overload 132
APRIL 2016 £3

Why Collaboration is Key for QA Teams in an Agile World

Agile processes demand significant changes of QA departments. We see how they can adapt to survive, and even thrive.

Knowledge-Sharing Architects as an Alternative to Coding Architects

Should architects write code?

Using Enum Classes as Bitfields

How to use C++'s new enumeration types to define bitfields

The Tao of Scratch

An environment designed to help young people learn to code

How to Diffuse Your Way Out of a Paper Bag

Applying diffusion models to paper bag escapology

QM Bites

Quality concerns using `while` and Microsoft platform definitions

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 132

April 2016

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 133 should be submitted by 1st May 2016 and those for Overload 134 by 1st July 2016.

Overload is a publication of the ACCU

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

4 The Tao of Scratch
Patrick Martin walks us through Scratch to help young people learn to code.

8 Knowledge-Sharing Architects As An Alternative to Coding Architects
Sergey Ignatchenko asks if architects should write code.

11 QM Bites: Understand Windows OS Identification Preprocessor Macros
Matthew Wilson outlines the differences between user-defined and predefined macros.

12 Why Collaboration is Key for QA Teams in an Agile World
Greg Law considers how QA departments need to change in an agile world.

13 How to Diffuse Your Way Out of a Paper Bag
Frances Buontempo diffuses her way out of a paper bag.

17 Stufftar
Ian Bruntlett shows us how he keeps files and folders in sync between various machines.

21 QM Bites: looping for-ever
Matthew Wilson advises on never-ending loops.

22 Using Enum Classes as Bitfields
Anthony Williams uses scoped enums as bitmasks.

24 9.7 Things Every Programmer Really, Really Should Know
Teedy Deigh introduces a 9.7 step plan for programmers.

The ACCU

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

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

Copyrights and Trade Marks

Where Does All The Time Go?

There's never enough time. Frances Buontempo wonders what this really means.

Time has flown by and I still haven't got around to writing an editorial. Given that time passes at a rate of 1 Hertz, or one second per second, claiming that time has flown faster than usual as an excuse is unconvincing. Can one second ever appear faster or slower than usual? Can an effect be observed before a cause? When I look back through text messages on my phone I often see a response to a question before I asked it, since its clock slips, making the timestamp on my message later than that of the reply. I can rearrange the conversion in my head, so it's not a problem. Does it matter when things happened? Or in the case of an editorial, if it happens at all?

Time can be a real issue in distributed computing. The machines can synchronise their clocks, so that at the same moment they each reset the current time to a chosen time. How do you define a moment anyway? We could become very philosophical very quickly but the requirement 'Synchronise watches at 06:00 hours' certainly begs a couple of questions: how can I tell it is 06:00 and where, since local time-zones vary? If you have managed to get all your machines to report the same time at the same moment, using radio signals, or a Network Time Protocol [NTP], being careful to avoid doing it too frequently thereby adding network load, you cannot ensure each clock will continue ticking at the same rate. Clocks are said to be 'syntonised' if they have the same frequency, even if they aren't running at 1 Hertz. Often crystals that vibrate at a known frequency are used for time keeping in a computer, however they are affected by various fluctuations in the environment, such as heat, and they age. You can buy a high quality more stable crystal on a PCI but that costs. The IEEE 1588 defines a precision time protocol [PTP] for synchronising multiple clocks over a network. While a group of people stood beside each other could get their watches to display 06:00 simultaneously if they turned up on time at the same place to synchronise their watches, the further the message about the current time has to travel the more significant the time taken for the message to arrive becomes. The PTP uses a 'sync' and 'follow-up' message to overcome this. This still does not deal with frequencies differing between machines, so will have to happen periodically. As noted, more expensive crystal oscillators are less prone to environmental changes, however the topology of a network of machines will still affect time signals. Jitter, or variation in latency, increases with more devices. Network traffic fluctuates. Using software, even following a well-defined protocol can make jitter worse. IEEE 1588 also can be used for hardware time stamping. Rather than stray well beyond my knowledge I will leave an interested reader to, well, read. See [Dopplinger and Innes] for example.

If we don't need sub-millisecond accuracy we do have one oscillator we can currently rely on. The earth rotates at a rate of one revolution per day. How much can you achieve in a day? If I have a day off I tend to make a to-do list, motivated by all the things I feel I need to do. I frequently fail to get more than half way through this. For a while I have started with 'Item 1. Make a to-do list'. This ensures I at least complete one task. I need to learn to be more realistic in my expectations. I am gradually learning to say "No" or at least "Not yet". It is ok to have a backlog and just concentrate on one thing at a time. I am noticing how easily distracted I become by emails or following the references in articles and academic papers I read as I try to find out about something. Minimising my web-browser, closing my email client and putting my phone on silent help me to concentrate. Going for a long walk armed only with pen and a notepad, and my door keys, can also help. Once I am 'in the zone' I find it easier to be more brutal with potential distractions and time wasting temptations. If a book is only vaguely relevant I can put it aside. The same might happen as I code. If I am concentrating, I can see I've gone off in a terrible direction and delete everything and start again much more readily than if it only has part of my attention. Then I am much more likely to slap in a few Boolean flags and experiment until it appears to work. I also tend to under-estimate how long something will take me. If I have a free hour this evening, I tend to think, "I have an ACCU conference talk to write. I have an hour. Therefore I will write a conference talk in an hour." This is clearly ridiculous. Based on previous experience, I will spend about 50 hours in total, coding up something I've never done before and doing a lot of background reading. Sometimes our estimates at work are formed, or even forced upon us, by sales people who have promised a client something will be ready next month, therefore we **must** ensure it is ready. This combination of backwards thinking and lack of experience inevitably leads to broken promises. The work we are asked to do is often something we have never done before and requires background reading. The sales people have often never written an application before. If they have and claim they could do it themselves in a week, it's good to avoid the temptation to tell them to get on with it themselves. In one of my first programming jobs, I saw this happen frequently. After many demands to work over the weekend I asked to go with the salesman and be involved in the up-front promise. We made a couple of moves towards a more iterative approach, delivering one small functional requirement that could form a basis of discussion about future work, rather than promising an entire system in three months and failing to get any one part of it working properly. For a small business this means the potential for more regular cash flows, which is a good thing. Again, though, I am straying from my main point.

Is time always important? Why does it always seem to slip by quicker when you need it the most? Does it matter what order things happen in? Lessons from relativity will tell us that the order in which things appear

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She works at Bloomberg, has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

to happen will depend on your standpoint. Sometimes a strictly linear order of events is not required. With my text messages, even without a sequence number, I can reassemble the conversation. With some messaging protocols a sequence number is provided so communications can be reassembled in order if required. This also draws attention to missing packets of data. We can launch things asynchronously and respond when they are done. This can speed things up – true multitasking, rather than constant context switching, is a beautiful thing. It can seem unnatural if you are used to thinking things through in a step-by-step known order. Messages can be time-stamped, though as discussed this will cause confusion if local machines each have their own idea of 'now'. Things get worse if they don't all use universal time coordinates, UTC, or GMT if you will. Imagine a log-file, with local-time timestamps and day-light saving kicking in midway through. All your log-files use UTC, right? When your customer says this happen at 6 a.m., do check where they were at the time, as well as what they were doing. One person's 6 a.m. is another's lunch-break. If you just have one application you probably just have one log file to search through on one machine. A multithreaded application should obviously have the manners to log the thread id as well as the time stamp. If you have a larger distributed system, with many moving parts including services, middleware, engines, databases and who knows what else, you may have many log files to look through if you want to find out what just happened and why. Chris Oldwood has rant^H^H^H^H written extensively on sensible logging. Through limiting my RIPs (reads in progress) I rediscovered his Overload article on distributed 'diagnostic contexts' [Causality] and forced myself to avoid re-reading all his blogs and articles. Without going into too many details, just thinking about what you need to see in a log file is important. Next time I write a new program I am tempted to start with the logging, and avoid a big mess that screams "ERROR" at me every few lines, only to be told, "Those errors are normal." In an ideal world, I would be able to hunt for a specific pattern and unique number in a log file, with a script, and extract the details I need to re-run the scenario to troubleshoot. Be nice to yourself and write better logging.

I am on the verge of running out of space, and time. Log files can leave a trace of what happened when, and are therefore akin to diary keeping. How many of us keep a diary? One of the reasons I am over-optimistic about how much I can achieve is because I do not have an accurate record of how long things usually take me. I have decided to start tracking my time spent on my to-do lists so I can come out with more sensible estimates next time. I might timetable slots for certain things – like reading emails – and only do this in batches to avoid interruptions. I know I waste ages reading things that turn out to be no help, arguing with people about things that don't really matter, debugging the wrong bits of code because log files lead me astray. Time is precious and easily wasted. Perhaps you can try turning down meeting invites for a month and see if you manage to be more productive. If you track how you spend a day, you might be surprised by the amount of time you spend doing admin tasks. Measuring what's actually going on often leads to surprising results. In order to learn new things, or tackle 'chewy' problems, you need long unbroken periods of concentration. We all recognise the clarity that comes with several hours of focus on one thing. Focus is not an 'F' word! If, like me, you feel time slipping away from you, take a leaf out of Richard Feynman's book, or rather lecture [Feynman]:

> To do real good physics work, you do need absolute solid lengths of time … it needs a lot of concentration … if you have a job administrating anything, you don't have the time. So I have invented another myth for myself: that I'm irresponsible. I'm actively irresponsible. I tell everyone I don't anything. If anyone asks me to be on a committee for admissions, 'no', I tell them: I'm irresponsible.

We can choose how we spend our time. It's ok, and in fact, important to spend time relaxing and having fun. You need to spend time with your family and friends. If you want to achieve great things, or even one specific thing, like writing an email or fixing a bug, you need to make space and time for it. Timetable in a weekend off, or a holiday and timetable in some space to keep learning and achieving great things. Being busy isn't the same as being productive.

> *It always takes longer than you expect, even when you take into account Hofstadter's Law.*
> ~ Douglas Hofstadter, Gödel, Escher, Bach: *An Eternal Golden Braid*

References

[Causality] 'Causality – Relating Distributed Diagnostic Contexts' Chris Oldwood, *Overload* 114, April 2013 http://accu.org/index.php/journals/1870

[Dopplinger and Innes] 'Using IEEE 1588 for synchronisation of network connected devices' 2007 http://www.embedded.com/design/connectivity/4007059/Using-IEEE-1588-for-synchronization-of-network-connected-devices

[Feynman] in various places, including in Study Hacks Blog: 'Feynman didn't win a Nobel by promptly responding to email' http://calnewport.com/blog/2014/04/20/richard-feynman-didnt-win-a-nobel-by-responding-promptly-to-e-mails/

[NTP] http://www.ntp.org/

[Neville-Neil] 'Time is an Illusion Lunchtime Doubly So' George V. Neville-Neil *Communications of the ACM*, Vol. 59 No. 1, pages 50–55 http://cacm.acm.org/magazines/2016/1/195723-time-is-an-illusion-lunchtime-doubly-so/fulltext

[PTP] 'Introduction to Distributed Clock Synchronization and the IEEE 1588 Precision Time Protocol' http://www.ni.com/tutorial/2822/en/

The Tao of Scratch

Scratch is an environment designed to help young people learn to code. Patrick Martin walks us through it.

When I look at Scratch, I see something 'different'.

I'd like persuade you to my viewpoint by going through what I see are the Good Points.

I'll be up-front here and state my target audience is the 'noble corporate toiler'. It has hopefully been a while since you were introduced / subjected to a computer based teaching tool. Scratch is a tool for implementing computing projects based upon many years of thought, and it is well worth taking a look.

Just what is Scratch?

The best place to start is the Scratch web site, where there is a description of the project [Scratch-1] and the main portal [Scratch-2]. The Wikipedia page is also good [Wikipedia-1].

It's a visual programming language, consisting of composable code blocks that form the elements of a program. It is available as a web application online, and implements a visual stage as the program output.

There is also an offline version now, which largely matches the basic capabilities of the web version, omitting online-only features.

A whistle stop tour

Scratch operates on a visual stage of 480 x 360 virtual pixels, and can be viewed at varying dpi – *Good Point 1: fixed stage extent*. The stage is a sprite [Wikipedia-2] and can be given script code, and any number of sprites can be placed on the stage to compose a visual scene. The sprites have a scripts container, which owns aggregations of code blocks that can interact with stage changes, key and mouse events and manipulate the sprites' properties. The sprites have a concept of their direction and can be made to orient in any direction, move and 'bounce' within the stage area automatically. Finally, the large array of primitive code blocks available for the scripts can react to events and messages and with the sprites, and allow the programmer to construct a range of different behaviours and visual experiences.

