overload|77

February 2007
ISSN: 1354-3172
www.accu.org

OVERLOAD 77

February 2007

ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Contributing editor

Paul Johnson
paul@all-the-johnsons.co.uk

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Alistair McDonald
alistair@inrevo.com

Anthony Williams
anthony.ajw@gmail.com

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Ric Parkin
ric.parkin@ntlworld.com

Roger Orr
rogero@howzatt.demon.co.uk

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 78 should be submitted to the editor by 1st March 2007 and for Overload 79 by 1st May 2007.

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.

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

The Power of Inertia

“The principle of inertia is one of the fundamental laws of classical physics which are used to describe the motion of matter and how it is affected by applied forces. Inertia is the property of an object to remain constant in velocity unless acted upon by an outside force.” [Wikipedia]

Inertia in individuals, teams and organisations

Like physical objects individuals, teams and organisations have the property that they will continue uniformly straight ahead unless acted upon by an external force. This can be a good thing, it can also be a bad thing. Sometimes it is necessary to find an external force sufficient to effect change.

When this happens in individuals it is often described as “a habit” – one can deliberately cultivate good habits (e.g. writing and running unit tests) and can accidentally fall into bad habits (e.g. failing to resynchronise the code being worked on with source control). Once developed these habits are unlikely to change unless some “external force” (perhaps in the form of a colleague’s comments) motivates a change.

Teams are the same – once a build & test server is in place reporting the status of the project after each check-in it will remain there (until something happens) but if it isn’t there very few teams will make the effort to put it into place. Actually, much the same can be said of source control.

When it comes to organisations the same is true, they will cling to processes and procedures that are demonstrably against their interests until there is sufficient pressure to cause a change.

It can be frustrating when trying to effect a change in a person, team or organisation – because it will often feel that one keeps pressing and pressing and nothing happens. On the other hand, once the desired change is, finally, under way it then has inertia on its side.

The latest in the ISO C++/CLI story

Since my editorial in Overload 75 about the ISO “Fast Track” last year there have been few obvious developments in the C++/CLI story. There has been no new revision of the standard submitted and the Ballot Resolution Meeting has been scheduled in Oxford (for the Friday, Saturday and Sunday at the end of the ACCU conference week).

Behind the scenes there have been things happening – the ECMA group responsible for submitting the standard (TC39/TG5) has been trying to decide what to do next. This isn’t as easy as it might appear – just as there is no provision for National Bodies to say “please withdraw this from the standards process”, there is no provision for a submitter to withdraw a standard once it has passed the “comments” stage.

And so, despite the significant concerns expressed by various experts in the field, things are rolling relentlessly on. As the ISO C++ working group is meeting in Oxford the week following the ACCU Conference, they should be well represented at the Ballot Resolution Meeting.

And now for something almost, but not entirely, the same

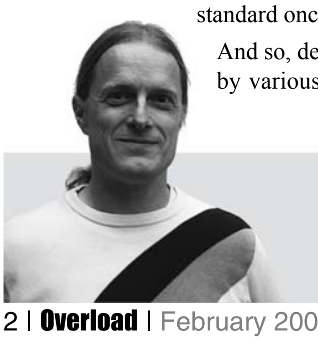
As expected, the ECMA Technical committee for Office Open XML Formats submitted the ECMA Office Open XML standard to ISO for fast track approval on the 5th January. According to their schedule of work [ECMATC45] this 6000+ page document is intended to standardise “Office Open XML” which is the latest way for Microsoft Office applications to store and interpret documents as files.

Making this information public is a commendable piece of openness on the part of Microsoft, but there is no advantage to the wider community from this being made an ISO standard. One does not have to read the whole thing to discover its true nature is documentation of a proprietary format. How else to explain the many references to the behaviour of Microsoft products (e.g. requiring the replication of bugs in Excel’s date handling) and the incorporation of Microsoft technologies that are clearly not open (e.g. Windows Metafiles).

One of the principles for ISO standards is “Consensus – The views of all interests are taken into account: manufacturers, vendors and users, consumer groups, testing laboratories, governments, engineering professions and research organizations.” [ISO] It doesn’t take much investigation to discover that most manufacturers of office software products are supporting the existing ISO standard for document formats. (For a list of products supporting the ISO OpenDocument format [ISOOpenDoc] see the list maintained by the OpenDocument Fellowship [OpenDocApps].)

As I noted before, the National Bodies have a thirty day period to identify “contradictions” that could prevent the standard being adopted by ISO. (So this period should be closed by the time you read this.) Andy Updegrove [Updegrove] summarises this process as follows:

> During the first one-month step, any member may submit ‘contradictions’, which, loosely defined, means aspects in which a proposed standard conflicts with already adopted ISO/IEC standards and Directives. Those contradictions must then be ‘resolved’ (which does not necessarily mean eliminated), and these resolutions are then presented back to the members during the second stage to consider as part of the voting package. During this second, five-month step, other objections, questions and comments can be offered by members.



Alan Griffiths is an independent software developer who has been using ‘Agile Methods’ since before they were called “Agile”, has been using C++ since before there was a standard, has been using Java since before it went server-side and is still interested in learning new stuff. His homepage is http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

There are also many users and consumers who do not feel that ISO standardisation of this standard is in their interests. There is even a collaborative (Wiki based) project to document contradictions on Grokdoc [Grokdoc]. It remains to be seen how many of the National Bodies can be persuaded to take action to oppose this standard as a result. (For the reasons I described in my earlier editorial the default position is, in practice, to vote for adopting a standard that is under discussion.)

In the case of Office Open XML, even more so than in the case of C++/CLI, the presumption that any standard being submitted to voting on by National Bodies has been properly reviewed is wrong. It is a large document (over 6,000 pages), has been created in a relatively short period of time and with a small number of reviewers.

There appears to be no advantage to anyone outside of Microsoft in ISO adopting this as a standard – when there are requirements like 'Emulate Word 6.x/95/97 Footnote Placement', this is not going to be easy for anyone else to implement. What is more, the licences I've seen for these products forbid reverse-engineering to discover how these features operate.

Despite all this I fear that Office Open XML will follow C++/CLI down the fast track. The responsible ISO (JTC-1) committee will not find the contradictions a sufficient cause to divert this standard from the Fast Track and the longer (5 month) voting period will begin. If this happens, a lot of effort will be required to produced sufficient informed "no votes" to force a ballot resolution meeting, and this may still decide to proceed with adoption.

The ISO Fast-Track process

I'm sure you'll realise by now that I feel that the ISO Fast-Track process isn't always working in the best interests of the IT industry (either consumers or producers) and not even in the interests of ISO itself.

I think the problem resolves to one thing – it doesn't impose a condition that there is sufficient interest from the National Bodies in the adoption of the standard. The first question that should be asked is not "does this standard contradict anything we have already?", but "do we want to work on adopting this as a standard?". The problems arise when too few of the National Bodies care enough to scrutinise a submission properly.

Hopefully, as a result of the input it is getting, JTC-1 will recognise the need for change and revise the process accordingly.

Overload and Overload

As I explained in my last editorial, there has been something of a crisis in the Overload team. You'll see that the names in the credits have changed yet again to reflect the new line-up. Thanks again to those leaving the team for efforts over the years that have passed, and to those joining the team for their efforts in times to come.

Thanks also to all of you that submitted articles – especially the few whose articles I had to turn down because they didn't quite seem to fit Overload. I do hope the feedback was helpful and that we will see more submissions from you in the future.

The current issue didn't quite run as smoothly as in the "golden age" I described last time – in particular I have to apologise to a couple of authors for feedback taking longer than it should have done, and also to the production editor for a slightly late delivery of the edited articles. We understand how to address these mistakes and – fortunately – no real harm was done. We intend to do better next time.

After the difficulties with Overload 76, I'm pleased to see that the new team has settled into the task and I confidently look forward to things running smoothly straight ahead for the foreseeable future.

All that remains to keep Overload going is for the rest of you to submit some articles for the next issue. And this is the time to do so: with many of the "usual suspects" focused on conference presentations, this is a chance for the rest of you to sneak in and write the article that everyone will be reading and talking about at the conference.

Those of you that are presenting should not rest on your laurels – you have the material for an article to hand – so now is the very best of times to write it up.

Overload on the web

My comments last time about making Overload available on the website seem to have overlapped with Allan Kelly making the archive publicly available. The result isn't what I've been intending: instead of a single PDF for each issue I plan for the articles to be presented separately, preferably as HTML, and for links to work. Once again, sorry for the delay.

CUTE download

I've had a short note from Peter Sommerlad to say that CUTE is now available for download [Sommerlad].

References

[Wikipedia] http://en.wikipedia.org/wiki/Inertia

[ECMATC45] http://www.ecma-international.org/memento/TC45.htm

[ISO] http://www.iso.org/iso/en/stdsdevelopment/whowhenhow/how.html

[ISOOpenDoc] http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=43485

[OpenDocApps] http://opendocumentfellowship.org/applications

[ECMA] http://www.ecma-international.org/activities/General/presentingecma.pdf

[Updegrove] http://www.consortiuminfo.org/standardsblog/article.php?story=20070117145745854

[Grokdoc] http://www.grokdoc.net/index.php/EOOXML_objections

[Sommerlad] http://wiki.hsr.ch/PeterSommerlad/wiki.cgi?CuTeDownload

Managing Technical Debt

Tom Brazier on budgeting for the cost of doing it wrong.

Short term vs. long term trade-offs

Most of us have experienced occasions where we've been required to take short-cuts to make delivery deadlines. These short-cuts are seen as bad by many prominent people in the software industry. In fact, Robert C. Martin puts it this way [Martin]:

The next time you say to yourself: "I don't have time to do it right." – stop! Rethink the issue. You don't have time not to do it right!

Nonetheless, short term hacks continue to proliferate and many developers are under a lot of pressure to make them. Given that the long term by definition lasts for longer than the short term, this is a problem.

In fact, it's a big old warty problem with attitude and bad breath. Taking short-cuts generally means that the next time the software is touched, it needs to be fixed before any further work can be done. So the second piece of work is even more expensive to do correctly than the original piece of work, and you can be sure that the deadlines are just as tight as they were the first time. Worse, developers generally prefer to play it safe – if someone has left them a dodgy-looking piece of code, they prefer to leave well enough alone. So, unless there are strong individuals present who are really dedicated to good engineering, the team takes another short-cut and works around the code affected by the previous short-cut[1]. The third change involves working around the first two short-cuts, and so on.

If one follows the trend to its logical conclusion, and in my experience many teams do, one finds that the code complexity grows at an increasing rate. After several changes to the software, it reaches the point where nothing can be changed without significant time and risk. Usually at some point, the team begins to realise that they need to fix the things they've broken. But by then it's too late because they are spending all their time just keeping a fragile system running and have no spare capacity to fix the code. They've painted themselves into a classic Catch 22 situation.

This article is not primarily about escalating long term costs, so I won't labour the point. We'll assume the above argument is sufficiently convincing. The trouble is that in many cases, a good counter-argument can be made for the short term benefits.

Consider a small company in a niche market – its entire future generally rests upon being first to market. Longer term software problems simply aren't important in this case because they are not visible to the customers until it's too late. And they are certainly less important to the software company for whom the long term won't exist unless it makes the sale in the short term.

Or take the financial industry. When new ideas come to the market, there is generally a small window during which very large amounts of money can be made. In this case, a company which spends too long writing the software might as well not have bothered in the first place.

Similar short term pressures occur on both large and small scale every day for any number of reasons.

As any weatherman will tell you, the short term is always clearer than the long term. So while short term costs might not be as great, they are easier to quantify. It can be very difficult, therefore, to determine just what the relative costs and benefits are when deciding whether to "do it the right way" or to take a "short-cut".

Technical debt

The phenomenon mentioned above of costs growing at an increasing rate led Ward Cunningham to liken poorly-engineered software to debt [Cunningham]. This turns out to be an extremely good analogy. We've all heard of people who've completely lost control of their credit cards, and who spend all the money they earn just servicing their debts. A company in the Catch 22 situation above is just like someone whose credit card debt has become out of hand.

Formalising the analogy, any time a software team follows bad engineering practices they incur two kinds of cost. First, there is the cost of repaying the "capital", i.e. undoing bad code and replacing it with well-engineered code. Second, there is the "interest", the ongoing increased cost of supporting, maintaining and enhancing the software.

It is generally fairly manageable to take on a small amount of technical debt, but if one doesn't pay the interest cost, or insists on taking on ever more and more technical debt, one soon loses control.

So the analogy serves as a great example because it allows us to draw on the commonly known human experience of taking on debt.

Managing technical debt

Having set the scene, we can now get to the real point of this article – how to manage technical debt. I observed above that sometimes it is vital to create software in the shortest possible time, regardless of quality. At other times it is not vital, but there is a lot of pressure nonetheless. At all times, it is hard to quantify what the long term benefits and costs are.

Human beings have thousands of years experience with managing financial debt. Can this experience teach us anything about managing technical debt? A little googling will confirm that many people, some of them quite prominent, think the answer is "yes".

Here are some key strategies for managing financial debt:

1. Only ever enter into debt if the benefits outweigh the costs. So, for example, a student loan, or a mortgage is generally considered good debt. In the long term the benefits of education or owning your own home outweigh the cost of the debt. Most credit card debt, on the

Tom Brazier has been fiddling with computers since getting a ZX Spectrum for his 11th birthday. Having progressed somewhat from Sinclair BASIC, he now works as a technical architect in the city. He can be contacted at tom@firstsolo.net

1 There is a good discussion about this under Tip 4, Don't Live With Broken Windows, in *The Pragmatic Programmer* [HuntThomas]

in many situations the **best outcome** for **society** as a **whole** only occurs when all **individuals** choose a **less than best** outcome for **themselves** personally

other hand, is not sensible because the exorbitant interest costs outweigh any benefit.

2. Know how you will repay the debt when you enter into it. This is pretty much mandatory with a mortgage, because the banks make it a prerequisite. Credit card debt, once again, often doesn't have this sort of planning associated with it.
3. Keep track of your debt. Ensure you are paying it off. Understand which debt is the most costly and pay that off first. These are the first principles that are discussed with people who've lost control of their financial debt.

If we adapt these to technical debt we get the following strategies:

Ensure the benefits outweigh the cost

This isn't as easy as it sounds because it is hard to quantify the cost. But we can work to educate our customers, and help them to see that there are generally significant long term costs caused by rushing in the short term.

Another practice, which seems obvious but is often neglected, is to understand why the customer thinks something is really urgent. Often software users give unreasonable deadlines out of ignorance and would be quite happy to allow more time if the software team talked to them. Other times, users ask for something they think will solve their problem, when a simpler and technically better alternative would do just as well.

Another idea is to break down the requirements into something that can be delivered quickly followed by something that will take a little longer. If you can get one or two high priority features in front of the users quickly, they will often be happy to wait for the lower priority ones.

The key points here are to communicate with the users and to be creative.

Know how you will repay the debt

Never take on technical debt without first spending some time thinking about a more strategic approach. Too often "tactical fixes" are at complete odds with any potential "strategic solutions" even though there exists an alternative short term approach which would go at least part way towards a strategic solution. With some thought you may be able to find a better tactical fix and reduce the debt straight away.

Having spent, say, half an hour working out a reasonable strategic direction, think about how and when the strategic solution will be implemented. Assign a value for the "capital", i.e. how many man days it will take to repay the debt.

Assign the debt an "interest rate". That is to say, get an idea of the increase in running costs caused by the debt – you might want to use several categories, like high, medium and low.

Finally, and this is really key, raise a change request ticket and add this information to your project plan.

Keep track of your debt

If you follow the advice above about adding technical debt to your project plan, then suddenly you'll be able to keep track of your level of debt. At any point, you can say, "I have X man weeks of outstanding high, medium or low interest technical debt". Then you can start setting thresholds, for example you may say that you won't accumulate more than 2 months of high interest debt, 4 months of medium interest debt or 6 months of low interest technical debt.

As debt increases, you can see it happening and you can therefore manage it. This will mean increasing resourcing or pushing back on some new work. But now you can justify the increased cost because you have real data about what needs to be fixed. This is far better than a vague, unspecified concern about poor engineering practice, or a system that just keeps failing unpredictably.

Keeping track of your debt also means re-assessing it from time to time because the capital cost of technical debt increases over time. There are at least two factors at work here. In the short term, the longer you wait before repaying the debt, the less you remember about how to repay it. This is particularly true when team members leave, taking the knowledge with them. Over the longer term, the technology industry moves on and it becomes harder to find people who know how to work with the technology in which the debt was incurred. For example, it's not as easy as it once was to find someone who can fix code written in FORTRAN or COBOL. Even C is moving (many would say has moved) into this camp.

