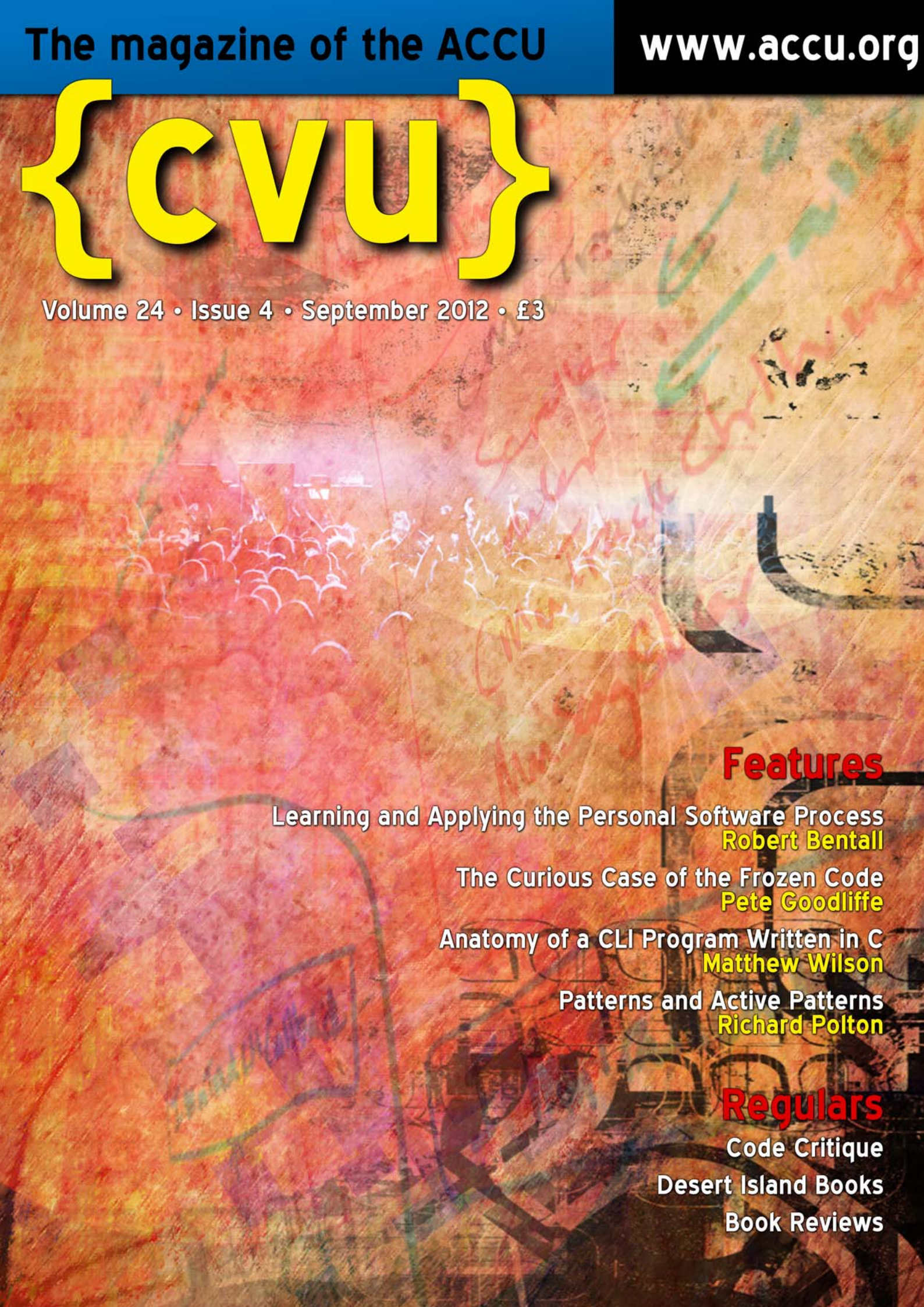
The magazine of the ACCU

www.accu.org

{cvu}

Volume 24 • Issue 4 • September 2012 • £3



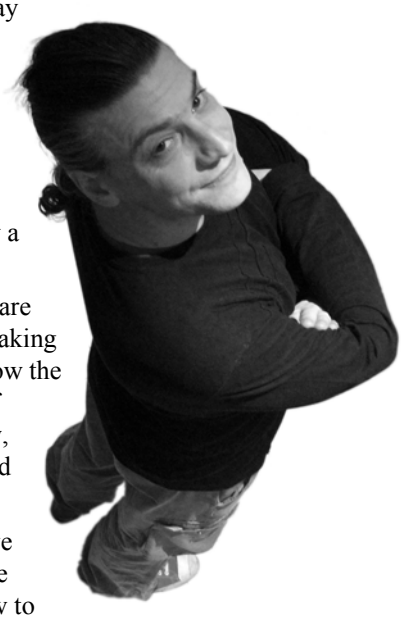
Features

Regulars

{cvu}
Volume 24 Issue 4
September 2012
ISSN 1354-3164
www.accu.org

accu

Different Strokes



Over the last few years I have found myself using a few favourite programming languages to do my day to day work as well as just practising or programming for the sake of it. These days I tend to reach for Python first, and it's a while since I've written C++ in anger, while my day job is largely C# with some Powershell thrown in (we'll leave arguments about whether or not that's programming for now). Lately I've also been dipping my toes in F# and gingerly fiddling with Javascript – but only a little.

When learning a new language, it's natural to try to compare features with one or more languages you already know. Taking this a stage further, it's also natural to try and figure out how the new language copes with your favourite funky features of other languages. It's important not to get too carried away, or course; common idioms in one language can be evil and disruptive in another.

It's also common to see this effect in reverse; when you've become somewhat familiar with your new toy, you'll have discovered its own funky features and start to wonder how to make use of those in the other languages you know. Learning the new has informed your use of something you thought you already knew well.

Just occasionally, this back-porting of features – and even programming styles – becomes mainstream. I doubt that C#'s designers foresaw the techniques familiar to anyone who uses Linq a lot, much less how that technology would introduce a more functional style of writing C# code in general. Support for templates in C++ was overhauled to help Alex Stepanov achieve his vision of a generic algorithms library. Few – if any – people then foresaw the surge of metaprogramming and how it in turn would inform how C++ libraries are written.

When taking features from one language and using them in another, it's important to make the distinction between features and styles that are informed by the language's existing idioms and add value, and those that only create a friction point. On the other hand, we should not be afraid to go against the grain in order to create new value.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

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

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 24.5: 1st October 2012
C Vu 24.6: 1st December 2012

Overload 112: 1st November 2012
Overload 113: 1st January 2013

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

The Curious Case of the Frozen Code

Pete Goodliffe describes the vagaries of the 'code freeze'.

There she blows!—there she blows!
A hump like a snow-hill! It is Moby Dick!
~ Herman Melville (*Moby Dick*)

Managers pronounce it in planning meetings. Developers utter it in reverent awe. Process ceremonies build up around it. And I have to stifle the gag reflex.

It's a shout I imagine coming from a sailor in *Moby Dick*. Not 'There she blows!' but 'Code freeze!' It's about as likely, and just as fictitious.

Our hunt for another mythical state of code.

Hunting the code freeze

Code freeze is a term bandied around with, presumably, good intentions. But often people don't intend to say what the words actually implies.

A code freeze denotes the period between some 'done' point – when no further work is expected to be performed – and the release date.

Exactly when are these points? And what happens in the middle?

- The **release date** is pretty easy to define: sometimes called *release to manufacture* or RTM.

  It's when the *Gold Master* of an installer disk is burnt and sent for duplication. In the enlightened 21st century we may not always ship physical media, but we tend to follow the mechanical conventions dictated by such a release schedule nonetheless. Is this useful and appropriate? Sometimes yes; sometimes no. It does lend a useful cadence to the delivery schedule.
- But what is the preceding **'done' point** that initiates code freeze? Clearly, it should be the point when we consider the code to be complete, with all features implemented and no egregious outstanding bugs. However, some 'freeze' their code at:
  - the *feature complete* point, when all functionality has been written, but not fully tested, and no bugs necessarily addressed.
  - The point of the first *alpha* or *beta release* being made (of course, the definition of these states is also beautifully ambiguous).
  - When a *release candidate* build is first made.

During this period we 'freeze' the code so that no further work ought to be performed on it. However this notion is *pure bunk*; the code never stands still. Whatever happens to the code, this is the phase when a final, exhaustive regression test sweep is run on the software to ensure that it is adequate for release.

> 'Code freeze' is the period leading up to a release when no changes are anticipated.

At best: frozen is figurative term. The code is considered frozen for development work, but is still open for final testing. We *anticipate* some changes being made in light of these tests – if it was not possible to change the code at all, we could just release it now regardless.

Since we're testing to find problems, we will probably uncover a few nasty things that need remedial work. What happens then? You must fix the faults; which implies that the code isn't as frozen as all that! It's not a very deep freeze.

At worst: the code freeze metaphor isn't particularly useful. It's a misnomer.

> 'Code freeze' is a misleading term. Code never stands still, even if you'd like it to.

A new world order

So, at code freeze, we do anticipate some final work will be required. But we carefully monitor the software's development, selectively including or excluding changes in the release code.

Rather than a complete lock-down on changes, 'code freeze' really signifies a new rule of order is in place for the development effort. Changes cannot be applied blindly. Even worthwhile changes must be added with careful agreement.

We work very hard to maintain the integrity of the release, so each change is reviewed very carefully before inclusion. We only include changes that are strictly necessary for the release. Not all issues or bugs found in the 'frozen' code will be considered for fixing post code freeze. Only 'show stoppers' that will prevent a release from being made will be addressed. Some lower priority issues may be queued for a later release, depending on their priority. We balance the risk: it may be more important to release the product than invest time and energy finding and fixing these faults.

Specifically, there is absolutely no more work on new features. No bugs are 'fixed' without prior agreement; we prioritise the issues that have to be addressed. This is a discipline; we do this since even the simplest feature addition or bug fix may introduce unexpected and unwanted side effects.

So this stage of development is not so much a 'freeze' of the code; it is more a very intentional deceleration. It is a mindful reduction of the rate of change of the code line.

> We slow down development work to carefully shepherd a code line to release, managing the final fixes and changes carefully.

Careless speed costs lines (of code).

During the freeze period, some larger (and more departmental) organisations will invoke the services of the 'installer team' to create the install/distribution systems, or get to work on any remaining collateral (artwork, text files, etc) for the final release. Personally, I believe this is wrong – by the time you enter a 'freeze' *all* work should have been completed, ready for final test.

Forms of freeze

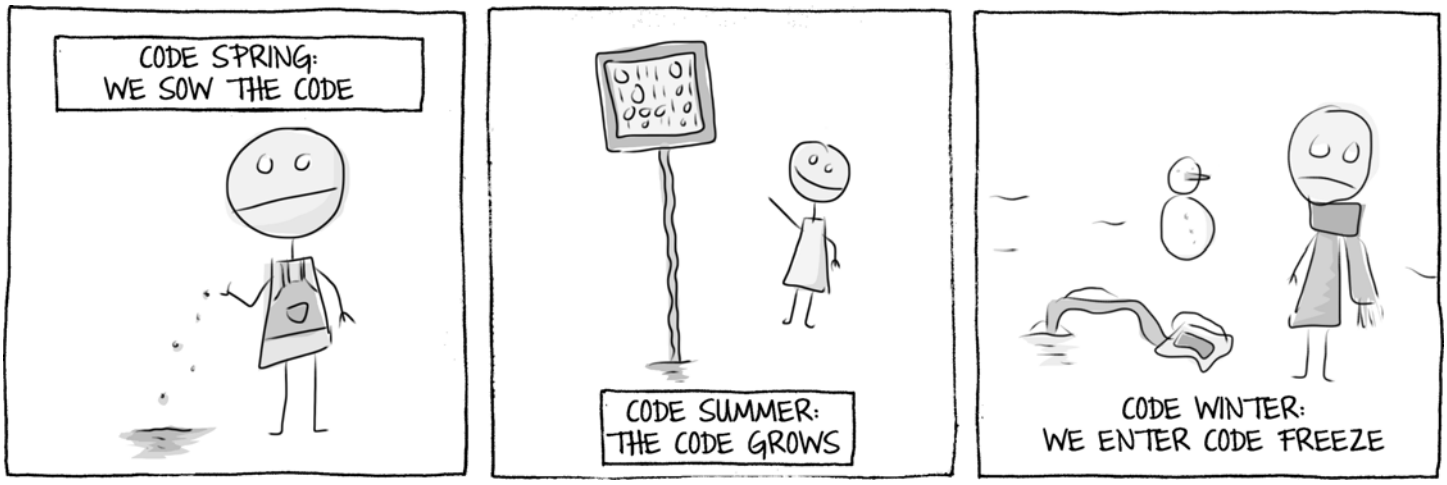
It helps to consider the three different forms of 'freeze', and to be specific in the terms we use. *Code freeze* itself is a bit too woolly and misleading.

- Feature freeze

  A feature freeze declares that only bug fixes may now be committed – no new features will be developed. This helps to avoid 'feature

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net or @petegoodliffe



creep' – as we get near a scheduled release it's always tempting to sneak that one extra little facility in without fully considering the risk or potential bugs that the change may introduce.

- Code freeze

  We no longer work on any features, nor on any bugs that have not been highly prioritised. We only accept fixes for 'show-stopping' issues. We dearly need a better, less ambiguous, name for this state.

- 'Hard' code freeze

  No changes are allowed *at all*. Any change required after this point is tantamount to bringing out the defibrillators and trying to revivify the development team. We never really consider this state, as by the time you get here the software has shipped, and the party has moved on to another code line.

Branches make it work

Typically, when a code freeze is declared, we *branch* the code in the revision control system. Specifically, we create a *release branch*. This allows the release's development code line to be frozen, without delaying other work that can continue on the main code branch.

It is best practice, when working with a release branch, that *absolutely no* code work takes place on the branch itself. The release branch remains, always, stable with no speculative changes applied.

Instead, all work takes place on a spongier branch, perhaps the development mainline. Each fix is tested and verified there and, only when ready, is *merged* to the release branch. By doing this, only acceptably good code ever arrives on the release branch.

Code should always flow between branches *towards* points of stability. We 'promote' change sets based on their proven quality.

Every change that is incorporated into the frozen branch goes through more rigour than previous development changes:

- They are each carefully reviewed.
- They are given focused testing effort.
- They are risk-analysed, so any potential differences they introduce are well understood and, if necessary, mitigated.
- They are prioritised – they will be carefully reviewed for appropriateness in the release.

Branches are pivotal to a team being able to manage code freezes. Without a release branch, all the developers would physically have to put down their tools, and stall work for the duration of the freeze. This isn't a good use of time or expensive resources. Developers like to develop; soon they'll get itchy feet, and write code anyway.

> Branch or bust.

That said, it's a good idea to avoid concurrent work as much as possible – it can be confusing and lead to conflicting goals and aims for the team.

But it's not really frozen!

Be careful that the 'code freeze' misnomer doesn't lead you into a false sense of confidence. Often the term code freeze is pronounced to managers to imply a more stable project state, to garner their confidence. It *sounds* great, doesn't it?

But don't believe your code is in a better state than it is. At all times it's important to have a realistic appraisal of the state of your project.

Be wary that the word 'freeze' doesn't tempt you to keep things rigid when they should not be. When changes must be made, they *must* be made.

Length of the freeze

You must declare a digital winter for the right length of time. Like the Narnian winter, you don't want an unnecessarily lengthy freeze where Christmas never comes! But have it too short, and the freeze is a pointless exercise.

The correct period depends on the complexity of your project, the test demands it imposes (both on people and resources: do we need to install/configure a whole separate test platform with administrators and boffins to keep it spinning), the scope of the changes that have gone into this release (which may influence the level of regression testing performed), and the resources available to devote to test and verification.

A typical freeze period size is two weeks.

Beware the Pareto principle: we often see in IT projects where the 'last' 20% of effort expands to take up 80% of the total time (or thereabouts). To avoid this, make sure you enter the freeze at the right point. Don't declare a freeze when you think you just need to 'finish off' a few things. You freeze once everything is finished off.

Feel the freeze

A code freeze is the hard road to release – not a picnic in the park. Set your expectations accordingly.

During a code freeze period expect to find bugs that you *will not be able to fix* because they are not important enough to risk inclusion. This is no longer a coding free-for-all where any code change is permissible; otherwise you wouldn't have declared freeze. Therefore expect to be disappointed, and to ship product that you'd hope would be better!

> It's not unusual (or wrong) to ship software which you know could be better.

Look on the bright side: since you've found them, you can fix those bugs in the next release.

Also expect to rack up *technical debt* during a freeze [1]. This is one of the few valid times to do so: when there's no scope to make wide-ranging repairs, you have to fix problems with stop-gap 'paper over the cracks' techniques to get a 'good enough' shipping product. But do remember to consider this kind of work *debt*, not normal practice, and plan to pay this off in the development cycles after the release.

> During code freeze you will accrue technical debt. Monitor this, and be prepared to pay the debt off soon after the release ships.

If you make a change during freeze that has serious implications, consider if the code should be thawed and re-frozen, with a thorough test cycle kicked off from scratch. Postpone the release and restart your code freeze period if you have to.

Scientists tell us that freezing-thawing-freezing is bad for your health. So be careful not to do this too many times, though, or you'll end up with food poisoning!

A long freeze period is a warning sign that you have not got a stable enough codebase.

