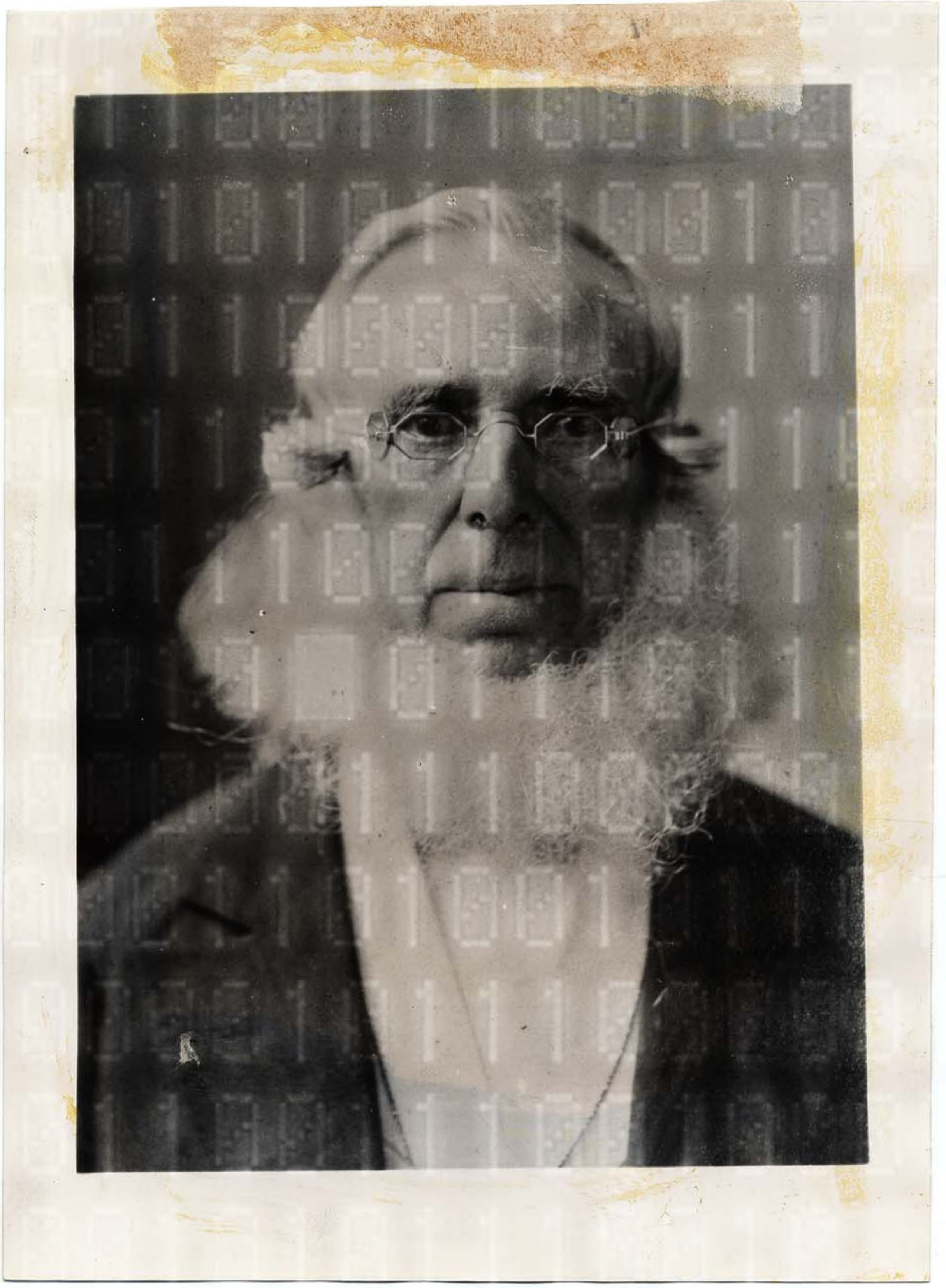
overload 100

DECEMBER 2010 £3



A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 100

December 2010

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copy deadlines

All articles intended for publication in Overload 101 should be submitted by 1st January 2011 and for Overload 102 by 1st March 2011.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Numbers and The Appliance of Science

How sure are you of something? Ric Parkin considers how we build models, and celebrates a milestone.

And welcome to Overload.....100! While the cynical mathematician in me knows that the significance of the number is just a coincidental artifact of the number of digits on a bilaterally symmetric semi-evolved simian's pentadactyl forelimbs, it's still a worthwhile moment to pause and look back at how we got here, and where we'll be going. Some good old fashioned navel-gazing in fact.

When did it all start? I must confess, having only come across ACCU back in 2000 or so, that I didn't know too much about its beginnings, and about Overload in particular. Fortunately, the internet (and some willing volunteers) have come to my rescue: we now have (almost) all the back issues on the website [Overload] (although many are currently only available as a pdf of the whole journal) so we can browse through and see what life was like back then.

The first Overload came out in April 1993, and an account of its genesis as a Special Interest Group of what was then called the C User's Group (UK) can be found in that first editorial, including the initial inspiration of teaching people about the new facilities in a new trendy language called C++. Many of the early writers are unfamiliar to me, but there are a few whose names are still associated with ACCU. Early editions were simple newsheets, with source code distributed on an accompanying floppy disk. Over the years things have moved on, with technology changes helping with better quality publishing, easier collaboration via email and other communication routes, and the hosting of documents and source code online. Sadly some things aren't so great – the ACCU journals are now quite rare in that we distribute print versions, with most computing magazines now being web only (is it just me, or do other people find online technical articles much harder to read, in small chunks at low dpi, and animated adverts flickering away? Perhaps better quality e-publishing via gadgets such as the Kindle and iPad may improve this) . The content has changed too, with Overload (and the wider ACCU) no longer being exclusively C++ focused, but now taking in other languages as well as project management, and even some philosophical musings.

As an aside, I noticed that this history has some parallels my own relationship with C++: having first come across it in 1993 when I had to maintain a DLL to allow access to a C library from a Pascal program, it became my main language for the next decade or so, and then I branched out to use a wider mix of languages and technologies as well as doing more project management.

So what of the future? Well, Overload is currently looking healthy, with a good stream of regular and occasional articles, which a great production team turns into a magazine that people really seem to be interested in reading. We'd always like new articles and writers though, and I have heard people saying that they've an idea but don't seem to have the time, or aren't sure people would be interested. I can reassure them that pretty much every idea is interesting in some way, so drop me an email and we can advise and help you get something into print ... With the upcoming new C++ standard there's plenty of great opportunities for article ideas, so get cracking and be part of the next 100!

Modelling the world

I mentioned that sometimes we have more philosophical articles, and this issue has an interesting one from Rafael Jay on the parallels between bug hunting and the scientific method. This generated plenty of comments from reviewers at how to extend the idea further, so I'm sure it'll inspire many of you equally. This is an area which has always fascinated me, and chimed in with some other thoughts I've had recently, especially after Bruce Schneier's talk at the ACCU security conference at Bletchley Park. [Schneier]

The basic idea I took away was that people can have security, and they can feel secure, but that the two were not necessarily as connected as you'd expect. For example, many airport security measures, such as restricting what can be taken in hand luggage, don't actually make you significantly safer but are really there to reassure you because you can see that *Something Is Being Done* (although paradoxically the extra attention can play on your fears and make you feel less safe too...) On the other hand, you might feel perfectly safe in your car because you're in control, and yet the chances of being injured or killed in an accident is much higher than the plane. He described this in terms of there being what you *Feel*, and *Reality*, and you should be aware of the differences when evaluating a security response. He also added a third element, a *Model*, which is what you use to try to understand the *Reality* part when it gets complicated. Ideally your *Model* should reliably reflect *Reality* (at least for the questions you're asking of it), but sometimes it can get out of sync, especially if the *Reality* changes and you don't update the model. For example, say you've forced your system to insist on changing passwords every month. Your *Model* tells you that that limits an attacker's window of exploitation if they crack a password, and you *Feel* secure. But after a couple of months people have got fed up of forgetting their new passwords, and have settled on 'Password1', followed by 'Password2' etc... *Reality* has just changed but your *Model* no longer reflects it and you have a false sense of security.

I was intrigued by this Reality-Model-Feel separation, and realised you can apply it in many other situations. For example in politics, where many policy decisions may be done because of an underlying *Model* (or ideology, which may or may not reflect *Reality*!), but the presentation is

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

often about manipulating the audiences *Feel* appropriately. User Interface design has related ideas too – a good UI should induce a user to have a particular mental *Model* that reflects the task that they are trying to do. The underlying *Reality* of how this task is achieved may be very different, but if you get the UI *Model* to mirror what they're trying to do, you'll have a good UI.

Closer to home, I recognised from Rafael's article how debugging is often about looking past your initial *Feel* about some code ('Of course this loop works – we've tried it before!'), building a new *Model* ('How many times does this loop really run in this case?'), and testing that against *Reality* to determine where the bug actually is ('Ooops, negative count passed in'). And as his article title suggests, the Scientific Method has many similar features, both in how it ought to work – by checking the results of your *Model* against *Reality*, you can check to see how accurate it is, and adjust it. And you can also use your *Feel* model to suggest possible improvements and checks to make sure your *Model* isn't resting on flawed assumptions, and suggest refinements and sometimes a complete rethink (although in practice, this happens very rarely in science as most models are already pretty good and just get adjusted. Plate Tectonics is one notable place where a true revolution occurred)

Sadly it can also be subverted. I've recently been reading *Bad Science* [Goldacre] and have found it troubling how badly science is reported in the media, and how our *Feel* model can be manipulated to not reflect reality, whether deliberately or just through misplaced hope or fear. The MMR vaccination scare is a classic case – while there was a large amount of evidence for its safety from this country and others, a very small scale study that found a borderline statistically significant correlation (ie it could well have been chance) was portrayed as a serious risk, which understandably concerned people. For some reason, many people just refused to believe any of the reassurances and further studies that demonstrated the safety – perhaps on the precautionary principle as there were children involved, or perhaps felt the concreteness of choosing to having the vaccine was too scary, whereas the abstract (and yet higher) risk of not having it wasn't. The problem is that everyone quite rightly uses their *Feel* model to quickly come to a conclusion, but if that model is not justified then you might get things wrong, and it can be very hard to change your initial gut feel. A striking example from Bletchley Park was that the Germans suspected that the Allies were getting intelligence on their activities, but were so certain that their codes were unbreakable that they never seriously contemplated that possibility.

Most of us do not have the time or expertise to check things out so rely on others to do so for us. The trouble is, who do you trust? People can be wrong for all sorts of reasons. An amusing example where common knowledge turns out to be just garbled tradition turned up on QI the other day [QI]: everyone knows that you should not drink alcohol when taking antibiotics, but why? I'd always thought it stops them working, but apparently that's not true (although with some it'd cause unpleasant side-effects): QI's answer was that one of the first major uses was to treat syphilis, especially in soldiers. But as people would still be infectious for a while after they started treatment, they were told not to drink to avoid them going out to celebrate with reduced inhibitions, which may cause one thing leading to another, so spreading the infection. And the advice stuck. (I thought I'd better do some research to see if this is plausible, and apparently it could well be true [Alcohol])

I've also been reading *Merchants Of Doubt* [MoD] which is much more troubling – this documents cases where people's *Feel* models are manipulated to discount evidence backing up a very different conclusion, whether for ideological or financial reasons (or just people being stubborn about not changing their own *Feel* model), often using the idea of a fair and balanced debate as a way of airing very minority views as if they were as well supported as 'the other side'. This way well be a reasonable thing to do in politics, but in science we can check our models to see how good they are so things are not just a matter of opinion. An old example was the campaign to cast doubt on the evidence that smoking was a major contributor to lung cancer. I doubt there are many people left who seriously disagree with that any more, but it took 30 odd years to get there, mainly because people were encouraged to think that 'the science isn't settled', or it wasn't 'proven'. Which of course science can never do, as it deals with finding models that are useful, but can always be improved as more evidence comes to light. This sort of tactic works very well in areas such as medicine where you are dealing with things that are highly complicated and probabilistic, as it plays on peoples desire for things being definitely one way or another. The book does cover more recent examples, some of which are still 'controversial', and yet the parallels are striking. It can be very hard to avoid prejudices, mistakes and misdirection (including your own) to build a good model that lets you come to a reliable conclusion, but I think it's something we should strive for.

References

[Alcohol] http://en.wikipedia.org/wiki/Antibacterial#Alcohol

[Goldacre] http://www.badscience.net/

[MoD] http://www.merchantsofdoubt.org/

[Overload] http://accu.org/index.php/journals/c78/

[QI] http://www.qi.com/

[Schneier] http://www.schneier.com/ and http://www.schneier.com/blog/archives/2008/04/the_feeling_and_1.html

Bug Hunting and the Scientific Method

Do you have a proper methodology when fixing bugs? Rafael Jay puts on his lab coat.

Bugs are a perennial part of the software development process. Despite our careful coding, our test-driven development, our peer reviews and our rigorous QA procedures, every release of more than a few lines of software seems inevitably to bring a swarm of bugs scuttling angrily behind it.

Much of the time it's reasonably easy to track down a bug. If you know which bit of code implements the troublesome behaviour you just take a look, prod a bit, and see what the problem is. I recall many years ago receiving a bug report that a particular script operation didn't seem to work. When I opened the offending source file it simply consisted of a comment, "`TODO: implement`". However not all bugs are this easy to diagnose. At the other end of the scale there are those that keep you scratching your head for days or weeks, feeling increasingly stupid at your inability to make the software – on which you are allegedly an expert – confess its misdemeanours. Or those which cause a live customer incident, where the chances of going home rapidly recede and you barely get time to think between managers importuning you for status updates. It's with these kinds of bug that I think the scientific method can be useful.

The scientific method is a process for answering questions about how the universe works. Why do the planets move through the night sky? Why do apples always fall towards the ground? Why can't I go to the pub every night while maintaining my perfect physique? It starts from observable reality and tries to construct a model which answers these questions.

We don't use it much in software development because mostly we already know the answers. Why does my application save to disk when I press Ctrl+S? Because that's what I told it to do. A physicist looking at the night sky cannot directly perceive the laws that govern the universe. But a developer looking at a running application can. In fact they're visible with automatic syntax highlighting in her favourite IDE.