Figure 1 is a grab of a minimal project put together from an empty project and the built in resources.

Visualisation

The entire (basic) environment is visual, with editors for all aspects of the sprites and code blocks. The stage is manipulated at both 'run time' and 'design time' in the same way, which is *Good Point 2: persistence of stage state*. This allows fine positioning by eye, avoiding too many demands on mental arithmetic for the initial learning stages.

Patrick Martin Patrick's github repo was classified using a machine learning gadget as belonging to a 'noble corporate toiler'. He can't top that. Patrick can be contacted at patrickmmartin@gmail.com.

Resources

Image and audio resources are embedded into to the project and are available in a standard library – *Good Point 3: built in resources* – which can be expanded by assets uploaded to the project. All of these can be copied, and modified once embedded.

Running the project

There is a Green button to start running, and a Red button to stop running. What running means can be deceptively simple – the simplest event is 'when flag clicked', which can have some code blocks attached underneath – this could perform initialisation or start a processing loop of some kind. You can have as many as you like.

Debugging

Debugging is an interesting proposition for the target audience.

- Adults (example) @garybernhardt: "I'm in a super good mood. Everything is still broken, but now it's funny instead of making me mad."
- Years 3–8: Now, I've seen their projects, and for some let me assure you: 'Everything is broken', but they're not fazed.

There is a learning curve on the road to learning to debug your project, and there are some useful built-ins, like message and comment notes. The coolest of which, though, is: *Good Point 4: code blocks can be modified at runtime in the designer*. All versions allow blocks to be dragged in, then dragged out while the project is running.

Can *your* tool do that?

Figure 2 is a grab of the code blocks being inspected *while the code is running* and Figure 3 is a grab of a chunk of code blocks after being pulled out of their container block *while the code is running*.

Did I mention you can do this *while the code is running*?

You can do this while just using the mouse – the selection extends naturally from your selected block to the end of the enclosing container block – making *Good Point 5: selection of blocks has some subtle but powerful affordances*. Making use of *Good Point 6: the positioning of code blocks in the scripts area can be arbitrary*, you can park a small chunk of logic within the visual context of where it was running to reduce the cognitive load.

This works particularly well on an interactive whiteboard to show the effect of a chunk of code blocks.

Even more impressively: the Old Scratch 1.x desktop version, has a similar single step, which illuminates the currently running block! Clearly this was too mind-blowing and is no longer available in the current version.

Why should I be interested in Scratch?

A good question. Here are some topics to persuade you.

There is a learning curve on the road to learning to debug your project, and there are some useful built-ins, like message and comment notes

Figure 1

It's coming to a 'classroom near you'

Good Point 7: 'This Thing is Happening' Note also that Scratch at this moment is on the way to being taught in the UK in primary and secondary schools – that's years 3–8! (around 7 to 12 years old). [gov.uk]

The Scratch heritage

The Scratch environment is inspired by the work of Seymour Papert [Papert], epitomised in the book *Mindstorms* [Papert93]. Listing the whole corpus of the back story is out of scope, so this is going to be left for the interested reader.

Some examples

You can explore for yourself by visiting the website [Scratch-2], making use of *Good Point 8: it has a single entry search box*. Having found a project of interest, you can dive straight in and take a look using *Good Point 9: Scratch has a 'See Inside' button*. If you like what you find you can then simply fork it using *Good Point 10: Scratch has a 'Remix' button* That new project is now available for you to modify, debug and run in any way you see fit.

Now, although it ruins the nice 3-part list rhetorical flourish, let me just mention here *Good Point 11: Scratch lets you edit before login **and then** allows you to login and save if you wish*.

Question: how many *paid for* services that we use for generating wealth can say the same? For the classroom environment, when dealing with the early years where it is a stretch to require detailed forward planning, this is a Good Thing.

What can be done in Scratch?

I won't promote individual projects, and further I will assert that I don't need to. Instead let's rely upon *Good Point 8: it has a single entry search box* and I can list some jump off points:

- Yorkshire to English dictionary [Scratch-Dictionary]
- Innumerable RPG type things about cat clans [Scratch-Cats]

  no, me neither
- Space Invaders [Scratch-Invaders]

  aww, yeah...
- Pacman [Scratch-Pacman]

Who are the users?

Although it seems to be primarily educationalists and UK scholastic years 3–8 (at least), there are 10s of millions of projects and millions of users, leading to *Good Point 12: a cast of thousands*. [Scratch-Stats1] [Scratch-Stats2]

A list of additional Good Points

There's a raft of features: *Good Points 13–21:*

Logo-like Sprite Primitives*, Code Blocks*, Message Passing, Composable Mathematical Operations*, Sprite Cloning, Entry Prompts, Text Messages, Pen operations and Sound. The annotated items (*) are all shown illustrated in Figure 2 in one way. The items listed are in my personal order of appreciation. A feature worth mentioning is Message Passing, which allows broadcasts of user defined messages to all items on a stage.

Figure 2

Figure 3

Supported development styles

So, the possibilities are limitless, but there is a definite set of approaches that practitioners will work through.

- Basic: sprites, backdrops, costumes

  Coupled with setting some properties, the bounce/direction/touching colour capabilities allow treating the visual stage as a mini engine with some rudimentary support for using the stage as a dressed 'set'.

- Advanced: explicit positioning/drawing

  To move onto to more dynamic generation of content, it is possible to use explicit positioning, and the use of pens to craft arbitrary shapes.

Immediacy

This is the type of benefit mentioned in the kind of thing Brett Victor has talked about [Victor] – the good stuff, in my opinion, started around 10:00. A brief summary is that he demonstrates a 'Braid'-like game [Braid] in a live editor, which has the ability to adjust variables using sliders at run time and to record and overlay timelines of the program state. The gist of the open question raised was "What could you do , if you could only visualise the outcomes?"

I still think about the impression that talk had on me. It's not quite possible to reproduce that talk right now in Scratch, but the immediacy of the run–change–run loop is still very forceful.

Accessibility

Why haven't I mentioned *entirely free to use* yet? If you care about diversity and granting access to self-improvement to all then this is huge.

Sharing

From *Good Point 10: Scratch has a 'Remix' button*

- It exists and works – if you like something and want to have a try at improving it, you just press the button
- Is this due to *Good Point 22: Scratch has No Merge Action* (for you corporate warriors)?

  This point maybe bears some examination – there are no libraries in basic Scratch and no code sharing. In order to add some code inspired by code from elsewhere, one has to understand what to splice in to a project, and actually do it. Because there are two ways to generate Scratch projects – remix and create new – there is at least some pseudo Darwininan process that improves the fitness of remixes, while the 'gene pool' gets a steady stream of new projects.

Deployment

From *Good Point 23: Scratch has a 'public' checkbox*, almost nothing could be simpler – you just click the 'public' checkbox.

Politeness

From *Good Point 24: Scratch seems to be incredibly good natured*, is it because they're all under 12? (or over 30?) Well, there are The Rules [Scratch-3] and there are the banned topics [Scratch-4]. There even seems to be a Scratch version of Herobrine [Scratch-Herobrine].

Don't worry: it's not like 'coding'

It is important to point out that 'just coding' is not the point – it's engagement with the environment, using creative approaches and problem solving that are the real end games here. [code.org]

Suitability for classroom and distributed teaching

Good Point 25: Scratch supports multiple sessions on the same account and concurrent work on different projects This is one setup I can advocate: given each project has a thumbnail in the account's list of projects: *Good Point 26: all projects have a thumbnail* makes it very easy for a supervisor to see what individuals are up to. This also allows the supervisor to investigate and debug a project remotely if needs be.

Teachable moments 1: big chunks of blocks vs message passing

When did you, dear reader, love to learn message passing as a style? In the Scratch environment, it seems to be a few months from a standing start. Very quickly, the users seem to grasp that giant chunks of nested blocks and variables can be replaced by sending the appropriate message to be handled by a smaller chunk of code blocks.

Teachable moments 2: Space Invaders vs Pacman

This seems to be one of challenges that sorts the population. Caveat: this is from my purely personal sampling. I have found the Space Invaders clones tend to be very high quality, whereas for the 'deceptively simple' Pacman there tends to be a raft of issues that challenge the users *Good Point 27: some tasks lend themselves to 'teachable moments'*. My suspicion is that there are a couple of required complex concepts in a maze game that Scratch neither delivers in its default toolbox, nor allows to be easily synthesised. This might be a fruitful area to extend the sprite's capabilities.

Confession time: for the second time (approx 30+ years after the first attempt) I am re-implementing Pacman and I'm finding that talking about the incomplete (broken?) versions I created on the way can be used for some 'teachable moments', for example:

- various 'baby step' projects that show the small adjustments to the code blocks that will implement the stages of a ghost
- moving smoothly between points on a grid
- choosing between N, E, W, S to head towards Pacman/some other target
- turning left instead of reversing direction on the next choice (which delivers the distinctive Pacman 'ghost patrolling in circles' behaviour for free)

User straw poll

It hit me that I should get an assessment from the horse's mouth, as it were. I therefore ran a very unscientific poll of a miniscule sample set of users, and asked them one question.

The results for answers to "What's the best thing about Scratch?" are:

Thing	Votes
You can draw costumes	1
You can use your imagination	2
You can make games	4
It's simple	1
It's easy to share	1
It's satisfying to finish things	1

So, making games wins it, but the ability to express yourself is clearly highly valued. It's very encouraging that user agency afforded by working in Scratch is prized over passive consumption.

Can it really all be Lovely In The Garden?

Sounds too good to be true, right?

- Well, it *is* a social environment with quite young users interacting over the internet running arbitrary code. Security concerns will always be with us, but there as mentioned are rules and admins which (touch wood) seem to work.
- Is it so addictive as to prevent people progressing to (say) HTML and python? This is a concern I have heard expressed: Scratch can seem to easily become the tool of choice to prototype an idea, which is dual edged: it's a good thing to get an idea captured quickly, but conversely it would be good to stretch oneself and branch out.
- There is a very serious number of games, some of which are quite impressive and diverting...

Conclusion and lessons for 'grown ups'

So here we come to the punchline: I have joked that some companies could figure out how to write and deploy their business apps using Scratch. That's not in fact a serious proposition, but I suggest we can see that the secret of the success is all the Good Things, working to make the environment as expressive as possible.

Many operations take one step

A case in point is the debugging example already demonstrated; in fact the users slip naturally into re-initialising or exploring different initial conditions for the stage in a rough manner *by dragging the sprites back into position while the project is running.* It's only upon writing the sentence that it has struck me how extraordinary that is. There is an analogue with, for example, the mavens of languages supporting a command line REPL for 'real work', and also raw web development. Yet for many commercial systems, many steps exist between us and the prize.

Most things are quick

This multiplies the time savings with the prior point, but more importantly is key to a learning environment that encourages and rewards exploration. This is why coding club 'classes' should ideally not need a squad of instructors to oversee and guide work as the users can very well formulate hypotheses and test them themselves in short order.

The 'Ah, of course' moments

Hopefully I have shied away from employing too much hyperbole (!) in going over Scratch, and have instead relied upon the Good Points to make my case. In re-reading I see a pattern emerging of key concepts that have been selected and made available in a frictionless manner. This is part of the power of the Scratch approach, as the cognitive load is minimized, driving out distractions from the flow of working with the concepts of the project. For me this is the key takeaway: a real commitment to reducing the superfluous complexity of our tool sets leads to better outcomes. ■

Acknowledgments

Bloomberg LP (http://www.bloomberg.net) very generously supports my volunteering of time to mentor at an after school club, under the auspices of Code Club [CodeClub].

