sometimes, *too formal* — equating it to a set of rules, conventions and etiquette, and therefore, by association, stuffy, impenetrable and elitist. To counterbalance — and hopefully displace — these rigid views, design can be considered an endeavour that is continuous and embraces both the code and the conceptualisation of the system. We can use models — drawings or prototypes — to demonstrate things that cannot be shown directly or conveniently in code. But design is a federal concept, and such models may be a part but they are not — except in hegemonies like RUP (the Rational Unified Process) — the whole. It is pragmatism that has acknowledged a more inclusive definition of design, placing it at the centre, recognising that design is indeed formal, but in the same sense that code is formal and in the sense of the word that means "concerned with form".

What properties should we expect of a good design? The influential ten-volume *de Architectura*, by first century BC Roman architect Marcus Vitruvius Pollio, suggested that all construction should possess "strength, utility and beauty" (sometimes translated as "firmness, commodity and delight"). We can relate these directly to code: robustness is clearly something that we value; use is the measure of what is built; and, yes, aesthetics matter, although a full exploration of beauty is outside the scope of this article.

I don't want to go on record as saying that a two-thousand year old book on the built environment has the last word on what software design is all about, but you have to admit that it's not a bad start. Are there other things that we should be looking for that could not be re-shelved under one of those headings? Quite possibly, but it would be fair to say that the quest for goodness in design has attracted a number of camp followers — wannabes that aspire to inclusion on that A-list of desirable properties. Not all are bad, but some have made more progress than they should — cyberspace is right next door to hypespace — and it's time we cleaned up a little. In particular, I would like to think that the grim reaper is slowly stalking the mantras of *reuse* and *flexibility*.

Reuse has proven to be a false idol to worship. It is at best an ill-defined term, and at worst an incorrect one. Moving to objects or components because of reuse is like buying a car because the advert says it's the best: for some people it is wishful thinking, for others it is grasping at straws, and for a few it is simply gullibility. What exactly is *reuse*? If we define it with respect to our experience, it would seem that reuse is often a time waster, an obfuscator, and more of a problem than a solution. I made this claim recently on a course and got a round of applause. What was telling is that most of the applause came from the managers in the group; the precise target audience for much of the reuse hype that kicked off in the late 1980s and is still rolling today.

The word *flexible* is like *reuse*: it should alert you that something nebulous is probably up. Classes and functions are not designed to be flexible, they are designed for a purpose: flexibility is not a purpose, nor is it either a quality or a quantity; it is a bucket term, a catch all, snake oil.

All of this is not to say that we cannot reuse software or that software cannot be flexible. Far from it, it is just that in common parlance they are either vague hand-waving terms ("Our architecture is flexible" — what does that mean? Can it bend over backwards so that it touches its own heels?), or incorrect ("We reused the third-party library" — err, doesn't that just mean you used it... for the purpose for which it was intended?). Without qualification these words mean nothing — but have tremendous power to mislead — and with qualification they do not turn out to be the gleaming foundations of a discipline for software development. Like the emperor's new clothes, there is not much there. When it comes to answering the question "What is important in design?" we should perhaps avert our eyes and look elsewhere.

simplicity before generality

A point I am often at pains to make is that minimalism in software is not about throwing everything out leaving you with nothing. That is nihilism. The emperor needs some clothes, just not too many, and obviously not none. And, of course, it is best if the clothes fit — a few clothes that fit is better than an apparent wealth of clothing that does not.

However, my real motivation in bringing the emperor story into all of this is not specifically the emperor's attire — or lack thereof — as related to a pragmatic interpretation of minimalism, but to point the finger and declare "This has no clothes, there's nothing there!". To be precise — as you may have already established — two fingers. The two virtual properties of reusability and flexibility are twinned with *generality*, and from generality flows a river of good intentions so deep you could — and many do — drown in it. It might at first glance be assumed that reuse would form the cornerstone of a minimalist philosophy of software development, but the opposite transpires.

I confess that the promise of reuse was never one that attracted me, and was a topic that I never felt entirely at home with, at least not in the sense that it was most talked about. My traditional stance was that reuse was a social issue not a technological one, a matter of culture rather than of mechanism — mechanism could assist but it could never cause. This view was still reachable from the published party line, although in truth most others in the party did not follow the line either.

In recent years I have called into question many of the buzzwords that pass for communication (but pass all understanding). The realisation has crept up on me that even the mainstream non-mainstream view, so to speak, is inaccurate and insufficient. These

days, I adopt a more republican stance: when it comes to clothing, so to speak, we should not even be talking about the emperor. It doesn't help. Reuse is not the main challenge facing software engineering; typing is not the main bottleneck in software development (which means that most third-party code generation tools are actively solving the wrong problem); bureaucracy is not the missing link in the development process. Sometimes the shine is taken off our ability, as a profession, to solve problems by a frequent and uncanny knack of identifying the wrong problem to solve. Let's try to simplify before we generalise.

The following is from an email I sent Bruce Eckel following a request on his list for design principles to include in his book, *Thinking in Patterns* [Eckel]:

> *Simplicity before generality: A common problem we find in frameworks is that they are designed to be general purpose without reference to actual systems. This leads to a dizzying array of options that are often unused, misused or just not useful. However, most developers work on specific systems, and the quest for generality does not always serve them well. The best route to generality is through understanding well-defined specific examples. So, this principle acts as the tiebreaker between otherwise equally viable design alternatives. Of course, it is entirely possible that the simpler solution is the more general one.*

The slightly flippant tone of the last sentence may hide my degree of conviction: it is not just that it is "entirely possible", it is actually "quite likely".

Many things that are designed to be general purpose often end up satisfying no purpose. Software components should, first and foremost, be designed for use, and to fulfil that use well. Designing for all seasons is both difficult and not always desirable, a realisation that helps explain the small markets for thermal bikinis and Ford Edsels, as well as the challenge of designing general-purpose software components.

Reflecting both on my work in library and framework development and on my role as a user of such commodities, I have seen the strong temptation and wasteful consequences of general featurism. I have also seen a more restrained approach bear fruit. It may be a cliché, but less really can be more.

Generality is not, of itself, necessarily bad, but we can often identify the odour of *speculative generality* [Fowler1999]:

> *Brian Foote suggested this name for a smell to which we are very sensitive. You get it when people say, "Oh, I think we need the ability to this kind of thing someday" and thus want all sorts of hooks and special cases to handle things that aren't required. The result is often harder to understand and maintain. If all this machinery were being used, it would be worth it. But if it isn't, it isn't. The machinery just gets in the way, so get rid of it.*

Generality should equate to simplicity and simplification. Generalisation can be used as a cognitive tool, allowing us to reduce a problem to something more essential, a clearer abstraction that offers greater compression [Henney2001]. However, too often generalisation becomes a work item in itself, and pulls in the opposite direction, adding to the complexity rather than reducing it. The initial sweetness of a general solution can become overwhelming as it grows, to the point that we feel like we are drowning in syrup.

Design is compromise, and all flexibility is a double-edged sword. Many people mistakenly see design decisions made in the name of flexibility as win-only situations, a narrowness that belies reality. If we equate flexibility with degrees of freedom, then the degrees of freedom in a design should be reasonable — which I mean that in the deepest sense of the word: based on reason. In pursuit of arbitrary flexibility you can often lose valuable properties, accidental or intended, of alternative designs [Petroski1999]:

> *In the mid-twentieth century it became the fashion in library architecture to design buildings as open-floored structures in which furniture, including bookcases, could be moved at will. The Green/Snead Library of Congress bookstack that six decades earlier had been declared "perfect" was now viewed as disadvantageously locking a stack arrangement into the configuration of its construction. In the new approach, reinforced concrete floors carry the loads of bookshelves, so that they can be arranged without regard for window placements. This apparently has the appeal of flexibility in the light of indecision, for planners need not look at the functional and aesthetic requirements of their space and its fittings with any degree of finality; they can always change the use of the space as whim and fashion and consultants dictate. It is unfortunate that such has become the case, for it reflects not only a lack of sensitivity to the historical roots of libraries and their use but also rejects the eminently sensible approach to using natural light as a means of energy conservation if nothing else. There is little more pleasing experience in a library than to stand before a bookshelf illuminated not by fluorescent lights but by the diffused light of the sun.*

And speaking of libraries, it is worth noting that one the strange things about a so-called reuse library is that it's one of the few libraries people only seem to deposit things in but never take things out. A more honest term is *reuse repository*.

A more pragmatic and minimal design style does not mean writing code that hugs its assumptions so closely that only major surgery will separate the two in the event of change. It is not about hardcoding everything: it is about both sufficiency and finding the right amount of space between the elements of your solution, offering the right amount of slippage or wriggle room. The challenge of design is in seeking and maintaining local minima of sufficient simplicity with sufficient generality that the integrity of the design is not easily disturbed, and the energy required to adapt to change is proportionate to the degree of change.

Generic programming often provides good examples of sufficient generality. Note that *generic* is not the same as *reusable*: something may be generic to express the simplest and most stable model. But at the same time, genericity can be a hard tap to turn off, whether expressed through template parameters, function arguments or an interpreted interface. It is tempting to create some kind of final solution — a killer class template that is parameterisable beyond belief, or indeed comprehension. In practice such parameterisation severely reduces the utility of code. One size does not fit all: I don't shop at a single place to buy all of my goods — food, cars, etc — and although this is possible, one is left with a sense of diminished quality through lack of appropriate specialisation. I get better fruit from the local grocer. Likewise, restaurants: if they try to cater for all types of food, they do so blandly and uniformly, squeezing out the variety and standardising the experience.

ex libris

The benefits of libraries are different to those of reuse repositories. The idea that an arbitrary piece of code is a candidate for reuse differs from the idea of taking a piece of code and generalising it — promoting it through practice and

empiricism— into a library, or conceiving of a code library that scratches a particular design itch, expresses a particular design idea, refactors a common code chunk.

You cannot reasonably refer to *reuse* in the context of a library — "We are reusing the AWT." "Oh, and what did you do with it the first time around?" — because there are only three things you can do with a library: use it; misuse it; not use it. Some might attempt to *leverage* a library, but it transpires that this is just a neologistic circumlocution for *use*. (Aside: In addition to a popular suggestion that its use as a verb should be banned, there could be an additional fine on native English speakers pronouncing it "l*evv*erage" if their common cultural pronunciation is "l*eev*erage". You could probably raise a lot of good money for charity with an office swear box filled on such jargon.)

Libraries offer value based on use, not reuse. And library architecture, whether in a loose confederation of parts or the more tightly knit community of a framework, is based ultimately on modular concepts. But what kind or scale of module forms the basis of design? Often use at the level of the small is dismissed as insignificant. And yet this is the level at which most libraries and infrastructure projects have been most successful. The view that we are striving ever upwards, always building neatly on the layer below, moving towards a greater object society or component order seems to have ensnared a mindshare. It has created a mantra all of its own: once I worried about structured control flow; then I worried about my classes; and then I worried about components; but now life is easy, all I need to worry about is how to interface and tune these off-the-shelf distributed systems — I no longer need to worry about any details. It's not true in software and it's not true elsewhere [Salingaros+2001]:

> *A free design process that allows for numerous subdivisions permits mathematical substructure on many different scales. By abandoning an empty modularity, one has access to solutions based on a far richer approach to design that creates visually successful buildings. Art Nouveau architects like Antoni Gaudí used small modular elements (such as standard bricks) to create curved large-scale structures. This freedom of form contrasts with those instances where a building reproduces the shape of an empty rectangular module. The small scale can link to the large scale mathematically, because scaling similarity in design is a connective mechanism of our perception. Therefore, the materials can and do influence the conception of the large-scale form, and the larger a module, the stronger the influence. When one chooses to use large, empty rectangular panels, these will necessarily influence the overall building; often implying a monotonous, empty rectangular façade.*

Uncannily, this sounds like many contemporary large-scale component-based architectures. It is not modules but inappropriate modularisation that causes problems: it can be as bad as the absence of modularity. On the one hand you have a monolithic slab of code and on the other lots of prefabricated slabs that somehow just don't seem to quite fit or make sense together [Salingaros+2001]:

> *There are arguments to be made in favor of modularity, but not for the way it is used in many buildings. If we have a large quantity of structural information, then modular design can organize this information to prevent randomness and sensory overload. In that case, the module is not an empty module, but a rich, complex module containing a considerable amount of substructure. Such a module organizes its internal information; it does not eliminate it. Empty modules, on the other hand, eliminate internal information, and their repetition eliminates information from the entire region that they cover. Modularity works in a positive sense only when there is substructure to organize.*

Undifferentiated modularity seems to be a genuine problem. We should work with more than one unit of modularity, and many programmers do so successfully: method, class, package, component, etc. Software design, and therefore architecture, is recursive. The reason many people move to object and component technologies is precisely because of the many and varied levels of granularity on offer — a better fit to their grasp of the problem and expression of a solution, a finer level of control over the detail and its relevance, better choice of abstraction, better resulting compression. Rather than the brusque and FORTRAN-esque levelling of program then function, we have a view that can zoom out or in to the system to the level that we find appropriate to understand a particular behaviour or solve a particular problem.

