The magazine of the ACCU

www.accu.org

{cvu}

Volume 24 • Issue 3 • July 2012 • £3

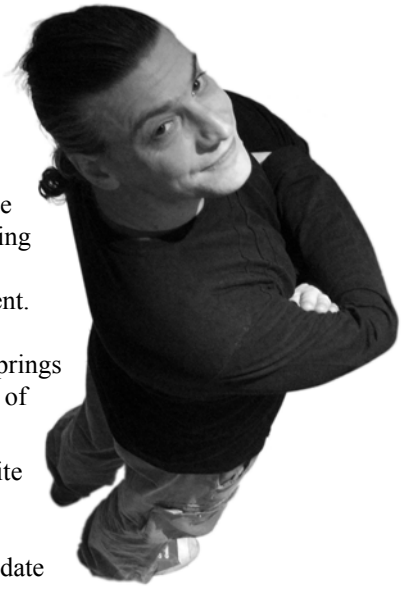
Features

Regulars

{cvu}
Volume 24 Issue 3
July 2012
ISSN 1354-3164
www.accu.org

Flesh on Bones

There's been quite a bit of activity on the mailing lists recently (well, recently as I write) about interviews and the whole recruitment process.



Working as a contractor means I get to take part in this little pantomime fairly often, sometimes two or three times in a year. Most often, of course, on the interviewee side of the table, but occasionally doing the interviewing.

For programming interviews, much is made of gauging the technical ability of a candidate with respect to actually being able to write code, and having more than a superficial understanding of a programming language and environment. It's easy – too easy – to become blind to all else when evaluating this level of competence. The expression that springs to mind is the one about being able to code one's way out of a paper bag.

I have done quite a few programming tests of both the 'write on a piece of paper' and the 'submit as homework by Tuesday' variety. I accept it can be interesting and enlightening to use this as a way of determining if a candidate has ever actually written any code, and I also accept that it seems that a large number of people who apply for programming jobs have not. As both interviewer and interviewee, though, I find the most enlightening thing is to discuss the relative merits of different practices and approaches to development problems. A short code example is sometimes the best vehicle for this debate but not always. Too often, in my experience, these little code tests are a vehicle for the interviewer to demonstrate their own prowess.

As an interviewer I often try to develop a philosophical discussion about approaches to unit testing, code management and delivery, version control, relative merits between different languages – even, or perhaps especially when the languages being discussed aren't even on the job spec. I find it rare that someone who engages meaningfully in such a conversation is actually a bad programmer.

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 24.4: 1st August 2012
C Vu 24.5: 1st October 2012

Overload 111: 1st September 2012
Overload 112: 1st November 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

Development Fuel: Software Testing in the Large

Seweryn Habdank-Wojewódzki and Adam Petersen have some advice for testing large systems.

As soon as a software project grows beyond the hands of a single individual, the challenges of communication and collaboration arise. We must ensure that the right features are developed, that the product works reliably as a whole and that features interact smoothly. And all that within certain time constraints. These aspects combined places testing at the heart of any large-scale software project.

This article grew out of a series of discussions around the role and practice of tests between its authors. It's an attempt to share the opinions and lessons learned with the community. Consider this article more a collection of ideas and tips on different levels than a comprehensive guide to software testing. There certainly is more to it.

Keeping knowledge in the tests

We humans are social creatures. Yes, even we programmers. To some extent we have evolved to communicate efficiently with each other. So why is it so hard to deliver that killer app the customer has in mind? Perhaps it's simply because the level of detail present in our average design discussion is way beyond what the past millenniums of evolution required. We've gone from sticks and stones to multi-cores and CPU caches. Yet we handle modern technology with basically the same biological prerequisites as our prehistoric ancestors.

Much can be said about the human memory. It's truly fascinating. But one thing it certainly isn't is *accurate*. It's also hard to back-up and duplicate. That's where documentation comes in on complex tasks such as software. Instead of struggling to maintain repetitive test procedures in our heads, we suggest relying on structured and automated test cases for recording knowledge. Done right, well-written test cases are an excellent communication tool that evolve together with the software during its whole life-cycle.

Large-scale software projects have challenges of their own. Bugs due to unexpected feature interactions are quite common. Ultimately, such bugs are a failure of communication. The complexity in such bugs is often significant. Not at least since the domain knowledge needed to track down the bug is often spread across different teams and individuals. Again, recording that domain knowledge in test cases makes the knowledge accessible.

Levels of test

It's beneficial to consider testing at all levels in a software project. Different levels allow us to capture different aspects and levels of details. The overall goal is to catch errors as early as possible, preferably on the lowest possible level in the testing chain. But our division is more than a technical solution. It's a communication tool. The tests build on each other towards the user level. The higher up we get, the more we can involve non-technical roles in the conversation.

Each level serves a distinct purpose:

1. **Unit tests** are driven by implicit design requirements. Unit tests are never directly mapped to formal requirements. This is the technically most challenging level. It's impossible to separate unit tests from design. Instead of fighting it, embrace it; unit tests are an excellent medium and opportunity for design. Unit tests are written solely by the developer responsible for a certain feature.

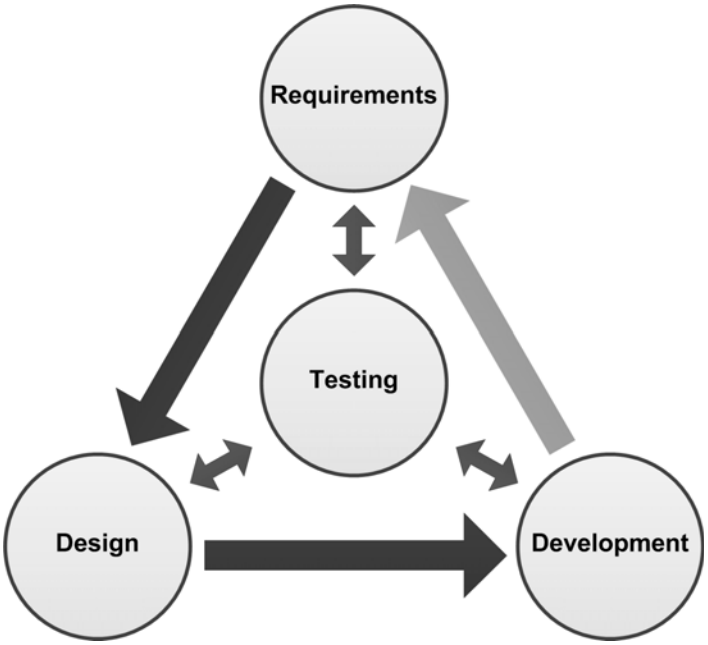


Figure 1

2. **Integrations tests** are where requirements and design meet. The purpose of integration tests is to find errors in interfaces and in the interaction between different units as early as possible. They are driven by use cases, but also by design knowledge.
3. **System tests** are the easiest ones to formulate. At least from a technical perspective. System tests are driven by requirements and user stories. We have found that most well-written suites are rather fine-grained. One requirement is typically tested by at least one test case. That is an important point to make; when something breaks, we want to know immediately what it was.

The big win: tap into testers' creativity by automating

Testing in Figure 1 (the software development cycle from a test-centric oint of view) refers to any kind of testing. Among them are exploratory and manual tests. These are the ones that could make a huge qualitative difference; it's under adverse conditions that the true quality of any product is exposed.

SEWERYN HABDANK-WOJEWÓDZKI

Seweryn Habdank-Wojewódzki specialises in high-performance distributed computing. He is also focused on the industrial quality standards of the code. In his leisure time he enjoys self-made art. Seweryn can be contacted at habdank@gmail.com

ADAM PETERSEN

Adam Petersen is a programmer and graduate student. His interests include Lisp, Erlang, parallel programming martial arts, music and modern history. He can be contacted at: adam@adampetersen.se

Thus the purpose of this level of testing is to try to break the software, exploiting its weaknesses, typically by trying to find unexpected scenarios. It requires a different mindset found in good testers; just like design, testing is a creative process. And if we manage to get a solid foundation by automating steps 1–3 above, we get the possibility to spend more time in this phase. As we see it, that's one of the big selling-points of automated tests.

