overload 121

JUNE 2014 £3

The Code Is Not the Most Important Thing

Choose the right sort of problem, focus on strategies for solving it, and the code will come easily

Minimal Overhead for Multiple Interfaces

A performant technique for a single class providing implementations of multiple interfaces

Branching Strategies

The main types of VCS branching strategy, and which you should use

Stop the Constant Shouting

Making C++ stylistic decisions in the modern world

Beauty in Code

Attractiveness matters. We show how generalisation can be more beautiful than special cases

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 121

June 2014

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 122 should be submitted by 1st July 2014 and those for Overload 123 by 1st September 2014.

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

Very small or very far away: A sense of perspective

A tension exists between being immersed in something and stepping back. Frances Buontempo considers the importance of taking a break.

I'd like to thank Ric Parkin for stepping in for the last issue of *Overload*. I needed to take a break and now I'm back. Taking a break can take various forms. Sometimes they are by choice, for example just to go outside and enjoy some sunshine for a brief moment, and on other occasions a hiatus is enforced either by others or by personal circumstances, often for longer. Though such an interruption can be irritating, it can also be useful. Just leaving the computer for a moment to get a glass of water will let you rehydrate, allowing better concentration. Simply standing up and looking out of the window is reputed to be good for our eyes. I personally find I frequently spot the cause of a bug or a problem by glancing away from the monitors then looking back. It seems the change of physical perspective can also shift my mental viewpoint. Stepping away from the keyboard is important for both the quality of your work, and for you. Many programmers have notoriously bad posture. Taking a moment to stretch and re-adjust the body can both help to avoid chronic problems such as lower back pain, upper limb disorder or repetitive strain injury (RSI) for which insufficient rest breaks are purported to be a risk factor [RSI]. I'm not suggesting stretching prevents RSI, just that self-awareness combined with some movement can avoid long term issues.

Other forms of rest are neither self-imposed nor preventative. After, in fact during, this year's ACCU conference I had so much fun talking to so many people, I lost my voice. I was left, literally speechless. This is very frustrating, but means you have to listen and may therefore spot things you would not otherwise have done had you been seeking a moment to share your words of wisdom or a timely wisecrack. I have lost my voice on a few occasions. It never ceases to amaze me the number of times people will try to phone me up, and even if warned still expect me to speak. Unsurprisingly, the sound of a ringing phone is not a miraculous cure for laryngitis. At the conference I was reduced to having a list of useful phrases to hand, which I recently rediscovered. The first on the list is, of course, "I've lost my voice," closely followed by "Geeks", "Drink!" and "Slackers". If you couldn't speak for a few days what phrases would you choose? It's interesting to look back and see what I 'said' to various people, and am aware that it probably reveals more about me than I want to know. Why did I not have at least one positive, life affirming stock phrase prepared? How many times did I really need to write down "WTF?!"?!

It is good to take time out to think about your regular routine. Do you take a lunch break? Many people grab an overpriced sandwich and then return to their keyboard to lunch 'al desko', or indeed get food delivered straight to their desk. The rationale seems to be that there is a linear relationship between time spent in front of a computer and the amount of productive work completed. Once we had decided how to quantify 'productive', I suspect the graph would be neither linear nor, importantly, increasing after a critical point. Even if there's nowhere nice to sit and eat your lunch, it is often worth just going for a walk round the block just to give yourself space to step away from the problem at hand and look at it differently. Do you go home at night? I have met several people who will stay until really late, almost every day. It can be difficult to tear yourself away from a very interesting problem, and yet it is often when we withdraw from the forefront of a situation we get a clearer perspective. Sometimes programmers will stay late trying to make last minute bug fixes in order to release on time. Many of us will have been at our desks in the small hours, hacking away, possibly making new bugs as we go. It might avoid reputational damage to delay the release rather than release last minute untested hacks and patches. Sleep is, after all, important, even though there seems to be a current trend among students for taking so called 'smart drugs' of late. These allow you to 'pull an all-nighter' presumably allowing you to make a deadline for course work or to cram all night before an exam. It would be interesting to find a way to measure the quality of the work produced under such circumstances. Many of us do turn to stimulants to stave off sleep. Famously, a "Mathematician is a machine for turning coffee into theorems." [Renyi] and programmers also turn coffee into source code, so perhaps staying awake at the appropriate time matters too.

The father of Russian vodka [vodka], Dmitri Mendeleev is also renowned for inventing (or discovering) the periodic table. Though others before him had tried to arrange the elements in some organised structure, his approach formed the basis of what we'd recognise today as the periodic table. He suspected some atomic weights were wrong and correctly predicted that some elements had not yet been discovered. The popular story tells us he discovered the ordering after writing the elements on playing cards and arranging them in different ways for days [Mendeleev]. In fact, it seems he spent many sleepless nights thinking about the problem and dozed off one day to realise a new arrangement in a dream:

> On the night of February 17, 1869, the Russian scientist Dmitri Mendeleyev went to bed frustrated by a puzzle he had been playing with for years: how the atomic weights of the chemical elements could be grouped in some meaningful way – and one that, with any luck, would open a window onto the hidden structure of nature. He dreamed, as he later recalled, of 'a table where all the elements fell into place as required.' His intuition that when the elements were listed in order of weight, their properties repeated in regular intervals, gave rise to the Periodic Table of the Elements – which, though much revised since, underlies modern chemistry. [dream]

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 12 years professionally, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

Kekulé is said to have discovered the shape of benzene in a day-dream, seeing the Ouroboros snake chasing its own tail [Kekulé]. Whether these visions showed a deep truth to the chemists or whether the moment of sleep and dreams allowed different, less conscious associations and reformulations to occur could be a matter for dispute. Furthermore there is every chance these claims about visions in dreams are simply fables [Baylor]. The tales of inspired dreams are often interwoven with long patches of sleep deprivation, while trying to grapple with a problem. When you are absorbed deeply in a problem it is hard to step back and you do need to give it enough time to be able to hold several of the pieces in your head. Nevertheless, I am certain that stepping away from either the lab bench, or the keyboard, can allow different perspectives to form. Perhaps all workplaces should have camp-beds or comfy chairs installed for a quick nap if you get stuck. Or perhaps you should just go home sometimes, take a holiday or even take a sabbatical if you can.

The term Sabbatical has its roots in the Old Testament, where we are told God takes a rest from Creation on the seventh day, or rather stops, having completed the task. This theme of rest emerges in other places, such as letting fields rest, which is still practised in agriculture, letting the land lie fallow from time to time, partly to stop pests and bugs from building up. It seems academic sabbaticals are frequently taken to finish a piece of work without distractions, rather than as a complete rest. Not all employers allow sabbaticals, though some will allow time for training, such as attending a conference. Perhaps you write code as a hobby rather than professionally, though you should still take time to sit back and relax or reflect. Sometimes a physical change of circumstances can help. Sitting at the same desk can make it hard to let go and find a new viewpoint. Some people manage to find a 'sacred space' or '(wo)man cave' to withdraw to. If you can't find a physical space, then you can step back in other ways, perhaps getting lost in music. For example, Adam Tornhill recently considered the relationship between music and concentration in *CVu* [Tornhill14].

Sabbaticals are for a specific period of time, be that one day, or a whole year. It can be surprisingly difficult to write a program or solve a difficult problem in a pre-specified time period, or indeed guess in advance how long might be needed. Perhaps one reason people stay too long at work is because either they are 'in the zone' or being distracted by various other things that might need dealing with first, or even are stuck and have a looming deadline, so feel staying late and hitting the problem with various hammers might just work. It can be helpful to time box activity, maybe just allowing yourself an hour, to the end of the working day, or the end of a week to do something. It can then be useful to get some feedback. Someone else's viewpoint will differ to yours. A code review can reveal better ways of doing things and potential problems. Watching someone use your shiny new code can be informative. It might be worth getting continuous feedback as you go, via tests and a continuous integration setup, from colleagues, maybe as you pair program. You could even wire yourself up, to measure what you are really up to. Ed Sykes recently mentioned Quantify Dev [QD] which logs stats on you as a developer, such as build time or if it succeeds or not. The theory is by exploring this data you can find ways to improve. I'm not completely sure I want stats kept on how many browser tabs I have open, or how much time I have spent playing 2048 [2048] recently, though a different perspective on how you are developing is always useful.