As developers we actually frame the laws that govern how our applications – our universes – work. This means our mental models of how they work, and what we believe they will do in any given situation, are usually pretty accurate. The instances where this is not so are generally what we call bugs: our mental model tells us that our application should behave in one way, but in fact it behaves in another. Instead of saving to disk it wipes the hard drive. To fix the bug we need to find out why, and this is where scientific method can help.

The scientific method starts from what we already know, constructs hypotheses about what might be true based on that knowledge, then conducts experiments to prove or disprove the hypotheses. This yields fresh knowledge and the cycle repeats until we've answered our question.

Figure 1 shows the components of the scientific method. Let's look at each of them in turn.

Rafael Jay is a senior developer and technical lead working on financial trading platforms in C++ and .NET. He can be contacted at rafaeljay@bcs.org.uk

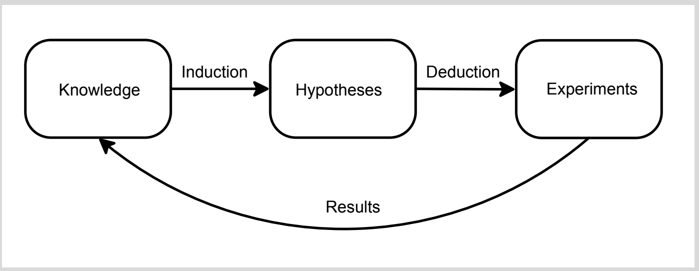


Figure 1

Knowledge

Sherlock Holmes once remarked that from a drop of water, a logician could infer the possibility of an Atlantic or a Niagara without having seen or heard of one or the other [Doyle]. But before said logician can do so, he must notice the drop.

I was once stumped for over a year by a bug that had caused CPUs to spin wildly for no apparent reason. Then one day I happened to notice that the system uptime just before the bug was very close to 4294967296, or 2^{32}. You can guess the rest – a classic 32-bit integer overflow bug. I hadn't spotted it previously because system uptime is reported in the system log once a day, at midday. The bug had occurred some hours after midday. So I didn't think to look. But like the drop of water, that piece of information was sitting there all along, waiting for someone to notice it and infer the CPU Niagara.

I now keep a checklist of possible information sources to consult when a tricky bug comes in. This helps me avoid missing key pieces of information. Some of these sources are specific to the products I work on, but some apply to the operating system or even just to software engineering in general. For example it's often a good idea to get a direct account of what happened from the people most directly involved, rather than relying on the circuitous word of mouth that can intervene between a live issue and a developer being summoned.

As you gather together what you know about a bug, it's a good idea to collect it in one place and keep it clearly labelled. This is especially so if more than one person is working on the bug, but even if it's just you it can get difficult to remember where each cryptically labelled crash dump file actually came from. I once wasted hours on a customer-critical bug trying to figure out why two of us were seeing different results from the same database dump, only to eventually realize that we were looking at different dumps with similar filenames.

Differentiate between what you actually know and what you merely presuppose. I recall one bug where a script ran fine on its own but got stuck when it ran as part of a batch file. After much investigation I realized that the script was in fact running fine as part of the batch file as well. It was actually the next script in the batch that got stuck. However the log wasn't flushed regularly enough and this made it look like the problem was in the original script. I had presupposed that the problem was in the original script but all I actually knew was that the last log message from the batch run came from that script.

I have seen more **bug-hunting time wasted** by false presuppositions than any other cause

Over my career I have seen more bug-hunting time wasted by false presuppositions than any other cause. It is very easy to start out with what seems like a reasonable presupposition, such as that a bug must be in a particular module, and forget to re-evaluate the presupposition as you dive deeper and deeper into technical investigations. Every time you find yourself back at the Knowledge stage of the scientific method, you should check your presuppositions and ask whether they still make sense in the light of whatever experiments you've conducted and the fresh knowledge thereby acquired.

The presuppositions on which physical science is based actually shade off into some very philosophical regions. For example, scientists presuppose that a physical universe exists at all, and that we are not merely butterflies dreaming of being humans. Such considerations don't generally impinge on software engineers. Even if I'm a butterfly dreaming of coding C++, I still have to fix that bug or I'll be a butterfly dreaming of a P45. But there are some points worth bearing in mind. A trap I've sometimes fallen into is where the code I'm working on is not the code I'm running. For example I'm building a debug version but running the release version. This can be mystifying when your code changes seem to have no effect. The problem is essentially that you're observing the wrong universe. Similarly it's worth considering whether the tools you use to perceive your universe, such as debuggers or profilers, are actually giving you an accurate view. Although it's relatively rare, those tools can have bugs in them. More commonly, your own brain – your most essential tool – can deceive you. Many of us will have experienced the [CardboardProgrammer] phenomenon from time to time, where simply talking through a bug will reveal an 'obvious' discrepancy between what we perceived the code to be doing and the reality. This can be a particular problem with code you wrote yourself, where it's all too easy to see what you meant to write rather than what you actually wrote.

Induction

Induction is the process of building hypotheses from knowledge. It moves from the particular to the general, starting from the particular facts we know to be true and building more general theories about why those things might be happening. For example, the stars all seem to whiz around the Earth: perhaps the Earth is at the centre of the stars? The application only crashes when I've been using feature *X*: perhaps feature *X* is corrupting the heap?

Induction in physical science is often a long and arduous process, requiring years of painstaking observation before a flash of creative genius draws out the pattern in the data. It was a long time before anyone put together enough knowledge to show that the Earth went around the Sun rather than vice versa. Things are simpler in software because we can peek behind the physical reality (a running application) to see the laws (source code) that govern it. This means it is relatively easy to look at the symptoms of a bug and enumerate the possible things that could be causing it. Nevertheless, it's often worth having as many people as possible involved in the process, because overlooking one of those possible things can result in a lot of wasted time.

Hypotheses

A hypothesis is a theory about what could be causing the things you know to be happening. For example, you hypothesize that the hard disk is wiped when you press CTRL+S because you've called the wrong function in your code. The goal of the induction process is to put together a set of hypotheses which is as complete as possible within your presupposed bounds; and then to assign probabilities to those hypotheses.

A complete hypothesis set is one which covers the entire range of possibilities, such that one of the hypotheses must be correct. For example, "either the Earth goes round the Sun or it doesn't" is a complete hypotheses set; whereas "either the Earth goes round the Sun or the Sun goes round the Earth" isn't. The latter ignores the possibility that neither Earth nor Sun goes round the other.

A hypotheses set can be too complete. For example: the hard disk is wiped because I've called the wrong function; or because I've written the function incorrectly; or because the operating system is broken; or because the compiler is broken; or because ninjas[1] sneak in and wipe the drive when I'm not looking; or in fact I am actually a butterfly dreaming of being a programmer and none of this is real. You can imagine an infinite number of outlandish hypotheses and the correspondingly infinite time that would be required to investigate them all (even assuming it were possible to do so). To avoid this, we make presuppositions, such as ruling out ninjas, to eliminate outlandish hypotheses.

To guide your bug investigation, you need to assign probabilities to each hypothesis. Is it more likely that you've called the wrong function or that the compiler is broken? The answer dictates which hypotheses you should investigate first, or in most depth. Experience is valuable here – both of programming in general and of your specific product. Sometimes you know that a particular module is flaky and more likely to be the source of issues than another. And one thing I've learned over the years is that the compiler is rarely broken. Third party software that is widely used in a variety of settings is much more likely to be working correctly than your own software. It's more probable that you haven't understood how to use it correctly. I've only ever encountered or heard tell of a handful of compiler bugs over the years, for example the Microsoft **`auto_ptr`** bug, which was widely documented online. If the compiler is broken, then chances are that someone else already knows about it. The same applies to widely-used third party libraries.

Deduction

Hypotheses need to be tested. It's no good hypothesising that the Earth goes around the Sun unless there's something you can do to prove it. Such proof comes from experiments. Deduction, or deductive logic, is the tool we use to devise experiments to prove or disprove particular hypotheses.

Deduction is the opposite of induction. It moves from the general to the particular. If the Earth is at the centre of the stars, then we ought to see Arcturus moving in a particular pattern. If feature *X* is causing a crash, we

1. For more information about ninjas, see http://www.realultimatepower.net

When a critical or difficult-to-diagnose bug comes in, you don't have infinite resources

should see evidence of feature *X* being run before each crash. We deduce a specific prediction from the general principle; then we devise an experiment to see if the prediction is true.

Deductive logic is a substantial subject in its own right and I won't attempt to cover it at all thoroughly here. Indeed, we generally don't need anything more than a common-sense grasp of the rules of logic to get by in the world of software engineering. However I think it might be useful, by way of an example, to look at one of the more commonly applied rules of logic and consider the potential pitfalls when applied to bug hunting. It also gives an excuse for some Latin, which is always nice.

Modus ponendo ponens translates as 'the way that affirms by affirming' and is a simple argument form in logic. It is generally abbreviated to just *modus ponens*. It works as follows:

- If *P* is true, then *Q* is also true
- *P* is true
- Therefore *Q* is true

For example: if feature *X* is run, then message *M* is written to the log file; feature *X* is run; therefore message *M* is written to the log file.

You can see how this might be useful in devising an experiment to prove the hypothesis that feature *X* is causing a crash. We can apply *modus ponens* to deduce a testable proposition which will be true if feature *X* did indeed cause the crash – specifically that message *M* will appear in the log file before the point where the application crashes. If this is true then the hypothesis becomes more likely; if false, the hypothesis is false.

At this point it is important to beware of logical fallacies. These are where you incorrectly construct a logical argument such that its conclusions are not valid. For example, a common fallacy afflicting *modus ponens* is that of 'affirming the consequent', which essentially means confusing cause and effect:

- If *P* is true, then *Q* is also true
- *Q* is true
- Therefore *P* is true

This is obviously wrong when written as a bare logical argument. But in the real world it can be harder to catch. For example: if feature *X* is run, message *M* is written to the log file; message *M* is written to the log file; therefore feature *X* was run. This takes the evidence of a log file message as proof that feature *X* was run. But what if feature *Y* happens to write an identical message when it's run? In that case the evidence of the log file message does not prove that feature X was run. You could waste a lot of time investigating feature *X* under the delusion that it was actually run before the program crashed. It's worth taking a bit of time to check your deductive logic and confirm that your experiments actually prove or disprove what you think they do.

Experiments

Experiments are the things you actually do to learn more about what might be causing the bug. They prove or disprove your predictions; which in turn strengthens or weakens the hypotheses from which you deduced those predictions. An experiment can be something as quick as looking in a log file or as involved as writing a scaled down version of a complex system to see if you can reproduce a bug with less surrounding clutter.

This last point is important: experiments cost. When a critical or difficult-to-diagnose bug comes in, you don't have infinite resources. There are only so many people you can take away from their regular duties, and there is only so much time they can spend on a bug before it becomes too costly for your company, or you lose a customer because you couldn't fix the problem quickly enough. It's also much more satisfying to be the guy who turned that critical bug around in 24 hours than the guy who went mad through working late nights for three solid weeks and had to be talked out of the cupboard.

I find it can be worth thinking like a laboratory administrator in these situations. You have a research problem – diagnosing the bug. You have a laboratory equipped with various computing hardware and software engineering tools. You have staff. What's the cheapest and quickest way of diagnosing the bug?

The starting point is to focus on the most likely hypotheses. What experiments can you deduce to strengthen or refute them? How much will those experiments cost? If one is expensive, is there something cheaper you can do to get the same result? Just as when coming up with hypotheses, it's worth having as many people involved as possible when you come up with experiments.

A further consideration is whether you have the right equipment, the right staff, and the right materials on which to run your experiments. Some thought and even research in this area can pay dividends. On one bug I was investigating we had ruled out using a performance profiler because we felt it would take too long to set up and execute on the large application we were dealing with. But once we actually talked to an expert from another team it turned out we could get some results in less than half an hour. On another occasion I wasted a lot of time messing about with system clocks trying to diagnose a time-triggered bug before I realised there were third party utilities that could achieve the same effect much more easily.

It's also worth considering whether you can modify the data you're looking at to make experiments more efficient. When a customer reports a bug, one of the first things it's common to ask for is a complete dump of their database so you can try to reproduce the problem in-house. A perennial problem I've faced is that this can be a lot of data – gigabytes in some cases. Running experiments on gigabyte data sets can be very time consuming as your developer machine struggles under the heavy resource load. It's worth considering whether some time invested up-front in eliminating irrelevant data to get a smaller data set will be a good investment in terms of making subsequent experiments more efficient. It's also worth considering whether modifying the application you're experimenting on might be useful, for example adding some extra logging or inserting a sleep to flush out suspected threading problems. You need to exercise discretion here as changing the data or the code you're experimenting on can invalidate the experiments; but it can mean the difference between running an experiment every two minutes or every two hours.

Our universe appears to be one in which a vast array of physical phenomena can be explained by a very small set of laws

Experiments yield new knowledge, and the scientific method cycles back to the beginning. If you're lucky your knowledge now includes the cause of the bug, at least with sufficient certainty to start putting together a fix. If not, you can go through the cycle again, using your new knowledge to focus more tightly on the most likely hypotheses. Repeat until solved. Or until you decide it's all too costly, brush the bug under the carpet, and keep your fingers crossed that it doesn't happen again. Sometimes, though rarely, that actually is the right business decision.

Conclusion

Most of the time in software engineering we don't need the full rigour of scientific method. Our privileged insight into the source code – the laws of our particular universes – actually makes an ad hoc 'prod it and see' approach more efficient. But for intractable or customer critical bugs I think it can be well worth applying scientific method more formally. Following a clearly defined process reduces the risk of forgetting or overlooking key pieces of information, and it provides a solid framework for deciding the best way to use the resources at your disposal. This can save a significant amount of time that might otherwise be wasted on blind alleys and missed opportunities, especially on those pressured occasions when customers are shouting down the phone and managers gesticulating behind your chair.

As a final thought, I think it is worth taking a step back and considering why the scientific method has proved so useful to humanity over the centuries. An important reason is the nature of the universe we live in. Our universe appears to be one in which a vast array of physical phenomena can be explained by a very small set of laws. This is sometimes referred to as the property of parsimony – the universe gets a lot done with a little. It means that when science uncovers some new underlying principle, that principle is generally pretty useful – it explains or predicts many observed phenomena, with many consequent practical applications. But there is no reason why the universe should be parsimonious. Every apple could fall towards the Earth for distinct reasons, rather than as the result of a general principle of gravity. Every atom could move according to its own unique rules of quantum electrodynamics. Such a universe would be tremendously more difficult than ours to study and understand and it is likely that humanity, if it came into existence at all, would never have figured out very much about how it all worked.