And, to be effective, a good grasp of modular diversity must be coupled with an appropriate sensibility, an understanding of its intent and reach [Gabriel+2000]:

> *The real problem with modular parts is that we took a good idea — modularity — and mixed it up with reuse. Modularity is about separation: When we worry about a small set of related things, we locate them in the same place. This is how thousands of programmers can work on the same source code and make progress. We get in trouble when we try to use that small set of related things in lots of places without preparing or repairing them.*

The module, at whatever scale, is one of our best tools. But identifying good modules is hard; software development is a matter of design. It may seem almost counterintuitive, but a design born of a minimal and pragmatic approach will often have more modular parts than one that does not. However, the parts will not all serve the same purpose — and nor will they serve any arbitrary purpose — and they will not all be the same size.

commodity

So if it is not reuse and flexibility, what role then do libraries and related infrastructure play? Commodity — which can be found, by happy coincidence, in the alternative translation of Vitruvius' three essential architectural properties. A commodity is quite a different thing to something that is reusable. A commodity is something of value and of — and for — use. A commodity defines a stable platform, something that can be assumed or taken for granted. A commodity is a product that has been built (or mined and refined) intentionally, rather than an accidental but serendipitous artefact. In real world terms we would never confuse the quite different ideas of commodity and reuse. Whilst there is certainly overlap, they are not even mistakably synonymous.

We can see that developing a commodity is quite a different undertaking to developing reusable code. A lot of per-project code that is designed to be reusable simply isn't, and is often borderline useable. The open pursuit of reuse is unfocused and adds an inappropriate overhead to some projects, often to the point of compromising quality or schedule. The return on investment — time, effort, complexity, money — is often not recouped. Far better to create code that is fit for purpose and fits with its purpose. If you follow some of the common practices for decoupling — easier testing, less impact of change — it is more likely that the code will see more general use, perhaps even as a commodity. Some projects recognise that it will cost them extra to get some kind of reuse, but if they replaced the word

reusable with *commodity* they might reconsider how much extra was really needed to be effective.

In a single project you don't have reuse if a piece of code is used in more than one place, that's just what should be going on. This is just good local design: usage with minimum duplication. A definition of reuse based on the simplistic and unqualified definition of "use more than once" would be trite and quite useless — what value would be gained by replacing "function *A* calls function *B*" with "function *A* reuses function *B*"?

If you view a system in terms of layering there is an interesting relationship between the use of refactoring and the balance of logic in the system [Collins-Cope+2000]:

Refactoring visibly lowers the centre of gravity of the application by finding the commonality and factoring out the difference.

This is like annealing. Refactoring provides enough energy to a system for it to relax into a new and more comfortable state, a new local minimum. The effect of refactoring commonality is to tame the complexity of your system. Repeated tempering, with a conscious effort to reshape, is more likely to move code towards commodity than a vague 'plan' or 'strategy' based on reuse.

And, as already noted, if you use a library or framework then this again is not reuse, this is the idea of commodity and platform. Features migrate into platforms over the years, e.g. threading, networking, GUIs, etc. Using an application server is not reuse, it is use of a commodity as was intended by its design.

Most concepts of reuse are invalid or unachievable in practice. Therefore, by definition, most *reuse strategies* are destined to fail. Consider the inherent contradictions or muddled goals that sometimes pop up on company technology adoption plans: "We will start with a pilot project to demonstrate code reuse on a three-programmer four-month development". Often what started out as the path of least resistance can become the path of least convenience.

accidentally on repurpose

It appears, in part, that the problem is one of vocabulary: we are misusing words. For some reason, *reuse* is seen as a more glamorous and elevated term than *use*. We can reduce most uses of *reuse* to something more precise: use of a commodity, such as a library, or the result of refactoring to avoid duplication of logic within a system. We're in programming not marketing: precision matters. At least *refactoring* now has a certain cachet. (Perhaps there's something going on with the prefix *re-*? *Rework*? *Recode*? *Retest*? Hmm, then again, perhaps not.)

So how did we end up with the reuse groupthink? And why has it so often been allied with inheritance in object-oriented approaches? Both hindsight and foresight suggest that it is a damaging mindset that upsets both schedules and design. We know that hype merchants are responsible in part, but the cause can also be traced to the more honest confidence and ebullience of expert individuals [Meyer1997]:

Here are the most important external quality factors, whose pursuit is the central task of object-oriented software construction....

Reusability

Reusability is the ability of software elements to serve for the construction of many different applications.

A few choice words come to mind when I read this, even without the exploration I have just undertaken of what reuse is not. However, a quick count to ten and I can respond with something a little more moderate: there is no such thing as reusable software, only software that has been reused. This response is adapted from the porting maxim that "there is no such thing as portable code, only code that has been ported". However, unlike reusability, portability is actually a quality that has some meaningful quantification, both empirically and theoretically, and can be reasoned about without resort to theological debate.

Whilst the definition given of *reusability* is not necessarily a bad one, the problem is that it is listed as being a core goal and activity of object development. The problem is then compounded by sewing up this tidy view of the world with the following mythtake [Meyer1997]:

Progress in either reusability or extendibility demands that we take advantage of the strong conceptual relations that hold between classes: a class may be an extension, specialization or combination of others. We need support from the method and the language to record and use these relations. Inheritance provides this support.

The perspective given, which some might consider as charmingly naïve and others would view as downright misleading, is countered by a wealth of theory and practice that suggests that such an approach to development, and especially such a view of inheritance, is unsound. Commentary reflects this [Murray1993]:

A myth in the object-oriented design community goes something like this:

If you use object-oriented technology, you can take any class someone else wrote, and, by using it as a base class, refine it to do a similar task.

Even in the design of commodity items such as class libraries — commonly and mistakenly equated with reuse — inheritance is not necessarily viewed in glowing terms, as witnessed by Martin Carroll and John Isner, designers of USL C++ Standard Components [Gabriel1996]:

We take the minimalist approach to inheritance. We use it only when it makes our components more efficient, or when it solves certain problems with the type system.

We do not intend for our components to serve as a collection of base classes that users extend via derivation. It is exceedingly difficult to make a class extensible in abstracto (it is tenfold harder when one is trying to provide classes that are as efficient as possible). Contrary to a common misconception, it is rarely possible for a programmer to take an arbitrary class and derive a new, useful, and correct subtype from it, unless that subtype is of a very specific kind anticipated by the designer of the base class. Classes can only be made extensible in certain directions, where each of these directions is consciously chosen (and programmed in) by the designer of the class. Class libraries that claim to be "fully extensible" are making an extravagant claim which frequently does not hold up in practice.... There is absolutely no reason to sacrifice efficiency for an elusive kind of "extensibility."

Yes, reuse does happen, but it is not in the tidy centralised or library-governed vision that you may have been sold, framed originally by many High-Modernist Object Gurus. It is normally a more ad hoc, grassroots, opportunistic and — yes — cut-and-paste affair [Raymond2000]. It has a haphazardness to it that belies the feed-forward, deterministic nature of many software development lifecycles that claim to address reuse. It is not sufficiently deterministic or controllable to form the basis for project planning or an economic model of software development. *Reuse strategy* is practically an oxymoron. How can you have a reasonable strategy based on good luck? We normally insure against accident, not for it.

Reuse in the outside world is also a less organised affair [Brand1994], sometimes built almost purely on compromise. It is normally concerned with recycling and repositioning; not necessarily the glamorous vision of rapid development, but a matter of development by necessity and disparate parts, subdivision and differentiation. The charm of something that is reused often arises from its accidental properties rather than from a reduced Cartesian order.

But I do not wish to misrepresent the object community. For most of us the reason to adopt an OO or related style has been about the qualities that it gives development, not the quantities. You can say what may actually amount to the same thing in two radically different ways: "I use *X* because it is more expressive, allowing me to articulate a more eloquent design" versus "I use *X* because it makes me more productive". The former is about the individual, the human, and the latter is about an automaton; more cog than cogitation. I tend to find surveys and articles that talk about productivity do so from a pseudo-scientific point of view. Such dehumanisation is also found in the rebranding of programmers as plug-and-play *resources*. Being a resource is not the same as being resourceful: oil, coal and tin are natural resources; a third-party code library is a software resource; even our skills and experience can be considered knowledge resources. However, we, as individuals, are not resources: it is our use of resources that makes us resourceful. The *productivity of a resource* does not sit on one side of an equation with *reuse* on the other. It would be a tidy but deceptive fiction.

But I digress. Where any form of reuse can be of assistance is in the reduction of a problem to something similar, something familiar. Where reuse is most prevalent is in our knowledge, the use of experience to resolve similar and out-of-context problems. In each case, the reuse is a matter of adaptation, repurposing, learning. It is distinctly unmodular, and rarely preplannable at a detailed level. But the absence of a fixed and fine plan does not denote chaos, any more than the omission of a detailed leaf plan filed in your local town hall suggests that trees do not represent some form of order.

So remember, design is central to software development and most of the worthwhile reuse and flexibility in software development comes from your side of the keyboard.

Kevlin Henney
kevlin@curbralan.com

references

[Brand1994] Stewart Brand, *How Buildings Learn*, Phoenix, 1994.

[Collins-Cope+2000] Mark Collins-Cope and Hubert Matthews, "Let's Get Layered: A Proposed Reference Architecture for Refactoring in XP", XP 2000, http://www.oxyware.com/Publications.html.

[Eckel] Bruce Eckel, *Thinking in Patterns*, work in progress that can be downloaded from http://www.mindview.net/Books/TIPatterns/.

[Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Gabriel1996] Richard P Gabriel, "Patterns of Software: Tales from the Software Community", Oxford, 1996.

[Gabriel+2000] Richard P Gabriel and Ron Goldman, "Mob Software: The Erotic Life of Code", delivered by Richard Gabriel as a keynote at OOPSLA 2000, http://oopsla.acm.org/oopsla2k/postconf/Gabriel.pdf.

[Henney2001] Kevlin Henney, "Minimalism: Omit Needless Code", *Overload 45*,

[Marx+1848] Karl Marx and Friedrich Engels, *The Communist Manifesto*, Phoenix, originally published 1848.

[Murray1993] Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

[Petroski1999] Henry Petroski, *The Book on the Bookshelf*, Vintage, 1999.

[Raymond2000] Eric S Raymond, *The Cathedral and the Bazaar*, O'Reilly, 2000, http://www.tuxedo.org/~esr/writings/cathedral-bazaar/.

[Salingaros+2001] Nikos A Salingaros and Debora M Tejada, "Modularity and the Number of Design Choices", *Nexus Network Journal* 3(2), 2001, http://www.nexusjournal.com/Sali-Teja.html.

Of Minimalism, Constructivism and Program Code

By Allan Kelly

In part this essay is a continuation of Kevlin Henney's arguments in "minimalism • omit needless code," but it is also, in part a response and counter argument. Let me say up front: I like minimalism, I like Kevlin's piece and I agree with much of what he says.

However, I worry about some of the advice. I worry about the direction of *modern C++*, I worry about what I perceive to be an increasingly elitist attitude in C++ coding. I worry about all these things because of accessibility.

Less is more

First let me take Kevlin's loop example. To remind you, this is a for-loop that prints out the content of a vector, example 1 is Kevlin's first pass using an integer to access the vector.

```
// example 1
std::vector<std::string> items;
...  // populate items
for (std::size_t at = 0;
          at != items.size(); ++at)
  std::cout << items[at] << std::endl;
```

Next the code is changed to use an iterator:

```
// example 2
std::vector<std::string> items;
... // populate items
for (std::vector<std::string>::iterator
                  at = items.begin();
     at != items.end(); ++at)
  std::cout << *at << std::endl;
```

What has this change given us? In terms of functionality: nothing; in terms of performance, well maybe we save a couple of index operations; in terms of style it is more "modern" – that is to say it looks more like *modern C++* because we are using iterators.

But in our haste for fashion what have we lost? We have increased code complexity, we are demanding a more detailed knowledge of C++ and its library than we did previously. This isn't a crime, after all, if you don't know iterators how can you claim to know C++?

On balance I think this is a judgement call, it depends on the style of the program. I can't get worked up about this.