Perhaps this had been a long-winded way for thanking Ric and owning up that I still have no idea what to write an editorial on. Hopefully I will be in a better position to write one next time.

References

[2048] http://gabrielecirulli.github.io/2048/

[Baylor] 'What do we really know about Mendeleev's Dream of the Periodic Table? A Note on Dreams of Scientific Problem Solving' George W Baylor *Dreaming*, Vol 11, No 2 2001

[dream] *Mendeleyev's Dream: The Quest for the Elements* Paul Strathern, St Martin's Press

[Kekulé] http://en.wikipedia.org/wiki/Friedrich_August_Kekul%C3%A9_von_Stradonitz#The_ouroboros_dream

[Mendeleev] http://www.rsc.org/education/teachers/resources/periodictable/pre16/develop/mendeleev.htm

[QD] http://www.quantifieddev.org/

[Renyi] http://en.wikiquote.org/wiki/Paul_Erd%C5%91s

[RSI] http://www.rsi.org.uk/

[Tornhill14] Adam Tornhill. The soundtrack to code. *CVu*, 25(6):7–9, January 2014

[vodka] http://en.wikipedia.org/wiki/Dmitri_Mendeleev

Branching Strategies

Branching can either be embraced or avoided. Chris Oldwood documents the pros and cons of three main branching strategies.

One area in software development that appears to have suffered from the malaise of the Cargo Cult [Wikipedia-1] is the use of branching within the version control system. The decision to use, or avoid, branches during the development of a software product sometimes seems to be made based on what the 'cool companies' are doing rather than what is suitable for the project and team itself.

What is often misunderstood about the whole affair is that it is not necessarily the branching strategy that allows these cool companies to deliver reliable software more frequently, but the other practices they use to support their entire process, such as automated testing, pair programming, code reviews, etc. These, along with a supportive organizational structure mean that less reliance needs to be made on the use of code branches to mitigate the risks that would otherwise exist.

This article describes the three main types of branching strategy and the forces that commonly dictate their use. From there it should be possible to understand how the problems inherent with branches themselves might be avoided and what it takes to live without them in some circumstances.

Codeline policies

Branches are lines, in the genealogy sense, of product development that reflect an evolution of the codebase in a way that is consistent for a given set of constraints. In essence each branch has a policy [Berczuk02] associated with it that dictates what types of change (commit) are acceptable into that codeline. When there is an 'impedance mismatch' [c2-1] between the code change and the policy, a branch may then be created to form a new codeline with a compatible policy.

All this talk of 'forces' and 'policies' is just posh speak for the various risks and mitigating techniques that we use when developing software. For example a common risk is making a change that breaks the product in a serious way thereby causing disruption to the entire team. One way of reducing the likelihood of that occurring is to ensure the code change is formally reviewed before being integrated. That in turn implies that the change must either be left hanging around in the developer's working copy until that process occurs or committed to a separate branch for review later. In the former case the developer is then blocked, whilst in the latter you start accruing features that aren't properly integrated. Neither of these options should sound very appealing and so perhaps it's the development process that needs reviewing instead.

Merging can be expensive

Branching is generally cheap, both in terms of version control system (VCS) resources and time spent by the developer in its creation. This is due to the use of immutability within the VCS storage engine which allows it to model a branch as a set of deltas on top of a fixed baseline.

Chris Oldwood is a freelance developer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race and can be contacted via gort@cix.co.uk or @chrisoldwood.

Whilst the branch creation is easy, keeping it up-to-date (forward integration) and/or integrating our changes later (reverse integration) can be far more expensive. It's somewhat ironic that we talk about 'branching' strategies and not 'merging' strategies because it's the latter aspect we're usually most interested in optimising. *Merge Debt* is a term that has sprung up in recent times to describe the ever increasing cost that can result from working in isolation on a branch without synchronising yourself with your surroundings.

Improvements in tooling have certainly made merging two text-based files slightly easier and there are new tools that try and understand the 'meaning' of a code change at a language level to further reduce the need for manual intervention. Of course even these cannot help when there are semantic conflicts (syntactically correct changes that just do the wrong thing) [Fowler]. And sadly binary files are still a handful. Refactoring can also be a major source of merge headaches when the physical structure of the codebase changes underneath you; this is compounded by the common practice of giving the folders and files the same names as the namespaces and classes.

As we shall see in the following sections, branches are essentially graded by their level of stability, or degree of risk. Consequently the preferable direction for any merging is from the more stable into the more volatile on the assumption that tried-and-tested is less risky. The reason 'cherry pick' merges [c2-2] get such a bad name is because they usually go against this advice – they are often used to pull a single feature 'up' from a more volatile branch. This carries with it the risk of having to drag in dependent changes or to try and divorce the desired change from its dependants without breaking anything else.

Integration branches

Before embarking on a full discussion of the main branching strategies we need to clear up some terminology differences that often come up as a result of the different naming conventions used by various VCS products.

Although there are three basic strategies, there are only two real types of branch – integration and private. Either you share the branch with others and collaborate or you own the branch and are solely responsible for its upkeep. It's when you share the branch with others that the sparks really start to fly and so these tend to be minimised.

For small teams there is usually only a single major integration branch and this often goes by the name of main, trunk or master. Sometimes this is known as *the* development branch to distinguish it from one of the other more specialised kinds. Either way it's expected that this will be the default branch where the majority of the integration will finally occur.

In larger organisations with much bigger teams there might be many integration branches for the same product, with perhaps one integration branch per project. At this scale the integration branch provides a point of isolation for the entire project and may spin off its own child branches. Multiple integration branches come with additional overhead, but that may well be less than the contention generated by a large team trying to share a single integration branch. If the project itself carries a large degree of

The notion of 'always be ready to ship' engenders an attitude of very small incremental change

uncertainty or cannot be delivered piecemeal then this project-level isolation can be more beneficial in the long run.

Release branch

Back in the days before VCS products supported the ability to branch you essentially had only one branch where all change took place. As the development process reached a point where the product was readying itself for a formal release, a code freeze was often put in place to reduce the changes to only those directly required to get the product out of the door. For those developers not directly working on 'finishing' the product they had to find other work to do, or find other ways to manage any code changes destined for later releases.

Once branching became available a common answer to this problem was to branch the codebase at a suitable moment so that work could continue on the next version of the product in parallel with the efforts to the stabilise the impending release. The codeline policy for a release branch is therefore based around making very few, well-reviewed, well-tested changes that should resolve outstanding issues without creating any further mess. As the release date approaches finding the time to continually test and re-test the entire product after every change can become much harder and therefore more time is often spent up front attempting to decide whether further change is even really desirable.

The branch is often not 'cut' from the development line at an arbitrary point in time – there will probably have been a reduction in high-risk changes leading up to the branch point so as to minimise the need to try and revert a complex feature at the last minute. By the time the release branch is ready to be created it should be anticipated that future additional changes will be kept to a bare minimum. This implies that during project planning the high-risk items are front-loaded to ensure they are given the longest time to 'bed in', i.e. you don't upgrade compilers the day before a release.

The most extreme form of a product release is probably a patch, or hotfix. Time is usually the most critical aspect and so it demands that any change be completed in total isolation as this allows it to be done with the highest degree of confidence that there are no other untoward side-effects. This kind of release branch is usually created directly from a revision label as that should be the most direct way to identify the part of the product's entire history that corresponds to the product version needing remediation. Whereas a branch is an evolving codeline, a label (or tag) is a snapshot that annotates a single set of revisions as a specific milestone.

What should be apparent about this particular strategy is that it's mostly about compensating for a lack of stability in the main development process. If you never have to worry about supporting multiple product versions, then *in theory* you can change your development process to avoid the need for formal release branches. By ensuring you have adequate automated feature and performance testing and a streamlined development pipeline you *should* be able to deliver directly from the main integration branch.

However, despite the development team's best efforts at working hard to minimise the delays in getting a feature into production, there can still be other organizational problems that get in the way of delivery. Maybe there needs to be formal sign-off of each release, e.g. for regulatory purposes, or the QA cycle is out of your hands. In these cases the release branch acts more like a quarantine zone while the corporate cogs slowly turn.

From a merging perspective release branches are generally a low-maintenance affair. As already stated the most desirable merge direction is from the stable codebase and release branches changes should be about the most carefully crafted of them all. Due to each one usually being an isolated change with high importance they can be merged into any ongoing integration branches the moment it becomes practical instead of waiting until the end.