The end draws near

At the end of a code freeze period, when we reach the RTM point, the code line is really, *honestly* frozen. No changes will now occur as the release has (finally) been made. Close the release branch. Archive the code line. Go and celebrate.

Any further changes will be made to a different code line.

> The only true 'code freeze' is when an acceptable release is made. This is the point that the code is finally *set in stone*.

This point is the real honest-to-goodness code freeze. But no one ever talks about this!

Antifreeze

If you work well enough, it is possible to actually avoid code freeze periods. You *can* skip this sordid dance altogether.

This is possible since many development teams no longer constrained by physical manufacturing process – they ship software over internet, or create web services that can be deployed into production servers in a heartbeat.

The 'disaster' of a bug making it through to an external release is minimised here – an online software update can be deployed to remedy the issue in the field before many users spot the problem. However, we still strive to avoid bugs appearing shortly after a release.

We can minimise code freeze periods by:

- Employing continuous delivery; setting up a pipeline that ushers each build into a full deployable state. This ensures that you are always ready to deploy.
- Establishing a good *automated* test coverage. These tests must cover the code, the integration, and final user-facing aspects of the system to give reliable feedback on the state of the product.
- Good acceptance criteria testing – tools like Cucumber [2] can be used to ensure that the full set of high-level user requirements have been met by the software.
- Reduce the test period – reduce scope/size of project so that you don't need a lengthy lock-down for each release.

Conclusion

Code freeze is a problematic term; it is a misleading metaphor. Code doesn't really freeze or thaw. Code is a malleable substance, constantly changing and adapting to the world around it. What really happens is the rate of change of development slows, and we change the focus of our work.

However, it is true that as we get near a software release we need more discipline in the development regimen to ensure the software is of a releasable quality. ■

Questions

1. Do you have a formal code freeze period in your development practice?
2. How do you ensure that changes applied in the freeze period are safe and appropriate?
3. Is a single person responsible for the quality of the build, or is it a team concern? Which is the right approach, and why?
4. Does it take your project a long time to get to the code freeze point? Why? How can you shorten this?

References

[1] The Technical Debt metaphor: http://martinfowler.com/bliki/TechnicalDebt.html

[2] Cucumber http://cukes.info

Learning and Applying the Personal Software Process

Robert Bentall shares his experiences from learning to measure his own performance.

In the mid-1990s, I spent a couple of years as a postgraduate student at the Royal Northern College of Music in Manchester (UK). I was focused on becoming a professional horn player, and the accepted route is to go through a conservatoire training.

As I did this, it became clear that the most important part of my training was learning to listen – to myself, my teachers, my peers, and those in the profession I wished to emulate. This relationship between between practice, observations, and goals is shown if Figure 1.

It's this awareness that is the foundation of performance improvement. Without it,

- I couldn't tell how well I was doing
- it became much harder for me to refine my goals
- my practice was less effective.

As a musician, I figured out how well I was doing by listening. But as a software engineer, I don't have access to such direct feedback. We engineers need tools to gather data on how well we are doing, models to help us interpret that data, and techniques we can use to solve common problems.

That is the essence of the PSP training. It teaches the fundamentals of performance management as applied to the individual software engineer. By undergoing the training, we engineers can learn how to

- gather data about our performance
- compute measurements from that data
- interpret those measurements
- use different techniques to improve performance.

Once we've grasped these concepts, we are in a position to determine which techniques and methods work best for us and to continually improve our performance.

I began the PSP training in July 2011 and completed it about six months later. The course was both challenging and rewarding, and it has already started to pay dividends. The techniques I learned can be applied to good effect on a day-to-day basis, and the breadth of the course has provided a solid base for my future development.

History

PSP was the brainchild of Watts Humphrey (1927–2010).[1] He spent most of his career at IBM as a senior executive, and after retiring from IBM, he moved to the Software Engineering Institute (SEI) at Carnegie Mellon University. While at SEI, he was instrumental in the development of the capability maturity model. He started applying the model's principles to writing software, and over time, this grew into the PSP. His obituary, found on the SEI website, contains a wealth of information on his life.[2] An oral history, recorded with Grady Booch, gives us an insight into the breadth and depth of his expertise.[3]

ROBERT BENTALL

Robert Bentall is a software engineer for an oilfield services business. He works on the development of reservoir engineering simulators using Microsoft C# and C++ / CLI. He can be contacted at robertbentall@supanet.com

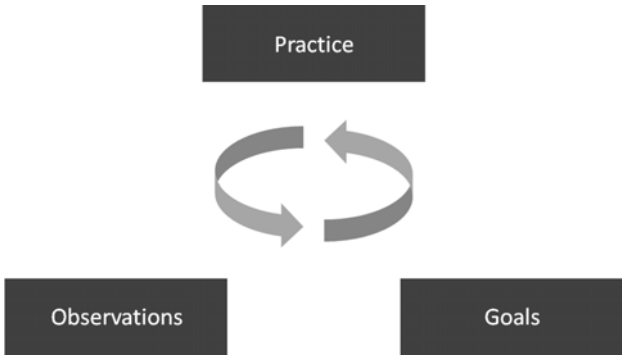


Figure 1

His intention was to apply personal quality management techniques to the development of 'module-sized' programs so that the work of each individual software developer would be of a very high quality.

Course structure

The course is structured as two 1-week parts, with report assignments at the conclusion of each part. Part 1 covers measurement, planning, and estimation. Part 2 addresses software quality, design, defect detection and removal techniques, and process management.

Each course includes lectures, programming exercises, and data analysis. The mix of theory, practice, and analysis is crucial to learning to apply the techniques and figuring out how to interpret the results.

The PSP course is based on a set of training processes; each process version builds techniques incrementally. Participants must write one or more programs using each process and undertake data analysis exercises at the mid-point and end of the course.

The processes have a simple linear structure comprising a sequence of up to eight steps:

- planning
- design
- design review
- coding
- code review
- compilation
- test
- postmortem.

This provides the framework to introduce the different techniques that are taught on the course (see Table 1).

The course covers a lot of ground in a short period of time, but because participants are always practising the techniques learned in previous versions of the process, it doesn't take too long to master them.

Scripts are used to guide workflow and these provide operational definitions [4] of each version of the process. This helps prevent important tasks from being missed and makes it much easier to apply the techniques. It also helps ensure consistency of data capture.

Table 1

Techniques	Process Version					
	Process discipline and measurement		Estimation and planning		Quality management and design	
	PSP 0	PSP 0.1	PSP 1.0	PSP 1.1	PSP 2.0	PSP 2.1
Process measurement	✓					
Coding standard process improvement proposals, size measurement		✓	✓	✓	✓	✓
Size estimation, test reports			✓	✓	✓	✓
Task planning, schedule planning				✓	✓	✓
Design reviews, code reviews					✓	✓
Design templates						✓

The first program I wrote used the PSP 0 process. This was intended to be a relatively small step from my existing development approach, and it introduced the basics of personal process measurement. By the time I got to PSP 2.1, the process had become a little more complex. But because the techniques had been built incrementally, it was still straightforward.

A common mistake when looking at the PSP is to think that it reduces the developer to 'development by checklist' and that it enforces the use of a waterfall. Both concerns are misplaced. It's straightforward to work iteratively within each version of the PSP, and while the use of scripts to guide workflow may seem alien, I found it liberating – I no longer had to worry about forgetting to do something. It freed me up to concentrate on the problem I should be solving.

What are the options for taking the course?

There are three options for people wanting to take the course:

- attend one of the SEI courses [5]
- obtain the materials and self-teach [6]
- work with a coach.

For me, the prospect of an SEI course was remote, so I opted to work with a coach. Working with a coach proved to be very interesting and worthwhile. Although anyone can just follow the book and materials, it is much easier with support and external feedback.

The course materials are readily available. [6] The course textbook, *PSP – A Self-Improvement Process for Software Engineers*, [1] provides all the theory needed, and an open-source tool, the Process Dashboard, makes metrics collection easier. [7]

Completing the course took me about 300 hours, including the time I spent writing extra programs and reworking several of my analyses. The target time for completing both parts of the course is 150 hours, and there are significant benefits to be gained just by completing the first part. This means that for an investment of 50 to 100 hours, we can gain much of the technique and insight needed to introduce personal quality management into our work.

Lessons learned during the PSP training

The course imparts a large amount of valuable information, and I know I will continue to use the techniques I've learned. The lessons I have learned can be divided logically into two groups:

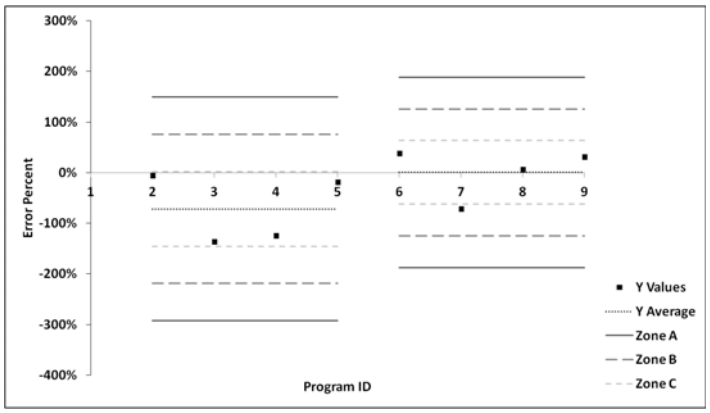
- techniques
- conclusions about my performance.

Techniques

The course provides an opportunity to practise new techniques in a safe environment:

- a structured approach to estimating the size, quality, and cost of software deliverables

Figure 2



- effective project tracking and status reporting
- classification and analysis of defects
- performing personal design and code reviews
- program verification
- measuring and managing quality.

Each one of these can be applied in isolation to target a specific problem. This means that once I'd learned the techniques I could introduce them progressively into my daily work. Although all the techniques are valuable, the design and code reviews stand out as surprisingly powerful.

My conclusions about my performance

It's important to be cautious about the conclusions we draw with respect to our own performance. I was learning new techniques and drawing conclusions from small sets of data. Both factors should cause us to be tentative in our interpretations, so please keep this in mind as you read my conclusions.

I found that my size and time estimation accuracy improved a little during the course (Figure 2, which shows size-estimating errors for Programs 2 to 9 and Figure 3, which shows time-estimating error for Programs 1 to 9).

I've used control charts [8] to show changes in my performance during the course. These were invented in the 1920s by Walter Shewhart and made popular by his pupil W. Edwards Deming. They can be used to understand the variability within a time series dataset, which enabled me to identify where significant changes have occurred. At least 5 points per process are needed, so these results are intended only for illustration. For more details on use of control charts, refer to the book by Don Wheeler. [9]

I have split the data so that Programs 1–4 are considered as one process and Programs 5–9 are considered as a separate process. The logic for this is that the earlier programs were written using different versions of the PSP but Programs 5–9 were all written using one version, PSP 2.1. I calculated the process mean and control limits using the normal rules. Zone A corresponds to ± 3 standard deviations, Zone B to 2, and Zone C to 1.

For the size estimation, I made these tentative conclusions:

- I was going from continually underestimating by a large amount to estimates balanced around zero.

Figure 3

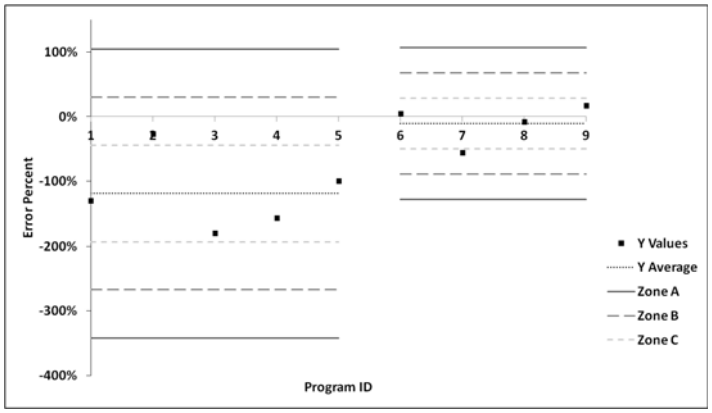
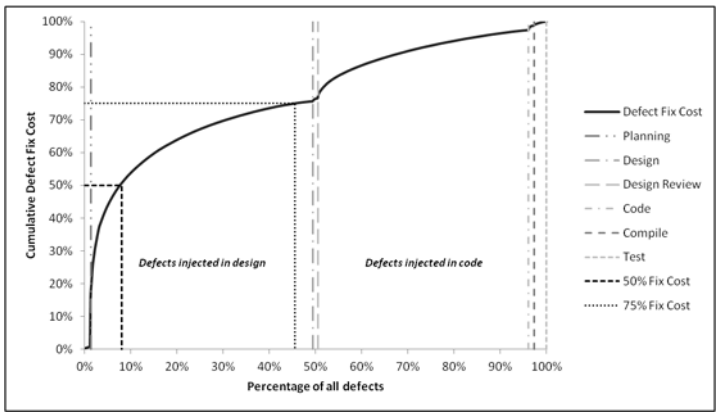


Figure 4



- The variability of estimates was improving slightly: Zone C was moving from ±85% to ±60%.

For time-estimating accuracy, the tentative conclusions were that I was improving a little:

- My process average improved from -120% in the first five programs to -10% in the second four programs.
- The variability of estimates improved: Zone C is ±75% for first five programs and ±40% in last four programs.

These results do not mean that my size- and time-estimating accuracy will continue to improve in the future, merely that I improved as I progressed through the programs.

There were some interesting patterns in the defects I injected. I rarely made significant errors in the structure of my programs (e.g., class design, relationships between classes, etc.), but the most expensive ones were design errors, typically related to algorithm structure. I also found that 50% of the total defect fix cost was due to only 8% of the defects (as seen in Figure 4).

This chart is derived by taking the cost of fixing each defect and ordering it by phase injected and descending fix cost. The fix cost values are then plotted cumulatively as a percentage of the total fix cost for all defects. Vertical lines mark phase boundaries. Fix cost is the time taken to fix a defect, from the point at which the defect is first found to the point at which the fix is completed.

This profile is really interesting, because once I recognized the pattern, it became possible to design filters to reduce or even eliminate the issue. For example, I found that in the earlier programs, I was not specifying error behaviour accurately enough to allow me to code it without error. I introduced a check into my design review to prompt me to validate the expected error behaviour. This went some way to trapping these defects earlier in the process.

More generally, the PSP training emphasizes the use of individual developer reviews of design and code as a very powerful defect-detection technique. Personal defect data provide an excellent basis for tuning reviews to find the kinds of defects typically injected.

Although individual developer reviews are not as effective as properly conducted team-based inspections, they can still find a very significant percentage of the defects in a software artefact. A 2006 study by Cisco Systems found that individual developers inspecting their own code would find approximately 50% of the defects that were found by the team review. [10]

In PSP, 'Yield' is used to measure the effectiveness of defect-removal processes and is the ratio of defects found divided by total defects in the system at that point in time. I found that I was consistently getting 70% to 80% yield before the compile stage, meaning that only 20% to 30% of the defects were being found during compile and test (Figure 5).

In PSP, a defect is counted each time an artefact from a previous phase needs to be corrected. This could be an error in a design document, in test data, or in code. This chart therefore tells us the percentage of errors that are corrected before compile. A similar chart can also be plotted using the fix cost of each defect, which allows derivation of a fix cost profile for the development process.

Again I've used a control chart to show changes in my performance during the course. Looking at this data, my performance seemed to stabilize fairly quickly. I've opted to treat Programs 3–9 as one process, purely because that seems the most natural separation based on the precompile yield.

I had less success with the design verification techniques. Although they are good if they can be mastered, I found they didn't significantly increase my defect detection rate but did consume significant effort. I suspect that with further practice on my part I will become a lot more efficient and effective.

PSP has a rich set of metrics that provide insight into our development habits. The examples given are just a small subset that I've used to illustrate some of the lessons I have learned.

How I apply lessons learned

The wonderful thing about the techniques is that they are all independent. I can employ just the techniques I need, when I need them. This allows me to introduce them to my work progressively. For example,

- I started tracking most of my projects using the techniques taught on the PSP course as soon as I had learned them. They provided a simple graphical approach to demonstrate progress against plan.
- I developed and used a process for guiding defect resolution with a client. This facilitated clear communication, more accurate plans, and better management of the project.
- when I'm estimating cost, size, and schedule, I use the PSP techniques to do the estimates.

Going beyond the PSP: the team software process

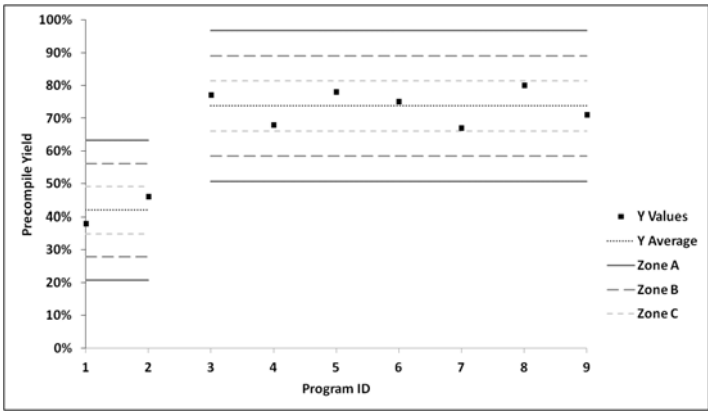
One of the interesting points identified by Watts Humphrey was that after completing the course, most PSP trained developers tended not to apply techniques in their day-to-day work. This was primarily because of the levels of self-discipline required and the environmental challenges faced by the developer. [11] To address this finding, he went on to develop the Team Software Process[SM]. [12] This process took the techniques of the PSP and applied them to the work of software teams. When applied correctly, the techniques have improved the quality, cost, and schedule record of software development teams. [13]

Closing thoughts

It's been a big investment to complete the course. However, the course teaches something pretty fundamental. It's about learning to listen to ourselves as software engineers, and about understanding how we can use the information we hear to improve our performance. In some areas of my performance, I am starting to see the benefits, although it has taken time.