References

[Braid] https://en.wikipedia.org/wiki/Braid_(video_game)

[CodeClub] http://codeclub.org.uk

[code.org] http://worrydream.com/MeanwhileAtCodeOrg/

[gov.uk] https://www.gov.uk/government/news/harmful-ict-curriculum-set-to-be-dropped-to-make-way-for-rigorous-computer-science

[Papert] http://www.papert.org/

[Papert93] Papert, Seymour (1993) *Mindstorms* http://www.goodreads.com/book/show/703532.Mindstorms

[Scratch-1] https://llk.media.mit.edu/projects/783/

[Scratch-2] http://scatch.mit.edu

[Scratch-3] http://wiki.scratch.mit.edu/wiki/Scratch_Rules

[Scratch-4] http://wiki.scratch.mit.edu/wiki/List_of_Controversial_Topics_on_Scratch

[Scratch-Cats] https://scratch.mit.edu/search/google_results/?q=cat+clans

[Scratch-Dictionary] https://scratch.mit.edu/search/google_results/?q=Yorkshire+Dictionary

[Scratch-Herobrine] http://minecraft.gamepedia.com/Herobrine

[Scratch-Invaders] https://scratch.mit.edu/search/google_results/?q=space+invaders

[Scratch-Pacman] https://scratch.mit.edu/search/google_results/?q=pacman

[Scratch-Stats1] http://wiki.scratch.mit.edu/wiki/Scratch_Statistics

[Scratch-Stats2] https://scratch.mit.edu/statistics

[Victor] http://worrydream.com/#!/InventingOnPrinciple

[Wikipedia-1] https://en.wikipedia.org/wiki/Scratch_(programming_language)

[Wikipedia-2] https://en.wikipedia.org/wiki/Sprite_(computer_graphics)

Knowledge-Sharing Architects As An Alternative to Coding Architects

Should architects write code? Sergy Ignatchenko explores this controversial subject.

In recent years, quite a few articles have appeared with pro and contra arguments (mostly pro ones) on the question of whether software architects should code. Just to give a few examples, [Bryson15], [ArchitectsDontCode], and [Mirakhorli16] all argue for architects coding. There are a few articles out there such as [Langsworth12] which are trying to discuss both sides of this choice, but they seem to be overwhelmed by what was probably started by [Coplien05] and is rapidly becoming a 'common wisdom' of 'Architects Should Code'. Moreover, the 'Architects Should Code' point of view is supported by lots of developers out there, often by those who suffered from the hands of idiotic architects (who are unfortunately as abundant as not-so-good developers).

We will take a closer look at the arguments presented, double-check them against anecdotal evidence available (a.k.a. what I've seen and heard myself), and see where such analysis leads us. Let's start with considering the arguments which are commonly presented in this debate.

Pro architect-coding arguments

Avoiding being ignorant of implementation details. In particular, [Coplien05] has said that:

> ...many software architects limit their thinking and direction to abstractions, and abstraction is a disciplined form of ignorance. Too many projects fail on 'details' of performance, subtleties of APIs, and interworking of components – or, at best, they discover such problems late.

I agree that this is a valid point. However, I don't agree that it can be only avoided by the architect coding (ways to deal with it without specifically coding will be discussed below).

Responsibility. The most common argument in favour of architects coding is that architects should be responsible for the delivery of the project. I am not arguing with the 'should be responsible' part, but I don't see that being responsible necessarily means coding. Yes, a good architect must work closely with the delivery team [Bryson15]. No, working closely with the delivery team doesn't necessarily mean coding (which is consistent with [Langsworth12]).

Feedback. The second common line of pro-architect-coding arguments is that development is an inherently iterative process, so architecture should evolve as the product is being developed. I am not arguing with this point either, but I agree with [Langsworth12] that not writing code doesn't necessarily mean lack of feedback.

Respect. For the project to be successful, the architect should be respected by the team. Once again, there is no argument against this point. Moreover, it does imply that architect should be able to write code.

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

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

Observation 1: "Developers have more difficulty [than architects] implementing architectural choices" [Mirakhorli16]. This observation is interesting because it has a solid science behind it (there was a scientific study, with statistical analysis etc.). And yes, my own experience does support this observation. Once again, there is nothing to argue about here.

Observation 2: "Non-architecture savvy developers introduce more defects into architecturally significant code snippets than architecture-savvy developers" [Mirakhorli16]. This one is also supported by solid science, and once again my own experience supports this observation as such. And no, I still don't see why it means that the architect should code.

As you see, I do not argue with any of the points commonly presented as arguments for architects coding. On the other hand, I do not see why they necessarily mean architects should be coding on a day-by-day basis. Of course, I realize that 'I don't see why this or that point means coding' is a very weak argument – that is, until I can articulate a development model which addresses all these issues without coding. Don't worry – it will follow below.

Contra architect-coding arguments

Now we should consider the other side of the story, and see why architects coding might not be that good an idea. Note that we're speaking about those architects who do work closely with code, but do not code themselves.

Not seeing the forest for the trees. A biggie. Working on one part of the whole large project reduces opportunities to see the Big Picture. This is especially risky if your project has the concept of 'code ownership', but even without it, an architect too busy with debugging a piece of code is less likely to 'swap out' of this task to see a substantial architectural problem in the adjacent project. Been there, seen that (and was guilty of it myself too).

Time is better spent on other parts of the same project. This one seems to be strongly underestimated, but it is just a fact of life. There are only 24 hours in a day, and it is much more efficient for a good architect to spend her time on the other code-related things, such as code reviews and sharing knowledge about the architecture of this specific project (as well as about programming practices in general) with team members. More on this below.

Context switches are expensive (and no, I'm not speaking about threads). By the very nature of the architect's job, on the one hand he needs to be 'readily available' as soon as somebody needs architectural advice. The usual pattern goes as follows: somebody wants to add a function (class, whatever) to an infrastructure-level API, effectively exposing certain previously hidden implementation details. At this point, it is the architect's job to say that it doesn't belong here, and should be done on top of existing API in such and such manner (which happens 80% of the time), or to agree that such a function is indeed necessary (remaining 20%). This pattern happens all the time for all the teams around the world (from a kernel team to a big government development, with everything else in between). Even for a team of 5, this process leads to interruptions

being quite frequent (and the larger the team, the more interruptions experienced). On the other hand, being involved in a significant development requires concentration for several days, and such interruptions cause 'context switches' from coding to architecture advice. These 'context switches' (just as for threads) tend to be extremely annoying and lead to suboptimal decisions both for coding and for architectural advice.

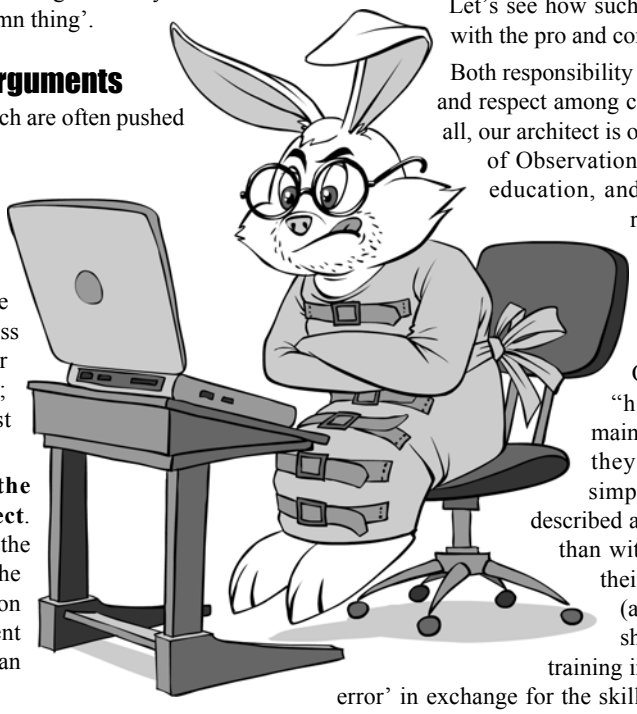
While DIY is easier in the short run, it doesn't scale, and knowledge-sharing does scale in the long run. Another biggie. If the architect is a hands-on one but she doesn't code, then to fix some architecture-related problem she'll need to explain the problem (and how to solve it) to somebody else. And as soon as you're explaining something, it means that the knowledge and, even more importantly, the 'feeling' about the architecture of your specific project is spread among developers. This is extremely beneficial in the long run (both according to Observation 2 above, and according to my own experience). Yes, for a good architect it is much easier and faster to code it herself. But after doing it herself, very little has changed in the minds of the other developers, so the next time she will need to code it herself again; and again; and again. It means that the development process doesn't really scale well. On the other side, if she spends more time now to explain things, the next time there will be more understanding of this kind of task (in terms of Observation 2, there will be more architecture-savvy developers in the long run), so more and more tasks can be safely delegated (and will be done according to the proper architecture for this project, too). I see this point as a Really Big Argument for knowledge sharing practices.

Dependency on the Architect. As noted above, non-coding (but working closely with code) architects are pretty much forced to share their knowledge to have things running. As a nice side effect, the very same knowledge sharing weakens a dependency on the architect (and reducing any dependency on a specific person is a Universally Good Thing™). While not-so-good architects might go nuts after realizing this point, good ones don't need to protect their jobs by being 'the only person who knows how to change this damn thing'.

Contra architect-coding non-arguments

There are also several non-arguments which are often pushed to support non-coding architects.

Architecture should not be concerned with implementation details. A Really Bad non-argument. In my book, an architect's job is to deliver a solution, plain and simple. Doing that means that the architect needs to be sure that more-or-less optimal implementations are possible for all the components of the architecture; moreover, the architect needs to suggest these implementations if the need arises.



Architecture is written once, then the architect can leave for another project. Note that this one is very different from the 'Time is better spent on other parts of the same project' point made above. Moving on to a different project, at least until current project enters the maintenance phase, is an almost-sure recipe for disaster.

Architect is a high-level role, is all about business lunches with customers, and developers even don't know how to play golf. I don't even want to discuss this one. If you're an architect and are thinking along these lines – you're horribly out of place.

Two things which work

Ok, we've stated all the arguments (and ignored non-arguments) from both sides. Now we need to architect a system which provides a reasonable balance of the answers.

Actually, there are at least two approaches which work. The first one is very simple – an architect who's coding (yes, current 'common wisdom'). It will work, at least as long as the architect is not too concerned about his own part of code and is not interrupted too often. However, knowledge sharing won't happen, pretty strong dependency on the architect will still exist (which is a good thing for the architect, but not necessarily for the project), and lack of time for reviewing code may lead to code quality taking a slippery road towards being unmanageable. All these issues are admittedly non-fatal, but I know that we can do better than that.

The second approach is the one I am arguing for, and I prefer to name it the 'knowledge-sharing architect'.

Knowledge-sharing architect

Here we'll be speaking about an architect who is one of the best coders around, but is not normally coding herself.

Note that for a knowledge-sharing architect, 'not normally coding' does not mean 'not involved with code', but the exact opposite: 'working with as much code as possible'. Such 'working with code' can and generally should involve most of the following:

- code reviews (formal ones and informal ongoing ones too)

  these often lead to showing 'how to code it better'
- pair programming (which was in particular suggested by [Coplien05], though I don't think that it is the only thing which can work in this regard)
- considering architecture-change (and often certain high-level API-change) requests from the team
  - these normally lead either to an explanation 'how to do it'…
  - …or to changes to APIs, code guidelines, etc.
- and discussions with team members on possible solution for arising problems (which lead to guidelines updates etc.).

Let's see how such a knowledge-sharing architect will deal with the pro and contra arguments listed above.

Both responsibility and feedback are right within the process, and respect among coders is not too difficult to achieve (after all, our architect is one of the best coders around). The impact of Observation 1 is not too bad (as there is a constant education, and enough time for code reviews). With regards to Observation 2, the long-terms effects of having a knowledge-sharing architect are much better than with the coding-architect approach due to, well, knowledge sharing.

One additional argument in this regard is: "how do knowledge-sharing architects maintain that level of brilliance in coding once they don't code anymore?" The answer is simple: as knowledge-sharing architects (as described above) are working with more code rather than with less code, this will help them to keep their coding skills top-notch. In other words (and for one specific example): knowledge-sharing architects are giving up up-to-date training in the skill of 'how to find that off-by-one error' in exchange for the skill of understanding the code of the others (and learning new tricks from them too). There is no magic here, and to improve one skill, you need to give up another one, but I am arguing that the skill of understanding what's going on is more important for an already good coder than day-to-day training in figuring out rather stupid bugs (and 90% of all the bugs are outright stupid, so dealing with them doesn't benefit coding skills much).

Now let's consider the arguments listed in the contra section. With a knowledge-sharing architect, 'Not seeing the forest for the trees' has much less chance to occur, and there is more time for working with code (so that much more code can be covered; this is also helped by not having

those expensive context switches). Even more importantly, due to knowledge sharing the development process becomes more scalable, and dependency on the architect is reduced.