Feature/task branch

If you think of the main development branch as the equator then a feature branch is the polar opposite of a release branch. Where the codeline policy for a release branch is aimed at providing maximum stability through low-risk changes, a feature branch has a policy aimed at volatile, high-risk changes. Instead of protecting the release from unwanted side-effects we're now protecting the main development pipeline from stalling for similar reasons.

The definition of 'feature' could be as small as a simple bug fix made by a single developer right up to an entire project involving many developers (the aforementioned project-level integration branch). Other terms that are synonymous are 'task branch' and 'private branch'. One suggests a narrower focus for the changes whilst the other promotes the notion of a single developer working in isolation. Either way the separation allows the contributor(s) to make changes in a more *ad hoc* fashion that suits their goal. As such they need not worry about breaking the build or even checking in code that doesn't compile, if that's how they need to work to be effective.

One common use for a feature branch is to investigate changes that are considered experimental in nature, sometimes called a spike [ExtremeProgramming]. This type of feature may well be discarded at the end of the investigation with the knowledge gained being the point of the exercise. Rather than pollute the integration branch with a load of code changes that have little value, it's easier to just throw the feature branch away and then develop the feature again in a 'cleaner' manner. Many version control systems don't handle file and folder renames very well and so this makes tracing the history across them hard. For example, during a period of heavy refactoring, files (i.e. classes) may get renamed and moved around which causes their history to become detached. Even if the changes are reverted and the files return to their original names the history can still remain divorced as the VCS just sees some files deleted and others added.

In some cases the changes themselves may be inherently risky, but it may also be that the person making the changes might be the major source of risk. New team members always need some time getting up to speed with a new codebase no matter how experienced they are. However, junior programmers will likely carry more risk than their more senior counterparts, therefore it might be preferable to keep their work at arms length until the level of confidence in their abilities (or the process itself

adapts) to empower them to decide for themselves how a change should best be made.

Once again it should be fairly apparent that what can mitigate some uses of feature branches is having a better development process in the first place. With a good automated test suite, pair programming, code reviews, etc. the feedback loop that detects a change which could destabilise the team will be unearthed much quicker and so headed it off before it can escalate.

What makes feature branches distasteful to many, though, is the continual need to refresh it by merging up from the main integration branch. The longer you leave it before refreshing, the more chance you have that the world has changed underneath you and you'll have the merge from hell to attend to. If the team culture is to refactor relentlessly then this will likely have a significant bearing on how long you leave it before bringing your own branch back in sync.

Frequently merging up from the main integration branch is not just about resolving the textual conflicts in the source code though. It's also about ensuring that your modifications are tested within the context of any surrounding changes to avoid the semantic conflicts described earlier. Whilst it might technically be possible to integrate your changes by just fixing any compiler warnings that occur in the final merge, you need to run the full set of smoke tests too (at a minimum) so that when you publish you have a high degree of confidence that your changes are sound.

Shelving

There is a special term for the degenerate case of a single-commit feature branch – shelving. If there is a need to suddenly switch focus and there are already changes in flight that you aren't ready to publish yet, some VCSs allow you to easily put them to one side until you're ready to continue. This is usually implemented by creating a branch based on the revision of the working copy and then committing any outstanding changes. When it's time to resume, the changes can be un-shelved by merging the temporary branch back into the working copy (assuming the ancestry allows it).

One alternative to shelving is to have multiple working folders all pointing at different branches. If you constantly have to switch between the development, release and production codebases, for example, it can be easier (and perhaps faster) to just switch working folders than to switch branches, especially now that disk space is so cheap.

Forking

The introduction of the Distributed Version Control System (D-VCS) adds another dimension to the branching strategy because a developer's machine no longer just holds a working set of changes, but an entire repository. Because it's possible to make changes and commit them to a local repo, the developer's machine becomes a feature branch in its own right. It is still subject to the same issues in that upstream changes must be integrated frequently, but it can provide far more flexibility in the way those changes are then published because of the flexibility modern D-VCSs provide.

No branch/feature toggle

Back in the days before version control systems were clever enough to support multiple threads of change through branches, there was just a single shared branch. This constraint in the tooling had an interesting side-effect that meant making changes had to be more carefully thought out.

Publishing a change that broke the build had different effects on different people. For some it meant that they kept everything locally for as long as possible and only committed once their feature was complete. Naturally this starts to get scary once you consider how unreliable hardware can be or what can go wrong every time you're forced to update your working folder, which would entail a merge. Corruption of uncommitted changes is entirely possible if you mess the merge up and have no backup to return to.

The other effect was that some developers learnt to break their work down into much more fine-grained tasks. In contrast they tried to find a way to commit more frequently but without making changes that had a high chance of screwing over the team. For example new features often involve some refactoring work to bring things into shape, the addition of some new code and the updating or removing of other sections. Through careful planning, some of this work can often be done alongside other people's changes without disturbing them, perhaps with some additional cost required to keep the world in check at all times. For instance, by definition, refactoring should not change the observable behaviour and so it must be possible to make those changes immediately (unanticipated performance problems notwithstanding).

This then is the premise behind using a single branch for development along with feature toggles to hide the functionality until it is ready for prime time. The notion of 'always be ready to ship' [c2-3] engenders an attitude of very small incremental change that continually edges the product forward. The upshot of this is that 'value' can be delivered continually too because even the refactoring work has some value and that can go into production before the entire feature might be implemented. Feature toggles are a mechanism for managing delivery whereas branches are a mechanism for managing collaboration. The desire to increase collaboration and deliver more frequently will usually lead to the use feature toggles as a way of resolving the tension created by partially implemented stories.

This method of development does not come easily though, it demands some serious discipline. Given that every change is published to the team and the build server straight away means that there must be plenty of good practices in place to minimise the likelihood of a bug or performance problem creeping in unnoticed. The practices will probably include a large, mostly automated test suite along with some form of reviewing/pairing to ensure there are many 'eyes' watching.

The way that the feature is 'toggled' can vary depending on whether its activation will be static (compile time) or dynamic (run time). From a continuous-testing point of view it makes far more sense to ensure any new feature is enabled dynamically otherwise there are more hoops to jump through to introduce it into the test suite. Doing it at runtime also helps facilitate A/B testing [Wikipedia-2] which allows old and new features to run side-by-side for comparison.

The nature of the toggle varies depending on what mechanisms are available, but either way the number of points in the code where the toggle appears should be kept to an absolute minimum. For example, instead of littering the code with **`#ifdef`** style pre-processor statements to elide the code from compilation it is preferable to have a single conditional statement that enables the relevant code path:

```
if (isNewFeatureEnabled)
  DoNewFeature();
```

The toggle could take the form of a menu item in the UI, an entry in a configuration file, an **`#ifdef`** compilation directive, a data file, an extra parameter in an HTTP request, a property in a message, the use of **`REM`** to comment in/out a command in a batch file, etc. Whatever the choice its absence will generally imply the old behaviour, with the new behaviour being the exception until it goes live for good. At that point it will disappear once again.

One side-effect of working with feature toggles is that there might be a clean-up exercise required at the end if it gets pulled or if it supplants another feature – this will happen after go live and so needs to be planned in. During development there will also be periods of time where 'unused code' exists in the production codebase because the feature hasn't been fully implemented yet. Whilst it's beneficial that others get early sight of the ongoing efforts they need to be sure not to delete what might seem to be 'dead code'.

The motivation for not branching is effectively to avoid merging at all. That won't happen, simply because you need to continually refresh your working folder and any update could require a merge. However, the likelihood that any conflicts will crop up should be greatly diminished. In particular 'noisy' refactorings can be easier to co-ordinate because the changes can be made, pushed and pulled by others with the minimum of fuss.

Hybrid approaches

The three core branching strategies are not in any way mutually exclusive. It's perfectly acceptable to do, say, the majority of development on the main integration branch with occasional feature branches for truly risky tasks and release branches to avoid getting stalled due to bureaucracy (e.g. waiting for the Change Review Board to process the paperwork).

Example: Visual Studio upgrade

Historically a tool like Visual C++ cannot be silently upgraded. Its project and solution data files are tied to a specific version and must match the version of the tool being used by the programmer. In the past this has created problems for larger teams where you all cannot just migrate at the same time without some serious groundwork. Aside from the project data file problems, in C++ at least, there is also the problem of the source code itself being compatible with the new toolchain. Visual C++ used to default to the non-standard scoping rules for a `for` loop meaning that the loop variable could leak outside the loop and into lower scopes. Bringing the codebase in line with the ISO standard also meant source code changes to be handled too.