Does this remind you of code bases you've worked on? As software engineers we create universes – running applications governed by the laws enshrined in source code. How parsimonious are the software universes you create? Can a developer understand a lot about your application with a relatively small set of principles? Or does each module and class have its own unique conventions that have to be understood piecemeal? It is well worth considering these issues as you develop because they determine how tractable your application will be to scientific method, and thus whether you will be the guy who fixed that bug or the guy who locked himself in the cupboard. ■

References

[CardboardProgrammer]
http://www.c2.com/cgi/wiki?CardboardProgrammer

[Doyle] *A Study in Scarlet*, Arthur Conan Doyle

From Occam's Razor to No Bugs' Axe

Designing good APIs that stand the test of time is notoriously hard. Sergey Ignatchenko suggests a radical guideline.

As usual, the opinions expressed within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with opinions of the translator and the Overload editor; please also keep in mind the difficulties in translating accurately from Lapine (like those described in [LoganBerry2004]). In addition, both the translator and Overload expressly disclaim all responsibility from any action or inaction resulting from reading this article.

"Fight Features. ...the only way to make software secure, reliable, and fast is to make it small."
Andrew S. Tanenbaum

Every time I start to develop a new API for fellow rabbits, I (and probably every other library developer) always face the same question: which functions might my users possibly want? Over the years, I have came to a seemingly paradoxical yet extremely practical approach to this problem, which I want to share here. It is not something entirely new, but I don't think it has ever been emphasized enough.

First, I'll try to analyze how it usually happens.

The usual approach: what else MIGHT our users want?

When the need for a new API (class or library) arises, the natural temptation of the developer is to complete the development of the API once and for all, and to provide everything any potential user could possibly want. Examples of such APIs are abundant, and such libraries are often successful, but there are several problems with this approach which leads to a reduction in productivity in the long run. These problems are:

- the inability to remove an unnecessary API: as soon as the API is released, it is extremely difficult to get rid of it. While there are attempts to introduce 'deprecation' into APIs (for example in Java [Java]), this process is usually *extremely* slow and only marginally helps with the problem.
- unexpected abuse of an API: as soon as an API is used by more that 3 developers you can count on it being used in all the ways you have thought of, and in all the ways you *didn't* think about. This often becomes a problem, especially if the API accidentally reveals certain details of the underlying implementation (which you hoped that nobody of sane mind would ever use, but this always turns out to be wishful thinking: there will be at least one person who disagrees with your definition of *sane*)
- these lead to the third, more important point: as soon as an API is released, you're essentially bound to maintain it virtually *forever*, including undocumented and unintentional quirks. Are you scared of maintaining it *forever*? I certainly am. *Tharn!*[1]
- this in turn greatly increases code *rigidity*: as you're bound to maintain the API *forever*, including all the unintended features, you're very often effectively prevented from changing the underlying implementation as someone may well be relying on some unintentionally exposed aspect.

In addition, this kind of *creeping featuritis* doesn't come free for developers who're *using* the library:

- it provides many features that most users of the library don't want: while few people will complain about the extra features, after exceeding a certain threshold it often causes the problem of 'I can't see the forest for the trees' when a developer just doesn't know where to start.
- it increases the risks of abuse, the effects of which can be significant. For instance, it is this risk which is one reason behind the decision of Linus Torvalds not to allow C++ into the Linux kernel [Torvalds2007].
- it is more likely for legitimate feature requests to be denied because implementing them would be a disaster: it often would be easier to implement if you weren't also required to support lots of existing features, many of which are unnecessary but required to be supported forever.

Waterfall vs Agile

One way to think about this 'include everything in sight' development approach is to compare it to the Waterfall development model. If the API, once designed, cannot be changed or extended, it pushes us to include everything which we think *might* be needed. Unfortunately, Waterfall development doesn't really work in practice because it is hard to predict what will *actually* be necessary. Fortunately, this was recognized in 2001 when the term Agile development was coined [Agile].

Within the Agile development model, the whole development process is iterative by design and changes are a part of the process and are welcome. Applying this principle to a library we can see that APIs can also change easily. At least in theory it means much less pressure on the API developer to put in 'everything a user *might* need' right away; in fact, many Agile methodologies explicitly state that developing features 'just in case' is not a good thing, for example XP states 'Never Add Functionality Early' [YAGNI].

No Bugs' axe

Unfortunately, up until now it has not been explicitly stated what constitutes 'too early' for a feature to be included and what does not. In my projects with fellow rabbits, I have used the following principle for a while with a considerable success:

1. The word *tharn* is difficult to translate into human language, but the closest meaning is 'stupefied by terror' [Loganberry2006]

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams [Adams].

Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

if you do not know what your users really want, you're not able to provide a reasonable API

If you do not have a concrete case of how a feature will be used – do not provide it.

I (without false humility) hereby propose to name it after yours truly, namely a "No Bugs' Axe" principle. In some very wide sense you can consider it as parallel of the classic 'Occam's razor' principle [Occam]: in the same way that this cuts off unnecessary entities needed to explain a phenomenon, No Bugs' Axe slashes away unnecessary features.

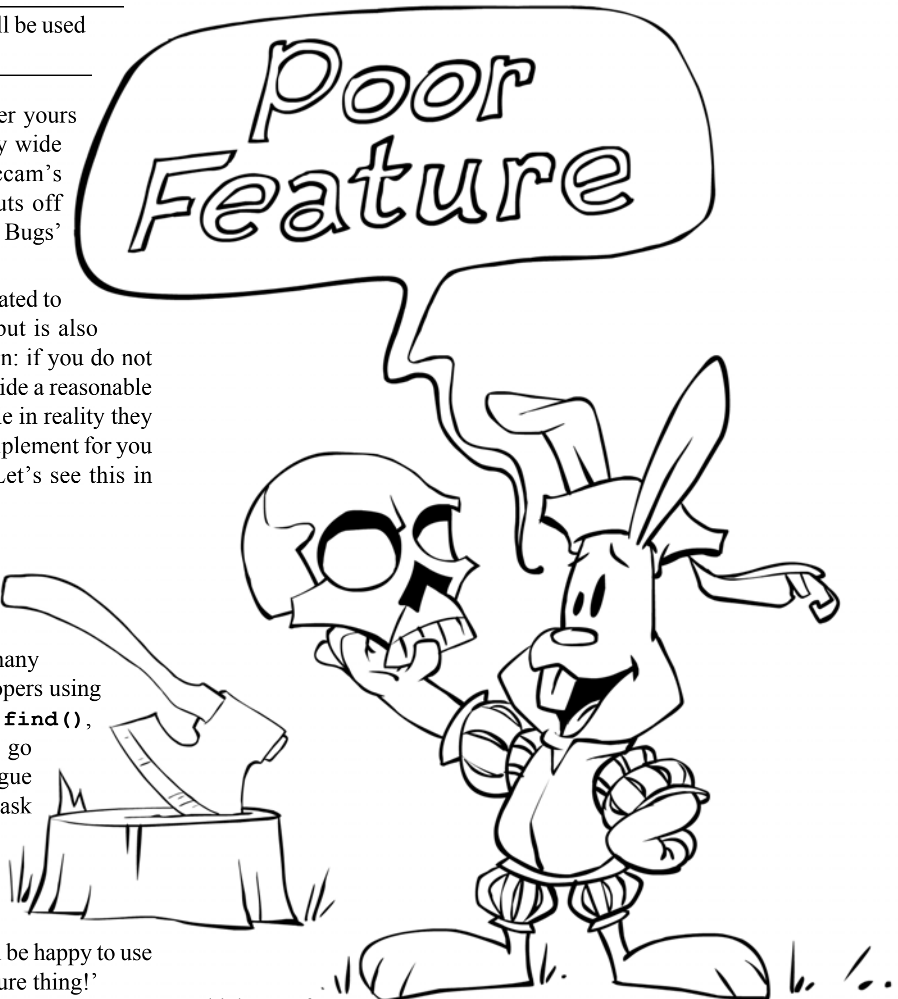
The rationale behind such a harsh approach is not only related to the problems of *creeping featuritis* mentioned above, but is also related to one obvious (though often ignored) observation: if you do not know what your users really want, you're not able to provide a reasonable API. It is very common that users request one thing, while in reality they need something very different which might be easier to implement for you and (much more importantly) easier for them to use. Let's see this in practice with a concrete example.

Suppose that you're developing your own String class for your project. Originally, following the Axe principle, you've made your String a bare-bones implementation which can only store a string and compare it to another one. Even such a simple implementation is useful for many practical applications. As time goes by, one of the developers using the library comes and asks you to introduce a function **`find()`**, similar to **`strstr()`** in C. It is certainly easier to just go ahead and implement such a simple function than to argue about it, but according to the Axe principle you should ask *why* the developer needs it. You've asked and s/he replied: 'Oh, I need to find out if the file extension is `.abc`, so I want to use **`find()`** to detect it.' After giving it a bit of thought, you ask, 'Using **`find()`** in such a manner is cumbersome and error-prone: would you be happy to use a Java-like **`endsWith()`** instead?'; and the answer is 'Sure thing!'

Next time, another developer comes and once again tells you that s/he needs **`find()`**. Again, you ask why does s/he need it? This time, the real need is to provide a substring search within an URL for a custom web server extension. After some thought and research, you realize that what users really need is not a substring search, but a pattern match, which the developer (knowing too well that its implementation is not trivial) was too humble to ask for. What was really necessary in this case was not simple **`find()`**, but some form of **`regexp`**.

What about code reuse?

One popular argument in support of including 'everything in sight' is 'if we provide an incomplete API, how it can be reused in the future?' There is only one problem with this argument: it is fundamentally flawed. As it has been observed for quite a long time, and recently articulated in [Kelly2010], it is not code reuse which really matters to deliver quality software: it is a set of other properties, like modularity and low coupling, which are of importance, but which are often considered too abstract. On the other hand, development aiming for code reuse tends to be as much as three time more expensive than developing single-use code. To address this conflict between code quality and development costs, [Kelly2010] proposes the approach of 'emergent reuse' – 'don't plan for reuse but look for opportunities to reuse something that has gone before' – which is perfectly consistent with No Bugs' Axe.



On hammer and nails

I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

Abraham Maslow, psychologist

As we have shown above, concrete use cases help to shape APIs in ways which can be hard to envision well in advance. But let's take a closer look: if a general **`find()`** function had been provided from the very beginning, would developers have ever come asking for better APIs? All my consistent with Agile development principles, **APIs evolve** with the project

experience convinces me that most developers will not ask for a new API when a workaround is available (unless it is really horrible to use, and even in this case it may still be misused though not as likely).

Therefore, restricting APIs unless concrete use cases are provided serves one more important purpose: it stimulates creating APIs which are the right tool for the job (instead of using a hammer on screws). As it has been shown above, it often helps to keep code as a whole more readable, less error-prone, and (paradoxically!) more functional.

Subtle points

There are a few important, though subtle, points to understand about the No Bugs' Axe principle:

- while it prohibits, or at least strongly discourages, implementing features until they're requested, it doesn't mean that you, as an API developer, should not *think* how you're going to implement a feature *if* it is requested in the future. The idea behind No Bugs' Axe is *not* to corner ourselves by relying on a feature never being requested, but exactly the opposite: to keep open all possible options, and the restriction on existing public APIs is one of the means to achieve this.
- it is important to understand that any refusal to implement a certain feature on the basis of the No Bugs' Axe is essentially temporary: if a satisfactory use case is demonstrated at some point later, the prior objections based on No Bugs' Axe are automatically revoked.
- nothing is carved in stone. Any attempt to follow a set of rules to the letter is doomed from the very beginning, therefore you should rely on your judgment and common sense. For example, despite being adamant on cutting off unnecessary APIs in the **`String`** class above, when the need for **`endsWith`** was demonstrated even I myself wouldn't argue against adding the complementary **`startsWith`** function. A rationale to provide **`startsWith`** would be not to provide a complete API, but to avoid confusion for developers who will reasonably expect to have **`startsWith`** when they see **`endsWith`**.

Pros and cons

Developing APIs under the No Bugs' Axe principle has some important implications:

- the API developer *must* be available to analyze the needs of API users, with round-trip times (from the moment of request to a reply, even if it is a negative one) being, not in the order of *months* (which is unfortunately often the case with API developers), but of *hours*, maximum *days*.
- developers who're using the API *should* be encouraged to submit requests when they feel new features are necessary. This includes treating even the most silly requests respectfully: in the worst case you can always write a FAQ about requests which you're not going to honour, with a list of workarounds.

In exchange for these (I'm sure minor) inconveniences, the following benefits are obtained from adhering to No Bugs' Axe:

- it encourages a coding style which uses exactly the right tool for the job
- it reduces the number of APIs which can be abused
- it reduces the number of APIs which need to be supported (most likely eternally)
- it reduces inter-dependencies, making code less *rigid* and less *fragile*;
- consistent with Agile development principles, APIs evolve with the project (which is inevitable anyway in the long run), but with much better backward compatibility as it is usually *much* easier to add a new API rather than to drop or change an existing one.

Other usages

As we have seen, the No Bugs' Axe principle works very well for APIs, but it can be easily extended into other areas, where it is also useful for similar reasons. In particular, the very same principle can be easily applied to user features. In practice, it is often not useful to take claims coming from BAs 'our user needs such and such checkbox here' uncritially – ask *why*. Usually the user (almost) *never* needs 'a checkbox', but instead the ability to specify something; and a checkbox might not always be the best way to do it. While asking questions like this might not be welcome within the current development culture of many companies, my experience shows that it often helps to improve quality of the end product, and therefore is beneficial for the company in general. ■

References

Agile] Manifesto for Agile Software Development, http://agilemanifesto.org/

[Java] 'How and When To Deprecate APIs', http://download.oracle.com/javase/1.4.2/docs/guide/misc/deprecation/deprecation.html

[Kelly2010] Allan Kelly, 'Reuse Myth – can you afford reusable code' http://allankelly.blogspot.com/2010/10/reuse-myth-can-you-afford-reusable-code.html

[Loganberry2004] David 'Loganberry', *Frithaes! – an Introduction to Colloquial Lapine!*, Unit 14: Feelings and Emotions; Parts of the Body (2), http://www.loganberry.furtopia.org/bnb/lapine/unit14.html

[Loganberry2006] David 'Loganberry', *Frithaes! – an Introduction to Colloquial Lapine!*, Dictionary – Lapine to English, http://www.loganberry.furtopia.org/bnb/lapine/dictlaptoeng.html

[Occam] http://en.wikipedia.org/wiki/Occam%27s_razor

[Torvalds2007] Linus Torvalds, 'Why C++ is a horrible language', http://article.gmane.org/gmane.comp.version-control.git/57918

[YAGNI] http://en.wikipedia.org/wiki/You_ain%27t_gonna_need_it and http://www.xprogramming.com/Practices/PracNotNeed.html

The Quartermaster’s Store

Be careful what you wish for. Phil Bass tries to simulate a missing language feature.

