OVERLOAD 73

June 2006
ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Contributing editor

Mark Radford
mark@twonline.co.uk

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddeaus Froggley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Tim Penhey
tim@penhey.net

Advertising enquiries

ads@accu.org

Cover art

Pete Goodliffe
pete@cthree.org

Design

Pete Goodliffe

Copy deadlines

All articles intended for publication in Overload 74 should be submitted to the editor by 1st July 2006 and for Overload 75 by 1st September 2006.

The ACCU

The 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 the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

The More Things Change, the More They Stay the Same

One of the great things about a community like ACCU is that people care. We care about contributing and we care about problems that affect us all. Maybe together we can do something about them!

Your magazine still needs you!

In my last editorial I found it necessary to remind you that Overload is your magazine and it requires your input. As you can see, a number of you have seized the opportunity and submitted the articles that appear in this issue.

I'm pleased with the response, and I hope that both the authors and the readers will be pleased with the results too. But don't let things rest there – if you have an article, or an idea for an article, then the advisors and I are still here and ready to help shepherd it through to print.

Overload now has a new look and some new authors – the very least you can do is write and tell us what you think!

What makes an Overload article?

At the conference I was asked a number of times about what might be suitable material for Overload. This isn't as easy to answer as it might seem: Overload isn't about a particular language, a particular platform or a particular technology. But it is about helping the readership make better, more informed choices about the way they develop software.

There are many types of article that can do this: but whether it is about a technology, a technique, a design pattern, a development process, good (or bad) practice, or something else, an article for Overload should offer the reader a new way to approach the problems of developing software.

In this issue we have a pair of articles that address aspects of the Singleton design pattern (or, as some might prefer it, "anti-pattern"). Steve Love tells a pattern story about the introduction of Singletons into a project and considers what happens next. Meanwhile, Adam Petersen asks Mr Singleton about his walk-on part in the State pattern – and discovers a notable lack of enthusiasm for this particular role.

We also cater for those that are interested in how clever stuff works. There are a couple of articles that address this: Anthony Williams explains the finer points of implementing menus in CSS and Chris Gibson examines how "boost::bind" works.

An Overload article need not be a definitive statement of the "one true way". In this issue you will find two opinion pieces questioning the received wisdom that comments are good. One of these is Thomas Guest's consideration of "TODO": is "TODO" a part of your programming toolbox? And does it work as well as you'd like to think? Our other opinion piece was inspired by Peter Sommerlad's session at the conference – Peter's agenda was to challenge some myths about software development. He certainly succeeded in getting Mark Easterbrook thinking. (There is more to follow – Allan Kelly was also inspired by this talk to submit an article that we don't have space for this time.)

Overload presumes that its readers are willing to think about what is written in these pages, and not to accept it blindly. These articles are part of a long standing tradition of printing material that may provoke discussion. We've had plenty of debate in these pages over the years, and long may this continue – it sometimes leads in surprising and fruitful directions! If these articles make you think about the way you develop software then I hope you can contribute something to the next issue that continues the discussion.

I should, perhaps, clarify that the pair-programming experience report by Rachel Davies is not the one I alluded to in my last editorial. This is a shame, because it would have been nice to see them together – and to compare conclusions. Unfortunately, like all of us, the potential author of the missing article is busy – and in addition to a day job has recently been working on other material I feel is more important than producing this Overload article. (If I said any more then I'd risk identifying someone who already knows who they are.)

Maybe you can think of a phrase to cover all all these articles. I can't – I just know that these are the articles my advisors and I agree they want to see in the magazine. I hope we are making the right choices. If you think we are not, then please don't suffer in silence (or resign your membership in disgust) – write and let us know.

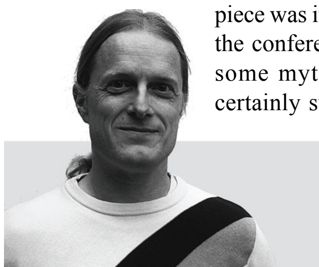
Digital Rights Management

I know of no safe depository of the ultimate powers of the society but the people themselves; and if we think them not enlightened enough to exercise their control with wholesome discretion, the remedy is not to take it from them, but to inform their discretion by education. This is the true corrective of abuses of constitutional power.

Thomas Jefferson

Judging by a recent thread on ACCU general it would seem that the attempts to restrict the rights of consumers by distributors of copyright material has not appeared on the radar of many of us. This is a shame because we should be far more aware of "digital rights management" issues than the general public.

Even in its current form this isn't a new issue: Richard Stallman has been attempting to highlight it for at least a decade as shown by his entertaining "The Right to Read" article from 1997 [Stallman97]. In that time all that has happened is that the threat to our rights has become clearer and the need to defend them more urgent.



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. Homepage: http://www.octopull.demon.co.uk and he can be contacted at overload@accu.org

I think my first brush with these problems was in 1998 or ’99 – the company I was working for bought a PDF of the ANSI C++ standard. ANSI chose to distribute this with a flag set that prevented text selection for copy and paste – which is a considerable pain when one wants to quote a section of the standard in discussion. Now, I’m not suggesting that they were acting illegally – clearly I could, and did, exercise my “fair use” right by retyping the text I needed. But it is notable that after a bit of lobbying by the C++ community they revised their policy and supplied a copy enabled version both to those that had complained and to new purchasers. What I found annoying was the lack of respect shown in making it less convenient to exercise my rights than it need be.

There are similar irritations with the music industry – I am (and always have been) entitled to play my LPs on a wide range of equipment or to lend them to friends. With electronic distribution of music increasing effort is being made to tie playback to a particular piece of kit (and we all know how quickly that gets replaced). There is nothing illegal about such restrictions – you have the legal right to agree to such terms when you make a purchase.

Of course, even before electronic distribution many of us stretched the “wide range of equipment” beyond legal limits by introducing an audio-cassette into the playback mechanism to make our music available in our cars (the dire quality of commercial audio cassettes was also a factor here). And I suspect that occasionally, people acquired tapes of material for which they didn’t buy a copy of their own. The industry wasn’t happy about this – but instead of chasing such petty infringements they lobbied government to impose a tax on blank audio-cassettes.

Ever since books have existed they have been lent to friends and associates, and the same has been true of music and video. The right to do these things is under threat.

The move to digital technologies changed things – the marginal cost of reproducing things is negligible. In the early days of the computer industry little though was given to controlling the distribution of software (it was considered) – but, as the level of investment in software and the benefits from using it became more apparent, many regimes of permissions were experimented with. Some of take a “like a book” approach, some tie use of the software to a particular piece of hardware (either a computer, or a “dongle”), and some actively promote sharing. While the more restrictive approaches are popular with vendors, they tend to inconvenience their customers.

With the move to digital technologies for distributing books, music and movies, the distributors have been attracted by availability of these more restrictive approaches – which prevent us doing what we’ve always been able, and are entitled, to do with traditional technologies. While it is true that the same facilities that allow us to do this could also be abused to break the law this isn’t new (although it has become cheaper and more reliable). More importantly, the current technological solutions are not adequate to prevent serious abuse and serve mainly to irritate customers. (For a particularly eggrarious example of this see [Russinovich].)

Given this background I was interested to see a Sydney Morning Herald article [SMH] describing proposed changes to Australian law making legal “format shifting” from private collections onto playback devices and “time shifting” (recording programs for later playback). I particularly enjoyed a quote from the Attorney-General Philip Ruddock: Everyday consumers shouldn’t be treated like copyright pirates, copyright pirates should not be treated like everyday consumers. Amen!

However, this action is very much against current trends. At the far side of the world a different story is unfolding: “The British Library - “The world’s knowledge” DRM’d and for a price” [Groklaw]. It seems that libraries are also introducing restrictions on the use of borrowed material that limit its use far more than would apply under copyright law (which guarantees various forms of “fair use”).

We hear far too much about the rights of the suppliers of digital content – we need to ensure that the rights of the consumer are also respected!

A short note on OpenDocument

I’ve made a few mentions of the standard format for office application documents produced by Oasis in the past few editorials. You may be interested to hear that OpenDocument has now been approved as an ISO standard – there are a few formalities to go through, but the decision making process has completed and all that remains is producing the final document. You can read more about this on Andy Updegrove’s blog [Updegrove].

References

[Stallman97] “The Right to Read”, Richard Stallman, 1997, http://www.gnu.org/philosophy/right-to-read.html

[Russinovich] “Sony’s Rootkit: First 4 Internet Responds”, Mark Russinovich, http://www.sysinternals.com/blog/2005/11/sonys-rootkit-first-4-internet.html

[SMH] “Cutting crime as easy as MP3”, http://www.smh.com.au/news/technology/cutting-crime-as-easy-as-mp3/2006/05/13/1146940771335.html

[Groklaw] “The British Library - “The world's knowledge” DRM’d and for a price”, http://www.groklaw.net/article.php?story=20060317044847293

[Updegrove] “OpenDocument Approved by ISO/IEC Members”, http://www.consortiuminfo.org/standardsblog/article.php?story=20060503080915835372.pdf

Pair Programming Explained

Rachel Davies explains how to implement pair programming and why it can be an effective practice for programmers.

It was 1999 when I first heard about Extreme Programming [Beck] and its counter-intuitive practices of Test-Driven Development and Pair Programming. At the time, I was embroiled in a telecomms project, managing a large team of C++ programmers.

On that project, some developers were new to C++ and object-oriented programming but had valuable knowledge about our existing systems written in CORAL. Whereas other programmers were C++ experts who had mostly worked on strategic proof-of-concept projects so were unused to writing high performance production code. We developed our software incrementally, using UML to capture designs, and held code inspections of programmers' work, but we struggled with effective knowledge sharing.

Problems that arose were:

- Programmers who reported they were "90% done" on the same tasks for several weeks.
- Lack of dependency management – long compile times.
- Programmers using new language features purely to try them out without considering the impact on performance.
- Programmers who reported they were finished when they had done little or no unit testing.
- Programmers who kept code checked out for long periods.

Since then I have become an XP practitioner and, in hindsight, I realise that these problems could have been avoided if we had applied the practice of pair programming.

Pair programming environment

In 2000, I decided to follow Martin Fowler's advice: If you can't change your organization, change your organization! I went for a job interview as a developer at an XP company. The team had just moved into new offices and they were being fitted out to create a customised work environment to support the XP team. Large screens and convex desks were the order of the day.

To start pair programming, as Ward Cunningham said, you need to "Arrange the furniture" [Cunningham]. Pair programming is not one person passively watching the other type. Each programmer needs to play an active role in determining the design, implementation and tests. The pair works together on the task in hand while passing keyboard control back and forth between them, writing code and tests as they go. Many offices have desks shaped for a single programmer to sit within an inset. It is impossible to pair program at these desks without squashing your partner. Instead, you need straight-sided tables or better convex curved desks. You also need wall space for whiteboards and pin boards, to create what Kent Beck now calls an Informative Workspace.

Try to get workstations with the highest specification you can (you only need to budget for half the number of workstations as for solo programming). A large screen area is essential so that both programmers can read the code without crowding each other. Dual screens have become a popular way to implement this. A cordless keyboard and mouse can also make it smoother to pass them between partners. Some teams even choose to set up workstations with two keyboards and two mice (although I haven't tried this myself). To make it easy for a pair to use any free workstation, each workstation ought be configured with the same set of development tools and a standard set of preferences in the team's chosen IDE.

A pair programming episode

Most XP teams make a pair programming session part of their interview process. My first pair programming experience was during my interview. This was quite nerve-racking, as I was new to the Java programming language and web development. However, from a candidate's point of view I appreciated a chance to see their code and get an impression of how they worked before accepting the job.

Here is how a typical pair programming episode works. The person at the keyboard takes the role of driver and directly implements the solution working at the tactical level; their partner takes the role of navigator thinking at a strategic level about next steps and potential pitfalls. Pairs switch frequently between these roles (sometimes passing the keyboard over every few minutes).

Each pair programming episode typically lasts a couple of hours and ends with code being integrated and checked into version control. In addition to switching roles during a pair programming episode, programmers in an XP team typically change partners more than once per day. Most teams I have worked with use use a simple pair rotation rule – each day the pair splits, leaving one person on the task for continuity and releasing the other to pair with another programmer on a different task.

Some teams have experimented with swapping even more frequently, using a synchronised time interval of as little as 45 minutes. Arlo Belshee reports that when swapping partners every 90 minutes: We were able to fully ramp up new hires who had never programmed in C++ before – to the level that they could write template metaprograms and train the next round of new hires – in one month [Belshee].

My experience of joining two XP teams with an existing code base is that, initially each pair programming episode is like key-hole surgery – you can work effectively on small clumps of related classes to add a new feature but you don't really get a good picture of the architecture by pair programming alone. My advice is to be aware that pair programming is not a substitute for training, new programmers will need time to learn a new technology and will probably benefit from some time on their own to explore the code base when they join the team.

Rachel Davies is an independent agile coach based in the UK, a frequent presenter at industry conferences and a director of the Agile Alliance. She has been working in the software industry for nearly 20 years. She can be reached via her website, www.agilexp.com, or at rachel@agilexp.com.

staying focused on the task at hand you are more likely to have **working code** at the end of the day rather than going home with **coding problems** still lodged in your head

The rationale

Pair programming sounds like it will be hard to justify to your team and management. If you only count number of number features coded per developer then it might appear that two programmers will produce more features when working separately than when working as a pair. But this is a simplistic view of productivity that ignores the contribution of defect rates and rework. Maximizing your output without attention to quality eventually slows a development team down. We are all prone to making mistakes and bugs cost additional time to fix – usually interrupting on-going development tasks. Code also needs to be consistently organised and well factored to allow subsequent features to be added.

Fixing defects found in a QA phase or, worse, after deployment is costly. Leaving defects in the code to be picked up downstream also slows down a software development team, as programmers try to build new features on over-complicated and possibly malfunctioning code. Pair programming can prevent this slowdown by keeping code clean and well organised at all times. Clutter and bugs are simply not allowed to accumulate.

It is well recorded that code inspections are cost-effective ways to reduce defect rates [Fagan]. However, industry adoption of formal inspections still remains low. Most teams that I meet see code reviews as a “nice to have” practice and drop them when the team is working under time pressure. Pair programming provides an alternative way to implement design and code reviews. The ongoing peer review that happens while pair programming prevents poorly designed code from ever being checked in. When programmers write code in pairs, they keep each other on task and on process.

Information rapidly diffuses through a pair programming team as pairs switch and learn from each other. Ken Auer, one of the early adopters of XP, calls this knowledge transfer effect the “pair-vine.” Although code inspections also support knowledge sharing, these meetings typically happen weeks apart, so they have an inbuilt latency. Also discussing a technique in a code review is not the same as actually trying it out. When you are pair programming, you learn good design by implementing new concepts on real code with your pair on hand to give guidance.

The evidence

There are some studies that provide empirical evidence in support of pair programming.

- A 1975 study of “two-person programming teams” reported a 127% gain in productivity and an error rate that was three orders of magnitude less than normal for the organization under study. [Crosstalk]
- In 1992, Larry Constantine wrote about a team that was set up by author P.J. Plauger, who developed code that was nearly 100% bug free. [Constantine]
- Dr. Laurie Williams, author of *Pair Programming Illuminated*, has carried out the main research on this topic. You can find much of her research on the pair programming website [Pair]. Williams found in her studies [Cockburn] that in exchange for a 15% increase in development time, pair programming improved design quality; reduced defects by an average of 15%; and was reported as more enjoyable by programmers at statistically significant levels. A less well-known finding was that pair programmers generate output that is more concise – implementing the same functionality in fewer lines of code.