So you need to keep this inflation effect in mind. Re-evaluate existing debt occasionally and be reticent about open-ended technical debt.

Does it really work?

Of course, there is no silver bullet. However, I have used some of the ideas above extensively and found they do help a lot. I have only fairly recently begun to follow the last of the suggestions labelled "keep track of your debt". So far it looks promising.

Googling confirms that I'm not the only one to find these ideas helpful.

In the long run we're all dead [Keynes]

One debt management strategy that hasn't been mentioned so far, but is frankly a very real for many people, is to leave the company (or even the country). How does one answer the team member who says, "What's in it for me, I won't be the guy who sees the long term benefits or costs"?

A less than motivational answer is that this is what we get paid for. So it's a matter of moral obligation.

For a more motivation answer we can appeal to the sense of a job well done.

But perhaps the best answer is a pragmatic one. If all software teams control technical debt well, then everyone's lot is improved. This introduces a whole new topic which deserves an article of its own, the Tragedy of the Commons. This is an observation that in many situations the best outcome for society as a whole only occurs when all individuals choose a less than best outcome for themselves personally.

The Tragedy of the Commons is related to a problem from Game Theory called the Prisoner's Dilemma. This is a game in which two players may either "co-operate" or "defect". The pay-offs are cleverly structured so that:

it is the responsibility of the software engineering team to take into account the short term benefits and costs as well as the long term benefits and costs

1. Each individual player is most motivated to defect.
2. The best pay-offs occur when both players co-operate.

According to Wikipedia [Wikipedia], Robert Axelrod [Axelrod] performed an experiment with automated agents which played an extension of the Prisoner's Dilemma. He found that greedy agents tended to fare worse than more altruistic agents in the long run.

So with a bit of hand waving and some smoke and mirrors, we can make a case for the pragmatic answer.

So what should you do?

The ideas in this article need buy-in from the entire software development team. Very importantly, they need buy-in from the team leader. So you'd think this article is aimed at software team leaders, letting the rest of us off the hook. Not so. In practice it is often the techies at the coal-face who have to exert influence over managers to get the point across.

Given that most of us are in a team of more than one, the first step will be to act as an influencer. Introduce the technical debt analogy. I've found that people tend to take to the basic idea quite quickly. From there, it is a matter of gently extending the analogy.

Finally, the ideas above are just a starting point. For example, at least one web article [ThinkBox] extends the analogy to concepts like taking payment holidays and making lump-sum repayments.

Conclusion

Let us assume that a software engineer's job is to create the most profitable software for the least cost. Then it is the responsibility of the software engineering team to take into account the short term benefits and costs as well as the long term benefits and costs. This is hard because we work with complex systems and incomplete data, Kevlin Henney would say that we lack visibility [Henney]. Thinking in term of managing technical debt like we'd manage financial debt gives us ways of collecting data about the long term and therefore increasing visibility. ■

References

[Martin] "The Tortoise and the Hare", Robert C. Martin, 2004, http://www.artima.com/weblogs/viewpost.jsp?thread=51769

[HuntThomas] Andrew Hunt and David Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999

[Cunningham] Ward Cunningham, *The WyCash Portfolio Management System,* 1992, http://c2.com/doc/oopsla92.html

[Keynes] John Maynard Keynes, *A Tract on Monetary Reform*,1923

[Wikipedia] *Prisoner's dilemma*, http://en.wikipedia.org/wiki/Prisoner's_dilemma

[Axelrod] Robert Axelrod, *The Evolution of Cooperation*, 1985

[ThinkBox] *Repaying technical debt*, November 2005, http://www.think-box.co.uk/blog/2005/11/repaying-technical-debt.html

[Henney] Kevlin Henney, *Five Considerations in Practice*, ACCU conference 2006

Programming – Abstraction by Design

Nigel Eke acts as a guide to aspect oriented programming using AspectJ as an example.

Two unrelated paragraphs...

Schadenfreude (pronounced 'Shar-den-froy-der'). This is a German word that has no direct translation into English. Look it up in a dictionary and it may get translated to 'gloat', but this is not the complete translation. Literally it means 'damage-joy'. Specifically it means 'taking delight in another's misfortune'.

The physicist Richard Feynman once served on a commission to select textbooks for schools. Many of the books were introducing 'the new maths' and set theory in particular. Feynman's comments on set theory were that it comprised new definitions for the sake of definition, a perfect case of introducing words without introducing ideas. ... specialised language should wait until it is needed, and the peculiar language of set theory is never needed. [Gleick]

This article introduces aspect-oriented programming, so why start with these two examples, especially when they appear to have no relationship to programming, let alone aspect-oriented programming?

If we take a look at programming, programming languages, and their history, a couple of concepts come to the fore. One is related to how we program, regardless of language. The second is related to what the programming languages themselves provide.

Before we dive into to some of the new concepts covered by aspect-oriented programming, I would like summarise what has happened during the development and creation of computer programming languages. However, if you really wish to get straight into the details then skip to the "Aspect-oriented programming" section.

Let us start by asking "What are we doing when we program computers?" In fact there is more than one answer to this question depending on your perspective. Generally we are telling the computer to follow a particular set of instructions to produce some output based on various information input into the program. But the program itself is not just a set of instructions to be executed. The program also comprises source code, which we have to read. The source code informs us, both as developers and future maintainers, what the intentions were when we originally wrote the code. Let us explore both of these briefly.

Programming a set of executable instructions

Program instructions are generally written in a programming language. They are translated by the language's compiler into the processor codes for a target processor. The codes get loaded into the target computer's memory and get executed. I use the term 'compiler' liberally here to cover genuinely compiled and interpreted languages.

If we take a look at the history of this translation step, a common theme emerges.

- Originally programs were loaded into the computer's memory by 'programming' through toggling the switches on the front panel of the computer. A program was then executed by loading the computer's program-counter register with the address of the start of the program. Once upon a time this used to be the only way to load a program into a computer – even if the 'toggled' program was just run once to read further programs from paper tape or disk-drives. As a minor aside – the external article [Wiki(1)] serves as tongue-in-cheek reminder of these 'Real Programmer' memories.
- Assembler languages provide mnemonics to represent the processor instruction codes. This enables the programmer to concentrate in terms of what the processor instructions actually do, rather than the binary code needed to perform those instructions. It also allows memory locations to be accessed by some descriptive name, rather than a physical address. A level of confidence is provided over and above the toggling of switch settings. It is easier to remember `ADD #2, R1;`[1] rather than loading two words of instructions with the values `65C1` and `0002`.
- High level languages – such as FORTRAN and COBOL – remove any dependency on the underlying processor architecture. This means that the programmer is no longer concerned with the instructions need to control the processor, but can focus on what the program itself is intended to do. High-level languages give us a level of confidence over and above having to choose the right registers for a processor instruction, or device address for an i/o operation. Writing:

```
IF Salary > 10000 GOTO nobonus;
    LET Salary = Salary + Bonus;
nobonus:
```

  is clearer, and less error-prone, than:

```
  MOV Salary, R0;
  SUB #2710, R0; 'Often Hex or Octal!
  BGT nobonus;
  MOV Salary, R0;
  ADD Bonus, R0;
  MOV R0, Salary;
nobonus:
```

- Structured languages – Algol-68, Coral-66 among others – are also independent of the underlying processor architecture. Their constructs provide more rigour than the procedural counterparts

1 Add the value '2' to the contents of register R1

Nigel Eke is a Software Engineer of some 30 years standing. He is a luddite who still uses a hand operated paper-tape punch to program. He also has a strange sense of humour. He is currently working and enjoying the lifestyle in Sydney, Australia, where he's planning on retiring and learning to use these new-fangled ASR-33 teletypes.

He can be contacted at me@nigel.eke.com

it is the vocabulary provided by the language keywords and how we use our naming that helps us write, understand and maintain programs

above, when describing the program flow. They also enable a divide and conquer approach to writing the program. This results in cleaner modularity and less of the monolithic spaghetti-code programs that tended to result from using the high-level languages.

- Object-oriented languages take this one stage further. These languages enable the program to use terms which focus on problem domain objects, e.g. `Customer`, `Account` or solution domain objects, e.g. `ButtonEventAdaptor`. This has the benefit that the modular responsibilities become more clearly defined than with structured approaches. Object-oriented programming also means that the relationships between objects are shown through inheritance, composition and aggregation. One of the benefits of inheritance is that it enables code reuse. Object-oriented programming languages take structure one stage further and give us a level of confidence regarding a module's responsibilities.
- There are also a few 'specialist' languages. Logic programming languages, such as Prolog, work simply by defining a set of predicates, goals and sub-goals, in order to seek out a solution. An automatic tree-search is performed for the prime goal. APL (Array Programming Language [Wiki (2)]) uses symbols, rather than text, to dictate what the program will do. These languages provide a method to write a solution in a form that serves a specific classes of problems.

Each one of these phases in the history of programming languages adds a level of abstraction over and above the previous level. Each advance or change in programming language provides a different way to describe the solution; each provides a different view of that solution with respect to the original problem.

This can also be said of the many programming languages that have not been given a mention [Wiki (3)]. The creator(s) of these languages must have a felt the need to design a language which provided something not available in other languages, albeit a new feature, new keywords, even a new syntax simply to make the compiler writer's job easier.

Program source code

Regardless of the language being used, the program source code also conveys other information.

Comments provide more information about a block of code, i.e. its intention – or perhaps we should say original intention, given the number of times code and comments become out of synch. Sometimes comments make statements about the requirements being satisfied.

Variable naming tells us what information is being held in that variable. Procedural naming tells us something out the task being performed by the procedure. Class naming tells us something of the responsibilities of that class and, perhaps, expected functions performed on objects of that class.

Preamble summary

So how does this relate to Schadenfreude and Feynman?

Correct, concise and consist use of names helps make a clearer program. A clear program provides us with confidence that the solution we're writing actually does work.

It will be better if we can use the term 'Schadenfreude' rather than a lengthy textual description. Of course I realise Schadenfreude is not the best description to use for an English speaking audience – again this comes back to choosing the right name in the right context.

Nevertheless, it is the vocabulary provided by the language keywords and how we use our naming that helps us write, understand and maintain programs.

The choice of the right tools for the job also helps us deliver clearer solutions to the problem. This means not only choosing the right language(s) but also the right subset of their keywords to satisfy the demands of your programming environment. Do not use 'set theory' just because it is there. A clever generically programmed C++ solution may not be appropriate if the remaining people in the development group are still wet behind the ears programmers.

Aspect-oriented programming

The history and comments above are all a rather long preamble into the main topic of this article. As we walk through some of the key elements of aspect-oriented programming keep in the back of your mind how each of the stages of development of other programming languages adds to the ability to design and code a solution.

The additional abstraction mechanisms introduced with aspect-oriented programming extend those provided in the aforementioned languages, particularly some concepts introduced in object-oriented languages.

Aspect-oriented programming is based on separation of concerns, i.e. breaking a program into distinct areas of functionality. In fact this is a common concept with structured programming and object-oriented programming languages, but aspect-oriented programming takes this view one stage further.

Even though the inheritance of object-oriented programming gives us code reuse we still find times when code is duplicated. For example, tracing call paths through code, where we are concerned about the aspect of 'tracing', or the bracketing of `start` and `commit` where we are concerned with making certain our database transactions are atomic.

Hello World example

I'm not intending this article to cover every detail of aspect-oriented programming, but rather highlight the coding and design abstractions that it provides. Nevertheless, it is worthwhile diving straight in to a simple AOP Hello World example.

Java and AspectJ (an AOP language whose syntax closely follows that of Java) will be used in the example languages. Aspect-oriented programming, however, is not limited to Java and AspectJ – more on this later.

the keywords of the AspectJ language help us with our abstraction

```
package com.nigeleke.accuexample.classes;
public class Name {

  public Name (String forename, String surname)
   {forename_ = forename; surname_ = surname;}

  public String getForename() {return forename_;}
  public void setForename(String forename) {
     forename_ = forename;}

  public String getSurname() {return surname_;}
  public void setSurname(String surname) {
     surname_ = surname; }

  private String forename_;
  private String surname_;
}

package com.nigeleke.accuexample.classes;
public class Address {

  public Address (String street, String town)
    {street_ = street; town_ = town;}

  public String getStreet() {return street_;}
  public void setStreet(String street) {
     street_ = street;}

  public String getTown() {return town_;}
  public void setTown(String town) {
     town_ = town;}

  private String street_;
  private String town_;
}
```

Listing 1

In this example we want to trace the execution of the methods in two classes, **`Name`** and **`Address`**.

The core classes look like Listing 1 – a couple of attributes and their corresponding getters and setters.

If we add tracing by more traditional methods we end up with Listing 2.

Similar coding is also inserted into the **`Name`** class. As you can see, not only does it detract from the real work of the class, it is also pretty repetitive. Not only that, but the repetition is across classes as well as the methods within a class. This is therefore error-prone – you only have to look at the (deliberately – yes, honest!) bugs introduced in **`getTown()`** and **`setTown()`**.

The common theme that we're considering here is one of tracing. Or, to put this in terms used in AOP, the **`concern`** we have is the **`aspect`** of tracing. (Note how the keywords of the AspectJ language help us with our abstraction).

```
package com.nigeleke.accuexample.classes;
public class Address {

  public Address (String street, String town) {
    System.out.println("Entered Address:ctor");
    street_ = street; town_ = town;
    System.out.println("Exiting Address:ctor");
  }

  public String getStreet() {
    System.out.println(
       "Entered Address::getSteet");
    System.out.println(
       "Exiting Address::getSteet");
    return street_;
  }

  public void setStreet(String street) {
    System.out.println(
       "Entered Address::setSteet");
    street_ = street;
    System.out.println(
       "Exiting Address::setSteet");
  }

  public String getTown() {
    System.out.println(
       "Entered Address::getTwin");
    System.out.println(
       "Exiting Address::getTown");
    return town_;
  }

  public void setTown(String town) {
    System.out.println(
       "Entered Address::setStreet");
    town_ = town;
    System.out.println(
       "Exiting Address::setStreet");
  }

  private String street_;
  private String town_;
}
```

Listing 2

So let's, first off, forget all the **`println`** statements from **`Address`** and go back to the original class, then define an **`aspect`** for performing tracing.

we are able to separate all common code related to tracing from the main classes

The **aspect** looks very much like a class definition:

```
package com.nigeleke.accuexample.aspects;
public aspect Tracing {

  // (1)
  public pointcut constructor() :
    execution(
      com.nigeleke.accuexample.classes.*.new(..));

  // (2)
  public pointcut anyMethod() :
    execution(
      * com.nigeleke.accuexample.classes.*.*(..));

  // (3)
  before() : constructor() || anyMethod() {
    System.out.println("Entered ctor or method");
  }


  // (4)
  after() : constructor() || anyMethod() {
    System.out.println("Exiting ctor or method");
  }
}
```

Declarations (1) and (2) declare **pointcut**s. The **pointcut**s are used in the selection of **join-point**s, i.e. the locations in the OO code that are of interest to us. For tracing, this will be each time we enter or exit a constructor or a method.

Declaration (1) states that we are interested in all constructors of all classes within the **com.nigeleke.accuexample.classes** package; declaration (2) states that we are interested in all methods in all classes, again within the **com.nigeleke.accuexample.classes** package. Specifically they state we are interested in the *execution* of the package's constructors or methods.

Statements (3) and (4) are known as **advice**s. They inject their **advice**, i.e. the body of code associated with them, at the **join-point**s defined by the **pointcut** filter expressions.

Statement (3) is saying that, *before* we *execute a constructor* (**pointcut** 1) or execute a method (**pointcut** 2) we will print out a trace to say we have entered. Similarly statement (4) is saying that, *after* we've executed the constructor or method will will print out a tracing to say we're exiting.

So when we run this code:

```
Address address = new Address("aStreet", "aTown");
address.setStreet("bStreet");
address.setTown("bTown");

Name name = new Name("aForename", "aSurname");
name.setForename("bForename");
name.setSurname("bSurname");
```

we get:

```
  Entered ctor or method
  Exiting ctor or method
  Entered ctor or method
  Exiting ctor or method
  Entered ctor or method
  Exiting ctor or method
  Entered ctor or method
  Exiting ctor or method
  Entered ctor or method
  Exiting ctor or method
  Entered ctor or method
  Exiting ctor or method
```

As this stands, the text to say we're entering or exiting isn't particularly useful. It doesn't convey the names of the classes or methods involved. Nor does it provide us any information about parameters passed, or values returned.

Although simplistic, what this does demonstrate is that, by defining the **Tracing** aspect, we are able to separate all common code related to tracing, from the main classes of **Name** and **Address**.

