The magazine of the ACCU

www.accu.org

{cvu}

Volume 24 • Issue 2 • May 2012 • £3

{cvu}

Volume 24 Issue 2
May 2012
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Lisa Crispin, Pete Goodliffe, Paul Grenyer, David Mansergh, Roger Orr, Nick Sabalausky, Joanna Simoes

ACCU Chair
Hubert Matthews
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

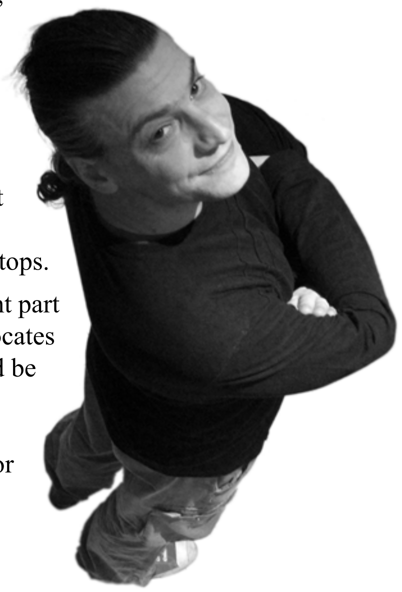
Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

Dedicated Follower of Fashion



What informs our choice of programming language when developing software systems? Not just languages, really, because I'm sure similar forces apply to the decision to deploy any new technology, from platforms to embedded databases.

Perhaps the choice is taken from us: 'We'll do it in C, as we don't have time to develop a compiler from scratch.' More generally, this gets the vote because it's too difficult to get Python deployed to the server farm, or the latest version of the .Net runtime installed on all the users' desktops.

Where there is choice to be had, fashion plays an important part in the decision. Fashionable technologies have many advocates and practitioners, meaning that hiring good people should be easy. Well, possible, then. Of course, following such fashions means that whoever is doing the hiring is in competition with lots of other people hiring for the same or similar skills, so this cuts two ways. Being at the cutting edge here can be a lonely spot.

Fashion is sometimes dressed up into rational, logical argument: 'Java is less dangerous than C++'. The trouble with such arguments is that new ones come along: 'C# is less simplistic than Java', or 'C++ is much faster than either Java or C#'. It's not always a bad thing to make technological advances this way, though; I'm not saying that COBOL is a bad thing, but I sure am glad I don't have to do it...

Sometimes a choice is based on just wanting to learn a new skill. 'We should rewrite it in F# so we can take advantage of new shiny Functional /and/ re-use parts of the existing codebase.' This urge shouldn't be ignored out of hand, either, because it can really motivate a team to learn something new together. Re-writing a large server application from scratch might not be the best way to do it, however.

But if you're stuck working in a language you hate in your job, try finding one you really like outside of work. Maybe if you get good enough, it'll become fashionable enough to get traction in your job.

The Amateur Programmer is still the technological pioneer.

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 24.3:	1st June 2012
C Vu 24.4:	1st August 2012
Overload 110:	1st July 2012
Overload 111:	1st September 2012

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

Some Thoughts on Writing a Scientific Application

Joana Simoes reflects on the challenges in writing software for scientists.

More than an extensive list of tools available for writing scientific applications or a compendium of useful techniques, this article is a reflection about what is involved in writing a scientific application in a non-ideal (and often realistic) situation, based on my personal experience.

I repeat: It does not pretend to be a manual on 'how-to' write scientific applications, but I hope it may present some of the challenges that developers may encounter in this context, and hopefully will be useful for someone. And before I continue: writing scientific applications is a lot of fun, although often it can become 'messy'! (I'll develop the 'messy' side further in the section called 'Ooops').

Why

Scientists are in great need of software, and although they don't always realise this, they need 'good' software. It is fairly easy to imagine why they need software: it may automate tasks that are boring or simply impossible to do manually (like revealing patterns in data), it allows cross referencing between different types of data, it speeds up the most difficult calculations, and so on. Software (and hardware) has had a big impact in most scientific fields, and this stimulated the development of a huge community of programmers within the scientific community. Often programming is not their main activity and they only adopt software engineering practices that they consider to be 'important'.

Often what they consider to be important is to have a piece of software that runs. Coupled with tight deadlines and a certain degree of illiteracy from managers (often themselves scientists with little knowledge of software engineering), this results in presenting a *functional* piece of software.

Unfortunately this approach leaves aside many of the non functional characteristics of code such as expressiveness, modularity, etc, making later tasks such as upgrades or even maintenance more difficult. Similarly, practices such as versioning, automated documentation and unit tests are often disregarded, which may also prevent the development of a solid software product.

This, more than choosing a language or a framework, is what I mean when I say that scientists 'need good software'. It is our task as developers to explain that, and present them with structured and solid software projects.

How

Like many people in ACCU I am a C++ developer, and I like to use C++ as much as I can. Is C++ adequate for writing scientific applications? I would say it really depends on the application.

Often scientists want a piece of software up and running as soon as possible, so I would say C++ is not good for that. On the other hand, often they also want applications to be memory efficient, and to be able to run on many platforms: in that case C++ could be quite a good choice. The initial time trade-off could be easily forgiven if the application is going to be widely used, adapted and improved, having a long life. On the other hand, I would say if you want to do something quickly just to use once, maybe it is not really worth the effort.

Unless you are programming numerical routines for a nuclear reactor, your application is going to be used by people

THE PROGRAMMER'S NIGHTMARE: A SCIENTIST WHO WANTS TOO MANY THINGS AND CANNOT EXPRESS HIMSELF

As C++ is a language that does not come with a standard framework such as .NET or JAVA Swing, it is generally a good idea to search for libraries that provide the extra functionality we may need for the application.

In my case I went for Nokia's QT framework, which is free, open source, and cross platform [1].

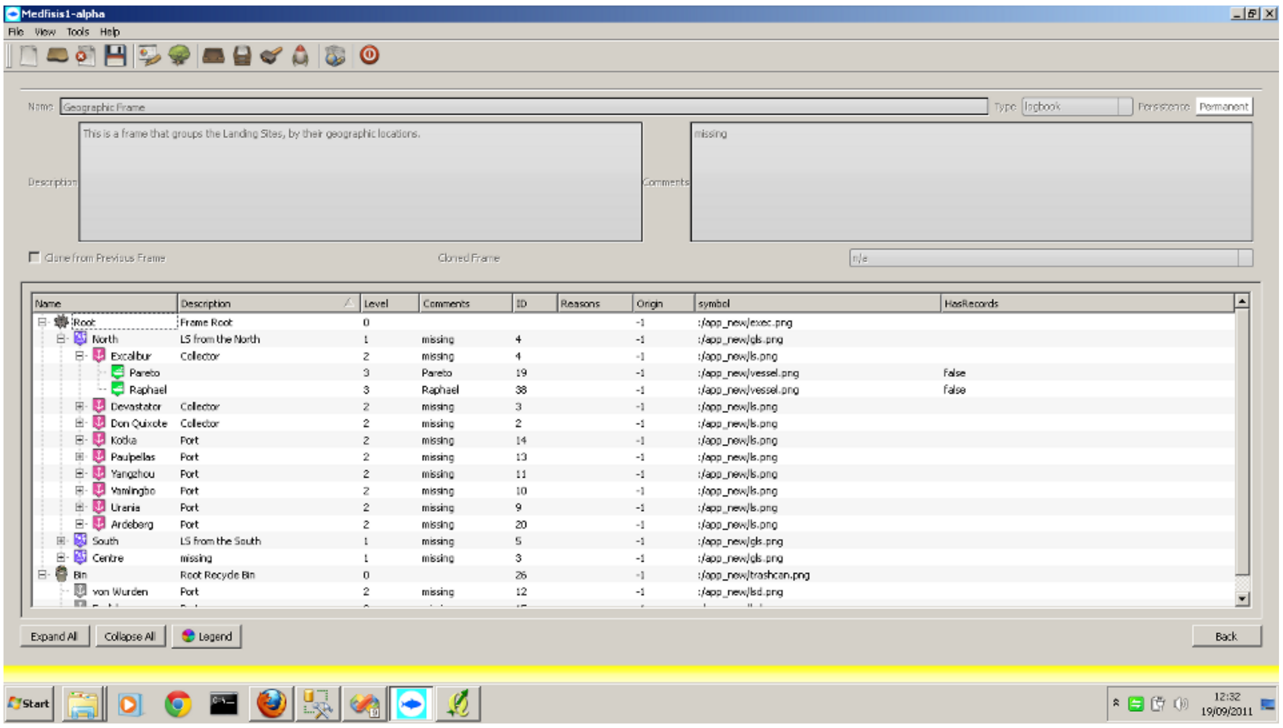
Qt is easily downloaded and installed from the web, and most of the important functionality is already in the compiled version. If you want some extra stuff you may compile it yourself, which is only *slightly* more complicated; maybe I could write an article about that, if people are interested!

If you use MS Visual Studio on Windows, Qt has a really neat integration, with intellisense and a designer launched inside Visual Studio.

Apart from a lot of other functionality, Qt provides classes for three important components of scientific applications, from my perspective: User Interface, Databases and XML (the last one is more questionable, but in my experience, scientists like XML a lot!). Unless you are programming numerical routines for a nuclear reactor, your application is going to be used by people (scientists) and the user interface is going to be the 'visible face' of the software: it is a very important, and often disregarded, part of

JOANA SIMOES
Joana is an AGILE software developer for the Food and Agriculture Organisation (UN), who dreams of becoming a comic artist one day. Contact her by email at doublebyte@gmail.com, or in person in Park Guell (Barcelona) at lunchtime

Figure 1


Medfisis1-alpha

File View Tools Help

Name: Geographic Frame   Type: logbook   Persistence: Permanent

Description: This is a frame that groups the Landing Sites, by their geographic locations.

Comments: missing

Clone from Previous Frame   Cloned Frame: n/a

Name	Description	Level	Comments	ID	Reasons	Origin	symbol	HasRecords
Root	Frame Root	0				-1	:/app_new/exec.png	
North	LS from the North	1	missing	4		-1	:/app_new/gls.png	
Excalibur	Collector	2	missing	4		-1	:/app_new/ls.png	
Pareto		3	Pareto	19		-1	:/app_new/vessel.png	False
Raphael		3	Raphael	38		-1	:/app_new/vessel.png	False
Devastator	Collector	2	missing	3		-1	:/app_new/ls.png	
Don Quixote	Collector	2	missing	2		-1	:/app_new/ls.png	
Kotka	Port	2	missing	14		-1	:/app_new/ls.png	
Paulpellas	Port	2	missing	13		-1	:/app_new/ls.png	
Yangzhou	Port	2	missing	11		-1	:/app_new/ls.png	
Yamingbo	Port	2	missing	10		-1	:/app_new/ls.png	
Urania	Port	2	missing	9		-1	:/app_new/ls.png	
Ardeberg	Port	2	missing	20		-1	:/app_new/ls.png	
South	LS from the South	1	missing	5		-1	:/app_new/gls.png	
Centre	missing	1	missing	3		-1	:/app_new/gls.png	
Bin	Root Recycle Bin	0		26		-1	:/app_new/trashcan.png	
von Wurden	Port	2	missing	12		-1	:/app_new/lsd.png	

scientific software. In my opinion, the popularity of Visual Basic among scientists (and many others) to a great extent is due to how easy it is to setup a UI and link it to code. Qt offers a native-looking UI and, in my opinion, is much easier to implement than Microsoft Foundation Classes (MFC). See Figure 1, which shows a tree-like widget used to help in designing a sampling frame.

I could probably mention many other libraries that could be useful to support the development of a scientific application, but one that I cannot avoid mentioning is Boost [2]. In fact it provides really useful classes for numerical calculations, and became so important that some parts of it have been incorporated in the C++ Standard [3]. As a side note, I would like to emphasise how the Boost 'Smart Pointers' [4] ease the task of memory management: one of the big 'monsters' of C++, which often keeps people away from the language (probably not the people of ACCU, but many scientists at least).

Ooops

Concerning the implementation, I like to keep AGILE [5], especially in scientific fields where requirements are so dynamic. Often the writing of the software is itself part of the research process, and since we are dealing with the unknown, it is basically impossible to define all requirements at the beginning of the project (if that is *ever* possible).

If we add to this the problem that scientists are not software engineers, and they may not express themselves correctly nor appreciate the trouble involved in frequently rewriting parts of the code, we may have a difficult situation here.

Often the writing of the software is itself part of the research process

C++ is not the best language for prototyping, especially when we do it in many iterations. Since the prototyping often involves the UI, I would strongly advise not to touch any code until ideas about the user interface have settled down (and even so, experience tells me there will be changes). For this, I find it very useful to use mockup tools such as Balsamiq [6] (see Figure 2).

This is a user-friendly and very 'fun' tool, that is almost 100% functional even after the trial period expired (except that you cannot save your work). If you want to go for a free and open source alternative, you can try pencil [7], which even runs on a browser (see Figure 3).

Producing mockups – as complete as possible – and discussing them widely before implementation revealed a really useful strategy for me that saved me many hours of (inglorious) work. Also encouraging the scientists to change (or build) the mockups themselves seems like an involving and recommendable practice to make the project more AGILE.