Side-by-side comparison

Let's summarize the above findings in a table:

	Coding architect	Knowledge-sharing architect
Ignorance about implementation details	No	No
Responsibility	Yes	Yes
Feedback handling	Time permitting	Yes
Respect of team members	Yes	Yes
Observation 1: difficulty with implementing architectural choices	Better in the short run, but no knowledge sharing in the long run	Worse in the short run, but better in the long run
Observation 2: more defects from non-architecture-savvy developers	Better in the short run, but no knowledge sharing in the long run	Worse in the short run, but better in the long run
Not seeing the forest for the trees	Medium risk	Low risk
Time available to spend on architectural issues	Less time	More time
Expensive 'context switches'	More	Less
Knowledge sharing	Worse	Better
Dependency on the architect	Stronger	Weaker

As you can easily see from the table above, I am a big fan of the 'knowledge-sharing architect' approach. And it has worked for me in quite a few projects too. Yes, a coding architect might work (and is indeed orders of magnitude better than an architect who has no clue about the code), but a knowledge-sharing architect will generally work better.

Exceptions

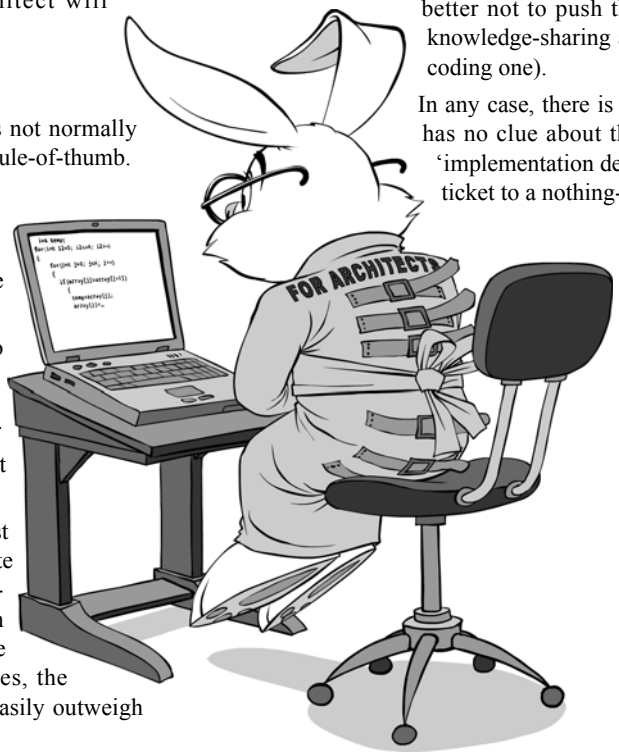
While the knowledge-sharing architect is not normally coding, there are a few exceptions to this rule-of-thumb.

The very beginning of the project

One Big Exception to the 'architect not coding' rule-of-thumb usually occurs at the very beginning of the project.

It is always a good idea for the architect to establish a framework which will be used for the project, and it is often a good idea for the architect to write a big chunk of such a framework (and the first implementation over it) himself.

At this point in the project, the team (at least the part which can meaningfully participate in development) is small, so knowledge-sharing is not that big issue; and as the team is small, lack of time is not a big issue either. It means that at these early stages, the advantages of the architect coding may easily outweigh the knowledge-sharing aspect.



However, it is a Really Good Idea to prepare for moving to the knowledge-sharing phase as soon as the framework that defines the architecture is written, and not to be 'the only person who knows about this piece of code' longer than is absolutely necessary.

Things which nobody else can do

The second exception occurs when (for the sake of your project, I really hope these occur really rarely) it happens that the architect is the only person who can implement a certain feature. (While in theory it shouldn't happen, in reality it does.) As we stated, the ideal architect should be one of the best coders around, and the other good coders might not have sufficient understanding of the Big Picture – or the time – to implement this specific feature.

In such rare cases you simply won't have an option other than for the architect to code this feature herself, and it won't be the end of the world. Note, though, that this is different from the 'architect coding only the difficult stuff' approach (which was criticized in [Bryson15], and I do agree with that criticism): here we're not speaking about cherry-picking the difficult (and interesting) stuff, but rather about doing it when there are simply no other options.

Conclusion

I hope that I have managed to convince you that a knowledge-sharing architect is better than a coding architect. The difference is subtle, but I've seen teams with knowledge-sharing architects scale better, and deliver higher quality code, than those teams with merely coding architects. While YMMV, and batteries not included, there are good reasons for these observations (which were outlined above).

However, there is one big practical problem with switching to the 'knowledge-sharing architect' development model. The problem is that most good architects won't be willing to give up coding (often 'cherry-picking' the most interesting pieces, but that's beyond the scope now), so the question of how to convince them to start knowledge-sharing isn't likely to be trivial.

On the other hand, as soon as it is understood that knowledge-sharing architects are beneficial to the project as a whole, the architect naturally faces a dilemma: either to continue to code (in the understanding that this is not the best way to serve the project), or to start knowledge-sharing. While certainly not an easy choice, this might lead to a switch for your architect from coding to knowledge-sharing (and if it doesn't, it is usually better not to push them too hard, as a persistently unhappy knowledge-sharing architect won't be any better than a happy coding one).

In any case, there is no argument that having an architect who has no clue about the code and doesn't bother himself with 'implementation details' is pretty much a guaranteed one-way ticket to a nothing-good-comes-out-of-it land. ■

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

References

[Coplien05] https://sites.google.com/a/gertrudandcope.com/info/Publications/Patterns/TopTenPatterns

[Bryson15] http://www.infoq.com/articles/architects-should-code-bryson

[Mirakhorli16] http://blog.ieeesoftware.org/2016/02/why-should-software-architects-write.html

[ArchitectsDontCode] http://c2.com/cgi/wiki?ArchitectsDontCode

[Langsworth12] http://randomactsofarchitecture.com/2012/11/20/should-software-architects-write-code/

[NoBugs15] 'No Bugs' Hare, 'Non-Superfluous People: Architects'. *Overload* #127.

QM Bites: Understand Windows OS Identification Preprocessor Macros

There's confusion between user-defined and predefined Windows 32/64-bit operating-system identification macros. Matthew Wilson shines light on the issue.

TL;DR:

Compiler defines **`_WIN32`** and **`_WIN64`**. You define **`WIN32`** or **`WIN64`**. Carefully discriminate.

Bite:

When compiling for Windows 32 and 64-bit architectures, there are four preprocessor object-like macro definitions for discriminating operating system that one may encounter:

- **`_WIN32`**
- **`_WIN64`**
- **`WIN32`**
- **`WIN64`**

You must take care that you understand the origins and meanings of these.

_WIN32 and WIN64

The symbol **`_WIN32`** is defined *by the compiler* to indicate that this is a (32bit) Windows compilation. Unfortunately, for historical reasons, it is also defined for 64-bit compilation.

The symbol **`_WIN64`** is defined *by the compiler* to indicate that this is a 64-bit Windows compilation.

Thus:

To identify unambiguously whether the compilation is 64-bit Windows, one tests only **`_WIN64`** as in:

```
#if defined(_WIN64)
/* Is Windows 64-bit */
#else
/* Is not Windows 64-bit */
#endif
```

To identify unambiguously whether the compilation is 32-bit Windows, one tests both **`_WIN32`** and **`_WIN64`** as in:

```
#if defined(_WIN32) && \
    !defined(_WIN64)
/* Is Windows 32-bit */
#else
/* Is not Windows 32-bit */
#endif
```

To identify unambiguously whether the compilation is a form of Windows one tests both **`_WIN32`** and **`_WIN64`** as in:

```
#if defined(_WIN64)
/* Is Windows 64-bit */
#elif defined(_WIN32)
/* Is Windows 32-bit */
#else
/* Not Windows */
#endif
```

WIN32 and WIN64

The symbol **`WIN32`** is defined by the user to indicate whatever the user chooses it to indicate. By convention, the definition of this symbol indicates a 32-bit Windows compilation, and nothing else! Microsoft (and other) tools generate projects with this symbol defined.

The symbol **`WIN64`** is defined by the user to indicate whatever the user chooses it to indicate. By convention, the definition of this symbol indicates a 64-bit Windows compilation, and nothing else!

When properly defined, these symbols can be used to indicate unambiguously the 32- and 64-bit Windows compilation contexts.

Caution with WIN32 / WIN64

Unfortunately, when duplicating a Win32 project to x64, the Microsoft Visual Studio wizards do not translate **`WIN32`** to **`WIN64`**. You must remember to do this yourself, in order for the inferences given above to hold. Do not add a separate **`WIN64`** to the x64 configuration settings: replace the existing **`WIN32`** with **`WIN64`**.

Why bother with WIN32 / WIN64 (and not simply rely on _WIN32 / _WIN64)?

There are doubtless many reasons. The reasons I adhere strictly to this are:

- it is a widely adopted and meaningful convention, so adheres to the *principle of least surprise* [PoLS].
- it facilitates the ability to emulate (parts of) other operating systems (e.g. UNIX [UNIXem]) while on Windows, which can be tremendously helpful when porting code. ■

References

[PoLS] *The Art of UNIX Programming*, Eric S. Raymond, AddisonWesley, 2003

[UNIXem] UNIXem is a simple, limited UNIXAPI emulation library for Windows. See http://synesis.com.au/software/unixem.html.

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.

Why Collaboration is Key for QA Teams in an Agile World

Agile processes can have an impact on QA departments. Greg Law considers how they can adapt to survive and even thrive.

The entire software industry is currently undergoing major, ongoing change. The rise of agile development, test-driven development, continuous integration and continuous deployment are all transforming how software is created and provided to customers. At the same time the spread of software into more and more of the devices around us and the interconnectivity of those devices means that the sector is growing in scale, complexity and importance.

These factors are having an enormous impact on testing and QA departments. If more software is being produced and needs to be deployed in shorter and shorter timeframes, traditional testing methodologies have to change, hence the rise of automation in the testing process (and corresponding worries among testers that their jobs will disappear).

Rather than fear change and automation, however, test departments need to embrace it. With QA at the beginning of the automation process, it is worth looking at the impact it has had on other industries. Automation tends to affect the number of people employed in an industry – for example, in 1900 41% of the US workforce was involved in agriculture, and in 2000 it was 1.9%. Agriculture is now many times more productive than it was when it relied on brawn, rather than brain. The same applies to the disruption caused by robots to factory jobs in the 20th century. In both cases, those who survived and thrived were the ones who embraced automation.

So how do test teams adapt to the new DevOps world? In my experience there are three areas to focus on.

Be part of the process

Agile is here to stay, and there is no point in denying that change is happening. So test departments need to understand agile, and look at how they can automate and work with developers to deliver the right services to meet overall business needs. In this world testers have to be able to become developers of a sort, scripting for automated test tools if they want to remain relevant.

At the same time, test and QA teams need to retain their independence from the development process, by challenging and testing development assumptions. This is one of the key strengths of having a separate function, and has to be preserved. Testers therefore need to work together with developers, but keep a certain distance from them, and testers should not be afraid to use their skills to ask potentially awkward questions.

Embrace new technology

Test automation brings a whole new set of opportunities and challenges. As I mentioned testers will need to be able to script tests to run on these tools, but more importantly they need to be able to understand, and communicate, the results.

The combination of agile, test automation and potentially unlimited compute resources via the cloud means more (and more detailed) tests are being run, more frequently, on more complex software. A software project of a given size can easily run two or three orders of magnitude more tests every day than the equivalent project would have run ten years ago. This means that there is a consequent growth in test failures, which threatens to overwhelm QA teams. If many thousands of tests run every hour, and 0.1% of them fail, triaging these failures can quickly become a nightmare. (And a failure rate as low as 0.1% is rare; I have spoken to companies where more than 10% of their overnight tests fail.)

Test teams therefore need to look at new technology that can help them not just to automate their tests, but they also need technology that will help them deal with the resulting failures. Tools such as Jenkins can help with more basic fails, allowing QA teams to focus their efforts on more complex or unpredictable issues. Software is becoming available to support test teams with these tougher cases as well, and can add greatly to productivity and speed.

Work more closely with developers

Software developers are QA's customers, so it is imperative that test teams provide a service that meets their needs. In pre-automated days, QA knew that simply handing over a list of red/green pass/fails was never going to help engineering find and fix the root cause of a problem. That's why they added verbose bug reports to give as much detail and context about a problem as they could. Obviously, this approach doesn't scale in the automated test world, which leads back to the point that using tools that can provide more details on what went wrong is highly valuable, even if it is as basic as "this was the commit that caused the code to fail". Talk to developers in their own language and give information in a usable form if you want to remain relevant and valued.

At the same time, testers need to be proactive and fight for their right to exist. For example, if you wait to be asked to attend development meetings there is a risk that the invitation will never arrive. So talk to developers, get involved early and use your skills to provide a higher level service that is valued by the whole company. If there are daily stand-up meetings, make sure you attend those.