Personally, I am a bit skeptical about these studies – the conditions that they were carried out under are unlikely to be representative of many software projects in industry. Over the last six years, I have witnessed pair programming working effectively on real projects in a variety of domains that have successfully delivered software and it has become my preferred way to write code.

Programmer’s concerns

Code quality is important but most developers who have not tried pair programming are concerned about how pair programming affects their creative process and how working with another programmer affects the pace of development. There’s no denying it feels different. Your partner frequently challenges your proposed approach so you have to stay alert and completely engaged in the task at hand. It’s hard work but nice to have someone to share your highs and lows with.

Programmers may worry that pair programming will expose gaps in their coding skills. This actually cuts both ways – it’s highly probable that you know tricks that your partner does not. My experience is that once programmers have tried pair programming this concern quickly evaporates as such gaps are quickly filled when you are learning from your programming partners. Working with a partner means there is always someone there to help you get unstuck. By staying focused on the task at hand you are more likely to have working code at the end of the day rather than going home with coding problems still lodged in your head.

As knowledge spreads throughout the team, fewer bottlenecks are caused by holidays or absence because most code has been worked on by more than one programmer. You stop having to plan around key people on the critical path. Because pair programming brings an increase in knowledge sharing, it can release you from specialist work that no one else knows how to do. No longer will you be the only person who can work on a specific component and feel under pressure whenever you need to ask for time off work. You will be free to move onto work on other parts of the system and broaden out your skill set.

Finally, curbing programmers with bad habits who create work for the rest of the team, by checking in their broken code for their fellow team mates to fix, can be a big relief if you were one of those dutifully cleaning up in their wake.

Getting to the next level

It does take time to become skilled at pair programming. The technique hinges on improving your communication skills; it is vital that you can articulate your ideas and also listen to your pair. I recently spent several months working in a dispersed team where each developer worked from home and we pair programmed using NetMeeting to share desktop control. Despite my initial doubts, this actually turned out to be a reasonably effective way to work. Although it did require even more attention to clear verbal explanation of potential problems that you see in your partner's approach and alternative strategies that you would like to be considered.

ideas are woven together rather than clashing over anticipated end solutions

When solo programming, we invest more effort in building a complete mental model of the solution and then transcribing it into code. However, we don't always think every scenario through thoroughly and so bugs can creep in. Pair programming slows down our thinking process so that we take a closer look at each small step in building the solution – working in the "here and now" using the simplest strategy that works for this moment. Each programmer reveals their experience a little bit at a time and their ideas are woven together rather than clashing over anticipated end solutions. You have probably heard about the XP saying "Do the simplest thing that can possibly work" – this heuristic is the key to resolving differences of opinion that arise in a pair programming team. This is used as a guide to evaluate micro-decisions at the strategic level within a pair programming episode and does not literally mean write naïve solutions.

You may anticipate that pairing inexperienced programmers with expert programmers may be frustrating because the expert will be forced to slow down to work at the pace of the novice. It is essential that the expert does take time to explain their thinking to their partner rather than dominating the keyboard or barking keyboard escape sequences for them to type. The novice is more likely to get up to speed as an effective team member when their partner takes care to explain their thinking and the expert is likely to be surprised that slowing down often results in a more rounded solution. The peer programming experience of two experts working together is usually very rewarding, as both partners will be generating good ideas and counter-challenges at a similar rate. Novice-novice programming pairs are typically more talkative as the pair are working hard to learn together – often making false starts when in new terrain but usually making more progress than when left to work on their own. It may be wise to track pairing combinations within your team to avoid pair cliques forming and to ensure that each team member gets the opportunity to pair with everyone.

Solo programming

Are there times when solo programming is appropriate? Yes. Sometimes code quality is not so important – as when developing a temporary solution or prototype. Splitting a pair makes sense when exploring alternative technical solutions or bug-busting where different root causes need to be eliminated. There will be days when you have an odd number of programmers. Most programmers find being locked into pair programming all the time a little claustrophobic and appreciate some time to work on their own now and then. In this case, try to make sure that any production code written without a pair is peer reviewed before being checked in.

After reading this article it might appear that pair programming will have a positive effect on every software development team. But as the mighty Frederick Brooks said, there is no "silver bullet." [Brooks]. Some scenarios are listed below where pair programming is unlikely to deliver the full set of benefits:

- **Flexi-time and Telecommuting**. For pair programming to get off the ground successfully, you do need to have programmers available for work in the office for core hours. This may be an issue if your company explicitly offers employees the option to work from home and programmers on the team do not wish to give this benefit up.
- **Distributed Teams**. Where a project team is split across different company sites, it may not make sense to attempt pair programming across these boundaries, as face-to-face communication is much richer. An alternative is to only pair program within the sub-teams at each site and then implement code reviews of selected work products across the boundaries. This strategy does mean that you lose some knowledge sharing across sites but rotating programmers between sites is another way to do this.
- **Diverse Teams**. A classic situation in which pair programming does not work well is when you have heterogeneous system architecture and the team is made up of a small number of specialists, each proficient in only one technology used in the system. A common case of this situation is the specialised database programmer. It probably does not make sense for an Oracle programmer to pair with a C++ programmer because the skills required to work proficiently in their respective technologies are not easy to absorb via pair programming alone and separate training support may be required.
- **Small Teams**. Very small teams (one to three programmers) will probably find that they do not benefit from formal pair programming as they already have a high level of interaction and would find working with the same person for long periods too claustrophobic.

Summary

Pair programming is a technique that improves both code quality and your programming skills. It helps by improving programmer discipline and providing a mechanism for knowledge sharing. The best way to evaluate pair programming is to use this practice on a small, low-risk project. To check whether the practice is delivering results, review your bug rates – you should find fewer critical bugs reported on code developed in pairs. It will also be important to discuss with the team what their experience has been with pair programming and whether they feel that they are learning more by using the technique. ■

Splitting a pair makes sense when exploring alternative technical solutions or bug-busting where different root causes need to be eliminated.

References

[Beck] *Extreme Programming Explained* – K. & C. Andres ISBN 0-321-27865-8

[Belshee] *Promiscuous Pairing and Beginner's Mind: Embrace Inexperience* by Arlo Belshee, Agile2005 http://agile2005.org/XR4.pdf

[Brooks] *The Mythical Man Month* by Brooks ISBN 0-201-83595-9

[Cockburn] *The Costs and Benefits of Pair Programming* XP2000 conference, A. Cockburn & L. Williams http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF

[Constantine] "The Benefits of Visibility" in *The Peopleware Papers* – Larry Constantine ISBN 0-13-060123-3

[Crosstalk] Crosstalk March 2003 available on-line at http://www.stsc.hill.af.mil/crosstalk/2003/03/jensen.html

[Cunningham] "EPISODES: A Pattern Language of Competitive Development" – Ward Cunningham, *Pattern Languages of Program Design 2*. ISBN 0-201-89527-7

[Fagan] "Advances in software inspections to reduce errors in program development" - Michael Fagan *IBM Systems Journal* 1976.

[Pair] http://www.pairprogramming.com/

Comments Considered Evil

We are taught that adding comments to code is a good thing and adds value. In practice this value is seldom, if ever, realised. Mark Easterbrook makes the case for a better way.

At the ACCU 2006 conference, the question was asked "Is the total value of all the comments in all of the code out there less than or more than zero?" [1] If the answer is less than zero, which judging from participants' reactions and war stories is quite possibly the case, then the ability to put comments in source code has had a negative impact on the programming community, and is therefore A Bad Thing™.

Learning from the experts

Why do programmers put comments in code? To answer this we need to look at how the language is taught, so I took a number of books by the most respected authors in the industry that are aimed at beginners or start from the basics, and looked to see how these experts recommend beginners use comments, and they all had something along the lines of:

```
  i++;/* Increment i */
```

So, if all these highly recommended books are consistent on what a code comment should look like, it must be right, right? It is a brave programmer to tell these esteemed writers that the code samples they use in their books are poor examples. Taking one particular example at random, on page 24 of *The C++ Programming Language, 3rd Ed*, by Bjarne Stroustrup we find the following example of how to write a C++ function:

```
void some_function() // function that doesn't return a
                     // value
  {
    double d = 2.2   // initialise floating-point
                     // number
    int i = 7;       // initialise integer
    d = d+i;         // assign sum to d
    i = d*i;         // assign product to i
  }
```

Thus Bjarne's book is teaching single character variables are good and comments should tell the code reader what the code is doing. Much much later on page 138 Bjarne does describe better use of comments, but by this time most beginners have given up reading and are out there hacking code with looming deadlines. Even those who do read more of the book have what they learned on page 24 reinforced with "[indent style]…I have my preferences, and this book reflects them. The same applies to comments". That's right, just copy Bjarne's "preferable" commenting style from his book. (Note: Before you email me, I have read to the end of the section, my point is that not everyone bothers to read all the way to the end.)

Examples of this Good Practice™ of commenting can be found everywhere. One example from some telecoms software had, in revision 1.1 in CVS:

```
  channel++;   //increment channel
```

The good thing about this line was that both the code and the comment agreed (experience shows this is quite often not the case). The bad thing about this code is that they are both wrong, the problem being that the original coder didn't understand the difference between channel and timeslot [2] and chose the wrong variable name. Fortunately, when he (or someone who came along later) realised the mistake he used a snazzy re-factoring tool that avoids the problems with global "find and replace" and understands the code enough to only change the relevant variable name and not other occurrences of the string or the same name in another scope. The code now reads:

```
  timeslot++;      //increment channel
```

You might ask why the re-factoring programmer didn't change the comment to match and it was because he wasn't looking at that particular line, but at this one:

```
  uint32_t channel; // I think this is probably
                    // timeslot, not channel.
```

The first person to come along realised the poor choice of variable name and "fixed" it by adding a comment. The second decided it would be better to change the variable name. Obviously, the second programmer believes in saying it in the code and disregarding comments, because the changed line now reads:

```
  uint32_t timeslot;  // I think this is probably
                      // timeslot, not channel.
```

It passed programmer testing, because the test suite only tested the code and disregarded the comments. It then passed code review with only an observation because it had already been tested and checked in, and the company policy was not to modify the source file after testing unless a potential customer visible fault is found. The rationale being that modifying the comment modifies the file, forcing a rebuild and requiring a re-test [3].

"If the code and comments disagree, both are probably wrong".
Norm Schryer

Better comments

The problem with all these code gurus is that they are code gurus, and comments are not code. If they stuck to teaching code and left teaching comments to comment gurus we would get much better comments. Or would we? I trawled various sources for good practice in commenting and found the following pieces of advice:

- Say why and not how.
- Meaningful.
- Say it in the code when you can, and in the comments when you must.
- Only use comments where the code is non-obvious and/or non-portable.
- Don't write comments that repeat the code.

Mark Easterbrook is a software developer specialising in technical domains. In his day job he works with embedded systems, high performance/reliability/availability systems, operating systems, and legacy code, sometimes all at the same time. The rest of his time is split between motorcycles, genealogy, linux, and food & drink, but never at the same time. Mark can be contacted at mark@easterbrook.co.uk

- Do write illuminating comments that explain approach and rationale.

This is not a complete list, but a representative sample. The last two are from *C++ Coding Standards* [4] where Herb and Andrei dedicate only 4 lines of their book to comments. This sends the message that the authors do not regard good comments as a subject to spend much time on.

I have stolen a good example of someone following the above guidelines from Roger Orr [5]:

```
// Clear collection before we start
myCollection.empty();
```

The code by itself would hopefully be spotted at code review because the reader would have to think about why there is a statement with no side-effects, and make sure it fails the review. Add in the comment following the above guidelines and the "why?" is obvious so the reader doesn't have to ponder the code and moves on to more important things. Would this code be better with or without the comment? Would the coding error be more obvious without the comment?

What is a comment?

If you ask a group of programmers what they understand by "comment" you will almost certainly start an argument. This will not be quite as heated or frequent as "vi or emacs", but neither will it be a minor disagreement. The descriptions I have heard vary from "line comments" – the "block headers are documentation not comments" school of thought, through to "everything the compiler ignores", bringing in commented-out code and all `#if 0` blocks into the definition. I suspect that more than a few out there will include comments that aren't even in the source code file (CVS check-in comments anyone? RCS maybe?).

And then there is self-documenting code: If comments are a form of documentation, and a programmer writes self-documenting code, surely the code is then a "comment". Just because the compiler reads it does not mean that it is checked for correctness - the compiler does not care if a variable is called **`channel`**, **`timeslot`**, or **`foo`** – variable names are for the human reader, not the machine, and thus are a type of commenting!

Mandating useless comments

Almost every code generating place I have worked has some form of mandating comments in code, either by having a coding standard which is selectively checked at code review (and comments are always selected, sometimes the only thing selected), or just going straight to a tick-box list somewhere that has "adequate comments" as a line item. Adequate by quantity is easy to check. Adequate by quality is not so easy. Most reviews choose the path of least resistance and go for a quantity check.

the misguided individuals who write coding standards and code review checklists don't intend to mandate useless comments

I know that the misguided individuals who write coding standards and code review checklists don't intend to mandate useless comments: the desire is to promote good practice. However, programmers are only human (for the sake of this argument I'm assuming this is the case, you may know some exceptions) and will take the easiest path to conformance – which is to follow what is actually mandated rather than what is intended to be mandated.

The most useless mandated comments in the wild are block headers. Some development teams even put an empty block header into an easily accessible file somewhere (they call it a skeleton header or some other similar excuse) so it can easily be copied and pasted into every source file. Mostly these headers remain as function separators making sure each function or procedure is kept away from the adjacent one by lots of screen real estate so you cannot ever see more than one on the screen at once.

Sometimes the cut-and-pasters go that bit further and fill out the fields in the block header. Unfortunately these are typically:

- Name of function – Don't copy and paste from the function name a few lines down, type it in to increase the chance of error.
- Purpose of function – Take the function name and add spaces between the words (and maybe put back the vowels and other letters if they are missing).
- Inputs – Another name for the argument list.
- Outputs – Another name for the return type.
- Algorithm – Pseudo code, which means the code written in the language you would prefer to use rather than the language that the compiler understands.
- Comments or Remarks – Your chance to demonstrate your wit and charm, or lack of it, to your fellow coders.

Some development teams even discourage the filling-in of block headers by surrounding them with a box of asterisks or similar characters. Any changes make the right hand side of the box irregular, so it is better to leave it in its original regular form.

It seems to me that the unwritten rule is: If you have a function that is similar to the one you want to write, or not similar at all, just copy both the header and the code. Because someone has already filled in the header for you it saves some time and allows you to concentrate on the code.

Furthermore, if the coding standards mandate block headers with an asterisk in column 2, it is possible to grep for just the headers and produce the documentation from the code files. Counting the duplicate functions then allows an estimate of how much cut-and-paste programming has taken place. But look at how much documentation there is!

Block Headers: Please, just say no.