When I tackled this with a medium sized team that were working on separate projects on multiple integration branches I had to use a combination of branching approaches as a big bang switchover was never going to work. Although the build and deployment process was somewhat arcane, the fact that multiple streams already existed meant that the parallelisation aspect was going to be less painful.

As part of an initial spike, I used a feature branch to investigate what I needed to upgrade the tooling and to see what the impact would be vis-à-vis source code changes. The end result of that were just some new build scripts to handle the tooling upgrade; everything else was ditched.

The next step was to bring as much of the existing codebase up to scratch by fixing the `for` loop scoping manually (where necessary) by inducing an extra scope (you just enclose the existing loop with another pair of braces). On one integration branch I upgraded the toolchain locally, fixed all the compiler errors and warnings, then reverted the toolchain upgrade, re-compiled with the current toolchain to verify backwards compatibility and finally committed just the source code changes. An email also went out too educating the other developers for the kinds of issues that might crop up in the future so that any new code would stand a better chance of being compatible at the final upgrade time.

Those code changes and the new upgrade scripts were then merged (cherry picked) into every integration branch so that each one could then be inspected and any new changes made since the original branch point occurred could made compatible too. At this point all the integration branches were in good shape and ready to migrate once we had ironed out the non-syntactic problems.

The next step was to verify that a build with the new toolchain worked at runtime too and so a new feature branch was taken from one of the integration branches which could be used to build and deploy the product for system testing. This allowed me to iron out any bugs in the code that only showed up with the new compiler behaviour and runtime libraries. Once fixed these changes could also be pushed across to the other integration branches so that all of the projects are now in a position to make the final switch.

My goal when doing this work was to avoid messing up any one project if at all possible. The uncertainty around the delivery schedule of each project meant that I didn't know at the start which one was going to be the best to use to 'bed in' the upgrade, so I made sure they were all candidates. Whilst it felt wasteful to continuously throw changes away (i.e. the upgraded project files) during the migration process, the painless way the final switchover was done probably meant more time was saved by my teammates in the long run.

Gatekeeper workflows

In recent times one particular hybrid approach has sprung up that attempts to formalise the need to pass some review stage before the change can be accepted into the main codeline. This review process (the aforementioned gatekeeper) can either be done entirely automatically via a continuous integration server; or via some form of manual intervention after the build server has given the change the green light.

This style of workflow is the opposite of the 'no branching' approach because it relies on not letting anyone commit directly to the integration branch. Instead each developer gets their own feature branch into which they make their changes. Any time a developer's branch has changed the continuous integration server will attempt to merge it with the integration branch, then build and run the test suite.

If that process succeeds and an automated gatekeeper is in play then the merge is accepted and the main branch is advanced. If a manual gatekeeper is involved, perhaps to review the change too, they can perform that knowing it has already passed all the tests which helps minimise wasting time on reviewing low quality changes. If the change fails at any stage, such as the merge, build or test run then the developer will need to resolve the issue before going around the loop again.

Whilst this has the benefit of ensuring the main development branch is always in a consistent state, build-wise, it does suffer from the same afflictions as any other feature branch – a need to continuing merge up from the integration branch. That said, where the no branching approach relies heavily on diligence by the programmer these workflows look to leverage the tooling in the continuous integration server to try and minimise the overhead. For example, just as they can automatically merge a feature branch to the integration branch on a successful build, they can also merge any changes from the integration branch back out to any feature branches when updated by the rest of the team. The net effect is that programmers can spend less time worrying about 'breaking the build' because they never contribute to it unless their changes are already known to be coherent.

This style of workflow could also be combined with feature toggles to aid in delivering functionality in a piecemeal fashion.

Summary

The goal of this article was to distil the folklore surrounding branching strategies down into the three key patterns – no branching, branching for a feature and branching for a release. We identified the policies that commonly drive the choice of strategy and the forces, often organisational in nature, that can push us in that direction. Finally we looked at how and when it might be suitable to combine them rather than blindly try to stick to the same strategy all the time, and how tooling is beginning to help reduce some of the overhead. ■

Acknowledgements

A big thumbs-up from me goes to Mike Long, Jez Higgins and the *Overload* review collective for their valuable input. And mostly to Fran, the *Overload* editor, for her patience as I made some significant last minute edits.

References

[Berczuk02] *Software Configuration Management Patterns*, Stephen P. Berczuk with Brad Appleton, 2002, Chapter 12: Codeline Policy

[c2-1] http://c2.com/cgi/wiki?ImpedanceMismatch

[c2-2] http://c2.com/cgi/wiki?CherryPicking

[c2-3] http://c2.com/cgi/wiki?AlwaysBeReadyToShip

[ExtremeProgramming] http://www.extremeprogramming.org/rules/spike.html

[Fowler] http://martinfowler.com/bliki/SemanticConflict.html

[Wikipedia-1] http://en.wikipedia.org/wiki/Cargo_cult_programming

[Wikipedia-2] http://en.wikipedia.org/wiki/A/B_testing

Beauty in Code

Attractiveness matters. Adam Tornhill uses a Clojure example to show how generalisation can be more beautiful than special cases.

The challenge of all software design is to control complexity. Less complexity means that our programs are easier to understand, reason about and evolve. This article shows how we can use beauty as a mental tool for that purpose. Starting in the field of the psychology of attractiveness, we'll expand its theories on physical beauty to also cover code. In the process we'll learn a bit about Clojure, meta-programming and destructuring.

The scandal

Beauty is a controversial subject. Research has shown that, everything else equal, a beautiful waitress gets more tips than a less attractive colleague. The politician with the better looks gets more votes than more ordinary opponents. Not even our judicial systems are immune to the power of beauty since offenders with the right type of facial features receive milder judgements. Beauty is indeed scandalous. And it's a scandal backed by solid science [Renz06].

Attractiveness is important to us. Often more important than we'd like to admit or even are aware of at a conscious level. So before we dive into code we'll make a short detour into attractiveness psychology.

Beauty is average

At the end of the 80s, scientist Judith Langlois performed an interesting experiment [Langlois90]. Aided by computers, she developed composite pictures by morphing photos of individual faces. As she tested the attractiveness of different pictures on a group, the results turned out to be both controversial and fascinating. Graded on physical attractiveness the composite pictures won. And they won big.

The idea of beauty as averageness seems counterintuitive. In our field of programming, I'd be surprised if the average enterprise codebase would receive praise for its astonishing beauty. But beauty is not average in the sense of ordinary, common or typical. Rather, beauty lies in the mathematical sense of averageness found in the composite faces.

The reason the composite pictures won is that individual imperfections get evened out with each additional morphed photo. As such beauty is more of a negative concept defined by what's absent rather than what's there. As the theory goes, it's a preference shaped by evolution to guide us away from bad genes towards good ones.

Beautiful code

Translated to our world of software the good genes theory means consistency. Beautiful code has a consistent level of expression free from special cases. Just as deviations from the mathematical averageness makes a face less attractive, so does any coding construct that deviates from the main flow. Classic examples of special cases include conditional logic, explicit looping constructs or differing programming models for sequential vs concurrent code.

These constructs all signal bad genes in our programs. With beauty as a design principle we steer away from such special cases towards solutions that are easier to understand and grow.

Adam Tornhill With degrees in engineering and psychology, Adam tries to unite these two worlds by making his technical solutions fit the human element. While he gets paid to code in C++, C#, Java and Python, he's more likely to hack Lisp or Erlang in his spare time.

A case study in named arguments

To classify code as beautiful, there has to be some tension in the design. Perhaps that tension comes from an alternative solution that, while initially tempting, would turn out less than optimal. Frequently, beauty in code arise from a hard problem whose solution is made to look easy and apparent.

My favourite example comes from Clojure. Clojure has grown at a rapid rate, but without losing track of its core tenets of simplicity and programmer power. While Clojure included novel concurrency constructs and strong meta-programming support from its very start, a more fundamental feature was missing: named arguments.

Named arguments are familiar to programmers in languages like Python and C#. The idea is that instead of relying on the positional placement of function arguments, the programmer specifies the name of the parameters. Based on the names, the compiler binds the objects to the right argument.