Figure 2

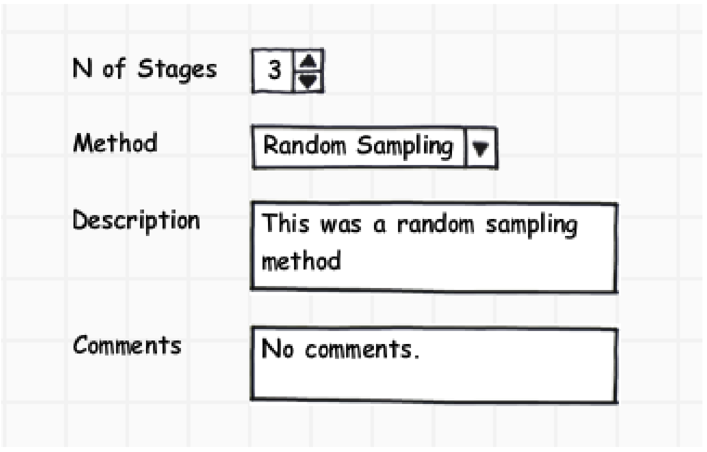
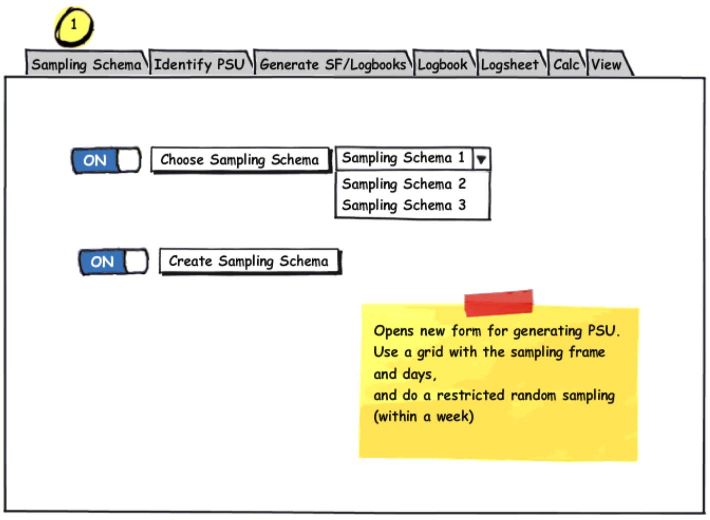

N of Stages	3
Method	Random Sampling
Description	This was a random sampling method
Comments	No comments.

Figure 3



Opens new form for generating PSU.
Use a grid with the sampling frame and days,
and do a restricted random sampling (within a week)

Conclusions

A one man (or one woman) software team seems quite unrealistic to me. Unfortunately, this seems to be extremely common in the scientific community. Often one developer has the role of database manager/developer, analyst, project manager, tester, designer, usability specialist, technical writer, and (almost forgot!) developer!

This should not be an excuse to disregard some aspects of the software project, but unfortunately time is limited so it often is. From my practical experience the most useful advice I can give is: don't overlook the 'project manager' role. Split the project into small tasks and try to explain clearly everything you do (difficulties you face, etc). Since you are likely to be the *only* software engineer in the team, you will probably do the job carefully (I mean, well). Establishing good communication with the scientists and involving them in the project (granting them responsibility for the successes and failures) is probably one of the most important components for building a scientific application; if it is well established it will save you a lot of time (and frustration) and it may well be the difference between success or failure.

See you soon

See you soon with some articles about installing and using the Qt framework. You can email me to suggest specific parts of Qt you would like to see covered, or to beg me to stop writing more articles! ■

References

[1] http://qt.nokia.com/

[2] http://www.boost.org/

[3] https://en.wikipedia.org/wiki/Boost_%28C%2B%2B_libraries%29

[4] http://www.boost.org/doc/libs/1_49_0/libs/smart_ptr/smart_ptr.htm

[5] http://en.wikipedia.org/wiki/Agile_software_development

[6] http://www.balsamiq.com/

[7] http://pencil.evolus.vn/en-US/Home.aspx

THE 'ILLUMINATED' DEVELOPER

Join the ACCU

visit www.accu.org for details

cqf.com

CERTIFICATE IN QUANTITATIVE FINANCE CQF

Expand Your Mind and Career

Designed by quant expert Dr Paul Wilmott, the CQF is a practical six month-part time course that covers every gamut of quantitative finance, including derivatives, development, quantitative trading and risk management.

Find out more at **cqf.com**.

ENGINEERED FOR THE FINANCIAL MARKETS

Our Differences Make Us Stronger

Pete Goodliffe works with QA to produce great software.

Whenever you're in conflict with someone,
there is one factor that can make the difference
between damaging your relationship and
deepening it. That factor is attitude.
~ William James (Philosopher and Psychologist)

In the previous instalment of this column [1] the Beatles told us to increase the love in our teams, and we learnt to stop shovelling manure on the QA department. Let's now look at the practical ways we can work better with the inhabitants of the QA kingdom. We'll do this by looking at the major places that developers interact with QA.

Releasing a build to QA

We know that the development process isn't a straight line; it's not a simple pipeline. We develop iteratively and release incremental improvements; either a new feature that needs validation or a fixed bug that should be validated. It's a cycle that we go round many times. Over the course of the construction process we will create numerous builds that will be sent to QA.

So we need a smooth build and handoff process.

This is a vital; the handoff of our code must be flawless: the code must be responsibly created and thoughtfully distributed. Anything less is an insult to our QA colleagues.

We must build with the right attitude: giving something to QA is not the act of throwing some dog-eared code, built on any-old machine, over the fence for them. It's not a slapdash or slipshod act.

Also, remember that this is not a battle: we don't aim to *slip* a release past QA, deftly avoiding their defence. Our work must be high quality, and our fixes correct. Don't cover over the *symptoms* of the more obvious bugs and hope they'll not have time to notice the underlying evils in the software.

Be very clear exactly what new functionality is and isn't implemented: exactly what is known to work and what is not.

Rather, we must do everything we can to ensure that we provide QA with something worthy of their time and effort. We must avoid any silly errors, or frustrating side-tracks. Not to do so shows a lack of respect to them.

Not creating a QA build thoughtfully and carefully shows a lack of respect to the testers.

This means:

- Prior to creating a release build, the developers should have done as good a job as possible to prove that it is correct. They should have tested the work they've done beforehand. Naturally, this is best achieved with a comprehensive suite of regularly run unit tests. This helps catch any behavioural *regressions* (reoccurrences of previous errors). Automated tests can weed out silly mistakes and embarrassing errors that would waste the tester's time and prevent them finding more important issues.

  With or without unit tests, the developers *must* have actually tried the new functionality, and satisfied themselves that it works as well as is required. This sounds obvious, but all too often changes or fixes that should 'just work' get released, and cause embarrassing problems. Or a developer sees their code working in a simple case, considers it adequate for release, and doesn't even think about the myriad ways it could fail or be mis-used.
- Of course, running a suite of unit tests is only as effective as the quality of those tests. Developers take full responsibility for this. The test set should be thorough and representative. Whenever a fault is reported from QA, demonstrative unit tests should be added to ensure those faults don't reappear after repair.
- When a build is being made, the developer must know exactly *how* a build is expected to work. We don't produce a build and just say '*see how this one works*'.

  Be very clear exactly what new functionality is and isn't implemented: exactly what is known to work and what is not. Without this information you cannot direct what testing is required. You will waste the testers' time. You communicate this in *release notes*.
- So: it's important to draw up a set of good, clear release notes. Bundle them with the build in an unambiguous way (for example: in the deployment file, or with a filename that matches the installer). The build must be given a (unique) version number (perhaps with an incrementing *build number* for each release). The release notes should be versioned with this same number.

  For each release, the release notes should clearly state what has changed, and what areas require greater testing effort.
- Never rush out a build, no matter how compelling it seems. The pressure to do this is greatest as a deadline looms, but it's also tempting to sneak a build out before leaving the office for the evening. Rushing work like this encourages you to cut corners, not check everything thoroughly, or pay careful attention to what you're doing. It's just too easy. And it's not the right way to give a release to QA. Don't do it.

Never rush the creation of a build. You will make mistakes.

If you feel like a school kid desperately trying to rush their homework and get 'something' in on time, in the full knowledge that the teacher will be annoyed and make you do it again, then something it wrong! Stop. And think.

- Some products have a more complex testing requirements than others. Only kick off an expensive test run across platforms/OSes if you think it's worthwhile; when an agreed number of features/fixes have been implemented.
- Automation of manual steps always removes the potential for human error. So automated your build/release process as much as

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

possible. If you can create a single script that automatically checks out the code, builds it, runs all unit tests, creates installers/deploys on a testing server and uploads the build with its release notes, then you remove the potential for human error for a number of steps. Avoiding human error with automation helps to create release that install properly each time and do not contain any regressions. The QA guys will love you for that.

The delivery of code into QA is the act of producing something stable and worthy of potential release, not the act of chucking the latest untested build at QA. Don't throw a code grenade over the fence, or pump raw software sewage at them.

On getting a fault report

We give the test guys a build. It's our best effort yet, and we're proud of it. They play with it. Then they find faults. Don't act surprised. You knew it was going to happen.

Testing will only reveal problems that software developers added to the system (by omission or commission). If they find a fault, it was your fault!

On finding a bug, they lodge a *fault report*: a trackable report of the problem. This report can be prioritised, managed and, once fixed, checked for later regression.

It is *their* responsibility to provide accurate, reliable fault reports, and to send them through in a structured and orderly way – using a good bug tracking system, for example. But faults can be maintained in a spreadsheet, or even by placing stories in a work backlog. (I've seen all these work.) As long as there's a clear system in place that records and announces changes to the state of a fault report.

So how do we respond to a fault report?

First, remember that QA *aren't* there to prove that you're an idiot and make you look bad. The fault report isn't a personal slight. So don't take it personally.

Don't take fault reports personally. They are not a personal insult!

Our 'professional' response is 'thanks, I'll look into it'. Just be glad it was QA who they found it, and not a customer. You *are* allowed to feel disappointed that a bug slipped through your net.

You should be worried if you are swamped by so many fault reports that you don't know where to start – this is a sign that something very fundamental is wrong and needs addressing. If you're in this kind of situation, it's easy to resent each new report that comes in.

But we have unit tests!

We are conscientious coders. We want to make rock-solid software. We want to craft great lines of code with a coherent design, that contribute to an awesome product.

That's what we do.

So we employ development practices that ensure our code is as good as possible. We review, we pair, we inspect. And we test. We write automated unit tests.

We have tests! And they pass! The software must be good. Mustn't it?

Even with unit tests coming out of our ears, we still can't guarantee that our software is perfect. The code might operate as the developers intended, with green test lights all the way. But that may not reflect what the software is supposed to do.

The tests may show that all input the developers envisioned are handled correctly. But that may not be what the user will actually do. Not all of the use cases (and abuse cases) of the software have been considered up-front. It is hard to consider all such cases – software is a mightily complex thing. Thinking about this is exactly what the QA people are great at.

Because of this, a rigorous testing and QA process are still a vital part of a software development process, even if we have comprehensive unit tests in place. Unit tests act as our responsible actions to prove that the code is good enough before we hand it on to the testers to work on.

Of course, we don't leap onto every single fault as soon as it is reported. Unless it is a trivial problem with a super-simple fix, there are almost certainly more important problems to address first. We must work in collaboration with all the project stakeholders (managers, product specialists, customers, and so on) to agree which are the most pressing issues to spend our time on.

Perhaps the fault report is ambiguous, unclear, or needs more information. If this is the case work *with* the reporter to clarify the issues so you can both understand the problem fully, can reproduce it reliably, and know when it has been closed.

QA can only uncover bugs from development, even if it's not a fault that *you* were the direct author of. Perhaps it stems from a design decision that you had no control over. Or perhaps it lurks in a section of code that you didn't write. But it is a healthy and professional attitude to take responsibility for the *whole product*, not just your little part of the codebase.

Our differences make us stronger

Effective working relationships stem from the right developer attitudes. When we're working with a QA team we must understand and exploit our differences:

- Testers are very different from developers. Developers often lack the correct mindset to test effectively. It requires a particular way of

YOU'VE BEEN WORKING IN QA FOR JUST ONE WEEK
YOU'VE FOUND SO MANY BUGS THAT YOU'RE MAKING US ALL LOOK BAD
SO... YOU'RE FIRED
RESULT: BUG COUNT REDUCED

looking at software, particular skills and peccadilloes to do well. We must respect the QA team for these skills – skills that are essential if we want to produce high-quality software.

- A tester is inclined to think more like a user than a computer; they can give valuable feedback on perceived product quality, not just on correctness. Listen to their opinions and value them.
- When a developer works on a feature, their natural instinct is to focus on the happy path – on how the code works when everything goes well (when all input is valid, when the system is working fully with maximum CPU, no memory/disk space issues, and every system call works perfectly).

  It's easy to overlook the many ways that software can be used incorrectly, or to overlook whole classes of invalid input. We are wired to consider our code through these natural cognitive biases. Testers tend not to be straight jacked by such biases.
- Never succumb to the fallacy that QA are just 'failed devs'. There is a common misconception that they are somehow less intelligent, or less able. This is a damaging point of view and must be avoided.

Cultivate a healthy respect for the QA team. Enjoy working with them to craft excellent software.

Pieces of the puzzle

We need to see testing *not* as the 'last activity' in a classic waterfall model; development just doesn't work like that. Once you get 90% of the way through a waterfall development process into testing, you will likely discover another 90% of effort is required to complete the project. You cannot predict how long testing will take, especially when you start it far too late in the process.

Just as code benefits from a test-first approach, so does the entire development process. Work with the QA department and get their input early on to help make your specifications verifiable, ensure their expertise feeds into product design, and that they will agree that the software is maximally testable before you even write a line of code.