The challenges of test automation

The relative success of a test automation project goes well behind any technical solutions; test automation raises questions about the roles in a project. It's all too easy to make a mental difference regarding the quality of the production code and the test code. It's a classic mistake. The test code will follow the product during its whole life cycle and the same aspects of quality and maintainability should apply here. That's why it's important to have developers responsible for developing these tools and frameworks. Perhaps even most test cases in close collaboration with the testers.

There's one caveat here though; far too many organizations aren't shaped to deal with cross-disciplinary tasks. We often find that although the developers have the skills to write competent test frameworks and tools, they're often not officially responsible for testing. In social psychology there's a well-known phenomenon known as diffusion of responsibility [1]. Simply put, a single individual is less likely to take responsibility for an action (or inaction) when others are present. The problem increases with group size and has been demonstrated in a range of spectacular experiments and fateful real-world events.

There's one caveat here though; far too many organizations aren't shaped to deal with cross-disciplinary tasks

The social game of large-scale software development is no exception. When an organization fails to adequately provide and assign responsibilities, we're often left with an unmaintainable mess of test scripts, simulators and utilities since the people developing them aren't responsible for them; they're not even the users. These factors combined prevent the original developers from gaining valuable feedback. At the end, the product suffers along with the organization.

Changing a large organization is probably one of the hardest tasks in our modern corporate world. We're better advised to accept and mitigate the problem within the given constraints. One simple approach is to put focus on mature test environments and/or frameworks. Either a custom self maintained framework or one off the shelf. A QA Manager should consider investing in a testing framework, to discipline and speed up testing. Especially if there already are existing test cases that shall be re-run over and over again. Such a task is usually quite boring and therefore error prone. Automating it minimizes the risks of errors due to human boredom. That's a double win.

System level testing

System level testing refers to requirements [2], user stories and use cases. Reading and analysing requirements, user stories and use cases is a vital part in the preparation of test cases. Use cases are very close to test cases. However their focus is more on describing how the user interacts with the system rather than specifying the input and expected results. That said, when there are good use cases and good test cases, they tend to be very close to each other.

Preparing test cases – the requirements link

With increasing automation the line between development and testing gets blurred; writing automated test cases is a development activity. But when developers maintain the frameworks, what's the role of the tester?

Well, let's climb the software hill and discuss requirements first. Requirements shall be treated and understood as generic versions of use cases. It is hard to write good requirements, but it is important to have them to keep an eye on all general aspects of the product. Now, on the highest level test cases are derived directly and in-directly from the requirements. That makes the test cases place holders for knowledge. It's the communicating role of the test cases. Well-written test cases can be used as the basis for communication around requirements and features. Increasingly, it becomes the role of the tester to communicate with Business Analysts [3] and Product Managers [4].

Once a certain requirement or user story has been clarified, that feedback goes into the test cases.

The formulation of test cases is done in close collaboration with the test specialists on the team. The tester is responsible for deciding what to test; the developer is responsible for how. In that context, there are two common problems with requirements; they get problematic when they're either too strict or too fuzzy. The following sections will explain the details and cures.

Avoiding too strict requirements

Some requirements are simply too strict, too detailed. Let's consider the following simple example. We want to write a calculator, so we write a requirement that our product shall fulfil the following: 2 + 2 = 4, 3 * 5 = 15, 9 / 3 = 3. How many such requirements shall we write? On this level of detail there will be lots of them (let's say infinite...). Writing test cases will immediately show that the requirements are too detailed. There is no generic statement capturing the abstraction behind, specifying what *really* shall be done. There are three examples on input and output. In a pathological case of testing we will write exactly three test cases (copy paste from requirements) and reduce the calculator to the look-up table that contains exactly those three rows with values and operations as above. It may be a trivial example but it expands to all computing.

Further, for scalability reasons it's important to limit the number of test cases with respect to their estimated static path count. One such technique is to introduce Equivalence Classes for input data by Equivalence Class Partitioning (ECP) [5]. That will help to limit number of tests for interface testing. ECP will also guide in the organization of the test cases by and dividing them in normal operation test cases, corner cases and error situations.

Test data based on the ECP technique makes up an excellent base for data-driven tests. Data-driven tests are another example on separating the mechanism (i.e. the common flow of operations) from the stimulus it operates on (i.e. the input data). Such a diversion scales well and expresses the general case clearer as well.

Cures for fuzzy requirements

Clearly too strict requirements pose a problem. On the other side of the spectrum we have fuzzy requirements. Consider a requirement like: 'during start everything shall be logged'. On the development team we might very well understand the gist in some concrete way, but our customer may have a completely different interpretation. Simply asking the customer: 'How will it be tested?' may go a long way towards converting that requirement to something like: 'During application start-up, it should be possible to any information to the logger. Where *any information* means: start of the main function of the application and all its plug-ins.'

How did our simple question to the customer helped us sort out the fuzziness in the requirement? First of all 'every' was transformed to 'any'. To get the conversation going we could ask the user if he/she is interested in *every* bit of information, like the spin of the electrons in CPU. Writing test cases or discussing them with the user often give us his perspective. Often, the user considers different information useful for different purposes. Consider our definition of 'any' information above. Here 'any'

for release 1.0 could imply logging the start of the main function and plug-ins. We see here that such a requirement does not limit the possible extensions for release 2.0.

The discussion also helped us in clarifying what 'logged' really meant. From a testing point of view we now see that test shall consider the presence of the logger. And later requirements may precisely define the term logger and what the logs looks like. Again requirements about the shape of the logs shall be verified by proper test cases and by keeping the customer in the loop. Preparing the test case may guide the whole development team towards a very precise definition of logs.

Consider another real-world example from a product that one of the authors was involved in. The requirements for that product specified that transactions must be used for all operations in the database. That's clearly not something the user cares about (unless we are developing an API for a database...). It's a design issue. The real requirements would be something related to persistent information in the context of multiple, concurrent users and leave the technical issues to the design without specifying a solution.

Done right, traceability is useful as a way to control the complexity, scale and progress of the development

One symptom of this problem is requirements that are hard to test. The example above ended up being verified by code inspection – hard to automate, and hard to change the implementation. Say we found something more useful than a relational database. As long as we provide the persistency needed, it would be perfectly fine for the end-user. But, such a design change would trigger a change in the requirements too.

Finally some words on Agile methodologies since they're commonplace these days. Agile approaches may help in test preparation as well as in defining the strategy, tools and writing test cases. The reason Agile methodologies may facilitate these aspects is indirect through the potentially improved communication within the project. But, the technical aspects remain to be solved independent of the actual methodology. Thus, all aspects of the software product have to be considered anyway from a test perspective; shipping, installation process, quality of documentation (which shall be specified in requirements as well) and so on.

Traceability

In safety-critical applications traceability is often a mandatory requirement in the regulatory process. We would like to stress that traceability is an important tool on any large-scale project. Done right, traceability is useful as a way to control the complexity, scale and progress of the development. By linking requirements to test cases we get an overview of the requirements coverage. Typically, each requirement is verified by one or more test cases. A requirement without test(s) is a warning flag; such a requirement is often useless, broken or simply too fuzzy.

From a practical perspective it's useful to have bi-directional links. Just like we should be able to trace a requirement to its test cases, the test cases should explain which requirement (or component thereof) it tests. Bi-directional traceability is of vital importance when preparing or generating test reports.

Such a link could be as simple as a comment or magical tag in each test case, it could be an entry in the test log, or the links could be maintained by one of the myriad of available tools for requirements tracing.

Design of test environments

Once we understand enough of the product to start sketching out designs we need to consider the test environments. As discussed earlier, we recommend testing on different complementary levels. With respect to the test environment, there may well be a certain overlap and synergies that allow parts to be shared and re-used across the different test levels. But once we start moving up from the solution domain of design (unit tests) towards the problem domain (system and acceptance tests), the interfaces change radically. For example, we may go from testing a programmatic API of one module with unit tests to a fully-fledged GUI for the end-user. Clearly, these different levels have radically different needs with respect to input stimulation, deployment and verification.

Test automation on GUI level

In large-scale projects automatic GUI tests are a necessity. The important thing is that the GUI automation is restricted to check the behaviour of the GUI itself. It's a common trap to try to test the underlying layers through the GUI (for example data access, business logic). Not only does it complicate the GUI tests and make the GUI design fragile to changes; it also makes it hard to inject errors in the software and simulate averse conditions.