Say we have a C# method to fire at objects:

```
void Fire(Location at, Weapon weapon)
{
  // implementation hidden as a
  // safety precaution
}
```

Thanks to named arguments, a client can invoke this method in any of the following three ways:

1. Positional:
   ```
   Fire(asteroid.LastObservation, ionCannon);
   ```
2. Named arguments, same order:
   ```
   Fire(at: asteroid.LastObservation,
        weapon: ionCannon);
   ```
3. Named arguments, arbitrary order:
   ```
   Fire(weapon: ionCannon,
        at: asteroid.LastObservation);
   ```

Named arguments serve to make the code on the call site more expressive. They also allow us to write intention revealing code by highlighting the core concept in the current context; if we write code for firing a weapon, prefer alternative #3. If we instead focus on shooting down asteroids, alternative #2 is a better fit.

The fire method is straightforward to translate into Clojure. We use `defn` to define a named function:

```
(defn fire
  [at-location, weapon]
  ; the implementation goes here...
)
```

On the call site, we fire at the location of asteroids as:

```
(fire asteroid-location ion-cannon)
```

In older versions of Clojure that was it. I couldn't name my arguments. But Clojure is a member of the Lisp family of programming languages and shares their characteristic of code as data. With that concept there's no limit to what we can do with the resulting meta-programming facilities. Lack a feature? Well, go ahead and implement it. In Clojure the programmer has the power of the language designer.

This aspect of Clojure is possible since Clojure code is expressed using its own data structures. Consider the code snippet above. The parenthesis around the **fire** function delimits a list. The square brackets represent a vector of two elements (the function arguments **at-location** and **weapon**).

Besides bending your mind in interesting ways, meta-programming allows the developer to hook into the compiler and write code that generates code. From a pragmatic perspective, it means that Clojure can grow without any changes to the core language itself. This is a radical difference to popular languages in use today.

Armed with meta-programming support, the early Clojure community developed an extension to the language: **defnk**. It was intended to complement Clojure's standard **defn** form to define functions. In addition, it allowed clients to use named arguments similar to the C# code above.

Having two different constructs to express the same basic thing affects our reasoning abilities. The largest cost is not the extra learning per se. The price we pay is during code reading. Suddenly we have one more concept to keep in our head. There's nothing attractive about that.

Towards beauty through destructuring

defnk never made it into Clojure. Instead Rich Hickey, Clojure's creator,made an interesting design choice. He decided to extend the usage of an existing feature to cover named arguments too. That feature was destructuring.

Destructuring is a distant relative to pattern matching as found in several functional programming languages (see for example: Fogus & Houser [Fogus11]). The feature makes it more concise to pull apart composite data structures into individual parts. Let's pretend for a while that C# supported destructuring. Listing 1 is how it could look.

Identified by the name of the keys, destructuring pulls out the associated values into the provided variables. At this point you're probably relieved to learn that I don't have any involvement in C#'s evolution. But let's leave my language design skills aside and translate the code to Clojure (see Listing 2). It's similar to my fantasy code in Listing 1. It also has the unmistakable advantage of being valid code.

By extending destructuring to function parameters, our fire function using plain **defn** would turn to this:

```
var incrediblyImportantNumbers =
  new Dictionary<string, double>
  {
    {"PI", 3.14},
    {"GoldenRatio", 1.6180339887},
    {"RubiksCubeCombos", 43252003274489856000.0}
  };
// Now, pretend we have got destructuring to
// pull apart the Dictionary:
var {PI, GoldenRatio, RubiksCubeCombos} =
    incrediblyImportantNumbers;
Console.WriteLine(PI + ", " + ", " + GoldenRatio +
    ", " + RubiksCubeCombos);
// Would print: 3.14, 1.6180339887,
// 43252003274489856000.0
```

Listing 1

```
; Define a map of key-value pairs
; (aka dictionary):
(def incredibly-important-numbers
   {:PI 3.14
    :golden-ratio 1.6180339887
    :rubiks-cube-combos 43252003274489856000})
; Destructure the map into symbols, one for
; each key.
; First the destructuring pattern...
(let [{:keys [PI golden-ratio rubiks-cube-
combos]}
       ; ..then the dictionary to destructure:
       incredibly-important-numbers]
       (println PI  ", " golden-ratio ", "
          rubiks-cube-combos))
  ; Will print: 3.14, 1.6180339887,
  ; 43252003274489856000
```

Listing 2

```
(defn fire
  ; destructuring pattern on function arguments:
  [& {:keys [at-location weapon]}]
  ; the implementation goes here...
```

The destructuring patterns form a mini language inside Clojure. With the extended scope of that mini language, Clojure programmers got the power of named arguments without the need to learn a new language construct. Better yet, we could fire weapons as:

```
(fire :weapon ion-cannon
      :at-location asteroid-location)
```

Similar to the process behind the attractive composite pictures this design evened-out individual differences between use cases. This has several benefits. Since we got rid of multiple ways to define functions, the resulting solution maintains conceptual integrity. From a programmer perspective we can re-use our mastery of how destructuring works and apply that knowledge to function argument lists too. This is a big advantage since there are fewer concepts to keep in our head.

Outro

Every design choice implies a trade-off. In beauty we have a mental tool that guides us towards evolutionary fit programs. As this short Clojure story illustrates, to generalize an existing concept is a more attractive path than to introduce a special case. It's a technique we can use to drive us towards consistency independent of the programming language we work in.

Beautiful code looks obvious in retrospect but may well require hard mental work up front. It's an investment that pays-off since code is read more frequently than it's written. The decisions we make in our programs today exert their influence over time. Guided by beauty we pave an attractive way for future maintainers. ■

Thanks

Thanks for Björn Granvik, Mattias Larsson, Tomas Malmsten and the Diversify/Webstep team for valuable feedback! Also thanks to the *Overload* editorial team for their corrections!

References

[Fogus11] Fogus, M., & Houser, C. (2011) *The Joy of Clojure*. Manning.

[Langlois90] Langlois, J. H., & Roggman, L. A. (1990) *Attractive faces are only average*.

[Renz06] Renz, U. (2006). *Schönheit*. Berliner Taschenbuch Verlag.

The Code Is Not the Most Important Thing

Choose the right sort of problem, focus on strategies for solving it, and the code will come easily. Seb Rose teaches kids to code.

I want to tell you a story. Two stories, actually.

A few years ago I found myself in one of those huge, soulless, air-conditioned conference halls, listening to one of those keynote speeches on a subject that had no discernible relevance to my work. But despite my preconceptions, I found myself surprisingly moved by the presentation. The speaker was inventor Dean Kamen, and he was talking about something called First Lego League.

I was inspired. Returning to the UK, I chased down our local FLL and registered. I coerced my children, aged ten and twelve, into participating. I put notices around the local junior schools, gathering another eight local offspring. I booked a room in a local hall one night a week for the rest of the year, bought the lumber needed to build the competition table, and waited for the kit to arrive.

The kit, right. I see I'm getting ahead of myself. I'd better back up and explain a bit about the FLL.

The First Lego League

The First Lego League [FLL] is an annual, international competition for teams of 9- to 16-year-olds that involves programming Lego MindStorms to solve challenges, doing a piece of original research, and presenting the results of that research. You can read more about it at the website. It's interesting reading, really.

Anyway, about the kit. Each year FLL chooses a theme of global significance (in 2010 it was Biomedical Engineering) and devises a number of robotic challenges around it. And they send you this kit to get you started. The competition board is 8' x 4' and they provide a roll-out map to use as the base for your constructions, as well as a box of Lego bits and pages of instructions about how to put together the Lego challenges. Just building all of the challenges took our group 2 whole sessions.

Once the challenges are assembled and laid out on the board, it's time to start designing and programming your robot. The robot construction rules are strict – only Lego-manufactured material allowed, no more than 4 motors, no more than 1 of each sensor, no non-standard software environments – and you only get 2½ minutes to complete as many challenges as you can. Whenever the robot returns to base, a small area at one corner of the board, you're allowed to touch it, modify its attachments, start a different program. At all other times it must be completely autonomous (NOT remote controlled) and any interference with it leads to penalty points.