My eyes are dim, I cannot see.
I have not brought my specs with me.

Mission impossible?

It’s said you can find anything in the Quartermaster’s Store, even impossible things like fairy wings and unicorn tears. So that’s where I went to find some virtual function templates.

The quartermaster is an old man now. His face is wrinkled, his hair grey, but his mind is sharp and there’s a glint in his eye when I explain what I’m looking for. ‘Virtual function templates?’, he asks rhetorically, ‘I think I can help you, but, tell me, what are you going to use them for?’

Persistence required

Several of our applications use information stored in a SQL database. Currently, each program accesses the database directly and there is some undesirable duplication of the read/write functions across those applications. We want to remove this duplication by introducing an Object Store layer which knows how to map the logic layer objects to database tables.

For example, our systems define Users and Groups a bit like the users and groups in Unix operating systems except that a User can only be in one Group and permissions are only associated with Groups. In this design reading a whole Group involves reading a Group record and multiple User records. One application might read the Group record and then, separately, read the corresponding User records; another application might use a SQL join to read the records in a single SQL statement.

A generic Object Store would provide the equivalent of SQL insert, delete, select and update operations, but working with C++ objects rather than database records. So, for the User/Group example, the Object Store would provide a single function that reads a Group object from the database and the several applications would all use it.

Double dispatch

Unit test programs will exercise logic layer components that use the Object Store and in these programs we will want to substitute an in-memory database for the usual on-disk SQL database. Although we could implement a fake version of the SQL database we use, we realised that the existence of an Object Store opens up the possibility of a radically different interface for the in-memory database – an interface with a trivial implementation. For example, instead of using the SQL database C API functions to read records from User and Group tables and building a Group object from the records, the test database could contain ready made Group objects in a **`std::map<Group_ID,Group>`** and simply return the result of a map look-up.

With this requirement the implementation of the Object Store functions depends on both the type of the storable object and the type of database that will store it.

Unwelcome visitor

The stock solution to the problem of implementing a double dispatch mechanism in C++ is the VISITOR PATTERN, as described in the Gang of Four book, *Design Patterns* [GoF]. Note, however, that the VISITOR PATTERN is asymmetric: although the **`Element`** base class only needs to know the type of the abstract **`Visitor`** class, the **`Visitor`** class has to provide separate overloaded functions for each concrete **`Element`** type. In the case of our Object Store that means that either there must be a Database base class that knows the concrete types of all the Storable objects or there must be a Storable base class that knows the concrete types of all the Databases.

```
// A persistent store for storable objects.
class Object_Store
{
public:
  template<typename Storable>
  virtual insert(Storable const&) = 0;
  . . .
private:
  . . .
};
```

Listing 1

It would be nice to be able to write Listing 1. If this was possible we could write one concrete **`Object_Store`** class that stores objects in a real SQL database on disk and another that stores objects in a test database in memory. An application that uses the real database would not depend on the test database and, conversely, test programs would not depend on the real database. Similarly, any program that only used Group objects would not depend on any other storable objects.

A change of perspective

The quartermaster had listened carefully. ‘Try these glasses’, he said when I had finished. ‘They might give you a different perspective on the problem.’ He seemed to over-emphasise the ‘spec’ in perspective. Was he making a joke, alluding to his notoriously poor eyesight and reminding me of the old scout song? He thrust a battered pair of spectacles towards me. ‘Go on’, he said, ‘Try them.’ Hesitantly I took the glasses and settled them on the bridge of my nose. They were surprisingly comfortable. I could see clearly, too. The lenses hadn’t changed what I could see in any way I could identify, but things did look different, somehow. ‘Now, close your eyes, relax and think about your persistent object store again’, said the quartermaster.

Without quite realising how bizarre this was I did as he said; and new thoughts began to take shape in my mind. Virtual functions are implemented by building a table of pointers to functions. Virtual function

Phil Bass has been learning how to write software for the last 35 years and intends to continue learning for a few years more. He believes in agile methodologies, design patterns and trust-the-programmer languages. He can be contacted at phil@stoneymanor.demon.co.uk

These functions could be **organised across source files** in several different ways

```
// Real database mapping functions.
void insert(Real_Database&, Group const&);
void insert(Real_Database&, User  const&);
. . .
// Test database mapping functions.
void insert(Test_Database&, Group const&);
void insert(Test_Database&, User  const&);
. . .
```

Listing 2

tables are one dimensional, but for double-dispatch we'd need a two-dimensional table. Furthermore, the size of a virtual function table is fixed at compile time, and we don't want to force applications to contain the full 2D table if they only need a few storable types. So we are looking for a dispatch mechanism that selects different functions according to the types of two parameters and doesn't require a look-up table whose size is determined at compile time.

'Overloading!', I cried, opening my eyes. The quartermaster was startled for a moment, but quickly regained his composure. 'Take your time', he said, 'Think it through.'

Seeing the light

In my Eureka moment I had realised that an Object Store could be implemented without using templates, virtual functions, or even classes. It could just be a collection of overloaded functions that define object ↔ database mappings (Listing 2).

There would, of course, be functions for `delete`, `select` and `update` operations, and functions for many more storable object types. These functions could be organised across source files in several different ways. Perhaps the natural organisation would be to put all the operations that map a particular object type to a particular type of database into a single file. The corresponding object modules could be stored in libraries, perhaps one library for the real database and another for the test database. This way a program that only used a test database wouldn't depend on a real database; and a program that used Group objects wouldn't depend on unrelated storable object types.

A fly in the ointment

Following the quartermaster's advice I started to think about how the object mapper functions would be used. Typically, the logic layer classes would access the database through an abstract interface. The application programs would pass a real database to the logic layer functions and the unit tests would pass a test database instead. So, for example, a function that creates a new Group might look like Listing 3.

But, calling one of the object mapper functions requires a concrete database type and that can't be recovered from the abstract interface unless there is a suitable virtual function in the Database base class. So it looked as though we'd need an `Object_Store` class with virtual function template members after all.

```
void create_group(Database& database)
{
    Group new_group = ...;
    insert(concrete(database), new_group);  // ???
}
```

Listing 3

A helping hand

I was gutted and my disappointment must have shown on my face because the quartermaster chose that moment to drop a big hint. 'Could you have a member function template that forwards to a virtual function?', he asked. I closed my eyes again and visualised Listing 4.

The `Any_Storable` class would have to be a wrapper that retains the concrete Storable type and provides a mechanism for its recovery. There is just such a class in the Boost library: `boost::any`. The `insert_impl()` functions would recover the Storable type and call the appropriate object mapper function.

The type of the object in a `boost::any` is provided by the `any::type()` function, which returns a `std::type_info const&`. Somehow the `insert_impl()` function must use this to generate a reference to the storable object held within the `boost::any` object. The `insert_impl()` function can't do this directly because it doesn't itself know the concrete type of the Storable object. But it can look up and call a function that does know the concrete Storable type. For example, the `Object_Store` class might store a function registry in the form of a `std::map<std::type_info const*, insert_function*>` where `insert_function` is a suitable function type.

The functions in the function registry also need to know the concrete database type so that they can invoke the corresponding object mapper functions. Providing the function registry, therefore, must be the responsibility of the concrete `Object_Store` classes. Listing 5 shows an

```
class Object_Store
{
public:
  template<typename Storable>
  void insert(Storable const& storable)
  {
    insert_impl( Any_Storable(storable) );
  }

  . . .
private:
  virtual void insert_impl(Any_Storable) = 0;
  . . .
};
```

Listing 4

Unlike designs based on virtual functions the function registry needs to be populated at run time and this is the responsibility of the programmer

```
class Real_Object_Store : public Object_Store
{
public:
  Real_Object_Store(
     std::string const& connection_parameters);
  . . .

private:
  virtual void insert_impl(boost::any storable);
  dtl::DBConnection db;
  typedef
    std::map
    <
      std::type_info const*,
      void (*)(dtl::DBConnection&,
         boost::any const&)
    >
    function_map_type;

  function_map_type insert_function_map;
  . . .
};
```

Listing 5

```
void Real_Object_Store::insert_impl(
   boost::any storable)
{
  function_map_type::iterator i =
     insert_function_map.find(&storable.type());
  if (i == insert_function_map.end())
  {
    // Report insert function look-up failure.
  }
  else
  {
    ((*i).second)(db, storable);
  }
}
```

Listing 6

```
template< typename Storable >
inline
void insert_any(dtl::DBConnection& db,
   boost::any const& storable)
{
  insert(db,
     boost::any_cast<Storable const&>(storable));
}
```

Listing 7

example of an Object Store class that accesses a SQL database via the Database Template Library [DTL].

The database is represented by a `dtl::DBConnection` object, which is initialised from the connection parameters passed to the constructor as a string. The function registry stores pointers to functions taking `dtl::DBConnection&` and `boost::any const&` parameters. The `insert_impl()` function can then be implemented as shown in Listing 6.

Each function in the function registry handles a particular type of storable object; it simply recovers that type from the `boost::any` and forwards to the appropriate overloaded object mapper function. The registry functions are identical except for the concrete Storable type, so they can be generated from a trivial function template as shown in Listing 7. Note that the `any_cast` will not fail if the function registry has been correctly initialised.

Unlike designs based on virtual functions the function registry needs to be populated at run time and this is the responsibility of the programmer (not the compiler). That is the price we must pay to avoid coupling the Object Store source files to all the concrete Storable types.

Mission accomplished

Had I succeeded in my quest? Well, virtual function templates don't exist, but we can emulate them with a combination of member function templates that forward to a virtual function and function registries built at run-time. That was more than good enough for me.

I handed the glasses back to the quartermaster, thanked him and shook his hand warmly. 'I'm sorry that I couldn't get any virtual function templates for you', he said. 'Not at all', I replied. 'You have given me something much more valuable – a deeper understanding of an important design problem, complete with a practical solution.' But he wouldn't take any money, so I pushed a note into his charity collection box and bid him farewell. ■

References

[DTL] The Database Template Library is available from SourceForge at http://dtemplatelib.sourceforge.net/. There's also an article by one of its co-authors in *Overload* 43 (http://accu.org/index.php/journals/445).

[GoF] *Design Patterns – Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. ISBN 0-201-63361-2.

[Love] For an *Overload* article applying the 'virtual function template' technique to iterators see http://accu.org/index.php/journals/479.

Why Fixed Point Won't Cure Your Floating Point Blues

Numeric computing is very difficult to do properly. Richard Harris looks at whether a common technique will help.

In the first article in this series we explored floating point arithmetic and its various modes of failure. We saw that rounding error, although often touted as the big problem with floating point arithmetic, isn't really much to be concerned about when compared to cancellation error; the catastrophic loss of precision that occurs when subtracting two nearly equal floating point numbers. Finally, we noted the rather surprising fact that the order of execution of mathematical operations will result in different accumulated rounding errors and that, if we wish our calculations to be as accurate is possible, we need to carefully consider how we construct them.

In this article we shall explore the most frequently proposed alternative to it: fixed point arithmetic.

Fixed point

Fixed point arithmetic is perhaps the simplest alternative to floating point. Fixed point numbers maintain a fixed number of decimal places, rather than digits of precision. Typically they are implemented as an integer with the assumption that some constant number of the least significant base 10 digits fall below the decimal point.

For example, using a long and assuming that the last 2 decimal digits lie below the decimal point, we would represent π as 314.

Listing 1 provides an implementation of a base 10 fixed point number class.

This class uses the type **`T`** to represent the fixed point number, with **`N`** decimal digits after the decimal point. Note that we shall not presume that there is a specialisation of **`std::numeric_limits`** for **`T`** and hence if there isn't enough room in the type **`T`** to fit the digits we shall have undefined behaviour rather than a compilation error.

Note that, given in-place arithmetic operations and a copy constructor, it is trivial to implement the free standing arithmetic operations. Those who would prefer to avoid the computational expense of the copy could instead use member data access functions to implement more efficient free functions.

It shall henceforth be assumed that such free functions have been implemented.

The **`scale`** method is a static helper that calculates the scaling factor which we need to divide an integer by to recover the required number of decimal places, illustrated in listing 2 together with the **`calc_scale`** helper function that actually calculates it and the data member access function.

Listing 3 illustrates the constructors of this class.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

```
template<class T, size_t N>
class fixed
{
public:
  enum{places=N};
  typedef T data_type;
  fixed();
  fixed(const data_type &x);
  fixed(const data_type &before,
     const data_type &after);
  static const data_type & scale();
  const data_type &  data() const
  int               compare(const fixed &x) const;
  fixed &            negate();
  fixed & operator+=(const fixed &x);
  fixed & operator-=(const fixed &x);
  fixed & operator*=(const fixed &x);
  fixed & operator/=(const fixed &x);

private:
  static data_type calc_scale();
  data_type x_;
};
```

Listing 1

```
template<class T, size_t N>
const fixed<T, N>::data_type &
fixed<T, N>::scale()
{
  static const data_type result = calc_scale();
  return result;
}

template<class T, size_t N>
fixed<T, N>::data_type
fixed<T, N>::calc_scale()
{
  data_type result(1);
  for(size_t i=0;i!=places;++i)
     result *= data_type(10);
  return result;
}

template<class T, size_t N>
const fixed<T, N>::data_type &
fixed<T, N>::data() const
{
  return x_;
}
```

Listing 2

if we have a **large number of digits** after the decimal point then **multiplication and division** are very likely to **overflow the internal integer** operations

```
template<class T, size_t N>
fixed<T, N>::fixed()
{
}

template<class T, size_t N>
fixed<T, N>::fixed(const data_type &x)
   : x_(x*scale())
{
}

template<class T, size_t N>
fixed<T, N>::fixed(const data_type &before,
const data_type &after)
   : x_(before*scale()+after)
{
  if(after<0 || after>=scale())
     throw std::invalid_argument("");
}
```

Listing 3

Note that the only error check we're making is that the number passed to represent the digits after the decimal point in the final constructor is within the valid range; we treat overflow as undefined behaviour to keep things simple.