However, there are valid cases for breaking this principle. One common case is when attempting to add automated tests to a legacy code base. No matter how well-designed the software is, there will be glitches with respect to test automation (e.g. lack of state inspection capabilities, tightly coupled layers, hidden interfaces, no possibility to stimulate the system, impossible to predictably inject errors). In this case, we've found it useful to record the existing behaviour as a suite of automated test cases. It may not capture every aspect of the software perfectly, but it's a valuable safety-net during re-design of the software.

The test cases used to get legacy code under test are usually not as well-factored as tests that evolve with the system during its development. The implication is that they tend to be more fragile and more inclined to change. The important point is to consider the tests as temporary in their nature; as the program under test becomes more testable, these initial tests should be removed or evolve into formal regression tests where each test cases captures one, specific responsibility of the system under test.

Integration defines error handling strategies

In large-scale software development one of the challenges is to ensure feature and interface compatibility between sub-systems and packages developed by different teams. It's of vital importance to get that feedback as early as possible, preferably on each committed code change. In this scope we need to design all tests to be sure that all possible connections are verified. The tests shall predict failures and test how one module will behave in case another other module fails. The reason is twofold.

First, it's in averse conditions that the real quality of any software is brutally exposed; we would be rich if given a penny for each Java stack trace we've seen in live systems on trains, airports, etc. Second, by focusing on inter-module failures we drive the development of an error handling strategy. And defining a common error handling policy is something that has to be done early on a multi-team software project. Error handling is classic example on cross-cutting functionality that cannot be considered locally.

Simulating the environment

Quite often we need to develop simulators and mock-ups as part of the test environment. Having or being able to have mock-ups will detect any lack of interfaces, especially when mock objects or modules has to be used instead of real ones. Further, simulators allow us to inject errors in the system that may be hard to provoke when using the real software modules.

Finally, a warning about mock objects based on hard-earned experience. With the increase in dynamic features in popular programming languages (reflection, etc) many teams tend to use a lot of mocks at the lower levels of test (unit and integration tests). That may be all well. Mocks may serve a purpose. The major problem we see is that mocks encourage interaction testing which tends to couple the test cases to a specific implementation. It's possible to avoid but any mock user should be aware of the potential problems.

Programming languages for testing

The different levels of tests introduced initially are pretty rough. Most projects will introduce more fine-grained levels. If we consider such more detailed layers of testing (e.g. acceptance testing, functional testing, production testing, unit testing) then except for unit testing, the most important part here is to separate the language used for testing from the development language. There are several reasons for this.

The development language is typically selected due to a range of constraints. These may be due to regulatory requirements in safety or medical domains, historical reasons, efficiency, or simply due to the availability of a certain technology on the target platform. In contrast, a testing language shall be as simple as possible. Further, by using different languages we enable cross-verification of the intent which may help in clarifying the details of the software under test. Developers responsible for supporting testing shall prepare high level routines that can be used by testers without harm for the tested software. It can be either commercial tools [6] or open sources [7].

When capturing test case we recommend using a formal language. In system or mission critical SW development there are formal processes built around standards like DO-178B and similar. In regular SW development using an automated testing framework forces developers to write test specifications in a dedicated high-level language. Most testing tools offer such support. This is important since formal language helps in the same way as normal source code. It can be verified, executed and is usually expressive in the test domain. If it is stored in plain text then comparison tools may help to check modifications and history. More advanced features are covered by Test Management tools.

TDD, unit tests and the missing link

A frequent discussion about unit tests concern their relationship to the requirements. Particularly in Test-Driven Development (TDD) [8] where the unit tests are used to drive the design of the software. With respect to TDD, The single most frequent question is: 'how do I know which tests to write?' It's an interesting question. The concept of TDD seems to trigger something in peoples mind; something that the design process perhaps isn't deterministic. It particularly interesting since we rarely hear the question 'how do I know what to program?' although it is exactly the same problem. As we answer something along the lines that design (as well as coding) always involves a certain amount of exploration and that TDD is just another tool for this exploration we get, probably with all rights, sceptical looks. The immediate follow-up question is: 'but what about the requirements?' Yes, what about them? It's clear that they guide the development but should the unit tests be traced to requirements?

Requirements describe the 'what' of software in the problem domain. And as we during the design move deeper and deeper into the solution domain, something dramatic happens. Our requirements explode. Robert L. Glass identifies requirements explosion as a fundamental fact of software development: 'there is an explosion of "derived requirements" [..] caused by the complexity of the solution process' [9]. How dramatic is this explosion? Glass continues: 'The list of these design requirements is often 50 times longer than the list of original requirements' [9]. It is requirements explosion that makes it unsuitable to map unit tests to requirements; in fact, many of the unit tests arise due to the 'derived requirements' that do not even exist in the problem space!

Avoid test dependencies on implementation details

Most mainstream languages have some concept of private data. These could be methods and members in message-passing OO languages. Even the languages that lack direct language support for private data (e.g. Python, JavaScript) tend to have established idioms and conventions to communicate the intent. In the presence of short-term goals and deadlines, it may very well be tempting to write tests against such private implementation details. Obviously, there's a deeper issue with it; most testers and developers understand that it's the wrong approach.

Before discussing the fallacies associated with exposed implementation details, let's consider the purpose of data hiding and abstraction. Why do we encapsulate our data and who are we protecting it from? Well, it turns out that most of the time we're protecting our implementations from ourselves. When we leak details in a design we make it harder to change. At some point we've probably all seen code bases where what we expected to be a localized change turned out to involve lots of minor changes rippling through the code base. Encapsulation is an investment into the future. It allows future maintainers to change the how of the software without affecting the what.

breaking the seal of encapsulation will have a negative impact on the maintainability and future life of the software

With that in mind, we see that the actual mechanisms aren't that important; whether a convention or a language concept, the important thing is to realize and express the appropriate level of abstraction in our everyday minor design decisions.

Tests are no different. Even here, breaking the seal of encapsulation will have a negative impact on the maintainability and future life of the software. Not only will the tests be fragile since a change in implementation details may break the tests. Even the tests themselves will be hard to evolve since they now concern themselves with the actual implementation which should be abstracted away.

That said, it may well exist cases where a piece of software simply isn't testable without relaying on and inspecting private data. Such a case is actually a valuable feedback since it often highlights a design flaw; if something is hard to test we may have a design problem. And that design problem may manifest itself in other usage contexts later. As the typical first user of a module, the test cases are the messenger and we better listen to him. Each case requires a separate analysis, but we've often found one of the following flaws as root cause:

1. Important state is not exposed – perhaps we shall think about some state of the module or class that shall be exposed in a kind of invariant way (e.g. by COW, const).
2. Class/Module is complicated with overly strong coupling.
3. The interface is too poor to write essential test cases.
4. A proper bridge (or in C++ pimpl) pattern is not used to really hide private details that shall not be visible. In this case it's simply a failure of the API to communicate by separating the public from the hidden parts.

Coping with feedback

As a tester starts to write test cases expected to be run in an automated way he will usually detect anomalies, asymmetric patterns and deviations in the code. Provided coding and testing are executed reasonably parallel in time, this is valuable feedback to the developer. On a well-functioning team, the following information would typically flow back to the designers of the code:

- Are there any missing interfaces? Or are there perhaps too many interfaces bloating the design?
- Is it supposed to work like this?
- Is the SW conceptually consistent?
- Are the differences between similar methods documented and clearly expressed?

Since the test cases typically are the first user of the software they are likely to run into other issues that have to be addressed earlier rather than becoming a maintenance cost. One prime example is the instantiation of individual software components and systems. The production and test code

tools which calculate code metrics point to areas that need improvement as well as how to test the code

may have different needs here. In some cases, the test code has to develop mechanisms for its own unique needs, for example factory objects to instantiate the components under test. In that case, the tester will immediately detect flaws and complicated dependency chains.

Automatic test cases has another positive influence on the software design. When we want to automate efficiently, we will have to separate different responsibilities. This split is typically done based on layers where each layer takes us one step further towards the user domain. Examples on such layers include DB access, communication, business logic and GUI. Another typical example involves presenting different usage views, for example providing both a GUI and a CLI.