Seb Rose is an independent software developer, trainer and consultant based in the UK. He specialises in working with teams adopting and refining their agile practices, with a particular focus on automated testing. He first worked as a programmer in 1980 writing applications in compiled BASIC on an Apple II. He has worked with many mainstream technologies since then, and is a regular conference speaker, contributor to O'Reilly's *97 Things Every Programmer Should Know* and co-author of the Pragmatic Programmer's *The Cucumber-JVM Book*.

So there I was, with ten 9- to 12-year-olds with no engineering or programming experience. Mercifully, the role of coach in FLL, they explain, is not to teach, but to enable. Good. So I printed out the challenges, loaded the Lego programming environment onto some school laptops, and asked them to pick a few challenges to have a go at. There were 17 to choose from, and no way to accomplish all of them in the short time available.

There was teaching involved, but once they had selected the challenges, it was pretty straightforward. We started with the engineering, because that was a more tractable problem, and you need to have an idea of the robot's physical characteristics before trying to control it with software. Then came the challenge of introducing them to software. The Lego programming environment is fully graphical – you drag action, sensor, conditional, and branching 'blocks' around, linking them together to achieve the required behaviour. We started with simple, dead-reckoning approaches – 'turn both wheels for 8 revolutions at 5 revolutions per minute,' 'turn left 90 degrees,' 'lift arm 45 degrees' – and just that is enough to get a lot of the challenges done.

What the team noticed, though, was that this style of control wasn't very robust. It was susceptible to small variations in the robot's initial position. The state of the batteries affected the distance the robot travelled. Even with identical starting conditions, the end position of the robot could differ significantly. This is when I introduced them to the sensors.

The most useful sensor was the light sensor. They used this to detect colour changes on the map and trigger a change in behaviour of the robot. Different groups of children would work on different parts of the robot's route, and these would then be joined together by 'blocks' switched by the sensors. This was particularly effective for the 'pill dispenser' challenge, where the robot needed to push a panel to dispense 2 Lego 'pills', but leave the third 'pill' in the dispenser. There were handy black markings on the map that could be used to count how far the robot had travelled, and hence how many 'pills' had been dispensed.

Another useful sensor was the push button, which was useful to ensure the robot kept the right distance from obstacles. We never found a use for the ultrasonic sensor-it was just too unreliable. And the microphone would probably have been thought of as a remote control, so we never even tried that.

What interested me about the whole experience was that we rarely talked about programming. The team was always trying to solve a problem – sometimes the solution was a new attachment for the robot, sometimes it was a new behaviour. They quickly picked up control and conditionals. Looping was harder, and I don't think they ever really got there – they were much happier copying and pasting sections of the program to get the number of iterations they wanted.

Second story

The local group lost its funding the next year and FLL in Scotland lapsed until this year, when a local company, Lamda Jam [JAM] brought it back. This year, as part of the local British Computer Society [BCS] branch I

What interested me about the whole experience was that we rarely talked about programming

was asked to judge the robot competition. There were seven schools competing – two high schools and five junior schools.

One difference in this competition was that, although there was a lot of inventive engineering, there was no use of sensors at all. Even the best teams were controlling their robots using dead reckoning alone. One of the side effects of this was that scores for the same team, using the same programs, varied wildly. Each team played 3 heats, and the winning team's scores varied between 125 and 348.

So what did I learn from these two experiences in teaching kids to code?

Having seen FLL from both the participant and judging sides I'm confident that the FLL can be an excellent vehicle for getting children to learn to code. Because there are so many different angles to the challenge, it's easy for the team to do a little bit of everything without getting overwhelmed by any one aspect. The programming part of the challenge does introduce them to simple constructs, and with the right coach this can go a lot further. I can easily see a team building on its experience over a number of years, with some of the members eventually getting to be quite sophisticated practitioners.

But the counterintuitive insight was that it's not so much about the code.

A comment in a thread on the accu-general [ACCU] mailing list with the subject line 'Programming for teenagers' captured nicely what I learned from my experiences:

> I think that finding an interesting and reasonable-sized project that can be expanded upon is more important than the choice of tools and environment. That is the hook to keep her interested and search for ways to learn more, and I also think the very first visible result must be reached quickly.

A good project, room to grow, and getting visible results quickly. Those are key. And one more thing: a good mentor. That, of course, would be you.

I encourage anyone with even the slightest interest in introducing children to programming to take a look at what FLL has to offer. You may end up doing more Lego than programming, but you'll be giving the children exactly the right role model – us. Because we're people who solve problems and get results quickly. ■

References

[ACCU] http://accu.org/mailman/listinfo/accu-general

[BCS] http://bcs.org

[FLL] http://www.firstlegoleague.org

[JAM] http://lambdajam.org

Originally published in by PragPub *Kids To Code* (a free special issue available from http://theprosegarden.com/).

Stop the Constant Shouting

CONSTANTS often shout. Jonathan Wakely considers why this happens in C and what the alternatives are in C++.

It's common in C and C++ to use entirely uppercase names for constants, for example:

```
const int LOWER = 0;
const int UPPER = 300;
const int STEP = 20;
enum MetaSyntacticVariable { FOO, BAR, BAZ };
```

I think this is a terrible convention in C++ code and if you don't already agree I hope I can convince you. Maybe one day we can rid the world of this affliction, except for a few carefully-controlled specimens kept far away where they can't hurt anyone, like smallpox.

Using uppercase names for constants dates back (at least) to the early days of C and the need to distinguish symbolic constants, defined as macros or enumerators, from variables:

> Symbolic constant names are conventionally written in upper case so they can be readily distinguished from lower case variable names. [Kernighan88]

The quoted text follows these macro definitions:

```
#define LOWER 0    /* lower limit of table */
#define UPPER 300  /* upper limit */
#define STEP  20   /* step size */
```

This convention makes sense in the context of C. For many years C didn't have the **const** keyword, and even today you can't use a **const** variable where C requires a constant-expression, such as declaring the bounds of a (non-variable length) array or the size of a bitfield. Furthermore, unlike variables, symbolic constants don't have an address and can't be assigned new values. So I will grudgingly admit that macros are a necessary evil for defining constants in C, and distinguishing them can be useful and a consistent naming convention helps with that. Reserving a set of identifiers (in this case, 'all names written in uppercase') for a particular purpose is a form of namespace, allowing you to tell at a glance that the names **STEP** and **step** are different and, by the traditional C convention, allowing you to assume one is a symbolic constant and the other is a variable.

Although some form of ad hoc namespace may be useful to tell symbolic constants and variables apart, I think it's very unfortunate that the traditional convention reserves names that are so VISIBLE in the code and draw your ATTENTION to something as mundane as symbolic constants. An alternative might have been to always use a common prefix, say **C_**, for symbolic constants, but it's too late to change nearly half a century of C convention now.

C's restrictions on defining constants aren't present in C++, where a **const** variable (of suitable type) initialised with a constant expression is itself a constant expression and where **constexpr** functions can produce compile-time constants involving non-trivial calculations on other constants. C++ also supports namespaces directly in the language, so the constants above could be defined as follows and referred to as **FahrenheitToCelsiusConstants::step** instead of **STEP**:

```
namespace FahrenheitToCelsiusConstants {
  // lower & upper limits of table, step size
  enum Type { lower=0, upper=300, step=20 };
}
```

That means C++ gives you much better tools than macros for defining properly typed and scoped constants.

> Macros are very important in C but have far fewer uses in C++. The first rule about macros is: Don't use them unless you have to. [Stroustrup00]

There are good reasons for avoiding macros apart from the fact that C++ provides higher-level alternatives. Many people are familiar with problems caused by the **min** and **max** macros defined in `<windows.h>`, which interfere with the names of function templates defined in the C++ standard library. The main problem is that macros don't respect lexical scoping, they'll stomp over any non-macro with the same name. Functions, variables, namespaces, you name it, the preprocessor will happily redefine it.

> Preprocessing is probably the most dangerous phase of C++ translation. The preprocessor is concerned with tokens (the "words" of which the C++ source is composed) and is ignorant of the subtleties of the rest of the C++ language, both syntactic and semantic. In effect the preprocessor doesn't know its own strength and, like many powerful ignoramuses, is capable of much damage. [Dewhurst02]

Stephen Dewhurst devotes a whole chapter to gotchas involving the preprocessor, demonstrating how constants defined as macros can behave in unexpected ways, and 'pseudofunctions' defined as macros may evaluate arguments more than once, or not at all. So given that macros are less necessary and (in an ideal codebase) less widely-used in C++, it is important when macros *are* used to limit the damage they can cause and to draw the reader's attention to their presence. We can't use C++ namespaces to limit their scope, but we can use an ad hoc namespace in the form of a set of names reserved only for macros to avoid the problem of clashing with non-macros and silently redefining them. Conventionally we use uppercase names (and not single-character names, not only are short names undescriptive and unhelpful for macros, single-character names like **T** are typically used for template parameters).