I'm currently seeing a big change when I talk to customers and prospects. Three years ago I was speaking to development teams; now I'm increasingly talking to smart QA departments who understand that they need to be closer to developers, and want the tools to deliver this change. Obviously, the bad news for testers is that automation is likely to reduce their numbers, but arguably those that remain will move up the stack and be seen as more strategic and important to the entire agile development process. The choice may seem stark, but if they embrace change, test departments can survive – and thrive – in an agile world. ■

Greg Law Greg is the CEO and co-founder of Undo Software (http://undo-software.com). He is a coder at heart with over 15 years experience in the software industry, but likes to bridge the gap between the business and software worlds. Greg has a PhD in Computer Science from City University, London, and can be contacted at contactus@undo-software.com.

How to Diffuse Your Way Out of a Paper Bag

Diffusion models can be used in many areas. Frances Buontempo applies them to paper bag escapology.

Many applications in variety of areas from finance to epidemiology use Monte Carlo simulations of stochastic models, that is one using random variables. This article will revise the basics of Monte Carlo simulation and show how to simulate standard and geometric Brownian motion, ending with jump diffusion. The aim will simply be to move points out of a paper bag, since everyone should know how to program their way out of a paper bag. The simplest case, standard Brownian motion, simulates a cloud of particles spreading out, or diffusing, over time. This will allow us to introduce a mathematical model to simulate in code and form a basis for further simulations – geometric and jump diffusion. Both of these will show how a stock price might move over time. We will resort to cheating in order to make the price eventually end higher than the top of the bag. The reader can then decide if we have actually programmed our way out of a paper bag, or simply developed a simulation which allows discovery of appropriate parameters to meet a requirement. This, of course, is an essential use for much simulation software and will thereby demonstrate the value of coding up models to see what happens. The three diffusion models produce animations which move over time, so a paper article cannot do full justice to these. The code is available on github [github] for the reader to experiment with if desired.

Monte Carlo simulation

A Monte Carlo simulation allows one to answer, with varying degrees of accuracy, numerical problems that cannot be solved directly. For example, in order to find the area under a curve various approaches can be used. If the curve has a known equation which is integrable, the definite integral in the desired range can be calculated to give the area under part of the curve, or the indefinite integral will give the area under the whole curve. If the function describing the curve cannot be integrated, this approach will not work. If the curve were a hand drawn squiggle, we would not even have a function to attempt to integrate. In both cases, various estimation schemes may be used. Consider the curve in Figure 1. If a grid is superimposed, the area enclosed by the squiggle can be estimated by counting how many unit squares contain some of the interior of the curve. Finer grained grids will give more accurate estimates. Alternatively, a Monte Carlo simulation can be used, illustrated by the 'darts' in the last containing rectangle. If we randomly throw some darts, here 30, at the rectangle and find 10 are inside, we have 10/30 = 1/3 inside the curve, so around 33% of the area of the rectangle approximates the area inside the curve. Several such experiments will inevitably give varying areas, which may then be averaged or used to give a lower and upper bound as desired. The essence of any such simulation is the same; run an experiment a few times, and either see what happens, or try to answer a question. This introduction asked "What's the area of a curve?" Our subsequent simulations will discover if we can diffuse our way out of a paper bag.

Brownian motion

There are various diffusion equations as indicated in the introduction. Diffusion is a process whereby substances move, apparently randomly,

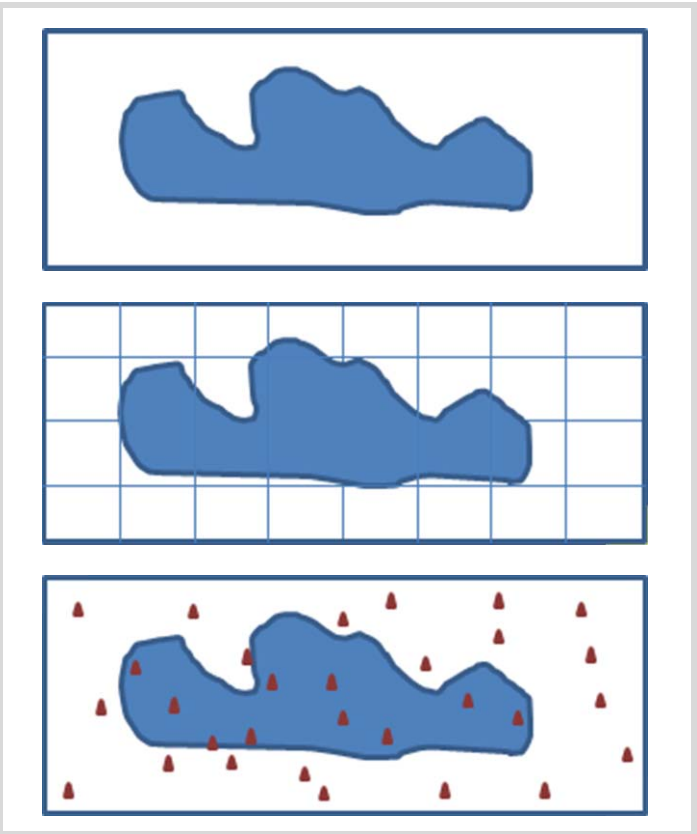


Figure 1

from places of higher concentration to those with lower concentration, eventually reaching equilibrium. This can be at the molecular level, in solids, liquids and gases, driven by pressure, temperature or electrical energy. Further details are available in various resources, for example see [Mostinsky]. For our simplest case, Brownian motion describes the motion of particles bouncing around in a fluid – either a liquid or a gas. Since we wish to model particles escaping from a paper bag, it is prudent to use the gas model.

Brownian motion is a special case of a random walk. It is a Markov process; that is, it has no memory. At any moment in time, the next position depends on where it is now, not how it got there. Other types of random walks are non-Markovian, for example taking their steps at random times. Brownian motion has two further properties: it has a mean

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

since each direction is equally likely the particles are quite free to sneak back inside the bag, even if briefly

change of zero and its change has a variance equal to the time step. In fact, the change, or step, is normally distributed. These properties make it a Wiener Process [Hull]. To get an intuitive sense of this, consider a simple random walk with a step size of one along a line, either going left or right. After starting at an origin, 0, the first step will take a particle either left, -1, or right, +1, with equal probability. So, for several experiments of one step, we will have an average of 0 steps. For experiments of more steps, we still get an average of zero, since the average of the total number of steps, is the total of the average step. If we now consider the step sized squared, we get 1. The distance squared will always be more than zero. In fact, the average of the sum of squared steps will be the number of steps. This gives us the variance property. The details can be found elsewhere, but [Schmidt] gives a simple explanation. If we then allow the step size to vary, in particular following a Gaussian distribution with mean 0 and variance 1, we nearly have Brownian motion. If we do this for two dimensions we are there. Consider a particular at some point (x, y). Given a source of two independent Gaussian random variables, mean 0, variance 1, called φ_1, φ_2 respectively, the particle will move to $(x + \sqrt{t}\varphi_1, y + \sqrt{t}\varphi_2)$. To implement this, we can make two independent draws from one random number generator. We could let $dW = \sqrt{dt}\varphi$ to simplify the equation.

Given a point, or particle starting at (x, y) we can then update it easily using a normal distribution in C++11 as seen in Listing 1. Details, such as the constructor implementation should be obvious, and how you would wish to report the current position is left to the reader. I used the Simple and Fast Media Library [SFML] to demonstrate the Brownian motion.

If we initialise several particles in the centre of a 'paper bag', indicated by three edges, and leave them to update according to this simple formula, they gradually diffuse. The greater the time step the faster they move. The next set of figures does indeed show a cloud of particles spreading out from the centre. The more astute reader may have realised that since each direction is equally likely the particles are quite free to sneak back inside the bag, even if briefly. It is possible to add paper bag edge detection to stop this happening, though this slows the dispersion down. The final figure shows some particles stopped just above the bag, since the code makes them stop when they have escaped from a paper bag.

```
class Particle {

public:
  Particle(double root_t,
           double x = 0,
           double y = 0,
           unsigned int seed = 1);
  void Update() {
    x += move_step();
    y += move_step();
  }

private:
  double root_t;
  double x;
  double y;
  std::mt19937 engine;
  std::normal_distribution<T> normal_dist;
  double move_step() {
    return root_t * normal_dist(engine);
  }

};
```

Listing 1

Geometric Brownian motion

We have seen how simple the Brownian motion simulation was. Since this allows particles to move freely in two dimensions, they will either go

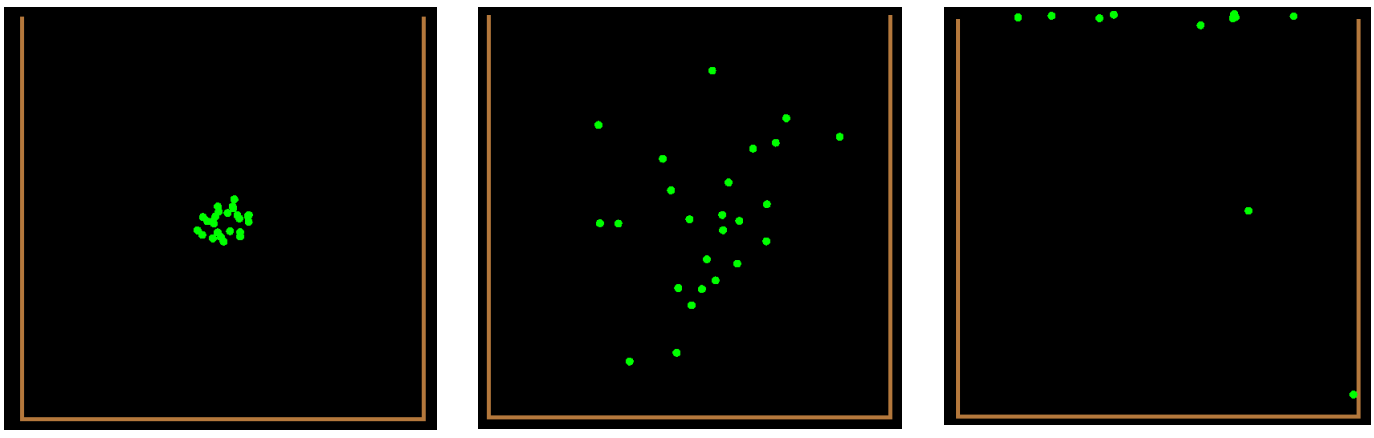


Figure 2

Given an initial stock price, and a drift and volatility parameter it is then easy to generate a sequence of possible stock prices after each time step

```
class PriceSimulation {
public:
  PriceSimulation(double price, double drift,
                  double vol, double dt,
                  unsigned int seed);
  double update() {
    double stochastic = normal_dist(engine);
    double increment =
      drift * dt + vol * sqrt(dt) * stochastic;
    price += price * increment;
    return price;
  }

private:
  std::mt19937 engine;
  std::normal_distribution<> normal_dist;
  double price;
  double drift;
  double vol;
  double dt;
};
```

Listing 2

through the sides of the paper bag, or if constrained not to, eventually go out of the top of the bag. If we now simulate a model of stock prices tending to go up, this will get us out of the bag more quickly. In order to do this we will use geometric Brownian motion. This is very similar to the first model. The logarithm of our new quantity follows Brownian motion but the process also has drift. In other words, taking the exponential of a Brownian motion with drift gives us geometric Brownian motion. This time, our quantity will be a fictitious stock price, S, giving us the y-coordinate, and we can just use the time, t, instead of an x-coordinate.

$$dS = \mu S dt + \sigma S dW$$

This stochastic differential equation can be discretised taking a previous stock price S and the step to the next stock price as $\Delta S = S(\mu\Delta t + \sigma\Delta W)$. Adding this step to the previous price gives us the next price. We still have the dW as before, but now we have a drift, μ, which is the rate of return on our investment, and a scale parameter, σ, which is usually described as volatility in a finance context. If this is zero, we have 'turned off' the stochastic, or random, part of the model and just see a simulation of the returns from a completely secure investment with a known return. Different stochastic differential equations (SDEs) will have different discretisation schemes, where we move from a stochastic differential equation to the move in a discrete step. Another common SDE is $S_{t+dt} = S_t e^{\mu dt + \sigma dW(t)}$.