I know of no other training program that provides a consistent, complete grounding in pretty much all the tools needed to improve performance. In that sense, I suspect the course is unique.

Figure 5



Achieving 'master level' performance in any field takes a lot of effort. The oft-quoted number is 10,000 hours of practice over 10 years.[14] Viewed in that light, 300 hours doesn't seem so bad. Perhaps we shouldn't be surprised that becoming a good programmer takes a lot of time.

In one of his interviews, Watts Humphrey noted that he was taking techniques already in existence and scaling them to apply to the individual software engineer. [15]

What has been fascinating as I've gone through the training is that I've become much more aware of this. Many of the problems that software developers face on a day-to-day basis have already been solved. We just need to recognize this and understand how to apply the solutions.

My advice? Do the course.■

References

[1] Humphrey, Watts S.: *PSP: A Self-Improvement Process for Software Engineers*, Upper Saddle River, New Jersey, USA, Addison-Wesley Professional, (2005).

[2] Carnegie Mellon University: http://www.cmu.edu/news/archive/2010/October/oct28_wattshumphreyobit.shtml (accessed 10 May 2012).

[3] Pearson Education, Informit, An Interview with Watts Humphrey: http://www.informit.com/promotions/promotion.aspx?promo=137746 (accessed 10 May 2012).

[4] W. Edwards Deming, *Out of the Crisis* (1st ed.), Cambridge, MIT Press (2000), 276.

[5] Carnegie Mellon University: SEI Training, http://www.sei.cmu.edu/training/?location=main-nav&source=1358 (accessed 11 May 2012).

[6] Carnegie Mellon University: SEI Training, http://www.sei.cmu.edu/tsp/tools/student/?location=tertiary-nav&source=5784 (accessed 2 July 2012).

[7] The Software Process Dashboard Initiative, http://www.processdash.com/ (accessed 11 May 2012).

[8] Wikipedia, Control chart, http://en.wikipedia.org/wiki/Control_chart (accessed 11 May 2012).

[9] Wheeler, Don: Understanding Variation-The Key to Managing Chaos, Knoxville, Tennessee, USA, SPC Press Inc. (1993).

[10] Cohen, Jason: Best Kept Secrets of Peer Code Review, Beverly, Massachusetts, USA, SmartBear Software (2006), as cited in Oram, Andy and Wilson, Greg (eds.), *Making Software: What Really Works, and Why We Believe It*, Cepastopol, California, USA, O'Reilly Media Inc. (2010) 336.

[11] Pearson Education, Informit, An Interview with Watts Humphrey: http://www.informit.com/articles/article.aspx?p=1625324 (Accessed 2 July 2012)

[12] Carnegie Mellon Software Institute, Team Software Process, http://www.sei.cmu.edu/tsp/ (accessed 11 May 2012).

[13] Jones, C., *Software Engineering Best Practices*, New York, McGraw-Hill (2009) 293-298.

[14] Ericsson, K.A., Krampe, R. Th., and Tesch-Romer, C.: 'The Role of Deliberate Practice in Expert Performance', *Psychological Review* (1993) 103, 363-406.

[15] Pearson Education, Informit, An Interview with Watts Humphrey, Part 21: The Personal Software Process, http://www.informit.com/articles/article.aspx?p=1614506 (accessed 11 May 2012).

Anatomy of a CLI Program Written in C

Matthew Wilson dissects a simple console application to reveal hidden complexity.

I want to stop thinking! More precisely, I want to stop thinking about basic things. More accurately, I want to stop thinking about *fundamental* things.

During our 2011–12 Christmas trip back to Blighty I had the singular pleasure of spending 90 densely-conversational minutes in a London pub with Chris Oldwood and Steve Love talking about, as Steve coined it 'fundamental, *not* basic' issues of programming. Steve related tales of former colleagues' frustrations with having to think about 'basic things', to which he offers the above apposite correction. Thinking about fundamental things is not a waste of time. The fact is, software development is still a very young field, as those of us who try hard at our practice know all too well – we do not even use basic terms such as 'error' properly or definitively [1].

The more software I write, the more I am concerned with fundamental things. The trains of thought, and the concomitant changes to my practice, that prompted me to start (and soon pick up again) my 'Quality Matters' column, mean that I can no longer develop software in quite the same ways as before. I must perforce consider quality, and most particularly failure, a lot more – diagnostics, contracts, testability, and so on – when I write even simple programs.

But there's a limit to how interesting such concerns can be, and how productive they can let one be, and I've reached it, at least in one area of programming: it's time for me to start drawing some lines in the sand when it comes to command-line interface (CLI) programming. I've been writing CLI programs in C for 25 years (and in other languages for considerable times too). I've been using program-generating wizards for almost 20 years. But these tools are well past use-by-date, not only in terms of the environments within which they run, but also regarding the state of the art of the language(s), libraries, and (good) practices that they employ.

Now I want to identify definitively the 'anatomies' of CLI programs, solidify them in libraries and program generating wizards, and just crack on. I also seek, wherever possible, to identify ways in which the boilerplate aspects of programs can be abstracted without detracting from flexibility or transparency, such that their visual impact can be hidden/diminished, thereby *increasing* the transparency of program-specific code (and, in a real sense, increasing the 'average' transparency of all the code that I write).

Ideally, I'd like to be able to begin this series of articles with an oracular stipulation of the definitive taxonomy of software anatomies and a presentation of matrices of program-area • module-type • language, then distil them in subsequent instalments for particular programming areas in a simple *fait accompli*. Certainly I do have some strong feelings in this area, e.g. where diagnostic facilities should be located in dependency graphs. Problem is, I don't (yet) know them all: that's what I'm hoping to identify as we go.

So, where to start? Because I've done a tremendous amount of CLI programming (in C, C++, C#, and Ruby) this last couple of years, I do have strong feelings about CLI program anatomies, and much (and varied) experience to back them up. So, the plan is to work bottom up, starting with CLI programs written in C. Naturally, I hope (and request!) to receive plenty of feedback to these articles from you gentle readers, since I cannot expect to have captured all good and bad practices, even in those areas in which I'm most experienced.

MATTHEW WILSON

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

Anatomy

First I should explain the use of the term 'anatomy'. Simply, I'm interested in both logical structure and physical structure, so I chose anatomy as an umbrella term, rather than having to constantly refer to 'both logical and physical structure'. Hopefully it'll catch on. ☺

For example, it's useful that the core elements of a CLI program be (more) testable, particularly in automated test harnesses. Both physical and logical dependencies impact on this. If the core program functionality is located within the same physical file as `main()`, that increases the difficulties in compiling and linking it into a test harness – we'll be forced to use some Feathers-like pre-processor manipulation [2]. Conversely, if the core logic depends on specific third-party 'general services' libraries – e.g. diagnostic logging, contract enforcement, database manipulation – these will significantly increase the difficulty and scope of the testing.

I hope to elucidate the impacts of both aspects of program anatomy as this series progresses.

Logical layers of components/services

Another aspect in which I'm very interested is the layering of the logical dependencies. Considering just a CLI program, we can identify a number of components/services that may be found:

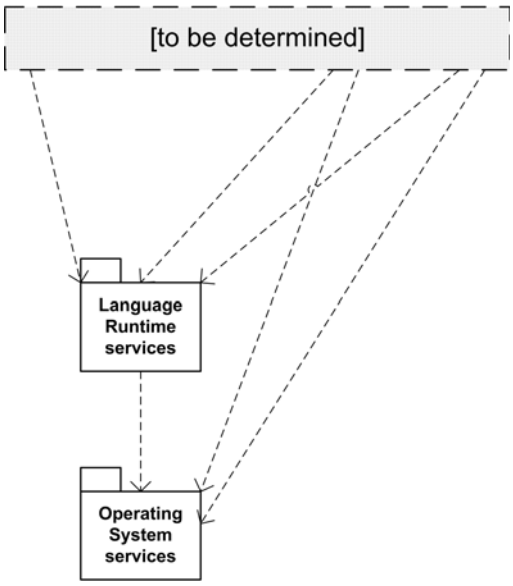
- Operating System services;
- Language Runtime services;
- Language Standard Library components;
- Diagnostic services (including Diagnostic Logging, Runtime Contract Enforcement, Code Coverage, Memory-leak Detection);
- Command-line Parsing component;
- other 3rd-party library services/components; and, of course
- all the programmer-written code: process command-line arguments; decide what to do; do it.

It seems pretty uncontentious to claim that Operating System services must be ready and available to Language Runtime services, and that Language Runtime services must be ready and available to all other components/services. However, I think there will be some equivocation on what the dependency graph looks like beyond that point, and that will also depend on language and program type. All I will stipulate for now is Figure 1; I'll revisit this graph many times in the coming series.

Example program: slsw

The only way I know to go about this is to use an example, starting simple and building up to what I consider to be a releasable standard, or until I run out of stream or space (or time!). After fluffing around with several different programs I've settled on rewriting an existing tool **slsw** (**sl**ash **sw**ap), which, er, swaps slashes in its input to its output.

Figure 1



For this first article there are several simplifications:

- no diagnostic logging;
- most-basic contract enforcement, using `assert()`; and
- assumption that it is a standalone program. In reality it is one of a suite of related, and similarly implemented tools, the significance of which, to program generation and coding practice, will be discussed at a later time.

Let's dig in.

Step 1 – Initial version

The first step is shown in Listing 1. I trust it's largely self-explanatory; the use of `stdin` and `stdout` via the `in` and `out` variables is a minor sop to *revisibility* (see sidebar) in light of what's to come.

The implementation is all very well if:

- we only want to read from standard input and write to standard output;
- we only want to swap backslashes to slashes; and
- you don't have to ask the program how to use it.

Step 2 – Read from file / stdin, write to file / stdout

Let's deal with the first of these issues, limitation to using standard input and standard output. While UNIX-like filter programs [3] are most often used in this manner, it is also useful, and therefore common practice, to allow filenames to be specified for the input and/or output, as in:

Listing 1

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
  FILE* in  = stdin;
  FILE* out = stdout;
  int   ch;
  for(; EOF != (ch = fgetc(in)); )
  {
    if('\\' == ch)
    {
      ch = '/';
    }
    fputc(ch, out);
  }
  return EXIT_SUCCESS;
}
```

Revisiology and Revisibility

Over the last decade of writing about software, I've become increasingly engaged with the notion of studying the history of a given source code entity throughout its life. As a consequence of many such studies, it's become clear that there are many practices – whether accidental or deliberate – that can significantly affect the ease with which such historical studies may be conducted.

As you may know, gentle readers, I'm always seeking out precise definitions for software development concepts (mainly to aid in clearing the ever-gathering fog of future shock in my own mind), and I occasionally suggest new names for such. I like to think I do the former reasonably well, but concede readily that I often stumble in the latter. And so you have been warned.

With a little help – though the blame remains all my own – from the ACCU General list members, I've devised two names for the two concepts described above:

- *Revisiology* is the study of source control entity histories;
- *Revisibility* is the degree of ease by which an understanding of the (the nature and purpose of) differences in revisions of a source control entity can be gleaned. Revisibility is of particular interest (to me, at least) where it may be strongly affected by decisions made in aspects of coding in which choices exist. For example, the new code added from Step 2 → Step 3 is highly *revisible*. (That it's not good code, in so far as it builds upon poor coding present in Step 2, is incidental to its revisibility, though not to its fitness otherwise!)

How far I will take these terms, and for how long, is yet to be determined, but I'll certainly be using them throughout this series of articles. (I reserve the right to rename them, though, if someone suggests better alternatives.)

```
$ slsw input.txt output.txt
```

Furthermore, in order to cope with the situation of wanting to write to a named file while still reading from standard input, it's also common to interpret a filename of '-' as meaning read from (or write to) standard input (or standard output); this was discussed in more detail in my article about CLASP [4].

Without resorting to use of any other libraries, we can support *almost* all of this properly by changing the two lines declaring and assigning to our `FILE*` variables, as shown in Listing 2. (NOTE: for reasons of brevity, *in this case* I do not test and close `FILE*` variables, since streams are closed by the runtime when the program exits; it is best practice to do so explicitly in general.)

The reason it's only *almost* proper is because the command-line flag `--`, by convention, is used to specify that all subsequent arguments be treated as a value, regardless of whether they begin with (or consist solely of) a hyphen. In the case of slsw, this would allow for specification of a file named '-'. I'm ignoring this, because the issue was dealt with in the CLASP article, and, as you've probably guessed, I'm going to have to plug CLASP in pretty soon.

The code in Listing 2 works well in the normative case. But it's opaque, and not something anyone would write with any justified pride. Much worse, it (mis-)handles non-normative behaviour by undefined behaviour:

Listing 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char** argv)
{
  FILE* in  = (argc < 2 || 0 == strcmp("-",
     argv[1])) ? stdin : fopen(argv[1], "r");
  FILE* out = (argc < 3 || 0 == strcmp("-",
     argv[2])) ? stdout : fopen(argv[2], "w");
  int   ch;
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}
```

Listing 3

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char** argv)
{
  char const* inName  = NULL;
  char const* outName = NULL;
  FILE* in  = (argc < 2 || 0 == strcmp("-",
     inName = argv[1])) ? stdin :
     fopen(inName, "r");
  FILE* out = (argc < 3 || 0 == strcmp("-",
     outName = argv[2])) ? stdout :
     fopen(outName, "w");
  int   ch;
  if(NULL == in)
  {
    int const e = errno;
    fprintf(stderr, "slsw: could not open '%s'
      for read access: %s\n", inName,
      strerror(e));
    return EXIT_FAILURE;
  }
  if(NULL == out)
  {
    int const e = errno;
    fprintf(stderr, "slsw: could not open '%s'
       for write access: %s\n", outName,
       strerror(e));
    return EXIT_FAILURE;
  }
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}
```

passing the name of an unreadable input file causes a segmentation fault (on Mac OS-X, and likely on other systems also). Yikes!

Clearly, we have to check for failure to open named files.

Step 3 – Failure handling

For reasons of pedagogy and revisibility alone, I fix the non-normative issue in the manner shown in Listing 3; if this were to be the (near-)final implementation of the program, I would instead process detecting and processing the path arguments together, and much more clearly. Thankfully, I don't have to, because that's already too much silliness and wasted effort in command-line argument processing. It's time to call in CLASP.

Note that, according to UNIX convention, the non-normative output – that which occurs when the program is not achieving its primary purpose: in this case the contingent reports in the unrecoverable condition handlers just added – is marked with the program name, and the normative output is not (as it would stop it being of any use as a filter).

Listing 4

```
#include <systemtools/clasp/clasp.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
static clasp_alias_t aliases[] =
{
  CLASP_ALIAS_ARRAY_TERMINATOR
};
```

Listing 4 (cont'd)

```
int main(int argc, char** argv)
{
  clasp_arguments_t const* args;

  int const cr = clasp_parseArguments(
    CLASP_F_TREAT_SINGLEHYPHEN_AS_VALUE
  , argc
  , argv
  , aliases
  , NULL
  , &args
  );

  if(0 != cr)
  {
    fprintf(stderr, "slsw: failed to parse
       command-line: %s\n", strerror(cr));
    return EXIT_FAILURE;
  }

  else
  {
    char const* inName  = NULL;
    char const* outName = NULL;
    FILE* in  = stdin;
    FILE* out = stdout;
    int   ch;
    clasp_argument_t const* arg;

    if(clasp_checkValue(args, 0, &inName,
                        NULL, &arg))
    {
      if(0 == arg->givenName.len)
      {
        if(NULL == (in = fopen(inName, "r")))
        {
          int const e = errno;
          fprintf(stderr, "slsw: could not open
             '%s' for read access: %s\n",
             inName, strerror(e));
          clasp_releaseArguments(args);
          return EXIT_FAILURE;
        }
      }
    }

    if(clasp_checkValue(args, 1, &outName,
                        NULL, &arg))
    {
      if(0 == arg->givenName.len)
      {
        if(NULL == (out = fopen(outName, "w")))
        {
          int const e = errno;
          fprintf(stderr, "slsw: could not open
             '%s' for write access: %s\n",
             outName, strerror(e));
          clasp_releaseArguments(args);
          return EXIT_FAILURE;
        }
      }
    }

    for(; EOF != (ch = fgetc(in)); )
    . . .
    clasp_releaseArguments(args);
    return EXIT_SUCCESS;
  }
}
```

Step 4 – Using CLASP (longhand)

Plugging CLASP straight into **main()**, giving Listing 4, results in a file of nearly triple the size. (For sure, it's a lot more transparent than the mess of Step 3, but still …) As touched upon in [4], it's almost never the right thing to plug it straight into **main()** along with your program logic. This issue of *where* different parts of the program should go is one of the main themes of this article, which I'll go into in a lot more detail later. For now, just go along with the incremental steps, if you don't mind.

Hopefully it's all pretty self-evident in light of the CLASP article [4], with the probable exception of the check on **givenName**'s length: this uses a feature of CLASP whereby '**-**' arguments that are not preceeded by **--** and thus would usually be interpreted as flags are instead interpreted as values if the **CLASP_F_TREAT_SINGLEHYPHEN_AS_VALUE** parsing flag is specified, and identified as such by having non-empty **givenName** (and **resolvedName**, for that matter), which no other values ever have; only when the value is present and is not '**-**' do we open a file, rather than accept the built-in stream. (Note to self: perhaps some more transparency-engendering macro/function – e.g. **clasp_valueIsSingleHyphen()** – might be a good addition to the next release.)

(NOTE: once again, revisibility influences the declarations of **clflags**, as it allows me to add/remove flags in a manner – one-per-line – that is isolated and unambiguous. This tactic I employ in real work.)

Step 5 – Using CLASP::Main

Although, in my opinion at least, the code in Step 4 is much improved in transparency – as well as actually properly handling all the required command-line permutations, don't forget – it is actually a good deal more verbose. Furthermore, although not shown in the listing, in the actual code I made two mistakes. The first was in comparing **cr** less than 0, rather than not equal 0: easy to do, hard to spot, even harder to test against (since **clasp_parseArgument()** failures are exceedingly rare [4]). The second was in omitting the first two calls to **clasp_releaseArguments()**: again, easy to do, and hard to spot.

Thankfully, **CLASP::Main**, a CLASP extension library, provides a way to avoid (mis-)writing this boilerplate from program to program, via initialisation-function layering, obviating both my real mistakes. Applying it gives Step 5, as shown in the differential Listing 5.

Listing 5