Automatic comment generation

Integrated development tools now take a lot of the effort out of adding comments to code, saving the programmer valuable minutes of coding time. Picking a popular IDE and using the tool to create a new Dialog Box obtains comments such as:

```
return TRUE;  // return TRUE unless you set the
              // focus to a control
              // EXCEPTION: OCX Property Pages
              // should return FALSE
```

Now I thought the idea of automatic code generation was that the code was generated automatically for the programmer, or am I being particularly naïve about this? Surely the system, either at code generation time or later, can work out if the focus is set to a control or not, and whether this is an OCX property page or not. This particular example was taken from production code – did the original programmer actually look at the code and decide that **`TRUE`** was correct but was too lazy to modify the comment appropriately, or not?

The nice thing about good code generators is that quite often the default behaviour is what is required, so there is no need to even go and look at the automatically generated code. Thus the presence of

```
// TODO: Add extra validation here
CDialog::OnOK();
```

throughout the code leaves the maintenance programmer with an uneasy feeling that perhaps the original coder did not finish the job.

If the above example had not auto-generated comments but pseudo code instead:

```
Perform custom validation (if any) then call:
CDialog::OnOK();
```

The coder would have been forced to look at the generated text and turn it into the correct code. The compiler would have insisted on it!

In another universe

Just imagine a world in which the compiler reads the whole of the source file instead of skipping the bits the coder cannot be bothered to code in the language. This is a fancy way of saying "No comments".

To imagine what this nirvana might be like, let's look at something else that the designers of programming languages decided wisely not to add to the language: Version Control. The first version control system [6], RCS, foolishly added the history of the code to the code file itself, but everyone saw this was bad (all right, almost everyone) and caused files to grow beyond the file system limit and lo, CVS was born where the code is the payload and the version information is the metadata. CVS makes putting the version information in the file so difficult nobody bothers to do it. [7]

change the approach of programmers so that newly written code is appropriately annotated

The nice things about having code and version control system loosely coupled are:

- If you don't like your version control system you can change it without changing your programming language.
- If you don't like your programming language you can change it without changing your version control system.
- You can view just the code, or you can view just the revision information, or a bit of one and a bit of the other. You choose to view just what is appropriate for the current task.
- There is a big market for version control systems, and add-ons for version control systems, and books and websites on version control systems.

In our brave new (comment-free) world, the more creative individuals will soon work on adding what is missing…

Someone will write a pre-processor that allows annotations and code to be mixed, but strips them out before the compiler or interpreter sees them. This will be a great development because the way annotations are added is language independent. The alternative, the Balkanisation of annotations, would be awful because you could end up with a crazy situation where lines starting with a certain symbol mean an annotation in an interpreted language but, say, a pre-processor instruction in a compiled language.

An obvious tool to support annotations is the programmer's editor, whether it be stand-alone such as Unix vi, or within a feature-rich IDE such as Microsoft Visual Studio. The potential in the integrated development tools is large: hover the cursor over the variable and a balloon pops up with any comment assigned to the variable. There could be options to open up further information such as seeing the definition or any comments assigned to the containing code (class if it is a member variable, function if it is a local variable, etc.). Annotations could be applied at any level, even right up to whole application or project. Even the simpler tools can use code colouring or other hints to indicate the presence of annotations, perhaps showing them in a separate but linked scrolling window that can be re-sized or hidden when not required.

Annotations could be a valuable part of pair programming with one of the pair taking the role of coder and the other the annotator. A positive feedback loop would occur: when the coder reads the annotations he obtains a qualitative measure of how much the annotator has understood so that both the code and the annotations can be improved. All this happens in a short timeframe when the original implementation knowledge is still fresh.

The majority of code in the wild comes under the category of legacy code. The constraints of maintaining such code are the rule of minimum change and no gratuitous modifications. Considerable time and effort is spent analysing and understanding the code in order to add necessary enhancements or fix problems without violating those constraints. As annotations don't modify the deliverable code or force automatic rebuilds (e.g. by make or similar tools), they can be added as the code is examined with little risk. Just imagine the understanding that would be gained and then immediately thrown away if those new annotations were hard-wired to the code and couldn't be modified!

Conclusion

The direction that programming comments, a.k.a. code annotation, has taken has helped keep the quality and maintainability of code low. The way that different programming languages implement comments is inconsistent and in some cases conflicting. The authors of books, industry gurus, and the quality [8] procedures in most companies have all perpetuated the status quo with the result that innovation, which could lift the usefulness of code annotation to higher levels, has been stifled.

Is it time to change the mindset of the programming community so that the total value of all the comments in all of the code moves into positive territory and then continues this upwards trend? This requires those with influence to lead the change:

- Authors of books and articles should stop putting bad examples of comments in code snippets. The topic being described should appear in the text, not in the code.

  These auto-generated comments are the result of lazy implementation of the tools (just write a comment instead of working out what the code should be) followed by lazy users (the computer generated it so it must be right). Definitely a case of two wrongs make an even worse wrong.
- Coding standards should stop mandating comments and move towards making code readable and understandable. A comment should reflect a failure to make the intent of the code clear, and therefore be a fall-back position and not a primary goal.
- Leaders of code reviews should look for justification for comments, and reject cases where the comments are only added because the coder is too lazy to write the code properly in the first place.
- Block headers should be actively discouraged. The total value of block headers is so far into negative territory that elimination of all block headers would be a significant improvement.
- Programming tools should be enhanced to support annotations to code that improve the readability, instead of comments that pollute it and make it more difficult and confusing to understand.

It may not be possible to go back and make existing commenting consistent and useful across multiple languages and the millions of files of existing source code, but it is possible to change the approach of programmers so that newly written code is appropriately annotated. All it needs is the realisation that we have been wrong all these years and that there is another way. ■

Notes and references

1. By Russel Winder in Peter Sommerlad's session called "Only the Code tells the Truth".
2. Timeslots typically start at zero and channels often start at one. In 2Mbit/s ISDN timeslots 0 and 16 are reserved so that channel 1 is in timeslot 1 and channel 16 is in timeslot 17. As long as you only test the first 16 channels you don't have to bother about the subtle difference.
3. This raises the perennial question: When do you do code reviews, before check-in and testing, or after? Both have their pros and cons, and neither stands out as being significantly better.
4. *C++ Coding Standards*. Herb Sutter and Andrei Alexandrescu. 2005.
5. Producing Better Bugs. Roger Orr. ACCU conference 2006.
6. I'm taking a little artistic licence here with the history of source code revision systems, but you get the picture.
7. It also makes it so hard to get it out again, nobody can work out how to, and thus exports and imports into a fresh CVS archive occur. This gets rid of the pollution in the code, but it gets rid of all the history as well.
8. As in ISO9000 type quality, as opposed to goodness.

How Do Those Funky Placeholders Work?

The current C++ standard function binders are notoriously difficult to use. Chris Gibson exposes the secret the Boost alternative, which is so much better it seems like magic.

Whenever I learn a new coding technique and try it out at work, it's very interesting to see how rapidly it's adopted throughout the department. One of the fastest to propagate so far is the boost **bind** library. So much easier to use than the standard library's **bind1st** or **bind2nd** adapters and much more flexible. Once you've got the hang of those funny looking placeholders, the **bind** library is elegant and simple to use, and the initial effort is well rewarded. Your code is clearer to read, easier to understand, quicker to review, quicker to test, it's less likely to contain bugs and is probably more efficient.

In my experience the average developer, by which of course I don't mean you, and certainly not me, when using the **bind** library for the first time will probably use either too many, too few, or the wrong placeholders; and will put at least one of them in the wrong place. Any of which will result in pages of compiler errors. The average developer will then decide that it's all too difficult, templates are the work of the devil, they haven't got time to read the manual, and they just need to get it working. So they hand code a solution instead and decide to do it properly before review or release, which in all probability never happens.

So I can't decide whether the **bind** library is intuitive or not. Once you've got the hang of it, it is, but doesn't intuitive mean you don't need to get the hang of it? Can it be true that the more you use the **bind** library the more intuitive it gets?

Once a developer has 'seen the light' he will invariably ask 'So how do those funky placeholders work?' In an ideal world you wouldn't concern yourself with the implementation details of a library, you would just use it. However, an understanding of what is going on behind the scenes is useful for several reasons. Firstly, with the error messages the average compiler gives you when you make a typo, it's a big help to have some understanding of what's going on. Secondly, studying the designs developed by experts is always instructive and can help to improve your own designs. Finally, most developers are naturally curious and, once comfortable with the placeholder syntax, often want to know how they work.

This article intends to demonstrate how the **bind** library's placeholders work by constructing a very basic **bind** library. It will be modest, very modest, but enough I hope to satisfy your curiosity. The examples are written for clarity above all else, other issues such as efficiency, const correctness, volatility, etc., are not considered. Note also that the function return types are not considered and assumed to be void.

Chris Gibson is a Chartered Electrical Engineer. He began his software career in 1994 when he developed Power Management Systems for the Jubilee Line extension. He went on to develop software for Industrial Automation and Fire & Gas Systems. He is currently developing software for Radiotherapy applications in C++. Normally he struggles to write so much as a postcard. Chris can be contacted at chris@eurous.co.uk

A quick recap

Suppose you have a collection of widgets and you want to perform an action on each of them; you would simply write something like this:

```
void DoSomething(Widget& widget)
{
  ...
}
for_each(widgets.begin(), widgets.end(),
  DoSomething);
```

Suppose, however, that you would like to pass each widget to an existing function such as:

```
void DoSomething(Gadget& gadget, Gizmo& gizmo,
   Widget& widget)
{
  ...
}
```

The **for_each** algorithm takes a unary function as its third parameter, which it calls once for each iterator in its range, passing the dereferenced iterator as the single argument to the unary function (see below). Our target function has three parameters, which causes two problems:

- The unary function and the target function have a different number of parameters.
- The dereferenced iterator could map to any one of the target function's parameters.

```
template<typename InIt, typename UnaryFn>
Fn for_each(InIt first, InIt last, UnaryFn fn)
{
  // perform function for each element
  for(; first != last; ++first)
      fn(*first);
  return fn;
}
```

Resolving the different number of parameters is done by applying the Fundamental Theory of Software Engineering and adding a level of indirection between the algorithm and the target function. The indirection takes the form of an object, let's call it a binder. The binder's responsibility is to adapt the interface the algorithm calls to the interface of the target function. The binder can be called by **for_each** because it has a unary **operator()**, and can call the target function because it's supplied with a pointer to it and the two extra arguments at construction.

Specifying which of **DoSomething**'s three parameters should be the widget is where placeholders come in. Using the **bind** library we can simply do this:

```
for_each(widgets.begin(), widgets.end(),
     bind(&DoSomething, aGadget, aGizmo, _1));
```

The binder sits between the algorithm that we want to use and the function that we want to call

The `bind` function is passed four arguments: the address of the target function `DoSomething`, a `Gadget` object, a `Gizmo` object and a placeholder `_1`. Notice that the order of the second, third and fourth arguments to `bind` is the same as the order of the arguments to the target function. The placeholder indicates the position of the argument passed to the unary function by `for_each`. If the signature of `DoSomething` had expected the widget as the second parameter we would have written:

```
for_each(widgets.begin(), widgets.end(),
       bind(&DoSomething, aGadget, _1, aGizmo));
```

The `_1` placeholder specifies the mapping of `for_each`'s single argument to the target signature. For algorithms that expect a unary function you must use a single placeholder, `_1`. For algorithms that expect a binary function you must use two placeholders, `_1` and `_2`. For algorithms that expect a ternary function you must use three placeholders, `_1`, `_2` and `_3` and so on.

For example, `adjacent_find` is an algorithm that takes a binary function and finds the first instance of two adjacent elements which match a given criteria. If we have a comparison function which takes a third parameter we can bind to it as below:

```
bool IsEqualWidget(Widget& w1, Widget& w2,
   int tolerance);
adjacent_find(widgets.begin(), widgets.end(),
   bind(IsEqualWidget, _1, _2, 42)); // tolerance = 42
```

For more information and examples see the boost website.

Constructing a basic bind library

I'm going to construct a basic bind library capable of calling the three parameter version of `DoSomething` shown earlier, once for each of the Widgets in our collection.

The `bind` function is one of a family of overloaded factory function templates. The responsibility of each of the overloads is to create a specific type of binder. In our example bind is a function template with four parameters: a target function with three parameters and three parameters to pass to it (see below).

```
template<typename F, typename A1, typename A2,
   typename A3>
binder<F, list3<A1, A2, A3> > bind(F f, A1 a1, A2 a2,
   A3 a3)
{
  typedef typename list3<A1, A2, A3> list_type;
  list_type  list(a1, a2, a3);
  return binder<F, list_type>(f, list);
}
```

The return type, the binder that `bind` creates, is of unspecified type in TR1. In this example it is a class template with two parameters. The first parameter is the target function and the second is an argument list.

```
template<typename F, typename List>
class binder
{
public:
    binder(F f, List  list):f_(f), list_(list)
    {}
    …
private:
    F f_;
    List  list_;
};
```

In this example, the binder is instantiated with an instantiation of `list3` as a template argument. `list3` is one of a family of class templates. Each of `list3`'s member variables represents a value passed to the original `bind` function. We have converted the three arguments to `bind` into an object with three member variables (see below). Notice the ellipsis, we'll be adding more responsibilities to `list3` and its siblings shortly.

```
template<typename A1, typename A2, typename A3>
class list3
{
public:
  list3(A1 a1, A2 a2, A3 a3): a1_(a1), a2_(a2),
   a3_(a3) {}
  …
private:
  A1 a1_;
  A2 a2_;
  A3 a3_;
};
```

Now that we have created our binder we need to turn our attention to what it does. The binder sits between the algorithm that we want to use and the function that we want to call. The `for_each` algorithm expects its third parameter to be a unary function so we must add a templated unary `operator()` to the binder.

```
template<typename F, typename List>
class binder
{
public:
    …    // As previously defined
  template<typename A1>
  void operator()(A1 a1)
  {
    list1<A1> list(a1); // Explanation coming
    list_(f_, list);
  }
};
```

Placeholders then aren't really anything special. They are simply a type that allows overload resolution to occur.

The **`operator()`** creates a **`list1`** (see below) object from its parameter. The **`list1`** object is very similar to **`list3`** but with just one parameter. We now have two lists of parameters, the list object for the argument passed by the algorithm and the **`list_`** member for the arguments passed into **`bind`**.

```
template<typename A1>
class list1
{
public:
  list1(A1 a1): a1_(a1){}
  …
private:
  A1 a1_;
};
```

We have to detect which of the arguments passed to the **`bind`** function are placeholders and substitute them for the arguments passed by the algorithm. For arguments that are not placeholders we must simply pass the argument to the function to be called. To achieve this we are going to rely on function overloading.

For **`list3`** we have to add the **`list3::operator()`** which is called from the unary **`binder::operator()`** (see below). In our first example the three member variables of **`list3`** are:

- **`a1_`** - a **`Gadget`**, which should be passed as the first argument of **`f`**.
- **`a2_`** - a **`Gizmo`**, which should be passed as the second argument of **`f`**.
- **`a3_`** - a placeholder (**`_1`**) indicating that the first parameter in the **`list1`** (a widget) should be passed as the third argument of **`f`**.

```
template<typename A1, typename A2, typename A3>
class list3
{
public:
  …    // As previously defined

  template<typename F, typename List1>
  void operator()(F f, List1 list1)
  {
    f(list1[a1_], list1[a2_], list1[a3_]);
    // Explanation coming
  }
};
```

At the moment there is no way to access the data in **`list1`**, we need to add two **`operator[]`**s (see below). Notice that one **`operator[]`** takes a **`placeholder<1>`** and returns the **`a1_ member`** (the widget passed by the **`for_each`** algorithm). The second **`operator[]`** is templated to take any other type and just return the value it is passed.

```
template<typename A1>
class list1
{
  …    // As previously defined

  A1 operator[](placeholder<1>) const { return a1_; }

  template<typename T>
  T operator[](T v) const { return v; }

  …    // As previously defined
};
```

When **`list1[a1_]`** is evaluated it will resolve to the templated **`operator[]`** in **`list1`** and simply return the gadget that was passed in. The same process will occur for **`list1[a2_]`** with a **`Gizmo`**. However, when **`list1[a3_]`** is evaluated the non-templated **`operator[] (placeholder1)`** will be selected and a widget will be returned. Hence **`DoSomething`** will be called with the correct arguments.