We allow implicit conversion from the underlying type to make using integers in fixed point calculations slightly easier. We do not provide any conversion between fixed and floating point numbers since this would rather defeat the purpose of using fixed point numbers in the first place.

The `compare` and `negate` member functions are provided to simplify the implementation of non-member relational and negation operators and their definitions are given in listing 4.

```
template<class T, size_t N>
int
fixed<T, N>::compare(const fixed &x) const
{
  if(x_<x.x_) return -1;
  if(x_>x.x_) return  1;
  return 0;
}

template<class T, size_t N>
fixed<T, N> &
fixed<T, N>::negate()
{
  x_ = -x_;
  return *this;
}
```

Listing 4

```
template<class T, size_t N>
fixed<T, N> &
fixed<T, N>::operator+=(const fixed &x)
{
  x_ += x.x_;
  return *this;
}

template<class T, size_t N>
fixed<T, N> &
fixed<T, N>::operator-=(const fixed &x)
{
  x_ -= x.x_;
  return *this;
}
```

Listing 5

Addition and subtraction are straightforward operations for fixed point numbers. Since both arguments are guaranteed to have the same number of digits after the decimal point, we simply use the corresponding integer operations on the underlying type, as shown in listing 5.

Note that whilst we are again treating overflow as undefined behaviour, these operations introduce *no* further error into the result above and beyond those that were in their arguments.

The naïve approach to multiplication is to multiply the underlying integer values and then divide by the scaling factor since

$$\frac{x_1}{scale} \times \frac{x_2}{scale} = \frac{(x_1 \times x_2) \div scale}{scale}$$

Similarly, for division we have

$$\frac{x_1}{scale} \div \frac{x_2}{scale} = \frac{x_1 \div x_2}{scale \div scale} = x_1 \div x_2$$

There are two problems with this approach. The first is that we are no longer rounding the result, we are merely truncating it. Without careful analysis it is possible that multiplicative operations will introduce twice as much error than they would if we had correctly rounded the result.

Fortunately, this problem is relatively easily fixed. For multiplication, we simply add half the scale before dividing by it. For division we add half of the number by which we are dividing before the actual division. Listing 6 illustrates the correctly rounded multiplication and division operators.

The second, and far worse, problem is that if we have a large number of digits after the decimal point, and consequently a large scaling factor, then multiplication and division are very likely to overflow the internal integer operations even for numbers close to 1.

Fixing this problem is a little trickier. What we really need is a larger integer type to store intermediate values. However, since we aren't forcing the user to use built in integers, it's not at all obvious how we should go about this, short of implementing one in terms of the internal integer type.

I'm sure you'll agree that this is a **fairly significant difference**; if you don't, **multiply the whole expression by a few million** for effect.

```
template<class T, size_t N>
fixed<T, N> &
fixed<T, N>::operator*=(const fixed &x)
{
  x_ *= x.x_;
  x_ += scale()/2;
  x_ /= scale();
  return *this;
}
template<class T, size_t N>
fixed<T, N> &
fixed<T, N>::operator/=(const fixed &x)
{
  x_ += x.x_/2;
  x_ *= scale();
  x_ /= x.x_;
  return *this;
}
```

Listing 6

Because of this, I shall stick with this implementation and declare caveat emptor for users who desire a large number of digits after the decimal point.

The advantages of fixed point

So now we know exactly how to implement fixed point numbers we should probably ask ourselves why we should want to.

Well, there are three principal advantages to fixed point numbers.

The first, and by far the least important, advantage is that unlike IEEE 754-1985 floating point, it can exactly represent decimal fractions. This is trotted out *ad nauseum* as a reason to use fixed point but, given that a revision of the standard describes the implementation of base 10 floating point numbers, it isn't particularly compelling.

The second, slightly more important, advantage is that fixed point arithmetic can be dramatically faster on some processors.

The third, and by far the most important, advantage is that, as described above, addition and subtraction introduce no error beyond that in their arguments. This means that it is *much* easier to reason about addition using fixed point arithmetic; if we are summing numbers that can be exactly represented we have no rounding issues at all and need only worry about overflow.

This, together with the ability to represent some decimal fractions exactly is why many financial transactions are mandated to use base 10 fixed point numbers.

The disadvantages of fixed point

Unfortunately, multiplication is rather more problematic.

Even assuming we are not much affected by overflow of the underlying type, if we multiply fixed point numbers our calculations may be severely affected by execution order. To demonstrate just how severely, let us now assume that we are using 2 decimal place base 10 fixed point numbers to calculate the value of the expression

$$1000 \times 1000 \div 1000000$$

If we calculate this in the order

$$(1000 \times 1000) \div 1000000$$

we trivially have a result of 1.

However, if we calculate it in the order

$$1000 \times (1000 \div 1000000)$$

we trivially have a result of 0.

I'm sure you'll agree that this is a fairly significant difference; if you don't, multiply the whole expression by a few million for effect. The problem is that multiplicative errors accumulate proportionally rather than absolutely. Additively, fixed point numbers behave themselves but the instant we start multiplying and dividing they turn on us.

For general purpose arithmetic, therefore, fixed point numbers seem to be something of a lame duck.

Quack.

Arbitrary precision

One way to fix the problem of overflow in fixed point numbers is to use arbitrary precision integers, also known as bignums, as the underlying type.

These are typically implemented using an array of integers which can grow as needed to represent any given integer.

Listing 7 provides an implementation of an arbitrary precision integer class.

Note that we represent our number with a **`vector`** of **`unsigned short`**s and an **`enum`** to indicate the sign. For the sake of simplicity, we shall assume that **`short`**s are exactly 16 bits wide and that **`long`**s are exactly 32 bits wide. Whilst many platforms conform to these assumptions, they are not mandated by the C++ standard. A robust implementation would therefore need to take a more cautious approach.

Our underlying representation will not admit leading zeros, except for the number zero which shall consist of a single zero digit. To that end we shall require a helper function to strip leading zeros in those circumstances that we may need to; its implementation is given in listing 8.

Note that this function assumes that we shall be using a little-endian representation in which the most significant digits appear at the end of the **`vector`** and consequently iterates in reverse seeking the last non-zero digit. This is because, if and when in place operations increase or decrease the number of digits, we won't really want to move the least significant digits, as we would need to if we used a big-endian representation.

The constructors are fairly straightforward and are illustrated in listing 9.

We shall perform addition in much the same way as **we were taught** to with pencil and paper **when we were children**

```
class bignum
{
public:
  typedef unsigned short          digit_type;
  typedef unsigned long           product_type;
  typedef std::vector<digit_type> data_type;

  enum sign_type{
     positive=1,
     negative=-1
  };
  enum {mask=0xffff, shift=16};

  bignum();
  bignum(short x);
  bignum(unsigned short x);
  bignum(long x);
  bignum(unsigned long x);
  bignum(sign_type s, const data_type &x);

  sign_type       sign() const;
  const data_type & magnitude() const;
  void            swap(bignum &x);
  int             compare(const bignum &x) const;
  bignum &        negate();

  bignum & operator+=(const bignum &x);
  bignum & operator-=(const bignum &x);
  bignum & operator*=(const bignum &x);
  bignum & operator/=(const bignum &x);

private:
  sign_type s_;
  data_type x_;
};
```

Listing 7

```
void
data_strip_leading_zeros(bignum::data_type &x)
{
  bignum::data_type::reverse_iterator first =
     x.rbegin();
  bignum::data_type::reverse_iterator last  =
     x.rend();

  while(first!=last && *first==0) ++first;

  if(first==last) x.resize(1, 0);
  else            x.resize(last-first);
}
```

Listing 8

```
bignum::bignum() : s_(positive), x_(1, 0)
{
}
bignum::bignum(const short x)
   : s_(x>=0 ? positive : negative),
   x_(abs(x))
{
}
bignum::bignum(const unsigned short x)
   : s_(positive), x_(1, x)
{
}
bignum::bignum(const long x)
   : s_(x>=0 ? positive : negative)
{
  const unsigned long ux = labs(x);
  x_.reserve(2);
  x_.push_back(digit_type(ux&mask));
  if(ux>mask)
     x_.push_back(digit_type(ux>>shift));
}
bignum::bignum(const unsigned long x)
   : s_(positive)
{
  x_.reserve(2);
  x_.push_back(digit_type(x&mask));
  if(x>mask)  x_.push_back(digit_type(x>>shift));
}
bignum::bignum(const sign_type s,
   const data_type &x) : s_(s), x_(x)
{
  data_strip_leading_zeros(x_);
  if(x_.back()==0) s_ = positive;
}
```

Listing 9

The data access functions `sign` and `magnitude` and the `swap` and `negate` member functions are similarly simple as shown in listing 10.

The choice of a little-endian representation is also reflected in the `compare` function, shown in listing 11, where we use `const_reverse_iterator`s in the `data_compare` helper function to search for the most significant digit that differs in the two numbers in the event that they are the same sign and have the same number of digits.

We shall perform addition in much the same way as we were taught to with pencil and paper when we were children. We begin with a helper function that we will use to add single digits and check whether or not we have to carry over a 1, as illustrated in listing 12.

Note that we pass in a flag to indicate whether a carry needs to be added to the digits and that we exploit the rules of unsigned arithmetic in C++ to check whether the result will itself generate a carry. Specifically, *n* bit

Subtraction is a little more difficult since it is not a symmetric operation

```
bignum::sign_type
bignum::sign() const
{
  return s_;
}
const bignum::data_type &
bignum::magnitude() const
{
  return x_;
}
void
bignum::swap(bignum &x)
{
  std::swap(s_, x.s_);
  std::swap(x_, x.x_);
}
bignum &
bignum::negate()
{
  if(x_.back()!=0) s_ = sign(-s_);
  return *this;
}
```

Listing 10

unsigned arithmetic in C++ is defined as being modulo 2^n and hence any result that wraps around, and therefore generates a carry, will be less than both of the terms in the sum or, if a carry was also added, perhaps equal to one of them.

We use this in conjunction with a second helper function that adds all of the digits contained in a pair of `bignum`'s `data_type`s, as shown in listing 13.

We first ensure that `x1` is as least as big as `x2` by resizing and padding it with zeros if it is smaller.

The first loop then adds, with carry, the digits up to the last in `x2`. The second loop ensures that further carries are properly added to any more significant digits of `x1`. Note that our initial padding of `x1` with zeros in the event that it has fewer digits than `x2` ensures that we don't have to worry about dealing with reaching the end of `x1` before we reach the end of `x2`.

Subtraction is a little more difficult since it is not a symmetric operation. We shall therefore wish to ensure that we always subtract the smaller number from the larger and shall need to explicitly keep track of the sign of the result.

This is reflected in the first helper function, shown in listing 14.

Note that we treat the operation in much the same way as we did addition, but since we don't know which number will be the larger, can no longer perform the operation in-place.

```
int
data_compare(const bignum::data_type &x1,
             const bignum::data_type &x2)
{
  assert(x1.size()==x2.size());

  bignum::data_type::const_reverse_iterator
     first1 = x1.rbegin();
  bignum::data_type::const_reverse_iterator
     last1  = x1.rend();
  bignum::data_type::const_reverse_iterator
     first2 = x2.rbegin();

  while(first1!=last1 && *first1==*first2)
  {
    ++first1;
    ++first2;
  }

  if(first1==last1)   return 0;
  if(*first1>*first2) return 1;
  return -1;
}

int
bignum::compare(const bignum &x) const
{
  const int dir = s_;

  if(x.s_!=s_)              return  dir;
  if(x_.size()>x.x_.size()) return  dir;
  if(x_.size()<x.x_.size()) return -dir;

  return data_compare(x_, x.x_) * dir;
}
```

Listing 11

```
bool
data_add(bignum::digit_type &x1,
   const bignum::digit_type x2, const bool carry)
{
  x1 += x2;
  if(carry) ++x1;
  return x1<x2 || (carry && x1==x2);
}
```

Listing 12

The second helper function, illustrated in listing 15, ensures that the digits of the larger number are passed as `x1` to this function and the digits of the smaller number are passed as `x2`.

we can be certain that the product will have **no more digits** than the sum of those in its term

```
void
data_add(bignum::data_type &x1,
   const bignum::data_type &x2)
{
  typedef bignum::data_type::iterator
     iterator;
  typedef bignum::data_type::const_iterator
    const_iterator;
  if(x1.size()<x2.size()) x1.resize(x2.size(),
     0);
  iterator first1 = x1.begin(),
     last1 = x1.end();
  const_iterator first2 = x2.begin(),
     last2 = x2.end();
  bool carry  = false;
  while(first2!=last2)
     carry = data_add (*first1++,
                       *first2++, carry);
  while(first1!=last1 && carry)
     carry = data_add (*first1++, 0, carry);
  if(carry) x1.push_back(1);
}
```

Listing 13

Note that, since we ensure that the smaller number always is subtracted from the larger, we don't need to deal with a final carry. We do however need to ensure that we strip any leading zeros since we are using the number of digits during the comparison operation. Finally, we must also return the sign of the result.

Now that we can add and subtract the underlying `data_type` we can implement addition and subtraction operators for the `bignum` type itself. We must take care to direct these operations to the addition and subtraction helper functions according to whether the numbers have the same sign or not, as shown in listing 16.

Irritatingly, multiplication is a little less straightforward. The scheme we were taught as children is an $O(n^2)$ operation for multiplying two n digit numbers. More efficient schemes exist but unfortunately they are much more complicated. One, for example, recasts multiplication as a discrete convolution operation and exploits the Fast Fourier Transform to compute it efficiently [Press92].

```
  bool
  data_subtract(bignum::digit_type &y,
                const bignum::digit_type x1,
                const bignum::digit_type x2,
                const bool carry)
  {
    y = x1-x2;
    if(carry) --y;
    return y>x1 || (carry && y==x1);
  }
```

Listing 14

```
int
data_subtract(bignum::data_type &x1,
   const bignum::data_type &x2)
{
  typedef bignum::data_type::iterator
     iterator;
  typedef bignum::data_type::const_iterator
     const_iterator;
  const int dir = data_compare(x1, x2);

  if(x1.size()<x2.size())
     x1.resize(x2.size(), 0);
  const_iterator first1 =
     (dir>=0) ? const_iterator(x1.begin())
              : x2.begin();
  const_iterator last1  =
     (dir>=0) ? const_iterator(x1.end())
              : x2.end();
  const_iterator first2 =
     (dir>=0) ? x2.begin()
              : const_iterator(x1.begin());
  const_iterator last2  =
     (dir>=0) ? x2.end()
              : const_iterator(x1.end());
  iterator       out    = x1.begin();
  bool           carry  = false;
  while(first2!=last2)
     carry = data_subtract(*out++, *first1++,
     *first2++, carry);
  while(first1!=last1)
     carry = data_subtract(*out++, *first1++,
     0, carry);
  data_strip_leading_zeros(x1);
  return dir;
}
```

Listing 15

For this treatment, the simpler, less efficient approach, as given in listing 17, will have to suffice.