Given an initial stock price, and a drift and volatility parameter it is then easy to generate a sequence of possible stock prices after each time step, dt. This is shown in Listing 2. If we treat the left most side of a paper bag, again indicated by three lines – a left side, the bottom and a right side, we can start the stock price somewhere above the bottom of the bag, which would be a value of 0, and see what happens. Clearly, if the stock is initially zero, each step ΔS will be zero since $0 \times (\mu\Delta t + \sigma\sqrt{\Delta t}\varphi) = 0$. Any initial stock value greater than zero will do. Similarly, if we take the left most side of the bag as time 0, and simulate the potential stock prices over a couple of years, each result can be plotted with a line drawn between each sample point. These then give scales for displaying the results. Figure 3 shows what happens with 10 simulations, using time steps of 0.1, with a drift of 0.5 and volatility of 0.2.

As expected the lines do resemble a stock price moving upwards over time. Unfortunately many of the final prices fail to end out of the bag. Some thought, or some time tweaking the parameters reveals that turning off the volatility and ramping up the drift to over 50% (recall this is a made up stock offering, sorry to disappoint!) will cause all of the simulated paths to escape the paper bag. This is shown in Figure 4. Since we have turned off the volatility, the stochastic variable will be zero, and each path of stock prices is deterministic and therefore follows the same path.

Jump diffusion

Resorting to turning off the volatility in the previous diffusion model is disappointing. With a higher drift rate and smaller volatility it is possible to end with most of the simulation paths escaping the paper bag. An alternative is to introduce jumps into the simulation. A key point of the Brownian motion models is their continuity. This is a precise mathematical concept, but it is sufficient to say a line or path is continuous if it can be drawn without taking your pen off the paper. A discontinuous path will have a jump – a point where the path breaks and pick up elsewhere. Jump diffusion can be used for a variety of application areas, but was introduced by Merton for financial pricing [Merton] and is often used in credit risk models. Without going into too many details, our initial stock model only allows each stock price change to be relatively small. A jump diffusion model will allow occasional large jumps, which might be more realistic. If we cheat and force these jumps to be positive, we are

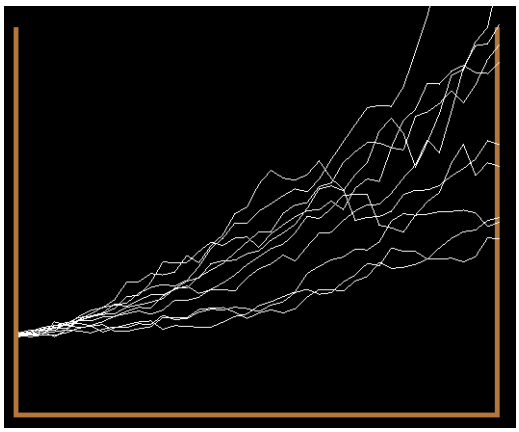


Figure 3

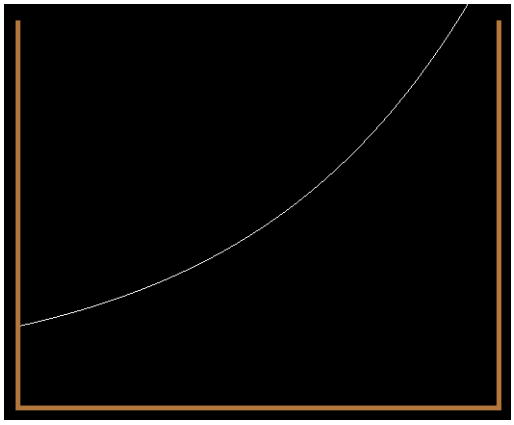


Figure 4

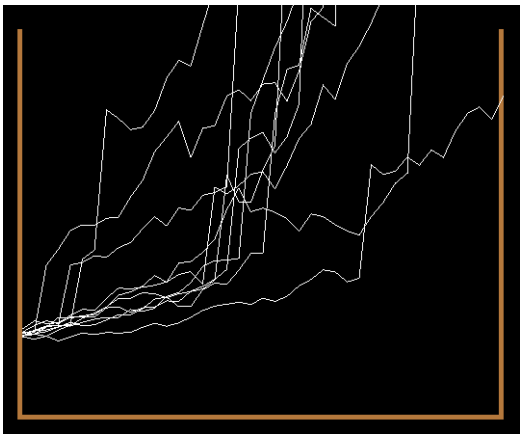


Figure 5

more likely to escape the paper bag. The obviously probability distribution to use to simulate something happening occasionally is the Poisson distribution. Calling this dN gives a stock price move in time dt of $\Delta S = S(\mu\Delta t + \sigma\Delta W + J\Delta N)$ where J is the jump size. When dN gives us 0, this collapses to our previous model. When it is non-zero we have introduced a discontinuity. It is not immediately obvious whether it is ok to use the same underlying generator for both dW and dN [StackOverflow] however, for this simple demonstration it suffices.

This requires a relatively small change to our `PriceSimulation` class. A new member variable is required :

```
std::poisson_distribution<> poisson_dist;
```

This is initialised with the probability of jumping, which can be passed through from the command line. The new update function is shown in Listing 3.

Figure 5 shows one outcome of our new jump diffusion model with 10 paths, time step 0.1, drift 0.5, volatility 0.2 and a jump size of 1 with a 50% chance of jumping. This time the majority of the stock price paths have managed to escape the paper bag. We could drop the drift and increase the probability of jumping in order to ensure the majority of paths escape. This indicates how useful coding up a model in order to experiment with the parameters can be.

```
double PriceSimulation::update() {
  double stochastic = normal_dist(engine);
  double dn = poisson_dist(engine);
  double increment  =
              drift * dt
              + vol * sqrt(dt) * stochastic
              + jump * dn;
  price += price * increment;
  return price;
}
```

Listing 3

Conclusion

This article has given an overview Brownian motion, geometric Brownian motion and jump diffusion. We have not considered how many simulations would be required to give realistic models. The accuracy of such a simulation improves with the square root of the number of runs, so a vast number are often required. Other approaches can mitigate this, for example antithetic variates can be used as a variance reduction technique – if dW gives us stochastic variable w we form a second path at the same time using -w. Another approach is the use of so-called 'low discrepancy numbers' rather than pseudo-random numbers. Both methods are beyond the scope of this short article. Further details can be found in the literature, for example see [Jäckel].

This article aimed to show how useful Monte-Carlo simulations can be, and gave a high level description of some diffusion models. The reader is encouraged to try to program their own way out of a paper bag and is free to use the github code, or snippets herein as a starting point. ■

Acknowledgements

My heartfelt thanks to Cassio Neri for taking time to read through this article and stop me from using an incorrect discretization for an SDE. I hope this has made the article both easy to read, and equally importantly, correct.

References

[github] https://github.com/doctorlove/paperbag

[Hull] p265 John C. Hull *Options, Futures and other Derivatives*. 6th Edition

[Jäckel] 2002 *Monte-Carlo methods in finance*. Wiley.

[Merton] 1976, 'Option pricing when underlying stock returns are discontinuous'. *Journal of Financial Economics* 3: 125-144

[Mostinsky] 'Diffusion' http://www.thermopedia.com/content/695/

[Schmidt] http://www.mit.edu/~kardar/teaching/projects/chemotaxis(AndreaSchmidt)/random.htm

[SFML] http://www.sfml-dev.org/

[StackOverflow] http://stackoverflow.com/questions/9870541/using-one-random-engine-for-multi-distributions-in-c11

Stufftar

How do you quickly transfer data from one machine to another? Ian Bruntlett shows us the bash script he uses.

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.
~ John Gall

Some time ago Frances Buontempo was looking for articles for *Overload*. I mentioned in an e-mail that I had a backup script called stufftar that I could write about. Frances kindly provided the questions that this article was built on.

What inspired it? Were you trying to solve a specific problem?

I use both Ubuntu and Lubuntu Linux. I am a volunteer tester of both. For personal use, I use Ubuntu and LUbuntu and I want to keep up to date so I needed some method to create backups and transport of key files and folders as flexibly as possible.

Initially I was backing up key files (all of `~/stuff`) to a .tar.gz file. Then I decided I wanted to automate it a bit so I worked out how to automatically create its destination filename:

```
DESTINATION_FILENAME=$1`date "+_%d_%B_%Y.tar.gz"`
```

Then I added commands to display the time taken to do the backups using the line:

```
/usr/bin/time -f "%E mins:secs " tar -czf
$DESTINATION_FILENAME $4 $5 $6 $7 $8 $9 ${10}
${11} ${12} ${13} ${14} ${15} ${16} ${17} ${18}
${19} ${20}
```

I had to call time from `/usr/bin` because bash was 'hiding' it with its own, less flexible, **`time`** implementation, which doesn't support the options I wanted.

Because of my inexperience with bash, testing was very important to me. As new features were developed, I would put a 'test' framework in place, thoroughly test each new feature and then remove the 'test' framework. Having functions ensure they had been given sufficient parameters was particularly important.

As time went by, I added more options. In particular I wanted to backup individual folders leading to the options (**`desktop`**, **`extra`**, **`rpg`**, **`home`** and **`localhost`**) being implemented. For convenience I wanted to be able to specify 'backup everything' so I implemented the **`all`** option. To keep things transparent, I implemented the **`verbose`** option for stufftar to output more information about what it is currently doing. I typically have a refurbished Lenovo ThinkPad T420 (from www.tier1online.com) as my main computer (hostname newton) running Ubuntu Linux. I have an old 32 bit Samsung NC10 that I take with me when visiting family (running lubuntu Linux). I back up my .tar.gz files to USB flash drive. I also back them up to external hard drives. I've got three external 500GB hard drives and, once a month, I copy that month's latest .tar.gz files to one of them. I rotate my use of external hard drive so that I'm backing up to the one containing the oldest backup of the set. About once a year I archive everything to dual-layer DVD-R – mainly as individual folders but I put the 'home' and 'localhost' .tar.gz files as is onto the DVD-R.

What does it do?

I've got a bunch of important digital files I carry around and backup to USB flash drives and hard drives. My `~/Desktop` files are my key files for general use. Other folders are `~/stuff` (the original folder I put my 'stuff' in), `~/extra` (the folder I moved from `~/stuff` when it became too large), `~/RPG` (the folder I use to keep RPG PDFs etc in). The **`home`** option backs up key shell scripts. The **`localhost`** option backs up key files (`/var/www/`) from LAMP studies.

How do you use it?

For help type in ./stufftar and it gives you a list of command line options (see Figure 1).

What future improvements do you envisage?

As a side-effect of writing this article, I've started a 'TO DO' comment section. Currently it does everything I need it to do. Also, despite having written a bash shell script helped by my copy of the Linux Pocket Guide (O'Reilly) I've never really studied bash. I've just muddled through. I have copies of *How Linux Works* and *The Linux Command Line* to make my way through sometime next year – I am concentrating on Ruby this year. The command line options are non-standard but OK for my purposes but could be modified to handle arguments like **`-f some_kind_of_parameter`**. There is a UNIX tool called TripWire, used to report changes to folders of files. I think I'll be looking at tackling that – in the future.

A walk through the code (edited highlights)

The line `#!/bin/bash` tells Linux that this is a bash shell script. Some people to specify **`sh`** instead of **`bash`** but as this script is for personal use, I'm using bash.

```
#!/bin/bash

# stufftar backup script by Ian Bruntlett,
# 2012 - December 2015, expanded and desktar
# merged in on August 11th 2012,
# March 2013 added "coder" file to Desktop tar,
# added BACKUP_HOME
```

As is usual, information about the script is stored at the start of the script (summarised for brevity).

Ian Bruntlett Ian refurbishes old Windows XP systems on behalf of people with mental health problems, installing Lubuntu and a selection of free software packages. As part of that effort, he is a volunteer tester for the Lubuntu project and has created an illustrated guide to Lubuntu, specifically targeted at people with mental health problems.