Let's just pause and repeat that – "we are able to separate all common code related to tracing from the main classes". Simply link this aspect with the original, simple, class code, and we get automatic tracing.

By making a small change to *only* the **Tracing** aspect more useful information about the objects and methods being traced is provided. So by changing the body of the **before()** and **after()** advices to use a further feature of AspectJ (**thisJoinPoint**) as shown in Listing 3, we get the output shown in Listing 4.

```
package com.nigeleke.accuexample.aspects;
public aspect Tracing {

  public pointcut constructor() :
     execution(
     com.nigeleke.accuexample.classes.*.new(..));

  public pointcut anyMethod() :
     execution(
     * com.nigeleke.accuexample.classes.*.*(..));

  before() : constructor() || anyMethod() {
    System.out.println(
       "Entered " + thisJoinPoint);
  }

  after() : constructor() || anyMethod() {
    System.out.println(
       "Exiting " + thisJoinPoint);
  }
}
```

Listing 3

The creator of the class need not even be aware that tracing is required, or have any knowledge about how tracing is performed.

```
Entered execution(com.nigeleke.accuexample.classes.Address(String, String))
Exiting execution(com.nigeleke.accuexample.classes.Address(String, String))
Entered execution(void com.nigeleke.accuexample.classes.Address.setStreet(String))
Exiting execution(void com.nigeleke.accuexample.classes.Address.setStreet(String))
Entered execution(void com.nigeleke.accuexample.classes.Address.setTown(String))
Exiting execution(void com.nigeleke.accuexample.classes.Address.setTown(String))
Entered execution(com.nigeleke.accuexample.classes.Name(String, String))
Exiting execution(com.nigeleke.accuexample.classes.Name(String, String))
Entered execution(void com.nigeleke.accuexample.classes.Name.setForename(String))
Exiting execution(void com.nigeleke.accuexample.classes.Name.setForename(String))
Entered execution(void com.nigeleke.accuexample.classes.Name.setSurname(String))
Exiting execution(void com.nigeleke.accuexample.classes.Name.setSurname(String))
```

Listing 4

Already more useful output, and we have not had to touch *any* of the methods in the **Name** or **Address** classes.

Yet another small change to the aspect and we are able to list the argument values and the return values. In Listing 5, the **around()** advice is used so that we can access the return value, which gives the output in Listing 6.

```
package com.nigeleke.accuexample.aspects;
public aspect Tracing {

  public pointcut constructor() :
    execution(
     com.nigeleke.accuexample.classes.*.new(..));

  public pointcut anyMethod() :
    execution(
     * com.nigeleke.accuexample.classes.*.*(..));

 Object around() : constructor() || anyMethod() {
    System.out.println(
     "Entered " + thisJoinPoint.toShortString());

    Object[] args = thisJoinPoint.getArgs();
    for (Object arg : args) {
      System.out.println(
       "with argument: " + arg);
    }

    Object o = proceed();

    System.out.println(
     "Exiting " + thisJoinPoint.toShortString());
    System.out.println("returning: " + o);

    return o;
  }
}
```

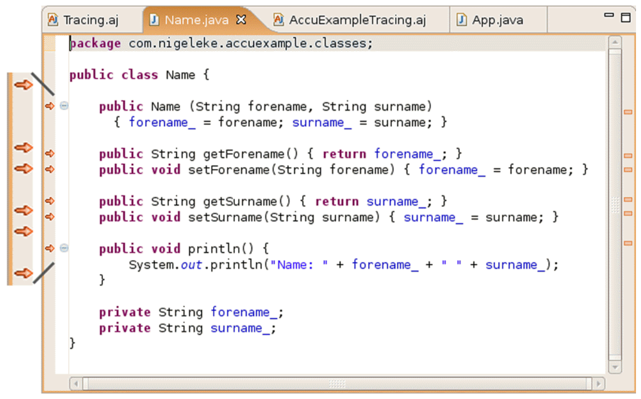
Listing 5

Again we have not had to touch any of the methods in the **Name** or **Address** classes.

Further, when we add a new class, **Account** for example, the **Tracing** aspect displays even more power. The new class will automatically get the tracing functionality required for the package. The creator of the class need not even be aware that tracing is required, or have any knowledge about how tracing is performed. They can simply concentrate on the responsibilities of the **Account** class.

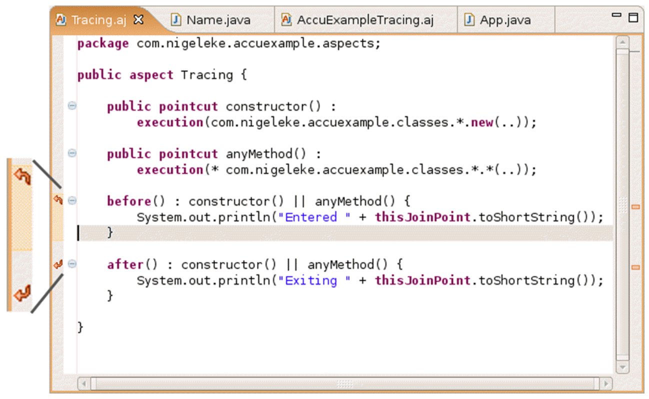
```
Entered execution(Address(..))
with argument: aStreet
with argument: aTown
Exiting execution(Address(..))
returning: null
Entered execution(Address.setStreet(..))
with argument: bStreet
Exiting execution(Address.setStreet(..))
returning: null
Entered execution(Address.setTown(..))
with argument: bTown
Exiting execution(Address.setTown(..))
returning: null
Entered execution(Name(..))
with argument: aForename
with argument: aSurname
Exiting execution(Name(..))
returning: null
Entered execution(Name.setForename(..))
with argument: bForename
Exiting execution(Name.setForename(..))
returning: null
Entered execution(Name.setSurname(..))
with argument: bSurname
Exiting execution(Name.setSurname(..))
returning: null
```

Listing 6



```
package com.nigeleke.accuexample.classes;

public class Name {

    public Name (String forename, String surname)
      { forename_ = forename; surname_ = surname; }

    public String getForename() { return forename_; }
    public void setForename(String forename) { forename_ = forename; }

    public String getSurname() { return surname_; }
    public void setSurname(String surname) { surname_ = surname; }

    public void println() {
        System.out.println("Name: " + forename_ + " " + surname_);
    }

    private String forename_;
    private String surname_;
}
```

Figure 1



```
package com.nigeleke.accuexample.aspects;

public aspect Tracing {

    public pointcut constructor() :
        execution(com.nigeleke.accuexample.classes.*.new(..));

    public pointcut anyMethod() :
        execution(* com.nigeleke.accuexample.classes.*.*(..));

    before() : constructor() || anyMethod() {
        System.out.println("Entered " + thisJoinPoint.toShortString());
    }

    after() : constructor() || anyMethod() {
        System.out.println("Exiting " + thisJoinPoint.toShortString());
    }

}
```

Figure 2

This simple example, therefore, shows the main concept introduced by aspect-oriented programming – cross-cutting concerns, i.e. common functionality across classes. The **Tracing** aspect manages common tracing functionality; the **Name**, **Address** and **Account** classes manage their sole responsibilities, and they do not need to be concerned with how to perform tracing in their methods. It also adds a consistency to what is traced, and how it is traced, which may otherwise be lost.

Visualising aspects

There is one immediate consequence of the **Tracing** aspect. It is not possible to know, simply by reading the **Name** and **Address** source code, that the code in **Tracing** has an impact on what gets executed.

This is not necessarily a bad thing. The writer of **Name**, **Address** and any other classes which get added to this package, *should* only be concerned with the simple responsibilities of these classes.

In some respects this is not *much* different to someone inspecting the code of some parent class in a hierarchy, without knowing, or needing to know, how the implementations of methods in the child classes override their parent.

In practise however, it is pragmatic to know what interactions occur between aspects and classes.

The AspectJ development environment [AJDT] provides us with two mechanisms to view the impact of aspects on other classes.

The first of these mechanisms is shown in figures 1 and 2, where there are small indicators in the left margin of the source code editor. The classes show inbound arrows, indicating where their methods are being advised by other aspects. The aspects show outbound arrows, indicating their advises will have a side effect on a class's methods.

The second mechanism provides a view of the bigger picture. Figure 3, taken from the space-war example provided with AJDT, shows a vertical bar for each class. The horizontal bars are colour coded with a different colour for each aspect. These show the approximate position within the class of the join-points affected by an advice. Double-clicking on them takes you to the appropriate position within the class source.

Pointcuts

With the **aspect**s previously defined we are able to see some of the power of aspect-oriented programming. The **pointcut** helps determine the **join-point** within the object-oriented code, which is affected by an **aspect**'s **advice**.

pointcuts are not just used to determine simple execution paths, however. **pointcut expression**s can also be used to determine:

1. calls made to any package, e.g. calls made to the standard java JDBC access methods.
2. read or write access to a class's attributes.
3. whether the execution path is within a given package.

In the first of these examples there is a distinction between the call of and the execution^2 of a method. The **join-point** for a call exists just before the call to the method is made, i.e. in the body of the client. The **join-point** for execution exists just after the call has been made, but before the main body of the method is executed, i.e. in the body of the method.

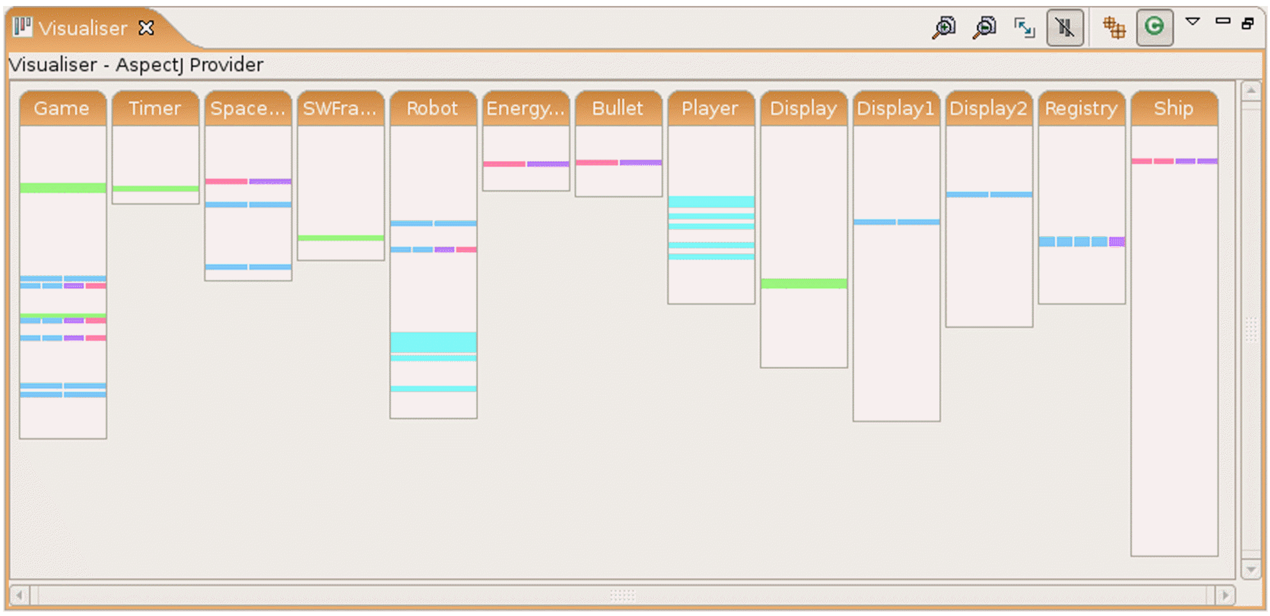


Figure 3

Initially it appears preferable to use ‘execution’ rather than ‘call’ as the advice code will only get injected once. However there are times when only a library is available, and it is not possible for the advice code to be inserted into the library code. These are the times when ‘call’ is used.

In the final example, it is possible to have an aspect which restricts how output is performed and not allow calls via **`System.out`**. However the aspect itself may want to use **`System.out`**, so it becomes necessary to be able to state ‘all calls to **`System.out`** that are not within this aspect’s package’.

`pointcut` expressions can be very general, or very specific, depending on requirements. It is possible to define a **`pointcut`** expression, for example, for ‘all public methods which start with **`set`**, take a String argument as the first parameter’, for example this may be used to make sure no client sets **`null`** values.

Advices

What about advices? These are not restricted simply to injecting code before or after a method is executed. Firstly there is a more general use the **`before()`** and **`after()`** advices of our initial example. **`around()`** embraces both **`before()`** and **`after()`** advices. We used this earlier in the **`Tracing`** example, but is shown again in Listing 7 as a cut-down example.

```
package com.nigeleke.accuexample.aspects;
public aspect Tracing {

  public pointcut constructor() :
    execution(
     com.nigeleke.accuexample.classes.*.new(..));

  public pointcut anyMethod() :
    execution(
     * com.nigeleke.accuexample.classes.*.*(..));

 Object around() : constructor() || anyMethod() {
    System.out.println("Entered ctor or method");
    Object o = proceed();
    System.out.println("Exiting ctor or method");
    return o;
  }
}
```

Listing 7

Advices can also be used to declare compile time warnings and exceptions. This is something we alluded to earlier when we were discussing restricting calls via **`System.out`**.

The aspect in Listing 8 applies a policy on the usage of **`System.out`**, and creates a compile-time warning if **`System.out`** is used externally.

The **`warning`** can be changed to **`error`** when the policy needs to be enforced more strongly. When an **`error`** is declared then the developer’s compilation will fail.

More advanced features of AOP

Given that this article started by looking at new programming concepts introduced in each generation of programming languages, I would like to address some of the more advanced features of AOP. Before I do though, I would also thoroughly recommend *Eclipse AspectJ* [Colyer et al] for a more detailed description. Although it is centred around the Java-based AspectJ language it still provides a very solid and easy to read explanation of AOP generally.

- **Abstract aspects**. In the same way classes can be abstract and extended by concrete classes, so can aspects. We could, therefore, have an abstract **`Tracing`** aspect which defines pointcuts but the implementation of the advices are left to the concrete aspects. This way a framework, for example, can determine *where* tracing should be performed within the framework structure, but the implementation is required to implement the actual tracing, i.e. users define *how*.
- **Abstract pointcuts**. Similarly pointcuts can be declared as being abstract, and the actual definition left for later. (Abstract pointcuts can only be declared within abstract aspects). If we look again at our framework tracing example – we may define the **`Tracing`** aspect so that its advices are implemented in the abstract aspect and provide the actual tracing wanted by the framework. The pointcut definitions can now be left to the concrete aspect, i.e. users of the framework determine *what* needs to be traced.
- **Method and field injection**. An aspect can inject new methods and fields into existing classes. At first sight this may seem an odd and conflicting abstraction. It certainly does not sound like concerns are being kept separate.

  Imagine, however, that we have the **`Address`** class, above and, as part of a user interface, we have an **`AddressUI`** class whose function is to display address objects. When the address object changes its observing addressUI object needs to be notified so that display fields are updated with the new values. This is a classic requirement for implementation of the Observer pattern [Gamma et al].

  We can define (standard Java) **`Subject`** and **`Observer`** interfaces. We can define an abstract **`ObserverProtocol`** aspect that it will operate on objects providing **`Subject`** and **`Observer`** interfaces.

  The abstract **`ObserverProtocol`** aspect:

  - injects a field into **`Observer`**s to hold the **`Subject`** being observed;
  - injects a method into **`Observer`**s to set the subject;
  - injects a field into **`Subject`**s to hold the set of **`Observer`**s;
  - injects methods into **`Subject`**s to add and remove an observer.

  Finally we define a concrete **`AddressUiObserver`** whose responsibilities are:

  - to define **`Address`** as implementing **`Subject`** and **`AddressUI`** and implementing **`Observer`**.
  - define the concrete **`pointcut`** to state what events constitute an update on the **`Subject`**.
  - define the **`Subject`**’s **`update()`** method as only this aspect knows what the **`Observer`** method needs to be called for notification of changes to the **`Subject`**.

  That’s it. The end result is a reusable observer protocol implementation and classes which know nothing of **`Subject`**s and **`Observer`**s which are, nevertheless, able to provide **`Subject`** and **`Observer`** interfaces.

Finding and using aspects

When we write in object-oriented programming, the class names are generally nouns, e.g. **`Account`**, or **`Customer`**.

```
package com.nigeleke.accuexample.aspects;
public aspect OutputPolicy {

  pointcut accessSystemOut() :
    get(* System.out);
    pointcut inThisPackage() :
    within(com.nigeleke.accuexample.aspects.*);

  declare warning :
    accessSystemOut() && !inThisPackage() :
    "Warning - System.out restrictions apply.";
}
```

Listing 8

2 Used earlier
3 Sad to say, still is...

Good aspects come from other grammatical areas – adverbs and adjectives for example, e.g. size, speed. Real-life examples include security and auditing.