> Also to warn readers, follow the convention to name macros using lots of capital letters. [Stroustrup]
>
> By convention, macro names are written in uppercase. Programs are easier to read when it is possible to tell at a glance which names are macros. [GCC]

Using uppercase names has the added benefit of SHOUTING to draw ATTENTION to names which don't obey the usual syntactic and semantic rules of C++.

Jonathan Wakely Jonathan's interest in C++ and free software began at university and led to working in the tools team at Red Hat, via the market research and financial sectors. He works on GCC's C++ Standard Library and participates in the C++ standards committee.

it is important when macros are used to limit the damage they can cause and to draw the reader's attention to their presence

> Do **#undefine** macros as soon as possible, always give them **SCREAMING_UPPERCASE_AND_UGLY** names, and avoid putting them in headers. [Sutter05]

When macro names stand out clearly from the rest of the code you can be careful to avoid reusing the name and you know to be careful using them e.g. be aware of side-effects being evaluated twice:

```
#define MIN(A,B) (A) < (B) ? (A) : (B)
const int limit = 100;
// ...
return MIN(++n, limit);
```

But if you also use all-uppercase names for non-macros then you pollute the namespace. You no longer have the advantage of knowing which names are going to summon a powerful ignoramus to stomp on your code (including the fact that your carefully-scoped enumerator named **FOO** might not be used because someone else defined a macro called **FOO** with a different value), and the names that stand out prominently from the rest of the code might be something harmless and mundane, like the bound of an array. Constants are pretty dull, the actual logic using them is usually more interesting and deserving of the reader's attention. Compare this to the previous code snippet, assuming the same definitions for the macro and constant, but with the case of the names changed:

```
return min(++n, LIMIT);
```

Is it more important to note that you're limiting the return value to some constant **LIMIT**, rather than the fact that **n** is incremented? Or that you're calling **min** rather than **max** or some other function? I don't think **LIMIT** should be what grabs your attention here, it doesn't even tell you what the limit is. It certainly isn't obvious that **n** will be incremented twice!

So I'd like to make a plea to the C++ programmers of the world: stop naming (non-macro) constants in uppercase. Only use all-uppercase for macros, to warn your readers and limit the damage that the powerful ignoramus can do. ■

References

[Dewhurst02] *C++ Gotchas*, Stephen C. Dewhurst, Addison Wesley, 2002.

[Kernighan88] *The C Programming Language*, Second Edition, Brian W. Kernighan & Dennis M. Ritchie, Prentice Hall, 1988.

[GCC] 'The GNU Compiler Collection: The C Preprocessor', Free Software Foundation, 2014, http://gcc.gnu.org/onlinedocs/cpp/Object-like-Macros.html#Object-like-Macros

[Stroustrup00] *The C++ Programming Language*, Special Edition, Bjarne Stroustrup, Addison-Wesley, 2000.

[Sutter05] *C++ Coding Standards*, Herb Sutter & Alexei Alexandrescu, Addison-Wesley, 2005.

Minimal Overhead for Multiple Interfaces

Using multiple interfaces can slow things down. Daniel Gutson and Pablo Oliva present an alternative.

When working on embedded systems, using C++, it is often desirable to have a single class implementing multiple interfaces. The naive way of doing this (i.e., using multiple inheritance from a set of interfaces), while perfectly reasonable for normal situations, does pose some performance and memory problems for its use on embedded systems.

Given these costs, an approach used in embedded systems is to do away with interfaces altogether, thus saving all the implementation overheads. But this also leads to more coupled designs for object interaction, since method invocations occur on concrete objects instead of on interfaces, and it might also lead to code duplication and a loss of maintainability.

We propose a technique to use interfaces within the tight constraints of an embedded environment, and discuss its merits against alternative solutions to the problem.

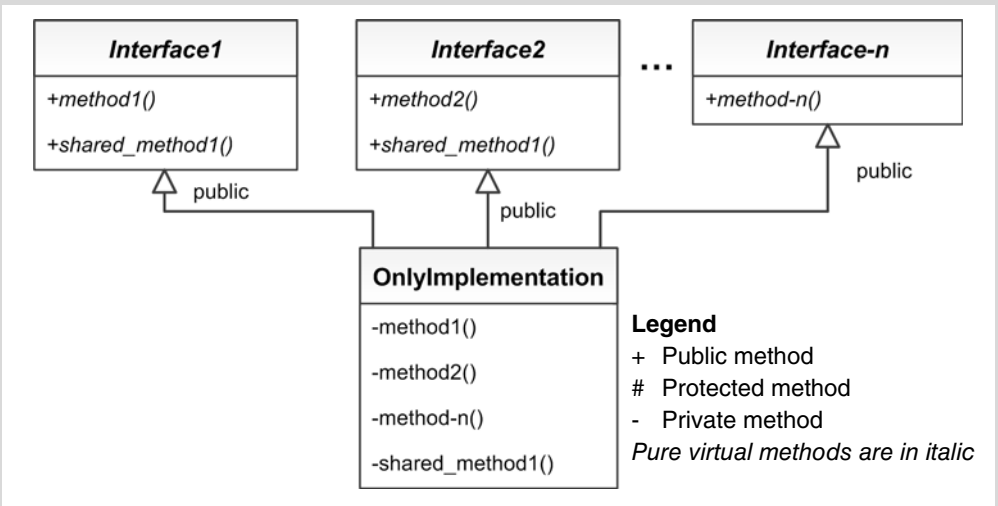

Figure 1

Use scenario

During the development process, it is often required that one class must implement multiple interfaces; in other words, a class must be developed that can receive different sets of messages from different types of users. It is worth noting that the term *interface* is used as defined in [GoF94], "The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond." This is a narrower definition of interface than the usual one, since it does not require the interface to have many implementations. In fact, the technique proposed here only works for single-implementation interfaces.

In embedded systems' development, there are often many restrictions in both the amount of memory available and the amount of processing resources available; these limitations make some of the most common ways to perform this multiple implementation undesirable. We will review some common solutions, along with their specific drawbacks for embedded systems.

Daniel Gutson is the Chief Engineering Officer and Director of the Special Projects Division at Taller Technologies, Argentina, and Informatics Director of the NGO FuDePAN. He has contributed to the GNU toolchain and has also submitted proposals to the C++ (WG21) Committee. He can be contacted at daniel.gutson@tallertechnologies.com

Pablo Oliva is a Software Engineer of the Special Projects Division at Taller Technologies Argentina. Along with Daniel, he is a collaborator at FuDePAN and contributes to the development of programs to aid people living with AIDS. He can be contacted at pablo.oliva@tallertechnologies.com

Alternative solutions

The typical solution for this problem is 'dynamic polymorphism' [Strachey67]: to create an interface (with its methods declared as *virtual*) for each user type, and to create a concrete class that inherits from all interfaces and implements their methods. Users of the class then access the base class object via pointers to the corresponding interface. Within embedded environments, where memory is scarce (which means that it is preferable to avoid having vtables) and there may be strenuous time constraints (which makes indirections undesirable due to their time cost), dynamic polymorphism is not an optimal solution. Figure 1 shows a typical solution's UML diagram.

Two techniques related to the use of dynamic polymorphism are using either *dynamic_cast* or the *pimpl* idiom [Coplien92] (where a pointer to the implementing class is kept in the interface), but any of these two options would also incur in the memory and processing overhead caused by the usage of virtual dispatching, because both alternatives require pointer indirections to be used. Since avoiding indirection costs (both in performance and in memory) is our goal, we will not analyze these alternatives any further.

Other alternatives considered were: the ROLE OBJECT design pattern [Baumer97] and relying on compiler's devirtualization [Namolaru06].

The ROLE OBJECT pattern imposes a significant overhead on runtime performance (because roles are resolved dynamically), which renders it unusable for embedded environments.

Devirtualization is the process of resolving virtual method calls on compile time, thus saving the dynamic dispatch overhead. It is not required by the standard, and there is no guarantee that it will be available on any toolchain chosen to build a project. Our technique, while far more limited in scope, can be used with any standard-compliant toolchain.