After a couple more iterations Kevlin gives us:

```
// example 3
std::vector<std::string> items;
... // populate items
typedef
  std::ostream_iterator<std::string> out;
std::copy(items.begin(),
          items.end(),
          out(std::cout, "\n");
```

Now we have less code still. However, we have significantly increased the amount of context information required to understand the code. We must now understand std::copy, std::ostream_iterator and the magic typedef. Further, we have lost our end-of-line abstraction, std::endl, now we must know the correct end of line terminator – CR? LF? CR and LF? Is this output intended for the screen where it hardly matters? Or is it intended for a file where it is more important?

So, although we have less code we actually require more information to understand it. This cuts to the heart of one of the fundamental problems of code reuse: to reuse code, that is, to be able to write less code, we must know more, that is we must increase our knowledge. In this case it is the standard C++ library, yes, every C++ programmer should know the library, but I ask you: do you know the entire library?

Is this really minimalism?

"The aim of Minimalism is to allow the viewer to experience the work more intensely without the distractions of composition, theme and so on[ii]."

Using this definition example 3 is not minimalist because it is full of distractions, to understand the code we must understand the context it is written in – the theme is critical to understanding the code. Example 1 may contain less code but it is actually closer to a minimalist solution.

"Constructivist art is marked by a commitment to total abstraction and a wholehearted acceptance of modernity.... Objective forms which were thought to have universal meaning were preferred over the subjective or the individual. The art is often very reductive as well, paring the artwork down to its basic elements[iii]."

In fact, example 3 is more constructivist in nature than minimalist. It is reduced to the basic elements using every abstraction available. One of the best known constructivists, Wassily Kandinsky, created whole pictures from his context theories:

"elementary shapes were yellow for the triangle, red for the square and blue for the circle, mixtures of colour would need to follow from mixtures of form. A pentagon mixes a square and triangles: it must therefore be orange[iv]."

As in art such minimalism or constructivism can make program code difficult to comprehend.

Need for context

"For me it was minimalism. I felt at home with it, felt I might of invented it. Yet I have totally failed to write about it..... [the ultra minimalist sculptor] Sandback can transfix and subjugate me with a length of twine strung across a corner of a room but I have found no way to write about the experience[v]."

So wrote the art critic David Sylvester in 1996. Let us examine this for a moment. Here is an acclaimed art critic, a man who is paid to understand and explain modern art, one who we expect to understand Pop Art, Conceptual Art, Abstract Art and more, yet here he is holding up his hands and saying "I can't explain minimalist art."

As we push further and further onwards in the direction of minimalism it becomes less and less accessible. This is true of art, literature and program code.

Simply claiming that program code is not written by dummies, for dummies is not enough. As professional developers we have a responsibility to ensure our code is maintainable. To paraphrase Arthur C Clarke: *"any sufficiently advanced software implementation is indistinguishable from unmaintainable code[vi]."*

Let us suppose you write your masterpiece of a system and you then leave the project. Who are you going to hand it over to? If you are lucky you have a team of equally capable developers who are ready to take over. More likely as Steve Maguire[vii] points out it is junior developers, we have a responsibility to these people to ensure the code is accessible.

If you are in a position to choose your own replacement you may be able to manage this situation. More likely you aren't, the company will go out and hire another contractor. Suppose you give them a few months notice, and suppose they choose a good employee and send him on the appropriate C++ courses. Can he now maintain your program? How many years' experience does someone need to maintain your masterpiece?

As I have observed before, we do not develop in a vacuum, we must be aware on the context we develop in. Are our universal forms truly universal or do they form a barrier to understanding?

Maintenance and reuse

There is not much maintenance in the arts community. Rauschenberg's *White Painting* owned by San Francisco Museum of Modern Art is an exception; the artist has given instructions that this all white piece should, and has been, from time to time repainted.

Software development, like art, as an exercise is almost pure intellectual creativity. But unlike artists we have to produce works that are practical, useful and maintainable.

As an exercise in intellectual effort art has already visited many of the same ideas as software. Dan Flavin, for example, practises component reuse, using standard neon-light tubes (which may be readily purchased by anyone) he constructs pieces such as his Monument to V. Tatlin.

Carl Andre (considered a minimalist) has pieces made in a factory. Consider Maholy-Nagy (a constructivist):

"just before he left for the Bauhaus [he] ordered a series of three paintings by telephone, giving a factory that made enamel signs precise verbal instructions and leaving the manufacture up to them[viii]."

As software developers we are in this space: we want to use standard components, we want to be able to produce software from precise specifications – some would even argue for a code factory. But, much of the art that arises from these techniques is considered inaccessible, this is not a quality we want in our code. Ironically, minimalist paintings created to be free on context are difficult to understand exactly because of the lack of context!

There is more than one way to skin a cat

Sculpture, *minimalism*, *installation*, *ready-mades* – these are the languages of modern art. They allow artists to express their ideas. C++, Java, Python, Perl – these are the languages of software. They allow developers to express their ideas.

We choose a language that is expressive enough to manipulate our ideas, we choose Java for internet applications, Visual Basic for desktop application and so on. None of these languages yield more computing power, they are all Turing equivalent, no more no less. Yet each in its own field is more expressive, the power of expression is what gives one language power over another. The expressiveness means nothing to the machine; it is expressive power to the human developer.

This richness of expression comes at a cost: the size and complexity of the language increases. Wirth's language family has gone down the path of minimalism, yet Oberon and Modula-3 are seldom used outside research environments. Oberon is so minimalist it eliminates the enumeration types and the for-loop, in becoming so minimalist it has removed the expressive power of its users. This pursuit leads to Orwell like *NewSpeak* where it is not possible to think an incorrect thought because the language does not permit it.

Instead the successful languages allow expression - and they allow us to make mistakes. Maybe the balance has gone the wrong way; maybe we do have too many features. Ironically many of these features are aimed at allowing reuse. In example 3, code reduction occurs because we use a ready-made algorithm. Yet the price of this is that we must understand the context.

Reuse is the Holy Grail of software development, yet we must not forget that the first reuse of our code occurs in the maintenance phase, MegaCalc version 1.1 reuses almost everything from version 1.0. Making code more accessible, that is, more understandable, benefits reuse in version 1.1 and in MegaWord 1.0.

We want to reuse and have added vocabulary to our language to allow expression of reuse. However, growing the language does not lead to minimalism, it leads to constructivism because we have increased the amount of context information required to understand the code.

The superfluous has no place in minimalism, constructivism or program code. However, we must consider our audience. We can do this through shared context or through explicit statements. Our shared context, our universal truth, is our language. But this truth is not singular, it is many and we cannot expect even a true believer to know the entire truth.

We can reduce the physical amount of code but if we simultaneously increase the amount of knowledge required to understand it what have we gained?

> *"C++ supports the notion of gradual introduction... programmers can remain productive while learning C++*[ix]*."*

Of the examples above no one piece of code is superior: they are simply different. They may simply be examples written by developers at different points in their learning cycle. To tell the author of example 1 that she should have written example 3 adds nothing to the code but you may have an adverse effect on her self esteem, her attitude and how she views you.

The place for minimalism

> *"The results were shown... in New York in 1966... the artist [Andre, Judd, Morris, Flavin] shared a concern with stripping sculpture down to its essence*[x]*."*

Most developers have rejected minimalism languages. Minimalism at the code level is self-defeating. Constructivist code is both advantageous and dangerous.

The place for minimalism is in our designs. A design should stand alone without distractions. Our design is the kernel of our system and as such we should ensure it is extendable. Extending a design should not introduction distractions and complications.

Conclusion

There is nothing wrong with minimalism. I too feel at home with it, I too feel I could have invented it. However we must direct it precisely to avoiding feature creep. We must delve down into the essence of our problem domain and sculpt our solution. We need the expressive power of our language but this brings the responsibility to ensure that we use it correctly.

> *"In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction*[xi]*."*

Constructivism and minimalism are examples of abstraction. They are also tools in their own right, which can be used to explain and understand software. Our challenge is to keep our products accessible, the blind pursuit of abstraction, minimalism or constructivism is evil when it deprives us of context or burdens us with too much context.

For me, Wassily Kandinsky could have been talking about software development when he said:

> *"The 'artist' gives birth to a creation in a mysterious way full of secrets and enigmas. Freed from him, it attains an independent life, becomes a personality, a subject whose spirit breathes on its own but also lives a real material life; who is a being*[xii]*."*

How many of us know the life our creations are leading today?

Allan Kelly
allan.kelly@bigfoot.com

References

[i] Kevlin Henney, Overload 45, October 2001
[ii] http://www.artmovements.co.uk/minimalism.htm
[iii] http://www.artmovements.co.uk/constructivism.htm
[iv] Frank Whitford, *Bauhaus*, Thames and Hudson, 1984
[v] David Sylvester, *Curriculum Vitae*, About Modern Art, Pimlico, 1997
[vi] I originally made this observation on the ACCU-general mailing list in September 2001, Ric Parkin was good enough to provide the original quote *"Any sufficiently advanced technology is indistinguishable from magic."*
[vii] Steve Maguire, *Debugging the Development Process*, Microsoft Press, 1994
[viii] Frank Whitford, *Bauhaus*, Thames and Hudson, 1984
[ix] Bjarne Stoustrup, *The C++ programming language, third edition*, Addison-Wesley, 1997
[x] Anna Moszynska, *Abstract Art*, Thames and Hudson, 1990
[xi] C. A. R. Hoare, "Notes on Data Structuring", in *Structured Programming*, Dahl, Dijkstra and Hoare, Academic Press. Thanks to Rob D'Entremont and Peter S Tillier on ACCU-general for providing the source of this quote.
[xii] Quoted at Berkeley Art Museum, 2001

Minimalist Constructive Criticism

By Kevlin Henney

I enjoyed Allan's article, but couldn't help thinking that it was about something else. With the exception of the use of three code fragments, I was left with the feeling that I was reading about an article I didn't write. I looked back over my article and saw most of the relevant points analysed, rejected or otherwise expressed. Perhaps I didn't express them clearly or directly as I might have done? So, I guess these comments are also a continuation, response and counter argument!

[0] Just to clarify, the reason that example 2 is presented is because it is often shallow orthodoxy — fashion without style.

[1] "\n" is an abstraction for CR, LF and CR+LF, depending on the platform. That is a requirement of the standard, and is the same in C as it is in C++. The only thing that std::endl offers us in addition to "\n" is an explicit std::flush for all stream types. For streams of uniform data this is not a reasonable necessity. And whether "\n" requires more or less context to understand than std::endl is more a matter of opinion rather than fact!

[2] Be careful with the use of the word "reuse"... ;-)

[3] On my use of the word "minimalism", it is at once intended to be both descriptive and provocative. In use as an ordinary noun "minimalism" refers to omission of the unnecessary and the inessential. The idea of reduction to sufficiency, but not to its elimination, is implicit. Following from this is the inherent coupling to a context and style of composition. This points to the understanding of "minimalism" in terms of "minimum", a mathematical concept, and in particular the application of that to design, which is both a political and a technical act — definitely the main topic of interest.

If one relates "minimalism" to "Minimalism", this points more to the provocative side of things, making reference to art rather than design or science. "Minimalism" has the specific meaning as a term that describes visual art and music movements. It would, however, not be correct to equate what I am talking about with descriptions — and they are only descriptions, not definitions — of the Minimalist movement (fond as I am of it). Refer, yes; equate, no.

The label "Minimalism" was coined originally by critics and not by its practitioners. It therefore suffers the volatility and subjectivity that such a classification will inevitably experience. Contrast:

> *"The aim of Minimalism is to allow the viewer to experience the work more intensely without the distractions of composition, theme and so on."*

with

> *"Minimalism here becomes deeply relativistic, supportive of the view that 'art' can only acquire value or worth in relation to external factors, such as its social or institutional meaning." (After Modern Art, David Hopkins)*

and

> *"Minimalism is against the subjective realm and for the objective realm." (This is Modern Art, Matthew Collings)*

Clearly a movement as much divided by critics as it was defined by them. For some there was the idea that Minimalism made the experience more intense and whole because of the removal of context, and for others there was the idea that intensity and wholeness arose precisely because the art acted as a lens through which the context could be seen.

So I would have said that although it makes for an interesting literal comparison, it is perhaps not a wholly valid conceptual one. The "minimalism" of my article is sympathetic to the "Minimalism" of art, but is more truly drawn from maths and science than art, and is deeply bound to context and purpose through design. And therefore in answer to the question raised: yes, this really is "minimalism", but it is only in part "Minimalism".