Temporal requirements also give an indication that an aspect may be appropriate, e.g. *before* performing this action the user must have been authenticated, or *after* writing to the database an audit record must be written.

Finally rule, or policy based requirements are also fine candidates for an aspect, e.g. loans over $10,000 must be approved by a lender.

There was[3] a tendency among those new to object-oriented programming to create inappropriate classes, with unclear responsibilities or incorrect inheritance hierarchies. Further, I still observe that monolithic object-oriented spaghetti modules abound which are still a maintainer's nightmare. Perhaps we should call these spaghetti-meatballs?

Aspect-oriented programming is not the silver-bullet to fix this as it will still depend on the skill of the designer / programmer. I am certain that the use of AOP will still fall foul to environments that do not follow formal design practices, or well structured design reviews, or have personnel with a solid OO skill-set.

Also, just as our OO design practices have changed over the years and we have learned what constitutes good and bad OO design. I am sure those of us new to AOP will follow a few common bad practices and select aspects when they are not appropriate, or not use them when we could have done.

To help the learning curve for individuals, and for companies, the following adoption phases are recommended [AOSD]:

- Explore AOP individually, or within small teams. Practical use would include enforcement policies, such as “do not use `System.out`” (as mentioned earlier). It is also possible enforcing package dependencies, to make certain these are not violated. These are compile time dependencies, so have no runtime impact. At this initial stage it is recommended to avoid runtime dependencies in a production environment.
- Create project specific aspects and aspect libraries for the project infrastructure. This would include aspects like `Tracing`, which is a low-risk part of the project development.
- Create core business aspects, i.e. those defining business rules for an individual application.
- Define use within an AOP architecture, and introduce to all future development. This would include Security, Authorisation and Authentication aspects.

Not just Java

The examples given here are for AspectJ, which interleaves itself with Java classes, but there is also AspectC++ [AspectC++], which is very similar to AspectJ, but the advice bodies are C++, and the pointcut syntax aligns itself more closely with the C++ language.

The Spring framework [Spring (1)] also provides mechanisms to use aspect-oriented techniques. I believe this integrates closely with AspectJ, but also allows linking of aspects to the join-points via XML declarations or via the use of annotations. There are advantages and disadvantages in all styles and [Spring (2)] will lead you through these if you are interested.

I’m not really familiar with the Spring framework, but from a casual glance it appears that AspectJ provides a semantic richness, and a separation, not present in the other manners of defining aspects. [I would be more than happy to see an article on ‘AOP with Spring’]. It is the expressiveness permitted in AspectJ that enables the abstractions, mentioned at the start of this article, to be clearly articulated in code, so this would be my preference over using XML to express the same.

Miscellanea

One point that came to light while researching this article – Xerox have a US patent 6,467,086 for AOP/AspectJ. The AspectJ source code is released under the Common Public License, which grants some patent rights. And this all leads to a whole other discussion point regarding software patents.

Also, on the future of aspect-oriented programming, Bjarne Stroustrup was asked his opinion of aspect-oriented programming [InfoTech], where he stated: I don’t see aspect-oriented programming escaping the “academic ghetto” any day soon, and if it does, it will be less pervasive than OO. When it works, aspect-oriented programming is elegant, but it’s not clear how many applications significantly benefit from its use. Also, with AO, it appears difficult to integrate the necessary tools into an industrial-scale programming environment.

Personally I would really like to think he is wrong about AOP escaping the “academic ghetto”, but I would certainly agree it requires the correct tools to be available. Although most of my experience has been with AspectJ, I will be following the development of AspectC++ with interest.

Summary

As you can see, programming languages have developed over the course of time so that we can communicate our ideas, our algorithms and our requirements more succinctly and more clearly to both the computer and the program maintainers.

Aspect-oriented programming languages provide a mechanism for the separation of the concerns, which were previously developed by common and repeated code snippets in object-oriented developments. I hope this article has given a small taste of aspect-oriented programming and what it can deliver for analysts, designers and developers.

One final thought from Feynman’s biography: We must remove the rigidity of thought. ... we must leave freedom for the mind to wander about in try to solve the problems.

Hopefully aspect-oriented programming languages will enable us more freedom to solve our problems. ■

Acknowledgements

I would like to express my thanks to Paul Johnson, Alan Griffiths and everyone else on the Overload editorial team who critiqued and help improve my original submission.

References

[Gleick] James Gleick, *Genius – Richard Feynman and Modern Physics*, Abacus; ISBN 0-349-10532-4.

[Wiki(1)] http://en.wikipedia.org/wiki/Real_Programmers_Don't_Use_Pascal

[Wiki (2)] http://en.wikipedia.org/wiki/APL_language#Overview

[Wiki (3)] http://en.wikipedia.org/wiki/Alphabetical_list_of_programming_languages

[AJDT] AspectJ Development Tools (AJDT) – http://www.eclipse.org/ajdt/

[Colyer et al] Eclipse AspectJ; Adrian Colyer, Andy Clement, George Harley, Matthew Webster; Addison-Wesley; ISBN 0-321-24587-3.

[Gamma et al] Design Patterns; Gamma, Helm, Johnson and Vlissides; Addison-Wesley; ISBN 0-201-63361-2.

[AOSD] http://aosd.net/

[AspectC++] AspectC++ – http://acdt.aspectc.org/

[Spring (1)] Spring AOP – http://www.springframework.org/

[Spring (2)] Spring AOP declaration style – http://www.springframework.org/docs/reference/aop.html#aop-choosing

[InfoTech] http://www.techreview.com/InfoTech/17868/

Useful links

http://en.wikipedia.org/wiki/Programming_language
http://en.wikipedia.org/wiki/Structured_programming
http://en.wikipedia.org/wiki/Object-oriented_programming
Aspect-oriented Software Development – http://www.aosd.net/
US Patent 6,467,086 – Aspect Oriented Programming

Exceptional Design

Hubert Matthews discusses some ways to design programs to use exceptions.

Introduction

This article describes the approach I have taken to designing applications that I have written over the last few years. It is also a distillation of the techniques I have been teaching on C++ courses for some years. It is not meant to be the one and only true way of designing using exceptions, but rather a report on techniques I have found useful. The second part of the article explores why this approach isn't universal and some of the barriers – psychological and technical – to its adoption.

In a nutshell, the core design concepts I use are:

- Many throws, few catches
- Placement of catches is determined by recovery points, which are based around business requirements not technical considerations
- Catches should also be placed at module boundaries
- Implement the strong exception safety guarantee whenever reasonable
- Make intermediate code exception-neutral
- Log at the point of detection, recovery (if any) at the point of handling
- Use local control structures in preference to throwing an exception
- Use standard exceptions or subclasses thereof
- Use as few subclasses as possible, ideally zero or one
- Add additional information to the exception class for multi-level recovery and for business and technical context
- Don't nest exceptions

These techniques are primarily focused on C++, the language I code in the most. C++ has a cleaner exception model than Java and C# as well as deterministic destruction (i.e. destructors instead of **`try`**/**`finally`** blocks), both of which make the use of exceptions easier and more elegant.

First step

The first thing I do when writing a C++ program is to place a double **`try`**/**`catch`** block in **`main()`**:

```
int main()
{
  try {
    // main body of code
  } catch (const std::exception & e) {
    cerr << "Caught exception: " << e.what()
     << endl;
    return 1;
  } catch (…) {
    cerr << "Caught unknown exception" << endl;
    return 1;
  }
}
```

This ensures that no exception can propagate out of **`main()`** and thereby cause the program to terminate unceremoniously (fragile programs that disappear without a trace are not popular!). It is an example of the maxim "catches at module boundaries". In this case the boundary is between my program and the host operating system (indirectly via the run-time library), which expects a return code. The **`catch(…)`** acts as a catch-all (literally) for other exceptions. These may be caused by an errant **`throw`** (for example someone throwing a plain integer or a string) or on earlier versions of Visual C++ errors such as access violations and divide-by-zero errors were translated into anonymous C++ exceptions using Windows' structured exception handling (SEH). The GNU g++ compiler can be persuaded to turning signals into exceptions, but again this is architecture and processor specific.

This universal "last gasp" handler therefore protects against these anonymous exceptions and adds perceived robustness to the program. One of the programs I wrote and maintained for 8 years on Windows used a third-party library that would occasionally cause access violations, so this **`catch(…)`** technique was very useful. **`try`**/**`catch`** blocks also need to be placed at the top of every thread function and probably around any code that executes in button handlers in GUI applications (as exceptions cannot propagate across GUI message pumps). In production code more error logging than this would be appropriate (and perhaps even a core dump on Linux or Unix).

Recovery strategy

The next step is to determine what the recovery strategy for the program should be. This can range from a graceful shut down for simple end-user programs through partial completion all the way up to resilient recovery, retry or fail over for long-running server programs. Recovery is a business-requirements issue more than it is technical design. The key question to ask is "what should happen now?" rather than "what can we do?". Sometimes recovery is multi-layered with some classes of error being handled at an intermediate level and others being escalated to a higher layer. As an example, one of the applications I wrote had two intermediate layers: the first layer was a list of command scripts to run and the second layer was the scripts themselves. There was a recovery point (i.e. a **`try`**/**`catch`** pair) around each script line invocation that enabled the script to continue despite an errant line, with a second recovery point around each script file to allow for missing or inaccessible script files. The "last gasp" handler at the outer layer was used to catch errors such as memory allocation errors.

In order to create such a layering, it was necessary to use a custom exception class instead of one of the standard subclasses of **`std::exception`**. This custom class inherited from **`std::exception`** and added an additional severity flag:

Hubert Matthews can be contacted at hubert@oxyware.com

Cross-component error reporting is not something that you want to tie into a component's control flow

```
class AppException : public std::exception {
public:
  enum Level { warning, severe, fatal };
  AppException(
     Level level, const std::string & msg)
     : std::exception(msg), level(level) {}
  const Level level;
};
```

This extra level information allows intermediate catch handlers to choose whether to recover or re-throw:

```
try {
  // inner-level code
} catch (const AppException & e) {
         if (e.level == AppException::warning)
             Log(e);  // "recovery"
         else
             throw;  // appeal to a higher authority
}
```

The severity level for each exception is decided at the call site when the exception is thrown:

```
if (FileIsNotAccessible(filename))
    throw AppException(AppException::severe,
    "Can't open file " + filename);
```

With such a structure in place, handling errors becomes conceptually much easier as if a subroutine detects an error it cannot handle locally then it can just `throw`. Handling errors locally is of course preferable and normal control structures should be given preference over the use of exceptions.

An alternative approach is to use a very limited exception hierarchy with one subclass per error level. This allows the multi-level catch handler shown above to use the type of the exception to catch only certain error levels:

```
class WarningException : public std::exception {};
class SevereException : public std::exception {};

try {
  // inner-level code
} catch (const WarningException & we) {
  // handle only warnings, let severe exceptions
  // propagate to the next level up
  Log(we);
}
```

The effective difference between these two approaches is minimal; both achieve the same effect. The catch handler is effectively identical and the throw site is also the same in all but syntax.

I use a single exception class rather than an application-specific exception hierarchy as I have found little need to differentiate between different types (rather than different severities) of exception when handling them. Doing so would require a whole chain of catch handlers that rapidly degenerates into the equivalent of an `if-else-if` chain testing against types:

```
try {
  // code
} catch (const SubException1 & e) {
  // handle exception case 1
} catch (const SubException2 & e) {
  // handle exception case 2
} catch (const SubException3 & e) {
  // handle exception case 3
}
```

There are a number of problems with such code. First, the order of the catches is now important as more derived classes must be caught before their bases. Second, adding a new exception type requires the catch handlers to be updated, which is a sign of undue coupling. Third, I can think of little significant difference from a recovery perspective that justifies the above problems. My other objection to the use of an exception hierarchy is that I feel that inheritance is primarily a way of coding variations in behaviour. In this case, however, the variation isn't in the exceptions; it's in the handler. Therefore normal control structures in the handler should suffice and having a single exception class reduces the coupling in the application.

Some sources – particularly in the Java camp – suggest nesting exceptions. Again, I have found little use for this or for other advice such as the use of checked exceptions in Java. The simplicity of the proposed approach is appealing and has served me well on small programs. Extending this approach unmodified to a larger scale is possible but unlikely. More likely is to place `try`/`catch` blocks at component boundaries and use return codes across the interface with exceptions internally. This allows each component to look after its own clean-up and resource management without imposing anything on the caller. Cross-component error reporting is not something that you want to tie into a component's control flow; the raising of events or logging of errors is easier, cleaner and more usual. Throwing exceptions across network or process boundaries isn't a clean idiom either, so good old error codes are best here too. Handling errors on a system-wide scale is a different matter altogether. Languages like Erlang lead the way here.

This style of design relies on the use of exception safe techniques throughout the code. The strong guarantee – which offers commit/rollback semantics – should be used wherever it is practicable. Intermediate code should be exception neutral (i.e. have no `try`/`catch` blocks for clean-up) as this makes code clearer and easier to test as there is then no difference between the normal path and the path taken when an exception is thrown from lower-level code. These techniques have been well documented by members of the C++ community: `auto_ptr`, RAII, copy-swap idiom, "do work on the side", etc. [Sutter99], [Stroustrup00]. One point to note is that the transactional nature of the strong guarantee is often provided for persistent data by using a relational database. For file storage, no such transactionality is available and developers must code it themselves. The same goes for in-memory state changes (although software transactional memory is a current research topic [STM05]). Rollback in such cases is provided not at the recovery point (the **`catch`** handler) but by the stack unwinding caused by the propagation of the exception. Designing commit mechanisms that do not throw can itself be a challenge!

Things that don't work and anti-patterns

I have seen a number of exception anti-patterns in C++ code in addition to the multiple subclasses anti-pattern:

- **`try`**/**`catch`** surrounding each call
- Exception squashing (no recovery)
- Overuse of exceptions
- **`throw`**s in **`catch`** blocks

One anti-pattern I have often seen is surrounding each call with a **`try`**/**`catch`** block in order to add error context for logging:

```
void FunctionX()
{
  try {
      FunctionY();
  } catch (const MyException & e) {
      Log(e);
      throw MyException(
         "Error: FunctionX -> FunctionY", e);
  }
  // similarly for call to FunctionZ()
}
```

This is worse than error codes for a number of reasons. The clarity of exception neutrality is lost, so the main application code is intertwined with error handling in exactly the way return codes are. When justifying this approach, programmers have told me that they believe that the extra context information is useful for debugging. I have never found it so as the most relevant information is available at the point where the error is detected, that is, where the exception is thrown. Adding in all of these intermediate **`try`**/**`catch`**es is also a heavy burden that bloats the code and adds no user-visible functionality or tackles the real purpose of exception: recovery. I presume that developers who do this do not understand the purpose of stack unwinding and are still thinking in a C-style idiom.

If this approach is avoided, then the next item up the anti-pattern food chain is the null recovery block. This is where the catch handler either ignores the exception totally:

```
try {
  //
} catch (const MyException &) {}
```

because the programmer doesn't know what to do (and the specification is silent) or attempts to stumble on regardless. This anti-pattern is more common in Java because of the presence of checked exceptions as it can eliminate the need to alter the exception specification of the function. (**`InterruptedException`** springs to mind here.) There is little need to say why this is not a recommended practice! Just logging the error and continuing is almost as bad. (The "last gasp" handler falls into this category but since it by definition is called only when recovery is not possible then we turn a blind eye in this case.)

Some people overuse exceptions and use them when local control structures would be more appropriate. An example might be using an end-of-file exception instead of an error code, an event that is entirely predictable and something that does not need long-range recovery.

The final anti-pattern – throwing from within a catch block – is rarer and is more of a sin of omission than one of poor design or limited understanding. Sometimes this is exactly what is required when recovery at an intermediate level has failed or isn't possible, but often it is caused by calling a function that itself throws. Caveat coder. Examples abound in Java of calling **`jdbcStatement.close()`** and friends in **`finally`** blocks with a null object reference. The use of exception neutral intermediates reduces the number of places that this anti-pattern can occur to a few recovery points where additional thought, care and review time can be expended.