The proposal is to reverse the normal order of inheritance, dividing the responsibilities

The technique we present is an alternative that has no pointer indirection overheads (thus saving the memory and processing costs incurred by using pointers), but that still allows interfaces to be used.

The technique

The proposal is to reverse the normal order of inheritance, dividing the responsibilities between three kinds of classes.

1. *Base implementation*: this is the base class for the entire hierarchy. It contains all the interfaces' method implementations. Its constructor is protected, to prevent instantiations of this class.
2. *Interfaces*: there is one of these classes for every type of user. One of the interfaces inherits from base implementation, and then every new interface inherits from the most derived interface thus far. All inheritances are protected, and so are all constructors. Each interface exposes its methods within its public part with *using* declarations.
3. *Getter implementer*: this is the last class in the hierarchy, and it inherits privately from the most derived interface. It is the only class in the entire hierarchy that can be instantiated, and it provides a getter method for each one of the interfaces in the hierarchy.

Figure 2 is the technique's UML diagram.

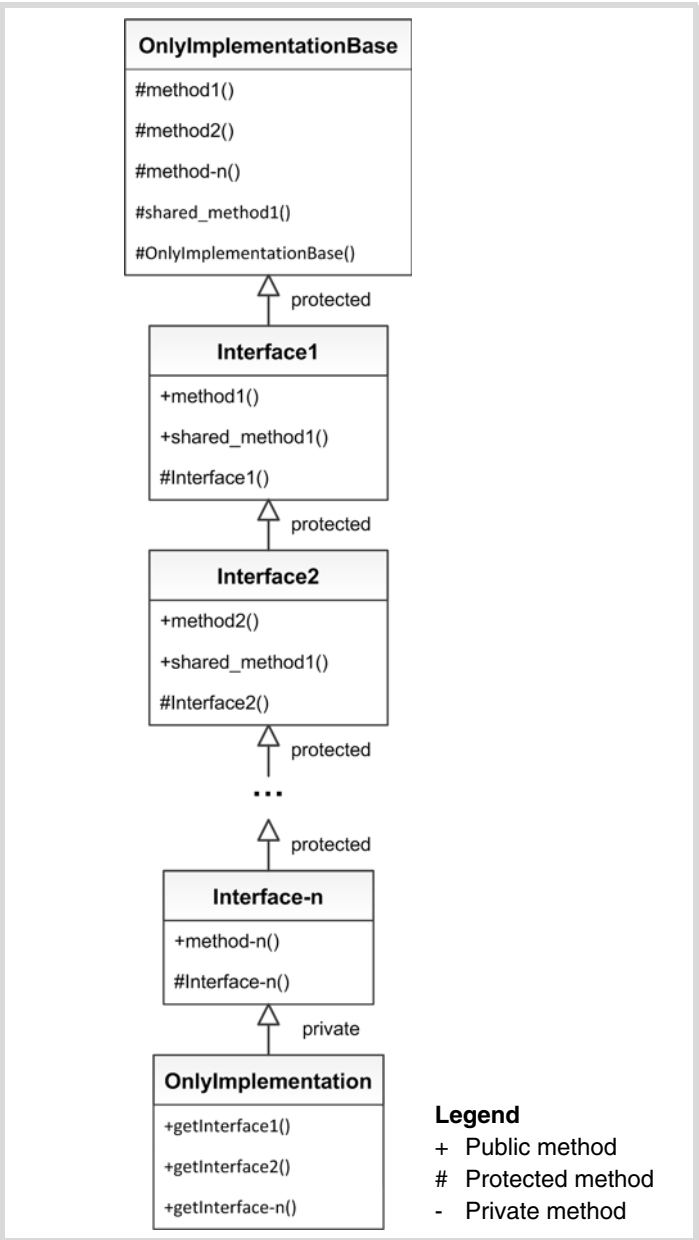

Figure 2

An example

Let there be two interfaces, `Writer` and `Reader`; `Writer` shows the `write` and `close` methods, and `Reader` shows `read` and `close` methods. `OnlyImplementation` implements both interfaces.

Using the typical solution (dynamic polymorphism) would yield something similar to Listing 1.

A usage example would be as in Listing 2.

In this design, the interfaces are base classes and implementation is on a derived class.

The previous example, adapted to the technique, is shown in Listing 3.

A usage example would be as in Listing 4.

Two C++11 features (`=default` for constructors [Crowl07] and `final` on the most derived class [ISO/IEC 14882-2011]) are used, but they both have equivalent expressions in C++03.

As you may see, the overhead on the user's side is an invocation of the getter method to get the desired interface. On the developer's side, the code is more contrived, but it does not incur in any type of pointer indirection overhead.

Conclusion

This technique was created to allow the use of multiple interfaces even under very rigorous performance and memory constraints. As it was shown, the overheads associated with all the other alternatives that were considered have been avoided.

The drawback of the technique is that there is no automatic upcast from implementation to interfaces, thus forcing the use of getters for the various interfaces. These calls to getters could be inlined to minimize their performance impact, but they do place an additional burden on the developer.

It should be stressed that this technique is only useful when the implementation is unique; when multiple implementations of one of the interfaces are necessary, this technique cannot be used. While this is a significant limitation, it is a frequent situation in embedded systems, where

This technique was created to allow the use of multiple interfaces even under very rigorous performance and memory constraints

```
struct WriterInterface
{
    virtual void write() = 0;
    virtual void close() = 0;
};
struct ReaderInterface
{
    virtual void read() = 0;
    virtual void close() = 0;
};
class OnlyImplementation : public
WriterInterface, public ReaderInterface
{
private:
    virtual void read()  { /* ... */ }
    virtual void write() { /* ... */ }
    virtual void close() { /* ... */ }
};
```

Listing 1

```
void writerUser()
{
  OnlyImplementation oi;
  WriterInterface* const wif = &oi;
  wif->write();
  wif->close();
}
```

Listing 2

single implementations of interfaces are not uncommon; thus we consider that this technique, albeit limited in scope, helps to solve a family of problems with a better trade-off than its alternatives. ■

Acknowledgements

We would like to thank Fabio Bustos, Fernando Cacciola and Angel Bustamante for their comments and suggestions. We would also like to thank the editing team of *Overload* for their helpful reviews.

References

[Baumer97] D. Bäumer, D. Riehle, W. Sibersky, M. Wulf (1997) 'The Role Object Pattern' http://st-www.cs.illinois.edu/users/hanmer/PLoP-97/Proceedings/riehle.pdf

[Coplien92] James O. Coplien (1992) *Advanced C++ Programming Styles and Idioms*, Addison-Wesley

[Crowl07] Crowl, Lawrence. *Defaulted and Deleted Functions*.

[GoF94] E. Gamma, R. Helm, R. Johnson, J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, First Edition, 1994.

[ISO/IEC 14882-2011] ISO/IEC 14882-2011. *Programming Languages – C++*, Third Edition, 2011.

```
class OnlyImplementationBase
{
protected:
  void read()  { /* ... */ }
  void write() { /* ... */ }
  void close() { /* ... */ }
  OnlyImplementationBase() = default;
};
class WriterInterface : protected
OnlyImplementationBase
{
public:
    using OnlyImplementationBase::write;
    using OnlyImplementationBase::close;
protected:
    WriterInterface() = default;
};
class ReaderInterface : protected WriterInterface
{
public:
    using OnlyImplementationBase::read;
    using OnlyImplementationBase::close;
protected:
    ReaderInterface() = default;
};
class OnlyImplementation final : private
ReaderInterface
{
public:
    ReaderInterface& getReader() { return *this; }
    WriterInterface& getWriter() { return *this; }
};
```

Listing 3

```
void writerUser()
{
  OnlyImplementation oi;
  WriterInterface& const wif = oi.getWriter();
  wif.write();
  wif.close();
}
```

Listing 4

[Namolaru06] M. Namolaru, 'Devirtualization in GCC', ols.fedoraproject.org/GCC/Reprints-2006/namolaru-reprint.pdf

[N2346= 07-0206]. 'Programming Language C++' Evolution Working Group.

[Strachey67] C. Strachey 'Fundamental Concepts in Programming Languages' http://www.itu.dk/courses/BPRD/E2009/fundamental-1967.pdf