A tester is inclined to think more like a user than a computer; they can give valuable feedback on perceived product quality

You cannot predict how long testing will take, especially when you start it far too late in the process

QA are not the owners of, nor the gatekeepers of 'quality'. It is everyone's responsibility.

To build quality into our software and to ensure we work well together, all developers should understand the full QA process and appreciate its intricate details. That process will not work without healthy relationships. *All we need is love.* ■

Questions

1. How healthy are your release procedures? How can you improve them? Ask the QA team what would help them most.
2. Who is responsible for the 'quality' of your software? Who gets the 'blame' when things go wrong? How healthy is this?
3. How good do you think your testing skills are? How methodically do you test a piece of code you're working on before you check in/hand off?
4. How many silly faults have you let slip through your coding net recently?
5. What could you add to your development regimen *in addition to unit tests* to ensure the quality of the software you hand to QA?

Acknowledgements

Many thanks to Lisa Crispin and Jon Moore for their comments on a draft of these articles.

References

[1] 'Getting One Past the Goalpost'. Pete Goodliffe, In: *CVu* 24.1

Have Your Efficiency, and Flexibility Too

Nick Sabalausky writes no-compromise code by metaprogramming with D.

Efficiency and flexibility are programming's classic odd couple. Both are undeniably useful, but they never seem to get along. You try to improve one, and the other just balks and storms off. Prima donna! That might be fine for some scenes, but often you can''t get by with only one: It breaks the whole dynamic!

Just look how the efficiency and flexibility knuckleheads bicker in even the simplest situations. Listing 1 (from `ex1_original.d`) is written in the D programming language [1], but it's the same sad old story in any language.

Ok, so I guess the fighting between efficiency and flexibility isn't so obvious at a glance. They're making some ordinary-looking Gizmos and nothing seems immediately wrong. There are no fists flying, no colourful primetime-banned language, no headlocks or piledrivers. But you have to look closer: it's passive-aggression. And experts say that's psychologically damaging, right?

Normally, code like that would be fine, so what's the problem? Well, we don't just have two or three Gizmos collecting dust until a rare special occasion when we decide to pull one out to use it. Oh, no. Gizmos are the main component in the real product: the UltraGiz. The UltraGiz is made of thousands of Gizmos of all different types, and it really gives those Gizmos a big workout. Plus, each port-zapping is fairly quick. The real expense comes from how many port-zaps occur.

Listing 1

```
struct Gizmo
{
  this(int numPorts, bool isSpinnable)
  {
    if(numPorts < 1)
      throw new Exception
         ("A portless Gizmo is useless!");
    ports.length = numPorts;
    _isSpinnable = isSpinnable;
  }
  private OutputPort[] ports;
  @property int numPorts()
  {
    return ports.length;
  }
  void doStuff()
  {
    foreach(port; ports)
      port.zap();
  }

  private bool _isSpinnable;
  @property int isSpinnable()
  {
    return _isSpinnable;
  }
  int spinCount;
  void spin()
  {
  // Trying to spin a non-spinnable Gizmo is OK.
  // Like insulting a fishtank,
  // it merely has no effect.
    if(isSpinnable)
        spinCount++; // Spinning! Wheeee!
  }
}
```

Source code

Full source code for this article is available on GitHub at: http://github.com/Abscissa/efficientAndFlexible

So even a small improvement to the Gizmo's size and speed will add up to a big improvement in the UltraGiz. And since many different types of Gizmos are needed, we know we need both efficiency and flexibility.

What flexibility is needed? For one, some Gizmos need to spin, and some don't. But every Gizmo is paying the price for that flexibility: There's always that **`spinCount`** variable, even for ones that don't spin. And they all have to take the time to check the **`isSpinnable`** variable. Heck, each Gizmo even has to use storage space just to know whether it's spinnable or not. They're not just stamped on the side with 'spinny' or 'no spinny'.

And then there's the output ports. Every time any Gizmo is called upon to do its stuff, it has to check how many ports there are, zap the first one, increment some internal value, check if it's done, zap the next one, etc. That's necessary for a few of the Gizmos, but most Gizmos only have one or two ports. Why can't they just zap their one or two ports and be done with it? Most Gizmos have no need for that extra overhead.

Ultimately, the problem boils down to all that flexibility coming from runtime variables. Since the flexibility happens at runtime, the compiler can't usually optimize away the speed issues. And since all the Gizmos are the same type, struct Gizmo, the compiler can't optimize the space issues either.

Let's see how the Gizmos currently perform. I'll use this test program to simulate an UltraGiz and time it in Listing 2 (taken from `ex1_original.d`).

On my 1.7 Ghz Celeron (Yes, I know that's old, please don't flame me!), compiling with DMD v2.053 in release mode with optimizations and inlining on, my result is 21 seconds. My system's task manager tells me it used 10.4 MB of RAM. Hmm, even on my old hardware, that could really be better.

Flexibility is really starting to push his co-star's buttons. Efficiency is even getting ready to take a swing at Flexibility. Uh, oh! At this point, many people just decide to prioritize either efficiency or flexibility. Often, this comes with reasons like 'Hardware just keeps getting faster' or 'This is built for speed, those who need flexibility can use a slower alternative'. I've never liked to compromise, and I think we can do better. So let's see if we can diffuse this odd couple's situation before it turns into an all-out brawl (and they consequently lose their lucrative time-slot due to prime-time decency standards).

NICK SABALAUSKY

Nick Sabalausky has been programming most of his life (low-power embedded systems, videogames and web development). His latest interests are training, computer language processing, and all aspects of software design. Nick can be contacted via http://semitwist.com/contact

First attempt: send Efficiency and Flexibility to Dr. Oop's couples therapy

Dr. Oop has had much success helping many couples overcome their differences. He's often the go-to guy for many programming difficulties, and for very good reason. After listening to our protagonists' story, he prescribes interfaces and subclassing. To avoid any need for multiple

Listing 2

```
struct OutputPort
{
  int numZaps;
  void zap()
  {
    numZaps++;
  }
}

struct UltraGiz
{
  Gizmo[] gizmos;
  int numTimesUsedSpinny;
  int numTimesUsedTwoPort;
  private void useGizmo(ref Gizmo gizmo)
  {
    gizmo.doStuff();
    gizmo.spin();
    if(gizmo.isSpinnable)
      numTimesUsedSpinny++;
    if(gizmo.numPorts == 2)
      numTimesUsedTwoPort++;
  }
  void run()
  {
    StopWatch stopWatch;
    stopWatch.start();

    //  Create gizmos
    gizmos.length = 50_000;
    foreach(i;      0..10_000)
       gizmos[i] = Gizmo(1, false);
    foreach(i; 10_000..20_000)
       gizmos[i] = Gizmo(1, true );
    foreach(i; 20_000..30_000)
       gizmos[i] = Gizmo(2, false);
    foreach(i; 30_000..40_000)
       gizmos[i] = Gizmo(2, true );
    foreach(i; 40_000..45_000)
       gizmos[i] = Gizmo(5, false);
    foreach(i; 45_000..50_000)
       gizmos[i] = Gizmo(5, true );

    // Use gizmos
    foreach(i; 0..10_000)
    foreach(ref gizmo; gizmos)
            useGizmo(gizmo);
    writeln(stopWatch.peek.msecs, "ms");
    assert(numTimesUsedSpinny
       == 25_000 * 10_000);
    assert(numTimesUsedTwoPort
       == 20_000 * 10_000);
  }
}

void main()
{
  UltraGiz ultra;
  ultra.run();
  // Runtime error: A portless Gizmo is useless!
  //auto g = Gizmo(0, true);
}
```

Listing 3

```
interface ISpinner
{
  @property bool isSpinnable();
  void spin();
}

final class SpinnerStub : ISpinner
{
  @property bool isSpinnable()
  {
    return false;
  }
  void spin()
  {
    // Do nothing
  }
}

final class Spinner : ISpinner
{
  @property bool isSpinnable()
  {
    return true;
  }
  int spinCount;
  void spin()
  {
    spinCount++; // Spinning! Wheeee!
  }
}

abstract class Gizmo
{
  this()
  {
    spinner = createSpinner();
  }

  @property int numPorts();
  void doStuff();

  ISpinner spinner;
  ISpinner createSpinner();
}

class OnePortGizmo : Gizmo
{
  override ISpinner createSpinner()
  {
    return new SpinnerStub();
  }

  private OutputPort[1] ports;
  override @property int numPorts()
  {
    return 1;
  }
  override void doStuff()
  {
    ports[0].zap();
  }
}

class TwoPortGizmo : Gizmo
{
  override ISpinner createSpinner()
  {
    return new SpinnerStub();
  }
```

Listing 3 (cont'd)

```
  private OutputPort[2] ports;
  override @property int numPorts()
  {
    return 2;
  }
  override void doStuff()
  {
    ports[0].zap();
    ports[1].zap();
  }
}

class MultiPortGizmo : Gizmo
{
  this(int numPorts)
  {
    if(numPorts < 1)
      throw new Exception("A portless Gizmo
      is useless!");

    if(numPorts == 1 || numPorts == 2)
      throw new Exception("Wrong type of
      Gizmo!");

    ports.length = numPorts;
  }
  override ISpinner createSpinner()
  {
    return new SpinnerStub();
  }
  private OutputPort[] ports;
  override @property int numPorts()
  {
    return ports.length;
  }
  override void doStuff()
  {
    foreach(port; ports)
      port.zap();
  }
}

final class SpinnyOnePortGizmo : OnePortGizmo
{
  override ISpinner createSpinner()
  {
    return new Spinner();
  }
}

final class SpinnyTwoPortGizmo : TwoPortGizmo
{
  override ISpinner createSpinner()
  {
    return new Spinner();
  }
}

final class SpinnyMultiPortGizmo : MultiPortGizmo
{
  this(int numPorts)
  {
    super(numPorts);
  }
  override ISpinner createSpinner()
  {
    return new Spinner();
  }
}
```

inheritance or code duplication (both are known to have problems), he'll also add in a touch of composition. See Listing 3 (from `ex2_objectOriented.d`).

Oh dear God, what have we done?! Blech!

Ok, calm down...Deep breaths now...Stay with me...Breathe...Breathe...Maybe it's not as bad as it seems. Maybe it'll be worth it. After all, it's technically flexible. Not pretty, but flexible. Maybe the efficiency will be good enough to make it a worthwhile compromise. Let's see...

The code to test this version is almost the same as before so I won't show it here. But you can view it in `ex2_objectOriented.d` if you'd like.

On my system, this takes 40 seconds and 11.3 MB. That's nearly twice the time and 10% more memory as before. Hmm, uhh...nope, no good. Well, that was a bust.

So what went wrong? The problem is, object orientation involves some overhead. Polymorphism requires each instance of any Gizmo type to store some extra hidden data, which not only increases memory usage but also allows fewer Gizmos to fit into the cache. Polymorphism also means an extra indirection when calling a member function. This extra indirection can only sometimes be optimized away. Each Gizmo needs to be individually allocated (although it's possible to get around that in certain languages, including D, but it's still yet another thing to do). The by-reference nature of objects means the Gizmo arrays only contain pointers. Not only does that mean greater memory usage, but it can also decrease data locality (how 'close together' related data is in memory) which leads to more cache misses. Using composition for the spin capability also decreased data locality, increased indirection, and increased memory usage. All things considered, we wound up doing the exact opposite of what we tried to do: Our attempts to decrease time and memory increased them instead, and also gave us less maintainable code.

In many cases, the overhead of object orientation isn't really a big problem, so object orientation can be a very useful tool (although perhaps not so much in this case). But in highly performance-sensitive sections, the overhead can definitely add up and cause trouble.

So for all the successes Dr. Oop has had, efficiency and flexibility are just too strongly opposed. Flexibility is left unhappy with the complexity required, and poor efficiency nearly had a heart attack! This time, Dr. Oop's solution just isn't quite what the patients has been hoping for. Oops, indeed.

Programmers familiar with templated classes might be annoyed at this point that I've rejected the object oriented approach without considering the use of template classes. Those familiar with D are likely screaming at me, 'Mixins! Mixins!' And then there's C++'s preprocessor, too. Well, frankly, I agree. Such things can certainly improve an object oriented design. But those are all forms of metaprogramming, which I haven't gotten to just yet. Besides, the main point I want to get across is this: Object orientation isn't a general substitute for metaprogramming and does have limitations in how well it can marry efficiency with flexibility.

Respecting the classics: old-school handcrafting

Object orientation may not have worked out well for efficiency, but efficient code has been around since long before objects became popular. How did they do it back then? With good old-fashioned handcrafting, of course. Time for Efficiency and Flexibility to pay a visit to the town elder...

After the elder introduces himself, Efficiency and Flexibility ask him to have a look at their problem.

> 'Eh? You want I should look at your problem?'
>
> 'Yes, we'd like you to help us out.'
>
> 'Help on your problem right? You want I should look at?'
>
> 'Umm...yes...'
>
> 'Ok...Hi! I'm the town elder!'