Filling data into classes or data containers

This topic brings many important design decision under consideration. Factories but in general construction of the SW is always tricky in terms of striking a balance between flexibility and safety. Let's consider a simple example class, `Authentication`. Let's assume the class contains two fields: `login` and `password`. If we will start to write test cases to check access using that class we could arrive at a table with the following test data: Authentication = {{A,B},{C,D},{E,F},{G,H},{I,J}}. If the class has two getters (`login`, `password`) and two setters (similar ones), it is very likely that we do not need to separate login and password. Changing login usually forces us to change password too. What about having two getters and one setter that takes two arguments and one constructor with two arguments? Seems to be good simplification. It means that by preparing the tests, we arrived at suggested improvements in the design of the class.

Gain feedback from code metrics

When testing against formal requirements the initial scope is rather fixed. By tracing the requirements to test cases we know the scope and extent of testing necessary. A more subjective weighting is needed on lower levels of test. Since unit tests (as discussed earlier) are written against implicit design requirements there's no clear test scope. How many tests shall we write?

Like so many other quality related tools, there's a point of diminishing return with unit tests. Even if we cover every corner of the code base with tests there's absolutely no guarantee that we get it right. There are just too many factors, too many possible ways different modules can interact with each other and too many ways the tests themselves may be broken. Instead, we recommend basing the decision on code metrics.

Calculating code metrics, in particular cyclomatic complexity and estimated static path count [10], may help us answer the question for a particular case. Code Complexity shows the minimal number of test actions or test cases that shall be considered. Estimate Static Path Count on the other hand shows a kind of maximal number (true maximal number is quite often infinity). It means that tools which calculate code metrics point to areas that need improvement as well as how to test the code. Basically, code metrics highlight parts of the code base that might be particularly tricky and may require extra attention. Note that these aspects are a good subject for automation. Automatic tests can be checked against coverage metrics and the code can be automatically checked with respect to cyclomatic complexity. Just don't forget to run the metrics on the test code itself; after all, it's going to evolve and live with the system too.

Summary

Test automation is a challenge. Automating software testing requires a project to focus on all areas of the development, from the high-level requirements down to the design of individual modules. Yet, technical solutions aren't enough; successful test automation requires a working communication and structured collaboration between a range of different roles on the project. This article has touched all those areas. While there's much more to write on the subject, we hope our brief coverage may serve as a starting-point and guide on your test automation tasks. ■

References and reading

[1] http://en.wikipedia.org/wiki/Diffusion_of_responsibility

[2] Requirements, http://en.wikipedia.org/wiki/Requirement

[3] Allan Kelly, 'On Management: The Business Analyst's Role', http://accu.org/index.php/journals/1559

[4] Allan Kelly, 'On Management: Product Managers', http://accu.org/index.php/journals/1552

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

[6] List of notable test management tools, http://en.wikipedia.org/wiki/Test_management_tools#List_of_notable_test_management_tools

[7] http://www.opensourcetesting.org/testmgt.php

[8] http://en.wikipedia.org/wiki/Test-driven_development

[9] Robert L. Glass, *Facts and Fallacies of Software Engineering*

[10] Krusko Armin, 'Complexity Analysis of Real Time Software – Using Software Complexity Metrics to Improve the Quality of Real Time Software', http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2004/rapporter04/krusko_armin_04032.pdf

Adam Petersen, 'The Roots of TDD', http://www.adampetersen.se/articles/designintdd.htm

'Requirements Analysis', http://en.wikipedia.org/wiki/Requirements_analysis

Metaprogramming Plus: The Flexibility Enhancements

Nick Sabalausky writes more no-compromise code by metaprogramming in D.

Picking up right where we left off in part 1, you may recall Flexibility was concerned that the metaprogramming approach seemed to prevent complex configurability. He didn't think he could use complex logic to decide what types of Gizmos needed to be made and how many. The problem is that Gizmo's settings are specified at compile-time, but the logic to determine the configuration may need to happen at runtime. Dr. Metaprogramming knew that could be worked around and promised to show various methods of handing this.

These methods will be demonstrated by making two basic changes to the existing metaprogramming example:

1. Currently, there are 1-port, 2-port and 5-port Gizmos. The 5-port ones will no longer be hardcoded as 5-port. The number of ports on these larger Gizmos will now be configurable via **`bigPorts`**.
2. Only 2,000 spinnable 2-port Gizmos will be made instead of 10,000. But 8,000 extra Gimos will be created. The type of these extra Gizmos will be configurable via **`extrasNumPorts`** and **`extrasIsSpinnable`**.

Naturally, if you want to compare the time and memory usage with all the previous versions, then these values should be set to **`bigPorts = 5`**, **`extrasNumPorts = 2`** and **`extrasIsSpinnable = true`**.

Note that none of these require any changes to the Gizmo type itself. Only the code in **`UltraGiz`** and **`main()`** is affected. That is to say, the only changes are in setting up and using the same Gizmo types that we've already written.

Method #1: Compile-time function execution

Frequently abbreviated as CTFE, this method can't be done in all languages. And in a language that does support it (like D) it can be the least powerful method. But it's the simplest and easiest, and is perfectly sufficient in many situations.

All that needs to be done is assign the return value of a function to a compile-time value. The compiler will execute the function itself (if it can) instead of waiting until runtime. Simple. This version is is shown in Listing 1 (taken from `ex6_meta_flex1_ctfe.d`).

Method #2: Compiling at runtime

On the downside, this method takes extra time (potentially very noticeable) whenever a setting needs to be changed. That may or may not be a problem depending on the nature of the program and the setting. Additionally, you'll need to distribute your configurable source code along with your program. Finally, this method requires either:

1. The user has the appropriate compiler set up.

2. You package the compiler along with your application.

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

Source code

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

Listing 1

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

  static int generateBigPorts()
  {
    // Big fancy computation to determine
    // number of ports
    int num=0;
    for(int i=0; i<10; i++)
    {
      if(i >= 5)
        num++;
    }
    return num; // Ultimately, the result is 5
  }

  static int generateExtrasNumPorts(int input)
  {
    return input - 3;
  }

  static
    bool generateExtrasIsSpinnable(int input=9)
  {
    if(input == 0)
      return false;
    return !generateExtrasIsSpinnable(input-1);
  }
  static immutable bigPort = generateBigPorts();
  static immutable extrasNumPorts =
     generateExtrasNumPorts(bigPort);
  static immutable extrasIsSpinnable =
     generateExtrasIsSpinnable();
```

Listing 1 (cont'd)

```
  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;

    // Use extrasNumPorts and extrasIsSpinnable
    // so 8,000 more of these will be made down
    // below.
    gizmos!(2, true ).length =  2_000;
    gizmos!(bigPort, false).length = 5_000;
    gizmos!(bigPort, true ).length = 5_000;

    // Add in the extra Gizmos
    gizmos!(extrasNumPorts,
            extrasIsSpinnable).length += 8_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!(bigPort, false))
        useGizmo(gizmo);
      foreach(ref gizmo; gizmos!(bigPort, true ))
        useGizmo(gizmo);
    }
    writeln(stopWatch.peek.msecs, "ms");
  }
}
void main()
{
  UltraGiz ultra;
  ultra.run();
  // Compile time error: A portless Gizmo is
  // useless!
  //auto g = Gizmo!(0, true);
}
```

Note that rules out using this method for most embedded targets unless you set up a compilation server. But that comes with its own set of concerns.

So ok, maybe this doesn't sound very good so far. But there are fairly major benefits: This method is extremely powerful, viable for a wide variety of languages, and only requires very simple changes to the code being configured. Additionally, the compiler requirement may not be as much of a problem as it may seem if you have permission to redistribute the compiler, or if you're targeting Linux (which generally has pretty good package management and dependency tools), or if your program is only intended for private use.

The trick here is to generate a small amount of source code at runtime, recompile your program, and then run the result.

For simplicity, this example will use a separate **frontend** program that will configure, compile and run the real example program. But you could also have it all in one program: After your program issues the command to recompile itself, it would then relaunch itself (possibly saving and restoring any important state in the process) much like auto-updating programs that download and launch newer versions of themselves would

Listing 2