We use `product_type` values during the multiplication of terms so that we can represent numbers that would overflow `digit_type`. Note that since we are assuming that the former is exactly twice as wide as the latter it is guaranteed to be large enough to represent any such product.

Similarly, we can be certain that the product will have no more digits than the sum of those in its terms, and hence do not need to check whether we are accessing the result within its bounds whilst accumulating carries.

Note that the sign of the result is positive if both terms have the same sign or if the result is zero and is negative otherwise. Checking that the leading

```
bignum &
bignum::operator+=(const bignum &x)
{
  int dir = 1;
  if(x.s_==s_) data_add(x_, x.x_);
  else         dir = data_subtract(x_, x.x_);

  if(dir < 0)       negate();
  else if(dir == 0) s_ = positive;
  return *this;
}

bignum &
bignum::operator-=(const bignum &x)
{
  int dir = 1;
  if(x.s_==s_) dir = data_subtract(x_, x.x_);
  else         data_add(x_, x.x_);

  if(dir < 0)       negate();
  else if(dir == 0) s_ = positive;
  return *this;
}
```

Listing 16

```
bignum &
bignum::operator*=(const bignum &x)
{
  data_type result(x_.size()+x.x_.size());

  for(size_t i=0;i!=x_.size();++i)
  {
    for(size_t j=0;j!=x.x_.size();++j)
    {
      bool carry; size_t k;
      const product_type term
         = product_type(x_[i]) *
         product_type(x.x_[j]);

      const digit_type lo(term &  mask);
      const digit_type hi(term >> shift);

      carry = data_add(result[i+j], lo, false);
      for(k=1;carry;++k) carry =
         data_add(result[i+j+k], 0, carry);

      carry = data_add(result[i+j+1], hi, false);
      for(k=2;carry;++k)  carry =
         data_add(result[i+j+k], 0, carry);
    }
  }
  data_strip_leading_zeros(result);

  if(s_==x.s_ || result.back()==0) s_ = positive;
  else                             s_ = negative;

  x_.swap(result);
  return *this;
}
```

Listing 17

digit is zero is sufficient for the latter since zero is the only number for which we allow one.

A further irritation is that fact that division is more difficult still. Once again we shall use the simple but inefficient scheme we learnt as children rather than the efficient but complicated schemes that we should want to use for any real-world implementation.

Specifically, we shall implement long division.

As it turns out, long division is algorithmically simplest in binary. This is chiefly due to the fact that in binary we only need to find how far to the left we can shift the number we are dividing and still be smaller than the current remainder, rather than guess at what multiple of it and the current power of 10 can be subtracted from the remainder.

For example, consider dividing 4735 by 68.

$$68\overline{)4735}$$

At first glance it might appear that there could be a 7 in the tens column, but on closer inspection we find it must be a 6, yielding

$$\begin{array}{r} 6 \\ 68\overline{)4735} \\ \underline{4080} \\ 655 \end{array}$$

It’s reasonably obvious that there must be a 9 in the units column, giving us

$$\begin{array}{r} 69 \\ 68\overline{)4735} \\ \underline{4080} \\ 655 \\ \underline{612} \\ 43 \end{array}$$

Performing the same division in binary requires more steps, but at none of them do we need to guess at the multiplicative factor that we shall place in each column; it is self-evident whether it should be a 0 or a 1.

Using the same example, but in binary, we have

$$1000100\overline{)1001001111111}$$

This first step in the calculation is trivially

$$\begin{array}{r} 1 \\ 1000100\overline{)1001001111111} \\ \underline{1000100} \\ 101111111 \end{array}$$

since this is largest shift which leaves a positive remainder.

Similarly, the remaining steps are

$$\begin{array}{r} 1000101 \\ 1000100\overline{)1001001111111} \\ \underline{1000100} \\ 101111111 \\ \underline{1000100} \\ 1101111 \\ \underline{1000100} \\ 0101011 \end{array}$$

giving us again a result of 69 and a remainder of 43, but this time using nothing but shift, compare and subtraction operations.

The first thing we shall need is a helper function to perform the shift operation, given in listing 18.

The next thing we need is a function to calculate the number of significant bits, as shown in listing 19. Note that, once again, we are relying upon the fact that we do not allow leading zeros except for zero itself which consists of a single digit and also that the underlying representation must consequently have at least 1 digit.

```
void
data_shift(bignum::data_type &y,
    const bignum::data_type &x,
    const size_t shift,   const size_t bits)
{
  assert(!x.empty() && x.back()!=0);

  const size_t digits = (
     shift+bits+bignum::shift-1)/bignum::shift;
  y.assign(digits, 0);

  const size_t major = shift / bignum::shift;
  const size_t minor = shift % bignum::shift;

  for(size_t i=0;i!=x.size();++i)
  {
    const size_t lo = x[i]<<minor;
    const size_t hi = x[i]>>(
       bignum::shift-minor);

    y[i+major] |= lo;
    if(hi!=0)  y[i+major+1] |= hi;
  }
}
```

Listing 18

The implementation of division is fairly straightforward, as the description of binary long division suggests. Once again we shall use a helper function since we can calculate both the result of the division and the remainder at the same time, which may prove useful. Its implementation is given in listing 20.

Note that the original value is replaced by the remainder and the result is returned.

Furthermore it is necessary to check whether the divisor is zero since the main loop will never terminate if it is.

Since all of the work is being done in the helper function, the division operator is fairly simple, as shown in listing 21.

Arbitrary precision fixed point

Using our arbitrary precision integer type as the underlying representation for a fixed point type, we can trivially solve the problem of overflow. If we are only performing additive operations on numbers that are sure to be representable with some specific number of decimal places, then we are set.

```
size_t
data_bits(const bignum::data_type &x)
{
  assert(!x.empty());
  size_t n = (x.size()-1)*bignum::bits;

  bignum::digit_type digit = x.back();
  while(digit)
  {
    digit >>= 1;
    ++n;
  }
  return n;
}
```

Listing 19

```
bignum::data_type
data_divide(bignum::data_type &x1,
   const bignum::data_type &x2)
{
  if(x2.back()==0)
     throw std::invalid_argument("");

  const size_t bits2 = data_bits(x2);

  bignum::data_type y;
  bignum::data_type result(x1.size(), 0);

  while(data_compare(x1, x2)>=0)
  {
    const size_t bits1 = data_bits(x1);
    size_t shift = bits1-bits2;

    assert(shift>0 || data_compare(x1, x2)==0);

    data_shift(y, x2, shift, bits2);
    if(data_compare(x1, y)<0)
       data_shift(y, x2, --shift, bits2);
    data_subtract(x1, y);

    const size_t major = shift / bignum::shift;
    const size_t minor = shift % bignum::mask;

    result[major] |= 1U<<minor;
  }

  data_strip_leading_zeros(result);
  return result;
}
```

Listing 20

Unfortunately it does nothing to solve the problems of truncation error, cancellation error or order of execution; as a case in point consider 1/3.

In addition it is not particularly efficient in time or space; to represent the maximum double precision IEEE value would take 128 bytes, as opposed to 8 for the former.

So, unfortunately, arbitrary precision fixed point also seems to be something of a lame duck as far as general purpose arithmetic is concerned.

Quack quack. ■

References and further reading

[Press92] *Press et al, Numerical Recipes in C* (2nd ed.), Cambridge University Press, 1992.

```
bignum &
bignum::operator/=(const bignum &x)
{
  x_ = data_divide(x_, x.x_);

  if(s_==x.s_ || x_.back()==0) s_ = positive;
  else                         s_ = negative;

  return *this;
}
```

Listing 21

Interface Versioning in C++

Updating a widely used DLL interface is non-trivial. Steve Love presents a practical approach.

Having code that requires more than a single (binary) version of a shared library is what is commonly referred to as 'DLL Hell' in Windows. This can arise in a number of ways, but in particular, a new version of a component is released which must be consumed by clients of the previous version, as well as the new. Just as there are several reasons this might occur, there are varied methods of handling the problem, from the requirement for client code to handle unsightly conversions all the way through to down-right nasty and undefined, or at best, non-portable, behaviour. The approach described here attempts to find a best-of-all-worlds leading to reasonable client code and well-defined, portable implementation.

The problem

Consider a shared library[1] which is used by multiple clients. Further, that the library is developed and maintained by a team (or company) that is different from those managing the clients. Any change to the library which causes the client code to recompile is a potential deployment nightmare. Release cycles need to be synchronised. Client teams need to co-ordinate the release of the shared library so that all interested clients either upgrade simultaneously, or else are quarantined to remain on the old version. In short, a lot of to-ing and fro-ing from all the parties involved.

The real problem is one of dependency management. If clients didn't need to recompile, there would be no problem with deploying the new version of the library, since (by definition) the client interface of the library must be unchanged. However, this would be an unacceptably onerous restriction on new library versions; the reality is that interfaces do grow stale from time to time, as new features are required.

The goal

This article explores how to add new methods to a class interface in such a way that client code need not recompile and redeploy if a new version of the library is released. Other changes to the library cannot be supported easily; deleting a method, or changing a signature (which is equivalent to adding a new method and removing the old one) aren't handled. It's possible to mark an old method as deprecated, allowing client code say one release cycle to change their code, but for the purposes of this article, a version upgrade means adding a new method to an existing type, or adding a new type.

This latter change is easily handled -clients running against a new version have no knowledge of the new type, and so cannot be dependent on it. This also applies to free-standing functions.[2] The interesting case is adding a new method to a type already in use.

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He can be contacted at steve@arventech.com

```
// part.h
#pragma once

namespace inventory
{
  class part
  {
    public:
      unsigned id() const;
    private:
      unsigned num;
  };
}
```

Listing 1

Consider the code in listing 1 (a concrete library). This code is intended to be part of a shared library called **inventory**. The client code is shown in listing 2.

As it stands, adding a new method to the part class would require the client application to recompile against the new header, relink with the new library and redeploy at the same time as the new library is released.

The goal is that when a new method is added to part, none of the above need to happen -the existing released client application will work against the new library version without changes.

```
// app.cpp
#include <part.h>
#include <iostream>
#include <memory>

// Also link to inventory.lib

int main()
{
  using namespace std;
  using namespace inventory;

  unique_ptr< part > p( new part );
  cout << p->id() << endl;
}
```

Listing 2

1 For the purposes of this discussion, the language is C++ and the platform is Windows and DLLs. However, the principles described also apply to other platforms and languages to a greater or lesser degree. In any case, a stable and consistent ABI between clients and libraries is presumed.

2 Care must be taken with adding a new overload for an existing function, of course!

with most compilers, the undefined behaviour is that the program will still appear to work

Undefined behaviour is the result if the client doesn't recompile, but much worse than that is with most compilers, the undefined behaviour is that the program will still appear to work in this example. The reasons why are explored in 'False hope' (below).

But first things first.

Untying the knot

Any change to the library which requires the client to recompile also requires the client to redeploy. Briefly, the changes which might cause that are:

1. Changes to the public member functions
2. Changes to the base-class list.
3. Changes to any protected or private member functions
4. Changes to any data members (presumed to be private in any sane system).

For all except number 1, there is a simple solution: introduce a level of indirection. If the client code depended on a true interface type representing a part, instead of a concrete class, then any implementation details would be encapsulated by an implementing type, rather than the interface itself. If any base classes are required by the resulting interface type, then those must also be made into interfaces, because 'Abstractions should not depend upon details. Details should depend upon abstractions.' [Martin96].

In the example shown in listing 1, only the private data needs hiding.

Listing 3 shows the changes required to make **`part`** an interface. Notice the simple factory function to create a new instance. This is required since the type that implements the **`part`** interface, **`realpart`**, is in effect *private* to the library. For brevity, the necessary plumbing to make part a complete interface type, such as handling or prohibiting copying, have been left out.

The public-facing interface for the library now hides all the implementation details from clients, such as data members and private or protected member functions. These are significant changes, because it means that **`realpart`** can change in any way, as long as it correctly implements the **`part`** interface, without requiring any recompilation on the part of clients.

The introduction of **`part`** as an interface is a necessary change to achieve the goal of being able to add a method to that interface, but it is not sufficient to achieve it. Adding a method to the interface *still* requires client code to recompile.

```
// part.h
#pragma once
namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual int id() const = 0;
  };
  part * create_part();
}

// part.cpp
#include "part.h"
namespace
{
  class realpart : public inventory::part
  {
    public:
      realpart();
      int id() const;
    private:
      int num;
  };
}
namespace inventory
{
  part * create_part()
  {
    return new realpart;
  }
}

// app.cpp
#include <part.h>
#include <iostream>
#include <memory>

// Also link to inventory.lib
int main()
{
  using namespace std;
  using namespace inventory;
  unique_ptr< part > p( create_part() );
  cout << p->id() << endl;
}
```

Listing 3

False hope

Listing 4 shows an update to the part interface – a new method has been added. Even though the client code doesn't use the new method, it must recompile against the new interface definition, and still needs to re-deploy at the same time as the new library is deployed.

Suppose for a moment that the new library were deployed, and client code remained as it was – using the old interface. The client code had compiled against a definition of **`part`** that had only one method, and the deployed library has a **`part`** type that has two methods.