Clearly this guy has a problem repeating himself. But eventually he pulls out his trusty oak-finished toolchain and gets to work. After what seems

Listing 4

```
struct OnePortGizmo
{
  static immutable isSpinnable = false;
  static immutable numPorts    = 1;

  private OutputPort[numPorts] ports;
  void doStuff()
  {
    ports[0].zap();
  }
  void spin()
  {
    // Do nothing
  }
}

struct TwoPortGizmo
{
  static immutable isSpinnable = false;
  static immutable numPorts    = 2;
  private OutputPort[numPorts] ports;
  void doStuff()
  {
    ports[0].zap();
    ports[1].zap();
  }

  void spin()
  {
    // Do nothing
  }
}

struct MultiPortGizmo
{
  this(int numPorts)
  {
    if(numPorts < 1)
      throw new Exception("A portless Gizmo is
      useless!");

    if(numPorts == 1 || numPorts == 2)
      throw new Exception("Wrong type of
      Gizmo!");
    ports.length = numPorts;
  }
  static immutable isSpinnable = false;

  private OutputPort[] ports;
  @property int numPorts()
  {
    return ports.length;
  }
  void doStuff()
  {
    foreach(port; ports)
      port.zap();
  }

  void spin()
  {
    // Do nothing
  }
}

struct SpinnyOnePortGizmo
{
  static immutable isSpinnable = true;
  static immutable numPorts    = 1;
```

Listing 4 (cont'd)

```
  private OutputPort[numPorts] ports;
  void doStuff()
  {
    ports[0].zap();
  }

  int spinCount;
  void spin()
  {
    spinCount++; // Spinning! Wheeee!
  }
}

struct SpinnyTwoPortGizmo
{
  static immutable isSpinnable = true;
  static immutable numPorts    = 2;

  private OutputPort[numPorts] ports;
  void doStuff()
  {
    ports[0].zap();
    ports[1].zap();
  }

  int spinCount;
  void spin()
  {
    spinCount++; // Spinning! Wheeee!
  }
}

struct SpinnyMultiPortGizmo
{
  this(int numPorts)
  {
    if(numPorts < 1)
      throw new Exception("A portless Gizmo is
      useless!");

    if(numPorts == 1 || numPorts == 2)
      throw new Exception("Wrong type of
      Gizmo!");

    ports.length = numPorts;
  }

  static immutable isSpinnable = true;

  private OutputPort[] ports;
  @property int numPorts()
  {
    return ports.length;
  }

  void doStuff()
  {
    foreach(port; ports)
      port.zap();
  }

  int spinCount;
  void spin()
  {
    spinCount++; // Spinning! Wheeee!
  }
}
```

Listing 5

```
struct UltraGiz
{
  OnePortGizmo[]          gizmosA;
  SpinnyOnePortGizmo[]    gizmosB;
  TwoPortGizmo[]          gizmosC;
  SpinnyTwoPortGizmo[]    gizmosD;
  MultiPortGizmo[]        gizmosE;
  SpinnyMultiPortGizmo[]  gizmosF;

  int numTimesUsedSpinny;
  int numTimesUsedTwoPort;

  // Ok, technically this is a simple form of
  // metaprogramming, so I'm cheating slightly.
  // But I just can't bring myself to copy/paste
  // the exact same function six times even for
  // the sake of an example.
  void useGizmo(T)(ref T gizmo)
  {
    gizmo.doStuff();
    gizmo.spin();
    if(gizmo.isSpinnable)
      numTimesUsedSpinny++;
    if(gizmo.numPorts == 2)
      numTimesUsedTwoPort++;
  }

  void run()
  {
    StopWatch stopWatch;
    stopWatch.start();

    //  Create gizmos
    gizmosA.length = 10_000;
    gizmosB.length = 10_000;
    gizmosC.length = 10_000;
    gizmosD.length = 10_000;
    gizmosE.length =  5_000;
    gizmosF.length =  5_000;

    foreach(i; 0..gizmosE.length)
      gizmosE[i] = MultiPortGizmo(5);
    foreach(i; 0..gizmosF.length)
      gizmosF[i] = SpinnyMultiPortGizmo(5);
    // Use gizmos

    foreach(i; 0..10_000)
    {
      foreach(ref gizmo; gizmosA)
         useGizmo(gizmo);
      foreach(ref gizmo; gizmosB)
         useGizmo(gizmo);
      foreach(ref gizmo; gizmosC)
         useGizmo(gizmo);
      foreach(ref gizmo; gizmosD)
         useGizmo(gizmo);
      foreach(ref gizmo; gizmosE)
         useGizmo(gizmo);
      foreach(ref gizmo; gizmosF)
         useGizmo(gizmo);
    }
    writeln(stopWatch.peek.msecs, "ms");
    assert
       (numTimesUsedSpinny  == 25_000 * 10_000);
    assert
       (numTimesUsedTwoPort == 20_000 * 10_000);
  }
}
```

like an eternity later, he's done with Listing 4 (which is from `ex3_handcrafted.d`).

It certainly matches the old man's speech patterns, but just look at the careful attention to detail and workmanship! Two handmade single-port Gizmos, one spinny and one not. Two handmade double-port Gizmos. And even a couple general-purpose multi-port jobs. Let's take 'er for a...ahem...a spin...

On my system, that clocks in at 10.5 seconds and 9.4 MB. Hey, not bad! That's definitely an improvement over the original. It's twice as fast, and uses about 10% less memory. Those memory savings are even better than they sound because the measurements include process and runtime overhead. If we had 10x or 100x as many Gizmos, the memory savings would be more than 10%.

Note that these Gizmos never spend time checking whether or not they're spinny. They just spin or they don't. For the one and two porters, there's no for loop when port-zapping, so no time is spent updating and checking an iteration variable. Additionally, the variables **`spinCount`** and **`ports`** only exist for Gizmos that actually need them – they don't take up space in other Gizmos. The **`isSpinny`** variable was even eliminated outright. All these tweaks add up to some real savings.

The old guy's a bit eccentric, and his methods may be a bit out of date, but he really knows his stuff. Too bad the approach is too meticulous and error-prone to be useful for the rest of us mere mortals. Or for those of us working on modern large-scale high-complexity software.

I should point out that since the Gizmos are now separate types, with no common base type, they can no longer be stored all in one array. But that's not a big problem, we can just keep a separate array for each type. No big deal. And if we wanted, we could create a struct **`GizmoGroup`** that kept arrays of all the different gizmo types in a convenient little package. The town elder didn't actually make such a struct, but in any case, Listing 5 has the updated UltraGiz (from `ex3_handcrafted.d`).

In reality, specially handcrafting only-slightly-different versions is such a meticulous, repetitive maintenance nightmare that you'd normally only make one or two specially-tweaked versions, and leave the rest of the cases up to a general-purpose version. And even that can be a pain. So as happy as efficiency may be, flexibility is storming out of the room. We're getting close, but haven't succeeded yet. What we need is a better twist on this handcrafting approach...

Success at Dr. Metaprogramming's Clinic

Dr. Metaprogramming listens to the story, pauses for a second, and replies, 'Turn your runtime options into compile-time options.' Say what? The metaprogramming doc takes the original problem, moves **`numPorts`** and **`isSpinnable`** up to the struct Gizmo line, makes just a few small changes, resulting in Listing 6 (from `ex4_metaprogramming.d`).

Dr. Metaprogramming points out, 'As an added bonus, trying to make a portless Gizmo is now caught at compile-time instead of runtime'.

Efficiency whines, 'Look at all those ifs!' The doc explains that those aren't real **`if`**s, they're static **`if`**. They only run at compile-time.

Efficiency responds, 'Oh, ok. So you're really making many different types, right? Isn't there an overhead for that, like with polymorphism?' The doc says it does make many different types, but there's no runtime polymorphism (just compile-time) and no overhead. Efficiency smiles.

Seeing Efficiency happy makes Flexibility concerned. Flexibility balks, 'We occasionally require some logic to determine a Gizmo's number of ports and spinnability, so I doubt we can do this.' The doc assures him that D can run many ordinary functions at compile-time. And in other languages, code can just be generated as a separate step before compiling, or a preprocessor can be used. He adds that even if runtime logic really is needed, there are ways to do that, too. This will all be demonstrated in detail in part 2. Flexibility smiles.

The code to test this is still very similar to the other versions. But since this is the first metaprogramming version, I'll show the new Gizmo-testing code in Listing 7 (from `ex4_metaprogramming.d`).

Listing 6

```
struct Gizmo(int _numPorts, bool _isSpinnable)
{
  // So other generic code can determine the
  // number of ports and spinnability:
  static immutable numPorts    = _numPorts;
  static immutable isSpinnable = _isSpinnable;

  static if(numPorts < 1)
    static assert(false,
       "A portless Gizmo is useless!");

  private OutputPort[numPorts] ports;
  void doStuff()
  {
    static if(numPorts == 1)
      ports[0].zap();
    else static if(numPorts == 2)
    {
      ports[0].zap();
      ports[1].zap();
    }
    else
    {
      foreach(port; ports)
        port.zap();
    }
  }

  static if(isSpinnable)
    int spinCount;

  void spin()
  {
    static if(isSpinnable)
      spinCount++; // Spinning! Wheeee!
  }
}
```

Listing 7

```
struct OutputPort
{
  int numZaps;
  void zap()
  {
    numZaps++;
  }
}

struct UltraGiz
{
  // We could still use gizmosA, gizmosB, etc.
  // just like before, but templating them will
  // make things a little easier:
  template gizmos(int numPorts, bool isSpinnable)
  {
    Gizmo!(numPorts, isSpinnable)[] gizmos;
  }

  int numTimesUsedSpinny;
  int numTimesUsedTwoPort;

  void useGizmo(T)(ref T gizmo)
  {
    gizmo.doStuff();
    gizmo.spin();

    if(gizmo.isSpinnable)
      numTimesUsedSpinny++;
```

Listing 7 (cont'd)

```
    if(gizmo.numPorts == 2)
      numTimesUsedTwoPort++;
  }

  void run()
  {
    StopWatch stopWatch;
    stopWatch.start();

    // Create gizmos
    gizmos!(1, false).length = 10_000;
    gizmos!(1, true ).length = 10_000;
    gizmos!(2, false).length = 10_000;
    gizmos!(2, true ).length = 10_000;
    gizmos!(5, false).length =  5_000;
    gizmos!(5, true ).length =  5_000;

    // Use gizmos
    foreach(i; 0..10_000)
    {
      foreach(ref gizmo;
        gizmos!(1, false)) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(1, true )) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(2, false)) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(2, true )) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(5, false)) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(5, true )) useGizmo(gizmo);
    }
    writeln(stopWatch.peek.msecs, "ms");
    assert(numTimesUsedSpinny
       == 25_000 * 10_000);
    assert(numTimesUsedTwoPort
       == 20_000 * 10_000);
  }
}

void main()
{
  UltraGiz ultra;
  ultra.run();
  // Compile time error: A portless Gizmo is
  // useless! auto g = Gizmo!(0, true);
}
```

One of the important things to note here is that the function **`useGizmo()`** is templated to accept any type. This is necessary since there are multiple Gizmo types instead of just one. So effectively, there is now a separate **`useGizmo()`** function for each Gizmo type (although a smart linker might combine identical versions of **`useGizmo()`** behind-the-scenes). In the next section, I'll get back to the matter of this function being templated, but for now, just take note of it.

Also, the arrays **`gizmosA`**, **`gizmosB`**, etc. were replaced by a templated array. This is just like the separate arrays from the handcrafted version, but it gives us a better way to refer to them. For example, we now say **`gizmos!(2, false)`** instead of **`gizmosC`**. This may seem to be of questionable benefit, especially since we could have just named it **`gizmos2NoSpinny`**. But it will come in handy in the later metaprogramming versions since it lets us use arbitrary compile-time values to specify the two parameters. That gives us more metaprogramming power. But that will come later.

This version gives me 10.1 seconds and 9.2 MB. That's just as sleek and slim as the handcrafted version and...wait no...huh? It's slightly better? Granted, it's not by much, but what's going on?

It may seem strange that generic code could be more efficient than a specially handcrafted non-generic version. But at least part of what's happening is that with metaprogramming, the compiler is essentially doing your handcrafting automatically as needed.

Remember, in the real handcrafted version, the town elder only handcrafted one-port and two-port versions. For everything else, he had to fallback to the original strategy of dealing with a variable number of ports at runtime. With the metaprogramming version on the other hand, the compiler automatically 'handcrafted' a special five-port version when we asked for five ports. If we had also asked for three-port and seven-port versions, it would have automatically 'handcrafted' those as well. It's possible to create and maintain all those special version manually, but it would be very impractical.

If you really do want a single type for general multi-port Gizmos just like the town elder's handcrafted version, that's certainly possible with metaprogramming, too. In fact, we'll get to that later.

Of course, I don't mean to imply that handcrafted optimization is obsolete. There are always optimizations a compiler can't do. But when your optimization involves creating alternate versions of the same thing, metaprogramming makes it quick and easy to apply the same technique on as many different versions as you want without significantly hindering maintainability.

I've alluded to a number of flexibility enhancements that can be made to this metaprogramming version. I'll explain these next time in part 2, as promised. But there's one enhancement I'd like to cover before I leave:

It walks like a duck and quacks like a duck...kill it!

Duck typing is a topic that divides programmers almost as much as 'tabs vs spaces' or 'vi vs emacs'. While I admit I'm personally on the anti-duck side of the pond, I'm not going to preach it here. I only bring it up because there are other anti-duckers out there, and for them, there's something about the metaprogramming example they may not be happy with – but there is a solution. If you are a duck fan, please pardon this section's title and feel free to skip ahead. I promise not to say anything about you behind your back...