Consequences

Viewing exceptions as recovery requests has a number of important consequences. The primary effect is that it alters the perspective of the developer from a historic "stuff happens" view to a forward-looking "so what am I going to do about it" view, a change of focus that opens up a lot of possibilities as well as posing a large number of requirements-level questions about what recovery means. In "agile" terminology it is the start of a conversation with the customer that may well lead to some additional user stories. In a classic RUP-style use-case based approach it starts to fill in the "errors, exception and alternative paths" section of a full-dress Cockburn-style use case [Cockburn00]. This change in viewpoint should not be underestimated as it turns exceptional conditions from technical "oops" moments into user-visible events, events that contribute to the robustness and resilience of the application and about which users will usually have strong opinions. Altering a "can't save file foo.doc" show-stopping modal pop-up into a sequence of alternative actions – such as saving under a different name, on a separate device, etc. – transforms an application and users' trust in it. The change of emphasis could be compared to the change that occurs when developers use test-driven development: you see things from another angle, one that illuminates the specification more than the implementation.

Once recovery from an exception becomes a user-visible event then it becomes a target for testing. Traditionally, error handlers have been poorly tested as it is often extremely difficult to provoke suitable errors. Exception neutrality now shines brightly for two reasons: reduction of error paths and ease of testing. Since exception-neutral code has no explicit error paths then any path through the code – normal or exceptional – provides adequate coverage. Thus, error handling does not add to the McCabe complexity of the intermediates and only the recovery points need testing for exceptional cases. There are only a handful of such **`catch`** blocks in the system so only these few places that contain complex recovery code have to be tested independently. Doing this by using mock objects that always throw often means inserting abstract base classes or interfaces as substitution points for mock objects. The overall testing burden for error handling is thus lower, it is less invasive and the orphan child Recovery need no longer feel like Cinderella on a bad night. (One point to remember is that exception neutrality and exception safety are orthogonal issues. The lack of **`catch`** blocks or calls to **`uncaught_exception()`** does not imply that throwing an exception will not lead to subtle errors or resource leaks!)

Why don't programmers do this currently?

Given the glowing overview and snake-oil claims for this approach, why do programmers not use it? Habit and old idioms, low-level detail and suspicion of exceptions are some of the reasons. Lots of developers are quite conservative creatures of habit and stick to what they know. Others are attracted to the Lorelei call of the new. Only a few, or so it seems to me, take the time to analyse how they could improve the use of their current languages and tools.

Habit and old idioms Developers have been using return codes for years and they are comfortable with them. Return codes are simple and fit well with local control structures. I am not advocating abandoning return codes as they should be the first port of call when handling errors. Exceptions are useful as an escalation request only when all local attempts have failed, in a similar manner to avoiding bothering your boss until you've tried everything at your disposal and need outside help.

Low-level detail By this I mean being submerged in lots of low-level detail and not being able to see the larger design-level decisions. This is why I advocate determining the recovery points early on in a design as it brings complexity-relieving structure to an otherwise bald and unconvincing area of programming. Knowing that help is only a throw away does much to alleviate the angst of error handling.

Suspicion of exceptions This is a definite issue with some developers, primarily those who are very concerned about performance and memory size. In order to reason about this we need to examine a little how exception handling is typically implemented and its associated costs.

There are two common implementation techniques: stack-based and table-based. The table-based approach incurs zero execution-time cost if an exception is not thrown but relies on a potentially large static table of locations. This table is used to determine which destructors need to be called based on the program counter when the exception is thrown. In memory-constrained environments this table may be unacceptable.

In comparison a typical stack-based approach has a number of parts to it. Catch handlers are required in the function where the catches are declared. Unwind handlers are required in every function that allocates an object on the stack that has a non-trivial destructor. This may seem heavy, but compare this to doing the whole thing manually. You have to write the error propagation mechanism yourself using return codes, `if` statements, early returns, etc. The overall amount of code in the executable is probably similar and the efficiency isn't that different either – some stacking of objects versus lots of return code creation and checking. So the run-time efficiency (both speed and size) isn't that different, I suspect.

Breaking the error process into stages, there are three separate parts: detection, propagation and recovery. The speed and size of detection is the same for both exceptions and return codes, as is recovery. It is only propagation that differs and this is typically the smallest cost of the three parts. Exceptions provide a built-in mechanism for this whereas with return codes you have to write it all yourself every time afresh. As with any other cut-and-paste type of code duplication, you now own that mechanism so you have to test it, maintain it, and so on for the lifetime of the code. Faced with this, a standard clean compiler-provided mechanism seems like a good deal to me.

There are other forms of efficiency to consider, however. With return codes there is always the chance that you will get it wrong by omitting to test a return code, plus the fact that your application code is now intimately entwined with error handling code which makes it more difficult to understand and get right. On top of this, consider how difficult it is to test your manual error handling code: you have to falsify all of the return codes for all of the possible paths. This leads me to think that in terms of programmer efficiency that exceptions win hands down.

Exceptions and error handling are afterthoughts

I often get the feeling that error handling is but an afterthought, something that gets smeared on afterwards to bring an application back to a superficially acceptable level of stability. I hope that encouraging developers to think of recovery rather than termination by providing an overall error-handling structure might encourage developers to avoid concentrating on only the "happy day" scenario. Perhaps one day their bosses might even give them time to do so too.

Exceptions guarantees and design

The modern C++ community has adopted Dave Abrahams' three exception guarantees: basic, strong and nothrow [Sutter99]. It is instructive to see how these essentially technical-level guarantees relate to higher-level design techniques such as statecharts. The basic guarantee ensures that an object is in a usable state that satisfies its invariant after an operation if an exception is thrown. In terms of a UML statechart this means that if an exception is thrown then that particular state machine instance can reappear in any of the states! The strong guarantee implies that either a transition occurs or the state machine stays in the original state (commit/rollback semantics), and the nothrow guarantee implies that the transition will occur. The lack of precision of the basic guarantee is one of the primary reasons for aiming for the strong guarantee whenever possible. For instance, in an e-commerce application that implements only the basic guarantee, an exception could empty your persistent shopping cart and log you off. Even worse it could mark your order as complete! Caveat guarantor.

If exceptions are used as user-visible recovery requests then the recovery strategy can be designed as part of the same state machine as the normal path. This may involve additional operations, compensating transactions for rollback (as in two-phase commit), timeouts, etc. This reduces the likelihood of error handling being left up to developers who will probably choose the easiest option for them in the absence of clear requirements. My personal hope is that software designed this way will be far less likely to present me with a message box containing hexadecimal information barely helpful even to a developer, leading to reasonable software that isn't brittle, software that I can trust.

An interesting side issue is the use of assertions in debug mode versus production mode. The standard `assert` macro stops the program when in debug mode but not in the release version. Tony Hoare [Hoare73] commented that removing assertions in release mode is like wearing your lifebelt when practising and removing it when venturing out for real. There would appear to be a case for having run-time assertions raise exceptions instead and then relying on the recovery mechanism to return the program to a usable state in both modes. This path seems to steer a fine line between asserts "stop the world" and "forget about assertions" modes, neither of which seems useful in production software.

Conclusion

Using exceptions as recovery requests is a technique I have been using successfully for a number of years in C++. The overall effect on the design is that the mainline code is simpler and clearer than with return codes and the exception handling provides a simple and stable structure that supports the mainline code by removing and isolating complexity. Sophisticated recovery strategies can be implemented, ones that involve a dialogue between the recovery code and the underlying functions.

This approach is very simple, seeming almost trivial compared to some of the anti-pattern approaches I have had the privilege to witness. I believe it expresses the essence of the approach in a clear manner, one that steers developers away from a historical technical viewpoint to a forward-looking user-oriented view of error handling. Perhaps this is its greatest strength, something only obvious in hindsight and something that grows on you over time. ■

Acknowledgements

The reviewers, Ric Parkin and Roger Orr, pointed out not only where I had stepped off the narrow path of fact and into the realms of rose-tinted memory but also suggested a number of useful clarifications and alternatives. My thanks to them both.

References

[Sutter99] H. Sutter, Exceptional C++, Addison-Wesley, 1999.

[Stroustrup00] B. Stroustrup, The C++ Programming Language (3rd Edition), Addison-Wesley, 2000.

[Hoare73] C.A.R. Hoare, Hints on Programming Language Design, Stanford University Artificial Intelligence memo AIM224/STAN-CS-73-403. Reprinted in [Hoare89], 193-214.

[Hoare89] C.A.R. Hoare/C.B. Jones (Eds.), Essays in Computing Science (reprints of Hoare's papers), Prentice Hall, 1989.

[STM06] http://research.microsoft.com/~simonpj/papers/stm/index.htm

[Cockburn00] A. Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000.

C++ Trivial Logger

When a fully functional logging subsystem isn't the answer what does one do?
Seweryn Habdank-Wojewódzki rolls his own.

Introduction

Sometimes there is a need to track some results of the program, but we do not want to put them in the output of the program – especially when the program is working either as a server or if we want to observe some results, but not show them in the GUI. There are times we want to debug the program but there are a lot of iterations to perform (such as work on huge data containers, or making many calculations and we would like to observe all iterations).

```
// cont is a huge container e.g. std::list<double>
// filled with values

for ( std::list<double>::const_iterator
    pos = cont.begin();
    pos != cont.end(); ++pos )
{
    // operate on *pos
    // and we want to observe
    // if there are any somehow critical values
    // inside cont
}
```

Listing 1

```
// Let assume that Editbox is a edit box widget
// where user can put value e.g. for currency like:
// 120 (by default e.g. in Euro), 120.11EUR,
// E120, 120.1E and so on.

// Let assume for simplification that we have
// a standard string as a result from Editbox.

std::string val_str = Editbox.get_text();

// we want to log what users put in the edit box
// to have an overview about typical
// errors in writing, or what is a preferable style,
// maybe string uses UTF-8, and we do NOT expect
// signs form extended set like "€".

// let assume that currency is a class
// that contains information about currency
// and it has overloaded operator>> and operator<<

currency value =
boost::lexical_cast<currency>(val_str);

// and we want to observe what are the result
// from currency parser.
```

Listing 2

Of course debuggers support conditional stops, but if we do not exactly know what the conditions are or if there is a problem to set a "stop condition" for the debugger, using a debugger can be problematic. The solution is to equip the application with the logger.

For logging, there are specially designed libraries.

They are equipped with

- sinks (appenders),
- filters which need to be configured in runtime but they have to be compiled as library.

These loggers are large (with a potential performance hit), because of existence of parsers for configuration files e.g. written in XML or in the style of UNIX configuration files.

As a simple case, let's consider simple code shown in listing 1 and 2.

Why we do not sometimes need big loggers? What do we have to do if we do not need all of that, but only need simple functionality like `Log ( variable )`? Do we need one logging stream? What if we do not separate levels of logging? And even then sometimes the big loggers' licences do not fit our project. If the code is simple then it has greater possibility to be portable. The solution for this can be the usage of a lightweight logger.

A typical and very rough solution for that problem is to write in every place where we need logger/debugger functionality preprocessor directives such as in listing 3:

```
std::string val_str = Editbox.get_text();

#if defined (DEBUG)
  std::cout << str << std::endl;
#elif defined (FILEDEBUG)
  file_stream << str <<std::endl;
#endif

currency value =
boost::lexical_cast<currency>(val_str);

#if defined (DEBUG)
  std::cout << value << std::endl;
#elif defined (FILEDEBUG)
  file_stream << value <<std::endl;
#endif
```

Listing 3

Seweryn Habdank-Wojewódzki is a PhD student, specialising in computational algorithms for data mining, prediction, classification and pattern recognition. In his free time he likes painting, writing poetry and reading philosophical books. He can be contacted at habdank@gmail.com

the **motivation** for preparing ones **own logger** is that **licences** of existent loggers may **not** be **suitable** for the project

It can be seen that there are a lot of lines which are not needed; especially all of the compiler directives. In this simple example, we have 5 lines of code for 1 line of logging functionality. Even if we prepare macro to use `Log ( )` instead of `file_stream << str <<std::endl;` there is still a problem with compiler directives.

In this paper, the author will present a very easy to use logger which solves the problem of compiler directives. The design is based on pointers to the output streams e.g. to the file or to the standard output stream (generally console output). There will be some examples of usage, and some possible extensions. Also, the author compares such a simple construction with other existing (free) logging libraries. In fact, the comparison is more a presentation, because they have much larger functionality – however they are really not lightweight.

In the end, the motivation for preparing ones own logger is that licences of existent loggers may not be suitable for the project and we do not want to extend too much our project.

Construction of the C++ Trivial Logger

We can describe some needs for such a logger.

Assumptions:

- Activate the logger at compilation time by using a flag. If flag is not set then logger has to be cleaned up from the code or set to the stream which ignores all input;
- Flag has to switch logger style;
- Usage as simple as possible;
- Debugger-like style for debugging purpose;
- Implement some basic configuration procedure.

The proposed construction of the header file `logger.hpp` is as shown in listing 4. We can observe that the usage of the `std::auto_ptr` helps with destroying/closing a file stream and also other streams based on the construction of the proper class [Josuttis99]. The definitions of the `Log(name)` construct is a very useful function like macro.

Three flags are defined; `FTLOG` (File Trivial Logger), `TLOG` (standard stream Trivial Logger) and `ETLOG` (standard Error stream Trivial Logger), each with different functionality.

```
// Copyright (c) 2005, 2006
// Seweryn Habdank-Wojewodzki
// Distributed under the Boost Software License,
// Version 1.0.
// ( copy at http://www.boost.org/LICENSE_1_0.txt )
#ifndef LOGGER_HPP_INCLUDED
#define LOGGER_HPP_INCLUDED

#include <ostream>
#include <memory>
```

Listing 4

```
class logger_t
{
public:
  static bool is_activated;
  static std::auto_ptr < std::ostream >
     outstream_helper_ptr;
  static std::ostream * outstream;
  logger_t ();private:
  logger_t ( const logger_t & );
  logger_t & operator= ( const logger_t & );
};
extern logger_t & logger();
#define LOG(name)do {if (logger().is_activated ){\
    *logger().outstream << __FILE__ \
    << " [" << __LINE__ << "] : " << #name \
    << " = " << (name)
    << std::endl;} }while(false)
namespace logger_n {
  template < typename T1, typename T2, \
     typename T3, typename T4 >
  void put_debug_info ( logger_t & log, \
      T1 const & t1, T2 const & t2, \
      T3 const & t3, T4 const & t4 )
  {
    if ( log.is_activated )
      {
      *(log.outstream) << t1 << " (" \
        << t2 << ") : ";
      *(log.outstream) << t3 << " = " \
        << t4 << std::endl;
      }
  }
}
#define LOG_FN(name) logger_n::put_debug_info ( \
   logger(), __FILE__, __LINE__, #name, (name) )
// place for user defined logger formating data
#define LOG_ON() do { \
  logger().is_activated = true; } while(false)
#define LOG_OFF() do { \
  logger().is_activated = false; } while(false)
#if defined(CLEANLOG)
#undef LOG
#undef LOG_ON
#undef LOG_OFF
#undef LOG_FN
#define LOG(name) do{}while(false)
#define LOG_FN(name) do{}while(false)
#define LOG_ON() do{}while(false)
#define LOG_OFF() do{}while(false)
#endif
#endif // LOGGER_HPP_INCLUDED
```

Listing 4 (cont'd)

```
// Copyright (c) 2005, 2006
// Seweryn Habdank-Wojewodzki
// Distributed under the Boost Software License,
// Version 1.0.
// (copy at http://www.boost.org/LICENSE_1_0.txt)
#include "logger.hpp"
#if !defined(CLEANLOG)

#if defined (FTLOG)
#include <fstream>
#else
#include <iostream>
// http://www.msobczak.com/prog/bin/nullstream.zip
#include "nullstream.h"
#endif
logger_t::logger_t()
{}
bool logger_t::is_activated = true;

#if defined(TLOG)
std::auto_ptr < std::ostream >
    logger_t::outstream_helper_ptr
    = std::auto_ptr < std::ostream > (
    new NullStream );
std::ostream * logger_t::outstream = &std::cout;

#elif defined (ETLOG)
std::auto_ptr < std::ostream >
    logger_t::outstream_helper_ptr
    = std::auto_ptr < std::ostream > (
    new NullStream );
std::ostream * logger_t::outstream = &std::cerr;

#elif defined (FTLOG)
std::auto_ptr < std::ostream >
    logger_t::outstream_helper_ptr
    = std::auto_ptr < std::ostream > (
        new std::ofstream ( "_logger.out" ));
std::ostream * logger_t::outstream
    = outstream_helper_ptr.get();

// here is a place for user defined output stream
// and compiler flag

#else
std::auto_ptr < std::ostream >
    logger_t::outstream_helper_ptr
    = std::auto_ptr < std::ostream > (
    new NullStream );
std::ostream * logger_t::outstream
    = outstream_helper_ptr.get();
```

Listing 5

```
#endif
logger_t & logger()
{
    static logger_t * ans = new logger_t ();
    return *ans;
}

#endif // !CLEANLOG
```

Listing 5 (cont'd)

FTLOG forces the logger to put information to the file logger stream (listing 5). **TLOG** indicates that the logger sends information to the standard output. The logger will put information into the standard error stream if **ETLOG** is enabled. An interesting event occurs when no flag is chosen, the stream is set to a safe null stream. If a flag is set to **CLEANLOG**, then all macros are cleaned up.