Placeholders then aren't really anything special. They are simply a type that allows overload resolution to occur. We need to have a different type for each of the number of placeholders we decide to scale our library to support. A trivial class template instantiated on an integer value will suffice, for example.

```
template<int I>
class placeholder{};
placeholder<1> _1;
placeholder<2> _2;
placeholder<3> _3;
```

The example we set out to achieve will now compile and execute correctly – try it. Next try calling a function with different arguments and the widget in a different position. Finally try calling a function with a different number of arguments, it will fail to compile. Can you work out what we need to add to our library? Hint: look in \boost\bind.hpp! ■

Implementing drop-down menus in pure CSS (no JavaScript)

Implementing drop-down menus to aid website navigation is usually thought to require lots of JavaScript. This article shows how to do it using just CSS.

A client of mine wanted his website to have drop-down menus, so I had a look round at the best way of doing this. I imagined that it would require JavaScript, but it turns out that it is possible in pure CSS, at least for fully compliant browsers. This article attempts to explain how the CSS works, and builds up the menu step by step.

Why CSS?

Why CSS, and not JavaScript? JavaScript is often disabled by users, as a security measure, and the necessary code for drop-down menus can be quite involved. Also, a pure JavaScript menu is not available for browsers that don't support it, such as text-only browsers. CSS-based menus are always available, even with JavaScript disabled — browsers that don't handle it will just render a list. With this technique, adding a menu to a page is as easy as creating an unordered list of links, with nested lists for the sub-menus, and including the appropriate style-sheet.

Other CSS-based menus — what's new here?

Tarquin's tutorial on CSS menus shows how to do menus, where the main menu is stacked vertically, and the sub-menus open out to the side, and links to CrazyTB's CSS menu page, which shows a horizontal top-level menu, with drop-downs, but which doesn't work with IE, and imposes a fixed width on the menu entries. This article describes a technique for doing drop-down menus in CSS, with a horizontal top-level menu, and variable-width menu entries – in other words, I've managed to overcome many of the limitations of the implementations I've seen.

Menu structure

The menus are just represented by nested **`UL`** lists. Each **`LI`** is a menu entry, and nested lists result in sub-menus. The top level **`UL`** must have the class attribute of **`navmenu`**, and everything follows from there. The menu items are just normal **`A`** links, or **`SPAN`**s where they are not links. For this example, I'm going to use the menu in Listing 1.

A menu bar should be horizontal

The first step is to take off all the normal list adornments, so we know what the indents are, and don't get bullet marks:

```
.navmenu,
.navmenu ul,
.navmenu li
{
    padding: 0px;
    margin: 0px;
}
.navmenu li
{
    list-style-type: none;
}
```

We make the top level menu horizontal by floating the menu items, but if we do that then the rest of the page now displays underneath them, so we need to follow the menu with a **`clear`** style. In compliant browsers, we can do this using the **`.navmenu + *`** selector, but IE doesn't support this,

```
<ul class="navmenu">
  <li><a href="/tl1">Top Level Link</a></li>
  <li><a href="/tl2">SubMenus</a><ul>
  <li><a href="/tl2/item1">Item 1</a></li>
  <li><a href="/tl2/item2">Item 2</a></li>
  <li><a href="/tl2/item3">with submenus</a>
    <ul>
      <li><a href="/tl2/item3/one">One</a></li>
      <li><a href="/tl2/item3/two">Two</a></li>
      <li><a href="/tl2/item3/three">Three</a></li>
    </ul>
  </li>
  <li><a href="/tl2/item4">Item 4</a></li>
</ul></li>
      <li><a href="/tl3">3rd entry</a><ul>
      <li><span>Submenu no link</span><ul>
      <li><a href="/tl3/item1/one">One</a></li>
      <li><a href="/tl3/item1/two">Two</a></li>
  <li><a href="/tl3/item1/three">Three</a></li>
           </ul></li>
       <li><a href="/tl3/item2">Item 2</a></li>
        </ul></li>
      <li><a href="/tl4">Fourth</a><ul>
     <li><a href="/tl4/item1">has items</a></li>
  <li><a href="/tl4/i2">but no submenus</a></li>
      </ul></li>
   <li><a href="/tl5">top level 5</a><ul>
        <li><a href="/tl5/i1">item 1</a></li>
          <li><a href="/tl5/i2">item 2</a></li>
        </ul></li>
      <li><a href="/tl6">entry 6</a><ul>
          <li><a href="/tl6/i1">foo</a></li>
          <li><a href="/tl6/i2">bar</a></li>
        </ul></li>
      <li><a href="/tl7">Final entry</a><ul>
          <li><a href="/tl7/i1">aaa</a></li>
          <li><a href="/tl7/i2">bbb</a></li>
          <li><a href="/tl7/i3">ccc</a></li>
        </ul></li>
    </ul>
```

Listing 1

Anthony is the Managing Director of Just Software Solutions Ltd. He has been programming professionally for over 10 years, having programmed as a hobby for a good many before that. He is a strong believer in the benefits of Test Driven Development, Refactoring, and being able to see the sea from his office. He can be contacted at anthony@justsoftwaresolutions.co.uk

we are using **float** to make the LI elements **stack sideways**, rather than their default of stacking vertically, in order to get a **horizontal menu**

Float and clear

In normal usage, the `float` property of the CSS removes an element from the normal flow of the document, and "floats" it over to either the left or right edge. Such "floating" elements stack sideways, so if two elements both have a `float: left` style, then the second one will be to the right of the first. Subsequent content is flowed to the side of the floating elements.

Here, we are using `float` to make the `LI` elements stack sideways, rather than their default of stacking vertically, in order to get a horizontal menu.

The `clear` property is used with `float`, to ensure that following content appears below any floating elements. The value can be `left` or `right`, to indicate that this element (and any following ones) should be below all prior floating elements on the specified side, or both, to indicate that it should be below floating elements from either side.

Here, `clear` is used to ensure that subsequent content comes below the menu bar.

so we need a tag with a class attribute of `.endmenu` following our menu (an empty `DIV` is good for that):

```
.navmenu li
{
  float: left;
}
.navmenu + *
{
  clear: left;
}
.endmenu
{
  clear: left;
}
```

Sub-menus only display on demand

Next off is to hide the sub-menus, and show them when the mouse moves over the parent. This requires a browser that supports :hover for `LI` tags. For IE, we can then simulate this with DHTML Behaviours, as suggested by Tarquin:

```
.navmenu ul
{
  display: none;
}
.navmenu li:hover > ul
{
  display: block;
}
.navmenu ul.parent_hover
{
  display: block;
}
```

To add the DHTML Behaviours for IE, we can add the following to the HTML:

```
<!--[if gte IE 5]><![if lt IE 7]>
<style type="text/css">
.navmenu li
{
    behavior: url( ie_menus.htc );
}
</style>
<![endif]><![endif]-->
```

The DHTML behaviour file (`ie_menus.htc`) is then quite straightforward – we simply set the `hover` class on the current element, and `parent_hover` on all the child elements when the mouse moves over the appropriate element, and then remove these classes when the mouse moves off again:

```
<attach event="onmouseover" handler="mouseover" />
<attach event="onmouseout" handler="mouseout" />
<script type="text/javascript">
function mouseover()
{
  element.className += ' hover';
  for( var x = 0; x!=element.childNodes.length; ++x )
  {
    if(element.childNodes[x].nodeType==1)
    {
      element.childNodes[x].className
         += ' parent_hover';
    }
  }
}

function mouseout()
{
  element.className =
     element.className.replace(/ ?hover$/,'');
  for( var x = 0; x!=element.childNodes.length; ++x )
  {
    if(element.childNodes[x].nodeType==1)
    {
      element.childNodes[x].className =
         element.childNodes[x].className.replace(
         / ?parent_hover$/,'');
    }
  }
}
</script>
```

Sub-menu layout should be nice and clean

This works OK, but as the menus expand, the content of the rest of the page gets shunted down to make room. Ideally, we'd like the menus to show on top of the rest of the page. We can do this by giving the sub-menus highlight the **entire box** when you hover the mouse, so you can see you're over a **menu item** that's actually a **link**

a position style of `absolute`, but if we do just that then they're hard to see over the top of the text below, and the menus don't work quite right in IE. Therefore, we will add a border, and background. Of course, if we set a background colour, we ought to set the foreground colour too. Links have a different default colour to other text, so we need to set that separately. We therefore need to add the following styles:

```
.navmenu ul
{
  position: absolute;
}
.navmenu li
{
  border: 1px solid #3366cc;
  color: #000033;
  background-color: #6699FF;
}
.navmenu a
{
  color: #000033;
}
```

In Mozilla, the drop-down menus are also horizontal, whereas in IE, they're vertical. We can fix that by making only the top-level menu entries `float`, rather than all of them:

```
.navmenu > li
{
  float: left;
}
```

However, this doesn't work in IE – to get a nice horizontal top-level menu, we need to `float` the menu entries, and specify a fixed width for them, so we need to update our IE-specific block to do this:

```
<!--[if gte IE 5]><![if lt IE 7]>
<style type="text/css">
.navmenu li
{
    float: left;
    width: 8em;
}
</style>
<![endif]><![endif]-->
```

Of course, you can vary the width as required.

Links should occupy the full box

I like the links to take up the full width of the box, so you don't have to click on the text. It's therefore nice to highlight the entire box when you hover the mouse, so you can see you're over a menu item that's actually a link.

In this case, because it's links we're referring to, IE is quite happy with :hover, so we can use the same styling for all browsers:

```
.navmenu a
{
  display:block;
  width: 100%;
  text-decoration: none;
}
.navmenu a:hover
{
    background-color: #f8f8fb;
}
```

Sub-sub menus should pop out to the side

We now have a new problem – if one of the drop-down menus has a sub-menu, then we can't get to the following menu items, as the sub-menu comes down on top of them. We therefore need to adjust the positioning of the sub-menu; we'll move it almost to the right-hand edge of the parent menu item. It is important that we don't move it completely off, as then users would have to move the mouse cursor off the parent to go to the sub menu, and so the menu would close. We accomplish this with the `left` style. If we just use that, then the menus also start a line down, so we should use `top` to ensure they start level. Finally, we need to make the `LI` elements have `relative` positioning, since we made the sub-menus have `absolute` positioning above. This resets the base position for each sub-menu as relative to its parent menu item, rather than relative to the whole page.

```
.navmenu li
{
  position: relative;
}
.navmenu ul ul
{
  top: 0;
  left: 99%;
}
```

The problem now is that the drop-down menus display underneath existing menus.

We could fix this with `z-index`, but IE doesn't handle that. Instead, and here's the fun bit, if we set `padding-left` to `1px`, then the menu items are shown on top, but the top specified above doesn't work – it aligns the sub-menu with the top of the parent sub-menu.

Instead, we can use `margin-top` with a negative offset, to shift the block up. I've chosen `-1.2em` as the offset, since this is the default line-height, so the menu should pop out level with the parent entry.

```
.navmenu li
{
  padding-left: 1px;
}
```

```
.navmenu ul ul
{
  /* top: 0; --- remove this*/
  margin-top: -1.2em;
  left: 99%;
}
```

This `left` padding shifts the drop-down menus right a fraction. Combined with the border, this makes the top-level sub-menus appear 2 pixels to the right of their parent, which is a bit untidy. The fix for this issue is to add a negative margin to the sub menus, which has the effect of shifting them back left, to compensate:

```
.navmenu ul
{
  margin-left: -2px;
}
```

Menu items that have sub-menus, but are not themselves links, still don't work quite right, since the text does not form a block for CSS layout purposes, and the sub-menu therefore is displayed too high up. This is why we put the non-link menu items in `SPAN` tags – the fix for this is to make these `SPAN` tags into block elements:

```
.navmenu span
{
  display: block;
}
```

Exposed background

If the top-level menu doesn't cover the full width of the browser window, then the background for the `BODY` element will show through in the exposed parts. To deal with this, we can set the width of the outer `UL` element to `100%`, and give it a background:

```
.navmenu
{
    width: 100%;
    background-color: #6699FF;
}
```

This works nicely in Opera and IE, but not in Firefox, which makes a change. If we also make it `float` to the left, then it works in all three browsers.

```
.navmenu
{
  float: left;
}
```

Spacing around text

Having the menu entries just display as minimal-sized blocks can mean that the text is quite close to the edges, and looks a bit crammed in. We can alleviate this by adding some padding to the `LI` elements:

```
.navmenu li
{
    padding: 2px;
}
```

Of course, this means that the previous `padding-left` entry should be removed, and the negative `margin-left` entry for the sub-menus needs adjusting. We also now need a `margin-top` entry for the first layer of sub-menus, to align the top of the sub-menu with the bottom of the parent item:

```
.navmenu li
{
  /* padding-left: 1px; --- remove this */
}
.navmenu ul
{
  margin-left: -3px; /* was -2px */
  margin-top: 2px;
}
```

Another consequence of this padding is that the highlighted links now have an extra border around them, as only the link text area highlights, not the whole `LI`. We can fix that by changing the background colour for `LI` elements that we're hovering over as well. This has the side effect that the menu entries leading to the currently displayed sub-menu are also highlighted, which works as quite a nice visual aid. We'll leave the highlighting in place for hovered links, too, so that browsers that can't handle hovering on `LI` elements still show some highlighting. We have to do two versions here – one for IE, and one not, as we're relying on the DHTML behaviours for the hover detection in IE.

```
.navmenu li:hover
{
  background-color: #f8f8fb;
}
.navmenu li.hover
{
  background-color: #f8f8fb;
}
```

Browser support

The key feature that this technique relies on is the ability to use the :hover modifier on arbitrary elements, and not just links. Older versions of browsers do not support this, but newer versions do. If this feature is not supported, just the top-level menu is shown. It is therefore important to ensure that the pages are not just accessible via the menu – I would recommend that each top-level menu entry is a link to a page with real links to the items on the appropriate sub-menu.

Internet Explorer doesn't support this usage of :hover, but it can be simulated with a small bit of JavaScript, as shown. If the user's security settings mean the JavaScript is not run, then IE will just display the top-level menu.

This technique is known to work in Opera 7.2 and 8.5, IE6 (with JavaScript), Mozilla Firefox 1.5, and Konqueror 3.4.3. It is known not to work in Opera 5, and Netscape 4.7. Of course, it doesn't work in text-only browsers such as Lynx, either – users of such browsers will see the menu just as a nested list.

Hiding things from old browsers

Old browsers such as Netscape Navigator 4.7 understand CSS, but get the rendering all wrong. Therefore, we need to mask our style-sheet from such browsers, so they just render the menu as a list. Of course, you could make a set of styles that worked with such browsers to make the menu render more nicely, if you wish. That's more effort than I'm willing to spend at the moment, so I'm just going to wrap the style-sheet in `@media all{}`, which will force such old browsers to completely ignore it.