Remember in the last section I pointed out the **`useGizmo()`** function was templated so it could accept all the various Gizmo types? Well, what happens when we pass it something that isn't a Gizmo? For most types, the compiler will just complain that **`doStuff()`**, **`spin`**, **`isSpinnable`**, and **`numPorts`** don't exist for the type. But what if it's a type that just happens to look like a Gizmo? (See Listing 8, from `snippet_notAGizmo.d`).

Listing 8

```
struct BoatDock_NotAGizmo
{
  // Places to park your boat
  int numPorts;
  void doStuff()
  {
    manageBoatTraffic();
  }

  // Due to past troubles with local salt-
  // stealing porcupines swimming around and
  // clogging up the hydraulics, some boat docks
  // feature a special safety mechanism:
  // "Salty-Porcupines in the Intake are
  // Nullified", affectionately called
  // "spin" by the locals.
  bool isSpinnable;
  void spin()
  {
      blastTheCrittersAway();
  }
}
```

I don't mean to imply that handcrafted optimization is obsolete

Listing 9

```
template isIGizmo(T)
{
  immutable bool isIGizmo = __traits(compiles,
  // This is just an anonymous function. We won't
  // actually run it, though. We're just making
  // sure all of this compiles for T.
    (){
      T t;
      static assert
        (T._this_implements_interface_IGizmo_);
      int n = t.numPorts;
      static if(T.isSpinnable)
        int s = t.spinCount;
      t.doStuff();
      t.spin();
    }
  );
}

// Almost identical to the original
// metaprogramming Gizmo in
// 'ex4_metaprogramming.d', but with two
// little things added:
struct Gizmo(int _numPorts, bool _isSpinnable)
{
  // So other generic code can determine the
  // number of ports and spinnability:
  static immutable numPorts    = _numPorts;
  static immutable isSpinnable = _isSpinnable;

  // Announce that this is a Gizmo.
  // An enum takes up no space.
  static enum _this_implements_interface_IGizmo_
    = true;

  // Verify this actually does implement the
  // interface
  static assert(
    isIGizmo!(Gizmo!(numPorts, isSpinnable)),
    "This type fails to implement IGizmo"
  );

  static if(numPorts < 1)
    static assert(false,
      "A portless Gizmo is useless!");

  private OutputPort[numPorts] ports;
  void doStuff()
  {
    static if(numPorts == 1)
      ports[0].zap();
    else static if(numPorts == 2)
    {
      ports[0].zap();
      ports[1].zap();
    }
    else
    {
      foreach(port; ports)
        port.zap();
    }
  }
```

Listing 9 (cont'd)

```
  static if(isSpinnable)
    int spinCount;
  void spin()
  {
    static if(isSpinnable)
      spinCount++; // Spinning! Wheeee!
  }
}
struct OutputPort
{
  int numZaps;
  void zap()
  {
    numZaps++;
  }
}
struct UltraGiz
{
  template gizmos(int numPorts, bool isSpinnable)
  {
    Gizmo!(numPorts, isSpinnable)[] gizmos;
  }
  int numTimesUsedSpinny;
  int numTimesUsedTwoPort;
  void useGizmo(T)(ref T gizmo) if(isIGizmo!T)
  {
    gizmo.doStuff();
    gizmo.spin();
    if(gizmo.isSpinnable)
      numTimesUsedSpinny++;
    if(gizmo.numPorts == 2)
      numTimesUsedTwoPort++;
  }
  void run()
  {
    StopWatch stopWatch;
    stopWatch.start();

    // Create gizmos
    gizmos!(1, false).length = 10_000;
    gizmos!(1, true ).length = 10_000;
    gizmos!(2, false).length = 10_000;
    gizmos!(2, true ).length = 10_000;
    gizmos!(5, false).length =  5_000;
    gizmos!(5, true ).length =  5_000;

    // Use gizmos
    foreach(i; 0..10_000)
    {
      foreach(ref gizmo;
        gizmos!(1, false)) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(1, true )) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(2, false)) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(2, true )) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(5, false)) useGizmo(gizmo);
      foreach(ref gizmo;
        gizmos!(5, true )) useGizmo(gizmo);
    }
    writeln(stopWatch.peek.msecs, "ms");
    assert(numTimesUsedSpinny
       == 25_000 * 10_000);
    assert(numTimesUsedTwoPort
       == 20_000 * 10_000);
  }
}
```

The templated **`useGizmo()`** function will happily accept that. Hmm, accepting a type based on its members rather than its declared name, what does that remind you of...? Yup, duck typing.

Granted, it's not exactly the same as the usual duck typing popularized by dynamic languages. It's more like a compile-time variation of it. But it still has the same basic effect: If something looks like a Gizmo, it will be treated as a Gizmo whether it was intended to be or not. Whether or not that's acceptable is a matter for The Great Duck Debate, but for those who dislike duck typing, it's possible to kill the duck with metaprogramming and constraints. Listing 9 (from `ex5_meta_deadDuck1.d`) is almost identical to the metaprogramming code, but with a few changes and additions that I've highlighted.

Those experienced with the D programming language may notice this is very similar to the way D's ranges are created and used, but with the added twist that a type must actually declare itself to be compatible with a certain interface.

Now, if you try to pass a boat dock to **`useGizmo()`**, it won't work because the boat dock hasn't been declared to implement the **`IGizmo`** interface. Instead, you'll just get a compiler error saying there's no **`useGizmo()`** overload that can accept a boat dock. As an extra bonus, if you change Gizmo and accidentally break its **`IGizmo`** interface (for instance, by deleting the **`doStuff()`** function), you'll get a better error message than before. Best of all, these changes have no impact on speed or memory since it all happens at compile-time.

Under the latest version of DMD at the time of this writing (DMD v2.053), if you break Gizmo's **`IGizmo`** interface, Figure 1 shows the error message you'll get:

So it plainly tells you what type failed to implement what interface. In a language like D that supports compile-time reflection, it's also possible to design the **`IGizmo`** interface so the error message will state which part of the interface wasn't implemented. But the specifics of that are beyond the scope of this article (That's author-speak for 'I ain't gonna write it.')

This is great, but announcing and verifying these dead-duck interfaces can be better generalized as in Listing 10 (taken from `ex5_meta_deadDuck2.d`). Changes from Listing 9 are highlighted.

If you ever want to create another type that also counts as an **`IGizmo`**, all you have to do is add the declaration line:

From `snippet_anotherGizmo.d`:

```
struct AnotherGizmo  // A class would work, too!
{
  mixin(declareInterface("IGizmo",
                         "AnotherGizmo"));
  // Implement all the required members of
  // IGizmo here...
}
```

Now **`isIGizmo`** will accept any **`AnotherGizmo`**, too. And just like a real class-based interface, if you forget to implement part of **`IGizmo`**, the compiler will tell you.

There are many further improvements that can be made to **`declareInterface()`**. For instance, although it's currently using a string mixin, it could be improved by taking advantage of D's template mixin feature. It could also be made to detect the name of your type so you only have to specify **`"IGizmo"`**, and not **`"AnotherGizmo"`**. But this at least demonstrates the basic principle.

It may seem strange that generic code could be more efficient than a specially handcrafted non-generic version

Figure 1

```
ex5_meta_deadDuck1.d(44):    Error: static assert  "This type fails to implement IGizmo"
ex5_meta_deadDuck1.d(92):    instantiated from here: Gizmo!(numPorts,isSpinnable)
ex5_meta_deadDuck1.d(116):   instantiated from here: gizmos!(1,false)
```

Listing 10

```
string declareInterface(string interfaceName,
string thisType)
{
  return `
  // Announce what interface this implements.
  // An enum takes up no space.
  static enum

_this_implements_interface_`~interfaceName~`_
     = true;
  // Verify this actually does implement the
  // interface
  static assert
     (is`~interfaceName~`!(`~thisType~`),
     "This type fails to implement
     `~interfaceName~`"
  );
  `;
}

// Almost identical to the original
// metaprogramming Gizmo in
// 'ex4_metaprogramming.d', but with
// *one* little thing added:
struct Gizmo(int _numPorts, bool _isSpinnable)
{
  // So other generic code can determine the
  // number of ports and spinnability:
  static immutable numPorts    = _numPorts;
  static immutable isSpinnable = _isSpinnable;

  // Announce and Verify that this is a Gizmo.
  mixin(declareInterface("IGizmo",
     "Gizmo!(numPorts, isSpinnable)"));
  static if(numPorts < 1)
     static assert(false,
     "A portless Gizmo is useless!");
  private OutputPort[numPorts] ports;
  void doStuff()
  {
    static if(numPorts == 1)
      ports[0].zap();
    else static if(numPorts == 2)
    {
      ports[0].zap();
      ports[1].zap();
    }
    else
    {
      foreach(port; ports)
        port.zap();
    }
  }
  static if(isSpinnable)
    int spinCount;
  void spin()
  {
    static if(isSpinnable)
      spinCount++; // Spinning! Wheeee!
  }
}
```

For the sake of simplicity, the examples in this article's upcoming second half will forgo the anti-duck typing techniques covered in this section.

Of course, none of this is needed if you're only using classes, which can truly inherit from one another. In that case, you can just use real inheritance-based interfaces. But if you want to avoid the overhead of classes, you can use these metaprogramming tricks to achieve much of the same flexibility.

Intermission

So far, we've examined the 'efficiency vs flexibility' conflict and identified limitations of a couple traditional approaches for reconciliation. We have also seen that metaprogramming offers ways around those limitations and promises fewer tradeoffs between efficiency and flexibility.

Next time will be a little more technical as we delve deeper into metaprogramming to see how it offers more flexibility than one might expect.

Stay tuned for the thrilling conclusion! ■

References

[1] http://dlang.org

Desert Island Books

Lisa Crispin is marooned on the island.

I have been a huge fan of Agile and testing in general for a very long time, so when the opportunity to review Lisa Crispin and Janet Gregory's *Agile Testing: A Practical Guide for Testers and Agile Teams* presented itself I was very pleased. As usual I posted my reivew [1] to my blog, sent it off to Jez for *CVu* and then thought no more about it.

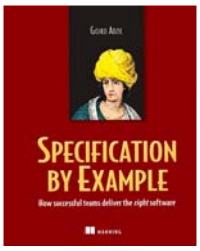
A few months later, completely out of the blue, I received an email from Lisa thanking me for the review and asking if I would be in London the following November as they were presenting at Skillsmatter. Unfortunately I couldn't make it, but I took the opportunity to mention the ACCU conference for which, another few months later, Lisa submitted and had accepted a session.

Unfortunately I missed that conference and so I have yet to meet Lisa in person. However, she expects to be over in the UK in 2013 and I'm hoping that's something I can put right.

Lisa Crispin

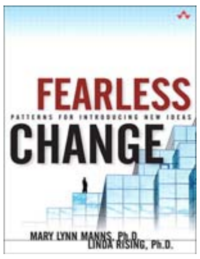
Disclaimer: I'm not actually a programmer, though I started out my software life as one. I do some coding of test scripts, but I'm a tester. That said...

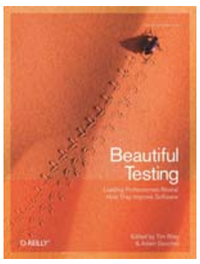
One book I'd bring on the desert island is *Everyday Scripting with Ruby: For Teams, Testers and You* by Brian Marick. I learned Ruby (at least, enough Ruby to competently write test scripts that drive UI tests with Watir) working through this book. It would provide me many happy hours of learning and practice on the desert island.



Specification by Example: How successful teams deliver the right software by Gojko Adzic is another book I'd want along, because I love reading the stories of how other teams succeeded in delivering what customers want, and I'd learn from the examples, and could imagine more examples for myself. I don't want to get bored on this island! It seems a bit recursive to talk about learning about examples through examples, but I think we all learn best by examples, and we also succeed with understanding what our business experts want by asking them for their examples.

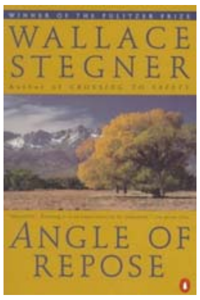
Since this is a desert island, I guess there is nobody else for me to try to influence. Nevertheless, I think I'd bring along *Fearless Change: Patterns for Introducing New Ideas* by Linda Rising and Mary Lynn Manns. A decade ago when I was a Kool-aid-drinking XPer, I joined a company that talked a lot about XP but really only did chaos. *Fearless Change* taught me why evangelizing about how great XP values, principles and practices are doesn't change anyone. I learned to try different patterns to try to influence people. More importantly, I understood patterns better, and patterns help with everything! I'd like to write some testing patterns, so if I have this book with me as an example, that will also keep me entertained on the desert island.





There are so many good books and apparently the carry-on limit to this island is small, so to get some variety in inspirational reading, I'd take *Beautiful Testing: Leading Professionals Reveal How They Improve Software*. What's better than reading about the beauty of testing, as written by 27 of my peers? And each bears reading several times, with new nuggets of learning each go-round. I contributed a chapter to this book, so choosing it seems a bit self-promotional, but I'd choose it even if I weren't in it.

It's even harder to choose just one novel to bring along, because reading is one of my favourite activities, has been all my life, and I have a house full of books. I've actually made some tough decisions and given some to the local library, because we hope to move to a horse property, and it's crazy to lug so many books around in the age of eBooks.



So to just pick ONE – I'd go with *Angle of Repose* by Wallace Stegner. It appeals to my love of history, and my love of the West (of the U.S.), since I live in the Rocky Mountains. (OK, right in the foothills of the Rocky Mountains, not the actual mountains). This book earned a well-deserved Pulitzer prize, and nobody knew or loved the Western U.S. as well as Stegner. He's one of my heroes. Plus, you get to feel smart afterward when you know what an 'angle of repose' is. I can read it over and over on my island and never get tired of it.