[4] In terms of Constructivism, what I am describing also falls a little short of that art movement as described by

> *"Constructivist art is marked by a commitment to total abstraction and a wholehearted acceptance of modernity.... Objective forms which were thought to have universal meaning were preferred over the subjective or the individual. The art is often very reductive as well, paring the artwork down to its basic elements."*

To specifically relate it back to the code fragments, the use of STL is not the same as embracing modernity — although I and others frequently use the term "Modern C++" as a description of the third age of C++. We must be careful in how we apply this label: there are many different styles of code that can claim this title, and some have contradictory aims. Some might be considered elitist and others not. To lump them all together and brand them the same way is not always appropriate. Although the spectre is raised, the question of elitism is never really explored or answered in the article. It's a good question, but a deep one so I will leave it to one side as it is slightly off the main track.

Is the STL necessarily modern? Not really. It represents the adaptation and adoption of functional programming techniques into C++. Is STL radical? Yes, from the perspective of Classic C++ its introduction pushes C++ in quite a different direction to a core Modernist Object creed (interesting mix of movements here...). Some would say that the direction is not different, that it is either a course correction or a clarification. But not all that is radical is modern. It is modern in the sense that "Modern English" is modern, but not truly modern in the sense of Modernism.

I think my article also makes clear that a total — as opposed to sufficient — commitment to abstraction is not the answer. However, there are clearly common points: universal meaning and a reductive tendency ring bells. If such a comparison is to be made, Constructivism more accurately describes generative programming in C++ (see the work of Andrei Alexandrescu, and Krzysztof Czarnecki and Ulrich Eisenecker's) than it does generic programming.

[5] Interestingly, the issue of universal forms actually points to the elimination of context as a factor in Constructivism. If the forms are not universal we have context, if they are universal the distinction between having context and not becomes at best academic — "*When everything is a triangulation point, then nothing is a triangulation point*"

(Software Requirements & Specifications, Michael Jackson). Of course, how this relates to software development is another question!

[6] It is interesting to note that Oberon fails the requirements of sufficiency and utility. Whilst it is has moments of elegance, its ability to express abstractions with appropriate compression is severely restricted. It is as if it has missed the minimum and punched its way through to the x-axis. As described either in terms of what I was exploring in my article or in terms of an art movement, Oberon is not a {m,M}inimalist tool. It increases the amount of code written and the amount of reading required to recover meaning. The idea of capturing a model of usage is missing, and is why the reduction is uninformed from a language user's perspective.

[7] A telling exercise is to attempt to describe the core of each C++ code fragment in English. The first example is quite long, the second example is longer and the third example is the shortest. I would contend that the last fragment is the one that has a structure that most closely captures the intent: "*copy from the beginning to the end of items out to the standard output using newline separation*".

[8] The aim of minimalism as described in my article is not to "*reduce the physical amount of code*". The reduction of code is not a principle; it is a side effect — "*Vigorous writing is concise.... This requires not that the writer makes all sentences short... but that every word tell*" (The Elements of Style, Strunk & White). If you reduce code for its own sake you have lost your audience. To further quote from my article, your code "*needs an audience, and it will be better off for having one in mind*". So I would say that this is a point of agreement rather than one of divergence between the articles.

[9] In software, coding is still designing. There is little distinction except in level of detail and range of formality: coding is a subset of design. To make a hard and fast distinction is often artificial, and I would say the cause of much misguided thinking in software development. The sculpting of a solution is the process of design, and it makes little sense to have one recommendation for design and a contradictory one for code: if the design should be minimalist then that already covers what we should be doing in code. Of course, we have to choose the right working definition of minimalism :-)

Kevlin Henney
kevlin@curbralan.com

How about a Turner Prize for software?

By Allan Kelly

Art moves forward because it is readily visible even when it is not accessible. Software like architecture usually hides the internals. Maybe, like Richard Rogers, we should try putting more of our code on the outside!

Source is openly available for FSF and OpenSource projects but only those who participate in the project look at it, and then not comparatively. Most commercial software is locked away where not even the customer can see it.

I'd like to suggest a competition for quality software focused on the code itself. Programming contests have existed for years but these are normally races against the clock. This contest would be run a bit like an art prize. Any organisation may enter provided:

- the software has been delivered to a customer during the last twelve months
- the software has been developed by a team
- source code is available for general viewing: together with documentation it would be made publicly available on the web

Judging would be by a panel, say five people including a chair, who would assess each entry and decide on a winner. Membership would vary year to year and would be made up of respected authorities, academics and jobbing programmers.

Following the conclusion of the contest the judges' notes would be added to the web site. Entries would remain publicly available and thus serve as a reference and teaching tool. (We may choose to move the entries to a virtual gallery.)

The contest would aim to promote best practices, advance current techniques and provide large-scale examples of quality projects. This is not designed as an effort to further the open source movement; it is intended to improve software quality.

Any organisation could enter as long as the software was delivered to a real customer. Although some companies may object to their source code being publicly available they would retain copyright and in return, they would be rewarded with publicity and a free external review of their project. That said, we might need to allow some elements of a project to be left out of a submission. I don't expect to see Microsoft submitting the Word code but I would hope to see Accenture submitting projects from the Department of Social Security benefits system.

I expect consultancies such as Accenture, Cap Gemini and Logica to be potential sponsors. The prize itself need not be big, say, $1,000. Naturally there will be administration costs but the whole thing should be produced for under $5,000. In time I hope the contest would become a sought after trophy which actually meant something more than ISO 9000!

Naturally, the award would be presented at the ACCU conference thus bringing publicity for the ACCU, the winners, contestants and sponsors. I can hear it now... "The 2002 Logica-ACCU award for quality software engineering goes to..."

Allan Kelly

Tiny Template Tidbit

By Oliver Schoenborn

I am constantly amazed at what can be done with templates. I started programming in C around 1985 and in C++ only around 1995. Most of my work over the past 6 years has involved using C++ to the maximum of my abilities, but only for three years have I been working with templates. In this article I would like to share a very small template tidbit, something I think is really cool. I hope it can stimulate your interest in using template capabilities for your own code, or just be another "trick" to put in your bag, something I think fairly easy to remember.

I will assume in this article that you are at least familiar with a few of the STL class templates, and, though you may not have written your own class templates before, you have "seen how it's done".

Problem statement

We have an application in which a set of objects is being generated by a third-party library and stored in a container. For simplicity, I will refer to the objects as "points". Objects of this class don't necessarily have much intrinsic behavior, and are used by other classes of objects or functions for complex computations. For instance, an object (which I call a *processor* for clarity) might use the set of points to create a spline curve that best fits the points. That is, each point is simply a 2D coordinate, an aggregate of two numbers:

```
struct Point {float x,y;};
std::set<Point> points;
```

Yet, as the application evolves, it is likely that objects of the `Point` class will become more complex. Perhaps other state data, such as local temperature, contact normal, density, etc, will be "carried around" along with the coordinates of the point. In this case, the `Point` class suddenly changes from *being the data* to being an aggregate of a data object and the other state data:

```
struct Coord {float x,y;};
       // coordinates in 2D space
struct Vec {float x,y;};
       // vector in 2D space
struct Point {
   Coord coords;
       // the coordinate pair of the point
   Vec contactNormal;
       // new state data associated with point
   float temperature, density;
       // other state data...
};
std::set<Point> points;
```

The "curve fitter" processor mentioned earlier suddenly no longer works since it was designed to use the point as a 2D coordinate pair, not as an object *containing* the 2D coordinate pair as a data member.

Assuming the curve-fitter could be designed to compile in both cases, the application may evolve such that the 2D coordinates of the point are no longer a publicly accessible attribute, but rather a private data member than can only be accessed via a const "get" method (or even computed on-the-fly instead of being a data member). This could happen when `Point` starts having behavior of its own, not necessarily relevant to the curve-fitting operation of the processor. Since the latter was designed to access an attribute, it no longer accepts the new class of `Point`. For instance,

```
struct Foo1 {int bar;};
struct Foo2 {int bar() const;};
```

Both should be adequate for the curve-fitter, yet the latter would fail to compile if `Foo2` is used instead of `Foo1`, complaining that `bar()` is not an attribute (or some equivalent hard-to-decode error message, different for every compiler). Does the processor really care how the data is acquired?

If this weren't enough, the container type may change, yet the processor object (curve-fitter in this example) does not care: a `std::vector`, a `std::list`, a `std::set`, or a `your::container`, as long as the data is there, the processor should be satisfied. This is trivial to adapt to, as you will see below, but more importantly, what if the container stores pointers to the points rather than the actual points? This is likely to happen if copying `Point` is onerous and you need to avoid copying those objects in your application, so you change the container type from containing `Point` to containing `Point*`. Suddenly, the curve-fitter no longer compiles since it expects each element of the container to be an object, not a pointer to an object.

Finally, our processor (object or function) may be used in different modules, or even different applications we build, perhaps for related yet distinct application domains. Do we want to have 10 different versions of the same code, one for each possible container type and point design?

It turns out that templates can be used in C++ to make one processor (object or function) that accepts any container that uses iterators AND, much more interestingly, is able to extract the needed data from the container elements, regardless of whether the container element type

- *Is* the data, or is a pointer to the data (as opposed to containing it)
- Is an object or pointer to object *containing* the data
- Makes data available as attribute or method

Moreover, templates allow you to do so *at compile time*, and *automatically,* without your intervention. Yep, no kidding! Hopefully, you can see that the above is a fairly generic problem, unrelated to the particular application domain (curve-fitting) in which I encountered it.

Solution

Let's start simple and try to keep things simple. Assume we have a function called `fitCurve` that takes a container of points and finds the curve that best fits the set, returning the result as a `Curve` object. We are not concerned with the definition of the `Curve` class. To be generic, let's call the point class `Coord` and the container of points `Coords`:

```
#include "Curve.hh"
#include "Coord.hh"
typedef std::list<Coord> Coords;
Curve fitCurve(const Coord& coords);
```

This function can be used only to fit a curve to a set of `Coord` objects stored in a `std::list`. If you have a set of `Coord`s in a `std::set` or a `std::vector`, `fitCurve` cannot be used. Why should the interface force us to use one type, when `fitCurve` only cares that the set is ordered? If you want to use STL containers, then the containers are not related by inheritance, but they do share common interfaces for certain

operations. For instance, they all have an iterator type nested. Let's make `fitCurve` independent of the container type. We use templates to parameterize `fitCurve` on the container type, for instance

```
// note: syntax wrong but you get the idea:
template <class Container>
Curve fitCurve(const Container<Coord>& coords)
{
   // ...
   // set coord1 to first Coord in coords
   // set coord2 to second Coord in coords
   Coord c = coord1 + coord2;
   // ...
}
```

The only requirement on the element type (`Coord`) in the container is that it have an `operator+` defined. The requirement on the container is that we use `Container::iterator` instead of other access operators such as `operator[]` to access the individual elements of the container.

What are some of the advantages and disadvantages of parameterizing? I can think of the following:

Advantages:

Any kind of container, whose concrete type is known at compile time, can be given to the curve fitter. For each new type of container, the compiler takes the "function template" (i.e. a template for a function) and generates a "template function" (i.e. a function that comes from a template) — the order of those words matters. There will be a few constraints on the operations required from the container, but this is already far better than being stuck with one container class. The alternative would be to copy all the elements from your container to the container expected by `fitCurve`.

Since `fitCurve` is a function, most compilers will know to generate the template function for the given container without your having to specify it explicitly. The mere act of calling `fitCurve` with your container is enough, since the compiler knows the type of container you are using, and can generate the template function automatically.

For the same reason, you can change the type of your container at some later stage of development, without having to do anything to `fitCurve`: the compiler and your build environment will know that the old "template function" should be changed to a new one, at compilation time, and will do the change for you.

Disadvantages:

Every template function that is generated by the compiler means extra code. This is the famous "code bloat" problem of class and function templates. This is a factor to consider *only* if you plan on using several different containers in the same application (which is not necessarily the case), and memory footprint matters.

It can be difficult to predict how "well behaved" is the `Container` (template) class. Can it throw exceptions? Does its interface properly identify what is const and non-const? Can this invalidate the algorithm that accesses the `Container`?

There will always be cases where the disadvantages outweigh the advantages, but I have found the reverse to be true much more often.