The code uses the **NullStream** class defined by Maciej Sobczak [Sobczak (1)].

We can easily observe that such a construction of compiler directives leads to usage that we do not need any more as the **Log(name)** macro is defined every time. However it is empty **do{}while(false)** if **CLEANLOG** flag is set. If there is none of ...**TFLAG** set, it leads to the creation of the null output stream.

In every file where we need to use the functionality, we need to include `logger.hpp` and add o the makefile, `logger.cpp`.

To highlight the problem with the creation of the pointer to the stream in `logger.cpp`, a simple test case file is created (`foo.cpp`, listing 6), which has to be included to the example project. In listing 6, the logger is used before it starts in **main()**. This can lead to a problem when other `cpp` files in the project use it before the pointer is constructed. The problem is called "static initialization order fiasco". Happily, a solution exists. It is simple in concept, but tricky to perform and of course the author tries to do his best to solve the problem in the code using remarks from the Marshall Cline – Parashift.com website [Parashift].

```
#include "logger.hpp"
struct Foo
{
    Foo()
    {
        LOG ( "Creation of the Foo object" );
    }
    ~Foo()
    {
        LOG ( "Destruction of the Foo object" );
    }
};
Foo foo;
```

Listing 6

Usage and results

So, what functionality do we have thanks to the Trivial Logger?

The first observation of the code is that we can easily write `Log (var);` to log values of the variable which has name `var`. The result in the logger stream are the contents of `__FILE__ [__LINE__]` and the actual value stored in var (`var = value`) – in other words:

```
__FILE__ [__LINE__] :  var = value
```

The code uses a natively defined `operator <<` for the considered type, so dependent on that, the operator logs can change. A complete example of usage is shown in listing 6 and 7.

After running the example program with the flag `FTLOG`, new file `_logger.out` is created and it contains:

```
foo.cpp [7] : "Creation of the Foo object"
   = Creation of the Foo object
main.cpp [20] : "Trivial logger in main()!"
   = Trivial logger in main()!
main.cpp [21] : a = 1
main.cpp [22] : str = test
main.cpp [23] : p = ( PI, 3.1415 )
main.cpp [24] : q = ( 10, EUR )
main.cpp (30) : "Trivial logger in main()!"
   = Trivial logger in main()!
foo.cpp [12] : "Destruction of the Foo object"
   = Destruction of the Foo object
```

If `TLOG` is set, the same content is written in the console output, respectively for `ETLOG` flag, standard error output is chosen.

As we can see, we have information stored in the logger output even if the object `foo` is created before `main()`. What is very important for debugging and logging static is the construction of the objects. A similar method is presented for the destruction of the `foo` object.

Also in log file is the highlighted effect of the switching on and off the logger by using `LOG_OFF()` and switching on by using `LOG_ON()` – in `main()` the function `LOG()` is seen twice; one is put into the logger results.

At the very end of the `main.cpp` file is the macro `LOG_FN()` which is more less equivalent to the `LOG()` macro, but the difference is presented as a change of brackets. This macro also shows how to prepare other formatting functions and macros for Trivial Logger.

Configuration and simple code changes

There is not much configuration opportunity at this stage of the Trivial Logger construction. However, for the user there is available the pointer to the stream instead of the static variable. The pointer gives the opportunity to be changed at runtime. It is also possible to switch off and then switch on the logger, and also it's possible to change formatting style.

In the construction of the `logger_t` class static fields are used; the class can be redesigned not to use static variables. There has to be some small changes in `logger.hpp` file – mostly remove the `static` keywords. There is a larger change to be done in `logger.cpp` file – the construction of the fields has to be moved to the constructor of the `logger_t` class – of course with respect to the creation of non static fields of the class. The basic functionality of the logger remains similar – extended functionality in that we can define many different loggers (see in the Further extensions section).

Further extensions

There are many extensions possible. The first one is that the logger is somehow a singleton, so the code can be redesigned to use a singleton design pattern, but it will *not* be "trivial" [Alexandrescu01].

Other propositions can be to define different streams for different purposes e.g. one for logger other for debugging purpose, another for collecting data from UI to prepare some statistics about usage of the UI. This can be solved by defining in parallel similar macros (with different names) or by using this functionality in the more general concept, what will be not trivial at all if we consider native system logging mechanisms, threads and so on. Static pointers have to be changed to the normal ones.

An interesting extension is that, in fact, Trivial Logger defines pointer to the stream and we could implement other streams to use with a user-defined compiling flag. Interesting examples of such a stream objects are implemented in streamed socket implementation [Sobczak (2)], also FASTreams [Sobczak (3)] and the Boost iostreams Library [Boost].

The last extension concerns the used file name which is fixed as `_logger.out`. Another possibility is to use function `std::tmpnam` from the C standard library to generate a temporary file for logging [Dinkumware]. This approach can be useful if we combine the functionality of Trivial Logger and use it in a multi-threaded application but with the separation of the log files for every critical thread. This approach can be treated as second extension, too.

With respect to multi-threaded and or Real-Time systems, logging information can be extended by using time stamps, which informs about a time of the logged value.

Also very simple is EzLogger [Axter]. It is a set of seven (in fact six) headers which are easy to include to the project. It supports levels of verbosity and works on streams, but the overhead is quite heavy for such a simple logger. It also supports changing formatting policy, but it seems to force the user to generate their own policy classes. There is no way to clean up its functionality, so it can not be used just only for debugging purpose.

Some not trivial loggers

More or less all extensions will lead to the construction of a really big logger. As examples of design we can compare: log4cxx, log4cpp and

```
#include <string>
#include <utility>

#include "logger.hpp"

template < typename T, typename U >
std::ostream & operator<< ( std::ostream & os,
   std::pair < T, U > const & p )
{
    os << "( " << p.first << ", "
       << p.second << " )";
    return os;
}

int main ()
{
    double const a = 1.0;
    std::string const str = "test";
    std::pair < std::string, double > p (
       "PI", 3.1415 );
    std::pair < unsigned long, std::string > q (
       10, "EUR" );

    LOG ("Trivial logger in main()!");
    LOG (a);
    LOG (str);
    LOG (p);
    if ( 1 ) LOG (q); else LOG(str);

    LOG_OFF();
    LOG ("Trivial logger in main()!");

    LOG_ON();
    LOG_FN ("Trivial logger in main()!");
}
```

Listing 6

log4cplus [Apache (1)] [Sourceforge (1)] [Sourceforge (2)]. As we can read, all of them are based on Java Log4j project design [Apache (2)].

- **log4cxx** is equipped with loggers and they give possible different levels of logging such as (shown with the relation): DEBUG < INFO < WARN < ERROR < FATAL. Additionally, it has different appenders, so the programmer can chose: console output, files, GUI components, remote socket servers, NT Event Loggers, and remote UNIX Syslog daemons.

  The usage is very simple e.g.:
  ```
    LOG4CXX_INFO(logger, "Exiting application.");
  ```
  log4cxx is distributed under the Apache Software License.
- **log4cpp** is smaller than log4cxx. However it has extended levels of logging: NOTSET < DEBUG < INFO < NOTICE < WARN < ERROR < CRIT < ALERT < FATAL = EMERG. Unfortunately the documentation is not good enough, so we need to look to the programmers documentation to recognize that appenders are: files, files with set maximal size (logs will rotation of the file contents), standard streams, strings (logging into memory), remote and local Syslog and the default system debugger on Win32 systems.

  The usage is different:
  ```
    file_appender.log(log4cpp::Priority::WARN,
                      "This will be a logged warning");
  ```
- log4cpp is released under the GNU Lesser General Public License (LGPL).

  log4cplus is even smaller than log4cpp. It has some levels: TRACE < DEBUG < INFO < WARN, ERROR < FATAL. Possible appenders are: console output, file, file with set maximal size (similar to log4cpp), file with set time of rotation (file is rolled over at a user chosen frequency), socket, Syslog and NT Event Log and at last NULL.
  The usage of log4cplus is as simply as in log4cxx:
  ```
    LOG4CPLUS_WARN(logger, "Hello, World!")
  ```
  log4cplus is released under the Apache Software License.

To summarize. All of none trivial loggers are: multi-threaded, equipped with many appenders, and they separate logging levels. They contain hierarchical loggers and filters. All support NDC (Nested Diagnostic Context), which is a design technique for loggers in multi-clients and multi-threaded applications to separate events from clients. They can be configured from pure text and/or XML files. They can change configuration in runtime.

The design is copied from Java design style which is very good kind of OO programming, but it is far from modern C++ design. The best documentation is prepared for log4cxx library, other libraries have only documentation of the API, and simple examples of usage. ■

Acknowledgement

Author would like to give thanks, in alphabetical order, to: Wiktor Adamski, Jedrzej Dudkiewicz, Alan Griffiths, Paul Johnson, Mateusz Loskot, Roger Orr, Ric Parkin and Piotr Wyderski for all remarks which helped improve the code and clarify all problems.

Reference

[Josuttis99] N. M. Josuttis, The C++ Standard Library: A Tutorial and Reference, Addison Wesley Professional, 1999.
[Sobczak (1)] http://www.msobczak.com/prog/bin/nullstream.zip
[Parashift] http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.12
[Alexandrescu01] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley Professional, 2001.
[Sobczak (2)] http://www.msobczak.com/prog/bin/sockets.zip
[Sobczak (3)] http://www.msobczak.com/prog/fastreams/
[Boost] http://www.boost.org/libs/iostreams/doc/index.html
[Dinkumware] http://www.dinkumware.com/manuals/?manual=compleat&page=stdio.html#tmpnam
[Axter] http://axter.com/ezlogger/
[Apache (1)] .http://logging.apache.org/log4cxx/
[Sourceforge (1)] http://log4cpp.sourceforge.net/
[Sourceforge (2)] http://log4cplus.sourceforge.net/
[Apache (2)] http://logging.apache.org/log4j/docs/index.html

FRUCTOSE – a C++ Unit Test Framework

Andrew Marlow describes the developmentof FRUCTOSE and how it is different from other unit test frameworks.

Introduction

FRUCTOSE is designed to be much smaller and simpler than frameworks such as CppUnit. It offers the ability to quickly create small standalone programs that produce output on standard out and that can be driven by the command line. The idea is that by using command line options to control which tests are run, the level of verbosity and whether or not the first error is fatal, it will be easier to develop classes in conjunction with their test harness. This makes FRUCTOSE an aid to Test Driven Development (TDD). This is a slightly different way of approaching unit test frameworks. Most other frameworks seem to assume the class has already been designed and coded and that the test framework is to be used to run some tests as part of the overnight build regime. Whilst FRUCTOSE can be used in this way, it is hoped that the class and its tests will be developed at the same time.

A simple unit test framework should be concerned with just the functions being tested (they do not have to be class member functions although they typically will be) and the collection of functions the developer has to write to get those tests done. These latter functions can be grouped into a testing class. This article shows by example how this testing class is typically written and what facilities are available.

There are certain kinds of assertion tests that are useful for any unit test framework to offer. This article discusses those and how they are offered by FRUCTOSE. In particular, FRUCTOSE offers loop assertion macros. These are designed to be used in tests that use static tables for their test data. An assertion failure from test data needs to report not only the line of code that contains the assertion but also the line from the test data table.

FRUCTOSE has been developed as free software [FSF]. This meant chosing an appropriate licence for it. The intent is to have a licence that is non-product specific with an element of copyleft but that still allows FRUCTOSE to be used in proprietary software. The licence chosen was the LGPL license (Lesser General Public License). The weaker copyleft provision of the LGPL allows FRUCTOSE to be used in proprietary software because the LGPL allows a program to be linked with non-free modules. A proprietary project can use FRUCTOSE to unit test its proprietary classes and does not have to release those classes under the GPL or LGPL. However, if such a project releases a modified version of FRUCTOSE then it must do so under the terms of LGPL. In the search for a suitable license the LGPL was the winner by default. It is the only licence listed on the Free Software Foundation's web site that is non-product specific and a GPL-compatible free software licence that has some copyleft provision.

Why another C++ unit test framework?

When I first started to look seriously at C++ unit testing, I looked at CppUnit. Conventional wisdom said to build on the work of others rather than reinvent a framework, and here was an established one which was itself built on the work of JUnit. The feature list is impressive and it looked like it would be mature enough to give a trouble-free build and be usable right away. Unfortunately, this proved not to be the case. A quick search of sourceforge reveals that there are a few products for C++ unit testing. These also turn out to have issues of their own (discussed later). In October's issue of Overload, there is an article by Peter Sommerlad on a C++ unit testing framework called CUTE [CUTE]. This article also mentions that there are issues with using CppUnit and that some developers want something else that is smaller and simpler. However, at the time I started this article CUTE was not available online and Peter was too busy to work on a collaboration. This provided the motivation for me to write something that is driven by the same need expressed in the CUTE article (smaller and simpler than CppUnit). There are some important differences between FRUCTOSE and CUTE. FRUCTOSE avoids two significant dependencies; one on Boost and the other on platform-specfic RTTI.

The FRUCTOSE approach

FRUCTOSE has a simple objective: provide enough functionality such that the developer can produce one standalone command line driven program per implementation file (usually one class per file). This means that most FRUCTOSE programs will use only two classes; the test class and the class being tested. This means that, unlike other test frameworks, FRUCTOSE is not designed to be extensible. It was felt that the flexibility of other frameworks comes at the cost of increased size and complexity. Most other frameworks expect the developer to derive other classes from the framework ones to modify or specialise behaviour in some way, e.g. to provide HTML output instead of simple TTY output. They also provide the ability to run multiple suites. FRUCTOSE does not offer this. The test harness is expected to simply consist of a test class with its test functions defined inline, then a brief main function to register those functions with test names and run them with any command line options.

The Curiously Recurring Template Pattern (CRTP)

Having said that FRUCTOSE is smaller and simpler than other frameworks, I have to confess that when one writes a test class that is to be used with FRUCTOSE, the the test class needs to inherit from a FRUCTOSE base class. Not all test frameworks require inheritance to be used but FRUCTOSE does. Also the style of inheritance employs a pattern that some people may not have seen before. It is known as the Curiously Recurring Template Pattern (CRTP) [Vandevoorde and Josuttis]. CRTP is where one inherits from a template base class whose template parameter is the derived class. This section explains why.

There is a need for machinery that can add tests to a test suite and run the tests. This is all done by the test class inheriting from the FRUCTOSE base class, `test_base`. This base class provides, amongst other things, the function `add_test`, which takes the name of a test and a function that runs that test. The functions that comprise the tests are members of the test class.

Andrew Marlow has been in software development for over twenty years, placing him firmly in the category of "Grumpy Old Programmer". He started with OS development but is now involved in financial data feeds in the City. Contact him through his website: http://www.andrewpetermarlow.co.uk.

The authors of some test frameworks feel that the requirement for a test class to have to inherit from anything is unreasonable

So **test_base** needs to define a function that takes a member function pointer for the test class. CRTP is used so that the base class can specify a function signature that uses the derived class.

Suppose our test class is called **simpletest**. Its declaration would start like this:

```
struct simpletest :
  public fructose::test_base<simpletest> {
     :
     :
```

Let's see how this works: **test_base** names its template parameter **test_container** (it is the class that contains the tests). **test_base** declares the typedef **test_case** expressed in terms of **test_container**:

```
typedef void (test_container::*test_case)
                     (const std::string&);
```

This enables it to declare **add_test** to take a parameter of type **test_case**. **test_base** maintains a map of test case function pointers, keyed by test name. The declaration for this map is:

```
std::map<std::string, test_case> m_tests;
```

The declaration of the **add_test** fuction is:

```
void add_test(const std::string& name,
              test_case the_test);
```

Listing 1 is an example of a complete test harness, showing how the use of CRTP means the test cases are defined in the test class and registered using **add_test**.

Named tests and command line options

The example in the previous section shows that tests are named when they are registered but the example does not actually make any practical use of this. Using the **run()** function above, all registered tests are run. FRUCTOSE does provide a way to select which tests are run via command line options. Basically, the names of the tests are given on the command line. This is done by using the overloaded function **int run(int argc, char* argv[]);**.