As new features were developed, I would put a 'test' framework in place, thoroughly test each new feature and then remove the 'test' framework

```
ian@newton:~$ ./stufftar
./stufftar Usage : stufftar followed by one or more commands: desktop,
extra, rpg, localhost, stuff, home and all
All data files are:-
1. Named after the relevant command name, followed by day number, month, year.
   For example: Desktop_01_March_2014.tar.gz
2. Are created using the tar command with file compression switched on.

Explanation of stufftar commands:-
desktop   - copy desktop files to a desktop tar file
extra     - copy extra to a tar file -  Linux Voice, Overload, QL Today
stuff     - copy stuff - anything I want to keep (main files are here)
rpg       - copy ~/RPG to a tar file
home      - copy refurb, stufftar, coder, removefiles.c to a home tar file
localhost - copy the whole /var/www/html subtree to a tar file
all       - execute stuff, extra, desktop and home commands in one go.
            Use when you want a full backup.
verbose   - display more details about the work being done.
status    - display status info about the stufftarred files on this system.

Also consider backing up Firefox bookmarks, and .emacs config file.
```

Figure 1

```
# echo_and_log(logfilename, text to put in log
# file and echo to screen)
function echo_and_log()
```

This is a 'helper' function that echoes its parameters both to the screen and to a specified log file. Useful to avoid repeated identical **echo** statements.

```
# if error code set ($1), display error messages
# and exit programme
function exit_if_failed()
```

This is another 'helper' function. It gets passed an error code by its caller. Normally it is 0 so this function does nothing. If it is non-zero then diagnostic information is echoed and it exits/aborts the script with a return code of 1.

```
# $1 log filename aka $LOG_FILE
# $2 file to get MD5 from aka $SOURCE_FILENAME
# example:- get_and_log_md5 "~/md5log.txt"
"localhost_04_May_2015.tar.gz"
function get_and_log_md5()
```

This function calculates the MD5 checksum of the file (function parameter **$2**) and logs it to a logfile (function parameter **$1**). It is a 'helper' function used by function **perform_backup** (Listing 1) when global variable **$STUFFTAR_VERBOSE** is greater than zero.

perform_backup is the 'engine' of the script. It validates its parameters. It does some logging, if running in verbose mode. It does the backup using both the **time** command and **tar**. The backup is created by **tar** and **time** outputs the amount of time taken. It also lists the number of files in the tar file by piping a list of file to the word count utility **wc**.

```
# perform_backup
# $1 - stub of .tar.gz filename
# $2 - name of log file
# e.g. "scripts/stufftarlog.txt"
# $3 - directory to do the tarring in
# $4 onwards - files/directories to put in .tar.gz
# file relative to $3
function perform_backup()
{
  if [ $# -lt 4  ]
  then
    echo "Error perform_backup() insufficient no
of parameters";
    return 1;
  fi;
 FILENAME_STUB=$1
 DESTINATION_FILENAME=$1`date
"+_%d_%B_%Y.tar.gz"`
 LOGFILE=$2
 TAR_DIR=$3

  if [ $STUFFTAR_VERBOSE -gt 0 ]
  then
    echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}
${12} ${13} ${14} ${15} ${16} ${17} ${18} ${19}
${20}
```

Listing 1

```
    echo
DESTINATION_FILENAME=$DESTINATION_FILENAME e.g.
Desktop_28_December_2014.tar.gz
    echo FILENAME_STUB=$FILENAME_STUB
    echo LOGFILE=$LOGFILE
    echo TAR_DIR=$TAR_DIR
  fi
  CURRENT_TIME=`date "+%H:%M:%S"`
  cd $TAR_DIR
  echo_and_log $LOGFILE $CURRENT_TIME Backing up
    key $TAR_DIR $4 $5 $6 $7 $8 $9 ${10} ${11}
{12} ${13} ${14} ${15} ${16} ${17} ${18} ${19}
${20}files to $DESTINATION_FILENAME
  /usr/bin/time -f "%E mins:secs " tar -czf
$DESTINATION_FILENAME $4 $5 $6 $7 $8 $9 ${10} {11}
${12} ${13} ${14} ${15} ${16} ${17} ${18} ${19}
${20}
  exit_if_failed $? "perform_backup to "
$DESTINATION_FILENAME
  echo File count:-
  tar -tvf $DESTINATION_FILENAME | wc -l
  ls -lh $DESTINATION_FILENAME
  if [ $STUFFTAR_VERBOSE -gt 0 ]
  then
   get_and_log_md5 $LOGFILE $DESTINATION_FILENAME
  fi
  echo
  cd;
  return 0;
}
# end function perform_backup
```

Listing 1 (cont'd)

```
  # show_last_line
  # $1 is the name to show
  # $2 is the log file to show the tail end of
  # $3 is the number of lines to show
```

This 'helper' function, **show_last_line**, echoes some information about a logfile – the name of the archive ('stuff', 'localhost' etc) and the last **$3** lines of the log file **$2**. See Listing 2.

The function in Listing 3 uses **show_last_line** to display the contents of all `stufftarlog.txt` files. The logfile performs two purposes. On the master computer, **show_status()** indicates when a particular folder was last backed up to tar file. On other computers, it shows the age of the data that has been transferred by tar file.

This function, **show_status**, is triggered when a parameter of **status** is passed on the command line. It can be used on its own or in conjunction with other commands.

```
function show_last_line()
{
  if [ $# -ne 3  ]
  then
   echo "Error show_last_line() insufficient no
of parameters ($#)";
    echo usage "show_last_line name of file,
source log file, no of lines to show"
    return 1;
  fi;
  echo -n "$1 - $2 "
  tail -$3 $2
  echo -n
  return 0;
}
# end function show_last_line
```

Listing 2

```
function show_status()
{
  echo STUFFTAR STATUS
  show_last_line "Desktop" "$HOME/Desktop/
stufftarlog.txt" 3
  echo
  show_last_line "extra" "extra/stufftarlog.txt" 3
  echo
  show_last_line "localhost" "/var/www/html/
stufftarlog.txt" 3
  echo
  show_last_line "RPG"   "RPG/stufftarlog.txt" 3
  echo
  show_last_line "scripts and home" "scripts/
stufftarlog.txt" 3
  echo
  show_last_line "stuff" "stuff/stufftarlog.txt" 3
  echo
  return 0;
}
# end function show_status
```

Listing 3

This is the main part of this script. If no parameters are passed, a help message is displayed explaining the use and parameters of the script.

The worker variables **BACKUP_DESKTOP**, **BACKUP_EXTRA**, **BACKUP_HOME**, **BACKUP_LOCALHOST**, **BACKUP_RPG**, **BACKUP_STUFF** are initialised to zero here. Another variable, **$STUFFTAR_VERBOSE**, is initialised near the start of the script.

Listing 4 is where I loop through the script's command line arguments, setting worker variables accordingly. Note if the parameter **all** is found, then a bunch of worker variables are set to 1.

For information purposes, the name of the script is echoed (**$0**) and if in verbose mode, **ls -lh** is used to show even more information about the script file. Also for information purposes, the status of worker variables is displayed.

```
  echo $# PARAMETER\(S\),
  BACKUP_STUFF=$BACKUP_STUFF
  BACKUP_EXTRA=$BACKUP_EXTRA,
  BACKUP_DESKTOP=$BACKUP_DESKTOP,
  BACKUP_HOME=$BACKUP_HOME,
  BACKUP_RPG=$BACKUP_RPG,
  BACKUP_LOCALHOST=$BACKUP_LOCALHOST,
  STUFFTAR_VERBOSE=$STUFFTAR_VERBOSE,
  STUFFTAR_STATUS=$STUFFTAR_STATUS
```

For consistency, the script changes the current directory to the current user's home directory before doing any file handling.

```
  cd ~
```

Then the .tar.gz files are created – basically checking to see if a worker variable is 1 and then calling **perform_backup** to do the work.

```
  # backup to a stuff tar
  if [ $BACKUP_STUFF -eq 1 ]
  then
    perform_backup "Stuff" "stuff/stufftarlog.txt"
    "$HOME" "stuff"
  fi
```

Similar clauses are used for the creation of the extra, rpg, desktop .tar.gz files. Backing up the key 'home' files is a little different:

```
  # backup key /home/ian things e.g this backup
  # script to a tar file
  if [ $BACKUP_HOME -eq 1 ]
  then
    perform_backup "Home" "scripts/stufftarlog.txt"
    "$HOME" refurb scripts synclamp stufftar coder
    removefiles.c;
  fi
```

And localhost is used to backup key LAMP files (see Listing 5).

As its final act, if the 'status' option has been activated, the status of every backup file is displayed.

```
if [ $STUFFTAR_STATUS -eq 1 ]
then
  show_status;
fi
```

And that is it. ■

Feedback from Overload technical reviewers

This script serves as a decent example of how to back things up in a reproducible way using tar. I didn't see any glaring errors in it, but I would comment that it is not tolerant of spaces in filenames (fixing that would require liberal use of double-quotes, and I usually find some trial-and-error is required to get this right).

> The script was written by me – a bash novice. Given the importance of the data, it was tested heavily as it evolved. As I expected to use the resulting .tar.gz files from the command line, I decided that my filenames would not contain spaces.

The article as it is now is a bit specific to one use case – I think it would be more useful if it explained the ideas and techniques being used rather than presenting the details of the script itself. E.g.:

- Interesting policy decisions like creating separate log files for each piece of work being done – it would be interesting to hear how this supports the workflow.

  I thought about having a single log file, `~/stufftarlog.txt` but it wasn't flexible enough and I'd have to somehow process that log file, looking for status information about each type of backup. By having separate log files, I avoid that problem and it means I can decide to just backup certain bunches of files instead of a full-blown backup of everything.

- How to get the last relevant line out of a log file (as the script does) – this seems more widely-applicable and a useful little nugget.

  I added the function body of **`show_last_line`** to this article. It is quite simple and uses the **`tail`** command.

- How to deal with command line arguments (the for loop used here looks quite convenient for simple applications like this).

  Yes. With a bit of effort it can be more flexible. At the moment it handles one word parameters that act as flags or specify a certain backup to perform.

  Supporting a syntax like **`-f some_kind_of_flag`** would be possible. I have some ideas about it, mainly involving extending the loop to set a flag (**`$ARGUMENT_F_EXPECTED`**) when a parameter of **`-f`** is detected and setting a flag. Then the head of the loop would need another set of **`if`** statements – followed by use of the **`continue`** loop modifier.

- The benefits of writing a script rather than doing this manually (e.g. reduced errors and less time take)

  Spot on. Being able to run a command to do all the backups I wanted, have standard filenames and contents, and walk away was crucial. That dealt with errors during creation of the backups. However, I used to transport my files to a Samsung NC10 NetBook (when it wasn't being wiped and used to test Lubuntu pre-releases), and I noticed that I was occasionally forgetting to install the contents of newer .tar.gz files. So I needed to know when a particular folder's files were created. This resulted in the function **`show_last_line`** (it can show more than one line) which was discussed earlier. When I'm working on the Samsung NC10, typing in **`./stufftar status`** means I can see how fresh this copy of my files is.

Reference

stufftar can be downloaded from:
https://sites.google.com/site/ianbruntlett/home/free-software/linux

```
for arg in $@
do
    if [ "$arg" = "verbose" ]
    then
      STUFFTAR_VERBOSE=1
    elif [ "$arg" = "status" ]
    then
     STUFFTAR_STATUS=1
    elif [ "$arg" = "stuff" ]
    then
      BACKUP_STUFF=1
   elif [ "$arg" = "extra" ]
    then
      BACKUP_EXTRA=1
    elif [ "$arg" = "rpg" ]
    then
      BACKUP_RPG=1
    elif [ "$arg" = "localhost" ]
    then
      BACKUP_LOCALHOST=1
    elif [ "$arg" = "desktop" ]
    then
      BACKUP_DESKTOP=1
    elif [ "$arg" = "home" ]
    then
      BACKUP_HOME=1
    elif [ $arg = "all" ]
    then
      BACKUP_DESKTOP=1
      BACKUP_EXTRA=1
      BACKUP_HOME=1
      BACKUP_LOCALHOST=1
      BACKUP_RPG=1
      BACKUP_STUFF=1
    else
      echo Warning! Unrecognised command : $arg
    fi
done
```

Listing 4

```
# localhost option not suitable to call
# perform_backup
if [ $BACKUP_LOCALHOST -eq 1 ]
then
# a try...   perform_backup "localhost"
# "/var/www/html/stufftarlog.txt" "/var/www/"
# "html";
  DESTINATION_LOCALHOST=
    `date "+localhost_%d_%B_%Y.tar.gz"`
  CURRENT_TIME=`date "+%H:%M:%S"`
  LOGFILE="/var/www/html/stufftarlog.txt"
  cd
  echo_and_log $LOGFILE $CURRENT_TIME Backing up
    localhost files to  $DESTINATION_LOCALHOST
  cd /var/www/html
  /usr/bin/time -f "%E mins:secs " tar -czf
    ~/$DESTINATION_LOCALHOST .
  exit_if_failed $? "tar localhost"
  cd
  ls -lh $DESTINATION_LOCALHOST
  echo
  cd ~
  echo localhost file count:-
  tar -tvf $DESTINATION_LOCALHOST | wc -l
  if [ $STUFFTAR_VERBOSE -gt 0 ]
  then
    get_and_log_md5 $LOGFILE
      $DESTINATION_LOCALHOST
  fi
fi
```

Listing 5

QM Bites: looping for-ever

Never-ending loop constructs can confound user and compiler in subtle ways. Matthew Wilson offers advice to maximise portability and transparency.