TWO albums – again, there's an impossible choice. I love music, too. But I will pick John Prine's *Souvenirs*, because it has many of my favorite John Prine tunes on it, and I'm limited here, I can't take *John Prine*, *Sweet Revenge* and *Jesus: The Missing Years* with me. Plus, I do like Prine's acoustic versions of some of my very favourites, such as *Far From Me*, *Angle From Montgomery*, *Christmas in Prison*, and

What's it all about?

Desert Island Disks is one of Radio 4's most popular and enduring programmes. The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml).

The format of 'Desert Island Books' is *slightly* different from the Radio 4 show. You choose about five books, one of which must be a novel, and up to two albums. Some people even throw in the odd film. Quite a few ACCUers have chosen their Desert Island Books to date and there are plenty more to go.

The rules aren't too strict but the programming books must have made a big impact on your programming life or be ones that you would take to a desert island. The inclusion of a novel and a couple of albums helps us to learn a little more about you. The ACCU has some amazing personalities and Desert Island Books has proved we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

ACCU Oxford

An evening of lightning talks, reviewed by David Mansergh.

On 29th February ACCU Oxford held an interesting and enjoyable evening of 'Lightning talks' on a variety of software and process topics. Lightning talks last only a few minutes and allow several speakers to deliver presentations in a single meeting. The chosen format on this occasion was 10-minute talks, each followed by time for a few questions. There were no restrictions on the topic, only on the time for presenting.

Arnaud Desitter gave the first talk on 'Not-A-Number (NaN) and floating point exceptions as defined by IEEE754'. The quote that 'This standard is arguably the most important in the industry' (Michael L Overton) helped to grab everyone's attention. He went on to explain the difference between 'quiet' and 'signaling' NaNs and how signaling NaNs can be used to detect uninitialized floating point variables without impacting performance.

Bibek Bhattacharya followed with a talk about using 'Native C++ with the new Microsoft PPL'. This started with a discussion of hardware trends and the latest parallel gizmos available to Windows software developers. The benefits of PPL were then powerfully demonstrated using the example of parallelizing millions of runs of Fibonacci number calculations.

Malcolm Noyes then presented 'Enforcing Code Feature Requirements in C++, revisited'. After lulling us into false sense of security with pictures of some famous C++ faces he then bent our minds with talk of templates, covariance and contravariance. After reading the C++ standard and some other books, a few iterations of code and some head scratching, Malcolm had been able to achieve what had been claimed to be impossible.

Robin Williams followed that with 'So what's the fuss about Lua?' He suggested that those seeking to learn a new scripting language might do well to choose Lua given that it can be learnt in a reasonably short time. Both the benefits and short comings of Lua were covered in what was a lightning speed lightning talk.

Jesus Bouzada described his experience of 'Using Visual Control to avoid broken build problems'. This emphasized the benefits of visually displaying build results when using a continuous integration process. We are bombarded with emails and overloaded with web pages so need the build status to be simply and clearly displayed on a screen for it to get our attention. He explained how he had been inspired by some Lean principles to use this solution. This led to a culture change in his team, reducing broken build times and increasing productivity.

Nigel Lester finished the evening with 'Retrospective: Timeline game', an insight into one way of running effective retrospective meetings as part of the software development process. This included many important and helpful practical details that enable a retrospective to run smoothly. The slides showed how a project timeline can be assembled on the wall with happy/surprised/sad/angry items. These are then grouped into clusters, discussed, the findings reviewed and possibly even acted upon!

The evening was well attended and well received by both ACCU members and others. The variety of talks meant that whatever your specialty there was much of interest and plenty to learn. The talks were of a high quality and provided a great opportunity to practice presentation skills. The slides will be available on the ACCU Oxford website (www.lunch.org.uk/wiki/accuoxford) if you would like to learn more.

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

Deserts Island Books (continued)

Blue Umbrella. I heard my first John Prine tunes when I worked at the coolest little bistro called the Grapevine in College Station, Texas while I was in university. At this bistro, each employee was free to work at the job he or she preferred: waiting tables, cash register, preparing food, recommending wine, washing dishes. Leland, a hippie who preferred running the dishwasher, also provided all the music, and he was a John Prine fan. When I listen to John Prine, I'm back at the Grapevine. One night, Leland asked me if I would run away to Hawaii with him. It turned out his parents were extremely wealthy, and lived in Hawaii. I was tempted, but I had a boyfriend already so I turned him down. My boyfriend turned out to be a loser, and I often wonder what my life might have been like if I had run off to Hawaii with a hippie. Seriously, that 'sliding doors' thing! I do love Hawaii, too!

The other album I choose is Nat King Cole's *The Christmas Song*. Though I'm a humanist, I love Christmas, and this is the album my family played first thing on Christmas morning when I was growing up. I've continued that tradition throughout my life. I'm so happy when I sing along with Nat!

Gosh, this was really fun! Sorry to ramble on so about my choices, but I've been thinking about them a lot.

References

[1] http://paulgrenyer.blogspot.co.uk/2009/05/agile-testing-practical-guide-for.html

Next issue: Mick Brooks

Code Critique Competition 75

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

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

Last issue's code

A classic little problem here: can you explain what might be problematic about the class in Listing 1, recently found in an actual production code base…

Listing 1

```
/**
  Singleton template definition
*/
#ifndef _SINGLETON_H
#define _SINGLETON_H

namespace utilities
{
  /** Singleton template */
  template<class T> class Singleton
  {
  public:
    static T* getInstance()
    { return theInstance; }
    static void setInstance(T *instance)
    { theInstance = instance; }

  protected:
    Singleton();
    ~Singleton();

    static T* theInstance;
  };
} // namespace utility

#endif // _SINGLETON_H
```

ROGER ORR

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

Critiques

Peter Sommerlad <peter.sommerlad@hsr.ch>

0. Singletons are at least as bad as global variables. DO NOT USE THEM!

I could go on for pages on that, but that is actually not the question.

Even if we would consider the SINGLETON design pattern OK today, the code given is very problematic.

1. First things first: naming

Naming is important. The first name introduced is the namespace `utility`. That is already a code smell. If something gets a generic name it is always a signal of bad structure and too little thought on where to put things or on what it is. I know the C++ ISO standard is guilty of that as well :-)

But if you have a project where you have a directory/namespace called 'utils', 'utilities', or similar, have a close look. It should be (nearly) empty. Try to classify its contents accordingly to what it does, even if that needs inventing more and better names.

The class template is named **`Singleton`** presuming to implement the SINGLETON design pattern, but it doesn't. SINGLETON's intent is 'Ensure a class only has one instance, and provide a global point of access to it.' The class template **`Singleton`** provides neither!

2. Singleton template design – CRTP?

The SINGLETON design pattern's properties are that of a class. So the first intuitive usage of the template (especially since it has protected members) is

```
struct aclass:utilities::Singleton<aclass>
  {
  };
```

This doesn't give you anything useful. It will fail to compile, because there are no implementations of **`Singleton`**'s default **`ctor`** and **`dtor`**. Well, we could fix that by providing them (and there is no one except the ODR) to let us do so:

```
namespace utilities{
template<class T>
Singleton<T>::Singleton(){}
template<class T>
Singleton<T>::~Singleton(){}
}
```

I consider that a failed attempt in usage.

3. Singleton template design – retrofit

Let us try to retrofit **`Singleton`** property to a class **`bclass`**:

```
struct bclass
{
};
using utilities::Singleton;
typedef Singleton<bclass> theB;
...
  bclass bc=*theB::getInstance();
```

When we use **`theB`** to access our single instance of **`bclass`** as if it were a SINGLETON, we end up in interesting linker error message that **`utilities::Singleton<bclass>::theInstance`** is not defined. This is a major deficiency of the approach, since we need to provide a definition for the declaration of a static member variable in a class. And for a template we need to make sure that such a definition is instantiated exactly once for every template instantiation.

We can 'fix' that by providing that definition in namespace scope: **`template<>  bclass *theB::theInstance=nullptr;`**

But wait, that will give us another problem in that we do not have an instance yet.

Well, if we want a global one and do not need to get the lazy initialization that the SINGLETON pattern provides, we can write

```
namespace {
  bclass ourB;
}
```

```
template<>
bclass *theB::theInstance=&ourB;
```

but with such a global `ourB`, what is the need for `Singleton<bclass>` anyway?

4. Singleton<T> design generals

In the context of multi-threading, setting and retrieving a global (pointer) value is dangerous. `getInstance()` happening concurrently with `setInstance()` is introducing a data race, which means the code could go out and fetch you a hot dog, or worse.

There are several broken attempts to fix that with the SINGLETON Design Pattern, e.g., POSA 2's DOUBLE-CHECKED LOCKING idiom. But it's best just to not go there.

5. Conclusion

`-> DELETE` class template `Singleton` and fix all compile errors resulting from that, if you have any?

I am not sure if I have hit all problems, but I believe it is irrelevant to discuss it further.

6. Way out – limit number of instances

Since globals have been considered REALLY BAD since the passing of BASIC and the introduction of languages with scoped variables and function/subroutine parameters, there is one problem remaining that the original SINGLETON design pattern addresses: 'limit the number of instances'. I'd like to share one of my solutions to that problem that I created to demonstrate the usefulness of non-type template for my students:

```
  #ifndef LIMITNUMBEROFINSTANCES_H_
  #define LIMITNUMBEROFINSTANCES_H_

  #include <stdexcept>
  namespace limitit {
  // meant to be used via CRTP, like class
  // mySingleOne:LimitNofInstances<mySingleOne,1>
  // {...};

template <typename TOBELIMITED,
  unsigned int maxNumberOfInstances>
class LimitNofInstances {
  static unsigned int counter;
protected:
  void checkNofInstances() {
    if(counter == maxNumberOfInstances)
      throw std::logic_error(
        "too many instances");
  }
  LimitNofInstances() {
    checkNofInstances();
    ++counter;
  }
  ~LimitNofInstances() {
    --counter;
  }
  LimitNofInstances(
    const LimitNofInstances &other){
    checkNofInstances();
    ++counter;
    }
  LimitNofInstances &operator=(
    LimitNofInstances const &other)=delete;
  };
  template <typename TOBELIMITED,
    unsigned int maxNumberOfInstances>
    unsigned int LimitNofInstances<TOBELIMITED,
      maxNumberOfInstances>::counter(0);
  } // namespace limitit
  #endif /* LIMITNUMBEROFINSTANCES_H_ */
```

The definition of the static counting member in the header is guaranteeing its instantiation.

Here are some test cases demonstrating its uses:

```
struct highlander: private
  limitit::LimitNofInstances<highlander,1>{
  highlander(){
    std::cout << "the one"<< std::endl;
  }
  ~highlander(){
    std::cout << "head off"<< std::endl;
  }
};

int main(){
  {
  highlander theOne;
  try {
    highlander theother(theOne);
  } catch (...){
    std::cout << "no other one" << std::endl;
  }
  }
  highlander otherscope;
}
```

You can get one `highlander` at a time, but when you try to construct another one while one persists, you get an exception. That even works with two or more and it becomes the feature of a class, but you do not have to provide global access. Note, passing by reference to functions called is still OK.

Herman Pijl <herman.pijl@telenet.be>

The original class template seems to be quite useless. It contains a getter and a setter. The static member is only a straight translation of a global variable into something 'object-oriented'. Let's see whether we can transform this into something more suiting the purpose.

The first problematic feature of the `Singleton` implementation is that the user has no clue whether `Singleton<T>::theInstance` contains something useful. Therefore it needs to be initialized, e.g. by `NULL`.

```
// Singleton.cpp
T* Singleton<T>::theInstance = NULL;
```

Now we have a static method `Singleton<T>:: getInstance()` that returns `NULL`. This is still useless. The user would have to check the result each time and then decide to throw an exception or create an instance and call `setInstance()`.

```
//
if (!Singleton<MyType>::getInstance())
{
  throwOrCreateAndSetInstance();
}
```

It would be better if the `getInstance` method itself could decide to create and set a new instance of `T`. We don't really need the setter in that case, so I will drop it.

```
//
...
public:
  static T * getInstance()
  {
     if (!theInstance)
     {
       theInstance = new T();
     }
     return theInstance;
  }
```

This works fine in a single threaded application, but in a multi threaded application, we could use a `std::mutex` and `std::lock_guard`. This can then be optimized with the DOUBLE-CHECKED LOCKING pattern.

```
  //
  ...
  public:
    static T * getInstance()
    {
      if (!theInstance)
      {
        std::lock_guard guard(theMutex);
        if (!theInstance)
        {
          theInstance = new T();
        }
      }
      return theInstance;
    }
  protected:
    static std::mutex theMutex;
```

Unfortunately, the static member **theMutex** suffers from the initialisation-order-problem.

Therefore I would opt to write

```
  //
  ...
  protected:
    static std::mutex & getMutex() {
      static std::mutex theMutex;
      return theMutex;
    }
```

which guarantees that the mutex is effectively initialized before it is referenced.

After this step, most authors start moaning about the fact that the **Singleton<T>** is not guaranteed to have one instance, so that you have to declare a default constructor, a copy constructor and an assignment operator that have no implementation and thus preventing the **Singleton<T>** class from having more than one instance.