The next step is to allow the container element type to be a pointer to a `Coord` as well as a `Coord`. Intuitively, we know it should be possible: the compiler knows exactly what is the type of what is contained: i.e., a `Coord` or a pointer to a `Coord`. When the element type is a pointer to `Coord`, the above piece of code would have `coord1` and `coord2` be pointers to `Coord` objects rather than actual `Coord` objects. Wouldn't it be nice if we could write, in the above code, something like:

```
const Coord& c1 = getCoord(coord1);
const Coord& c2 = getCoord(coord2);
Coord sum = c1 + c2;
```

If the element type is a `Coord`, then `getCoord(coord1)` just returns `coord1`. If on the other hand it is a `Coord*`, then `getCoord(coord1)` returns `*coord1`. Simple overloading allows us to do that:

```
const Coord& getCoord(const Coord& v)
   {return v;}
const Coord& getCoord(const Coord* v)
   {return *v;}
```

Now things are going to get a little more interesting. We want to allow the element type of the container be something else than a `Coord`, i.e. it could be a `Point` that *contains* a `Coord`. All we need is to overload `getCoord()` once more for the case where `v` is a `Point`:

```
const Coord& getCoord(const Point& v)
   {return v.coords;}
```

Of course, we want to be able to use our own class instead of `Point`. Templates come to the rescue once again:

```
template <class POINT>
const Coord& getCoord(const POINT& v)
   {return v.coords;}
```

This requires that `POINT`, the type of object contained by Container in `fitCurve`, have a publicly visible "`coords`" data member. So it's not totally generic but again, far better than being stuck with only `Point` and `Coord` objects!

Before dealing with the member "attributes vs. methods" problem, the last requirement is that the container can contain pointers to point types that contain the `Coord` data as a data member. What we need is a template overload for pointers:

```
template <class POINT>
const Coord& getCoord(const POINT* p)
   {return p->coords;}
```

It is somewhat like "partial specialization" because some of the type information is still general and unknown (`POINT` could be anything) but some is specific: it must be a *pointer* to something. However "partial specialization" specifically refers to specialization of a subset of the template parameters in a multi-parameter template definition.

Let's summarize what we have so far:

```
/// General function template
template <typename POINT> inline
const Coord& getCoord(const POINT& p)
   {return p.coords;}

/// Partial specialization for pointers
/// to things
template <class POINT>
const Coord& getCoord(const POINT* p)
   {return p->coords;}
```

```
/// Complete specialization for Coord
inline
const Coord& getCoord(const Coord& p)
   {return p;}

/// Complete specialization for pointer
/// to Coord
inline
const Coord& getCoord(const Coord* p)
   {return *p;}
```

With those four `getCoord`s put together, we have the following scenario:

If the element type of the container is anything but a `Coord` or `Coord*`, it will use the first or second definition of `getCoord`, depending on whether or not `getCoord` is called with a pointer to a `POINT` or a `POINT` itself;

Otherwise, it will use the appropriate complete specialization of `getCoord`, depending on whether or not `getCoord` is called with a pointer to a `Coord` or a `Coord` itself.

Our `fitCurve()` is now able to handle four cases of data sets:

```
Container<Coord>
Container<Coord*>
Container<Something>
Container<Something*>
```

The compiler can choose the right version of `getCoord` to use at compilation time. In the last two cases, as long as `Something` has a public data member called "`coords`", we are ok. `Container` can be any class that provides an iterator type (or whatever operations are required by the processor object or function).

The last generalization that we must do is to allow the "`coords`" to be a method instead of a data member. For instance,

```
class Point {
public:
   const Coord& coords() const
      {return coords_;}
private:
   Point coords_;
};
```

You probably know how to "point" to a method in a class. For instance, the "`coords`" method can be pointed to with a pointer of type "`const Coord& (Point::*)() const`". Given a `typedef`

```
typedef const Coord&
              (Point::* Method)() const;
```

This `typedef` defines a new type called `Method`, as a pointer to a const method of the `Point` class returning a reference to a const `Coord`. If this seems horribly difficult to read, don't worry, you're not the only one who feels that way. Pointers to functions and methods is one of the idiosyncrasies of C++. The syntax is atrociously difficult to get your mind around, but I guess it could be worse. In any case once you've done a few examples on your own it gets better. Here is an example of using a `Method` object:

```
Method mm = & Point::coords;
     // store pointer to coords method
Point point; // create a Point object
Coord coords = (point.*mm)();
     // #1: call coords method on it
Coord coords = point.coords();
     // #2: exact same thing
```

The line #2 does *exactly* the same thing as #1.

So the question is whether there is a way of doing the same thing with *data members*. Indeed, when `getCoord` accesses the `coords` field of the class, the compiler knows exactly what "`coords`" is: a data member or a method. So if we could write the general `getCoord` template like

```
template <class POINT>
const Coord& getCoord(const POINT& v) {
   return getData(v, &POINT::coords);
}

// partial specialization for pointers
template <class POINT>
const Coord& getCoord(const POINT* v) {
   return getData(*v, &POINT::coords);
}
```

and properly define two `getData` functions, one accepting a pointer to a method as second argument, and another accepting a pointer to a *data member* as second argument, we have accomplished our goal. The compiler will know which to choose without us having to tell it. In pseudo-language:

```
inline const Coord&
getData(const Point& pp, Method mm) {
   return (pp.*mm)();
}

inline const Coord&
getData(const Point& pp, ptr_dmem_Point dd) {
   return ...;
}
```

where `Method` is the typedef described earlier, `ptr_dmem_Point` stands for "pointer to data member of `Point`" and the ellipsis just means "don't know the syntax yet". It took me a bit of trial and error to find out the correct syntax, but it turns out it's not too bad: a "pointer" to a data member of type `T` for a class `Point` is "`T Point::*`". Simpler than defining a pointer to a method.

Our final solution for the `getData` is therefore, after replacing `Point` by a template parameter*:

```
template <typename POINT>
inline const Coord&
getData(const POINT& pp, Coord POINT::* dd) {
   return pp.*dd;
}

template <typename POINT>
inline const Coord&
getData(const POINT& pp,
        const Coord& (POINT::* mm)() const) {
   return (pp.*mm)();
}
```

with the four function templates for `getCoord` mentioned above.

---

* It is not currently possible, most unfortunately, to define templated typedefs. Hence writing the following is not possible, even though it would make perfect sense:

```
template <typename POINT> typedef
   const Coord& (POINT::* Method)() const;
```

Discussion

What does this give us?

Our **fitCurve** can use any container of **Coord**, **Coord***, **Something**, or **Something***, as long as the container has an "**iterator**" type defined in it, or whatever methods are required by **fitCurve** (**!=**, **++**, etc).

"**Something**" must have a "**coords**" field, which must be either a **Coord** or a "const method returning a reference to a const **Coord**."

If we make any changes to our container type, no changes are needed to **fitCurve**, as long as the changes satisfy the minimal requirements of 1.

If we make any changes to the type of object stored in the container, no changes are needed to **fitCurve**, as long as the new object is something that satisfies 2. above;

The compiler, as it recompiles the class that uses **fitCurve**, will make use of the correct **getCoord** and `getData` *without our intervention.* This is probably the biggest gain.

Since everything is inlined, there are no function calls involved: everything is resolved at compile time and is equivalent to your having typed the **coords** data member straight where it is used by the processor object or function.

We now have a far more generic processor with only 15 lines of extra code, and one that requires almost NO intervention on our part when using the fitter with a new or different type of container and contained objects. Pretty good for 15 lines of code.

For a library developer, this level of generality may still be insufficient: the member in **POINT** must be called "**coords**". This is similar to STL containers which all have a member called "**iterator**" that defines an iterator appropriate for the container. To make it completely general, we would need a way of allowing the user to specify that they want something other than "**coords**" to be used by **getCoord()**. One work-around would be to define further complete specializations for your point class, using the appropriate member:

```
template <>
inline const Coord&
getCoord(const YourPoint& v) {
   return getData(v, &YourPoint::yourCoords);
}
```

and similarly for the pointer version. This is tedious to type and error-prone, but only a macro could be used to simplify this to a one-liner:

```
#define GET_COORD(YourPoint, yourData) \
template <> \
   inline const Coord& \
   getCoord(const YourPoint& v) { \
      return getData(v, &YourPoint::yourData);\
} \
template <> \
   inline const Coord& \
   getCoord(const YourPoint* v) { \
      return getData(*v, &YourPoint::yourData);\
   }
```

which would allow you to type

```
GET_COORD(YourPointClass, yourCoordMember)
```

in your source file to get the specialization. The good news is that if you forget to define this, you get a compile time error. The bad news is that you have to define it in a place where it is visible to the compiler by the time your processor object or function calls `getCoord`, which is likely to increase the coupling between header files. The only other way that I can think of is to derive your class and provide a "**coords**" alias for the member, but this will be useful only if you have any control over the type of object stored in the container. For instance, you could derive `YourPointClass` as

```
struct PointDerived: YourPointClass {
   Coord& coords;
      // alias for member in YourPointClass
   PointDerived: coords(yourCoordMember) {}
};
```

and use a container of **PointDerived** objects instead of a container of **YourPointClass** objects. This only works if you have control over the element type of the container.

It is interesting to notice that templates are necessary in C++ only because C++ is such a strongly typed language. Consider Python, which is essentially untyped: classes are not differentiated by type, only by name. Hence a List class in Python that takes as argument the name of an object to store in the list, can have that object be any class of object. The idea of templates in Python is implicit in some ways, and unnecessary in others. There are probably many other languages where this applies. The strong-typing nature of C++ requires extra typing (sorry for pun), but it allows the compiler to know much more about the data being handled, and therefore to do much more stringent error-checking and optimizations not possible in weakly typed languages. Templates in C++ are necessary so C++ can support very useful features found in other high-level languages in a strongly-typed and optimized context.

Note: At the time of this writing, GNU g++ 3.0 (on SGI) has problems picking the correct **getCoord**: it seems to get confused by the pointer overloads. The code compiles and works perfectly, however, with SGI's MipsPro C++ compiler 7.3.1.

Summary

I showed how a processor (object or function) that uses data from a set of objects can be generalized with a very small number of lines of code, using templates and template specialization, to support application evolution, without requiring you to adapt the processor. More specifically, your processor object or function is able to extract the needed data from the container elements, regardless of whether the container element type

- *Is* the data, or is a pointer to the data (as opposed to containing it)
- Is an object or pointer to object *containing* the data
- Makes data available as *attribute OR method*

Moreover, the compiler does so for you, automatically, without requiring your intervention. I'll gladly reply via email to suggestions and ideas. Sincere thanks to Phil Bass for his critical comments and ideas.

Oliver Schoenborn
Oliver.Schoenborn@utoronto.ca

References

Bjarne Stroustrup, *C++ Programming Language, third edition*
Andrei Alexandrescu, *Modern C++ Design*
Mark Lutz, *Python Programming, second edition*

Introduction to WOC: Abstracting OpenGL 3-D Model Definition and Rendering with C++.

By Steve White

This article introduces a library of C++ classes which I have named Windows/OpenGL Classes, or WOC for short. WOC leverages the substantial functionality of OpenGL and hides its complexity behind a hierarchy of user-derivable base classes and leaf classes. In addition, a basic Win32 application-and-windowing framework is offered, as well as some very flexible value-generating and member-function-calling class templates whose purpose is to constitute, and relay values around, 'virtual circuits' for the purposes of either animation or geometry generation.

Casual users of WOC, and anyone tempted by the instant gratification of some pretty graphics pictures, are welcome to visit the WOC section of my website at `www.barkbark.demon.co.uk/woc`. The WOC header and implementation files (127kb zipped) can be found in the same place. This article is aimed at those interested in WOC 'under the hood' but it also includes a first tutorial on its use.

What do I have to know to use WOC?

Because WOC hides the OpenGL API, you don't need to know OpenGL. It helps to have heard of, and to be able to visualise, 3-D cartesian coordinate space and to have the gist of the basic translation, rotation and scaling transformations, particularly the significance of applying either translation or rotation before the other. With the exception of the animation templates, only a very basic knowledge of C++ is required to use WOC; it is in the ballpark of rudimentary MFC. Use of the animation templates is optional but more demanding as it requires a good knowledge of the generic programming techniques of modern C++.

What, briefly, is OpenGL?

OpenGL (Open Graphics Library) was developed by Silicon Graphics and it is a hardware-independent specification of a graphics programming interface. Although windowing tasks and user input are not part of the OpenGL specification, implementations for different platforms all have a standard core of functionality and are packaged with the OpenGL Utility Library (GLU) which does offer a common abstraction of windowing support hiding a specific implementation for each platform. GLU is not a perfect solution and on Win32 I prefer to use the Win32 Extensions.

What does WOC offer?

OpenGL's interface is at the level of geometric primitives – points, lines and polygons – and no higher. WOC also allows the geometry of a 3-D model to be defined at this level, either manually or generated automatically, but also introduces types representing higher-level elements, defined once and referenced many times, in model and scene hierarchies created by the user. WOC controls OpenGL's state transparently and is responsible for managing, transforming and rendering the user's scene and animations. Also, for Win32, the basics of registering and creating windows, a message-loop, window procedure, and creating and managing an OpenGL rendering context and default animated model are all taken care of by WOC upon the instantiation of, in the simplest case, a single Application class object.