```
import std.conv;
import std.file;
import std.process;
import std.stdio;

void main(string[] args)
{
  immutable configFile =
     "ex6_meta_flex2_config.d";
  immutable mainProgram =
     "ex6_meta_flex2_compilingAtRuntime";
  immutable mainProgramSrc =
     "ex6_meta_flex2_compilingAtRuntime.d";
  version(Windows)
    immutable exeSuffix = ".exe";
  else
    immutable exeSuffix = "";

  // Number of ports on each of the many-port
  // Gizmos. Normally 5
  int bigPort;
  // 8,000 extra Gizmos will be created with
  // this many ports and this spinnability.
  // Normally 2-port spinnable
  int  extrasNumPorts;
  bool extrasIsSpinnable;
  try
  {
    bigPort           = to!int (args[1]);
    extrasNumPorts    = to!int (args[2]);
    extrasIsSpinnable = to!bool(args[3]);
  }
  catch(Throwable e)
  {
    writeln("Usage:");
    writeln("  ex6_meta_flex2_frontend "~
       "{bigPort} {extrasNumPorts}
        {extrasIsSpinnable}");
    writeln("Example: ex6_meta_flex2_frontend
       5 2 true");
    return;
  }

  // This is the content of the
  // "ex6_meta_flex2_config.d" file to be
  // generated.
  auto configContent = `
    immutable conf_bigPort =
       `~to!string(bigPort)~`;
    immutable conf_extrasNumPorts =
       `~to!string(extrasNumPorts)~`;
    immutable conf_extrasIsSpinnable =
       `~to!string(extrasIsSpinnable)~`;
  `;

  // Load old configuration
  writefln("Checking  \t%s...", configFile);
  string oldContent;
  if(exists(configFile))
    oldContent =
       cast(string)std.file.read(configFile);

  // Did the configuration change?
  bool configChanged = false;
  if(configContent != oldContent)
  {
    writefln("Saving  \t%s...", configFile);
    std.file.write(configFile, configContent);
    configChanged = true;
  }
```

Listing 2 (cont'd)

```
  // Need to recompile?
  if(configChanged ||
     !exists(mainProgram~exeSuffix))
  {
    writefln("Compiling  \t%s...",
       mainProgramSrc);
    system("dmd "~mainProgramSrc~" -release
       -inline -O -J.");
  }

  // Run the main program
  writefln("Running  \t%s...", mainProgram);
  version(Windows)
    system(mainProgram);
  else
    system("./"~mainProgram);
}

//From ex6_meta_flex2_compilingAtRuntime.d,
// the main program:
void main()
{
  mixin(import("ex6_meta_flex2_config.d"));
  UltraGiz!(conf_bigPort, conf_extrasNumPorts,
            conf_extrasIsSpinnable) ultra;
  ultra.run();

  // Compile time error: A portless Gizmo
  // is useless!
  //auto g = Gizmo!(0, true);
}
```

do. Or, you could keep the configurable routines in a dynamic library, unload the dynamic library, recompile it, and then reload it. See Listing 2, which is taken from `ex6_meta_flex2_frontend.d`, the frontend program.

The actual definition of the **`UltraGiz`** is exactly the same as in method #1, but without all the **`generate*()`** functions, and the signature changed from:

```
struct UltraGiz
```

to:

```
struct UltraGiz(int bigPort, int extrasNumPorts,
                bool extrasIsSpinnable)
```

Method #3: Convert a runtime value to compile-time

Yes, you read that right. Though it may sound bizarre, as if it would require time-travel, it is possible to convert a runtime value to compile-time. It does have some restrictions:

1. The runtime value can't cause anything to happen differently at compile-time. Which is to be expected, since runtime occurs after compile-time. But the runtime value can 'pass-through' the compile-time code paths and result in other runtime effects.
2. Every possible value that the variable might hold must be individually handled.

What essentially happens is you take all the compile-time code paths you may want to trigger at runtime, and you trigger all of them at compile-time. Each one of them will produce a result that can be accessed at runtime. Then, at runtime, you just 'choose your effect'.

If that sounds confusing, don't worry. It's really much more straightforward than it sounds. Listing 3 shows a simple example, taken from `example_runtimeToCompileTime.d`.

Of course, given the repetition in there, metaprogramming can be used to automatically generate the code to handle large numbers of possible values. Or even the entire range of certain types, such as **`enum`**, **`bool`**, **`byte`** or maybe even a 16-bit value. A 32-bit value would be unrealistic

Listing 3

```
import std.conv;
import std.stdio;

// Remember, this is a completely different type
// for every value of compileTimeValue.
class Foo(int compileTimeValue)
{
  static immutable theCompileTimeValue =
     compileTimeValue;
  static int count = 0;
  this()
  {
    count++;
  }
  static void display()
  {
    writefln("Foo!(%s).count == %s",
             theCompileTimeValue, count);
  }
}

void main(string[] args)
{
  foreach(arg; args[1..$])
  {
    int runtimeValue = to!int(arg);
    // Dispatch runtime value to compile-time
    switch(runtimeValue)
    {
      // Note:
      // case {runtime value}:
      // new Foo!{equivalent compile
      // time value}();
      case  0: new Foo!0();  break;
      case  1: new Foo!1();  break;
      case  2: new Foo!2();  break;
      case  3: new Foo!3();  break;
      case 10: new Foo!10(); break;
      case 99: new Foo!99(); break;
      default:
        throw new
           Exception(text("Value ",runtimeValue,"
                     not supported."));
      }
  }
  Foo!( 0).display();
  Foo!( 1).display();
  Foo!( 2).display();
  Foo!( 3).display();
  Foo!(10).display();
  Foo!(99).display();
}
```

Listing 4

```
struct UltraGiz
{
  [Omitting other members for brevity. They're
   the same as before.]
  // Note this is templated
  void addGizmosTo(int numPorts,
     bool isSpinnable)(int numGizmos)
  {
    gizmos!(numPorts,
            isSpinnable).length += numGizmos;
  }
  void addGizmos(int numPorts, bool isSpinnable,
                 int numGizmos)
  {
    // Dispatch to correct version of
    // addGizmosTo.
```

Listing 4 (cont'd)

```
    // Effectively converts a runtime value
    // to compile-time.
    if(numPorts == 1)
    {
      if(isSpinnable)
        addGizmosTo!(1, true )(numGizmos);
      else
        addGizmosTo!(1, false)(numGizmos);
    }
    else if(numPorts == 2)
    {
      if(isSpinnable)
        addGizmosTo!(2, true )(numGizmos);
      else
        addGizmosTo!(2, false)(numGizmos);
    }
    [Same code repeated here for values 3, 5
     and 10.]
    else
      throw new Exception(to!string(numPorts)~"
         -port Gizmo not supported.");
  }
  void run(int bigPort)(int extrasNumPorts,
                        bool extrasIsSpinnable)
  {
    StopWatch stopWatch;
    stopWatch.start();
    // Create gizmos
    gizmos!(1, false).length = 10_000;
    gizmos!(1, true ).length = 10_000;
    gizmos!(2, false).length = 10_000;
    // Use the commandline parameters
    // extrasNumPorts and extrasIsSpinnable
    // so 8,000 more of these will be made below.
    gizmos!(2, true ).length = 2_000;
    gizmos!(bigPort, false).length = 5_000;
    gizmos!(bigPort, true ).length = 5_000;
    // Add in the extra Gizmos
    addGizmos(extrasNumPorts,
              extrasIsSpinnable, 8_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!(bigPort, false))
        useGizmo(gizmo);
      foreach(ref gizmo; gizmos!(bigPort, true ))
        useGizmo(gizmo);
    }
    writeln(stopWatch.peek.msecs, "ms");
  }
}
void main(string[] args)
{
  // Number of ports on each of the many-port
  // Gizmos. Normally 5
  int bigPort;
  // 8,000 extra Gizmos will be created with
  // this many ports and this spinnability.
  // Normally 2-port spinnable
  int  extrasNumPorts;
  bool extrasIsSpinnable;
```

Listing 4 (cont'd)