There are numerous other techniques which can be used to adjust the style-sheet for specific older browsers, but they're beyond the scope of this article.

Conclusion

So, there you have it, drop-down menus in pure CSS, with a tiny bit of JavaScript for Internet Explorer. Supported in a wide range of browsers, with graceful degradation where it is not supported, this technique allows you to add menus to your website, without delving into JavaScript.

Final style-sheet

The final style sheet is shown on the next page, as Listing 2. The IE-specific styling, which needs to go directly into the `HEAD` part of the HTML, is shown in Listing 3. The IE-specific DHTML behaviour code from `ie_menus.htc` is in Listing 4. ■

```
 @media all{
.navmenu,
.navmenu ul,
.navmenu li
{
  padding: 0px;
  margin: 0px;
}
.navmenu > li
{
  float: left;
}
.navmenu li
{
  list-style-type: none;
  border: 1px solid #3366cc;
  color: #000000;
  background-color: #6699FF;
  padding: 2px;
}
.navmenu ul
{
  display: none;
  position: absolute;
  margin-left: -3px;
  margin-top: 2px;
}
.navmenu li:hover > ul
{
  display: block;
}
.navmenu ul.parent_hover
{
  display: block;
}
.navmenu a
{
  display: block;
  width: 100%;
  text-decoration: none;
}
.navmenu li:hover
{
  background-color: #f8f8fb;
}
.navmenu li.hover,
.navmenu a:hover
{
  background-color: #f8f8fb;
}
.navmenu ul ul
{
  margin-top: -1.2em;
  left: 99%;
}
.navmenu span
{
  display: block;
}
.navmenu
{
  float: left;
  width: 100%;
  background-color: #6699FF;
}
.endmenu
{
  clear: left;
}
}
```

Listing 2

```
<!--[if gte IE 5]><![if lt IE 7]>
<style type="text/css">
.navmenu li
{
  float: left;
  width: 8em;
   behavior: url( ie_menus.htc );
}
</style>
<![endif]><![endif]-->
```

Listing 3

```
<attach event="onmouseover" handler="mouseover" />
<attach event="onmouseout" handler="mouseout" />
<script type="text/javascript">
function mouseover()
{
  element.className += ' hover';
  for( var x = 0; x!=element.childNodes.length; ++x )
  {
    if(element.childNodes[x].nodeType==1)
    {
      element.childNodes[x].className +=
          ' parent_hover';
    }
  }
}

function mouseout()
{
  element.className =
     element.className.replace(/ ?hover$/,'');
  for( var x = 0; x!=element.childNodes.length; ++x )
  {
    if(element.childNodes[x].nodeType==1)
    {
      element.childNodes[x].className =
         element.childNodes[x].className.replace(
         / ?parent_hover$/,'');
    }
  }
}
</script>
```

Listing 4

The Rise and Fall of Singleton Threaded

Steve Love explores how “Singletons” in design can seem a good idea at the time, why they are generally a mistake, and what to do if you have them.

For a while now, it has become increasingly accepted that the use of Singleton is a Bad Thing™. In general terms we hear the case against Singletons: hard to test; don’t play nicely with multiple threads; hide dependencies; etc. By and large, I think, programmers are becoming less enamoured with Singletons, and this, I also think, is a Good Thing™. But it’s not as widespread as I thought. Singleton is still pretty popular: it is, after all, simple to implement. Programmers use it with the very best of intentions, even when those intentions are to find an alternative to real global data.

Instead of trotting out the same arguments that others have used, however, I want to explore a little, and implement a small system in terms of Singleton and see where it leads. This (I hope) will also allow some exploration into how to return from Singleton-ness to a more ordered world.

The code examples here are in C#, mainly because it’s compact and simple. There shouldn’t be anything that can’t easily be translated to Java, C++, or others.

The descent

For the sake of argument, and because it makes the code simple, the example application is a data transformation program; it takes some input, and writes a different output.

A simple **Main** function might contain:

```
// Create a new reader using the first command-line
// argument as a file name.
   Reader reader = new Reader( args[ 0 ] );

// Create a new writer which performs the
// transformations
   Writer writer = new Writer();

// Read all lines from the reader. null gets returned
// when EOF is reached. Write resulting text to the
// output.
   string text;
   while( ( text = reader.Read() ) != null )
   {
     writer.Write( text );
   }
```

Now this may or may not be grand design, but it suffices to provide a framework within which to think about the design. The **Reader** object may be doing some kind of validation of input, and the **Writer** collating and transforming the data.

At some point in the development of this application, the requirement arises that a run report is produced, indicating errors such as opening the file, and a record of the number of lines of text read: a logging facility for the **Reader.**

Having added such a facility, it doesn’t take much imagination to see that the **Writer** component could also use the logging facility, and that making it easily accessible from anywhere in the program would ease development.

The fall

A key intent of the Singleton pattern is that it provides access to some service or data globally within an application, i.e. that it is always easily available [1]. Making a Singleton logging class in C# is straightforward:

```
public sealed class Logging
{
  public static readonly Logging Instance =
     new Logging( "log.txt" );

  public void Write( string text )
  {
    using( StreamWriter log = new StreamWriter(
       filename, true ) )
    {
      log.WriteLine( text );
    }
  }
   private Logging( string fileName )
  {
    this.fileName = fileName;
  }

  private string fileName;
}
```

This allows the **Writer** class to perform the following trick in its **Write** method (for the sake of the example, **Writer** sends its output to a file) :

```
public void Write( string text )
{
  try
  {
    // some processing on text for transforming it
    // ...
    output.WriteLine( text );
  }
  catch
  {
    Logging.Instance.Write(
      "Failed to write a line" );
  }
}
```

A similar **Read** method in the **Reader** is also achievable, and does much the same thing, with no changes to the client code.

Steve Love is an OO developer who can’t make his mind up between C++ and C#. This coupled with a penchant for English beer and Italian food conspire to reduce the time he gets to take part in extreme sports. He can be contacted at steve@essennell.co.uk

calls into the logging component are now so **tightly synchronised** that any **benefit** of using multiple threads **is lost**

A nasty knock

Our little neat design works as intended, and all is fine with the world, until one day, the requirements ask for multiple `Writer`s operating concurrently to speed up processing. As well as writing to a back-end store, results need to be transferred to some other location as a backup, in a different format. Instead of actually exposing multiple `Writer` objects in the main application, the existing `Writer` implementation spawns a thread for each target, writing its results. The client code has no need to know of the change. Each separate thread looks like the original `Write` and the application appears to run no slower than the original, which is handy because that was also a requirement ☺. All is fine with the world…

Until someone inspects the log files and sees gibberish!

After perhaps only a little head-scratching, it becomes obvious that the problem is concurrent access to the logging facility. In C# we have a handy short-hand for obtaining a lock on some object to provide a critical section, so our logging `Write` method now becomes:

```
public void Write( string text )
{
  lock( this )
  {
    using( StreamWriter log =
       new StreamWriter( filename, true ) )
    {
      log.WriteLine( text );
    }
  }
}
```

The `lock` statement effectively allows only a single thread of execution into the protected code at a time. A second thread cannot enter that section of code until the first thread completes it and releases the lock.

So the problem of the gibberish log has been fixed. However…

The hard landing

All calls into the logging component are now so tightly synchronised that any benefit of using multiple threads is lost. Whilst thread one is logging its output, thread two *must* wait until it has finished before continuing and vice-versa. This code probably performs *worse* than the single-threaded alternative due to the overhead of obtaining the lock in each thread.

One solution to this would be to make the logger entirely asynchronous. A call to `Logging.Write` would merely add the message to a queue (which itself would have to be locked each time, but the latency would be greatly reduced). The logging component would have a thread of its own looking for available messages to actually write to the output. This seems neat and tidy, but suffers mainly from the fact that the logging facility is now many times more complicated, introducing a large potential for error, and additionally is much more complex than the code that uses it!

…and kicked when you're down

As work is under way trying to "fix" the logging problem, some users report that the transformed output sometimes contains errors. Initial investigation suggests a bug in the `Writer` component, but reproducing the error is hard, making the location of the error hard to pin down. It seems the only course of action is to run the entire application in a debugging environment, and keep trying this with different input until the bug is found. Not only is this time consuming, it's *boring* for the poor soul selected to do it – and that makes it error-prone, too.

What would be really nice is to be able to test the `Writer` component in splendid isolation, away from the rest of the application, under laboratory conditions, and just have it perform its tricks on small sets of test input.

To achieve this separation entirely, the `Writer` component needs to be stripped of all its dependencies. For true lab conditions the `Writer` object needs to exist in a test harness that might look like this:

```
public void TestInputsToOutputs()
  {
    Writer writer = new Writer();
    writer.Write( "test text" );

    // How do we test the output here?
    // Perhaps open the output file, and check its
    // contents?
  }
```

It has now become clear that `Writer` has some hidden dependency which we need to break: `Writer` sends its output to file each time its `Write` method is called – a hidden dependency! In our ideal test, we'd like to be able to tell the `Writer` where to send its output, so our test can become (for example):

```
  public void InputsToOutputs()
  {
    TextWriter output =
       new StringWriter( new StringBuilder() );
    Writer writer = new Writer( output );

    string testText = "test text";
    writer.Write( testText );

    Assert.AreEqual( testText, output.ToString() );
  }
```

This associates a `Writer` object with the output target as a `TextWriter` – an abstract base class for both `StringWriter` used in the test, and a `StreamWriter` which can be used to write to a file.

The key point here is that `Writer` is *parameterised* with the needed dependency. Obviously the dependency itself has not been removed altogether – that would be pointless because it removes needed functionality. Instead the dependency is moved from a concrete class to an interface, and a reference to the object is *passed in*. Amongst other

What if we could **inject** the Singleton object to allow us to **control** its **dependency** on a real live log file?

things, this is called *Dependency Inversion*, achieved using a technique called Parameterise From Above [3] [4].

A healing injection

Having unhidden the dependency on an actual file location, the **`Writer`** object still has the dependency on the logger. Unless we need to test the logged output, this dependency isn't a great problem. It isn't ideal, though.

Firstly the tests are slowed down by the need to write to the logging component. In the test environment, there is no need for logged output, so this is merely a waste. The second issue is one of style: the object under test *cannot* be tested in true isolation.

Resolving the dependency is harder for this case than for the output location, because the dependency is upon a Singleton. We could re-factor the code to pass in the **`Logging`** component, having first modified the **`Logging`** class to be an implementation of some interface. Given that we have no tests to ensure that this refactoring is successful (that's what we're trying to achieve!), this approach is somewhat risky.

A different approach is to inject the dependency into the object. The *Parameterise From Above* Pattern used just now is an example of injecting the dependency into the user-object. What if we could inject the Singleton object to allow us to control its dependency on a real live log file? This would allow us to keep the existing code – which reduces the risk of breaking the application – but does muddy the waters a little by adding indirection to the **`Logging`** component. In its very simplest form, the new logging component might look like (please remember this is for brevity!):

```
public interface LoggingInterface
{
  void Write( string text );
}

public sealed class DefaultLogging :
   LoggingInterface
{
  public void Write( string text )
  {
    // The actual writing to log file goes here
  }
}

public sealed class Logging
{
  public static LoggingInterface Instance =
     new DefaultLogging();
}
```

The idea here is that the class actually writing to the log file now implements a common interface and the Singleton itself just passes requests off to the implementor. It is possible to *assign* a new implementation to this variable, as long as it implements **`LoggingInterface`** [8].

Test code can now look like:

```
class NullLogger : LoggingInterface
{
  public void Write( string text )
  {
  }
}
[Test]
public void InputsToOutputs()
{
  TextWriter output =
     new StringWriter( new StringBuilder() );
  Writer writer = new Writer( output );

  // Inject a do-nothing logger into the singleton
  Logging.Instance = new NullLogger();

  string testText = "test text";
  writer.Write( testText );

  Assert.AreEqual( testText, output.ToString() );
}
```

This is not entirely nice, and the dependency on the Singleton remains, but there is still value here: note that the code that uses the **`Logging`** class requires no changes. This therefore gives you enough of a framework to write tests for the **`Writer`** class, which in turn allows you to refactor the **`Logging`** and **`Writer`** classes more safely at a later date, perhaps to use the *Parameterise From Above* pattern described before.

Healing time (multiple remedies)

There are other ways to isolate and manage the Singleton Dependency Problem, and what you end up using depends very much on your circumstances (doesn't everything? There's no substitute for using your brain!).

One simple technique is to have some globally accessible *Factory* [1], which can be asked for services. In simple form, this may manifest itself as a simple *Global Service Locator* [2], such as:

```
public sealed class ServiceLocator
{
  public LoggingInterface Logging
  {
    get{ return logging; }
  }
  // Other application services
  // ...
}
```

This of course bears a striking resemblance to the Singleton we've already seen, but it has the benefits that access to the services provided is all in one place, and that **`ServiceLocator`** can be easily parameterised by injection for testability, etc., using Dependency Injection. It does still have the same inside-out dependency that accessing a Singleton instance has, and the dependency is still hidden.

This form of Service Locator is a kind of *Application Context* [5]. It is an easily accessible single point of contact where different parts of an application can access the *common* context needed to run. It doesn't take much imagination to see that we could combine this with *Parameterise From Above* so that instead of the dependency being inside-out, it becomes outside-in again:

```
public class Writer
{
  public void Write( string text,
     ServiceLocator context )
  {
    // ...
    context.Logging.Write( "Failed to write a line" );
  }
}
```

One of the motivating reasons behind *Encapsulated Context* is that using *Parameterise From Above* can cause over-long parameter lists. This can certainly be a problem where you require several services, and many objects need access to the majority of them, although this should rarely be the case. However, using a single *Encapsulated Context* object which covers all of an application's needs can appear to be little better than lots of (possibly Singleton) Service objects.

Splitting a single multi-purpose *Encapsulated Context* into different *Role-Partitioned Context* objects [6] can alleviate this problem, and has the added benefit of allowing the code to directly publish its immediate dependencies, and provide a certain level of in-code documentation, too.

The small example used here would not really benefit from either *Encapsulated Context* or *Role-Partitioned Context*, and a *Service Locator* would be too heavy-weight. However, few real-world applications have the luxury of being so simple.

The emergence

If we had written our little example using a Test First approach, as advocated by Test Driven Development [7], I think it likely we would have ended up with an interface to represent the **Logging** object being passed in to the **Writer** object – *Parameterise From Above*. It is the simplest approach that fulfils *all* our requirements: the **Writer** object logs to a file in the real application, and is easily testable in isolation from other objects.

As further requirements became apparent, it may have become more useful to use *Encapsulated Context* or *Role-Partitioned Context* objects to pass in the needed services. Unless the requirements really need several independent services, however, *Parameterise From Above* is superior on the basis of its simplicity.

I remain suspicious of *Global Service Locator* partly because the word "global" conjures the wrong connotations for me, but mainly because it has many of the hallmarks (and problems) of Singleton.

A return

Our little journey has taken us through a design that, though very simple – even trivial – lent itself to a very plausible solution using the Singleton Design Pattern, and highlighted some of the difficulties resulting from this decision – however well-intentioned. There are *other* issues with Singletons: they can be particularly difficult to manage in C++, where the tension between lifetime management and the use of static data members causes *real* headaches. In this, maybe C# does us a *dis*service making Singletons so easy to implement.