```
#include <systemtools/clasp/clasp.h>
#include <systemtools/clasp/main.h>
#include <errno.h>
. . .
static clasp_alias_t aliases[] =
. . .
static
int clasp_main(clasp_arguments_t const* args)
{
  char const* inName  = NULL;
  . . .
  if(clasp_checkValue(args, 0, &inName, NULL,
                      &arg))
  . . .
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}

int main(int argc, char** argv)
{
  int const clflags = 0
     | CLASP_F_TREAT_SINGLEHYPHEN_AS_VALUE
     ;
  return clasp_main_invoke(argc, argv,
     clasp_main, "slsw", aliases, clflags, NULL);
}
```

Listing 6

```
. . .
static clasp_alias_t aliases[] =
{
  CLASP_FLAG(NULL, "--help",
             "invokes this help and terminates"),
  CLASP_ALIAS_ARRAY_TERMINATOR
};
static
int clasp_main(clasp_arguments_t const* args)
{
  . . .
  clasp_argument_t const* arg;
  if(clasp_flagIsSpecified(args, "--help"))
  {
    clasp_showUsage(
      NULL
    , aliases
    , "slsw"
    , "Synesis Software SystemTools
      (http://synesis.com.au/systools)"
    , NULL
    , "Swaps slashes in text"
    , "slsw [ ... options ... ] [<input-file>|-]
      [<output-file>|-]"
    , 0
    , 1
    , 6
    , clasp_showHeaderByFILE
    , clasp_showBodyByFILE
    , stdout
    , 0
    , 76
    , -4
    , 1
    );
    return EXIT_SUCCESS;
  }
  if(0 != clasp_reportUnusedFlagsAndOptions(args,
     &arg, 0))
  {
    fprintf(
      stderr
    , "slsw: unrecognised argument: %s\n"
    , arg->givenName.ptr
    );
    return EXIT_FAILURE;
  }
  if(clasp_checkValue(args, 0, &inName, NULL,
     &arg))
  . . .
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}
. . .
```

Although the code is a lot clearer, we've still not much reduced the number of source lines. Now is a good time for me to foreshadow one of the themes of this study: considering how much of program source is dedicated to (uninteresting) boilerplate.

The still-small actual 'doing' logic – the **for**-loop – is drowning in a much bigger function steeped in 'deciding' logic – the command-line handling – and support/boilerplate logic. The delineations between the deciding and the doing, and the interesting and the uninteresting, are points of interest in program anatomy.

Step 6 – Implementing '--help' flag

It's time to start handling some flags, starting with the conventional '--help' flag: display usage information and quit. Listing 6 shows the differential changes for Step 6.

Disregarding the somewhat alarming use of magic numbers, and assuming your willingness to read the CLASP docs for `clasp_showUsage()`, this should be reasonably easy to understand. But none of it's interesting: nothing more than more boilerplate.

I've also corrected an earlier oversight: So far, passing an unrecognised flag to the program will be treated as a file name (Steps 1–3) or silently ignored (Steps 4–5), neither of which is appropriate. We need to employ `clasp_reportUnusedFlagsAndOptions()` (see [4]), as shown, after all known flags/options are processed explicitly.

Packaging concerns

As of Step 6, only 8 of the 100+ source code lines are actually to do with the business of swapping slashes! Even though that's probably a smaller ratio than would be the case in most programs of greater complexity of purpose, it's still not unrepresentatively low. From my experience in writing CLI programs, it's invariably the case that the wood is obscured by the trees. And this brings me back to two of my areas of interest: coupling and code generation.

As I mentioned in the introduction, I want to be able to start to think less about fundamental issues upon which there is general agreement, and to update my long-in-the-tooth code generation wizards accordingly, and, as a consequence of both, write better software more rapidly.

I've been threatening *C Vu* with writing a series of articles on program anatomy for an embarrassingly long time now, and despite my procrastinations (righteous and otherwise) have been thinking about the subject a lot. Consequently, I've come to the position that all CLI program logic that is 'written' by the author – i.e. is not part of standard, system,or third-party libraries; this includes code that might be wizard-generated at the author's behest – can be considered to comprise the following behavioural/anatomical groups:

- **Decision logic**: the code that works out what needs to be done and which component(s) will do it;
- **Action logic**: the code that does the work deemed necessary by the decision-logic; and
- **Support logic**: all the other stuff, including command-line parsing, diagnostic logging, and so forth.

Of course, now I've said it, it looks blindingly obvious, and not the least original. Furthermore, it's very likely to apply, albeit with differences, to other types of link-units; I've just not given them as much thought yet, so don't want to jump the gun.

But the point I want to proselytise in this article (and the others that'll look at different types of link-units and different languages) is that it's not just a *thinking* taxonomy: it's a *doing* one.

Let's consider again our little **slsw** program. As of Step 6 we can divide the code into the three groups as follows:

- Decision:
  - Detection of the '--help' flag, and invocation of third-party (CLASP) library functions to respond; or
  - Detection of 0–2 command-line values, and invocation of third-party (CLASP) and standard library functions to open named files (and deal with failure to do so);
  - Invocation of the `fgetc()` / `fputc()` `for`-loop; and
  - Issuing of return value `EXIT_SUCCESS`.
- Action:
  - The `fgetc()` / `fputc()` `for`-loop; and
  - The invocation of `clasp_showUsage()`.
- Support:
  - All 6 `#include`s;
  - Definition of aliases array; and
  - All of `main()`.

One could argue that detecting and handling the '--help' flag could, by virtue of its conventional nature, be classed as support logic, but I don't think that's helpful. Rather, it's decision and action logic that can be wizard generated.

Listing 7

```
. . .

static
void show_help(FILE* stm);

static
int clasp_main(clasp_arguments_t const* args)
{
  . . .
  if(clasp_flagIsSpecified(args, "--help"))
  {
    show_help(stdout);
    return EXIT_SUCCESS;
  }
  if(clasp_checkValue(args, 0, &inName, NULL,
                      &arg))
  . . .
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}
. . .
static
void show_help(FILE* stm)
{
  clasp_showUsage(
    NULL
  , aliases
  . . .
  );
}
```

Step 7 – Abstracting out '--help' action logic

The obvious next step is to implement '--version'. But if we follow what was done for '--help' that's going to pad out our 'main' (`clasp_main()`) even more. Innately (or experientially, at least) I have qualms about the anatomy of the program as it stands: all logic is clumped together. So, let's first start making things a bit more transparent by abstracting out the '--help' implementation – the action logic – into a worker function `show_help()` (see Listing 7).

Step 8 – Implementing '--version' flag

Given the foregoing two steps, the implementation of the '--version' flag is simple and obvious, as shown in Listing 8. Take note that the major, minor, and revision version numbers – 0, 1, and X (currently 8) – are specified in two places! This is a clear violation of DRY SPOT ([5], [6], [7]), and it won't surprise you in the least to learn that I actually fluffed it and got them out of step during the development. We'll deal with this soon, after we deal with the problem that our slash swapping action logic is drowning in a sea of `main()`s.

Step 9 – Abstracting out slash-swapping action logic

You can really see the influence of revisibility in this one: I've abstracted out the slash-swapping action logic into the `slsw()` function by providing a forward function declaration and then extract-as-function refactored right where it sits, as shown in Listing 9. Now `clasp_main()` is almost entirely, 'cleanly', composed of decision logic: I think that's a major

Listing 8

```
. . .

static clasp_alias_t aliases[] =
{
  CLASP_FLAG(NULL, "--help",
             "invokes this help and terminates"),
  CLASP_FLAG(NULL, "--version",
             "displays version and terminates"),
  CLASP_ALIAS_ARRAY_TERMINATOR
};

static
void show_help(FILE* stm);
static
void show_version(FILE* stm);

static
int clasp_main(clasp_arguments_t const* args)
{
  . . .
  if(clasp_flagIsSpecified(args, "--help"))
  {
    show_help(stdout);
    return EXIT_SUCCESS;
  }
  if(clasp_flagIsSpecified(args, "--version"))
  {
    show_version(stdout);
    return EXIT_SUCCESS;
  }
  if(clasp_checkValue(args, 0, &inName,
                      NULL, &arg))
  . . .
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}
. . .
static
void show_help(FILE* stm)
. . .
static
void show_version(FILE* stm)
{
  clasp_showVersion(
    NULL
  , "slsw"
  , 0
  , 1
  , 8
  , clasp_showVersionByFILE
  , stm
  , 0
  );
}
```

improvement, and sits well with our understanding of its purpose in deciding *what* to do, and not worrying about *how* that's done.

Note that every code change since Step 1 is code that can, and probably should, be generated by a wizard.

Step 10 – Windows-compatible swapping

So far, we've spent most of our time looking at command-line handling, and haven't taken a look at all at the slash-swapping action logic itself. One of the first things that jumps out is that it is UNIX-specific: it assumes that backslashes are 'wrong' and forward slashes are 'right'. (To be sure, this is true, but it's not the view of the entire computational world.) We can

Listing 9

```
. . .
static clasp_alias_t aliases[] =
. . .
static
int slsw(
  FILE* in
, FILE* out
);
static
void show_help(FILE* stm);
static
void show_version(FILE* stm);

static
int clasp_main(clasp_arguments_t const* args)
{
  . . .
  if(clasp_checkValue(args, 1, &outName,
                      NULL, &arg))
  {
    . . .
  }
  return slsw(
    in
  , out
  );
}
static
int slsw(
  FILE* in
, FILE* out
)
{
  int ch;
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}
. . .
```

Listing 10

```
static
int slsw(
  FILE* in
, FILE* out
)
{
#if defined(_WIN32)
# define  SLSW_AMBIENT_CHAR_  '\\'
# define  SLSW_ALT_CHAR_      '/'
#elif defined(UNIX) || \
      defined(unix)
# define  SLSW_AMBIENT_CHAR_  '/'
# define  SLSW_ALT_CHAR_      '\\'
#else
# error Operating-system not discriminated
#endif
  char const srch = SLSW_ALT_CHAR_;
  char const repl = SLSW_AMBIENT_CHAR_;
  int ch;
  for(; EOF != (ch = fgetc(in)); )
  {
    if(srch == ch)
    {
      ch = repl;
    }
    fputc(ch, out);
  }
  return EXIT_SUCCESS;
}
```

Listing 11

```
. . .
static clasp_alias_t aliases[] =
{
  CLASP_FLAG("-r", "--reverse",
     "reverses the swapping from non-ambient=>
     ambient to ambient=>non-ambient"),
  CLASP_FLAG(NULL, "--help",
     "invokes this help and terminates"),
  CLASP_FLAG(NULL, "--version",
     "displays version and terminates"),
  CLASP_ALIAS_ARRAY_TERMINATOR
};
static
int slsw(
  FILE* in
, FILE* out
, int   reverse
);
. . .
static
int clasp_main(clasp_arguments_t const* args)
{
  . . .
  clasp_argument_t const* arg;
  int reverse = 0;
  if(clasp_flagIsSpecified(args, "--help"))
  . . .
  if(clasp_flagIsSpecified(args, "--version"))
  . . .
  reverse = clasp_flagIsSpecified(args,
                                  "--reverse");
  if(0 != clasp_reportUnusedFlagsAndOptions(args,
     &arg, 0))
  . . .
  if(clasp_checkValue(args, 1, &outName,
                      NULL, &arg))
  . . .
  return slsw(
    in
  , out
  , reverse
  );
}
static
int slsw(
  FILE* in
, FILE* out
, int   reverse
)
{
#if defined(_WIN32)
# define  SLSW_AMBIENT_CHAR_  '\\'
# define  SLSW_ALT_CHAR_      '/'
#elif defined(UNIX) || \
      defined(unix)
# define  SLSW_AMBIENT_CHAR_  '/'
# define  SLSW_ALT_CHAR_      '\\'
#else
# error Operating-system not discriminated
#endif

  char const srch = reverse ? SLSW_AMBIENT_CHAR_ :
     SLSW_ALT_CHAR_;
  char const repl = reverse ? SLSW_ALT_CHAR_ :
     SLSW_AMBIENT_CHAR_;
  int ch;
  for(; EOF != (ch = fgetc(in)); )
  . . .
  return EXIT_SUCCESS;
}
```

address this within the newly separated **`slsw()`** function, as shown in Listing 10.

Step 11 – Implementing '--reverse' flag

Having got to an implementation of **`slsw()`** that works correctly on both UNIX and Windows, it's now time to provide the more sophisticated behaviour that is provided by the extant **slsw** tool: to be able to 'reverse' the ambient swapping (something that is very useful when writing on one operating system about coding on another, as it happens). We'll support this by adding support for a '--reverse' flag, as shown in Listing 11.

Step 12 – Added '--mode' option, and sophisticated behaviour

Of course, once you start to add sophistication, it's often tempting to add more. We can readily imagine a future version of such a tool needing to

Listing 12

```
. . .
static clasp_alias_t aliases[] =
{
  CLASP_OPTION("-m", "--mode",
     "specifies the mode for slash swapping.
     'ambient' changes non-ambient slashes to
     ambient slashes, and is the default if mode
     not specified; 'back' changes slashes to
     backslashes; 'forward' changes backslashes
     to slashes; 'invert' inverts all slashes;
     'reverse' does the opposite of 'ambient'.",
     "|ambient|back|forward|invert|reverse"),
  CLASP_OPTION_ALIAS("-a", "--mode=ambient"),
  CLASP_OPTION_ALIAS("-b", "--mode=back"),
  CLASP_OPTION_ALIAS("-f", "--mode=forward"),
  CLASP_OPTION_ALIAS("-i", "--mode=invert"),
  CLASP_OPTION_ALIAS("-r", "--mode=reverse"),
#ifndef SLSW_NO_BACKWARDS_COMPATIBILITY
  CLASP_OPTION_ALIAS("--reverse",
     "--mode=reverse"),
     /* backwards compatibility */
#endif
     /* SLSW_NO_BACKWARDS_COMPATIBILITY */
  CLASP_FLAG(NULL, "--help",
             "invokes this help and terminates"),
  CLASP_FLAG(NULL, "--version",
             "displays version and terminates"),
  CLASP_ALIAS_ARRAY_TERMINATOR
};
/* detect operating system */
#if defined(_WIN32)
# define        SLSW_OS_IS_WINDOWS
#elif defined(UNIX) || \
      defined(unix)
# define        SLSW_OS_IS_UNIX
#else
# error Operating-system not discriminated
#endif
enum slsw_mode_t
{
  /* pseudo-modes */
  SLSW_MODE_AMBIENT = 0,
  SLSW_MODE_REVERSE,
  /* real modes */
  SLSW_MODE_INVERT,
#ifdef SLSW_OS_IS_UNIX
  SLSW_MODE_B2F     = SLSW_MODE_AMBIENT,
  SLSW_MODE_F2B     = SLSW_MODE_REVERSE,
#endif
#ifdef SLSW_OS_IS_WINDOWS
  SLSW_MODE_B2F     = SLSW_MODE_REVERSE,
  SLSW_MODE_F2B     = SLSW_MODE_AMBIENT,
```

Listing 12 (cont'd)

```
#endif
  SLSW_MAX_VALUE
};

typedef enum slsw_mode_t slsw_mode_t;
static
int slsw(
  FILE*       in
, FILE*       out
, slsw_mode_t mode
);
. . .
static
int clasp_main(clasp_arguments_t const* args)
{
  . . .
  clasp_argument_t const* arg;
  slsw_mode_t mode    = SLSW_MODE_AMBIENT;
  if(clasp_flagIsSpecified(args, "--help"))
  . . .
  if(clasp_flagIsSpecified(args, "--version"))
  . . .