```
  try
  {
    bigPort           = to!int (args[1]);
    extrasNumPorts    = to!int (args[2]);
    extrasIsSpinnable = to!bool(args[3]);

    if(bigPort != 3 && bigPort != 5 &&
       bigPort != 10)
      throw new
        Exception("Invalid choice for bigPort");
  }

  catch(Throwable e)
  {
    writeln("Usage:");
    writeln
      ("  ex6_meta_flex3_runtimeToCompileTime1 "~
      "{bigPort} {extrasNumPorts}
      {extrasIsSpinnable}");
    writeln("bigPort must be 3, 5 or 10");
    writeln("Example:
      ex6_meta_flex3_runtimeToCompileTime1
      5 2 true");
    return;
  }

  UltraGiz ultra;
  // Dispatch to correct version of UltraGiz.run.
  // Effectively converts a runtime value to
  // compile-time.
  if(bigPort == 3)
    ultra.run!3(extrasNumPorts,
                extrasIsSpinnable);
  else if(bigPort == 5)
    ultra.run!5(extrasNumPorts,
                extrasIsSpinnable);
  else if(bigPort == 10)
    ultra.run!10(extrasNumPorts,
                 extrasIsSpinnable);

  // Compile time error: A portless Gizmo
  // is useless!
  //auto g = Gizmo!(0, true);
}
```

on modern hardware, though. And arbitrary strings would be out of the question unless you limited them to a predetermined set of strings, or to a couple of bytes in length (or more if you limited the allowable characters). But even with these limitations, this can still be a useful technique.

In any case, the fact remains: With certain restrictions, it is possible to convert a runtime value into a compile-time value. Listing 4 (taken from `ex6_meta_flex3_runtimeToCompileTime1.d`) shows how it can be applied to our Gizmo example.

That will work, but there are two potential problems with it.

The first problem is that it involves extra runtime code. That could cut into, or possibly even eliminate, the efficiency savings from metaprogramming. However, the extra runtime code is only run once when setting up the Gizmos, not while the Gizmos are actually being used. So as long as the Gizmo usage is enough to overshadow the extra overhead, it should still be worth it.

The second problem is that the **`addGizmos()`** function is an incredibly repetitive mess of copy-pasted code. It's a total violation of DRY: Don't Repeat Yourself. Maintaining that function would be very error-prone. Fortunately, that's easily fixed with a preprocessor, macros, or in D's case, a compile-time **`foreach`**; see Listing 5, which is taken from `ex6_meta_flex3_runtimeToCompileTime2.d`.

Listing 5

```
void addGizmos(int numPorts, bool isSpinnable,
               int numGizmos)
{
  // Dispatch to correct version of addGizmosTo.
  // Effectively converts a runtime value to
  // compile-time.

  // A 'foreach' over a TypeTuple is unrolled at
  // *compile time*
  foreach(np; TypeTuple!(1, 2, 3, 5, 10))
  if(numPorts == np)
  {
    if(isSpinnable)
       addGizmosTo!(np, true)(numGizmos);
    else
       addGizmosTo!(np, false)(numGizmos);
    return;
  }

  throw new Exception(to!string(numPorts)~"
     -port Gizmo not supported.");
}
```

Method #4: Dynamic fallback

Just like the town elder who made the handcrafted version, we can fall back on a dynamic version that uses runtime options instead of compile-time options.

This is easier and more flexible than the previous method. In fact, method #1, compile-time function execution, is probably the only method easier than this, but this is more powerful and supported by more languages. So this is a pretty good option.

However, the downside is this would naturally be the least efficient of all the methods, since some of the Gizmos would forgo the metaprogramming benefits. But as long as you don't need runtime configurability for all your Gizmos, then you can still get a net savings over the original non-metaprogramming version.

To do this, we'll use the same metaprogramming Gizmo we've been using for all the other methods in this section. But we'll also add a **`DynamicGizmo`** which is identical to the original Gizmo in `ex1_original.d`, just with a different name. The **`main()`** function is trivially similar to method #3 above, so I won't show it here. Check the online code listings for this article if you're interested. Listing 6 is the short version, from `ex6_meta_flex4_dynamicFallback1.d`.

The original Gizmo with the runtime options, i.e. **`DynamicGizmo`**, is used where necessary, while the more common cases are optimized with metaprogramming techniques. Not a bad compromise.

As you can see if you look at the full online code listing for `ex6_meta_flex4_dynamicFallback1.d`, I opted to make a completely separate definition for the dynamic version of **`Gizmo`**; that is, the **`DynamicGizmo`**. It would have also been possible to use a single definition for both the metaprogramming **`Gizmo`** and the **`DynamicGizmo`**. To do that, you'd just need to add another compile-time parameter, say **`bool dynamicGizmo`**, to go along with **`numPorts`** and **`isSpinnable`**. Doing so would probably be a good idea if only part of your struct is affected by the change from runtime options to compile-time options. But with **`Gizmo`**, the metaprogramming version converted practically everything to compile-time options, so in this case it was a little cleaner to just leave **`DynamicGizmo`** defined separately.

One other notable change I made was to the gizmos template (ie, the arrays that had been named **`gizmosA, gizmosB`**, etc. in the earlier handcrafted version). In all the other metaprogramming versions, gizmos had been templated on number of ports and spinnability. That worked fine, but now we have **`DynamicGizmo`** which doesn't really fit into that. So now gizmos is templated on the Gizmo's type so the dynamic Gizmos can be accessed

Listing 6

```
struct UltraGiz
{
  template gizmos(T)
  {
    T[] gizmos;
  }

  // Shortcut for non-dynamic gizmos, so we can
  // still say:
  // gizmos!(2, true)
  // instead of needing to use the more verbose:
  // gizmos!( Gizmos!(2, true) )
  template gizmos(int numPorts, bool isSpinnable)
  {
    alias gizmos!( Gizmo!(numPorts,
                   isSpinnable) ) gizmos;
  }
  int numTimesUsedSpinny;
  int numTimesUsedTwoPort;

  [Same useGizmo() as before, omitted for
   brevity]

  void run(int bigPort, int extrasNumPorts,
           bool extrasIsSpinnable)
  {
    StopWatch stopWatch;
    stopWatch.start();

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

    // Use the commandline parameters
    // extrasNumPorts and extrasIsSpinnable
    // so 8,000 more of these will be made down
    // below as dynamic gizmos.
    gizmos!(2, true ).length = 2_000;

    gizmos!(DynamicGizmo).length = 18_000;
    foreach(i;       0..5_000)
      gizmos!(DynamicGizmo)[i] =
        DynamicGizmo(bigPort, false);

    foreach(i;  5_000..10_000)
      gizmos!(DynamicGizmo)[i] =
        DynamicGizmo(bigPort, true);

    foreach(i; 10_000..18_000)
      gizmos!(DynamicGizmo)[i] =
        DynamicGizmo(extrasNumPorts,
                     extrasIsSpinnable);
    // 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!DynamicGizmo)
        useGizmo(gizmo);
    }
    writeln(stopWatch.peek.msecs, "ms");
  }
}
```

Listing 7

```
// Use gizmos
foreach(i; 0..10_000)
{
  // Think of this as an array of types:
  alias TypeTuple!(
    Gizmo!(1, false),
    Gizmo!(1, true ),
    Gizmo!(2, false),
    Gizmo!(2, true ),
    DynamicGizmo,
  ) AllGizmoTypes;

  foreach(T; AllGizmoTypes)
  foreach(ref gizmo; gizmos!T)
    useGizmo(gizmo);
}
```

with `gizmos!(DynamicGizmo)`. Unfortunately, that also means the nice simple:

```
gizmos!(2, true)
```

becomes the ugly:

```
gizmos!( Gizmos!(2, true) )
```

So I created an overload of the gizmos template which maps the nice simple old syntax to the new one.

As an extra benefit, templating gizmos on type makes it easy to clean up all those repetitive foreach statements in `UltraGiz.run()` (Listing 7).

The last remaining elephant in the room: Runtime conversion

Don't worry, I'm not really going to introduce an elephant into the story...

We've already seen various methods of configuring a compile-time option at runtime. But that was limited to creating a Gizmo. Once created, a Gizmo was stuck with those settings. So what if the number of ports and spinnability of an existing Gizmo needs to change? If only a few Gizmos need to be able to change, we can just make those few out of the `DynamicGizmo` from earlier with only trivial modifications. But suppose all the Gizmos need to be changeable: Now how do we change the settings of an existing Gizmo without giving up the metaprogramming benefits?

In our original example way back in part 1, it would have been trivial to modify the Gizmo code so you could change a Gizmo's number of ports and spinnability after it was created. They were already runtime values. But our metaprogramming examples have taken advantage of the fact that such values were fixed when a Gizmo is created. In fact, that's the fundamental key of the metaprogramming versions: 'Turn your runtime options into compile-time options.'

Listing 8