As for the alternatives? Well, *Parameterise From Above* and *Dependency Injection* both are often criticised for causing contorted, spaghetti logic caused by over-long parameter lists. This criticism can often be overcome by judicious use of *Encapsulated Context* and its relatives, and if it cannot be overcome that way, that could be an indication of a flaw in the design. Using Singletons or globals to reduce parameter lists are non-solutions because they don't reduce the dependencies in the code – they just hide them. If long parameter lists are the symptom, high coupling is the likely disease.

The technique of allowing the Singleton itself to have its Instance changed by client code is a *very* useful one when working with legacy code for which you want to introduce unit tests. It's probably not recommended for new code.

However, regardless of implementation language, problems with lifetime, threading or hidden dependencies, and irrespective of difficulties with testing, there are very few reasons to use Singletons anyway. It's important to understand the difference between needing a single instance of some object or service *right now*, and the necessity that there is only one possible instance of some object or service [9]. Consider the need to use a Stub Object for the logging object – even a test can be an important instance of an object. ■

Singletons . . . can be particularly difficult to manage in C++, where the tension between lifetime management and the use of static data members causes real headaches

Acknowledgements

The motivation for this article arose from a discussion on the accu-general mailing list, which was really focused on replacing instances of Singleton objects with something else. Several *somethings else* were suggested by various respondents. In particular, John Jagger suggested being able to *inject* a Singleton instance with a different implementation (his version was much tidier than mine ☺); Hubert Matthews suggested the *Factory Method Service Locator*, and Adrian Fagg suggested *Encapsulated Context*, with references to detailed information on it and variations on it provided by Allan Kelly and Kevlin Henney.

Notes and references

1 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns – Elements of Reusable Object Oriented Software*, Addison Wesley, 1995

2 Martin Fowler, 'Inversion of Control Containers and the Dependency Injection pattern' http://www.martinfowler.com/articles/injection.html

3 Kevlin Henney, 'Minimalism: A Practical Guide to Writing Less Code', 2002, http://www.two-sdg.demon.co.uk/curbralan/papers/jaoo/Minimalism.pdf

4 Kevlin Henney, 'Programmer's Dozen', 2003, http://www.two-sdg.demon.co.uk/curbralan/courses/ProgrammersDozen.pdf

5 Alan Kelly, 'Encapsulated Context Pattern', http://www.allankelly.net/patterns/encapsulatecontext.pdf (printed in Overload 63 (October 2004) as 'Encapsulate Context')

6 Kevlin Henney, 'Context Encapsulation', http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ContextEncapsulation.pdf

7 Kent Beck, *Test Driven Development By Example*, Addison Wesley, 2002 (See also http://www.testdriven.com)

8 Even with readonly left in, it's actually possible to assign to this variable, but I that too counter-intuitive even for an example!

9 And don't get me started on "Multitons" :-)

The Case Against TODO

TODO - a neat way to label work in progress or an easy way to disguise the flaws in a codebase?

TODO, TO_DO, TO DO, @todo, FIXME, FIX_ME, FIX ME, HACK No other keyword has so many aliases. No other keyword is quite as open to interpretation.

Don't worry, our compiler hasn't gone soft on us. `TODO` isn't really a keyword: it lives in comments and can therefore take whatever form a programmer chooses, safe in the knowledge it won't cause trouble at compile or run time.

On the surface, `TODO` seems both useful and inevitable. No piece of code is ever really finished: there will always be something more to do, and where better to record this information than in the code itself? If we think more carefully though, we realise that `TODO` actually indicates a point at which a decision was made – a decision to defer action, a decision, in fact, to **not** do something. Clearly this decision is less than ideal.

This article investigates the use of `TODO` and friends more closely.

First, we shall consider what it is meant to mean and what it often turns out to mean. Next, we'll search through some code, uncover some use cases, and think about the alternatives.

These alternatives are usually better. `TODO`, it turns out, is not so innocent after all. When used as a shorthand for "more work required" it tells us too little – and often too late – and when used as a convenient label for broken code, it can cause serious damage to a codebase.

What does 'TODO' mean?

As stated in the introduction, `TODO` isn't a keyword, it's a comment. Its exact meaning will depend on the local coding culture and often individual programming style. For example, the Sun Java programming conventions [Reference Java Conventions] state:

> 10.5.4 Special Comments
>
> Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.

Already we've found a shocking new member of our `TODO` set: `XXX`!

I think it's clear, though, that the phrase `TODO` in a comment indicates something more is required. With any luck, the rest of the comment will indicate exactly what that something is.

`FIXME` often seems to be used interchangeably with `TODO` but carries a stronger suggestion that something is broken – if `TODO` is a feature request, then `FIXME` is a defect report. Again, with luck, the rest of the comment will explain what needs fixing.

Unfortunately the rest of the comment is often inadequate. Sometimes we'll find a bare:

```
// TODO
```

sometimes an initialled note-to-self:

```
// FIXME TAG
```

sometimes a plea for attention:

```
// TODO Fix this hack !!!
```

and sometimes even a garbled attempt to cover all bases:

```
// TODO FIXME XXX HACK
```

Of course, `TODO` isn't meant to be pretty. The capital letters shout at the reader, drawing attention to the deficiency. The noise will continue until something gets done.

Ask the code

Maybe I'm being unfair. Let's take a look at `TODO` in action by searching some source code. On a Unix platform the following command should output matches in any files beneath the current working directory:

```
grep -ERi "TO[_ ]?DO|FIX[_ ]?ME|HACK|XXX" *
```

I cannot publish the output of this search on any of the proprietary code I work on. It's confidential information. (The small amount of open source code I have written or contributed to is `TODO` free.) Nor am I going to publish the output of this search on any of the open source code I use – I do not think it would be fair, since this article questions the use of `TODO`. I should stress, though, that I'm grateful to have source access, so that any such comments are at least visible to me.

I would be interested to know how useful the output of your search is. Does the `TODO` list correlate with work-in-progress? Are the `FIXME`s actively being fixed? Or have we merely generated a list of half-baked ideas, abandoned experiments and neglected suggestions?

The next section considers some specific use cases which I hope overlap with our search results.

Use cases and alternatives

Place holders

In another article [Reference Guest], I describe my use of an Emacs **`elisp`** program to generate a skeleton C++ file, which – amongst other things – inserts placeholders for 'Doxygen' comments.

These placeholders had `TODO`s in them for me to fill in. Or at least they used to! I have now decided I would rather leave a class undocumented than have it ever look like this:

```
/**
 * TODO write a description of MyNewClass.
 */
class MyNewClass {
    …
};
```

The `TODO` in the example above serves no useful purpose – I'm well aware the new class needs describing, 'Doxygen' will warn me should I forget

Thomas Guest is an enthusiastic and experienced computer programmer. He has developed software for everything from embedded devices to clustered servers. His website can be found at http://www.wordaligned.org and you can contact him at thomas.guest@gmail.com

the programmer sensibly wishes to put her **work-in-progress** into the source management system as soon as possible

– and as a consequence my **elisp** now generates smaller but better-formed skeleton files. This reflects more accurately the way I aspire to work: no broken code.

While we're on the subject, note that editor macros make it all too easy to create unfinished or extravagent comments. A block of asterisks is a poor wayto fill a source file:

```
/* * * * * * * * * * * * * * * * * * * ** * * * * * * *
*                                                      *
*           M Y   N E W   C L A S S                    *
*                                                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

Generated code

My **elisp** metaprogram is of course a code-generator. When I realised it was producing broken output I stepped in and fixed it. With third-party code generators we may not be so lucky.

Consider a GUI builder which allows us to design our user interface and map buttons events to actions. The output of this builder is some auto-generated code with placeholders:

```
void MyDialog::OnButtonClick(Button button) {
    // TODO. Insert button action code here.
}
```

Clearly there's a potential problem if the auto-generation is repeated. Will any callbacks we implement be replaced with `TODO`s?

There's not much we can do about this – unless we are GUI builder writers, in which case we might consider better ways to separate the generated code from the application-specific implementation. (See [Reference Brown] for a more detailed discussion of this issue.)

Perhaps this is why many seasoned GUI developers drop the framework early on, preferring the control they get from hand-crafted code?

Work-in-progress

Suppose a programmer creates a new class, `NewClass`, whose interface offers clients, amongst other things, a pair of serialization functions:

```
NewClass {
    …
public: // Serialization
    void put(std::ostream & put_to) const;
    void get(std::istream & get_from);
    …
};
```

The output function is quickly coded up. The input function turns out to be rather trickier. Now, the programmer sensibly wishes to put her work-in-progress into the source management system as soon as possible. She checks in an empty implementation:

```
void NewClass::get(std::istream & get_from) {
    // TODO
}
```

Here, we understand the `TODO` to mean that, although the code compiles, the implementation is incomplete. Client code shouldn't try to 'get' instances of this new class just yet.

The trouble is, as far as client code is concerned – at both compile and run time – the comment has no effect. It's not hard to imagine a scenario where we accidentally end up reading in a **NewClass**, leading to some unwanted effect downstream, possibly in an apparently unrelated piece of code.

Any time spent tracking down such a problem has been wasted. The moment the `TODO` was written, we were aware of its exact location and cause.

A better technique is to call a function:

```
void NewClass::get(std::istream & from) const {
    NotYetImplemented();
}
```

Here, **NotYetImplemented()** might fire an assertion, raise an exception, or log an error message. At the very simplest, we could put a rough and ready macro into service:

```
#define NotYetImplemented()
   assert(!"Not yet implemented!")
```

As usual, the move from comment to code improves things. Now the system offers more useful diagnostics in the event of an unimplemented function being called: but it won't save us from trouble if this event happens once the software has been shipped. We are still reliant on someone remembering to finish what got started. Here, test-driven development techniques are invaluable. They are so important they merit fuller discussion later in this article.

Notes To self

Consider the following struct:

```
struct FileSize { // TODO 64 bit
    unsigned long ms_4_bytes;
    unsigned long ls_4_bytes;
};
```

I classify this `TODO` as a note-to-self since it may not be obvious to anyone but the programmer who wrote the comment exactly what should be done to the code on a 64 bit platform. If we **grep** around surrounding source files, we'll probably find a few similar comments and get a better idea of what's required.

Sometimes it's even more clear that we're dealing with a note-to-self.

```
struct FileSize { // TODO 64 bit. TAG
    …
};
```

Here, the author uses his initials to stake a claim on the **struct**. He seems to be saying, "I know what needs doing, and I'll probably be the one who does it". Ideally, of course, when we undertake a 64 bit port, he will still be around to tie up these loose ends; even if he isn't, he has left us some useful pointers.

This isn't a bad use of TODO. It's certainly preferable to the programmer keeping the TODO list in his head, or even in his log book. The information is where it needs to be, often down to individual lines of code. My only quibble is with the note-to-self attitude. When we check in code, we release it to a wider audience; we are publishing our work. These notes-to-self should really be notes-to-everyone.

In an ideal world, then, there will be a little more documentation explaining the porting task in more detail. Pair-programming, peer review and open source development can help us maintain a disciplined approach.

Latent defects

Suppose we're digging around some legacy code and we discover the following bizarre integer literal in an assignment:

```
    flags &= 0xFFFFFFF7;
```

What to do? Evidently bit three of the **unsigned int flags** variable is being cleared, but so are bits thirty-two and above. Is this the intended behaviour?

Now, the code in question is regarded as sound. It works. Target platforms have always had thirty-two bit **int**s, and will continue to do so for the forseeable future.

We could make a note in the code and move on:

```
  flags &= 0xFFFFFFF7;//FIXME this assumes 32 bit ints
```

If we're feeling less confident, the note might read:

```
  flags &= 0xFFFFFFF7;
  // TODO doesn't this assume 32 bit ints !?
```

Or, if we're feeling more confident, we make the fix directly:

```
    flags &= ~(1U << 3);
```

Personally, I dislike adding comments as shown above. Yes, it's better than ignoring the problem, but only marginally. By adding a comment we're ducking the issue. On the other hand it's also risky to modify legacy code even when we think we're fixing it (see Sidebar, Legacy Code). Who knows what compensating code there might be elsewhere in the system?

A more responsible reaction is to dig a little deeper and find exactly how pervasive the problem is. The assumption that **int**s occupy thirty-two bits may cut across many lines in many files – and yet, as stated earlier, this assumption may actually be reasonable, at least for the forseeable future. Not all code needs to be portable.

Even if we are prepared to continue with this assumption about target platforms, the important thing is not to throw away our insight. With barely any extra effort we can replace the TODO with a compile-time assertion [Reference Boost]:

```
    BOOST_STATIC_ASSERT(sizeof(int) * CHAR_BIT == 32);
```

With this assertion in place, the code won't even compile if we try and port to a platform which doesn't meet this assumption.

On the other hand, if digging deeper reveals more latent defects, and if we already have reason to believe the legacy code is in poor shape, then more radical action may be needed (see Legacy Code).

Legacy Code

I haven't defined what I mean by legacy code here – but the risk associated with change is surely a defining characteristic. In his book *Working Effectively with Legacy Code*, Michael Feathers [Reference Feathers] chooses a specific and objective definition: legacy code is code which has no unit tests. He goes on to offer some sound advice on how to work with such code: that is, how to put it under test. Once the tests are in place, the risks associated with change reduce.

Grand designs

At the other end of the scale are the TODOs which claim a glorious future for a sound – if straightforward – block of code.

```
  // TODO use an adaptive search algorithm:
  // 1) keep a log of past commands
  // 2) when the system is idle, review this log to
  //    predict the next command
  // 3) pre-fetch the results of the predicted
  //    command.
  //    This should make the UI more responsive.
```

Let's hope this ingenious scheme isn't attempted until careful analysis shows that it really is necessary and calibrated test runs prove that it really can improve response times.

Until then, this particular TODO is simply an incitement to over-engineering.

Future optimisations

Closely related to Grand Designs are those points in a code-base where a progammer uses TODO to indicate where a sub-optimal solution has been used.

Maybe a hand-crafted container might offer client code quicker lookups than the one which was picked, for convenience, from the C++ Standard Library.

```
    typedef std::map<person, phone_number> phone_book;
    // TODO replace with a hash map for efficiency
```

This TODO looks reasonable enough, but again I think it may lead us astray.

Why should we consider making this replacement? Why should the suggestion be repeated every time we look at this code?

If we do find our code runs too slowly we need to measure first. The hand-rolled hash map might make no perceptible difference; but simplifying the arithmetic in that innocent looking loop the profiler warned us about might realise a 50% speed-up with far less effort.

The hideous hack

Consider the following scenario. Testing reveals that a peculiar – but reproducible – combination of events can lead to deadlock. A SHOWSTOPPER defect is raised and assigned to some unfortunate programmer. The programmer must delay her current task to investigate. After a couple of frustrating days of poring over log files she narrows the problem down. It looks very like a race condition in a particular function.

To test her suspicions, she injects a ten millisecond delay into one of the calling threads. The defect goes away!

Armed with this evidence, she consults the programmers involved. The original author of the function has nothing helpful to say – clearly it's the client code (which does the synchronisation) which is at fault. Equally unhelpfully, the authors of the client code suggest the proper solution is to move responsibility for synchronisation into the function itself.

All this blame-storming is holding up development. The project manager makes the call: check in the hack, change the defect to LOW priority, get on with new features.

I don't think it will come as a surprise to find out that the code still reads as follows:

```
    Sleep(10); // HACK. Workaround a race condition.
               // See DEFECT 5678 for details.
```

Nor should it surprise us to learn that the defect hasn't really gone away, it has just gone under cover. The software still deadlocks. Less often, less reproducibly, but just as disastrously.

This article is about the use of TODO, not dysfunctional development teams. A proper solution will involve the organisation as much as the code-base, and will have to remain beyond the scope of this article.

Clearly, though, something very wrong has happened. I'm not claiming that TODO – or, in this case, its unsavoury relative, HACK – is to blame.

What I do suggest is that this use of HACK opens the door (breaks the window perhaps? see bottom of page) to similar abuse in future. When we introduce such code into our system we sanction the approach it takes, inviting more of the same.

Sure enough, as features continue to be added, we find more and more **Sleep()**s, HACKs and TODOs attempting to disguise a broken threading model.

Test frameworks

Remember the programmer who checked a stubbed function into the source management system?