The Open Source library TCLAP is used to parse the command line [TCLAP]. The command line is considered to consist of optional parameters followed by an optional list of test names. During parsing, TCLAP sets various private data members according to the flags seen on the command line. These flags are available via accessors such as **verbose()**. This is another reason why the test class has to inherit from a base class. It provides access to these flags.

```
#include "fructose/test_base.h"
const int neighbour_of_the_beast = 668;
struct simpletest :
  public fructose::test_base<simpletest> {
    void beast(const std::string& test_name) {
      fructose_assert(
       neighbour_of_the_beast == 668)
    }
};
int main(int argc, char* argv[]) {
  simpletest tests;
  tests.add_test("beast", &simpletest::beast);
  return tests.run();
}
```

Listing 1

To inherit or not to inherit

The authors of some test frameworks feel that the requirement for a test class to have to inherit from anything is unreasonable. It it said that such a requirement makes the test framework hard to use and causes undesirable coupling between the test class and the framework. FRUCTOSE has several things to say in response to this:

- Some test frameworks that employ inheritance may be hard to use (e.g. CppUnit) but it does not follow that inheritance is the cause. Some frameworks have just become large and complex during their evolution, and offer the developer and bewildering number of choices for the design of their test classes.
- CRTP can seem slightly daunting to those that have not seen it before. Howver, the use of CRTP by FRUCTOSE is quite simple and is there simply to enforce that the code that does the tests is in functions of the test class. If anyone knows of any other way to enforce this, I would be most interested to hear from them.
- FRUCTOSE applications are intended to be run as command line programs with the ability to use various command line options that come as part of FRUCTOSE. The test class gets this capability by inheritance. If anyone knows of a better way to do, this I would be most interested to hear from them.
- FRUCTOSE only makes a handful of functions available. There are **add_test** to register the tests and **run** to run them. The assertion testing macros (discussed later) rely on a function in the base class but that is an implementation detail that is of no concern to the programmer. Other FRUCTOSE functionality such as the command line options accessors is optional. Hence, the amount of coupling between the test class and the FRUCTOSE base class is actually quite low.
- The FRUCTOSE assert macros include the test name in any assertion failure message because the function **get_test_name()** is available to any class that inherits from **test_base**. This is part of what enables FRUCTOSE to have named tests. Without using this technique it is hard to see how user-friendly test names can be registered and used by the framework. This was a difficulty mentioned in the CUTE article. CUTE overcame the problem by

an assertion failure does not abort the function from which it was invoked

using a compiler-specific demangle routine. Other frameworks just don't allow the user to name the tests at all.

- Some of the test frameworks I have examined avoid the need for the test class to inherit from a base class by their assert macros expanding to a large volume of code. The sort of code these macros expand to is the kind that FRUCTOSE places in the base class. FRUCTOSE takes the view that in general macros should be avoided in C++. However, FRUCTOSE does use macros for its asserts. It does this for two reasons: first it needs to get the assertion expression as a complete string much as the C assert facility does; and second, it uses the `__FILE__` and `__LINE__` macros to report the filename and line number at which the assert occurs. The macro definitions are small.
- Because the test class inherits from a base class to get all the functionality required, the writer of the test harness only needs to worry about two classes; the one being tested and the one doing the testing. This is felt to be a great simplication compared to other frameworks.

FRUCTOSE actually has two classes, `test_base` and `test_root`. The user sees the former since it must be inherited from but does not see the latter (it is an implementation detail). `test_base` inherits from `test_root`. This is a division of labour; `test_base` contains code that uses the template argument. `test_root` contains code that is not required to be in a template class. The reason for this is largely historical; during the early stages of FRUCTOSE design it came as a library that had to be built, then linked with. `test_root` was in the library whilst the template code was all in the headers and instantied at compile time. When FRUCTOSE changed to be implemented entirely in header files (inspired by the same approach in TCLAP), the separation of classes was retained. It still provides a distinction between code that is required to be template code and that which does not.

`test_root` contains the following:

- private data members and associated accessors for the flags read from the command line.
- A private data member and associated accessor for the name of the current test.
- A count of the number of assertion errors, plus an associated accessor and mutator.
- Functions that implement most of the assertion code.
- Default `setup` and `teardown` functions.

The assert macros

The FRUCTOSE assert macros are compared with the classic C assert and the assert macros of other unit test frameworks.

First, there are a couple of minor style points about the naming of the FRUCTOSE macros.

1. FRUCTOSE uses the namespace `fructose` to scope all its externally visible symbols. Macros have global scope so they are prepended with `fructose_`. This makes the naming and scoping as consistent as possible. Too many unit test frameworks call their main assert macro `ASSERT`.
2. Generally one chooses uppercase for macros in order to tip the reader off that the token is a macro. However, where the macro is intended to look and feel like a function call macros are sometimes in lowercase. Well known examples include `assert` and `get_char`. Also, `toupper` and `tolower` are sometimes implemented as macros. The FRUCTOSE macros are expected to be used in a similar way to the standard C assert macros and are designed to look and behave as function calls. Hence they are in lowercase.

The `fructose_assert` macro expands to a call to a function, `test_assert`, which takes the boolean result of the assertion expression, the test name (obtained via the function call `get_test_name()`), the assertion expression as a string, the source filename and line number. Here is the definition:

```
#define fructose_assert(X) \
    { fructose::test_root::test_assert((X), \
      get_test_name(), #X, __FILE__, __LINE__);}
```

The `test_assert` function takes no action if the condition evaluates to true. However, when the condition is false it reports the name of the test that failed, the filename and line number and the assertion expression. It also increments the error report (for a final report at the end of all tests) and optionally halts (depends on whether or not the command line option has been specified to halt on first failure).

Note, unlike some other test frameworks, an assertion failure does not abort the function from which it was invoked. This is deliberate. Other test frameworks take the view that when a test fails all bets are off and the safest thing to do is to abort. FRUCTOSE takes a different view. FRUCTOSE assumes it is being used as part of a TDD effort where the developer will typically be developing the class and its test harness at the same time. This means that the developer will want to execute as many tests as possible so he can see which pass and which fail. It also means that the developer needs to be aware that code immediately after a FRUCTOSE assert will still be executed so it should not rely on the previous lines having worked. One style of writing test cases that is in keeping with this is to use a table of test data. Each row in the table contains the input and the expected output. This allows the test code to loop over the table performing lots of tests without a test having to rely on successful execution of the previous test.

The FRUCTOSE command line

The following command line options come as standard for every unit test built using FRUCTOSE:

- `-h[elp]` provides built-in help. Not only does it produce help on all the options here but also it names the tests available so they can be run selectively.

the user of the framework does not have to worry about the existence of any classes other than the one he is testing and the test class he is testing it with

- `-a[ssert_fatal]` causes the test harness to exit upon the first failure. Normally the harness would continue through any assertion failures and produce a report on the number of failed tests at the end. The developer needs to be aware of this when coding the tests and ensure that in a given test function that has a number of assertions, the correct working of the tests does not depend on the assertions passing. If the developer finds there are such dependencies these tests should be rearranged into separately named tests. Sometimes this is awkward to do so the flag is provided for these cases. When this flag is enabled, the first assertion failure results in an exception being thrown, which is caught and reported by the run function.
- `-v[erbose]` this sets a flag which can be tested in each test function. This allows test functions to output diagnostic trace when the option has been enabled. This is of particular use during TDD. The developer may find it useful to leave a certain amount of this trace in, even when all the tests pass, in case there is a regression, as a debugging aid.
- `-r[everse]` reserve the sense of all assertion tests. This is primarly of use when testing the framework itself.
- The remaining command line options are taken to be test names. All supplied test names are checked against the registered test names. Only registered test names are allowed.

When the example program above is run with the `-help` option, the following output is produced:

```
USAGE:

  ./example  [-h] [-r] [-a] [-v] [--]
    <testNameString> ...

Where:

   -h,  --help
     -h[elp]

   -r,  --reverse
     -r[everse]

   -a,  --assert_fatal
     -a[ssert_fatal]

   -v,  --verbose
     -v[erbose]

   --,  --ignore_rest
     Ignores the rest of the labeled arguments
     following this flag.

   <testNameString>  (accepted multiple times)
     test names

  -verbose turns on extra trace for those tests
  that have made use of it.

  -assert_fatal is used to make the first test
  failure fatal.

  -help produces this help.

  -reverse reverses the sense of test assertions.

  It is  only used to test the test framework
  itself.

  Any number of test names may be supplied on the
  command line. If the name 'all' is included then
  all tests will be run.

  Supported test names are:
    beast
```

The above test should pass, so to see the kind of report that is given when a test fails, run the harness with the `-reverse` option. It produces the following output:

```
Error: beast in ex3.cpp(6): neighbour_of_the_beast
== 668 failed.

Test driver failed: 1 error
```

How FRUCTOSE works

Consider the `simpletest` example above. The `simpletest` class is referred to as the test class. This provides all the functions that do the testing. It must inherit from `test_base` using the curiously recurring template pattern (CRTP). It does this for several reasons.

1. FRUCTOSE maintains a map of function pointers for when it has to invoke those functions. The function pointer type needs to be declared which means establishing a calling convention. FRUCTOSE takes the view that the function should be a member function of the test class whose return type is void and that takes the test name as a `const std::string` reference. This is enforced at compile time by use of CRTP, which allows `test_base` to declare the function pointer type using the name of the test class.
2. `test_base` makes the function `bool verbose() const` available, which returns `true` if the verbose flag was given on the command line.
3. FRUCTOSE avoids users having to know about several classes by providing the test registration function `add_test` and the test runner function, `run`, via the base class. It also provides an overloaded `run` function that parses the command line and runs the tests specified. These functions are not only convenient, they ensure that the user of the framework does not have to worry about the

FRUCTOSE is designed to work in commercial environments as well as open source environments

existence of any classes other than the one he is testing and the test class he is testing it with.

It is felt by some that when a framework forces the test class to inherit from anything this is an unreasonable requirement.

The argument says that because inheritance is very strong coupling between classes this arrangement couples the test case too tightly to the framework. The trouble is, there has to be some coupling between the test class and the machinery that invokes its functions. If nothing else, the invoking machinery must establish a call convention for the functions that it calls.

FRUCTOSE uses a convention of the test function having the void return type and taking a const `std::string` reference to the test name, which must be called `test_name` (this is required by the FRUCTOSE test macros). As another example, CUTE requires that the test function have the void return type and have no function arguments in order that a boost functor may be implicitly created when a test function is passed to the CUTE macro. The CUTE approach does eliminate the coupling by inheritance but does so at the cost of not being able to explicitly name the test. Also, there is no particular advantage in allowing the test functions to be unrelated. In fact one could argue that they should be all grouped in the same test class on the principle of grouping related things together.

FRUCTOSE has the idea that tests are registered by name. The name has to be explicitly given in the `add_test` function. It is not deduced from the names or functions or the use of RTTI. This allows the test to be referred to from the command line. The `test_case` class provides the registration and command line parsing functions.

FRUCTOSE coding techniques

FRUCTOSE is designed to work in commercial environments as well as open source environments. Commercial environments place some constraints on the C++ coding techniques that can be used. In the commercial world ancient compilers are still very much alive and well (for various reasons). Also, multiple operating systems have to be supported (Microsoft Windows and Solaris are probably the most important). This means that the lowest common denominator approach has to be taken for the dialect of C++ chosen. Platform-dependent RTTI, use of complex template meta-programming, and other advanced C++ techniques were avoided. So were dependencies upon packages that use such techniques.

FRUCTOSE achieves what it needs by simple inheritance. True, it uses CRTP, but the main reason for this is so the function pointers it maintains have to be functions that belong to the test class.

The STL is also used. Usage is kept very simple. Strings are always of type `std::string`. The function pointers to run are held in a `std::map`, keyed by test name string. The list of named tests to run is held in a `std::vector`. The exception handling macros also use the standard exception header `stdexcept` for things such as catching exceptions by the standard base class, `std::exception`. The headers required for these uses of the STL and standard exceptions are automatically included by FRUCTOSE so there is no need for the test harness to include them again. All the functions are inlined, so there is no library to link against; just use the header files.

Assertions

Although FRUCTOSE uses words such as ‘assert’ and ‘assertions’, one must bear in mind that these are not C-style `assert` statements. They do not cause core dumps. They produce diagnostic output and increment an error count in the event that a tested condition returns `false` (i.e. they are asserting that the supplied condition should be true). They are macros and use the `__FILE__` and `__LINE__` macros to show which file and line the error occurred on. The simple case is the `fructose_assert` macro, which produces an error if the supplied condition is false.

A number of other assertion macros are provided:

- `fructose_assert_eq(X,Y)` Assert that X equals Y. If they are not than the names of X and Y and their values are reported. This level of detail would not be present if the developer used `fructose_assert(X == Y)` instead.
- `fructose_assert_double_eq(X,Y)` A familiy of floating point assertions is provided. Substitute `lt`, `gt`, `le`, or `ge` for `eq` to check for less than, greater than, less than or equal to and greater than or equal to. Floating point assertions are a special case for two reasons: first, floating point compares need to be done with configurable relative tolerance and or absolute tolerance levels. Second, the default left shift operator is insufficient to show enough precision when a floating point compare has failed. The macro family mentioned does floating point compares using default tolerances. The macro family

  ```
  fructose_assert_double_<test>_rel_abs(
   X,Y,rel_tol,abs_tol)
  ```
  provides the same tests but with the tolerances specified explicitly.
- Loop assertions. Macros are provided that help the developer track down the reason for assertion failures for data held in static tables. What is needed in these cases in addition to the file and line number of the assertion is the line number of the data that was tested in the assert and the loop index. There is a family of macros for this named `fructose_loop<n>_assert`, where `<n>` is the of looping subscripts. For example, when the array has one subscript the macro is `fructose_loop1_assert(LN,I,X)` where `X` is the condition, `LN` is the line number of the data in the static table and `I` is the loop counter. `fructose_loop2_assert(LN,I,J,X)` tests condition `X` with loop counters `I` and `J`.
- Exception assertions.The test harness may assert that a condition should result in the throwing of an exception of a specified type. If it does not then the assertion fails. Similarly, a harness may assert that no exception is to be thrown upon the evaluation of a condition; if one is then the assertion fails.

  `fructose_assert_exception(X,E)` asserts that when the condition `X` is evaluated, an exception of type `E` is thrown.

```
#include "fructose/test_base.h"
#include <cmath>

struct simpletest :
  public fructose::test_base<simpletest> {
    void floating(const std::string& test_name) {
      double mypi = 4.0 * std::atan(1.0);
      fructose_assert_double_eq(M_PI, mypi);
    }
};
int main(int argc, char* argv[]) {
  simpletest tests;
  tests.add_test("float", &simpletest::floating);
  return tests.run(tests.get_suite(argc, argv));
}
```

Listing 2

Floating point assertions

Comparing floating point numbers for exact equality is not always reliable, given the limitations of precision and the fact that there are some real numbers that can be expressed precisely in decimal but not precisely in binary. The common way around this is to compare floating point numbers with a degree of fuzziness. If the numbers are "close enough" then they are judged to be equal.

A common mistake that is made in fuzzy floating point comparisons is for closeness check to be done using an absolute value for the allowed difference between the two values. The perils of doing this are explained in great detail in section 4.2 of *The Art of Computer Programming* [Knuth], where Knuth explains how to use relative tolerances to overcome problems. Very few unit test harnesses seem to provide much to help here. See Listing 2 for a simple floating point compare assertion with default tolerances.

Since `PI` is equal to four times `arctan(1)`, this assertion will pass. Using the `-reverse` option shows the kind of output that is produced when such a test fails:

```
Error: float in ex3.cpp(8):
   M_PI == mypi (3.141592653589793e+00 ==
         3.141592653589793e+00) failed floating
point compare.

Test driver failed: 1 error
```

Note that the numbers are output in scientific format with the maximum number of significant figures available in most implementations of `double`s.

Loop assertions

Consider the class `multiplication` which provides a function `times` that returns the constructor arguments x and y multiplied together.

```
class multiplication {
  double m_x, m_y;
public:
  multiplication(double x, double y)
     : m_x(x), m_y(y) {};
  double times() const {return m_x * m_y; };
};
```

A FRUCTOSE unit test can use a table of test data of values for x and y and the expected result: One could use the `fructose_assert` macro to test that the expected value is equal to the computed value (see Listing 3).