```
// This is a member of the Gizmo type:
TOut convertTo(TOut)()
{
  TOut newGizmo;

  static if(isSpinnable && TOut.isSpinnable)
    newGizmo.spinCount = this.spinCount;

  int portsToCopy = min(newGizmo.numPorts,
                        this.numPorts);
  newGizmo.ports[0..portsToCopy] =
     this.ports[0..portsToCopy];

  // If there were any other data, we'd copy it
  // to the newGizmo here

  return newGizmo;
}
```

Listing 9

```
// Use gizmos
foreach(i; 0..10_000)
{
  // Modify some gimos:
  // Save a 1-port non-spinny
  auto old1PortNoSpin = gizmos!(1, false)[i];

  // Convert a 2-port spinny to 1-port non-spinny
  gizmos!(1, false)[i] =
    gizmos!(2, true)[i].convertTo!( Gizmo!(1,
       false) )();

  // Convert a 5-port spinny to 2-port spinny
  gizmos!(2, true)[i] =
    gizmos!(5, true)[i%2].convertTo!( Gizmo!(2,
                                      true) )();

  // Convert the old 1-port non-spinny to
  // 5-port spinny
  gizmos!(5, true)[i%2] =
    old1PortNoSpin.convertTo!( Gizmo!(5,
                               true) )();

  // Use gizmos as usual
  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);
}
```

If one of these settings needs to change frequently during a program's run, then naturally it must be a runtime option. Turning it into a compile-time value won't improve its efficiency. That's because the 'runtime to compile-time' trick works by taking advantage of the fact that certain values don't really need to change. So unfortunately, if a setting needs to change frequently, it must remain a runtime value. The only way to optimize it out is to eliminate the feature entirely.

However, if the settings only need to change occasionally, then we're still in business.

'What? Have you completely lost it? Configuring a compile-time option at runtime was crazy enough. And now you intend to change a compile-time option at runtime?! They're immutable!'

It not so strange, really. In fact, functional programming experts probably already know where I'm going with this...

Listen carefully: We converted the runtime values to compile-time ones by encoding them as part of the type, right? So to change the compile-time value, all we have to do is convert the variable to a different type. Just like converting an integer to a string. Easy. Except this is even easier because the types we're converting are fundamentally very similar.

Naturally, this does take extra processing. Likely much more than just simply changing an ordinary runtime variable. That's why I made a big deal about how frequently you need the value to change. If it changes a lot, you'll just slow things down from all the converting. But as long as it doesn't change enough to use up the efficiency savings from metaprogramming, you'll still end up ahead.

I'll demonstrate this by shuffling around a few Gizmos on every iteration. You'll notice that I'm always keeping the same number of each type of Gizmo, but that's not a real limitation: That's only so we can still compare benchmarks. It would be trivial to actually change a Gizmo without

keeping everything balanced. All you'd have to do is remove the old Gizmo from the array for the old type, and append the newly converted one to the array for the new type. Of course, if you were to do this, you'd probably want to use a linked list or a stack instead of an array, since arrays have poor performance with such a usage pattern.

For simplicity, I won't use any of the 'Flexibility Enhancements' covered in the previous section. But the techniques can certainly still be combined.

This time around, the Gizmo is exactly the same as the original metaprogramming version from part 1, Listing 6 (`ex4_metaprogramming.d`), but with one member function added. See Listing 8.

The **`UltraGiz`** is also nearly identical to the original metaprogramming example. Except now we swap some Gizmos around when using them in **`UltraGiz`**'s **`run()`** in Listing 9.

Conversion downsides and possible fixes

There are a couple downsides to the runtime conversions above. But, they can be dealt with.

Hold it, Gizmo! Quit squirming around!

One issue with converting a Gizmo is that doing so puts it in a different memory location. This means that if anything was keeping a reference to the Gizmo, after conversion the reference is no longer pointing to the newly converted Gizmo. It's still referring to the old location.

This may not always be an issue. But if it does matter, you can solve it by using a tagged union. Something like Listing 10 (which is taken from `example_anyGizmo.d`).

As you see in **`main()`** in Listing 10, before you use an **`AnyGizmo`**, you first check the type via **`currentType`** and then use the appropriate member of **`gizmoUnion`**.

Alternatively, D's standard library offers an **`Algebraic`** type [1], which is the same thing, but safer and easier:.

```
import std.variant;
alias Algebraic!(
  Gizmo!(1, false),
  Gizmo!(1, true ),
  Gizmo!(2, false),
  Gizmo!(2, true ),
  Gizmo!(5, false),
  Gizmo!(5, true ),
) AnyGizmo;
```

Either way, this approach does have a few gotchas:

First, you have the runtime cost of checking the type before using the Gizmo. But this may be minimal, since you won't usually need to check the type on every single member access, only when it might have changed.

Second, since a union must be the size of its largest member, you won't get as much space savings when using an **`AnyGizmo`**. But, if the mere existence of one of your optional features requires extra time-consuming processing, you can at least save time because any Gizmos that forgo that feature won't incur the extra cost. Plus, this issue will only affect an **`AnyGizmo`**, not any other Gizmos you may have in use.

The final gotcha is that this makes converting a Gizmo more costly since the newly converted Gizmo needs to be copied back to the memory location of the original Gizmo. But if the Gizmo only needs to be converted occasionally, this shouldn't be a problem. If it is a problem, though, it may be possible to do the conversion in-place in memory.

Hey data, get back here!

The other main issue with converting Gizmos is that some conversions can lose data. Suppose we convert a spinny Gizmo to a non-spinny and back again. The non-spinny Gizmos don't have a **`spinCount`**, so our newly re-spinified Gizmo has lost its original **`spinCount`** data. It's back at zero again. Same thing with the ports: Convert a 5-port to a 2-port and back again and you've lost the **`numZaps`** from the last three ports.

Listing 10

```
import std.stdio;

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;
  // The rest here...
}

struct AnyGizmo
{
  TypeInfo currentType;
  GizmoUnion gizmoUnion;

  union GizmoUnion
  {
    Gizmo!(1, false) gizmo1NS;
    Gizmo!(1, true ) gizmo1S;
    Gizmo!(2, false) gizmo2NS;
    Gizmo!(2, true ) gizmo2S;
    Gizmo!(5, false) gizmo5NS;
    Gizmo!(5, true ) gizmo5S;
  }

  void set(T)(T value)
  {
    static if(is(T==Gizmo!(1, false)))
      gizmoUnion.gizmo1NS = value;
    else static if(is(T==Gizmo!(2, true)))
      gizmoUnion.gizmo2S = value;
    [etc...]

    currentType = typeid(T);
  }
}

void useGizmo(T)(T gizmo)
{
  writeln("Using a gizmo:");
  writeln("  ports=", gizmo.numPorts);
  writeln("  isSpinnable=", gizmo.isSpinnable);
}

void main()
{
  AnyGizmo anyGizmo;
  anyGizmo.set( Gizmo!(2, true)() );

  if( anyGizmo.currentType ==
      typeid(Gizmo!(1, false)) )
    useGizmo( anyGizmo.gizmoUnion.gizmo1NS );

  else if( anyGizmo.currentType ==
      typeid(Gizmo!(2, true)) )
    useGizmo( anyGizmo.gizmoUnion.gizmo2S );

  [etc...]