```
void NewClass::get(std::istream & get_from) {
    // TODO
}
```

In a test-first environment, the TODO is superfluous. The accompanying unit tests show exactly what needs doing. In a console window, we see something like:

```
------------------------------------------------------
   Test    : testNewClassGet
   testNewClass.cpp(57): Expected "foo" but got "bar"
------------------------------------------------------
   Ran 13 tests, 12 Passed, 1 Failed.
```

Here, the feedback is swift and accurate, and continues to be so even once the class is complete (that is to say, once it passes its tests). Should something cause **NewClass** to regress, a good set of tests will isolate the error even before the offending code gets checked in.

In other words, the TODO list and the FIXME list have been replaced by test results. We have done the best we can to ensure our **NewClass** does not end up being yet another **LegacyClass**.

Now remember the HACK which covered up a broken threading model. We will need to work much harder to stop the rot advancing any further (in both the code-base and the organisation), but again, my main recommendation would be to invest in a test framework – a system test framework in this case.

If we can create a suite of tests which exercise the code to systematically expose the threading problems, we may have a chance of understanding them. If we can automate these tests and publish results in a user-friendly form – bearing in mind that users are not just engineers, but everyone with an interest in the code-base – then we may yet fix them.

The case against TODO

This article has covered the use and abuse of TODO. In some cases, it is redundant; in others inadequate; in others misleading; and in yet others it could more usefully be replaced by code. TODO is sometimes a note written in some personal shorthand, which, like many such notes, is in danger of becoming meaningless to even its originator.

I have no major problem with any of these uses, though personally I avoid them. It's the times when TODO (or FIXME or HACK) gets roped in to defer the proper treatment of a defect which make it suspect.

In Overload 68, Alan Griffiths [Reference Griffiths] writes:

> The worst thing that can be done on encountering a problem is to ignore it on the basis that "someone else" should deal with it. The next worst thing is to bury it in a write-only "issues list" in the hope that one day someone will deal with it. If everyone behaves like that then nobody deals with anything.

Griffiths is talking about problems in a wider sense, perhaps, than this article, but he expresses my frustration with TODO perfectly. A search through code for TODOs is all too likely to reveal a "write only issues list". Too often, FIXME silently marks a place where we know something is wrong, but we haven't bothered to do anything about it. Worst of all, HACK gets deployed when we know something is wrong and we fear we might have made it worse. ■

in some cases, it is **redundant**; in others **inadequate**; in others **misleading**; and in yet others it could more usefully be replaced by code

Acknowledgements

I would like to thank Dan Tallis and the editorial teams at The C++ Source and Overload for their help with this article.

References

Boost - http://www.boost.org

Brown - 'Automatically-Generated Nightmares', Silas S Brown, CVu 16.6, ACCU

Feathers - *Working Effectively With Legacy Code,* Michael C. Feathers, Prentice Hall, ISBN: 0131177052

Griffiths - 'Editorial: Size Does Matter', Alan Griffiths, Overload 68, ACCU

Guest - 'Metaprogramming is Your Friend', Thomas Guest, Overload 66, ACCU, Also available at: http://www.wordaligned.org/

Hunt & Thomas - *The Pragmatic Programmer: From Journeyman to Master*, Andrew Hunt and David Thomas, Addison-Wesley Oct 1999, ISBN: 020161622X

Java Conventions - 'Code Conventions for the Java Programming Language', http://java.sun.com/docs/codeconv/

Broken Windows

In *The Pragmatic Programmer: From Journeyman to Master* [Reference Hunt, Thomas] Andrew Hunt and Dave Thomas advise us not to live with broken windows:

> One broken window, left unrepaired for any substantial length of time, instills in the inhabitants of the building a sense of abandonment – a sense that the powers that be don't care about the building. So anothern window gets broken. People start littering. Graffiti appears. Serious structural damage begins. In a relatively short space of time, the building becomes damaged beyond the owner's desire to fix it, and the sense of abandonment becomes reality.

Hunt and Thomas suggest the same is true of software: when we discover something is broken, we must repair it promptly. Deferring the repair work with a TODO or a FIXME risks making things worse. The next programmer to visit the code is unlikely to make the fix, but he may well be encouraged to adopt TODO to defer the treatment of similar problems elsewhere.

Objects for States

Originally captured in Design Patterns, Objects for States is described in close conjunction with the Singleton pattern. This article investigates better alternatives for implementing the pattern in C++.

The Singleton connection

Design Patterns [1] includes the name Objects for States only as an alias and the pattern is probably better known for its primary name: State. I prefer the name Objects for States because it expresses both the intent and resulting structure in a much better way. After all, the main idea captured in the pattern is to represent each state as an object of its own.

Besides the naming issue, everything starts just fine in the pattern description and nothing indicates that Singleton is about to enter the scene. Not even as *Design Patterns* discusses implementation issues concerning the lifetime of state-objects do they actually mention Singleton. Turn the page and suddenly the pattern appears in the sample code with each state implemented as Singleton. Later on *Design Patterns* officially relates the two patterns by concluding that “State objects are often Singletons” [1]. However true that statement may be, is it a good design decision?

The case against Mr Singleton

The Singleton pattern is on the verge of being officially demoted to anti-pattern status. In order to get the freshest insider information possible, I decided to carry out an interview with the subject himself. Mr Singleton surprised me with his honesty and introspective nature.

> “Mr Singleton, you have been accused of causing design damage [6] and of leading programmers to erroneous abstractions by masquerading your tendencies to global domination as a cool object-oriented solution. What are your feelings?”
>
> “I’m just an innocent pattern, I did nothing wrong. I feel truly misunderstood.”
>
> “But your class diagram included in *Design Patterns* looks rather straightforward. It doesn’t get simpler than that – one class only – how could anyone possibly misunderstand that?”
>
> “Well, that’s the dilemma.” He continues with a mystic look on his face: “I look simple but my true personality is rather complex, if I may put it that way.”

I understand he has more to say on the subject. I’ll see if we can get further.

> “Interesting! Care to elaborate?” It seems like he just waited for this opportunity. Mr Singleton immediately answers, not without a tone of pride in his voice:
>
> “Sure, first of all I’m hard to implement.”
>
> “Yes, I’m aware of that. Most writings about you are actually descriptions of the problems you introduce. What springs to my mind is, hmm, well, no offence, discussions about killing Singletons [9], a subject which makes matters even worse. There’s also all the multithreading issues with you involved [10].”
>
> “Yeah, right, but my implementation is the minor problem. Can you keep a secret?”
>
> “Sure”, I reply crossing my fingers.
>
> “Hmm, I shouldn’t really mention this, but Design Patterns are over-using me.”
>
> “Wow! You mean that you are inappropriately used to implement other patterns?”
>
> “Yes, you may put it that way. I mean, part of my intent is to ensure that a class only has one instance. But if an object doesn’t have any internal state, then what’s the point of using me? If there isn’t any true uniqueness constraint, why implement mechanisms for guaranteeing only one, single instance?”

Reflecting on the above dialogue I notice that it describes a common problem with many implementations using Objects for States. In most designs the state objects are stateless, yet many programmers, including my younger self, implement them as Singletons. Sounds like some serious tradeoffs are made. After all, I like to take a test-driven approach and writing unit tests with Singletons involved is a downright scary thought. Mr Singleton agrees:

> “It’s sad, isn’t it? You end up solving the solution. Not only does it mean writing unnecessary code and that’s a true waste; worse is that I’m wrong from a design perspective too.”

There it is! Implementing Objects for States using Singleton is, I quote once more, “wrong from a design perspective”. He said it himself. The good news is that in this case a better design also means less code and less complexity. But before jumping into the details of why and how, let’s leave Mr Singleton for a while and recap the details of Objects for States.

Objects for States recap

Objects for States works by emulating a dynamic change of type and the parts to be exchanged are encapsulated in different states. A state transition simply means changing the pointer in the context from one of the concrete states to the other. Consider a simple, digital stop-watch. In its most basic version, it has two states: started and stopped. Applying Objects for States to such a stop-watch results in the structure shown in Figure 1.

Before developing a concrete implementation, let’s investigate the involved participants and their responsibilities:

- `stop_watch`: *Design Patterns* defines this as the context. The context has a pointer to one of our concrete states, without knowing exactly which one. It is the context that specifies the interface to the clients.
- `watch_state`: Defines the interface of the state machine, specifying all supported events. Depending upon the problem domain, `watch_state` may also implement default actions for different events. The default actions may range from throwing

Adam Petersen is a software developer whose prime professional interests include C++, patterns, agile development, modeling and Lisp. Besides spending way too much time reading tech books, Adam also has somewhat healthier hobbies like chess, music, modern history and Russian literature. He can be contacted at adampetersen75@yahoo.se

Such a dependency is an **obstacle** to unit tests and leads to **big-bang integrations**, although limited to the micro-universe of the context

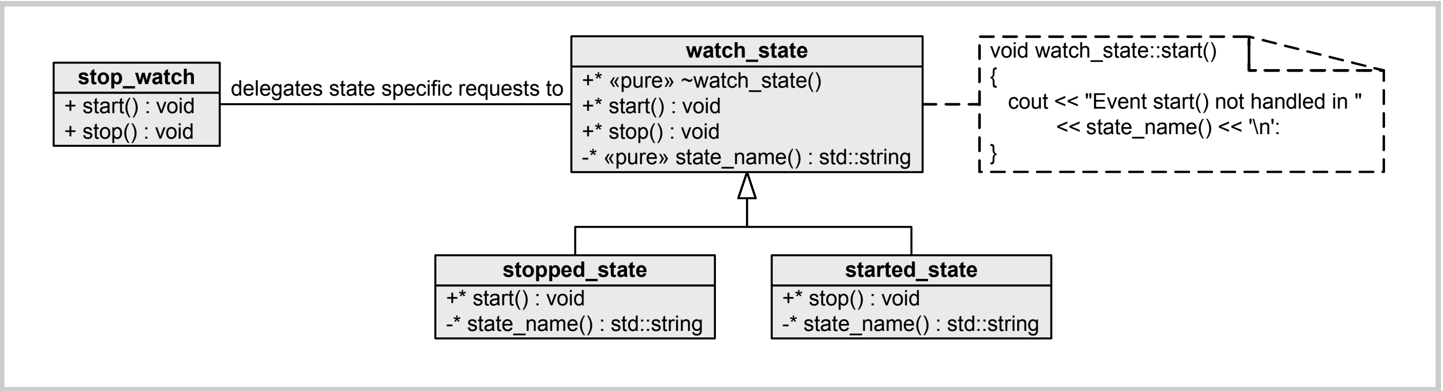

stop_watch
+ start() : void
+ stop() : void
delegates state specific requests to
watch_state
+* «pure» ~watch_state()
+* start() : void
+* stop() : void
-* «pure» state_name() : std::string
void watch_state::start()
{
cout << "Event start() not handled in "
<< state_name() << '\n':
}
stopped_state
+* start() : void
-* state_name() : std::string
started_state
+* stop() : void
-* state_name() : std::string

Figure 1

exceptions and logging to silently ignoring the events (the UML note in Figure 1 shows an example of a default action implemented in the `start()` function that sends a debug trace to standard output).

- `stopped_state` and `started_state`: These are concrete states and each one of them encapsulates the behaviour associated with the state it represents.

It depends

Design Patterns includes many examples of good OO designs. An example is its adherence to one of the most important design principles: “Programming to an interface, not an implementation”. In fact all patterns in the catalogue, with one notable pathological exception – Singleton, adhere to this principle. Yet there are some subtle nuances to watch out for. Upon state transitions the pointer in the context has to be changed to the new state. The typical approach is to let each concrete state specify their successor state and trigger the transition. This way each state needs a link back to its context.

In its canonical form, Objects for States uses a `friend` declaration to allow states to access their context object. A `friend` declaration used this way breaks encapsulation, but that’s not really the main problem; the problem is that it introduces a cyclic dependency between the context and the classes representing states. Such a dependency is an obstacle to unit tests and leads to big-bang integrations, although limited to the micro-universe of the context. Fortunately enough it is rather straightforward to break this dependency cycle. The first step is to introduce an interface to be used by the states:

```
class watch_state;

class watch_access
{
 public:
  virtual void change_state_to(
     watch_state* new_state) = 0;

 protected:
  ~watch_access() {}
};
```

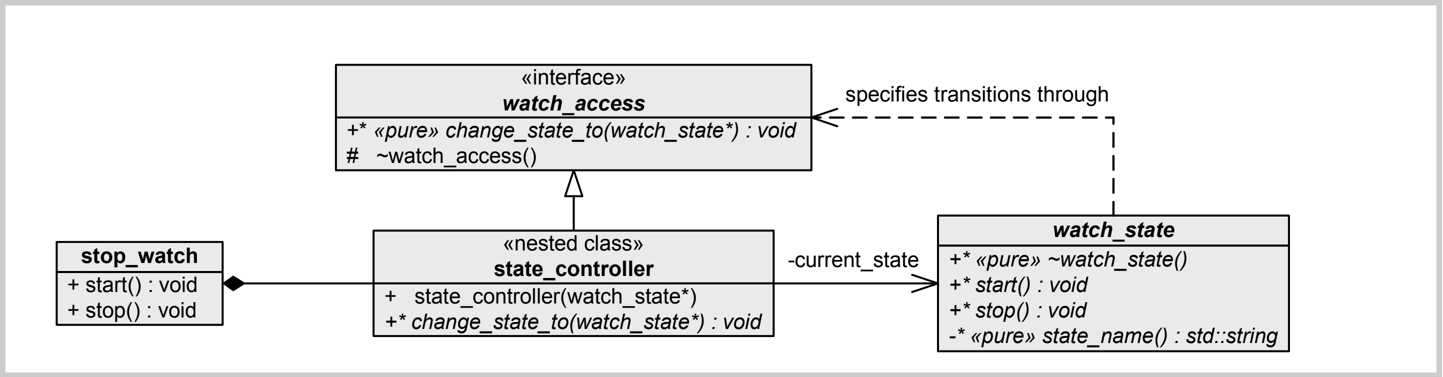


Figure 2

inheritance, private or not, puts strong compile-time **dependencies** upon the clients

This interface is realized in the context and each state is given a reference to it. A state can now make a transition by invoking **`change_state_to()`**. Now, I deliberately didn't write exactly how the context shall implement the interface. From a design and usability perspective public inheritance isn't a good idea; **`watch_access`** is a result of our implementation efforts of weakening the dependencies and we really don't want to expose implementation details to clients of the **`stop_watch`**.

The perhaps simplest solution is offered by the idiom Private Interface [3]. All there is to it is to let **`stop_watch`** inherit **`watch_access`** privately. Now a conversion from **`stop_watch`** to **`watch_access`** is only allowed within the **`stop_watch`** itself. That is, the **`stop_watch`** can grant controlled access to its states and clients are shielded from the **`watch_access`** interface. Or are they really? Well, they are shielded from the conceptual overhead of the interface but there's more to it.

What worries me is that inheritance, private or not, puts strong compile-time dependencies upon the clients of **`stop_watch`**. In his classic book *Effective C++*, Scott Meyers advices us to "use private inheritance judiciously" [2]. Meyers also proposes an alternative that I find more attractive, albeit with increased complexity: declare a private nested class in the context and let this class inherit publicly. The context now uses composition to hold an instance of this class as illustrated in Figure 2. Not only is it cleaner with respect to encapsulation, it also allows us to control the compilation dependencies of our clients as it is possible to refactor it to a Pimpl [8] solution if needed.

Enough of fancy diagrams – let's carve it out in code..