The WOC Class Model

At the bottom of WOC's geometry class hierarchy [*class diagram on following page*] is the **`VectorT`** template. This represents a vector, or one-dimensional matrix. A vector has magnitude and a direction in space. **`VectorT`** is used as a set of either three or four scalar values which together represent a vector-like concept. So, it can be used to represent any of: a set of homogeneous or non-homogeneous coordinates in three-space (i.e. a point); a free vector in three-space (e.g. a normal vector); ray rotations, translations and scalings. The class diagrams shown in the figures are from Rational Rose and give the class names without their generated 'C-' prefixes. I will do the same in this discussion. WOC specialises **`VectorT`** with the **`GLfloat`** type and typedefs the result to **`Vector`**. **`GLfloat`** is itself a typedef for the built-in type **`float`**. At a similarly low level the **`UV`** class represents a set of floating-point texture coordinates (**`u`** and **`v`**, corresponding to the x and y directions respectively) which identify a point on a texture map (an image). A **`Model`** instance is a collection of all the 3-D points (**`Vector`**), lighting normals (**`Vector`**) and texture coordinates (**`UV`**) from which its polygons are constructed. This repository of geometric resources is then referenced by **`Triangle`** instances so that the points, normals and texture coordinates can be re-used and mixed and matched as required. A logical collection of **`Triangle`**s is placed into a **`Geometry`** – e.g. the triangles defining the surface of a sphere – also with re-use in mind. **`Geometries`** are collected and managed by the **`Model`**, but are referenced by the **`Model`**'s **`Group`**s. I will say more about material and transformation in due course, but a **`Group`** applies a **`Material`** and a set of **`Transformation`**s to a **`Geometry`** so that the same **`Geometry`** (e.g. our sphere) may be stretched, scaled, translated, textured or coloured many times depending on the properties of each **`Group`** which references it.

As you can see, WOC's abstraction of a 3-D model is factored into several classes. It is possible to construct a model at either low, medium or high level as desired. The lowest level involves defining points, normals and texture coordinates and then defining **`Triangle`**s in terms of the returned indices of the points. Alternatively, **`Triangle`**s can be constructed with their points, normals and texture coordinates as parameters, with the option to re-use duplicate existing points, normals, etc, within a threshold of similarity. Normals are optional: they may be supplied or alternatively WOC will on request calculate face or vertex normals. Finally, the highest (and easiest) level of model definition is afforded by the polymorphic model-loader classes. You simply point these at a **`Model`** instance and, together with some optional parameters, instruct them to load. Several model-loaders (cube, grid, tetrahedron, sphere, tube) are built into WOC but you can derive your own. The sphere loader re-uses existing points it has already placed in the **`Model`**'s repository, because the spherical to cartesian coordinate conversion formula it uses generates a large number of proximate points at the poles. There is also a model-

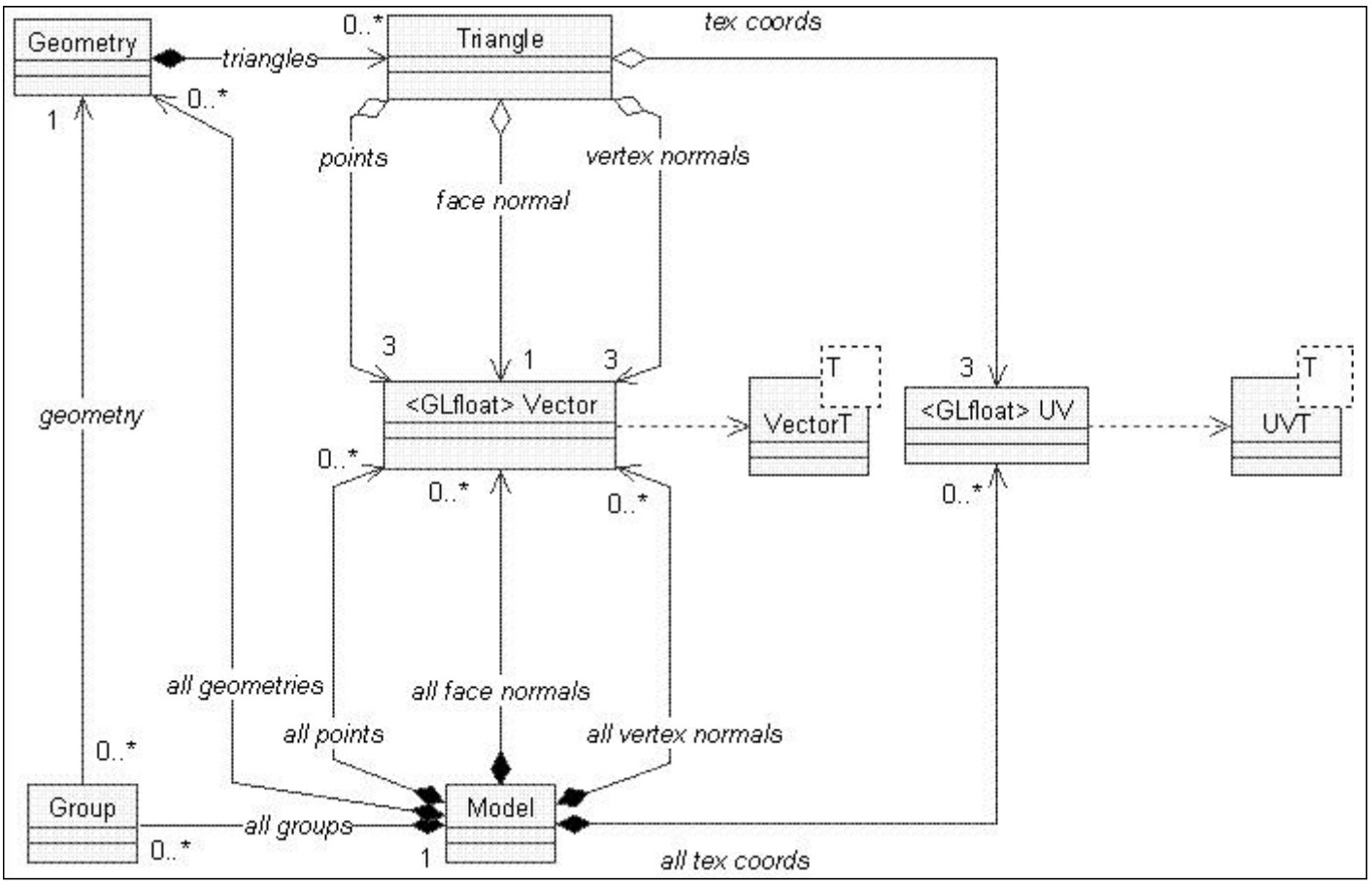

Figure 1: Geometry Classes

loader specialised for reading from a disk file in Wavefront .OBJ format, making it childsplay to build an .OBJ model-viewer with WOC. A more **Model**-centric view of the classes already mentioned can be found in the WOC Class Reference on my website; I won't reproduce it here.

Let's look at **Transformation**(s) next.

Earlier I mentioned that a **Group** applies a set of **Transformation**s to a **Geometry**. A **Transformation** is an abstract base class for **Translation**, **Scaling** and two types of **Rotation**. WOC has a class which is a collection of **Transformation**-derived types, and is known as a **Transformations**. A **Transformations**' elements are applied in the order in which they were added because matrix multiplication is not generally commutative. As can be seen in Figure 2, an **OGLWnd** also has its own **Transformations** instance. An **OGLWnd** is a window with an OpenGL Rendering Context in its client area, and its **Translations** and **Rotations** are generally sufficient to give the correct view (or 'camera angle') onto the scene as a whole, although **Scaling**s can be applied if required. An **OGLWnd** owns a collection of **Model**s and once the scene itself has been transformed, each **Model** in the scene is transformed and rendered and that process in turn involves transforming and rendering each **Model**s' **Group**s. A **Model**'s **Transformations** collection (not shown in Figure 2) exists so that transformations common to all **Group**s in a **Model** can be factored up to the **Model**. Immediately before transforming a **Model** or a **Group**, the current transformation matrix is pushed onto OpenGL's matrix stack and later popped once the **Model** or **Group** has been rendered. This ensures the current transformation always keeps in step with the inorder walk of the scene tree. It probably also bears mentioning that **Group**s themselves may have a further collection of (sub)**Group**s in the way that **Model**s do. This allows the definition of a scene tree to go to any depth and also provides for **Group** nodes to contain only a **Transformations** collection without a **Geometry** nor a **Material**; this provides for further factoring out of **Transformations** common to child **Group**s. The OGLWnd's **Transformations** are accessed by the class's built-in mouse interface which allows the user to translate, rotate and zoom the scene. Mouse sensitivity along with a host of other settings are available from the OGLWnd's context menu.

Materials are another part of a **Model**'s repository of resources which are re-usable by its **Group**s. A **Material** is essentially a definition of how the triangles in the **Group** should reflect the colour components of the lights illuminating them. A number of stock **Material** definitions are built into WOC (e.g. emerald, ruby, pearl, brass, bronze, red enamel, various colours of plastic and rubber to name but a few) so the casual WOC user need never get into the technicalities. The definition of lights is taken care of by the **Light** class which wraps OpenGL's light-related APIs. A **Light** contains a **Model** instance which defines the appearance of a **Light** should it need to be represented visually. By default a **Light**'s **Model** is loaded with low-resolution sphere geometry but you can derive from **Light** and change the **Model** used. **Light**s also have their own **Transformations** collection so that they can be placed anywhere, or even animated.

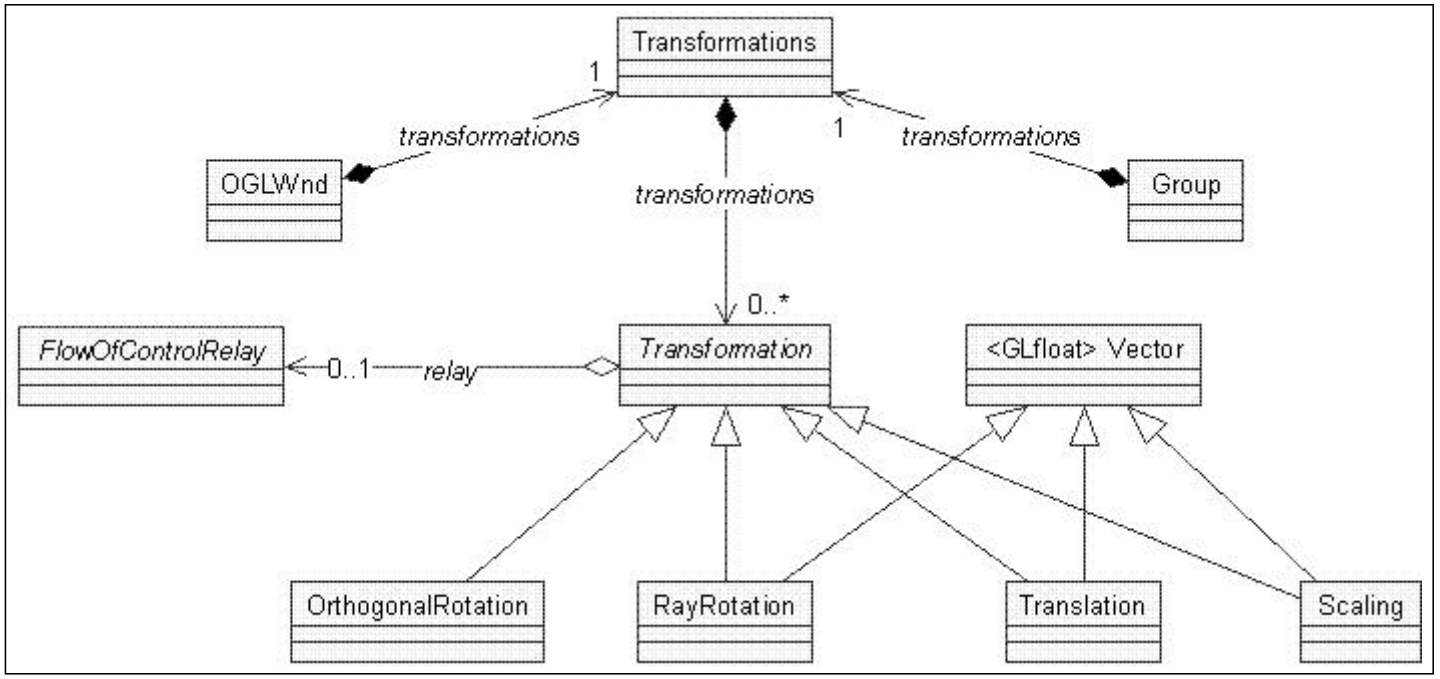

Figure 2: Transformation Classes

The remaining corners of the WOC class model can be explored by checking out the WOC Class Reference on my website.