  arg = clasp_findFlagOrOption(args, "--mode", 0);
  if(NULL != arg)
  {
    if(0 == strcmp(arg->value.ptr, "ambient")) {
       mode = SLSW_MODE_AMBIENT; }
    else
    if(0 == strcmp(arg->value.ptr, "back")) {
       mode = SLSW_MODE_F2B; }
    else
    if(0 == strcmp(arg->value.ptr, "forward")) {
       mode = SLSW_MODE_B2F; }
    else
    if(0 == strcmp(arg->value.ptr, "invert")) {
       mode = SLSW_MODE_INVERT; }
    else
    if(0 == strcmp(arg->value.ptr, "reverse")) {
       mode = SLSW_MODE_REVERSE; }
    else
    {
      fprintf(stderr,
              "slsw: invalid mode specified\n");
      return EXIT_FAILURE;
    }
  }
  if(0 != clasp_reportUnusedFlagsAndOptions(args,
     &arg, 0))
  . . .
  if(clasp_checkValue(args, 1, &outName, NULL,
     &arg))
  . . .
  return slsw(
    in
  , out
  , mode
  );
}
static
int slsw(
  FILE*       in
, FILE*       out
, slsw_mode_t mode
)
{
#ifdef SLSW_OS_IS_WINDOWS
# define  SLSW_AMBIENT_CHAR_  '\\'
# define  SLSW_ALT_CHAR_      '/'
#endif
#ifdef SLSW_OS_IS_UNIX
# define  SLSW_AMBIENT_CHAR_  '/'
# define  SLSW_ALT_CHAR_      '\\'
#endif
  int ch;
  for(; EOF != (ch = fgetc(in)); )
  {
    switch(ch)
    {
      case  SLSW_AMBIENT_CHAR_:
        if(SLSW_MODE_AMBIENT != mode)
        {
          ch = SLSW_ALT_CHAR_;
        }
        break;
      case  SLSW_ALT_CHAR_:
        if(SLSW_MODE_REVERSE != mode)
        {
          ch = SLSW_AMBIENT_CHAR_;
        }
        break;
    }
    fputc(ch, out);
  }
  return EXIT_SUCCESS;
}

. . .
```

Listing 12 (cont'd)

expand its abilities as illustrated by Listing 12: in addition to the existing 'ambient' and 'reverse' modes, it now also supports 'backward' and 'forward' slashes, and inverting of whatever is encountered, all via the new '--mode' option. Each mode has a flag alias, and the '--reverse' flag becomes a flag alias for backwards-compatibility.

I leave as an exercise for the reader an examination of the new action logic – I do rather like the clever interplay 'twixt enumerator values and **switch**, but I'm probably kidding myself – and instead point out how the revisibility is pretty good for such a large change. In and of itself it doesn't make the code good, but it does help to follow what's happening, which we might presume is an indirect aid to software quality.

Step 13 – Added precondition enforcements to slsw()

Having separated the action logic into a separate function, it behoves us to enforce precondition enforcements. The precondition is simple: neither **in** nor **out** can be **NULL**. It is enforced by the standard function-like macro **assert()**, introduced by <assert.h>. For brevity, no listing is shown of the changes.

Step 14 – Handling DRY SPOT violations

Now to tackle another of the issues that are important to program anatomy: DRY SPOT violations! Specifically, there are four outright violations, and one somewhat subtle one. The outright violations are the multiple uses of literals – '**slsw**', **0**, **1**, and **15** (now **16**) – for specifying program name and version numbers. The subtle one is the widespread further use of the string **"slsw"** within various longer literal strings (used for contingent reports). If we choose to change the program name in the future, we'd better hope to be using a good search-replace tool. Better to DRY it now, and have a SPOT.

This was easy to achieve in this case (see Listing 13) via the four object-like macros **PROGRAM_NAME**, **PROGRAM_VER_MAJOR**, **PROGRAM_VER_MINOR**, and **PROGRAM_VER_REVISION**. That ease is, in part, due to the simplicity of **slsw**: it is written in C; it is a standalone tool; it does not (yet) use diagnostic logging; the version/usage information is statically determined.

Listing 13

```
. . .
#include <string.h>

#define PROGRAM_NAME          "slsw"
#define PROGRAM_VER_MAJOR     0
#define PROGRAM_VER_MINOR     1
#define PROGRAM_VER_REVISION  16
static clasp_alias_t aliases[] =
. . .
static

int clasp_main(clasp_arguments_t const* args)
{
  . . .
  arg = clasp_findFlagOrOption(args, "--mode", 0);
  if(NULL != arg)
  {
    if(0 == strcmp(arg->value.ptr, "ambient")) {
       mode = SLSW_MODE_AMBIENT; }
    . . .
    else
    {
      fprintf(
        stderr
      , "%s: invalid mode specified\n"
      , PROGRAM_NAME
      );
      return EXIT_FAILURE;
    }
  }
  if(0 != clasp_reportUnusedFlagsAndOptions(args,
     &arg, 0))
  {
    fprintf(
      stderr
    , "%s: unrecognised argument: %s\n"
    , PROGRAM_NAME
    , arg->givenName.ptr
    );
    return EXIT_FAILURE;
  }
  if(clasp_checkValue(args, 0, &inName, NULL,
                      &arg))
  {
    if(0 == arg->givenName.len)
    {
      if(NULL == (in = fopen(inName, "r")))
      {
        int const e = errno;
        fprintf(
          stderr
        , "%s: could not open '%s' for read
          access: %s\n"
        , PROGRAM_NAME
        , inName
        , strerror(e)
        );
        return EXIT_FAILURE;
      }
    }
  }
  if(clasp_checkValue(args, 1, &outName, NULL,
                      &arg))
  {
    if(0 == arg->givenName.len)
    {
      if(NULL == (out = fopen(outName, "w")))
      {
        int const e = errno;
```

Listing 13 (cont'd)

```
        fprintf(
          stderr
        , "%s: could not open '%s' for write
          access: %s\n"
        , PROGRAM_NAME
        , outName
        , strerror(e)
        );
        return EXIT_FAILURE;
      }
    }
  }
  . . .
}
static
int slsw(
  FILE*       in
, FILE*       out
, slsw_mode_t mode
)
. . .
int main(int argc, char** argv)
{
  int const clflags = 0
     | CLASP_F_TREAT_SINGLEHYPHEN_AS_VALUE
     ;
  return clasp_main_invoke(argc, argv,
    clasp_main, PROGRAM_NAME, aliases, clflags,
    NULL);
}
static
void show_help(FILE* stm)
{
  clasp_showUsage(
    NULL
  , aliases
  , PROGRAM_NAME
  , "Synesis Software SystemTools
    (http://synesis.com.au/systools)"
  , NULL
  , "Swaps slashes in text"
  , PROGRAM_NAME " [ ... options ... ]
    [<input-file>|-] [<output-file>|-]"
  , PROGRAM_VER_MAJOR
  , PROGRAM_VER_MINOR
  , PROGRAM_VER_REVISION
  , clasp_showHeaderByFILE
  , clasp_showBodyByFILE
  , stm
  , 0
  , 76
  , -4
  , 1
  );
}
static
void show_version(FILE* stm)
{
  clasp_showVersion(
    NULL
  , PROGRAM_NAME
  , PROGRAM_VER_MAJOR
  , PROGRAM_VER_MINOR
  , PROGRAM_VER_REVISION
  , clasp_showVersionByFILE
  , stm
  , 0
  );
}
```

Listing 14

```
. . .
typedef enum slsw_mode_t slsw_mode_t;
/** Swaps slashes in \c in to \c out,
 *  according to \c mode
 *
 * \retval 0 The function succeeded
 * \retval !0 The function failed. errno will
 *  indicate reason
 *
 * \pre (NULL != in)
 * \pre (NULL != out)
 */
int slsw(
  FILE*       in
, FILE*       out
, slsw_mode_t mode
);
```

Step 15 – Splitting into library and main: 'program design is library design'

Over the years, I've misremembered a Bjarne Stroustrup quote of longstanding. With the assistance of the good folks on ACCU General, I've now ascertained that the original quote is 'language design is library design' (and there's also one that says 'library design is language design', for good measure), which I realise now doesn't really capture what I want to say here.

Listing 15

```
int slsw(
  FILE*       in
, FILE*       out
, slsw_mode_t mode
)
{
#ifdef SLSW_OS_IS_WINDOWS
# define  SLSW_AMBIENT_CHAR_  '\\'
# define  SLSW_ALT_CHAR_      '/'
#endif
#ifdef SLSW_OS_IS_UNIX
# define  SLSW_AMBIENT_CHAR_  '/'
# define  SLSW_ALT_CHAR_      '\\'
#endif
  int ch;
  assert(NULL != in);
  assert(NULL != out);
  for(; EOF != (ch = fgetc(in)); )
  {
    switch(ch)
    {
      case  SLSW_AMBIENT_CHAR_:
        if(SLSW_MODE_AMBIENT != mode)
        {
          ch = SLSW_ALT_CHAR_;
        }
        break;
      case  SLSW_ALT_CHAR_:
        if(SLSW_MODE_REVERSE != mode)
        {
          ch = SLSW_AMBIENT_CHAR_;
        }
        break;
    }
    if(ch != fputc(ch, out))
    {
      return -1;
    }
  }
  if(ferror(in))
  {
    return -1;
  }
  return 0;
}
```

Listing 16

```
static
int clasp_main(clasp_arguments_t const* args)
{
  . . .
  if(clasp_checkValue(args, 0, &inName, NULL,
                      &arg))
  . . .
  if(clasp_checkValue(args, 1, &outName, NULL,
                      &arg))
  . . .
  if(0 == slsw(
    in
  , out
  , mode
  ))
  {
    return EXIT_SUCCESS;
  }
  else
  {
    int const e = errno;
    fprintf(
      stderr
    , "%s: failed to complete slash-swapping:
      %s\n"
    , PROGRAM_NAME
    , strerror(e)
    );
    return EXIT_FAILURE;
  }
}
```

Instead, I'm starting my own quote about a form of good practice in program design: 'program design is library design'. You all have my express permission to propagate this to the end of time (with due attribution ☺).

Let's now split up the code we've arrived at thus far along the lines of decision vs action logic, giving three files: `slsw.h`, `slsw.c`, and `main.c`.

`slsw.h` contains the following:

- a **`#include`** for `stdio.h` (because **`slsw()`** references the **`FILE`** type);
- operating system discrimination (because **`slsw_mode_t`** requires it);
- definition of the **`slsw_mode_t`** enumeration; and
- declaration of the **`slsw()`** function.

`slsw.c` contains the following:

- required **`#include`**s, starting with `slsw.h`; and
- implementation of the **`slsw()`** function.

`main.c` contains the following (as it did previously):

- required **`#include`**s: `slsw.h`; then CLASP headers; then standard headers; and
- SPOTs for **`PROGRAM_NAME`**, etc;
- **`aliases`** array;
- forward declarations for **`show_help()`** and **`show_version()`**;
- **`clasp_main()`**;
- **`main()`**; and
- implementations of **`show_help()`** and **`show_version()`**.

I hope it's now clear that the `slsw.h` (declarations) and `slsw.c` (implementation) together form a library, which can be used independently of any notion of CLI (or any other particular) execution context. As well as being used within the **slsw** program, the library may be reused by other programs (e.g. **slswgui**), and, of particular importance for software quality, in automated test harnesses.

Step 16 – Fixing up coupling and semantics of slsw()

Now we've abstracted **`slsw()`** into a separate file, its coupling – both physical and semantic – to the command-line is evident: it relies on `stdlib.h` and its return value is **`EXIT_SUCCESS`** (or, by implication, **`EXIT_FAILURE`**). This is wrong.

We can fix this very easily, simply by changing it to return 0 for success, and non-0 for failure, relying on **`errno`** (as set by **`fgetc()`** or **`fputc()`**) for more detailed failure information, as shown in Listings 14–16. As you may know, gentle readers, the standard requires that a program return value of **`EXIT_SUCCESS`** is treated as equivalent to 0, and that **`EXIT_FAILURE`** is not 0. So we've cunningly done nothing to reduce backwards-compatibility while reducing coupling. Which is nice.

Note that **`slsw()`** is simple enough that we don't have to do diagnostic logging and contingent reporting here (though even in this we lose the knowledge of whether it's input (**`fgetc()`**) or output (**`fputc()`**) that fails. More complex action-logic components may have to use more complex failure reporting to their decision-logic callers, including process/thread-global error state variables (a la **`errno`**), return codes, exceptions, callbacks, diagnostic logging and contingent reports.

Summary

This article has examined the incremental development of a simple but real program written in C as a basis for analysis of some of the issues pertaining to CLI program anatomy. In particular, it has discussed the delineation of program logic into decision, action, and support, and demonstrated how separation of the code on such lines brings several benefits: separation of the action logic into a library increases clarity, scope for reuse, testability, and modularity. This principle of *program design is library design* will be a constant feature of the series.

As a by-product of this exercise, the article has also provided a simple example of function layering: simplifying a large and complex **`main()`** by abstracting out the boilerplate support logic in the form of a function to which we pass the address of a smaller, specific 'main'. Subsequent articles will consider how other services can be initialised in a similar manner, enabling access to sophisticated (albeit uninteresting) functionality with minimal intrusion into the code, preferably in a way that can be wizard-generated.

Finally, the article described the identification and elimination of sources of repetition in the program name and version numbers. In the simple case presented, these 'identity attributes' were defined as pre-processor object-like macros. Subsequent articles will consider alternatives, reflecting requirements of language and good practice as well as considering how such attributes may be obtained dynamically (such as from a program's Windows version resource), and how (and when) they must be defined to interact reliably with the phases of 'main's and various support services.

Next

In the next article, I will complete the look at CLI programs written in C, including those that are much bigger than **slsw**, spanning multiple source files. I will then turn to the subject of CLI programs written in C++, and discuss the advantages and disadvantages as compared to C: by then, all being well, I will have kept my writing momentum up and completed the next 'Quality Matters' instalment – the third in the series on C++ exceptions – for the next issue of *Overload* and will be able to draw on that also, and so keep down the length.

Further issues of interest to be covered in the next article will include some/all of the following:

- Character encodings – multibyte and/or widestring;
- Removable Diagnostic Measures – how to facilitate high quality software without undue coupling;
- Names – for identity attributes, for namespaces, for files, for project-related directories;
- Directories – where to place the decision logic, action logic, the support logic, and the project files;
- Testing – how much can be auto-generated by the wizard; and
- Function Layering to the Max!

Finally, before the next article in the series I intend to complete the first wizard rewrite, encapsulating all the issues discussed herein, and hope to be able to report back on being able to generate sophisticated, modular, program projects according to the principles and techniques presented thus far. We might even have some downloadable goodies! ■

Acknowledgements

Many thanks go to Chris Oldwood and Garth Lancaster for helping me despite what has become typically eleventh-hour preparation of the draft. Usual thanks/apologies go to Steve Love. I'd promise to write the next article in plenty of time, but he knows I'd find some reason to break it. Ah well.

References

[1] 'Quality Matters, Part 5: Exceptions: The Worst Form of 'Error' Handling, Except For All The Others', Matthew Wilson, *Overload* 98, October 2010

[2] *Working Effectively with Legacy Code*, Michael Feathers, Pearson, 2004

[3] *A Practical Guide to Linux*, Mark G. Sobell, Prentice Hall, 2005

[4] 'An Introduction to CLASP', Matthew Wilson, *CVu*, volume 23 number 6, January 2012

[5] *The Pragmatic Programmer*, Andy Hunt and Dave Thomas, Addison-Wesley, 1999.

[6] *Imperfect C++*, Matthew Wilson, Addison-Wesley, 2004

[7] *The Art of UNIX Programming*, Eric S. Raymond, Addison-Wesley, 2003

[8] CLASP is an open-source library for Command-Line Argument Sorting and Parsing, available via the Subversion respository at http://sourceforge.net/projects/systemtools

Patterns and Active Patterns

Richard Polton continues to explore how functional style can improve imperative programs.

Suppose we have branching code which is used to declare a variable, **x**.

```
var x = new X();
if( condition(a))
  x = x1;
else
x = x2;
```

We can improve this code because we have the ternary operator **?**: this allows us to write declarative code such as

```
var x = condition(a) ? x1 : x2;
```

but this doesn't allow us to replace something as simple as

```
var y = new X();
if( condition(a))
{
  log("Condition was true");
  y = x1;
}
else
{
  y = x2;
}
```

A first attempt looks like

```
Func<X> f = () =>
{
  log("Condition was true");
  return x1;
};
var y = condition(a) ? f() : x2;
```

which is acceptable in this simple case.

What we want is an **if** function. But recall the exercise from SICP [1] where the reader was invited to create such a function. The trap for the unwary lies in the nature of the function parameters. As C# et al. are immediate languages, ie not lazy, then the function parameters are evaluated before the function is called. (The alternative would be lazy evaluation in which the function parameters are not evaluated until they are used, within the function body.)

The naive implementation fails this requirement:

```
public static B if_<A,B>(bool b, B thenValue,
                         B elseValue)
```

In this case, **b**, **thenValue** and **elseValue** will all be evaluated before the choice of branch is taken. Instead

```
public static B if_<A,B>(Predicate<A> predicate,
  Func<A,B> thenClause,
  Func<A,B> elseClause, A input)
```

has the desired effect. Typical usage might be

```
var b = if_<A,B>(a=>condition(a),
// which could be replaced in this instance with
// the method group 'condition'
  i=>{log("Condition was true"); return x1;},
  i=>x2,
  a);
```

This is just fine and dandy, until we come to a sequence of **if ... else if ... else if ... else** constructs. Of course this is representable in this new format but it becomes quite deeply nested quickly. However, we could observe that this construct is not dissimilar to a switch block. So let's reproduce the C# **switch** statement now.