```
class stop_watch
{
 public:
  ...
 private:
  // Meyers Item 39: Prefer public inheritance
  // plus composition in favour of private inheritance.
  class state_controller : public watch_access
  {
     ...
     public:
       ...
       virtual void change_state_to(
          watch_state* new_state)
       {
          ...
       }
     };
     state_controller state;
};
```

With the main structure of the context in place, we're ready to tackle the allocation of states.

A dynamic allocation scheme

Our first approach is to allocate the states dynamically as they are needed. A state transition simply means allocating the new state, wrapped in a suitable smart pointer from boost [4], and passing it to the context. Here's an example on the **`stopped-state`**:

```
  // watch_state.h
...
typedef boost::shared_ptr<watch_state>
watch_state_ptr;

// stopped_state.cpp
void stopped_state::start(watch_access& watch)
{
   watch_state_ptr started(new started_state);

   watch.change_state_to(started);
}
```

The started-state has an identical mechanism, but of course it allocates **`stopped_state`** as its successor. With the allocation scheme in place we can implement the context, shown in Listing 1, overleaf.

Here we let the **`stop_watch`** specify its initial state upon construction:

```
// stop_watch.cpp
stop_watch::stop_watch()
  : state(watch_state_ptr(new stopped_state))
{
}
```

Our preference of public inheritance in combination with composition over private inheritance leads to an extra level of indirection. We can hide this indirection by overloading **`operator->`** in the **`state_controller`**, which makes the context's delegation to the states straightforward:

```
// stop_watch.cpp
void stop_watch::start()
{
  state->start(state);
}

void stop_watch::stop()
{
  state->stop(state);
}
```

Dynamic allocation of the states is a simple solution, yet it makes several tradeoffs:

nobody wants to get caught using globals, yet **global variables** are more **honest** about the intent than to **camouflage** them as Singletons

```
  // stop_watch.h
class stop_watch
{
 public:
  stop_watch();
  void start();
  void stop();

 private:

  class state_controller : public watch_access
  {
      watch_state_ptr current_state;

    public:
      state_controller(
        watch_state_ptr initial_state)
        : current_state(initial_state)
      {
      }

      // Hide the extra indirection for the
      // client by using en masse delegation.
      watch_state_ptr operator->() const
      {
         return current_state;
      }

      virtual void change_state_to(
         watch_state_ptr new_state)
      {
         current_state = new_state;
      }
    };

  state_controller state;
  };
```

Listing 1

- + Allows for stateful states, i.e. instance variables in the states.
- – Potentially many and frequent heap allocations may have negative performance impact.
- – Hard to change to sharing states (such a change ripples through all states).
- – Dependent upon a concrete class (i.e. the next state), which is a barrier to unit tests

Sharing states - the return of the Singletons

With instance variables in the states, dynamic allocation is a simple solution. However, in most applications of Objects for States the state-objects are there just to provide a unique type and do not need any instance variables. *Design Patterns* describes this as "If State objects have no instance variables […] then contexts can share a State object" [1]. In their sample code, *Design Patterns* notes that this is the case; only one instance of each state-object is required, and with that motivation makes each state a Singleton.

After my interview with Mr Singleton I promised to explain why this is the wrong abstraction. The reason is that the responsibility of managing state-instances is put on the wrong object, namely the state itself, and an object should better not assume anything about the context in which it is used. *Design Patterns* describes a particular case where only one instance is needed. This need, however, doesn't imply a uniqueness constraint on the state-objects themselves that would motivate the Singletons. Further, whether states should be shared or not should be decided in the context. Obviously the Singleton approach breaks this rule and, for all practical purposes, forces all states to be stateless.

To summarize, Singleton leads to:

1. an erroneous abstraction,
2. unnecessary code complexity,
3. superfluous uniqueness constraints,
4. and it seriously limits unit testing capabilities.

Clearly another approach would be preferable. However, before sharing any states, I would like to point to Joshua Kerievsky's advice that "it's always best to add state-sharing code after your users experience system delays and a profiler points you to the state-instantiation code as a prime bottleneck" [5].

Going global

When implementing Objects for States the uniqueness constraint of Singleton is actually an unwanted by-product of the solution. So, let's focus on the second part of Singleton's intent: "provide a global point of access" [1]. These are the things programmers speak low about – nobody wants to get caught using globals, yet global variables are more honest about the intent than to camouflage them as Singletons. Consider the following code snippet:

```
// possible_states.h
class watch_state;

namespace possible_states{
extern watch_state* stopped;
extern watch_state* started;
}

// stopped_state.cpp
#include "possible_states.h"
```

What's left to the states is **specifying** their successors in **abstract** terms

```
void stopped_state::start(watch_access& watch)
{
   using possible_states::started;

   watch.change_state_to(started);
}
```

No constraints on the number of possible instances in the states themselves. But who defines them? The context seems like a good candidate:

```
// stop_watch.cpp
namespace{
stopped_state stopped_instance;
started_state started_instance;
}

namespace possible_states{
watch_state* stopped = &stopped_instance;
watch_state* started = &started_instance;
}
```

Except for the construction (we have to initialize our `state_controller` with `possible_states::stopped` instead of a dynamically allocated state), the rest of the context code stays the same. Any tradeoffs made? Yes, always. Here they are
:

- + Conceptually simple and definitely simpler than the classic Singleton approach (same characteristics, but more honest in its intent).
- + No dependencies from the states upon concrete classes (only a forward declaration is actually used in `possible_states.h`).
- + Primitive but possible way to unit test individual states by use of link-time polymorphism (this technique uses the linker to link in different state definitions, i.e. test-stubs, instead of the real ones in `stop_watch.cpp`).
- +– States are shared.
- – Forced to share states, which makes it virtually impossible to use stateful states.
- – Still not quite true to the 'program to an interface' principle.
- – Scalability problems with `possible_states.h`, which must be updated each time a state is added or removed.

In control

Using link-time polymorphism to unit test? Yuck! Not particularly OO, is it? No, it sure isn't, but I wouldn't discard a solution just by that objection. Anyway, what about finally approaching a solution that removes the dependencies between the sub-states? Moving the state management into the `state_controller` makes it possible.

```
// watch_access.h
class watch_access
{
public:
   virtual void change_to_started() = 0;

   virtual void change_to_stopped() = 0;
  ...
};

// stop_watch.h
class stop_watch
{
  ...

  class state_controller : public watch_access
  {
      started_state started;
      stopped_state stopped;

      watch_state* current_state;

    public:
      state_controller()
         : current_state(&stopped)
      {
      }
      virtual void change_to_started()
      {
         current_state = &started;
      }

      virtual void change_to_stopped()
      {
         current_state = &stopped;
      }

      ...
   };
  ...
};
```

The `state_controller` allocates all possible states and switches between them as requested by the states. What's left to the states is specifying their successors in abstract terms:

```
// stopped_state.cpp
void stopped_state::start(watch_access& watch)
{
   watch.change_to_started();
}

// started_state.cpp
void started_state::stop(watch_access& watch)
{
   watch.change_to_stopped();
}
```

And here we are, finally programming to an interface and not an implementation. Let’s look at the resulting context:

- + The responsibility for the allocation scheme is where it should be: in the context.
- + States are easily shared among instances by making them static. Such a decision is taken in the context and not coded into the states themselves as in the traditional Singleton approach.
- + All states written towards an interface, which make them easy to unit test.
- – Doesn’t scale well. **`watch_access`** runs the risk of growing fat as it has to provide methods for all possible states, which is a similar problem to the global approach with `possible_states.h`.

A generative approach

The previous solution indicated potential scalability problems; adding new states requires modifications to **`watch_access`** and its implementer, **`state_controller`**. In my experience this has been an acceptable trade-off for most cases; as long as the state-machine is stable and relatively few states are used (5 - 10 unique states) I wouldn't think twice about it. However, in the ideal world, introducing a new state should only affect the states that need transitions to it. Reflecting upon our last example, although limited to only two states, the pattern is clear: the different methods for changing state (change_to_started(), change_to_stopped()) are identical except for the type encoded in the function name. Sounds like a clear candidate for compile-time polymorphism. The core idea is simple: each state instantiates a member function template with the next state as argument.

```
// Example from stopped_state.h
void stopped_state::start(watch_access& watch)
{
   watch.change_state_to<started_state>();
}
```

Each member function template instantiation creates the new state object and changes the reference in the context. Something along the lines of:

```
class X
{
...
   template<class new_state>
   void change_state_to()
   {
      watch_state_ptr created_state(new new_state);

      current_state = created_state;
   }
};
```

A quick quiz: in the listing above, what class should X be? The states specify their transitions by invoking methods on **`watch_access`** and by means of the virtual function mechanism the call is dispatched to the context. Now, there’s no such beast as virtual member function templates in C++. The solution is to intercept the call chain and capture the template argument in an, necessarily non-virtual, member function template, create the new state instance there and delegate to the context by a virtual function (see Listing 2).

Considering the tradeoffs shows that the one step forward in scalability pushed us back with respect to dependency management:

:

- + Scales well, no know-them-all class; the compiler generates code to instantiate states.
- – The states depend upon concrete classes

```
// watch_access.h
class watch_access
{
public:

   template<class new_state>
   void change_state_to()
   {
      watch_state_ptr created_state(new new_state);

      change_state_to(created_state);
   }

protected:
   ~watch_access() {}
   typedef boost::shared_ptr<watch_state>
      watch_state_ptr;
private:
   // Delegate the actual state management to the
   // derived class through this method.
   virtual void change_state_to(
      watch_state_ptr new_state) = 0;
};

// stop_watch.h
class state_controller : public watch_access
{
   watch_state_ptr current_state;

 public:
   state_controller()
   {
     // Specify the initial state.
     watch_access::change_state_to<stopped_state>();
   }

   virtual void change_state_to(
      watch_state_ptr new_state)
   {
      current_state = new_state;
   }
   ...
};
```

Listing 2

Recycling states

The last example brought us back to a dynamic allocation scheme. However, that knowledge is encapsulated within **`watch_access`** and we can easily switch to another allocation strategy. For example, in a single-threaded context static objects are a straightforward way to share states and avoid frequent allocations:

```
// watch_access.h
class watch_access
{
public:
   template<class new_state>
   void change_state_to()
   {
      static new_state created_state;

      change_state_to(&created_state);
   }
   ...
};
```

State objects can also be recycled by introducing a variation of the design pattern Flyweight [1]. In fact, *Design Patterns* links these two patterns together with its statement that “it’s often best to implement State […] objects as flyweights”. Does the claim hold true? Let’s try it out and see.

```
template<class flyweight>
class flyweight_factory
{
public:
   typedef boost::shared_ptr<flyweight>
    flyweight_ptr;

   template<class concrete_flyweight>
   flyweight_ptr get_flyweight()
   {
      const std::string key(
       typeid(concrete_flyweight).name());

      typename pool_type::const_iterator
       existing_flyweight(pool.find(key));

      if(pool.end() != existing_flyweight) {
         return existing_flyweight->second;
      }
      else {
         flyweight_ptr new_flyweight(
          new concrete_flyweight);

         const bool inserted = pool.insert(
          std::make_pair(key, new_flyweight)).second;
         assert(inserted);

         return new_flyweight;
      }
   }

private:
   typedef std::map<std::string, flyweight_ptr>
    pool_type;
   pool_type pool;
};
```

Listing 3

First each object is associated with a unique key. The idea is, that the first time an object is requested from the flyweight factory, a look-up is performed. If an object with the requested key already exists a pointer to that object is returned. Otherwise the object is created, stored in the factory, and a pointer to the newly created object returned. The example below introduces a pool for state objects in a `flyweight_factory` using the unique type-name as key (Listing 3).

The flyweights are fetched from the instantiations of the member function template in `watch_access` (Listing 4).

The state_controller stays as before because the internal protocol, `change_state_to(watch_state_ptr)`, is left untouched:

- + Scales well, no know-them-all class; the compiler generates code to instantiate states.
- + Allows for sharing states among all instances of `stop_watch` by making the `flyweight_factory` static in `watch_access`.
- + Generic `flyweight_factory` for all default-constructable types.
- – The states depend upon concrete classes.
- – Relatively high design complexity

Conclusion

As this article has highlighted the problems inherent in a Singleton based Objects for States solution, it feels fair to let Mr Singleton get the final word. After all, if I was successful his career may suffer. Will the two patterns finally be separated?

"I sure hope so", Mr Singleton answers, "Clearly there are better alternatives and if I ever get the opportunity I'm prepared to sacrifice my link in Objects for States in the name of good design."

"That's a great attitude and I'm delighted you take it that way. Speaking of design, any particular solution you would recommend?"

"I don't think you can put it that way. Like all design alternatives each one of them comes with its own set of tradeoffs, which must be carefully balanced depending on the problem at hand." ■

```
class watch_access
{
   typedef flyweight_factory<watch_state>
    state_factory;
   state_factory factory;

public:
   template<class new_state>
   void change_state_to()
   {
     change_state_to(
      factory.get_flyweight<new_state>());
   }
   protected:
   ~watch_access() {}

  typedef state_factory::flyweight_ptr
   watch_state_ptr;
private:
   // Delegate the actual state management to the
derived
  // class through this method.
   virtual void change_state_to(
   watch_state_ptr new_state) = 0;
};
```

Listing 4

References

1. Gamma, Helm, Johnson & Vlissides, *Design Patterns*, Addison-Wesley, 1995
2. Scott Meyers, *Effective C++ Third Edition*, Addison-Wesley, 2005
3. James Newkirk, *Private interface*, 1997, http://www.objectmentor.com/
4. http://www.boost.org
5. Joshua Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004
6. Mark Radford, 'SINGLETON – The Anti-Pattern!', Overload 57
7. The complete source code for this article: www.adampetersen.se
8. Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000

   The Pimpl idiom was originally described by John Carolan as the "Cheshire Cat".
9. John Vlissides, *Pattern Hatching*, Addison-Wesley, 1998
10. Meyers & Alexandrescu, 'C++ and the Perils of Double-Checked Locking', 2004, http://www.aristeia.com/

Acknowledgements

I would like to thank Drago Krznaric, Alan Griffiths, Phil Bass, and Richard Blundell for their valuable feedback.

All of the code for this article is available from my website: http://www.adampetersen.se