It is not safe to **depend on the order** of the vtable matching the order of declaration

```
??_7properties@@6B@ DD FLAT:??_R4properties@@6B@ ; properties::`vftable'
  DD FLAT:?integer@properties@@EBEHXZ
  DD FLAT:?floatingpoint@properties@@EBENXZ
; Function compile flags: /Odtp
```

Figure 1

```
??_7properties@@6B@ DD FLAT:??_R4properties@@6B@ ; properties::`vftable'
  DD FLAT:?integer@properties@@EAEXH@Z
  DD FLAT:?integer@properties@@EBEHXZ
  DD FLAT:?floatingpoint@properties@@EAEXN@Z
  DD FLAT:?floatingpoint@properties@@EBENXZ
; Function compile flags: /Odtp
```

Figure 2

Common – but wrong

It's a fairly common practice in, for example, COM to enhance an existing interface by adding a new method to the end.[3]

This technique works, but does result in undefined behaviour, due to the one definition rule. However, since COM is defined according to strict compilation rules, and uses IDL to precisely specify object layout, this can be passed-off as platform-specific behaviour -just taking advantage of the code generated by the right compiler. It is not considered good practice in any case; interfaces are supposed to be immutable.

However, COM is not C++ -at its most basic level it is C, and so therefore doesn't use the C++ virtual despatch mechanism.

Even so, adding methods to the end of the interface isn't the real problem.

The real problem is with the implementation class, **`realpart`**.

Out of order

Changing the methods in a pure abstract class in C++ doesn't cause much of a problem at runtime (which is the point in the lifecycle about which we're most concerned here) because at the end of the day, a C++ interface is largely a compile time animal; it's purpose has to do with *type*, something the runtime environment knows and cares little about.

In order to see the real problem here, we'll have to start looking at the assembly code. The following examples were compiled with the Microsoft C++ compiler from Visual Studio 2010 (version 16 of `cl.exe`).

Listing 5 shows a very simple polymorphic class, **`properties`**. It exposes two virtual functions, **`integer`** and **`floatingpoint`**. The fact that they're inlined is not relevant. Note, however, they are not pure virtual, so a "real" vtable is defined. This file is then compiled with the following command:

```
cl /EHs /FAs test.cpp
```

3 There are many caveats to this regarding changing UUIDs which are not really relevant to this article.

`/EHs` means use ordinary C++ exceptions only (synchronous exceptions). **`/FAs`** causes the compiler to generate an assembly file `-test.asm` – with inline-source included. The interesting entries can be found by searching for **`properties::`vftable`** (That's a back-tick character there).

The first such entry shows the general layout of the **`properties`** class, including RTTI descriptors. The second instance shows the layout of the virtual function table, and should look similar to Figure 1.

This section shows the physical storage for the vtable – the order of entries in it.

If the **`properties`** class is now changed to that shown in listing 6, with new method overloads for the same names, and recompiled with the same options, the result is as shown in Figure 2

What's really interesting about this result is the order of entries in the vtable. Refer back to listing 6, and compare the order.

What has actually occurred is that functions with the same name are grouped together, even though the actual order of declaration split the functions by getter and setter behaviour. It should be obvious what this means for our proposed method of adding new functions to the bottom of an interface:

It won't work.

It is not safe to depend on the order of the vtable matching the order of declaration. It's therefore unsafe to use a new version of the library without recompiling against its declared classes. Without that recompilation, when the client code calls on a virtual function, the wrong entry in the vtable is invoked (in this example), ultimately calling the wrong function. The results of that are hard to guess. Depending on the vtable order for a particular compiler is, at best, non-portable.

The true path

As was previously mentioned, this means that turning the **`part`** type into an interface isn't sufficient, on its own, to achieve what we need, but it is a necessary step.

instead of adding methods to an interface which is part of a deployed library, the new methods are added to a new interface

```
// part.h
#pragma once
#include <string>
namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual int id() const = 0;
      virtual std::string name() const = 0;
  };
  part * create_part();
}

// part.cpp
#include "part.h"
namespace
{
  class realpart : public inventory::part
  {
    public:
      realpart();
      int id() const;
      std::string name() const;
    private:
      int num;
      std::string namestr;
  };
}
namespace inventory
{
  part * create_part()
  {
    return new realpart;
  }
}

// app.cpp
#include <iostream>
#include <memory>
#include <inventory.h>

// Also link to inventory.lib
int main()
{
  using namespace std;
  std::unique_ptr< inventory::part > p(
     inventory::create_part() );
  cout << p->number() << endl;
}
```

Listing 4

```
class properties
{
public:
  virtual int integer() const { return 0; }
  virtual double floatingpoint() const {
     return 0; }
};

int main()
{
  properties p;
}
```

Listing 5

```
class properties
{
  public:
    virtual int integer() const { return 0; }
    virtual double floatingpoint() const {
      return 0;
    }
    virtual void integer( int ){}
    virtual void floatingpoint( double ){}
  };
  int main()
  {
    properties p;
  }
```

Listing 6

The solution hinges around an observation made earlier in this article: adding a new type to a library is easily handled – clients running against a new version have no knowledge of the new type, and so cannot be dependent on it.

Extending interfaces

The basic premise of this solution is that instead of adding methods to an interface which is part of a deployed library, the new methods are added to a new interface.

The key to this working is that the new interface inherits publicly from the existing one.

Listing 7 shows the basic interface, `part`, which introduces simple get properties called `number` and `name`, along with how the client code may use it.

New requirements arise to have the properties' values set by the client.

Listing 8 introduces `part_v2`, which extends `part` to add setters for the properties. Note that the names are (deliberately) overloaded, and imported to `part_v2` with `using` statements.[4].

A new version of the same compiler might choose to **group the functions** in a different way

```
// part.h
#pragma once
#include <string>

namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual unsigned id() const = 0;
      virtual const std::string &
         name() const = 0;
  };
  part * create_part();
}

// app.cpp
#include <part.h>
#include <iostream>
#include <memory>

// Also link to inventory.lib
int main()
{
  using namespace std;
  using namespace inventory;
  unique_ptr< part > p( create_part() );
  cout << p->id() << endl;
  cout << p->name() << endl;
}
```

Listing 7

Existing clients (as shown in listing 7) have no need to recompile, since the object returned from the factory is still an ordinary `part`, which has not changed. New clients wishing to take advantage of the new functionality, such as in listing 8, must compile against the new library.

A wrong turn

The interface required to extend the original `part` type has been presented (as `part_v2`), but what of the implementation? The factory must have something to create, and clients must, ultimately, have a real implementing object to do real work.

Listing 9 shows how one might approach the problem. Since clients only ever use the interface, `part`, the details of the implementing class are irrelevant.

4 In reality, a new header file for `part_v2` would be better than adding the new version to the end of the existing file.

```
// part.h

#pragma once
#include <string>
namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual unsigned id() const = 0;
      virtual const std::string &
         name() const = 0;
  };
  class part_v2 : public part
  {
    public:
      using part::id;
      using part::name;
      virtual void id( unsigned val ) = 0;
      virtual void name(
         const std::string & val ) = 0;
  };
  part * create_part();
}

// app.cpp
#include <part.h>
#include <iostream>
#include <memory>

// Also link to inventory.lib
int main()
{
  using namespace std;
  using namespace inventory;
  unique_ptr< part_v2 > p(
     dynamic_cast< part_v2* >( create_part() ) );
  p->id( 100 );
  p->name( "wingnut" );
  cout << p->id() << endl;
  cout << p->name() << endl;
}
```

Listing 8

This code, however, suffers the same problem as the examples in section 2; it is the vtable of the *implementing* class that causes the problem, not the interface at all.

As it happens, using the same compiler and flags as before, we can see that this approach actually works in practice. For brevity, the code for `part`,

```
    ??_7realpart@@6B@ DD FLAT:??_R4realpart@@6B@ ; realpart::`vftable'
      DD FLAT:?id@realpart@@UBEHXZ
      DD FLAT:?name@realpart@@UBEABV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@XZ
    ; Function compile flags: /Odtp

    ??_7realpart_v2@@6B@ DD FLAT:??_R4realpart_v2@@6B@ ; realpart_v2::`vftable'
      DD FLAT:?id@realpart_v2@@UBEHXZ
      DD
    FLAT:?name@realpart_v2@@UBEABV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@XZ
      DD FLAT:?id@realpart_v2@@UAEXH@Z
      DD
    FLAT:?name@realpart_v2@@UAEXABV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@@Z
    ; Function compile flags: /Odtp
```

Figure 3

```
#include "part.h"

namespace
{
  class realpart : public inventory::part_v2
  {
    public:
      realpart();
      unsigned id() const;
      const std::string & name() const;
      void id( unsigned val );
      void name( const std::string & val );
    private:
      unsigned num;
      std::string namestr;
  };
}

namespace inventory
{
  part * create_part()
  {
    return new realpart;
  }
}
```

Listing 9

part_v2, plus both complete versions of **realpart** have been put in a single file.

Listing 10 shows two versions of the **part** interface, and two independent implementing classes. **realpart_v2** represents the code from listing 9 – a complete implementation of the **part_v2** interface.

Compiled with the Microsoft Visual Studio 2010 C++ compiler as before:

```
cl /EHs /FAs test.cpp
```

figure 3 shows the vtable layouts corresponding to **realpart** and **realpart_v2**. As you can see, the compiler has helpfully laid the **realpart_v2** vtable out by ensuring that the *derived* interface, **part_v2**, appears entirely after the base interface **part**.

This makes sense: **realpart_v2** derives directly from **part_v2**, which derives from **part**. It would be easy at this point to consider the job done.

It's still not portable however. The code for **realpart_v2** is still dependent upon the implementation specific behaviour of the compiler in laying out the vtable this way. A new version of the same compiler might choose to group the functions in a different way, again resulting in undefined behaviour unless client code recompiles.

Virtually done

As with splitting the interface in two so that new methods are added by creating a new interface, so the implementation is split in a similar fashion, and follows the same pattern.

```
#include <string>

class part
{
  public:
    virtual int id() const = 0;
    virtual const std::string & name() const = 0;
};
class part_v2 : public part
{
  public:
    virtual void id( int val ) = 0;
    virtual void name(
       const std::string & val ) = 0;
};

class realpart : public part
{
  public:
    int id() const { return num; }
    const std::string & name() const {
       return namestr; }
  private:
    int num;
    std::string namestr;
};

class realpart_v2 : public part_v2
{
  public:
    int id() const { return num; }
    const std::string & name() const {
       return namestr; }
    void id( int val ) { }
    void name( const std::string & val ) { }
  private:
    int num;
    std::string namestr;
};

int main()
{
  realpart r1;
  realpart_v2 r2;
}
```

Listing 10

The **part** class interface in listing 11 is identical to that in listing 8, showing the **part_v2** interface deriving from **part**. The implementation of **part_v2** in a new class, **realpart_v2**, which derives not only from **part_v2**, but from the original **realpart** concrete implementation of the **part** interface.

```
// part.h
#pragma once
#include <string>
namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual unsigned id() const = 0;
      virtual const std::string &
         name() const = 0;
  };
  class part_v2 : public part
  {
    public:
      using part::id;
      using part::name;
      virtual void id( unsigned val ) = 0;
      virtual void name(
         const std::string & val ) = 0;
  };
  part * create_part();
}
// part.cpp
#include "part.h"
namespace
{
  class realpart : public inventory::part
  {
    public:
      realpart();
      unsigned id() const;
      const std::string & name() const;
    protected:
      unsigned num;
      std::string namestr;
  };
  class realpart_v2 : public inventory::part_v2,
     public realpart
  {
    public:
      using part::id;
      using part::name;
      void id( unsigned val );
      void name( const std::string & val );
  };
}
namespace inventory
{
  part * create_part()
  {
    return new realpart_v2;
  }
}
```

Listing 11

The **realpart_v2** class uses implementation inheritance to bring in the pure-virtual declarations *and* implementations of the **part** interface, and with **using** declarations brings those names into scope to allow them to be overloaded with the new, setter, functions. Finally, the factory function **create_part** is changed to return an instance of the *new* implementing class.

In order to achieve the required behaviour, the data members of **realpart** (the base class) have been made protected, since **realpart_v2** inherits from **realpart** and requires access to those members. A (possibly) neater but more verbose way of achieving this would be to add protected data

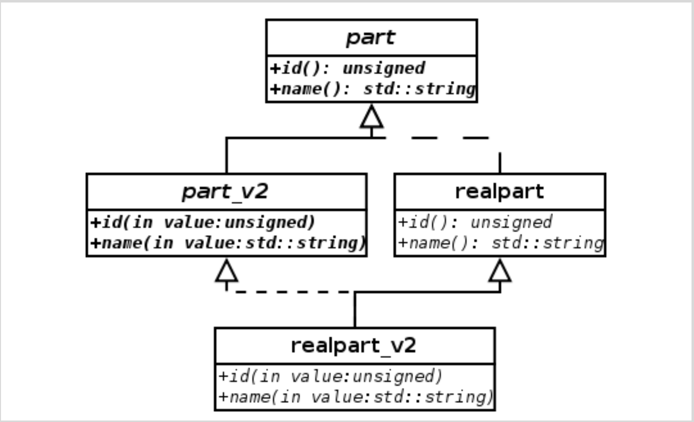


Figure 4

accessors to **realpart**. Since client code has no knowledge of either implementing type, protected data in this instance is not a great problem.

Much more of a problem is the fact that this code will not (or at least, *should* not) compile.

Ambiguity banishment

Figure 4 shows the problem in stark relief. The relationships between **realpart**, **realpart_v2**, **part** and **part_v2** form the dreaded diamond of multiple inheritance nightmares. The difficulty in compiling comes from the ambiguity between the views of **part** as seen by **realpart_v2**: one via **part_v2**, and another via **realpart**.

The solution to this is to use *virtual* inheritance, a technique that should be infrequently required, but is essential in this instance.

Listing 12 shows the changes required.

part_v2 must virtually inherit from **part**, as must **realpart**. So, indeed, must **realpart_v2**, since it is intended as a base class for a (as yet non-existent) new extension, **realpart_v3**.

This is an ABI-breaking change in **realpart**, causing clients to recompile, because it changes the way the vtables are organised. In order to take advantage of the technique, it is necessary to plan for the future and ensure all classes derive virtually *from the outset* to avoid ambiguity.

Virtual inheritance is normally used to avoid ambiguity between the data members of a base class that appears twice in the inheritance family. The ambiguity here is caused not by data members, but by the necessity of overriding pure virtual functions. Without the virtual inheritance, **realpart_v2** remains abstract, since it overrides only *one* set of **part**'s functions, which are pure virtual. Virtual inheritance ensures that only one instance of the multiply-inherited base class appears in the derived class.

Finishing polish

With the technical problems solved, the necessary infrastructure is in place to achieve the primary goal of allowing the shared library to redeploy without requiring client code to also recompile and redeploy. However, there is more that can still be done to make the necessary client code easier to use.