The problem I have with this is that a SINGLETON should force you to have one instance of **T**. I don't care that there are a million instances of **Singleton<T>**. I remember well the case of a new team member who managed to write

```
  //
  ...
    MyType myType;
    myType.doSomeStuff();
```

whereas **MyType** was supposed to be used only as **Singleton<MyType>::theInstance()**. But the code was not preventing him from having multiple instances of **MyType**.

In object-oriented software development, the class design is led by decisions based on relationships like is-a or has-a. Therefore I thought it would be useful to express the relationship that a certain class is-a SINGLETON.

I would express it as with an inheritance relationship

```
  //
  ...
    class MyType: public Singleton<MyType>
    {
    };
```

I wanted to check in the **Singleton<T>** constructor that the the instance being created was **Singleton<T>::theInstance**. This way the user is prevented from instantiating **T** on the stack or elsewhere on the heap.

```
  //
  ...
  template<typename T>
  Singleton<T>::Singleton()
  {
    if (this != getInstance())
    {
      throw std::runtime_error("This object can"
        " only exist in one location");
    }
  }
```

That was rather naive because the call to **Singleton<T>::getInstance()** calls new **T**, which calls the **Singleton** constructor, which calls **Singleton<T>::getInstance()** and the lock guard is not recursive and thus waits forever...

What I really want is to check the memory address. So I decided to separate the allocation of the memory for **T** and the construction of **T**.

```
  //
  ...
  template<typename T>
  void * Singleton<T>::getInstanceAddress()
  {
    static void * theInstanceAddress =
      new char[sizeof(T)];
    return theInstanceAddress;
  }
```

Now the check in the **Singleton<T>** constructor becomes

```
  //
  ...
    if (this != getInstanceAddress())
    {
      throw std::runtime_error("This object can"
        " only exist in one location");
    }
```

and the code in **Singleton<T>::getInstance**

```
  //
  ...
    if (!theInstanceSet)
    {
      std::lock_guard<std::mutex>
      guard(getMutex());
      if (!theInstanceSet)
      {
        new (getInstanceAddress()) T();
        theInstanceSet = true;
      }
    }
```

I think we are nearly at the end of the story.

What happens at the end of the program? The **Singleton** doesn't get destroyed. To avoid resource leaks, like network connections, it is best to properly clean up. With static locals, this should be achieved by the code that is inserted by the compiler. The object **T** is destroyed in place. This can be accomplished by creating a static function **destroyObject** and by registering this function after the constructor call with the **atexit** function.

The whole sixth chapter of Alexandrescu's *Modern C++ Design* is devoted to the implementation of a **SingletonHolder** class using several creation, lifetime and threading model policies. It is recommended literature when introducing or updating a SINGLETON implementation.

Seweryn Habdank-Wojewódzki <habdank@gmail.com>

The question in CC 74 was: '... what might be problematic about the class in Listing 2'.

Short answer: Almost everything :-).

Detailed review:

Let's start with code review, as code review may help find many problems. The code consists of one header file, which contains proper header guard and comment. Comment states that file shall contain **Singleton** definition. After header guard there is namespace **utilities**. However, closing of the namespace that has comment closing namespace **utility**, a bit inconsistent, but that is only a comment.

Then we reach the class code. It is a template with one parameter. The class has the name **`Singleton`**.

And here the true story begins. The class contains static accessors and one static pointer member. The class has protected **`ctor`** and **`d-ctor`**. That is not enough to be a SINGLETON. According to the literature [1], [2] there are certainly more requirements for SINGLETON. Even so, the implementation is not correct. We can enumerate the following requirements:

1. Existence of the instance – the object must exist and have global access,
2. The instance is unique in the proper scope,
3. **`Singleton`** class owns the object instance (lifetime management) and guarantees managing proper access (e.g. in multithreaded environment),
4. There are no resource leaks (general requirement for SW).

Let's collect some remarks about those requirements and how they will be tested. Existence will be tested in the way that **`getInstance()`** function always returns valid and usable object.

Uniqueness is tested by checking if **`getInstance`** is always giving the same object. Here equality of the object (the term 'objects are the same') can be defined in some ways. Very strong equality requires that pointer is always the same, but in fact good SW shall not rely on pointers, so we can relax that requirement to be able to access objects that have the same or continued state. Also uniqueness comes with clear question in what scope the class shall be unique. Is it thread scope, is it process scope, is it machine scope or SW system scope (when e.g. SW system may work on many machines). An interesting example of system wide singleton with continuous state can be 'transaction ID generator' in a distributed system.

Ownership is a weak requirement; however, if the **`singleton`** class does not own instance of the object, it may be very easy to break 1 and 2. Also if we dig into 2 we can see how important is managing of the lifetime for Singleton class.

The last requirement is general purpose, but we should keep it as most SINGLETON implementations are falling foul of it, which is causing problems. In particular, the template does not say anything about resources managed by **`T`**, which could be connections to a database.

There is a little problem that a static member is not defined, but it is easy to fix.

Let's imagine a bit about how the code works. The sequence of commands: **`getInstance`**, **`setInstance`**, **`getInstance`** in simple program immediately breaks requirement about existence and is different than **`setInstance`**, **`getInstance`**, **`getInstance`**. The sequence **`setInstance`**, **`setInstance`**, **`getInstance`** breaks requirement 2, when **`setInstance`** will bring two different pointers, also it will be a lot of fun if pointer will hold object that creates e.g. a thread that prints something on the console output. Also we can see by code review, that object access management requirement is broken with respect to the scope of the process (with many possible threads). The class does not offer any locking mechanism, so the user implements it on top of the class.

Let's follow next problems, if we consider sequence of the commands: **`ptr = new T`**, **`setInstance(ptr)`**, **`getInstance`**, **`delete ptr`**, **`getInstance`** we will see that ownership or lifetime will be violated. The sequence **`setInstance`**, **`setInstance`** will also cause leaks especially when resources will be e.g. DB connections that are not destroyed automatically at the end of process (thread) existence. I would like to avoid rewriting items in the earlier mentioned literature such as other details related to re-creation of the singleton, when it gets destroyed, and the issues where one singleton is used by another singleton at the end of application life (**`atexit`** + SINGLETON long lifetime) as well as problems with multithreaded access (possible usage of DOUBLE CHECKED LOCKING pattern or directly use **`::boost::call_once`**).

Generic SINGLETON implementation is very complex so I will not write up all the fixes needed by the code, but I think it is worth highlighting a couple more problems that may happen. Usually I prefer to say: try to avoid Singletons as much as you can. However, sometimes it is possible to have singletons, depending on the design of the system (separate single process that offers some services, or separate single instance of some app in distributed system).

If we have more Singletons to use in the system, we should consider writing a single Master-Singleton that will manage the lifetime of the rest and also tries to mediate communication between them. As it is stated in C++ FAQ [3], it is quite a big problem to synchronize two statically initialized variables (singletons as well). So if one singleton uses another one we can end up in big trouble.

Finally I would recommend not using singletons at all. If they are absolutely necessary, I would suggest writing clear requirements for them, and try to design them in the system (not only implement them). Finally if an implementation is needed I would follow remarks from literature e.g. mentioned here [2], [3].

References

[1] GoF, *Design Patterns – Elements of reusable Object-Oriented Software*

[2] A. Alexandrescu, *Modern C++ design – Generic Programming and Design Patterns Applied*

[3] http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.14

Commentary

The most important problem with the code presented is that the class is not a singleton. As Peter and Seweryn both point out explicitly, and Herman implicitly, the singleton design pattern is to ensure only a single instance of a class is created and to provide access to it. This class does neither – it is effectively just way of writing a global pointer acessed via **`Singleton<T>::getInstance`** and **`setInstance`**.

Pattern languages are useful when they provide a way of describing something consistently. Abusing the name of a pattern, even for a similar usage, is misleading. To quote from Lewis Carroll's *Alice in Wonderland*: 'When I use a word,' Humpty Dumpty said in rather a scornful tone, 'it means just what I choose it to mean – neither more nor less.' Unfortunately this does not aid communication.

The singleton pattern itself does have a number of problems – as shown above by the various attempts to fix the **`singleton`** class by turning it into an actual implementation of the SINGLETON pattern. This gets more complicated for multi-threaded programs as it can be hard to ensure initialisation happens correctly. [As a side note on this, the C++11 standard guarantees that 'If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.' This means Herman's static variable **`theMutex`** will be initialized just once even in a multi-threaded program that is fully C++11 compliant. However, note that MSVC (even the recent VS11 beta) does not provide this guarantee.]

The second problem with the example code is that, since the class makes no attempt to resolve the issue of ownership the pointed-to object may be leaked. In the case I was looking at this class was used in a dynamic link library plug-in and so one instance of the pointed to object was leaked each time the plug-in was unloaded.

In the event, I followed Peter's advice and simply deleted the class and then resolved the compilation errors – I manged to replace usage of the class with a couple of class-scope static data members.

One final note on Seweryn's mention of **`boost::call_once`**: the new standard contains a standardised version **`std::call_once`** in the **`<mutex>`** header (and this is available in both MSVC 11 and gcc 4.7.)

The Winner of CC 74

All three critiques gave a good explanation of what was wrong with the code supplied, and recommended not using singleton where possible. I was interested by Peter's **`LimitNumberOfInstances`** class as this removes some of the 'global variable' nature of the SINGLETON pattern, although I was unclear how it would work in a multi-threaded program.

I found it hard to decide between the three critiques but eventually picked on Seweryn's as I felt his description (via various use cases) of what was wrong with the singleton template as supplied was the best and so I have awarded him this issue's prize.

Code Critique 75

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

I've got a C component that generates call-backs (in a header `callback.h`). I am trying to use it from C++ so I've wrapped it in a class, but it doesn't quite work. I've put together a test harness using a dummy implementation of the call-back and a **`counter`** class. I expected to see this output:

```
Counter: 1
Counter: 2
Counter: 3
Counter: 4
```

But what I actually got is something like

```
Counter: 1
Counter: 2619565
Counter: 2619566
Counter: 2619567
```

Please help me work out what's going wrong.

The listings are:

- Listing 2: `callback.h`
- Listing 3: `cb.h`
- Listing 4: `counter.h`
- Listing 5: `callbackTest.cpp`

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

Listing 2

```
#ifdef  __cplusplus
extern "C" {
#endif

// Register a callback
//  fn - the function to call back
//  arg - the argument to pass back
void registerCB(
  void (*fn)(void* arg),
  void *arg);

// Unregister a callback
void unregisterCB(
  void (*fn)(void* arg),
  void *arg);

#ifdef  __cplusplus
}
#endif
```

Listing 3

```
#ifndef cb_h_
#define cb_h_

template <typename T>
class CB
{
public:
  CB() : registered(true) {
    ::registerCB(&fn, this);
  }

  static void fn(void* arg)
  {
    static_cast<T*>(arg)->callback();
  }

  ~CB() {
    if (registered) {
      unregisterCB(&fn, this);
    }
  }

  // ...
private:
  bool registered;};

#endif // cb_h_
```

Listing 4

```
#ifndef counter_h_
#define counter_h_

#include "cb.h"

class Counter : CB<Counter>
{
public:
  Counter() : counter(0) {}
  void callback() {
    std::cout << "Counter: "
      << ++counter << std::endl;
  }
  virtual ~Counter() {}
private:
  int counter;
};

#endif // counter_h_
```

Listing 5

```
#include <iostream>

#include "callback.h"
#include "counter.h"

// dummy callback
void (*fn)(void* arg);
void *arg;

void registerCB(
  void (*fn)(void* arg),
  void *arg)
{
  ::fn = fn;
  ::arg = arg;
}
```

Where is ACCU going?

Steve Love invites you to exert some influence.

Those of you who follow the mailing list on Accu-General will not have failed to notice the recent threads about what ACCU is all about. In particular, there was quite a lot of discussion about the contents of the magazines – although mostly focusing on *Overload* – and what the purpose and focus of ACCU is.

Although we certainly have a large core of members whose main attention is drawn by C++ and C, I think a large proportion of us use different technologies and languages, whether instead of C and C++, or in addition to them. It's probably fair to say that in recent times, C++ has not made much of a showing in either *Overload* or *C Vu*. It is even longer, I think, since we've seen an article on C. For some this seems to be an indication that interest in it has waned in recent times, for others it is evidence that ACCU has been somehow 'taken over' by people with differing interests.

My feeling is that it is neither – and both. Perhaps I should explain.

I know of many programmers who come from a background of writing C++ who now do less of it, in the main because they now write code in other languages: Java, C# and Python are the main ones in my experience. It doesn't mean that they are no longer interested in reading about C++ (although that might be true of some), but it does mean they find material about their 'other' languages useful, as well. I also know many programmers who *do* still count C++ as their core language, who also find interest in material not specifically about C++, but for whom it is of less direct relevance.

There is no big conspiracy among the ACCU committees or magazine editors to push C++ or C out. If ACCU has been taken over by anyone, it is the members. The conference has sessions presented largely by people who've submitted a proposal for something they're interested in. A tiny minority of the sessions are 'invited' by the conference committee or chair. Similarly, the vast majority of articles in the magazines are those submitted by members.

This doesn't mean that we have a problem with no resolution. It presents us with an opportunity to satisfy most people most of the time. My message here is really this: if you feel disenfranchised by ACCU, or you don't find the material you see of direct relevance to you, do something about it: write an article on a topic you *do* find interesting. I guarantee – whatever that topic is – there will be someone else who also finds it interesting.

We actively encourage articles in *C Vu* from first-time authors (there have in fact been several of those over the last few editions), and articles on pretty much any broadly technical topic. If *you* want to see more of a particular topic – whatever it is – then it's up to *you* to submit an article on it.