WOC Tutorial One: A Skeleton Project

To follow along with this tutorial you will need to download the WOC header and implementation files from my website. I recommend that you unzip them into a folder named woc and locate it at the same level as (i.e. a sibling of) the project folders which use it. This is because the projects look for the WOC files at the path: ..\woc\ as we'll see later.

About the tutorial

If you like WOC and find it useful then you may want to use it more than once - perhaps even lots of times. In that case it's nice to have a skeleton or template project to model new WOC projects on. That's where the Skeleton Project (WOCSkeleton) comes in. This tutorial shows you how to integrate the WOC source files into a Visual C++ project, but you can then save the project and re-use it as a starting-point for new projects. You can either follow along with the steps or just download the files of the completed project. Of course you don't have to use a skeleton project if you don't want to, but you'll find it more convenient than following these steps each time you make a new project. For those who are unable to use Visual C++, or who prefer not to, WOC will build under the GNU Compiler Collection (gcc/g++). I have more to say about this on my website.

The steps

Step 1

Launch Visual C++ and use the Win32 Application wizard to create a new project named WocSkeleton. Locate the project folder as a sibling of the woc folder containing the WOC header and implementation files. The steps which follow apply to the 'Hello World' option (on Step 1 of the Win32 Application Wizard), but feel free to choose one of the others if you're happy to add the appropriate files, code and resources on your own to make a minimal Win32 application.

Step 2

Open the file stdafx.h for editing. There is usually a comment near the end of the file indicating where to place your own headers:

```
// TODO: reference additional headers
your program requires here
```

Even if you don't have this comment, just find a suitable place near the end of the file before the close of the include guard and type this:

```
#include "woc.h" // directory path set in
project settings.
using namespace woc;
```

This is an include of the main WOC header file which in turn includes several other WOC header files. Together these files contain the declarations of all of WOC's types, and some complete definitions. Because I have opted for a using directive, and because stdafx.h is included by the other source files in the project, all names in the woc namespace will now be visible in the global namespace throughout the project without further qualification. If you don't like this, you can omit the using directive and explicitly qualify. At this stage the project doesn't yet know where to find the woc.h file - that's done in step 3. However, you can opt to hard-code the path to the file here (even a relative path) and skip step 3. It's up to you.

Step 3

If a project includes a lot of header files from the same folder, and the name or location of that folder may change, then you wouldn't want to have to edit the path in every #include. Although WOC's include structure presently obviates the explicit including of more than one header file, that may not always be the case. So, if you followed step 2 to the letter then now you'll need to let the project know where to look for additional header files, specifically woc.h. You could make this

setting for the whole of Visual C++ by adding a new include file directory on the Directories tab of the Options dialog (Tools/Options... menu), but I prefer to make the setting apply only to the project at hand so that it will easily transfer between Visual C++ installations. To do this, choose Project/Settings..., choose Settings For: All Configurations, choose the C/C++ tab, Category: Preprocessor, and in the Additional include directories: edit box, type:

```
..\woc\
```

This is correct for the case where the `woc` folder is a sibling of the new project's folder. You may choose a different arrangement but, if you do, then you should edit the above include directory path to match. Note that the Additional include directories: edit box may contain more than one path, comma-separated. There is one more change to make whilst you're editing the project settings. Use of the dynamic_cast operator requires run-time type information which is enabled with the /GR compiler switch. So, choose Settings For: All Configurations, the C/C++ tab, Category: C++ Language, and check the Enable Run-Time Type Information (RTTI) checkbox.

Step 4

The project will now compile, but so that it will also link when we come to using the WOC classes, you'll need to add the implementation file to the project's Source Files folder on the FileView tab of the Workspace pane. Right-click the Source Files folder, choose Add Files to Folder... from the context menu, navigate to the woc folder and choose the file woc.cpp. Whilst you're at the FileView tab you can also add all the WOC header files (`woc.h` and all the others you'll find in the folder) to the Header Files folder so that all the WOC classes will appear on the ClassView tab.

Step 5

One final step before the project will link is to reference the static library files for OpenGL and the Win32 Common Controls. To do this, choose Project/Settings..., choose Settings For: All Configurations, choose the Link tab, Category: General, and in the Object/library modules: edit box, add:

```
opengl32.lib glu32.lib glaux.lib comctl32.lib
```

Now the project will build. Just check that it does.

Step 6

At present the project is using no WOC features and, if you run it, it will behave as it did when it was first generated by the wizard. Now we need to remove most of the wizard-generated Windows code and replace it with a small amout of WOC code. Open the file WocSkeleton.cpp for editing and delete everything from it except the #include directives and the WinMain function. Next, delete all the code from the body of the WinMain function and type this in its place:

```
// Perform application initialization:
if (!theApp.InitInstance(hInstance,
                         nCmdShow,
                         IDC_WOCSKELETON,
                         IDI_WOCSKELETON,
                         IDI_SMALL,
                         NULL,
                         IDS_APP_TITLE))
{ return FALSE; }

return
 theApp.MessageLoop((LPCTSTR)IDC_WOCSKELETON);
```

You may be wondering about the identifier theApp - where is it declared? Nowhere as yet, so add the following declaration after the includes but before WinMain:

```
// The one and only application object.
CWocApp < CWocFrameWnd < CWocOGLWnd > >
     theApp;
```

The meaning of this code is that we are declaring an identifier named `theApp` which is of type `CWocApp`. This is a class template whose single template parameter specifies the type of window to use for the application's main window (defaults to `CWocFrameWnd`). The parameter can be any `CWocWnd`-derived class so long as it implements a CreateFrame method as `CWocFrameWnd` and `CWocOGLWnd` do. The `CWocFrameWnd` class is another class template whose single parameter specifies the type of the view window it may be required to use to overlay the client area of the frame window. The parameter must either be `CWocWnd` (the default) or a class derived from it. Incidentally, the constructor of the frame window object takes a `BOOL` parameter, defaulting to `TRUE`, indicating whether or not the frame window is required to create a view. If this parameter is `FALSE` then the view type is ignored. If we wanted to override the default of creating a view then we have to wait until the application object has created the frame window then call `SetCreateView(FALSE)` on the frame window at any time, but most logically in an overriden `OnCreate` handler.

Step 7

Now you can build and run the sample, so let's leave further code editing until the next step whilst we look at some of the default features of the classes. The code that earlier I directed you to insert specifies the view of the main frame window to be an OpenGL rendering context window (`CWocOGLWnd`). When you run the sample you'll see the default behaviour of the `CWocOGLWnd` class. Firstly, a default 3-D model is displayed which is lit and rotating and its normals are shown. How this happens is that an (overridable) initialiser function in the OpenGL window class (the member is `CWocOGLWnd::InitialiseGL` if you want to take a look at it) creates a new model object, loads some geometry and face normals into it and then calls a method on the model to require it to show its normals. The model then makes some changes to the OpenGL state to reflect its requirements and, since normals now exist, the model requests the view class to activate lighting and `GL_LIGHT0`. The view class also defaults to rotating the scene a small amount on a timer which fires every few milliseconds. The `CWocOGLWnd` class has two significant features: a mouse interface to manipulate the view transformations, and a Properties Dialog.

Mouse manipulation is a mode which you can toggle into and out of by holding down Ctrl and right-mouse clicking inside the view. When you're in mouse manipulating mode the mouse cursor will disappear and you can manipulate the scene in several ways, even whilst model animation is taking place, by moving the mouse with various combinations of the mouse buttons depressed. With no buttons depressed the view is rotated about the X and Y axes; the left button causes rotation about the Z axis; the right button causes zooming in and out; and both mouse buttons depressed together causes the view to pan.

Getting the OpenGL Window Properties Dialog to display can be done either programmatically by calling **CWocOGLWnd::PropertiesDialogDoModeless** or by the user double-clicking the *right* mouse button anywhere inside an OpenGL Window. For a full explanation of all the controls on the OpenGL Window Properties Dialog together with the theory behind them, please see the documentation for the **CWocOGLWndPropertiesDialog** class (nested within the **CWocOGLWnd** class) in the WOC Class Reference.

Step 8

Now to reactivate the application's main menu. The project wizard created an About dialog box resource along with a dialog procedure and command handler to display the dialog. Earlier we deleted that code but we still have the dialog resource and, as you'll see, it's very easy to add a handler to your project to handle the menu commands and to create and display the dialog. First, in order to control the handling of specific commands, we need to override the command handling functionality in the default frame window class **CWocFrameWnd**. Command handlers exist in all of WOC's window classes: standard windows, frame windows and consequently any window used as a view. This means that you can either derive your own frame window and add a command handler to it or do the same with a view window; the only difference being that frame windows get to handle commands before their views. In this case, because the purposes of the menu commands being handled are 1. to close the application and 2. to display the application's About box, the most appropriate place to handle these commands is in the application's main frame window. The plan then is to derive a class from the existing **CWocFrameWnd** template class and implement the virtual **OnCommand** method on the derived class. Type the following code into **WocSkeleton.cpp** immediately before your declaration of **theApp**:

```
template < class _TyView = CWocOGLWnd > class
CWocSkeletonFrameWnd :
          public CWocFrameWnd < _TyView >
{
public:
  CWocSkeletonFrameWnd
     (BOOL nCreateView = TRUE) :
      CWocFrameWnd < _TyView >(nCreateView){};
  virtual ~CWocSkeletonFrameWnd(){};

  virtual BOOL
        OnMenuOrAcceleratorCommand (UINT nId)
  {
    switch (nId)
    {
      case IDM_EXIT :
        return theApp.Exit();
      case IDM_ABOUT :
      {
        CWocDialog dlgAbout(IDD_ABOUTBOX,
                                        this);
        dlgAbout.DoModal();
        break;
      }
      default:
        return CWocFrameWnd < _TyView
           >::OnMenuOrAcceleratorCommand(nId);
    }
    return 0; // indicate that the message
              // has been handled.
  }
};
```

So what are the handlers doing? The **IDM_EXIT** handler is simply calling a method on your application object to destroy the main window and thus quit the application. The **IDM_ABOUT** handler makes use of the **CWocDialog** class which in this case needs no specialisation as it handles **IDOK** and **IDCANCEL** straight out of the box. The arguments passed to the constructor of **CWocDialog** are: the dialog's template resource ID, and a pointer to the window object which owns the dialog.

Finally, we have to amend the type of our application object as its main window is no longer the base frame window class but rather the class we've just defined. So replace your **theApp** declaration with this line:

```
CWocApp < CWocSkeletonFrameWnd < > > theApp;
```

And that's it. If you build and run now you'll find that your menu works again. The **WocSkeleton** is referred-to by further tutorials on my website as they all use it as a starting point. For this reason I suggest you save your project and put it aside if you've been following along, or just download the **WOCSkeleton** project files if you prefer.

And That's All We Have Time For

I hope this introduction to WOC has been of some interest. Please visit my website if you wish to follow the remaining four tutorials and learn how to define your own models in WOC. There is also a gallery of sample demos built using WOC at:

`www.barkbark.demon.co.uk/graphicssamples`

Naturally any feedback regarding WOC, good or bad, is welcome via email.

Steve White
swhite@barkbark.demon.co.uk

Bibliography

The best introduction to OpenGL is the 'Red Book':
Woo, M., J. Neider, and T Davis: *OpenGL Programming Guide*, Addison Wesley.

The standard computer graphics canon is:
Foley, J., A. van Dam, et al: *Computer Graphics: Principles and Practice*, Addison Wesley.

Aspiring 3-D game programmers are directed to the excellent:
Abrash, M: *Graphics Programming Black Book Special Edition*, Coriolis Group Books.

What is Boost?

by Björn Karlsson

Welcome to an introduction to Boost, and especially the community behind the Boost libraries. Although many of the libraries are amazing, the people that are Boost are the reason for the rapidly growing interest. For those of you that aren't yet Boosters, let me give you some fast facts on Boost:

Boost is a collection of class libraries for C++. The libraries range from math to threading, built on Standard C++ and highly portable. Moreover, it is also a community of expert C++ programmers, all contributing by taking part in reviews, submitting libraries, helping users etceteras.

There are currently 35 Boost libraries, and several of them are proposed for inclusion in the next C++ Standard. Beyond being an excellent source of inspiration, top-quality libraries and interesting discussions, Boost is also a place to learn. C++ Library design, design patterns and idioms are honed to perfection in the peer-reviewed Boost community.