Names have power

Note in listing 12 that the factory function, **create_part**, continues to return a **part** pointer. This cannot change to return a **part_v2**, because that would violate the one-definition rule, morally the same as changing a member function on an interface. Clients of the original library don't care, but clients of the new version (as in listing 8) must cast the result to the new version.

Ideally, clients should be able to use the result of the factory *out of the box*, confident that its type is the latest version, and that if the library is updated under their feet, so to speak, it will continue to work as before.

We can, in fact, go further than that.

```
// part.h
#pragma once
#include <string>
namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual unsigned id() const = 0;
      virtual const std::string & name() const
         = 0;
  };
  class part_v2 : public virtual part
  {
    public:
      using part::id;
      using part::name;
      virtual void id( unsigned val ) = 0;
      virtual void name(
         const std::string & val ) = 0;
  };
  part * create_part();
}

// part.cpp
#include "part.h"
namespace
{
  class realpart : public virtual inventory::part
  {
    public:
      realpart();
      unsigned id() const;
      const std::string & name() const;
    protected:
      unsigned num;
      std::string namestr;
  };
  class realpart_v2
     : public virtual inventory::part_v2,
     public realpart
  {
    public:
      using part::id;
      using part::name;
      void id( unsigned val );
      void name( const std::string & val );
  };
}
namespace inventory
{
  part * create_part()
  {
    return new realpart_v2;
  }
}
```

Listing 12

It would be nice if the new version of the interface could be named the same as the old one. Then, clients who now require the new functionality don't need to find all the places they refer to `part`, and rename to `part_v2`.

We can go further still, and take the explicit responsibility for managing the lifetime of the returned object away from the client.

Called by a common name

Since the second version of the interface is `part_v2`, it makes sense to call the first one `part_v1`, and have something else which clients can refer to as `part`. An initial idea might be to make `part` a typedef of whatever

```
#include "part_v1.h"
#include <memory>

namespace inventory
{
  typedef part_v1 part_current;
  typedef std::unique_ptr< part_current > part;
}
```

Listing 13

```
namespace inventory
{
  INVENTORY_LIB part_v1 * create_part_ptr();

  template< typename type_version >
  part create_part()
  {
    return part( dynamic_cast< type_version * >(
       create_part_ptr() ) );
  }
}
```

Listing 14

the latest version of the `part` interface is, but even better than that, the goal of removing explicit management of lifetime away from the client can be met by making `part` a typedef to a smart pointer. Visual Studio 2010 comes with the right tool for this job as part of its C++0x (actually C++TR1) libraries.

Listing 13 shows a simple scheme to achieve this. The typedef `part_current` is used by the client to insulate it from the actual name of the latest interface version. When a new version of the part interface is added (e.g. `part_v2`), the `part_current` typedef also needs to change to reflect that.

Cast-free client

It has already been pointed out that the factory function used to instantiate a `part` cannot be modified to just return a pointer to whatever the current interface version is. Even using the typedef described above for this still results in a modified function when the typedef changes.

If clients are to be agnostic with regard to the actual version of the `part` interface, then it follows that there needs to be some intermediate place that can sensibly perform the right cast, and return the correct instance to the client.

This is crying out for a simple function template that just performs the right cast on the returned pointer from `create_part()`.

Listing 14 shows how this can be done. The new `create_part()` function now calls into the renamed `create_part_ptr()` factory which performs the actual instantiation.

Making `create_part` a template neatly sidesteps the one-definition rule violation; a function specialised on a new version of the interface is a *different* function, and being a template, is compiled into the client, *not* the library. It does mean, however, that clients must still refer to the name of the interface's current version, and this is where the `part_current` typedef comes into play.

Listing 15 shows how these two facilities are used by the client.

Dependency management

The final piece of the puzzle is how to organise the libraries so that the facilities are all available, without placing undue dependency strain on clients. The key to this is in the principle alluded to earlier – that abstractions should not depend upon details.

It's a prime-directive of our craft -separate interface from implementation – and to that end, the library will be split into two parts. One part contains *only* the interfaces necessary for clients to refer to objects. The second part,

```
#include <part_factory.h>
#include <iostream>

int main()
{
  using namespace std;
  using namespace inventory;

  part p = create_part< part_current >();
  cout << p->number() << endl;
  cout << p->name() << endl;
}
```

Listing 15

```
// part_v1.h
#include <string>
namespace inventory
{
  class INVENTORY_LIB part_v1
  {
    public:
      virtual ~part_v1();
      virtual unsigned number() const = 0;
      virtual const std::string & name()
         const = 0;
  };
}

// part.h
#pragma once
#include "part_v1.h"
#include <memory>
namespace inventory
{
  typedef part_v1 part_current;
  typedef std::unique_ptr< part_current > part;
}
```

Listing 16

the implementation, also contains the necessary facilities to instantiate objects of the required interfaces.

This separation means that client code that has no need to create objects need depend only on the interfaces themselves.

Interface-only project

The main currency of the library here is the **`part`** type, which is actually an alias for a smart pointer to a specific version of an interface. It therefore makes sense that the definition of the name **`part`** exists in the context of the interface.

Listing 16 shows the contents of the library. This *interface-only* library is also the one that will be used by the most clients, and so should bear the name **`inventory`**. Client code need only include `part.h`, and ignore **`part_v1`** as effectively implementation detail.

The real thing

The implementation of the interface, and the means to instantiate it, are the responsibility of the second library.

Since it's expected to have fewer dependents than the interface library, the implementation library in listing 17 can have a less obvious name, e.g. **`inventory_impl`**. Note the virtual inheritance in listing 17; even though there is as yet no multiple inheritance occurring, the base must be derived *virtually* to avoid breaking changes when a new interface is added.

Version up!

When the time comes to add new functionality, four things are required (Listing 18):

```
// realpart.h
#pragma once

#include <part_v1.h>

namespace inventory_impl
{
  class realpart_v1 : public virtual
inventory::part_v1
  {
    public:
      realpart_v1();
      virtual unsigned number() const;
      virtual const std::string & name() const;

    private:
      unsigned num;
      std::string namestr;
  };
}

// part_factory.h
#include <part.h>

namespace inventory
{
  INVENTORY_LIB part_v1 * create_part_ptr();

  template< typename type_version >
  part create_part()
  {
    return part( dynamic_cast< type_version * >(
create_part_ptr() ) );
  }
}

// part_factory.cpp
#include "realpart.h"
#include <part.h>

namespace inventory
{
  using namespace inventory_impl;

  INVENTORY_LIB part_v1 * create_part_ptr()
  {
    return new realpart_v1;
  }
}
```

Listing 17

1. Add a new interface to the inventory project
2. Update the **`part_current`** typedef
3. Add a new class to implement the new interface
4. Return an instance of the new implementing class from the factory

For brevity, the **`realpart_v1`** and **`realpart_v2`** code shares the same file.

At this point, the shared library can be released, and existing clients can upgrade at leisure. The reason this works is due to the code in listing 19.

The template function compiled into those clients would effectively be as follows:

```
part create_part()
{
  return part( dynamic_cast< part_v1 * >(
     create_part_ptr() ) );
}
```

Even though the new version of the library is returning an object which now derives from **`part_v2`**, those clients have no knowledge of that fact.

```
// part_v2.h
#include "part_v1.h"
#include <string>
namespace inventory
{
  class INVENTORY_LIB part_v2
     : public virtual part_v1
  {
    public:
      using part_v1::number;
      using part_v1::name;
      virtual void number( unsigned ) = 0;
      virtual void name(
         const std::string & ) = 0;
  };
}

// part.h
#include "part_v2.h"
#include <memory>
namespace inventory
{
  typedef part_v2 part_current;
  typedef std::unique_ptr< part_current > part;
}

// realpart.h
#pragma once
#include <part_v2.h>
namespace inventory_impl
{
  class realpart_v1
     : public virtual inventory::part_v1
  {
    public:
      realpart_v1();
      virtual unsigned number() const;
      virtual const std::string & name() const;
    protected:
      unsigned num;
      std::string namestr;
  };
  class realpart_v2
     : public virtual   inventory::part_v2,
     public realpart_v1
  {
    public:
      using realpart_v1::id;
      using realpart_v1::name;
      virtual void number( unsigned );
      virtual void name( const std::string & );
  };
}

// part_factory.cpp
#include "realpart.h"
#include <part.h>

namespace inventory
{
  using namespace inventory_impl;
  INVENTORY_LIB part_v1 * create_part_ptr()
  {
    return new realpart_v2;
  }
}
```

Listing 18

```
#include <part.h>

namespace inventory
{
  INVENTORY_LIB part_v1 * create_part_ptr();
  template< typename type_version >
  part create_part()
  {
    return part( dynamic_cast< type_version * >(
       create_part_ptr() ) );
  }
}
```

Listing 19

Clients who now compile against the new version of the library effectively compile against this:

```
part create_part()
{
  return part( dynamic_cast< part_v2 * >(
     create_part_ptr() ) );
}
```

And so can see the new functionality.

Justifying the means

As with many things technical and otherwise, this solution is a trade-off between convenience and effort. The convenience is provided by the fact that the shared library is backwards-compatible with clients who are already deployed, and does not require them to recompile and be re-released with a new library version.

This convenience comes at the expense of the effort requried to understand the interfaces in use, along with the fairly advanced techniques required to make it work portably. Instead of a single point of reference for all the facilities offered by an interface, the user must now follow a chain of base interfaces to determine how to use the whole. Similarly, following the chain of implementing classes is a definite obstacle to comprehending the code.

The use-case for which this code was originally developed was specifically focussed on the deployment dependencies between library and clients, in particular allowing clients of a previous version to continue unchanged when a new library was deployed. The cost of extra complexity in understanding the library was accepted as a necessary one to provide this feature. It is an idiom in common use, however, and understanding the *idiom* can help to reduce the complexity of understanding the solution.

The original requirement is all about loosening the coupling between client and library, achieved by judicious use of interfaces, then splitting the library into separate interface and implementation libraries. This separation allows clients to choose whether or not a dependency on the implementation -and the factory to create one -is required. If it is not needed, then the client can restrict their dependency to just the interface library. ■

Acknowledgements

Many thanks to Roger Orr for identifying the real problem with extending interfaces by adding methods to the end. It was he who spotted that the vtable layout cannot be relied upon to match the declaration order, and who showed me how to find that information from the compiler. Thanks to Pete Goodliffe, Chris Oldwood and Frances Buontempo for providing valuable feedback on early drafts, and to all those who attended the presentation at Skills Matter in London, especially Sam Saariste, James Slaughter and Martin Waplington for making me think harder about it, and for pointing out some of the errors!

References

[Martin96] Robert C. Martin, 'The Dependency Inversion Principle', *C++ Report*, May 1996

Quality Matters Christmas Intermezzo

Sometimes it's good to reflect. Matthew Wilson considers what he's learnt so far.

Sadly, despite Ric Parkin's last-minute exhortation to contribute just 'a single nugget of wisdom, and [an apology that] the next part is delayed', work pressures, chronic procrastination, and falling foul of my own game of deadline-skirting-as-inspiration, I find myself coming up short for a contribution for what will be, righteously, a grand celebration of 100 instalments of Overload. Had I been able to do so, I might have imagined a Quality Matters Yearly Round-up in the form of one of those awkwardly self-regarding missives that circulate during the festive season. Something like:

> Well, another year has gone by in a flash. The family is just doing so well. Daddy has been juggling work, exercise, being a pompous ass, and writing his columns and books: He says he's happy, so we just let him get on with it. The older children – FastFormat, Pantheios, recls, and STLSoft – are all doing well, in their own way, and, despite their various 'challenges', plod along with reasonable success. FF has settled into his life as a niche player: he does his bit well, though we do worry that he's not making enough friends. recls and STLSoft still can't get their homework complete, and have yet to properly put on their school uniforms, even after all these years! What good is having good ideas if you can't even do up your tie, I ask you? We had hoped that, with all his quiet successes, our Pantheios would finally get all his course elements complete and go on to University; but no, another year in class B(eta)! He keeps making friends with C++ programmers from all around the world, he excels at sprinting, gymnastics, and composition. But we can't get him to pay attention in writing his essays or in presenting his work clearly. Perhaps with our youngest, CLASP, set to go to school this year – as part of an exchange program with her uncle Garth (Lancaster) – she can show her elder siblings how to do it all right from start to finish in 2011. We live in hope.

Thankfully, I didn't have time to write any such drivel. Instead, I'll say that it continues to be a real treat to write 'Quality Matters', and also a major effort; though in a good way. I am definitely living up to James Coplien's philosophy that 'an author should learn at least as much as his or her constituency does through the process of writing and publication'. When I started, I knew that there were some sacred cows that I would be slaughtering, but I never imagined how many. That I've worked out that *hello, world* is wrong is still quite amazing to me. ('Uncle' Garth and I are about to embark on writing a C++ introductory programming book with a difference, and so I'll be putting myself in the position of all those authors whose first code examples to their readerships were wrong. Eeep!) That I've demonstrated – to myself at least; to QM readers in two instalments'time – that exceptions are antithetical to correctness, yet essential for robustness, and that there's merit in Java's checked exceptions, is another entirely unanticipated outcome of this enterprise.

Next year (and probably the one after, given my proven difficulties in hitting all the QM deadlines) will see me facing down more difficult areas, several of which I'm (as yet) imperfectly prepared for. The first two will be the promised third and fourth in the exceptions series, about which I have a reasonably clear idea of what's coming. Next I'm going to have to tackle the issue of contract programming and irrecoverability: adherents of the eggregious practice of defensive programming better get their defenses ready, as I plan to take no prisoners on that subject. Then we're probably going to get into diagnostics. All of those subjects I'm reasonably clear on. After that, it'll be unchartered territory: defining 'debugging' and examining its uses; seeing how far TDD can be taken; looking at coding with interface layers for increased reliability (and thread-safety). I'm also going to bite the bullet – and embrace my own (open-)sources of shame – and look at areas in which I've traditionally done very poorly: packaging and documentation. In this regard, I may use my own efforts to have Pantheios move out of its four years of beta and be a properly packaged and documented library, as the gritty exemplar.

Doubtless all of these subjects will see me learning more than I teach, and I will continue to be grateful that the Overload editor tolerates my reliability, and humbled that the Overload readers tolerate my presents (sic.), such that I have this forum in which to learn.

So, seasons greetings to all, and I look forward to blathering at you some more next year. :-) ■

Matthew Wilson is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au