However, this is not very useful when there is an assertion failure because it doesn't tell you which assertion has failed. One way is to add verbose tracing that gives all the detail, as shown in Listing 4.

```
#include "fructose/test_base.h"

struct timestest :
  public fructose::test_base<timestest> {
    void loops(const std::string& test_name) {
      static const struct {
          int line_number;
          double x, y, expected;
          } data[] = {
        { __LINE__, 3,      4,    12}
      , { __LINE__, 5.2,    6.8,  35.36}
      , { __LINE__, -8.1,   -9.2, 74.52}
      , { __LINE__, 0.1,    90,   9}
      };
      for (unsigned int i = 0;
           i < sizeof(data)/sizeof(data[0]); ++i) {
             multiplication m(data[i].x,
data[i].y);
             double result = m.times();
         fructose_assert(
           result == data[i].expected);
      }
  }
};

int main(int argc, char* argv[]) {
  timestest tests;
  tests.add_test("loops",
                 &timestest::loops);
  return tests.run(argc, argv);
}
```

Listing 3

However, another way which does not rely on the verbose flag is to use the loop assert macro family. These macros take the source line number of the test data and loop indexes as macro parameters. The code below shows the test modified to indicate the line of data and the loop index value. This makes the use of the verbose flag unnecessary.

```
      for (unsigned int i = 0;
           i < sizeof(data)/sizeof(data[0]); ++i) {
             multiplication m(data[i].x, data[i].y);
             double result = m.times();
             fructose_loop1_assert(
                    data[i].line_number, i,
                    result == data[i].expected);
      }
```

If the code code employed an array with two dimensions and thus had nested loops, one would use the macro:

```
fructose_loop2_assert(lineNumber, i, j, assertion
```

```
  for (unsigned int i = 0;
      i < sizeof(data)/sizeof(data[0]); ++i) {
        multiplication m(data[i].x, data[i].y);
        double result = m.times();
        if (verbose()) {
          std::cout << data[i].x
                    << " * " << data[i].y
                    << " got " << result
                    << " expected "
                    << data[i].expected
                    << std::endl;
        }
        fructose_assert(
         result == data[i].expected);
      }
```

Listing 4

```
#include "fructose/test_base.h"
#include <stdexcept>
#include <vector>

struct timestest :
  public fructose::test_base<timestest> {
    void array_bounds(
       const std::string& test_name) {
      std::vector<int> v;
      v.push_back(1234);
      fructose_assert_exception(v.at(2),
                                std::out_of_range);
    };
};

int main(int argc, char* argv[]) {
  timestest tests;
  tests.add_test("array_bounds",
           &timestest::array_bounds);
  return tests.run(tests.run(argc, argv);
}
```

Listing 5

No other unit test framework that was examined provides loop asserts. These are very useful because they encourage the developer to to do systematic testing by covering more cases more conveniently.

The convenience of loop assert testing does not mean the developer can provide large volumes of test data just for the sake of appearing to do large amounts of tests. It is hoped that the loop asserts will lead to an increased use of a technique used in testing known as equivalence partitioning [Pressman]. This is a method that divides the input domain into classes of data from which test cases can be derived. All the data for the individual cases in all these data classes for a given FRUCTOSE test would be in static data table such as the one shown above. The classes would be grouped in the table with comments to show the grouping of classes of errors. An ideal test case uncovers a whole class of errors on its own that might otherwise require many cases to be executed. There is another technique called Boundary Value Analysis [Pressman], which loop asserts are well suited to.

Exception handling

FRUCTOSE only uses exceptions to deal with errors if the **`-assert_fatal`** flag is given on the command line. But FRUCTOSE realises that a class being tested may throw an exception which the developer did not expect to occur during the run of the unit test. This is treated as a fatal error. It is caught and reported and causes the unit test to terminate.

The developer may wish to test that certain exceptions are thrown when they are meant to be (Listing 5). The macro

```
fructose_assert_exception(X,E)
```

is provided for this. It asserts that during the evaluation of the condition **`X`**, an exception of type **`E`** is thrown. If exception of a different type is thrown, or no exception is thrown, then the assertion fails.

Setup and teardown

When FRUCTOSE was first developed it was felt that the setup and teardown machinery offered by other frameworks would not be required. However, it was later discovered that on relatively rare occasions it is useful. Most of the time if any setup and teardown procedure is required at all it is needed once at the start and finish of the program. In these cases it can be done in the constructor and destructor of the test class. But there will be cases where the setup and teardown need to be done for each test. Hence, **`setup`** and **`teardown`** functions are provided as virtual functions with an empty default implelementation in **`test_root`**. If the test class needs to override these then it can do so by providing its own **`setup`** and **`teardown`** functions. These get called by the **`run`** function of **`test_base`** before and after each test invocation.

The problems with CppUnit

When I first looked into providing a unit test environment for a commercial project I was working on, I was advised to look at Cppunit, so I did. I found that it would not build on the Solaris development environment we had (Forte 6.0). A port was in progress at the time but we needed something immediately. It was too much work to do a port ourselves when we were supposed to be using something off the shelf. We were also using a non-standard version of the STL and had to ensure that everything we built would build with that STL. The build procedure for CppUnit made this awkward. These problems are very specific to the development environment I was in. I made the recommendation that CppUnit not be used. Some of the reasons I gave were these environment-specific reasons but I also quoted the following reasons, which it seems have been the experience of others:

- Other people also find it hard to build. There is even a manual on the Wiki pages explaining how to build for various platforms. It should not be that complicated!
- CppUnit is very large for a unit test framework. The tarball is over 3MB once uncompressed. Admittedly, quite a bit of this is documentation and examples but no other C++ unit test framework I looked at was anywhere near this size. It takes quite a while to build it too (over four minutes on my dedicated Linux machine).
- It is hard to use. That's why there are lots of tutorials, lots of documentation and even a cookbook, discussion forums and an FAQ.
- CppUnit is too large and complex for many people's needs: I am sure that the reason for this volume of documentation is that CppUnit has a lot of functionality to offer. However, in my opinion it does so at the cost of frightening off the developer who only needs something simple. The number of C++ unit test frameworks that have sprung up, all with the goal of providing something smaller, cutdown and simpler, are a testimony to the size and complexity of CppUnit.
- Even in the simple cases, CppUnit places too many complex requirements on the developer, particularly regarding new classes to be written and which base classes they are supposed to inherit from. In most of the alternatives to CppUnit, there is only one class to inherit from (in the case of CUTE there is no need even for that). In CppUnit there is a choice of inheriting from **`TestCase`** or **`TestFixture`**. There are also **`TestRunner`**s, **`TestCaller`**s and **`TestSuite`**s to worry about.

The problems with other frameworks

My search of sourceforge revealed several C++ unit test frameworks. However, the problem was usually that it was either for a specific environment or it did not allow the test selection and verbose flag setting via the command line that are so useful when doing TDD. The packages considered include the following:

- unit---- [Unit]. The Unit test aid for C++. Judging from the examples and in the opinion of this author, unit---- is too cut-down, not providing much of the basic functionality provided by the other packages.
- csUnit [csUnit]. It is for managed C++.
- Symbian OS C++ Unit Testing Framework [Symbian]. It is only for the Symbian operating system,
- RapidoTest [Rapido]. It is for Unix only with particular emphasis on Linux.
- Mock Objects for C++ [MockObject]. It is a framework that builds on a framework; it provides mock objects by building on either CppUnit or cxxunit.

- UnitTest++ [UnitTest++]. This actually comes quite close. It is much smaller and simpler than CppUnit and I did not encounter any build problems. It is multi-platform. However, there is no built-in control over which tests to run, neither can the test be identified during the run. This is fine for overnight regression testing but is not so good for TDD.
- QuickTest [QuickTest]. This was discovered after FRUCTOSE was released. QuickTest only consists of one very small header file with no documentation and no examples. It's approach has alot in common but FRUCTOSE is slightly richer in functionality, particularly with the test assertions than can be made and the command line and named test features. Again, this makes it more suitable for TDD. FRUCTOSE also comes with documentation and a couple of examples.
- CppTest [CppTest]. A slight wrinkle was found in the build procedure – doxygen is mandory otherwise the configure script will not produce a Makefile. Apart from that this package looked to be quite good, more mature and than QuickTest, also with better documentation. However, in common with QuickTest and UnitTest++ there does not seem to be a mechanism for selecting which tests to run, controlling verbosity and so on.
- Boost test. This was avoided because of the direct dependency on Boost. The test library has to be built in a similar way to Boost, which is well known for having a complex Unix-centric build procedure. Like other frameworks, Boost test has separate classes for tests and the ability to runs the tests. It lacks the ability to name the tests and run them selectively.

The advantages of FRUCTOSE

The main strength of FRUCTOSE compared to the packages above, is its emphasis on its use during code development. This is via its features for reporting in detail the test that failed, the ability to supplement this trace with diagnostics controlled by the verbose flag on the command line, and the ability to select tests by name and optionally fail at the first test failure. None of the other packages provide this; their focus seems to be on running batches of testing in an overnight run to detect regressions. FRUCTOSE will also do that but provides the command line flexibility as well.

Another strength of FRUCTOSE is that it only has one external package dependency. It uses TCLAP [TCLAP] for command line argument handling. This is a very small dependency, since TCLAP is quite small and is implemented entirely in header files. Some other frameworks have a larger set of dependency requirements and some of these dependencies are non-trivial. For example, some depend on Boost. Whilst Boost is recognised to be a fine set of high-quality libraries, some projects, particularly those in some commercial environments, do not want to depend on it. This is for several reasons:

1. Boost does not yet build out of the box in some commercial environments. This is the fault of the compilers, not the fault of Boost. But sometimes a commercial project has little choice of which compiler to use. This can be dictated by company policy, use of other closed-source third-party C++ libraries, and/or customer support obligations where the customer has an old compiler environment.
2. Boost is huge and complex. This turns off many projects/companies from looking at it and using it, even though there are many benefits.
3. If a project is already using Boost (and many are) then a unit test framework that also uses it is not a problem. But given the buildability issues with Boost and its size and complexity, some projects/companies would be reluctant to be forced to use it just because the unit test framework requires it.

Possible future work

FRUCTOSE deliberately does not provide any machinery for producing HTML test summaries, reports, or ways of running multiple test suites. Yet anything but the smallest projects will probably want this facility as part of the overnight build and test regime. One way to do this would be for FRUCTOSE to provide scripts, say in perl or python, that used some convention for naming and grouping the test harnesses so they can be run and the results organised into groups.

FRUCTOSE may provide some facility in the future to augment the command line options with additional options that a developer needs for their particular needs. For example, a harness that uses a database may wish to pass in the database parameters (database name, machine name, username, password). At the moment a FRUCTOSE harness would have to find some other way to receive these parameters.

FRUCTOSE does not provide any means to assess code coverage in its tests. There are separate tools to do this. For example, a FRUCTOSE test harness could be run with PureCoverage to assess how much code was exercised. Such tools do not often provide output or reports in way that lend themselves to brief reporting via such things as an overnight build and test run. One possible enhancement of FRUCTOSE would be to develop scripts that work with tools like PureCoverage to give a brief summary of which functions were called and which were not, and what the percentage code coverage was of the functions that were called.

Conclusion

It is hoped that FRUCTOSE provides enough unit test machinery to enable projects to develop unit tests that can be used both in overnight regression tests and to help TDD. I welcome any feedback on the usefulness(or otherwise) of FRUCTOSE in other projects.

FRUCTOSE's simple implementation and cutdown approach mean that providing anything more complex will probably be permanently outside of the project's scope. However, this does not mean that FRUCTOSE is not open to changes. Any suggestions on how FRUCTOSE can be further simplified without reducing functionality will be gratefully received. It may be downloaded from sourceforge [FRUCTOSE]. ■

References

[Unit] Unit---- at https://sourceforge.net/projects/unitmm

[csUnit] csUnit at https://sourceforge.net/projects/csunit

[Symbian] Symbian OS C++ Unit Testing Framework at https://sourceforge.net/projects/symbianosunit

[Rapido] Rapido test at https://sourceforge.net/projects/rapidotest

[MockObject] Mock Objects for C++ at https://sourceforge.net/projects/mockpp

[UnitTest++] UnitTest++ at https://sourceforge.net/projects/unittest-cpp

[QuickTest] QuickTest at https://sourceforge.net/projects/quicktest

[CppTest] CppTest at https://sourceforge.net/projects/cpptest

[TCLAP] TCLAP Templatised C++ Command Line Parser Library at http://tclap.sourceforge.net.

[Knuth] *The Art of Computer Programming*, Volume 2, by Professor Don Knuth.

[Pressman] *Software Engineering; a practioners approach*, by Roger Pressman (4th Edition). Section 16.6.2.

[FRUCTOSE] FRUCTOSE at https://sourceforge.net/projects/fructose.

[CUTE] Overload 75, October 2006, CUTE.

[Vandevoorde and Josuttis] *C++ Templates, The Complete Guide*, section 16.3, by Vandevoorde and Josuttis.

[FSF] The Free Software Foundation, http://www.fsf.org

Letter to the Editor

Alexander Nasonov writes more on singleton.

Hi Alan,

I posted a reference to the Overload 76 to the C++ forum of Russian Software Developer Network (http://www.rsdn.ru) and I have had a few replies regarding my article. Since the article has not been reviewed, these comments can be considered as postmortem peer reviews.

An anonymous reader replied (translated from Russian):

> Why did you say nothing about the most popular Meyers singleton where the LOCAL static variable is being used? You showed 2 most awful realizations, which nobody uses (I hope!) and it makes little sense to speak about them. I do not see any advantages of your realization over the Meyers singleton.

I agree that I should have discussed the Meyers singleton in the article. Single-threaded implementation is very simple and it manages dependencies automatically:

```
Singleton& instance()
{
  static Singleton inst;
  return inst;
}
```

But it is not so simple in a multithreaded program. Although the C++ standard does not define the term ‘thread’, my experience says that, if **`main()`** is already entered, a thread safety is often guaranteed for static objects at namespace scope but not for local static variables. As a result, the **`inst`** object may be initialized more than once if two threads call the **`instance()`** simultaneously.

A naive modification of the code above:

```
mutex mtx;
Singleton& intance()
{
  lock l(mtx); // lock mtx now, unlock in dtor
  static Singleton inst;
  return inst;
}
```

would break a dependency tracking because now the **`instance()`** can’t be called before the **`mtx`** is initialized. On POSIX platforms, it can be fixed by using **`PTHREAD_MUTEX_INITIALIZER`** to initialize the **`mtx`** object at static phase:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
Singleton& intance()
{
  lock l(mtx); // lock mtx now, unlock in dtor
  static Singleton inst;
  return inst;
}
```

Unfortunately, a static initializer for **`CRITICAL_SECTION`** is not available on Windows. One has to write a wrapper but it’s not a trivial task.

This code can be optimized further using double locking technique but you should be very careful. Refer to [DCLocking] from the references section of the article.

Another anonymous reader wrote a test program to check a thread safety of the Meyers singleton:

```
template <typename T>
class singleton_meyers
{
public:
  static T& instance()
  {
    static T obj;
    std::cout << "instance finished\n" ;;
    return obj;
  }
};
struct sleep_in_ctor
{
  sleep_in_ctor()
  {
    std::cout << "ctor started\n";
  ;    sleep(5);
    std::cout << "ctor finished\n";
  }
};
void stupid_func()
{
  std::cout << "stupid func\n";
  singleton_meyers<sleep_in_ctor> tmp;
  tmp.instance();
}
int main()
{
  boost::thread thrd1(stupid_func);
  boost::thread thrd2(stupid_func);
  thrd1.join();
  thrd2.join();
}
```

Before doing thread-safety analysis, I’d like to note that this program uses I/O (**`cout`**) and process scheduling calls (**`sleep`**). In general, these calls should be avoided in tests that try to detect race conditions. The output is differ depending on the version of gcc it is compiled with.

Compiled with gcc 4.1	Compiled with gcc 3.4
`stupid func`	`stupid func`
`  ctor started`	`  ctor started`
`  stupid func`	`  stupid func`
`  <<< 5 sec pause >>>`	`  ctor started`
`  ctor finished`	`    <<< 5 sec pause >>>`
`  instance finished`	`  ctor finished`
`  instance finished`	`  instance finished`
	`  ctor finished`
	`  instance finished`

As you see, gcc 4.1 correctly initializes the instance while gcc 3.4 incorrectly initializes two instances. Starting from version 4.0, gcc supports one-time construction API: http://www.codesourcery.com/cxx-abi/abi.html#once-ctor.

It is on by default but you can disable it with **`-fno-threadsafe-statics`** option. Note that it’s not a portable extension and you shouldn’t rely on it, though it’s worth trying it out to detect recursive initialization (refer to 6.7 [stmt.dcl], bullet 4: If control re-enters the declaration (recursively) while the object is being initialized, the behaviour is undefined).

To summarize the reviews, I missed one important case which can be used in multithreaded programs if code is written properly, though it may be slower than a solution presented in the article because synchronization is required.

Alexander Nasonov (alexander.nasonov@gmail.com)

http://nasonov.blogspot.com, http://alnsn.livejournal.com