```
public static B switch_<A,B>(A input, A label1,
  Func<A,B> case1, A label2, Func<A,B> case2,
  Func<A,B> defaultCase)
```

As you can see, there is a problem with the arbitrary number of pairs of labels and cases. After taking a moment for reflection, we can see that what this function requires is a container of pairs followed by the default case and the input variable. This can be achieved by using a helper function, **toCase**:

```
public static tuple2<A,Func<A,B>> toCase<A,B>
    (A label,Func<A,B> caseN)
```

and so the declaration of **switch_** becomes

```
public static B switch_<A,B>(A input,
  IEnumerable<tuple2<A,Func<A,B>>> cases,
  Func<A,B> defaultCase)
```

Then all that is required of the programmer is an anonymous array declaration:

```
var z = switch_<A,B>(a, new []{
  toCase("a", i=>i+","),
  toCase("B", i=>i.ToLower())},
  i=>string.Empty);
```

Now that we have a simple **switch** function, let us look at the **if ... else if ... else if ... else** construct.

```
var x = new X();
if(condition1(a))
  x = x1;
else if(condition2(a))
  x = x2;
else if(condition3(a))
  x = x3;
else
  x = new X();
```

At first glance, this is not representable as a **switch** block because the conditions are not generally known at compile-time. However, in our declarative **switch_** function, we do not require this restriction. Instead, we modify the signature slightly such that the labels are functions and we have

```
public static B switch_<A,B>(A input,
    IEnumerable<tuple2<Predicate<A>,
    Func<A,B>>> cases, Func<A,B>defaultCase)
```

and

```
public static tuple2<Predicate<A>,
    Func<A,B>> toCase<A,B>(Predicate<A> label,
                           Func<A,B> caseN)
```

RICHARD POLTON

Richard has enjoyed functional programming ever since discovering SICP and feels heartened that programming languages are evolving back to LISP. He likes 'making it better' and enjoys riding his bike when he can't. He can be contacted at richard.polton@shaftesbury.me

Listing 1

```
Dictionary<A,B> d;
var x = new C();
Func<Dictionary<A,B>,C> doA;
Func<Dictionary<A,B>,C> doB;
Func<Dictionary<A,B>,C> doC;
Func<Dictionary<A,B>,C> doD;

if(d.ContainsKey("A"))
    x = doA(d);
else if(d.ContainsKey("B"))
    x = doB(d);
else if(d.ContainsKey("C"))
    x = doC(d);
else
    x = doD(d);
```

which can be used like this

```
var x = switch_<A,B>(a, new[]{
  toCase<A,B>(condition1,i=>x1),
  toCase<A,B>(condition2,i=>x2),
  toCase<A,B>(condition3,i=>x3)},
  i=>new X());
```

Unfortunately, in .NET 3.5, it would appear to be necessary to specify either the type of the generic parameters or the type of the predicate lambda function parameter.

Nevertheless, this is remarkably close to a clever construct in F# called an 'Active Pattern' which I shall describe shortly.

Now, suppose you have code like Listing 1.

Clearly, this has the form of a **switch** construct but, because the labels are not compile-time constants, we cannot use the C# **switch** syntax, instead having to resort to a series of **if .. else if .. else**. In itself this would be acceptable were it not for the fact that the variable **x** is declared and then re-initialised with the results of the appropriate **do** function. There are a number of ways of modifying this code without changing the behaviour. For example, we could store all the **do** functions in an array and then to initialise **x**

```
var doXs = new Dictionary<string,
           Func <Dictionary<A,B>,C>>{
  {"A",doA}, {"B",doB}, {"C",doC}, {"",doD}
};
var x = doXs[key](d)
```

but we need a practical way to initialise the default case in the dictionary of results. Alternatively, we could use the **switch_** defined above.

```
var x = switch_(d, new[]{
  toCase(dict=>dict.ContainsKey("A"),doA),
  toCase(dict=>dict.ContainsKey("B"),doB),
  toCase(dict=>dict.ContainsKey("C"),doC)},
  doD);
```

This has the advantage of retaining the original feel, more or less, of the C# switch statement whilst also being a declarative initialiser.

However, this example is still a little cumbersome if we wish to pass the value from the dictionary into the case function. In the current example, the value has been encoded into the **do** function, eg **doA** knows that the **key** was **A**. We could work around this by passing the **key** in the function parameters and the making use of partial function application (Listing 2).

Listing 2

```
Func<B,C> transform;
Func<A, Func<Dictionary<A,B>, C>> doX = (A key)
  => ((Dictionary<A,B> dict)
  => transform(dict[key]));
var x = switch_(d, new[]{
  toCase(dict=>dict.ContainsKey("A"),doX("A")),
  toCase(dict=>dict.ContainsKey("B"),doX("B")),
  toCase(dict=>dict.ContainsKey("C"),doX("C"))},
  doD);
```

In F#, as mentioned above, there is a construct called the 'Active Pattern'. Here is an example active pattern which I have used in my code.

```
let (|ContainsKey|_|) key
(dict:System.Collections.Generic.Dictionary<_,_>)
= if dict.ContainsKey key then Some(dict.[key])
  else None
```

The analogue of the previous C# code is

```
let x = match d with
| ContainsKey "A" value -> transform value
| ContainsKey "B" value -> transform value
| ContainsKey "C" value -> transform value
| _ -> doD(d)
```

Admittedly the syntax for active patterns can look a little clunky, but they work well.

And now for something completely different....

The .NET framework presents the discriminated union **System.Nullable<T>**. This generic class wraps the value-type **T** and encapsulates the **null / not null** value of the underlying object. Simply put, an object of type **Nullable<T>** either has a value, which resolves to an object of type **T**, or does not have a value. This could be used to replace all of those horrendous and problematic **null** checks but, and this is the key point, **T** must be a value type - which can't be null anyway. The framework as it stands does not provide the analogue for reference types, thus requiring the user to keep track of nulls (and frequently forget to check for them).

And so we create **OptionType<T>** where **T** is a reference type. We create an interface which is similar to that of **Nullable**, but the design decision was taken early on to force the user to make the value check before referencing (This was more of a training exercise than a design necessity). The value check is done in a manner reminiscent of normal reference types, ie if it is accessed when the value is **null** then an exception is thrown. In this way, the users are schooled into checking for **null** and, in general, function return types as there are many vendor API calls which return null to indicate a failure of some description. (See Listing 3.)

In addition to this class we have some utility functions (Listing 4).

The implicit function in **OptionType** is used to wrap existing types without having to resort to the usual C# level of verbosity. Therefore, we can call a vendor API function which returns a **T**, say, and is known to return **null** on occasion and capture it in a local variable of type **OptionType<T>**. For example, suppose we have

```
public class T1
{
  public static T1 create() {}
}
```

then we might use

```
OptionType<T1> myT = T1.create();
if( myT.None) return; // The API returned null
  myT.Some.doSomething();
```

It's a little clunky because the user is forced to dereference using **Some**. The possibility of making this function implicit was considered initially but it was discounted because **OptionType** was created to encourage thinking explicitly about return types.

We can also adjust our own API functions to use **OptionType<T>** instead of **T**. If we do this then we must remember our own rule, check for invalid inputs. To this end, a further function has been created in the **Check** class, namely **IsNullOrNone**, which expects an object of **OptionType<T>** and returns **bool**.

```
public class Check
{
  public static bool
    IsNullOrNone<T>(OptionType<T> t)
    where
      T : class { return t == null || t.None; }
}
```

Listing 3

```
public class EmptyOptionTypeException :
   Exception {}
// Use OptionType<T> when the reference type T
// could be null.
// It behaves likesystem.Nullable<T>.
// If you want to use this for Value Types, use
// System.Nullable<T>
// Also, OptionType<string> behaves slightly
// differently, in that the empty string is also
// considered as an empty OptionType,
// ie OptionType<string>(string.Empty).None
// is true.
public class OptionType<T> : IDisposable
   where T:class
{
  private readonly bool _isEmpty = true;
  protected T _t;
  public OptionType() {}
  public OptionType(T t)
  {
    if ((typeof(T) == typeof(string) &&
          Check.IsNotNullString(t as string)) ||
        (typeof(T) != typeof(string) &&
          Check.IsNotNull(t)))
    {
      _t = t;
      _isEmpty = false;
    }
    else
    {
      _isEmpty = true;
    }
  }
  public T Some { get { if (!_isEmpty) return _t;
    else throw new EmptyOptionTypeException();} }
  public bool None { get { return _isEmpty; } }
  public static OptionType<T> Null { get {
    return new OptionType<T>(); } }
  public static implicit operator OptionType<T>
    (T t) { return new OptionType<T>(t);}
  // We do not have an implicit operator T
  // because we want to be explicit about
  // checking for null
  public void Dispose() {
    if (typeof(T) is IDisposable)
    ((IDisposable)_t).Dispose(); }
}
```

Therefore, take our API function

```
public static R DoSomething<T,R>(T input) {
  return new R(input);}
```

and modify it

```
public static OptionType<R>
  DoSomething<T,R>(OptionType<T> input) { return
  Check.IsNullOrNone(input) ?
  OptionType<R>.None : new R(input.Some); }
```

Lovely!

Listing 4

```
// Generic null checking with a special case for
// string, where string.Empty is treated
// equivalently to (string)null.
public static class Check
{
  public static bool IsNull<T>(T t) where T :
class
  {
    return t == null;
  }
  public
     static bool IsNotNull<T>(T t) where T :
class
  {
    return !IsNull(t);
  }

  // In .NET2.0, we need these two because
  // OptionType<T> cannot call IsNull(string),
  // it can only call IsNull<string>(string)
  public static bool IsNullString(string s)
  {
    return IsNull(s);
  }
  public static bool IsNotNullString(string s)
  {
    return IsNotNull(s);
  }
}
```

Of course, now that we have abstracted the `null` checking out of the main code path, we can chain together consecutive API functions that use `OptionType`. For example, where once we had (or should have had if we had performed the checks)

```
T1 myT = T1.create();
if(myT!=null)
{
  R r = DoSomething(myT);
  return r!=null ? r : new R();
}
return new R();
```

we can now write

```
return DoSomething(T1.create());
```

Bootiful! (Of course, we have to understand that `OptionType<R>.None` has the same meaning as the default constructed `R`.)

To conclude, I should mention that all of this comes for free, almost, in F# because the discriminated union type on which this is based works for any type you may decide to use. That is

```
type 'T option =
| None
| Some of 'T
```

References

[1] 'Structure and Interpretation of Computer Programs 2nd Ed', Abelson, Sussman, 1996 MIT Press, also available at http://mitpress.mit.edu/sicp/full-text/book/book.html

If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

Reach us at cvu@accu.org

Keeping up-to-date

Paul Grenyer reflects on what we need to do to stay on top of things.

It is not uncommon for people to ask me how I keep my technical knowledge up-to-date, especially after I've just given a presentation on a subject that is new to them. This is odd because I don't consider myself as someone who does keep up-to-date and many of my presentations are based on things I and other people have been doing for many years, so it's hardly up-to-date.

I've said this many times, but in my experience there are generally two types of people (or developers if you prefer to narrow it further), those that enjoy their work (live-to-work) and those that are just there for the money (work-to-live). I am firmly in the live-to-work camp and very grateful that I am lucky enough to do a job I enjoy every day. Not only do I enjoy what I do at work, I do more of it at home and get together with like-minded people at local specialist interest groups and conferences as often as possible. I'm constantly striving to be the best at what I do and that generally requires a lot of reading (books, the internet, journals, etc). Of course being the best is usually an unachievable goal, but it is important to have something to strive for.

What better way to meet other software engineers that attending local Special Interest Groups (SIG) and conferences

It is unfortunate that, in my experience, most people are in the work-to-live camp. Even developers. Actually, maybe they have it right and have a much better work life balance than I do. However most of those who work-to-live don't do anything in their own time to improve their skills, they don't communicate with other people outside of their organisation and a lot of them don't even look to people within their organisation to try and help themselves improve. They've always done it a certain way and they don't see why they should change now, even if there is an industry out there that has moved on to bigger and better things. I hope it's obvious that these are two extremes of the spectrum and there are plenty of people that fall somewhere in between.

It's clear that to keep up-to-date (for some value of up-to-date) the first hurdle is wanting to. The chances are that if you want to you are closer to the live-to-work end of the spectrum and in my opinion that's the best place to be. After that what do you actually do? Here are some of the things that I do in no particular order.

Read like there is no tomorrow and you have to know everything now. Read books, read journals, read blogs, read the things your friends and colleagues have written. You will learn an extreme amount from reading. Then do. If you read and then don't do you'll forget it.

Write software in your own time. The goal is to be up-to-date and to do that you need to use up-to-date technologies. Find the latest technologies that you are interested in and use them to write software. You'll learn the most if you try and solve a real world problem. Aim to write a complete library, tool or application. By solving real world problems you will learn the most about how to use the technology in a useful way.

When I first joined the ACCU one of the existing members took me under his wing. One of the things he told me was that I needed a website. 'What for?' I asked. He told me it would be for all the articles I was going to write. I didn't think for a moment that I would have anything to write about. Of course I was wrong. Writing turned out to be enjoyable, rewarding and most important of all a learning activity. When you write about something you examine it in detail to make sure you understand it correctly. This usually reveals that you don't, so you look even closer until you do and of course learn much more in the process. Even if you've learnt something it would be a real shame for people not to read what you've written. So write for your website and other websites, write for your blog, write for journals, write for your colleagues. Do everything you can to get get people to read what you've written and make sure you get feedback so you can improve.

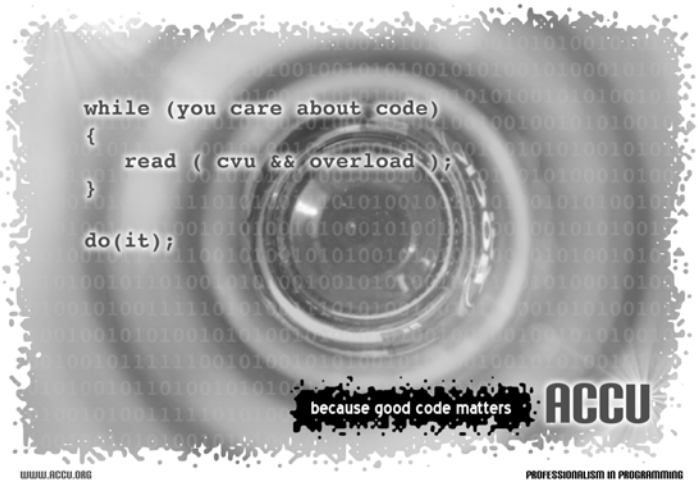
Talk to other software engineers. It's very difficult to learn in a vacuum and arrogant to think that you don't need to speak to other people doing similar things to you to learn. Even if you have other software engineers working with you join software related groups, like the ACCU, where you can talk to other software engineers. Discuss your ideas and problems and ask questions. You'll learn a lot this way and the chances are someone in the group is working with the latest technology that's interesting to you or knows someone else who is.

What better way to meet other software engineers that attending local Special Interest Groups (SIG) and conferences. You can go along and hear about how other people have solved problems with new technologies that interest you and then speak and learn with them afterwards. Writing and giving your own presentation is a great way to learn in the same way as writing articles. People will want to ask you questions so you need to know your subject inside out. People in your audience may know different things to you and answering their questions and interacting with them is another great way to learn.

To keep up-to-date find out what works for you and keep doing it. If something doesn't work for you, stop doing it. Above all learn and have fun. ■

PAUL GRENYER

Paul Grenyer is a husband, father, software consultant, author, testing and agile evangelist. He can be contacted at paul.grenyer@gmail.com



Code Critique Competition 77

Set and collated by Roger Orr. A book prize is awarded for the best entry.

Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Last issue's code

I've written an exception class that can throws itself, can collect a context stack and also can't be ignored as it re-throws itself unless it has been printed at least once. My idea works with Visual Studio but doesn't work reliably with gcc – the exception doesn't always get rethrown – any idea why?

Listing 1

```
#include <iostream>
#include <vector>
// exception class that stacks up context and
// re-throws itself until it is printed by a
// top-level handler.
// use the function operator to add context in
// a catch clause
class exception
{
public:
  exception() {}
  exception(const char *cause)
  : stack(1, cause) {}
  exception(const exception &rhs)
  {
    stack.swap(rhs.stack);
  }
  // throw a copy of myself,
  // if I've not been printed yet.
  ~exception()
  {
    if (rethrow) throw *this;
  }
  // Add context to the exception
  void operator()(const char *context)
  {
    stack.push_back(context);
  }
  // print (and dismiss) the exception
  void print()
  {
    std::copy(stack.begin(), stack.end(),
      std::ostream_iterator<const char *>
      (std::cout, "\n"));
    std::cout << std::flush;
    rethrow = false;
  }
  // like std::exception
  virtual const char *what() const
  {
    return stack[0];
  }
protected:
  mutable std::vector<const char *> stack;
  bool rethrow;
};
```

- Listing 1 is `exception.h`
- Listing 2 (continued on next page) is `test_exception.cpp`

Listing 2