The libraries

Although reading through a list of libraries with short descriptions may seem dull to some, I assure you that it's worth your time: You will find at least five libraries that would be very useful to you, no matter what type of application you're currently working on.

any
Safe, generic container for single values of different value types, from Kevlin Henney.

array
STL compliant container wrapper for arrays of constant size, from Nicolai Josuttis.

bind and mem_fn
Generalized binders for function/object/pointers and member functions, from Peter Dimov.

call_traits
Defines types for passing parameters, from John Maddock, Howard Hinnant, et al.

compatibility
Help for non-conforming standard libraries, from Ralf Grosse-Kunstleve and Jens Maurer.

compose
Functional composition adapters for the STL, from Nicolai Josuttis.

compressed_pair
Empty member optimization, from John Maddock, Howard Hinnant, et al.

concept check
Tools for generic programming, from Jeremy Siek.

config
Helps boost library developers adapt to compiler idiosyncrasies; not intended for library users.

conversion
Numeric, polymorphic, and lexical casts, from Dave Abrahams and Kevlin Henney.

crc
Cyclic Redundancy Code, from Daryle Walker.

function
Function object wrappers for deferred calls or callbacks, from Doug Gregor.

functional
Enhanced function object adaptors, from Mark Rodgers.

graph
Generic graph components and algorithms, from Jeremy Siek and a University of Notre Dame team.

integer
Headers to ease dealing with integral types.

iterator adaptors
Adapt a base type into a standard conforming iterator, and more, from Dave Abrahams, Jeremy Siek, and John Potter.

math
Several contributions in the domain of mathematics, from various authors.

math/common_factor
Greatest common divisor and least common multiple, from Daryle Walker.

math/octonion
Octonions, from Hubert Holin

math/quaterion
Quaterions, from Hubert Holin.

math/special_functions
Mathematical special functions such as atanh, sinc, and sinhc, from Hubert Holin.

operators
Templates ease arithmetic classes and iterators, from Dave Abrahams and Jeremy Siek.

pool
Memory pool management, from Steve Cleary.

preprocessor
Preprocessor metaprogramming tools including repetition and recursion, from Vesa Karvonen.

property map
Concepts defining interfaces which map key objects to value objects, from Jeremy Siek.

python
Reflects C++ classes and functions into Python, from Dave Abrahams.

random
A complete system for random number generation, from Jens Maurer.

rational
A rational number class, from Paul Moore.

regex
Regular expression library, from John Maddock.

smart_ptr
Four smart pointer classes, from Greg Colvin and Beman Dawes.

static_assert
Static assertions (compile time assertions), from John Maddock.

test
Support for program testing and execution, from Beman Dawes.

thread
Portable C++ multi-threading, from William Kempf.

timer
Event timer, progress timer, and progress display classes, from Beman Dawes.

tokenizer
Break of a string or other character sequence into a series of tokens, from John Bandela.

tuple
Ease definition of functions returning multiple values, and more, from Jaakko Järvi.

type_traits

Templates for fundamental properties of types, from John Maddock, Steve Cleary, et al.

utility

Class noncopyable plus checked_delete(), checked_array_delete(), next(), prior() function templates, plus base-from-member idiom, from Dave Abrahams and others.

Behind the libraries

To understand more about the mysterious powers that summons so many of the very best C++ programmers, I interviewed three of the original Boosters; Beman Dawes, David Abrahams and Jens Maurer. They are important people for the C++ community, and I want to thank them for taking the time to give us their answers.

Q: What is the essence of Boost's importance?

Beman: It has created a community of C++ library users and developers.

Dave: Boost is one of the only communities working on the process and practice of library design. In today's professional software development world, it can be hard to make the case for long-term investment in reusable components. Developers are (often rightly) expected to do the simplest thing that could possibly work, under the assumption that the generalization won't be needed. As the problems they need to solve become more complex, however, they need library components that can help them keep their solutions simple. The C++ standard library goes some distance towards filling that role, but programmers will continue to need more than it provides. What should be the design, documentation, and coding practices for the libraries that programmers need? Boost's emergent collaborative process provides one answer.

Jens: First and foremost, Boost gives examples how C++ libraries should be designed. There are lots of libraries available elsewhere whose design pre-dates the ISO C++ standard. These libraries often fail to exploit the potential of compile-time evaluation (i.e. templates) and thus miss type checking and optimization opportunities. Furthermore, these libraries are occasionally difficult to extend or poorly documented.

Second, Boost produces ready-to-use, well-documented, high-quality libraries from the practice of programming. The license requirements allow the use of Boost libraries in commercial and non-commercial projects free of charge, thereby helping them to produce better programs.

Third, Boost is a test-bed for components that may end up in a future revised C++ standard. I am proud to be part of a respected effort with worldwide recognition.

Fourth, Boost attracts some of the best C++ programmers in the world. Its open discussions and peer-reviews cause every participant to learn quite a lot.

Q: Which parts of Boost do you consider most important for inclusion in the Standard?

Jens: The small things: Smart pointers, utility.hpp, "functional", probably "any" and "function"

Q: When the next C++ standard is final, what do you think will happen to Boost?

Beman: The potential for Boost libraries goes way beyond the C++ Standards effort. Standardization of some of the Boost libraries may help validate the Boost concept, but just plain user interest is really important too. As long as users ask for new libraries, and developers are willing to write them, Boost will grow and prosper.

Dave: Yes, I'm sure it will continue. Boost's value to the C++ community goes well beyond standardization.

Q: The members of Boost are some of the most talented and experienced C++ developers around. What is the reason that all this talent is present in Boost?

Dave: I think most talented programmers - in any language - love to collaborate and are dying to work with others of similar caliber. A really good programmer craves thoughtful and careful review of his code, useful feedback and ideas about design alternatives. That sort of environment can be hard to find, especially If you are at the top of your game in your daily work. The process we've established at Boost and the highly professional tone of the discussion give the best programmers what they crave, and gives others a path to reaching that level.

Jens: Boost has an open discussion policy. Its peer-review atmosphere, well established in the scientific community, fosters learning for every participant. Plus, we've had the luck to distribute very little off-topic e-mail in the mailing list so far, and we as the moderators try our best so that it will stay that way. Also, Boost library authors and maintainers are asked to acknowledge contributions, another well-established behaviour in the scientific and open-source communities. This gives contributors a permanent record to put pride in.

Q: Why should C++ users start using the Boost libraries?

Dave: I can think of four reasons:

0. Would you rather write it yourself? The Boost libraries are time-savers and code-savers. You'll spend less time developing your application and have less code to maintain if you can re-use the appropriate Boost library.

1. Quality. Most of the boost libraries have had a more thorough review by more talented and critical eyes than I've ever seen in a commercial environment.

2. Open Source. If you absolutely need to change the library to suit your own application, the source is all there.

3. Tests. No boost libraries are accepted without a comprehensive test suite, and all are tested against a wide variety of platforms and compilers.

Q: Do you see an upper limit to the number of libraries in Boost?

Jens: No. We may have to adjust our organization to cope, though.

Q: In what areas do you think Boost will grow in the near future?

Dave: Several interesting and ambitious libraries seem to be on the immediate horizon. There is a template metaprogramming library that brings STL-like algorithms and functional programming idioms to compile-time computation, we have an effort to develop next-generation linear algebra and matrix computation, and the developers of the Spirit parsing framework have expressed an intention to "Boostify" soon.

Q: How (and to what) do you think that Boost will develop?

Beman: More of the same; more libraries, more interesting discussions. It all depends on the C++ programmers who are members of Boost, but there is no sign of interest leveling off. Every measure we have continues to show strong growth. Quality of new libraries being submitted seems strong. The future looks bright!

Dave: One of the best things about Boost is the way that the participants generate its direction, and only when enough people are sufficiently committed to pull it off. For example, in the past week a group of dedicated Boosters who straddle the developer/user line set up a separate mailing list and a Wiki for Boost users. I am convinced that this couldn't have happened if the boost moderators had tried to spearhead the effort.

Q: As Boost grows; do you plan to organize differently?

Dave: As shown by the spontaneous creation of the users list and Wiki, when Boost begins to grow beyond its capacity, I expect it to reorganize. We have been discussing different ways to enable better collaboration as this happens: we want to have the required knowledge when the time comes, but I think we've learned not to try to plan too far in advance.

Jens: We may have separate moderators for library sub-collections, e.g. for the maths and numerics libraries. We may have to split the mailing list further, though there are good arguments against doing so. Namespaces in C++ and directories in the file system are hierarchical, so we have all the tools we need to effectively sub-divide organizationally.

Q: When a library is submitted to Boost, the members decide whether it should be accepted. Will that still be possible if the number of members keep increasing?

Jens: I think so. Not all members cast a vote on every submission, and it is assumed that members casting a vote have had a look at the library before. Also, the review manager is entitled to disregard votes that are cast without observing the review guidelines.

Q: Does Boost ever organize meetings or conferences?

Jens: Boost meets every half a year on the Sunday afternoon before the ISO C++ meeting starts, at the same place than the ISO C++ meeting.

Q: Boost as an independent organization has become a "waterhole" for developers from all over the world. Is this a part of the intent for Boost, or is it just a lucky side effect?

Dave: Definitely a lucky side effect. I guess that's what happens when you provide services and code that people want.

Jens: For me, it's part of the intent.

[Author's note: Select the answer that you like!]

Q: If companies hesitate to use Boost because it is free, and therefore don't offer support agreements etc, what would be your advice to them?

Beman: If the company doesn't normally have support agreements for software libraries of about the same scope as Boost, they probably don't need a support agreement for Boost either. If the company does want a support agreement, they should ask their usual support contractors for Boost support. I know several folks at one of the support companies, and they would be perfectly capable of supporting the Boost libraries.

Dave: Re-use first; rewrite if necessary. In other words, start by using the boost libraries to achieve short time-to-market with high quality. If independence from open source is important to your long-term mission, you can always go back and replace boost components with internally developed and maintained ones.

Jens: Other open-source projects (e.g. Linux) have shown that when there's money being offered, someone shows up offering support for the project. I'm confident that some of Boost's members would take a support contract when one was offered.

Q: Boost adds value to users but also aims to impact the next C++ standard. Which one is more important?

Dave: So far, making it possible for users to cover more domains with cross-platform code has been our main focus. Although many libraries seem to fit the bill for addition to the standard, a few are less likely candidates. We have also made occasional concessions to nonconforming compilers and compilers that don't implement all of the optimizations one would like, because these things have practical importance for users. One would not expect to see such concessions to appear in the C++ standard library, however. As the C++ committee prepares to start considering what the next version of C++ will look like, standardization will likely increase in importance.

Beman: The issues really go hand-in-hand. Well-developed, widely used libraries benefit both users and the C++ Standard. Users get the use of the libraries, while the standards committee gets "existing-practice" to standardize, and that is what standards committees do best. That leads to a better standard, which in turn benefits users. The circle is closed.

Jens: I agree, there isn't a clear separation between these two aims: The next C++ standard is also aimed at providing added value to users. For me, adding value to users is the major aim of Boost; be it directly (and rather short-term) or indirectly by impacting the next C++ standard. Formal standardization has the advantage of broad recognition even in non-technical circles, but takes longer to produce results.

Becoming a Booster

There are two different mailing lists of interest for Boosters – the Boost and Boost-Users. The Boost-Users list was created only a few months ago, to accommodate the needs of the many users of the Boost libraries. The list is dedicated to questions and answers for Boost library usage. The original Boost list on the other hand deals with library design, submissions, and reviews etc. The list is rather high-volume, and very suitable for those interested in library design in general, or Boost libraries in particular. Many Boost members subscribe to both lists. There is also the new Boost Wiki web (not officially maintained by Boost developers), which is another promising venture for spreading the word and increasing the discussion about Boost.

Of course, you don't have to be a member of the mailing lists to use the Boost libraries, but it is nice to know that there is a strong commitment to helping users of the libraries. The tone is always friendly, and the level of expertise is outstanding.

I find it very rewarding to follow the discussions even on libraries that I'm yet not using, because there is always new things to learn and new ideas being tossed around.

Literature

Just recently, Jeremy Siek, Lie-Quan Lee and Andrew Lumsdaine finished "The Boost Graph Library User Guide and Reference Manual", a book about the Boost Graph Library. I highly recommend it! (ISBN 0201729148).

Although there are references to Boost in many books and articles, there are no books other than the BGL book available. I'm confident this will change in the near future, as Boost is being recognized for excellent libraries, strong standards committee presence and a rapidly growing user base.

Meanwhile, be sure to read the top computer magazines – several of the Boost libraries and the techniques used in the libraries are being featured in articles.

Björn Karlsson
Bjorn.Karlsson@readsoft.com

Boost links

Boost website, `http://www.boost.org`
Boost mailing list, `http://groups.yahoo.com/group/boost`
Boost-Users mailing list,
`http://groups.yahoo.com/group/Boost-Users`
Boost Wiki, `http://www.crystalclearsoftware.com/cgi-bin/boost_wiki/wiki.pl`