TL;DR:

do not use **while** to loop **for(;;)** ever

Bite:

Sometimes we want to use a loop that runs forever. This might be for a worker thread that simply waits for receipt of work, performs the work, and waits again, ad infinitum.

At other times the invariant for a loop may be too complex to be expressed transparently as a single expression in the loop-construct's loop invariant clause, instead being handled using one or more **break**s. (Note: such cases should always give one pause for thought, and to question whether the whole construct is too complex and should be broken down.)

Either way, we want a loop construct that loops forever, which means we want a loop invariant that is always true.

One common method to achieve this is as follows:

```
/* Commonly seen in C */
while(1)
{
. . . things that are repeated forever
}
```

Or the equivalent:

```
/* Commonly seen in C++ */
while(true)
{
. . . things that are repeated forever
}
```

There are two problems with this. First, the practical. Some compilers issue a warning about the use of a constant within the invariant of a loop statement. For example, the Visual C++ compiler will issue warning 4127 "conditional expression is constant". Since we always want to use maximum warnings wherever possible, and we always want to treat warnings as errors, this construct is something to avoid to ensure portability and maximum effective use of warnings.

Second, the philosophical. To be sure, this is a question of perception/taste (but I know others share my apprehension): it's just kind of weird and clumsy to say "loop for as long as **true** is true".

The answer to both concerns is the same: instead use a **for**-statement with a blank invariant, which is interpreted as "forever" (something for which it is designed):

```
for(;;)
{
. . . things that are repeated forever
}
```

Further, you're not required to have a fully empty **for**-statement to have infinite looping. The following will loop forever while providing you some indication as to how many loops have been done:

```
for(int i = 0;; ++i)
{
. . . things that are repeated forever
}
```

Just be aware that this will wrap around eventually (and repeatedly).

Afterthoughts

There are other aspects to the issue of constant expressions in loop expressions. Coming to a Bite-near-you sometime soon. ■

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.

Using Enum Classes as Bitfields

Scope enums have many advantages over standard enums. Anthony Williams shows how to use them as bitmasks.

C++11 introduced a new feature in the form of *scoped enumerations*, also referred to as *enum classes*, since they are introduced with the double keyword `enum class` (though `enum struct` is also permissible, to identical effect). To a large extent, these are like standard enumerated types: you can declare a list of enumerators, which you may assign explicit values to, or which you may let the compiler assign values to. You can then assign these values to variables of that type. However, they have additional properties which make them ideal for use as bitfields.

Key features of scoped enumerations

The key features provided by scoped enumerations are:

- The enumerators must always be prefixed with the type name when referred to outside the scope of the enumeration definition. e.g. for a scoped enumeration `colour` which has an enumerator `green`, this must be referred to as `colour::green` in the rest of the code. This avoids the problem of name clashes which can be common with plain enumerations.
- The underlying type of the enumeration can be specified, to allow forward declaration, and avoid surprising consequences of the compiler's choice. This is also allowed for plain `enum` in C++11. If no underlying type is specified for a scoped enumeration, the underlying type is fixed as `int`. The underlying type of a given enumeration can be found using the `std::underlying_type` template from the `<type_traits>` header.
- There is no implicit conversion to and from the underlying type, though such a conversion can be done explicitly with a cast.

This means that they are ideal for cases where there is a limited set of values, and there are several such cases in the C++ Standard itself: `std::errc`, `std::pointer_safety`, and `std::launch` for example. The lack of implicit conversions are particularly useful here, as it means that you cannot pass raw integers such as 3 to a function expecting a scoped enumeration: you have to pass a value of the enumeration, though this is of course true for unscoped enumerations as well. The lack of implicit conversions *to* integers does mean that you can overload a function taking a numeric type without having to worry about any potential ambiguity due to numeric conversion orderings.

Bitmask types

Whereas the implicit conversions of plain enumerations mean that expressions such as `red | green` and `red & green` are valid if `red` and `green` are enumerators, the downside is that `red * green` or `red / green` are equally valid, if nonsensical. With scoped enumerations, none of these expressions are valid unless the relevant operators are defined, which means you can explicitly define what you want to permit.

`std::launch` is a scoped enumeration that is also a bitmask type. This means that expressions such as:

```
std::launch::async | std::launch::deferred
```

and

```
std::launch::any & std::launch::async
```

are valid, but you cannot multiply or divide launch policies. The requirements on such a type are defined in section 17.5.2.1.3 [bitmask.types] of the C++ Standard, but they amount to providing definitions for the operators `|`, `&`, `^`, `~`, `|=`, `&=` and `^=` with the expected semantics.

The implementation of these operators is trivial, so it is easy to create your own bitmask types, but having to actually define the operators for each bitmask type is undesirable.

Bitmask operator templates

These operators can be templates, so you could define a template for each operator, e.g.

```
template<typename E>
E operator|(E lhs,E rhs){
  typedef typename
    std::underlying_type<E>::type underlying;
  return static_cast<E>(
    static_cast<underlying>(lhs)
  | static_cast<underlying>(rhs));
}
```

Then you could write `mask::x | mask::y` for some enumeration `mask` with enumerators `x` and `y`. The downside here is that it is too greedy: every type will match this template. Not only would you would be able to write:

```
std::errc::bad_message | std::errc::broken_pipe
```

which is clearly nonsensical, but you would also be able to write:

```
"some string" | "some other string"
```

though this would give a compile error on the use of `std::underlying_type`, since it is only defined for enumerations. There would also be potential clashes with other overloads of `operator|`, such as the one for `std::launch`.

What is needed is a constrained template, so only those types which you want to support the operator will match.

SFINAE to the rescue

SFINAE is a term coined by David Vandevoorde and Nicolai Josuttis in their book *C++ Templates: The Complete Guide*. It stands for 'Substitution Failure is Not an Error', and highlights a feature of expanding function templates during overload resolution: if substituting

Anthony Williams Anthony is the author of *C++ Concurrency in Action*. As well as working on multi-threading libraries, he develops custom software for clients, and does training and consultancy. Despite frequent forays into other languages, he keeps returning to C++. He is a keen practitioner of TDD, and likes solving tricky problems. Contact him at anthony@justsoftwaresolutions.co.uk

the template parameters into the function declaration fails to produce a valid declaration then the template is removed from the overload set without causing a compilation error.

This is a key feature used to constrain templates, both within the C++ Standard Library, and in many other libraries and application code. It is such a key feature that the C++ Standard Library even provides a library facility to assist with its use: `std::enable_if`.

We can therefore use it to constrain our template to just those scoped enumerations that we want to act as bitmasks (see Listing 1).

If `enable_bitmask_operators<E>::enable` is `false` (which it is unless specialized) then:

```
std::enable_if<enable_bitmask_operators<E>
   ::enable,E>::type
```

```
template<typename E>
struct enable_bitmask_operators{
    static constexpr bool enable=false;
};
template<typename E>

typename
  std::enable_if<enable_bitmask_operators<E>
     ::enable,E>::type
operator|(E lhs,E rhs){
  typedef typename std::underlying_type<E>
    ::type underlying;
  return static_cast<E>(
    static_cast<underlying>(lhs)
  | static_cast<underlying>(rhs));
}
```

Listing 1

will not exist, and so this `operator|` will be discarded without error. It will thus not compete with other overloads of `operator|`, and the compilation will fail if and only if there are no other matching overloads.

```
std::errc::bad_message | std::errc::broken_pipe
```

will thus fail to compile, whilst

```
std::launch::async | std::launch::deferred
```

will continue to work.

For those types that we do want to work as bitmasks, we can then just specialize `enable_bitmask_operators`:

```
enum class my_bitmask{
    first=1,second=2,third=4
};
template<>
struct enable_bitmask_operators<my_bitmask>{
    static constexpr bool enable=true;
};
```

Now,

```
std::enable_if
  <enable_bitmask_operators<E>::enable,E>::type
```

will exist when `E` is `my_bitmask`, so this operator| will be considered by overload resolution, and

```
my_bitmas::first | my_bitmask::second
```

will now compile.

Final code

The final code is available as a header file along with a simple example demonstrating its use (see https://www.justsoftwaresolutions.co.uk/cplusplus/using-enum-classes-as-bitfields.html). It has been tested with g++ 4.7, 4.8 and 4.9 in C++11 mode, and with MSVC 2012 and 2013, and is released under the Boost Software License. ■

9.7 Things Every Programmer Really, Really Should Know

Most of us have heard of the twelve step program. Teedy Deigh introduces a 9.7 step plan for programmers.

0. Start from zero

Zero is the magic number. For everything. (Except Coke.)

For example, atomic numbers start at zero, not one. Hydrogen is bohring. Neutronium is much more fun. Be sure to deploy this fascinating nugget at length during pub quizzes and family meals and, in case no real people find it as fascinating or redefining as you do, in online forums.

The day begins at 00:00. Have no truck with the twelve-hour clock. It is a nonsense that demands unnecessary modulo arithmetic and flags.

Count your indexes from zero. If your programming language doesn't support zero indexing, either change language or plough on regardless, ignoring any exceptions, undefined behaviour or program failures. You know you're right. That's all that matters.

1. The programmer is right

In any interaction, whether between programmer and manager or programmer and customer or programmer and computer, the default assumption should be that the programmer is right.

Unfortunately, not everyone seems to either know or appreciate this – the zeroth rule of *Being Right Club* is, after all, "You don't talk about *Being Right Club*" – which gives rises to all those long discussions and disagreements and disciplinary hearings.

The real problem occurs when two programmers get together. They are both right. Even when they hold different truths. This apparent contradiction is found in the heart of quantum mechanics, thus giving programmers a unique, foundational and paradoxical position in the pecking order of the universe.

It has been said that managing programmers is like herding cats. Almost. It is more like herding Schrödinger's cats.

2. Binary

01000110 01010100 01010111 00100001

3. Logic

Logic is a resource. It should, therefore, be applied sparingly. Of course, employ logic in code, but it doesn't necessarily have to be used elsewhere (e.g., in life or dealings with colleagues). As with other resources, you don't want to risk using it all up.

4. Information hiding

Fundamental to resilient and reasoned program structuring is the idea of information hiding. Keep secrets. Make sure your implementation is not obvious and your interface is inscrutably yet charmingly quirky. Such information hiding enhances security – especially job security.

It follows from the principle of information hiding that you shouldn't share things. Avoidance of sharing simplifies not only multithreaded programming, but also means enforcing a strictly personal model of code ownership, a desk configuration antagonistic to pairing and a ring-fenced (and castellated and moated) spot in the office fridge.

5. Wabi-sabi

Wasabi for your sushi, wabi-sabi for your code. Spice up your source with this Japanese aesthetic.

Instead of aiming for a purist view of perfection, wabi-sabi teaches us that a greater whole can be achieved through carefully placed imperfection, a recognition of impermanence and the essential completeness of incompleteness.

Embrace this philosophy in your code.

But why settle for subtle imperfections that enhance the whole? Move beyond the whole. Embrace imperfection more deeply and overtly. Clean code? Better with a splash of muddy logic. Abstract classes? Weigh them down with something concrete. A rich and healthy domain model? Introduce poverty with an underclass of malnourished, anaemic domain objects.

Flaunt impermanence. Comment your code with the fixes you've done or the fixes you think should be done. But don't do them. Use this also to highlight the incompleteness of existence. Suggest the eternity of nature that subsumes our fleeting lives. Don't finish things. Done done? Not even close.

6. Hexadecimals

Because *The Martian*.

7. Test your code

Do this by giving it to the customer. (If they ask nicely, compile it first.)

8. There are eight planets in the solar system

There is a sun, two groupings of planets, four terrestrial planets in the inner solar system and four gas giants in the outer solar system, giving a total of eight planets. This all conveniently aligns with powers of two, so therefore must be right. And there are patterns within: *pow(2, 3)* is *8* and Earth is the third planet. Coincidence? Of course not.

Don't fall for the Disney conspiracy.

9. The secret to writing great code

Much has been written on what it means to write great code and how to write it. Most of it is wrong. There is only one thing you need to know: always– ■

Teedy Deigh Teedy Deigh considers herself to be a source of software development wisdom, negotiating (and crossing) the fine line between the code face and being in your face. Over the years Teedy Deigh has made just over 9.7 contributions to *Overload*. Opinion is divided as to whether the value of her contributions is second to none or closer.