STEVE LOVE

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He is also currently the Features Editor for C Vu, and in this capacity can be contacted at cvu@accu.org

Code Critique Competition (continued)

Listing 5 (cont'd)

```
void unregisterCB(
  void (*fn)(void* arg),
  void *arg)
{
  fn = 0;
  arg = 0;
}

void exercise()
{
  if (fn) fn(arg);
}

// test program
int main()
{
  Counter test;

  // call it myself
  test.callback();

  // use the (dummy) callback mechanism
  exercise();
  exercise();
  exercise();
};
```

while (you care about code)
{
read (cvu && overload);
}
do(it);
because good code matters
ACCU
WWW.ACCU.ORG
PROFESSIONALISM IN PROGRAMMING

Bookcase

The latest roundup of book reviews.

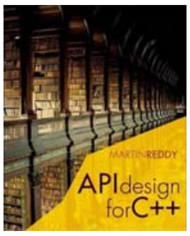
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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

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

Jez Higgins (jez@jezuk.co.uk)

API design for C++



By Martin Reddy, published by Morgan Kaufmann, ISBN: 978-0-12-385003-4

Reviewed by Paul Floyd

Highly Recommended

When I first got this book I flicked through the pages and my first impression was 'a lot of effort has gone into making this'. There's a decent cover photo, the book feels solidly bound and is printed on quality paper and the text is well balanced (not too many bullet lists, a reasonable number of diagrams and code examples). As soon as I started reading it, my impressions were confirmed. Reddy doesn't just cover the act of designing an API, rather he covers most of the field of C++ software development such as design, testing, performance and documentation, but at each stage presents the activity with respect to APIs. Occasionally I felt that he'd wandered a little too far from the API subject, but I can forgive such short digressions.

So what does the book cover? It starts with the basics of what an API consists of. Then it delves into design and different types of API (like OO and template). There's a chapter devoted to C++ aspects of APIs. The end part of the book covers libraries, versioning and extensibility (through plugins and scripting).

The text is well thought out and contains many references (the bibliography runs to four pages, always a good sign). If you read this book, not only will you learn a lot about APIs but you'll also learn a lot about C++ software development.

Business Patterns for Software Developers

By Allan Kelly, published by Wiley, ISBN: 978-1119999249

Reviewed by Paul Grenyer

Nat Pryce, one of the authors of *Growing Object Oriented Software Guided by Tests*, is quoted as saying that pattern books generally have two sections. First is the is a highly interesting preamble, then come the patterns. Allan Kelly has certainly written a patterns book in the traditional two parts, the difference is that both parts are very interesting indeed. The description of software company lifecycles was both informative and sensational to read and is by far my favourite part of the book.

I had genuine trouble putting the book down. Gone is the often dry pattern descriptions common among other patterns authors. What give this book the edge for me is that I'm learning about something other than software and I am easily able to relate the patterns to my own experiences. This is the first time I've really understood what a pattern language is and, like all good patterns books, it taught me names for, and gave me a greater understanding of, the patterns I see around me. It helped me understand the forces and the solutions in greater detail.

Exploratory Software Testing

By James A. Whittaker, published by Addison Wesley, ISBN: 978-0321636416

Reviewed by Guiseppe Vacanti

This is a book about testing large software systems with complex user interfaces. By large I mean as large as the entire Windows operating system. The author is Test Engineering Director at Google, and was formerly at Microsoft, and he has been developing his ideas about testing first in academia and then industry.

The central tenet of this book is that more manual testing, more sophisticated and more repeatable, is required in order to improve the quality of software systems, especially large software systems.

We are all familiar with the fact that testing even a small function by providing the complete set of possible inputs it might one day see is not always possible, because in most cases that set is at best very very large, and often infinite. Put a complex GUI in front of a complex system, and hand this over to a user, and you need a special mindset to even begin imagining what kind of key combinations might be presented to your software.

We need a new approach, says Whittaker. Software techniques have significantly evolved over the last couple of decades, but testing techniques have not. While automatic testing can help, automatic testing alone cannot help improve the quality of software. In passing, the author remarks that he believes too much reliance on automatic testing is what made Windows Vista a less successful product than it might otherwise have been.

The solution is to give testers more autonomy in their work, give them a broad description of what parts of the software they are supposed to explore, but not prescribe exactly what inputs they are supposed to use and in which order. This the author calls exploratory testing.

How to go about this? By introducing a metaphor: the tester is like a tourist visiting a large city, with a lot of attractions, cultural and otherwise. Should (s)he take a guided tour of the must-not-miss venues, or take the tube to a random station and start walking through the less known neighbourhoods? These are two of the many tourist metaphors Whittaker introduces. The list is long, but the test team is

Bookshops

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

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

invited to select the approaches that best fit the system they are working on.

For instance, the Money Tour. In the Money Tour the tester identifies the features of a software package that are key to a user. Many of these features are those used by the sales people, who give demos to potential new clients. They will know what the 'money' features are, and they will also use a number of shortcuts to make their demos run smoothly. These are the features the tester targets in this tour. Or the Back Alley Tour, where the least used features are targeted, exactly because they are less used and therefore less tested.

In order to make his story more concrete, Whittaker asked some of his (ex-)colleagues to describe some of the tours they took, explaining why they thought a certain approach was warranted, and what they discovered that they might otherwise not have. This is an important contribution to the book, as otherwise the metaphor approach would have risked coming across as too abstract, more like a caricature of what testers do than a concrete technical solution.

The appendix (that accounts for almost half of the book) reproduces large sections of the author's blog, with commentary. This is interesting to understand how some of the ideas behind exploratory testing came about and evolved.

All in all an interesting book, and although the examples all cover large software systems, some of the ideas presented could be fruitfully applied to the testing of smaller systems too.

How We Test Software at Microsoft

By Alan Page, Ken Johnston and Bj Rollison, published by Microsoft Press, ISBN: 978-0735624252

Reviewed by Paul Floyd

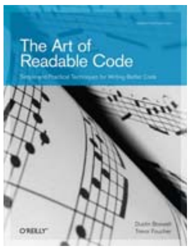
Recommended.

With such a provocative title, I can't resist taking one cheap shot. In chapter 1, all the boxes around the text boxes are shifted to the right, and in some cases the text is covered by the 'bright idea' lightbulb. I did wonder if this was intentional, and that on page two hundred and something there would be 'and that was how we found the shifted text box bug'. But no, it looks like it was just some ordinary error (human, Word, or typesetting).

I was expecting more in the way of war stories. There are some, mostly adding a bit of humour. The main themes that are covered are the testing organization within Microsoft (which can be a bit strong in the positive attitude department at times), testing techniques and testing of services. The parts covering testing techniques are reasonably technical, going into the details of equivalence partitioning (i.e., don't run tests that add nothing new to the testing) and boundary value analysis. However, it is fairly theoretical. The authors point out that Microsoft is a huge organization and that many different methods and tools are used within the company. They also mention that there are thousands of in-house developed tools used for testing, which is nice if you work for Microsoft, but not much use to the rest of us.

If there's one thing from this book that I should apply to my work, it would be to have something like the 'Customer Experience Improvement Program'. This is the little dialog that pops up when you run some new software for the first time and asks if you would be so kind as to allow anonymous information about how you use the application to be sent back to Microsoft. This can then be used to see what features customers use and to garner comments about usability.

The Art of Readable Code



By Dustin Boswell and Trevor Foucher, published byO'Reilly, ISBN: 978-0596802295

Reviewed by Alexander Demin

There are plenty of books about how to write code. So, I was quite skeptical when a friend of mine forwarded me a fragment of yet another one. Surprisingly, without long introduction it went straight to the point and coined the following in the first chapter:

- Code should be easy to understand.
- What makes code 'better'?
- The fundamental theorem of readability.
- Code should be written to minimize the time it would take for someone else to understand it.
- Is smaller always better?
- It is better to clean and precise that to be cute.

The style was precise and concrete. I felt it quite ambitious to cover 'the art' in less than two hundred pages, and decided to order the book to find out the approach.

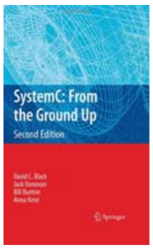
A few hours of reading on weekend turned out to be worthwhile. Though an experienced programmer will not find any too startling in the book, but this is a compact, concise and solid handout for more juniors programmers. Without too much theory, always based on real examples, the authors go through many key points of writing code: how to name variables, functions and classes, how to structure the code, how deal with the efficiency/readability trade off, how to comment, where to compromise and where to remain being the perfectionist. Again, it is all in less then two hundred pages. Plus they briefly touch on unit testing.

The authors not only tell you what is good and bad, they always show why through the examples by making 'regular' code better. At the end they put a real example of a class counting network traffic and returning a number of bytes transferred in the last hour and the last day.

They begin with a naive implementation and work through two more versions to find that sensitive balance between efficiency and readability. I think even experienced developers may find this example interesting to tackle.

To sum up, this book can fit perfectly onto your team book shelf and be used as a quick reference of howtos. Buying for yourself is perhaps of less benefit, because at home you'd probably prefer something more fundamental.

SystemC: From the Ground Up



By David C. Black, Jack Donovan, Bill Bunton, Anna Keist, published by Springer, ISBN: 978-0-387-69957-8

Reviewed by Paul Floyd

Recommended

I wouldn't normally submit reviews of books that are in the domain of microelectronics, but this is sufficiently 'C++' to warrant it. SystemC is intended for electronics system design, at a somewhat higher level than Verilog or VHDL. Whereas with Verilog or VHDL, you have no choice but to use tools like a simulator if you want to do anything with them, you can play with SystemC with just a C++ compiler. SystemC is implemented as a C++ library; all you need to do is to download the source (from www.accellera.org, registration and login required), build it and then you can write your code much as you would with any third party C++ library. The big advantage compared to other hardware description languages is that you can link existing functions and libraries to your models easily. The classic example would be something like an MPEG decoder. Basically the SystemC library gives you three things: an event-driven system, additional types and a framework for connecting components together.

On the whole, I was quite impressed with the level of the C++ presented. Many of the scientific or engineering C++ books I've read show a rudimentary knowledge of C++. Here we have pure virtual functions and fairly sophisticated use of templates. There's even C++-style variable initialization. The explanations of the simulation aspects are clear, like how signal assignment differs from ordinary assignment to variables in C++. There are some good tips on how to make simulations run faster.

Like most Springer books, this one is not cheap (90UKP at the time of writing). My biggest complaint is that some of the diagrams are a bit cheesy with jagged bitmap egdes, and the code examples use bold font for typenames, and for some reason sometimes the space is missing between the typenames and the identifiers.

Disclaimer: my employer does produce tools that include SystemC capabilities.

View From The Chair

Hubert Matthews
chair@accu.org

Software development: a game of people played with source code. It is interesting how software is usually thought of as intellectual property. For a lot of developers it has a strong emotional content, particularly if they've had to sweat to create it. Jerry Weinberg's notion of "egoless" programming seems far far away when people are wrestling with difficult problems. After all, it is those sorts of problems that are part of the attraction of programming. Some software is just plumbing that pushes data around; it is almost formulaic and with little satisfaction to be had other than from getting something working and finished. I suspect that ACCU members much prefer the thorny-problem type of software and that writing yet another end-of-month MIS report just doesn't do it for them - they invest a lot of themselves in what they do. I was reminded of this difference recently when working in Turkey. There, as in Japan I'm told, programming is just a stepping stone to becoming a manager; programming is something you learn at college and do as your first job afterwards but not as a career. The idea that anyone could have started programming before going to college and for fun was a totally strange notion to them, let alone that anyone as old and wizened as me would choose to continue programming beyond the age of thirty. Since it takes ten years at least to become good at programming these people will probably never learn to master their craft and subsequently will never experience the deep aesthetic satisfaction that skilled programming and design brings. How cultures affect views is forever fascinating and a culture that doesn't encourage people to engage and develop misses out on their commitment and their ability to achieve greater things over time.

One of the other aspects of the emotional side of software development is how we relate not to the code but to each other in teams. Humans form teams because we can achieve more together than we can as individuals. Nowhere is this more evident than in sport. I was reminded of this recently after watching the Oxford and Cambridge Boat Race. I have been involved in coaching rowing for thirty years and one thing it makes you acutely aware of is human motivation. Getting people voluntarily to put themselves in extreme physical pain on a regular basis for no tangible reward is a great (and humbling) learning experience. Duress of this magnitude focuses the spotlight strongly on team issues such as trust, loyalty and commitment, and without these a crew will most likely tear itself apart, act as a bunch of individuals and consequently underperform. However, when present, the bonds of trust and good fellowship run deep and are often the things that are remembered fondly in later years. In such an environment there is room for - and for peak performance it is necessary to have - differing opinions. These can be discussed openly and any differences ironed out and resolved without breaking these bonds. Strangely, doing so can actually strengthen the team spirit by overcoming hurdles together. One thing that is poisonous, however, is blame. Rowing, like most amateur sports, is a gift culture: people's status is based on what they give. Negativity is a sure way of stopping this dead in its tracks, as is disloyalty.

Voluntary organisations such as the ACCU are also gift cultures: we value people by what they contribute. If we descend into negativity and blame we defeat the whole purpose of coming together as an organisation. We break the bonds of trust and loyalty that make our community what it is. People will stop contributing and we will fall back to being individuals with no common purpose. It is so much easier to criticise than to do, to destroy rather than to create. We forget this at our peril.