```
#include <iostream>
#include <iterator>
#include "exception.h"
int func(int i)
{
  try
  {
    if (i <= 0)
    {
      throw exception("i must be positive");
    }
    int result(i*i);
    if (result < i)
    {
      throw exception("overflow");
    }
    return result;
  }
  catch (exception & ex)
  {
    ex("in func");
  }
  std::cout << "Shouldn't get here"
            << std::endl;
}
int mid(int i)
{
  try
  {
    return func(1) * func(i);
  }
  catch (exception & ex)
  {
    ex("in mid");
  }
  std::cout << "Shouldn't get here"
            << std::endl;
}
void test(int i)
{
  try
  {
    std::cout << "mid() => " << mid(i)
              << std::endl;
  }
```

ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

Listing 2 (cont'd)

```
  catch (exception & ex)
  {
    std::cout << "Caught exception"
              << std::endl;
    ex.print();
  }
}
int main()
{
  test(0);          // should fail first test
  test(0x7fffffff); // should fail second
}
```

Critiques

Paul Floyd <paulf@free.fr>

First impressions. Two bad smells.

The class **exception** when there is already a class exception in the std namespace. This is the sort of thing that the namespaces are intended to protect us from, but there's no point asking for trouble. All it takes is a user to type **using namespace std;** and there'll be a conflict. An easy way to avoid any clashes with classes in namespace **std** is to start all your class names with a capital letter. So, in this case, I renamed it to **StackContextException**.

The next point had me reaching for Scott Meyers' 'Effective C++' 3e. Item 8: *Prevent exceptions from leaving destructors.* In this case, the design requires the destructor to throw. Basically the problem is that you're likely to end up with an exception thrown whilst an exception is still being handled, which is either undefined behaviour or will cause a program crash.

For the sake of it, I tried to get the program to work with GCC. I noted that the class exception constructors do not initialize **rethrow** (to **true**). I made that change, and also added default constructor calls to stack, which gave me:

```
StackContextException()
 : stack(), rethrow(true) {}
StackContextException(const char *cause)
 : stack(1, cause), rethrow(true) {}
StackContextException(
  const StackContextException &rhs)
 : rethrow(true)
{
  stack.swap(rhs.stack);
}
```

Compiling and running that I get:

```
Caught exception
i must be positive
in func
in mid
Caught exception
overflow
in func
in mid
```

Looks OK, problem solved? Well no, not really. I tried it with Oracle Solaris Studio 12.3. In this case, it gets to here:

```
(dbx) where -h
=>[1] StackContextException::
      StackContextException(this = 0xfee4c56c,
      rhs = CLASS), line 20 in "exception.h"
  [2] StackContextException::
      ~StackContextException(this=0xfee4c50c),
      line 27 in "exception.h"
  [3] __Cimpl::ex_free(0x8046eb8, 0x0, 0x1,
      0x8046ed3, 0x8046ee8, 0xfefc4a84,
      0xfee4c2b4, 0xfee4b940, 0x80528be,
      0x8065688, 0x0),
      at 0xfee34277
  [4] __Crun::ex_clean(0x80470d0, 0x8046f84,
      0xfeffb910, 0x8046ef8, 0xfee4c50c,
      0xfee42a00), at 0xfee34546
  [5] func(i = 0), line 24
      in "test_exception.cpp"
  [6] mid(i = 0), line 34
      in "test_exception.cpp"
  [7] test(i = 0), line 49
      in "test_exception.cpp"
  [8] main(), line 62 in "test_exception.cpp"
```

In that call stack, we have the call to **func()** at level 5, below that at levels 4 and 3 are the C++ runtime library routines for cleaning up the exception that was caught in **func**. Level 2 is the destructor. When I stepped through the assembly, I saw the exception being allocated and the address of **~StackContextException** being pushed on to the stack and then the exception being called. This is the exception cleaning up. At level 1 there is the copy constructor making a copy of the object that is being destroyed in order to throw it. I did try to step through the assembly even more, and waded through some elf, mutex and runtime functions but I was soon out of my depth. The debugger stopped at an abort, with this call stack

```
(dbx) stepi up
signal ABRT (Abort) in __lwp_kill
  at 0xfed4b7c7
0xfed4b7c7: __lwp_kill+0x0007:
  jae     __lwp_kill+0x15 [0xfed4b7d5, .+0xe ]
Current function is func
(dbx) where -h
  [1] __lwp_kill(0x1, 0x6), at 0xfed4b7c7
  [2] _thr_kill(0x1, 0x6), at 0xfed46f29
  [3] raise(0x6), at 0xfecf35f3
  [4] abort(0xfee4c2b0), at 0xfecd2951
  [5] __Cimpl::default_terminate(0x8046e10),
      at 0xfee34cfc
  [6] std::terminate(0xfee4c2b0, 0xfee4b940,
      0x8046e20, 0xfee34695, 0xfee4c520,
      0xfee4b940), at 0xfee3471f
  [7] std::bad_exception::bad_exception(0x1,
      0x6, 0xfedbe000, 0x8046d9c, 0xfecf35f3,
      0x1), at 0xfee34d1c
  [8] std::unexpected(0x8046e48, 0x8046e28,
      0xfee4c210, 0xfee4b940, 0xfee35a38,
      0xfee4c56c), at 0xfee34695
  [9] __Cimpl::ex_unexpected(0x8053f60, 0x0,
      0xfee4c56c, 0x0, 0x8052e88, 0x0),
      at 0xfee34c96
  [10] __Crun::ex_chk_unexpected(0x8046ea8,
      0xfefc4a84, 0x8046e88, 0xfee4b940,
      0xfee34546, 0x8065688), at 0xfee35a38
  [11] __Cimpl::ex_free(0x8046eb8, 0x0, 0x1,
      0x8046ed3, 0x8046ee8, 0xfefc4a84,
      0xfee4c2b4, 0xfee4b940, 0x80528be,
      0x8065688, 0x0), at 0xfee3431a
  [12] __Crun::ex_clean(0x80470d0, 0x8046f84,
      0xfeffb910, 0x8046ef8, 0xfee4c50c,
      0xfee42a00), at 0xfee34546
=>[13] func(i = 0), line 24
       in "test_exception.cpp"
  [14] mid(i = 0), line 34
       in "test_exception.cpp"
  [15] test(i = 0), line 49
       in "test_exception.cpp"
  [16] main(), line 62 in "test_exception.cpp"
```

and the message

```
Exception of type class StackContextException is
unexpected
```

So there we have it. Scott was right. Visual Studio (which I didn't test) and GCC strike it lucky and manage to get an application with undefined behaviour to seem to work correctly. It also worked with code 4's clang++ on Mac OS X. The Oracle compiler gave a program crash (possibly a blessing in disguise).

No compiler warned about throwing from the destructor (not even g++ with `-Weffc++` which warns about some of the items in Meyers' book).

Usually when I want users to sit up and pay attention to an error, I call `abort`.

Huw Lewis <huw.lewis2409@gmail.com>

The headline bug causing the observed problems are the incomplete constructors. Initialisation of the `rethrow` member to `true` solves the issue and the exception is re-thrown reliably.

However, there's more to say about this piece of code.

The exception header file is missing inclusion guards to prevent multiple inclusion. I'd recommend using `#pragma once` for this purpose if working with well known modern compilers. Otherwise use the old style guards:

```
#ifndef BLAH_H
#define BLAH_H
```

The name '`exception`' is unwise as it could lead to confusion or ambiguity with the `std::exception` type. I would recommend renaming this class something like `AccumulativeException` as it is an exception type that accumulates context information.

Throwing an exception from a destructor is normally one of those things you *should* never do. It introduces many problems that are well discussed and documented (search google for destructor throw...):

- a destructor that throws during stack unwinding due to an original exception will result in the program terminating (catastrophe).
- Memory management becomes difficult – a resource leak is likely as the partially destroyed object may be left in limbo.
- Containers of objects that may throw during destruction cannot complete the destruction of its remaining objects or itself – leaving significant resource leakage.

This exception type doesn't come near any of the problem areas and is therefore of the very few conditions where throwing from a destructor could be just about acceptable. I still think it is a little dangerous to rethrow this partially destroyed object. I'd prefer to see a new exception object constructed for the new exception. I've passed into the constructor a non-const reference to stack which can be efficiently swapped into this new object.

On the subject of the destructor – it isn't `virtual`. This in itself isn't an error, but an exception class like this would often be derived from for more specific exception conditions.

```
virtual ~AccumulativeException()
{
  if (rethrow)
    throw AccumulativeException(stack);
}
```

This exception type does not derive from `std::exception`. I have personally had some big headaches over exceptions that aren't derived from any sensible base. It makes it very difficult for a maintainer to work out what is going on if they don't know what exception type they're looking for. It is good practice to always use `std::exception` as the base for any exception type. In this case, we can't have `std::exception` as a base because our own destructor has a looser throw specification than `std::exception` which specifies that its destructor will never throw (unlike our own). This is something we'll just have to live with *as long as we stick with the throwing destructor.*

The storage (in stack) of `const char*` raw pointers means that only statically allocated raw strings can be used with the exception. If any dynamically constructed string were used, the data would be destroyed *before* the exception object and would cause a crash when the now invalid pointer is dereferenced. The stack container should store `std::string` objects to provide this extra flexibility and robustness.

The vector container is a sub-optimal choice of container for '`stack`' as it gets re-allocated in its entirety when extended (resized by each `push_back` call). The `deque` container type is a much better fit.

The `const` correctness of this class is a little screwy. The `stack` member variable has been declared mutable so that the `copy` constructor can modify (swap) the `rhs` parameter's `stack`. It's not wrong, but I don't like it. The client code calling this constructor wouldn't usually expect their original object to be modified by this. I'd suggest making the plain copy constructor be just that. It won't be as efficient by copying the `stack`, but efficiency is the last priority during exception stack unwinding which is notoriously slow. This is a clear example of premature optimisation.

The `print` method writes to the standard output stream, but this might not be what the client code requires. What if the information is wanted in a log file or the `cerr` stream? Modify the print method so that it returns the text representation of the exception stack as a `std::string` object. The client code may then do what it likes with it.

I'd also question the assumption that calling `print` means that the exception has been dealt with such that it need not continue to re-throw. I'd suggest a simple `deactivate` method to indicate this, then the `print` method can become `const`. That way the `print` method does just that without any side effects.

```
// Print the exception
std::string print() const
{
  std::ostringstream str;
  std::copy(stack.begin(), stack.end(),
    std::ostream_iterator<const std::string&>
    (str, "\n"));
  return str.str();
}
// de-activate the exception so that it won't
// re-throw.
void deactivate()
{ rethrow = false;}
```

The `what` method assumes there is always a non-empty stack. A crash would be the rather ungraceful result if this method were called on a 'default constructed' object.

```
const char* what() const
{
  if (!stack.empty())
    return stack.front().c_str();
  else
    return "Empty exception stack";
}
```

If this exception type is to be used as designed, we could offer some help for the poor developers that would have to add `try`/`catch` blocks throughout their code to add the context information to the exception as it unwinds. I'd suggest some simple macros as follows:

```
// Make this the first line of a function
#define BEGIN_EXCEPTION(x) \
std::string exceptionContext_(x); try{
// Make this the last line of a function
#define END_EXCEPTION() \
}catch(exception& e){ e(exceptionContext_42);}
```

The above macros make that tedious task a little easier, and provide the bonus that they can be easily removed from production code if required. Some developers have it in for macros, but used in the appropriate context they are a powerful tool.

Finally, with the addition of the above helper macros, I believe that the self-throwing exception type is not worth the trouble and controversy. A simple modification to the `END_EXCEPTION` macro provides the same behaviour without the trouble...

```
// Make this the last line of a function
#define END_EXCEPTION() \
  }catch(AccumulativeException& e) \
  { e(exceptionContext_); throw;}
```

The complete exception module is given below with my suggested modifications. I realise this no longer fulfils the original brief, but has additional benefits including simpler code (no need for **rethrow** flag or copy semantics) and use of **std::exception** as base. Also the **throw** statement means that this **AccumulativeException** class can be used itself as a base for other exception types without being sliced as it is re-thrown – the original (complete concrete) object gets re-thrown.

```
#pragma once
#include <deque>
#include <sstream>
#include <exception>
class AccumulativeException : public
   std::exception
{
public:
  AccumulativeException()
  : stack(){}
  AccumulativeException(
    const std::string& cause)
  : stack(1, cause){}
  virtual ~AccumulativeException() throw ()
  {}
  // Add context to the exception
  void operator()(const std::string& context)
  {
    stack.push_back(context);
  }
    // Print the exception
  std::string print() const
  {
    std::ostringstream str;
    std::copy(stack.begin(), stack.end(),
      std::ostream_iterator
        <const std::string&>(str, "\n"));
    return str.str();
  }
  // override standard 'what'
  virtual const char* what() const throw()
  {
    if (!stack.empty())
      return stack.front().c_str();
    else
      return "Empty exception stack";
  }
protected:
  std::deque<std::string> stack;
};
// Make this the first line of a function
#define BEGIN_ACCUM_EXCEPTION(x) \
  std::string exceptionContext_(x); try{
// Make this the last line of a function
#define END_ACCUM_EXCEPTION() \
  }catch(AccumulativeException& e){
  e(exceptionContext_); throw;}
```

Commentary

The critique has two main, unrelated, issues. The first one is the uninitialised variable **rethrow** which causes the difference in behaviour between the two compilers the user tried.

With g++ the **-effc++** flag (mentioned by Huw) does highlight this issue:

```
exception.h:14:3: warning: 'exception::rethrow'
should be initialized in  the member initialization
list [-Weffc++]
```

(I couldn't find a way to automatically detect this with MSVC.)

In my experience uninitialised variables are still a major cause of problematic behaviour. There was an example on accu-general a few weeks ago where an uninitialised **bool** results in a variable that neither tests **true** nor **false**!

The second issue is whether the concept in the code is valid. The relevant section of the C++11 standard is the **std::terminate** function [15.5.1p1] which is called under various cases and in particular: 'when the destruction of an object during stack unwinding (15.2) terminates by throwing an exception'.

The exception object throws a copy of itself in the destructor. The question is whether or not this is 'during stack unwinding'.

I added an instrumentation call to **std::uncaught_exception** in the destructor and both g++ and MSVC return 0 (no uncaught exception) during the execution of the destructor of **exception**.

This is correct as the exception is considered caught when the handler for the exception becomes active. However, exception handling proceeds as if the **throw** creates a temporary object. The compiler is *allowed* to elide this copy but is not *required* to do so. However, the exception class is unsafe for copy as **both** the original and the copy will have **rethrow** set to **true**. This means **both** objects will throw in their destructor and so the second one to be destroyed *will* try to throw while an exception is active. This would result in a call to **std::terminate**. I wonder whether the Oracle Solaris compiler is doing this, which is why Paul found the program aborted with that compiler? The second call stack he shows does include a call to **std::terminate**.

Note that this behaviour on copying also means that catching **exception** objects by copy rather than by reference will prove fatal! It is possible that if the copy constructor were to set the **rethrow** member to **false** in the **rhs** object this would be safe as it would prevent the problematic double throw.

This does also point to another problem with the proposed exception class. If you catch an **exception** object and throw a *different* exception then during the stack unwinding caused by this throw the original exception will be destroyed – and throw. This double exception will invoke **std::terminate** and abort the program.

You want exception code to be absolutely robust, so relying on potentially troublesome code is, in my opinion, not worth the trouble. I was unable to prove to my complete satisfaction that the code was valid and, as Paul found, it is very hard to debug problems occurring during exception handling. It doesn't really matter anyway – the code breaks on at least one implementation so should be avoided, I think, even if it is valid according to the letter of the standard!

Additionally, and slightly controversially, C++11 introduces a rule that destructors, by default, may not throw. Hence this code would need slightly modifying to work at all on a conforming C++11 compiler!

The winner of CC 76

It is clear Paul and Huw both disliked the attempted design; and for good reason. Both of them explicitly answered the presenting problem – the lack of consistent behaviour with g++ – but proposed rather different solutions to the design issue. I think I prefer Huw's proposal, as it doesn't unconditionally stop the program, which the use of **abort** does. So I have awarded this issue's prize to Huw.

Code critique 78

(Submissions to scc@accu.org by Oct 1st)

I'm trying to get started with Python ... tell me why the programme below doesn't work. The program is supposed to find a nine digit number using all of the digits 1 to 9 which is divisible by 9 (not difficult!) but is such that

- removing the last digit gives an 8 digit number divisible by 8
- then removing the last digit gives a 7 digit number divisible by 7 … and so on down to
- then removing the last digit gives a 2 figure number divisible by 2

Standards Report

Mark Radford presents the latest news from the ongoing C++ Standards process.

A significant date - or rather, range of dates - in the UK standards calendar must be 15th - 20th Apr 2013. This is when the UK next plays host to the ISO C++ Standards Committee meeting. We now know that Bristol has been chosen as the location for the ACCU conference, and so it will also be the location of the ISO C++ meeting.

In the nearer future, the next meeting of the C++ Standards Committee takes place on the dates 15th – 19th October, this year, in Portland, Oregon, USA. I'm not going, but a delegation from the UK will be going.

Towards C++2017

A topic that has recently interested the BSI C++ Panel is that of scheduling pieces of work for processing. Normally this would be associated with scheduling work across multiple threads, but work could (transparently) be scheduled on a single thread, also. This is something I mentioned in my last column, and I would like to go into a little more detail this time.

It all started with a proposal from Google [1]. This is a paper that the Panel has spent quite a lot of time discussing. We want to see the paper make progress, but some members thought certain issues need addressing. Further, we thought it important that these issues are addressed sooner rather than later, to mitigate the risk of the paper running into problems later on in the process. To this end we put together a response and sent it to the authors. The response consisted of nine points in total, that individual panel members believed should be raised.

I don't have the space here to present the entire contents of our response. The points made in it contained some detail and therefore it was rather long. However, here are short versions of a couple of the points made:

1. The C++ in the Google proposal is reminiscent of C++1998/2003, and come close to pre-standard (ARM) C++. It ignores modern C++ features such as templates and exceptions. This is not saying that templates and exceptions should be used just because they are there, but the benefits of using them should not be ignored. Looking at it another way: modern C++ features are in the language because there is benefit to having them.
2. Using `std::function` as the unit of work requires copying. From the performance point of view, an approach that only requires move semantics would be better because this facilitates the preservation of cache locality. Cache locality is very important for performance.

One of the great strengths of Google's proposal is that it is based on a library that they use, so it has existing experience behind it. However, point 1 leads to another important consideration: if we want a library based on the current version of the language, we can't have one based on existing experience. After all, you can't have experience of a language that does not yet exist. This is another topic I may well return to in a future column.

I'm now interested to see if there is a revised version of Google's proposal among the papers submitted for the Portland ISO meeting.

Just before I close this column I would like, once again, to mention that the call for library proposals is still out there [2].

References

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3378.pdf

[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3370.html

Code Critique Competition (continued)

- then removing the last digit gives a 1 digit number divisible by 1 (not difficult!).

Why does it tell me that `i` is not iterable:

```
$ ./program.py
Traceback (most recent call last):
  File "./program.py", line 8, in <module>
    for i in x:
TypeError: 'float' object is not iterable
```

The program is in Listing 3. [Editor's note: to make this printable I've used one space, not the recommended four, for each indentation and split long lines with a continuation character (\)].

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 3

```
#!/usr/bin/python
x=[1,3,7,9]
y=[2,4,6,8]
e=5
for a in x:
 for c in x:
  for g in x:
   for i in x:
    if a<>c and a<>g and a<>i and c<>g and \
       c<>i and g<>i:
     for b in y:
      for d in y:
       for f in y:
        for h in y:
         if b<>d and b<>f and b<>h and d<>f \
            and d<>h and f<>h:
```

Listing 3 (cont'd)

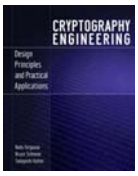
```
            z=10**8*a+10**7*b+10**6*c+ \
              10**5*d+10**4*e+ \
              10**3*f +100*g+10*h+i
            p,q,r,s,t,u,v,w= \
             z-z%10,z-z%100,z-z%1000, \
             z-z%10000,z-z%100000, \
             z-z%1000000,z-z%10000000, \
             z-z%100000000
            p,q,r,s,t,u,v,w= \
             p/10.,q/100.,r/1000.,s/10000., \
             t/100000.,u/1000000.,v/10000000., \
             w/100000000.
            p,q,r,s,t,u,v= \
             p%8,q%7,r%6,s%5,t%4,u%3,v%2,
            x=p+q+r+s+t+u+v
            if x==0:
             print z
```

Desert Island Books

Mark Ridgewell packs for the island.

It's difficult to choose which technical books I'd want to take. I have a library of books that I've collected over the years, many of them are becoming increasingly obsolete. I have no intention of taking Petzold's *Programming Windows 3.11* no matter how useful it was at the time. Just thinking about the segmented memory model and the complexities of near far and huge pointers just makes me feel ill. And that's before trying to wrap my head around the craziness of Hungarian notation.

My first real computer was an Acorn Electron. My brother taught me BBC BASIC and started off writing games in that. After a while we ran into issues of both speed and code size so started learning 6502 assembler and slowly switched to using that. It was here where my first book, *The Advanced User Guide for the Acorn Electron*, became incredibly useful. I remember spending ages looking at the memory maps finding where there was space to put code when in the graphics modes and, when attached, the floppy drive, where code could be put that would survive a reboot, so that could get around the copy protection on some tape games and re-save them out to the floppy.



Skipping forward to my first job programming in C++, here there are quite a few books and it is more difficult to choose which one to pick. Stroustrup's *C++ Programming Language*, Meyer's *Effective C++* and its sibling *More Effective C++* and Sutter's *Exceptional C++* and *More Exceptional C++* rank highly in terms of teaching me things. Of the three I probably gained the most out of *Exceptional C++*.

[continued overleaf]

What's it all about?

Desert Island Books is based (loosely) on the popular BBC Radio 4 programme, Desert Island Disks (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml). Many ACCU members have chosen their Desert Island Books, and there are plenty more to go. If you would like to share your Desert Island Books, please email cvu@accu.org

Choose 4 'technical' books – books that have influenced your programming life or that you would like to read – and explain what attracts you to them. Include a novel and two albums – you can slip in a film if you want – as this helps us get to know you better as a person.

ACCU Bristol & Bath Launched

Thomas Guest reports from the inaugural meeting.

Lightning talks. In a pub. Me first! I hadn't actually practised but I knew what I wanted to say and had picked a subject so trivial I couldn't possibly overrun.

Yes, it was time, at last, for the first ACCU Bristol & Bath[1] meeting, to be held in an upstairs room at the Cornubia. We'd reconnoitred the venue a few weeks earlier. Although the room was dingy and we couldn't work out where to put a screen, and despite disturbance from the increasingly raucous CAMRA meeting next door, the location was ideal and the beer superb. I looked forward to returning.

Plans change. In an agile last minute switch the meeting relocated to the Marriot – which, coincidentally, had just been announced as the host of next year's ACCU conference[2]. I shuffled through revolving doors into the hotel's vacant lobby rehearsing my talk in my head. Where was everyone? It took some backtracking and interrogation to locate the subterranean room but fortunately they hadn't started without me.

Now **this** was a proper meeting room. Panelled walls, no windows. A blank TV screen; green apples; red glasses; bottled water.

Ewan welcomed me. 'Have you got a macbook display adapter?'

No. I didn't even have the slides to my own presentation – I'd emailed them ahead to be merged into a single deck.

The screen flicked to life. Nine talks, five minutes each. We'd be done in an hour. After a brief welcome my slides were on screen and I was off.

Unfortunately I ran out of time, laughing too long at my own lightning anecdote which framed a talk about ellipses, the triple-dots ... which mean different things in different places in different programming languages. Next up was Dan Towner who walked us through the algorithm used by compilers for allocating registers. It's a greedy colouring of a planar map, he said, wrapped in a bail-and-retry loop. Dan Tallis spoke about the single committer model which works so well on open source projects. Developers don't have write access to the repository and must submit patches to the committer for review, a protocol which encourages incremental and considered changes to a codebase. Kevlin Henney needed just a single slide to clear up some misconceptions in exactly five minutes. Chris Simons didn't need any slides to describe where designs come from. Pacing the floor and waving his fingers, he explained that computer systems were punchlines; design was a matter of figuring out the joke. Attack the solution space with ants! No ACCU meeting would be complete without a discourse on C++ test frameworks and Malcolm Noyes duly dazzled us developing a C++ mocking library before our very eyes. Jim Thomson compared before and after binaries to prove his source code rearrangements hadn't done any damage. Ewan Milne, who'd not only organised and chaired the meeting, also contributed a talk on (guess what?) planning, subtitled how agile can Kanban be (say it!)

Jon Jagger postponed his closing talk. Macs just work if you've got the right connectors. We hadn't. The audience wanted more but that's no bad thing. We regathered in the hotel bar to crunch apples and chew over the evening. The ACCU Bristol & Bath launch had been a success! The price of a pint and anodyne surroundings discouraged lingering. We drank up and headed off towards trains, homes, and, for a select few, the Cornubia.

Notes and references

ACCU Bristol & Bath meets every couple of months.

[1] Follow ACCU Bristol & Bath on twitter: https://twitter.com/accuBristol or [@accuBristol]

[2] ACCU 2013 comes to Bristol! http://accu.org/index.php/conferences

[3] Subscribe to the mailing list: http://lists.accu.org/mailman/listinfo/accu-bristol-bath

Bookcase

The latest roundup of book reviews.

If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous 'not recommended' rating, you are entitled to another book completely free.

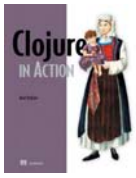
I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

Clojure in Action

By Amit Rathore, published by Manning, ISBN: 978-1-935182-59-7

Reviewed by Stephen Jackson



Clojure is a Lisp that runs on the JVM. Amit Rathore's 'Clojure in Action' is the third Clojure book I have read, following 'Programming Clojure' by Stuart Halloway and 'The Joy of Clojure' by Michael Fogus and Chris Houser.

In the section 'About This Book', the author states, 'To get the most out of the book, I've assumed you're familiar with an OO language like Java, C# or C++, but no background in Lisp or Clojure is required.' However, I did not find the core of the book to be particularly Java oriented - it's more a case of the reader being left to their own devices to sort out classpaths and what not. (If you want to work through all the examples in Part 2, you also need to install various additional packages.)

The book is divided into two parts; the first part (40%), 'Getting Started', is meant to teach the basics of Clojure, and the second part (60%) 'Getting Real' is about using Clojure to do real work.

The first couple of chapters of 'Getting Started' present a very good overview of Clojure. However, I think that the rest of this section, which goes into the details, rather glosses over a lot of those details. Perhaps the reader is expected to go and look things up in the Clojure API, but there are times when the newcomer to Clojure would not realise where the holes are. I think Halloway's book is much better for the newcomer than part 1 of this book.

The second section looked very promising, opening up with an excellent chapter on TDD. Subsequent chapters cover interfacing to data stores (MySQL, HBase and Redis), web services, RabbitMQ and DSLs. Although this section did provide some useful insights, I was left ultimately disappointed by part 2.

The book has received some criticism (from an erstwhile editor of *Overload* among others) because there are examples that do not work in

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Holborn Books Ltd** (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford (01865 792792)
  blackwells.extra@blackwell.co.uk

Desert Island Books (continued)

To this day the models of exception safety the books introduced me to serve well in other languages.

Skipping forward many years, and changing more to a mixed C++ and C# world Chen's *The Old New Thing* provided lots of valuable insight into why things in Windows work the way they do and how much goes on behind the scenes for compatibility purposes.

For current its a lot more difficult. There is no book that stands out. The *Effective C#* and *More Effective C#* books while good aren't as good as their C++ counterparts. *JavaScript The Good Parts*: I couldn't find the good parts. *Working with Legacy Code* was good, but largely overlaps various other books that I have.

I am tempted to pick something I haven't read, and so learn something new while on the island. This is more difficult as I'd ideally want something that will be useful. Perhaps a book on raft building, or planning. *Growing Object Orientated Software Guided By Tests* would be an interesting one that has been recommended by many and has been sat on my desk waiting to be read for months. I would go for something that would allow me to learn a new language, but without a computer that could get quite frustrating, so something that I can do without a computer, but with endless drawings in the sand. On this basis, I'll take Schneier *et al*'s *Cryptography Engineering: Design Principles and Practical Applications* so can work through the examples, and be in a better position to know if what I'm working on makes sense or is fundamentally flawed.

In terms of novels, this is the easiest choice of the lot – *Good Omens*, written by two of my favourite authors Terry Pratchett and Neil Gaiman. Novels written by either I enjoy, but this one written together has me laughing from start till finish.

Given I've got two albums I think I'll partially cheat and get an album that's long that I've loved for years – Pink Floyd's *The Wall*, which feels at least as relevant as it was when it first heard it. As I've also seen both the film version and live with the Roger Waters tour in 2011, I can remember/re-live the different experiences I have had of it.

The second album I've found difficult to choose and at one point was thinking of flipping coins to choose one as there's no one album where I really like everything on it. Two Pink Floyd albums would be too much to take. Counting Crows, Madness, Ke$ha, Nelly Furtado, Queen are all out as I have to be in the right mood to listen to them. Perhaps I should pick something so abhorrent so that it forces me to build a raft and get off the island as soon as possible rather than lying back and relaxing. Although a Justin Beiber CD would fit this it might be too much and I would try swimming without the raft. Wish I could have remembered to pick up my MP3 player stuffed full of music and this wouldn't be having this problem.

In the end I'm going to settle for Seven Mary Three's *Rock Crown*. Their other albums may have sold better, but of their albums this is the one I like the most.

View From the Chair

Alan Griffiths
chair@accu.org

I've always found I get a lot out of the ACCU – hearing about new (or renewed) ideas in software development, new languages and technologies (and a place to discuss ways to cope with the insanities imposed on us software developers by the businesses that employ us). But understanding or explaining the ACCU and how it operates has always been a challenge.

The core of the ACCU is a community of like-minded individuals with a common interest in exchanging ideas about our craft. When I first encountered the ACCU this was something unique, but other such communities have sprung up over the decades. Regardless, the ACCU still has something precious and I am proud to be a member.

One of the problems facing us at the moment is to clarify how the organisation operates. I've always thought it simple: the membership gives the committee authority to run the organisation and the committee has corresponding responsibilities to the membership. If anyone wants something to happen they can contact the committee who will usually give delegate to them the necessary authority in return for them taking the responsibility for taking the necessary action.

If this sounds somewhat haphazard and informal then that is because the ACCU is a somewhat haphazard and informal organisation. That is both a strength and a weakness – it means that anyone can spot a worthwhile activity and get involved on our behalf, it also means that sometimes no-one steps up to do something that would be good to do. Everyone involved is volunteering time, expertise and effort – so we have to accept that other priorities will sometimes take precedence.

Over the two committee meetings that I've chaired this year the committee has taken steps to make the operation of the organisation more transparent. As part of this minutes of the committee meetings are now published on the accu-members mailing list once they are approved (at the following committee meeting).

The committee has also formed two new working groups: Dirk Haun, with some other volunteers, has taken on the task of figuring out how to enhance the website and ensure that it remains current, dynamic and interesting. And Giovanni Asproni (the secretary) is co-ordinating work on updating the constitution to reflect the global nature of the ACCU and the availability of fast electronic communication. Neither group has yet had time to produce results, but they can act faster than the full committee – and are open to active participation by any interested members.

Another new point of contact on the committee is Matthew Jones – he has volunteered to co-ordinate the support we offer to local groups.

There has also been a change to the 'standards officer' – Lois Goldthwate has acted in this role since the 1990s but has now stepped down in favour of Mark Radford. In thanks for her long and valuable service the committee proposes that she receives an honorary life membership. Mark has previously been an ACCU committee member and has worked with Lois and others on the BSI C++ Panel.

I'm sure that all of the above (and also committee members of longer standing) would welcome support from any members willing to help out with the activities that make the ACCU what it is.

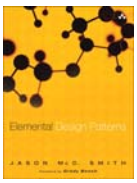
If you want to help and don't know who else to talk to then please contact me at chair@accu.org.

Bookcase (continued)

Clojure 1.3 which had been out for 2 months at the time of publication in November 2011. I don't think that is such a big deal, but the reader should be aware that because Clojure is young and rapidly evolving, any book about it will soon become out of date in some respects. (The 2nd edition of *Programming Clojure,* which was updated for 1.3, came out the same month as 1.4 was released.)

I don't think that *Clojure in Action* is a bad book by any means, and it contains a few chapters that are real gems, but I struggle to think of the sort of person I would recommend it to. *Programming Clojure* is a better bet if you want a gentler hand-holding introduction, and *The Joy of Clojure* is a must read for those who really want to 'get it'.

Elemental Design Patterns



By Jason McC. Smith, published by Addison-Wesley Professional, ISBN: 978-0321711922

Reviewed by Bob Corrick

It's patterns all the way down: a review of *Elemental Design Patterns*.

The author sums up his own research at the end of the first chapter: 'Elemental Design Patterns are the building blocks of computer science.' He is not suggesting that we build software pattern by pattern – rather that patterns, in their simplest form, are there to be discovered and can be used to assess and improve programs. The research produced a pattern discovery tool 'SPQR', and the twenty or so conceptual patterns are shown in diagrams and in code examples. The examples mostly use C++ or Java, and some use Objective C or C.

Overall the book is thoughtfully and enthusiastically written, and well produced. Sometimes the author's enthusiasm packed rather more into a paragraph than I could digest, but it hangs together on re-reading. I liked his use of 'reliance' to refer to the relationship between parts of patterns, which seemed a fresh and more accurate word to me than 'dependency', which would have suggested something rather too inflexible. It all seems accessible to a programmer who wants to know more about patterns, whereas the original Design Patterns book [GoF] seemed difficult to apply at the time.

Elemental design patterns are at the simplest level: calls from one place to another in a body of software mean that some class or module relies on another, forming a pattern. Some of the pattern names are familiar and well-researched (cohesion, coupling, recursion) and all are grouped together in some simple diagrams. These 'design space' diagrams help to distinguish between the various patterns, based on the similarities in location, type, and name of the related calls.

Six patterns from GoF are described in detail, broken down into elemental design patterns. To support this, Pattern Instance Notation (PIN) has been used – unfortunately, some of the PIN diagrams look like a jigsaw puzzle. I can't immediately pick out the lines that relate labels to components, and I couldn't help wondering what Edward Tufte would have made of these. The writing rescues this section, and it is particularly good on refactoring.

Elemental design patterns can be formed by relationships between objects, fields, and types as well as between methods. The analogy with the periodic table of elements is backed up by an appendix on the mathematics (rho-calculus), which I freely admit I haven't read. I've learned more about software in general and patterns in particular by reading this book, and I'll keep it.