  else
    throw new Exception("Unexpected type");
}
```

Sometimes that might matter, sometimes it might not. Or it might matter for only some pieces of data. In any case, there's an easy fix: just make sure every Gizmo type has the data you don't want to lose. Even if a certain type of Gizmo doesn't actually use that data, let the Gizmo hold on to that data anyway.

If you need to hold on to all the data, then as with the `AnyGizmo` above, each Gizmo type will take up as much space as the largest Gizmo. In fact, in such a case, using the `AnyGizmo` above may be a good idea. Except this time, the performance cost of doing conversions can be almost completely eliminated.

How? Suppose every type of Gizmo does need to hold all the possible Gizmo data, and you use `AnyGizmo`. In that case, if you arrange all the data members of all the Gizmo types so everything is always laid out in memory the same way, then to convert the `AnyGizmo` from one type to another, all you need to do is change the `currentType` member. That's it. Everything else will already be in the right place.

At this point, it might sound like we've merely re-invented the `DynamicGizmo`. While it is similar, there are two important differences:

First, this method allows us to have completely different functions for each Gizmo type. Each type can have a different set of functions, and each function can be specially-tailored to the features that Gizmo is supposed to support. So the Gizmos don't have to spend any extra processing time dealing with being flexible (for instance, by checking their own `isSpinnable` variables to see whether or not to actually spin). Essentially, we have a cheaper form of polymorphism. In fact, if a `DynamicGizmo` isn't included in the `AnyGizmo`, then there can still be a space savings over `DynamicGizmo` because variables like `isSpinnable` still won't need to exist at runtime at all.

Second, if we want to change multiple settings on a Gizmo at the same time, we only need to actually modify one value: `currentType`.

Curtain call

Although the examples throughout this article have focused on situations with large numbers of simplistic objects, these techniques can also work very well with smaller numbers of objects that do a lot of processing. The templated versions of `UltraGiz` give just a small glimpse of this.

We've seen that you can use metaprogramming with structs to achieve much of the same flexibility as classes while avoiding the class overhead. But even if you do go with classes, these metaprogramming techniques can still aid in optimization without forcing you to cut features and give up flexibility.

Ok, so towards the end we did start running into more tension between efficiency and flexibility. Perhaps those wacky nuts will never fully resolve their differences. But even so, they've made some major progress.

Whenever you have values or settings that don't need to change, you don't have to choose between eliminating them for efficiency and keeping them for flexibility. You can have both, with no compromise. And even if your values and settings do need to change, but just not constantly, you still have many options available for getting your efficiency and flexibility to cohabitate peacefully. So go ahead, use metaprogramming to have your efficiency, and flexibility too. ■

Acknowledgements

Thanks to Lars T. Kyllingstad, bearophile and Timon Gehr for their suggestions.

References

[1] http://dlang.org/phobos/std_variant.html

An electronic version of this article is available at http://semitwist.com/articles/EfficientAndFlexible/PartTwo/

ACCU Conference 2012

Chris Oldwood recalls his experiences of the ACCU 2012 conference.

Another year had flown by and all of a sudden the annual ACCU Conference was upon me once more. This being my fifth year I was pretty comfortable that I knew roughly what to expect. However, a change in hands of both the conference chair (now Jon Jagger) and the hotel owner (whilst we were staying there!) meant that there was always going to be something new to delight us.

Tuesday

The main conference is preceded by a tutorial day that I didn't attend this year. For those trying to digest as much about C++11 as possible this would have been a fine start as Nico was back in the fold with an update to his fine book on the *C++ Standard Library*. Not that the other two options – Java garbage collector tuning and IT design/architecture – should be glossed over, but C++ 11 definitely appeared to be the primary focus.

I arrived early that evening with the intended goal of checking out my presentation on a projector, but after checking in and walking back towards the main part of the hotel I bumped into various people who I've not seen since last year. So I decided that a quick pint and catch-up wouldn't hurt as there was plenty of time before tomorrow. I guess some of us just never learn…

Wednesday

Tim Lister got to kick-off this year's conference keynotes with a talk on Project Patterns, on which he and his colleagues have written a new book. He suggested that Project Habits would have been a more suitable name as it was not about defining best practices but documenting common behaviours. Although he courted controversy with LESSONS UNLEARNED, the one that generated the most interest was DEAD FISH – a project that smells rotten but nobody's willing to speak up about it. This was a delightful start to the conference that set the tone in more ways than one.

DEAD FISH – a project that smells rotten but nobody's willing to speak up about it

As someone who does little C++ these days, I wasn't there to soak up all the C++ 11 goodies, but I still wanted to keep my eye in. C++ has its share of warts and Nico Josuttis put together a talk on the 'Best & Worst New Features' that nicely illustrated how to get the most out of the recent changes, but more importantly identified some of the new traps you may fall into. And there are a few peaches! I think overall Nico had more Best than Worst features to discuss so it's clearly a net gain.

The lunchtime routine changed slightly this year and it felt as though the bottleneck of past years was now behind us – I certainly got served more quickly each day. So I had ample time to chat before spending my early afternoon listening to Detlef Vollmnan describe 'Parallel Architectures'. This session was a nice spot of revision on the changes in CPU architectures over the last decade and a look forward to what we can expect in the near future. He briefly covered the classic problem with the DOUBLE-CHECKED LOCK pattern to illustrate how re-ordering under the covers affects us and also the costs associated with NUMA configurations. This provided a pleasant prelude to a talk I would attend the following morning.

Last year I gave my first ever talk and so I was hoping that by this year I'd have learned some new tricks and would give a slightly more polished show. I think the slides were a little more colourful and I felt brave enough to do a little live coding that I hope added to the experience. With a strong C++11 track I wasn't expecting to get many interested in TDD based database development, but there were more than enough still awake at the end to make me feel pleased with my efforts.

The lightning talks are now a staple part of the conference and provide a forum at the tale end of the day to either add a poignant finale to a keynote or allow the more disgruntled among us to let off steam with a good old rant. There is generally more of the latter which was perfectly illustrated by Pete Goodliffe with some choice examples from his current codebase. On the more constructive side Jonathan Wakely reminded us about the virtues of `shared_ptr` and `make_shared` in particular. Aaron Ridout got to reply to a talk from Seb Rose last year suggesting that *not* taking up new employee references can be detrimental to your company too. With eight talks on the first night there were clearly no problem finding willing volunteers.

Thursday

It was business as usual Thursday morning as I bolted down my fry-up to get to the keynote after a late night in the bar chewing various programming 'fats'. Phil Nash had the honour of representing ACCU's home-grown talent with a piece titled 'The Congruent Programmer'. Given that he could barely speak a week before due to a nasty cold, he did an excellent job of trying to answer that tricky question about what it is that drives us and what introspection might help us to develop ourselves in the future. I thought the analogy to the Alexander Technique was a little tenuous but his point was well made.

After the morning coffee break I decided I needed to feed my concurrency habit further and so embraced Russel Winder's talk on the multi-core revolution. Russel is not shy of telling us how wrong the shared memory approach is and that the Actor, DataFlow and CSP models are the way forward. To him shared memory multi-threading is technique for OS implementers, not application programmers. He decided to cement these ideas by implementing the various forms in Go, D and C++ to show how they might look; with an eye on whether the new C++ threading additions are too little too late as it doesn't have the high-level constructs, yet. Given what I had planned to see in the afternoon this was a fine bit of middle ground.

The vast majority of the sessions after lunch were 45 minutes long so I decided to pick two opposite views on the state of the C language – Dirk Haun's 'Is C going the Way of the Dodo' and Andrew Stitcher's 'When Only C Will Do'. The former was part-musing from the speaker and part-workshop to allow the audience to provide some thoughts on why there are still so many C programmers and whether that will continue. It was interesting listening to the various opinions and small doses of nostalgia always goes down well. I felt a little less satisfied by the follow-up talk because I didn't find the argument for pure C to be at all compelling (as opposed to using a subset of C++). Andrew's example did reinforce the notion that C is the common ground between all languages due to its ABI,

CHRIS OLDWOOD

Chris started as a bedroom coder in the 80s, writing assember on 8-bit micros. Now it's C++ and C# on WIndows. He is the commentator for the Godmanchester Gala Day Duck Race and can be reached at gort@cix.co.uk or @chrisoldwood

but perhaps sometimes it's only the C ABI that will do, not necessarily the entire language…

More coffee was required before my final concurrency session, this time with Anthony Williams. I wondered if I had entered a parallel universe with Russel talking earlier about C++ and Anthony using Groovy for his examples! This was a great companion to that earlier session because Anthony added in the ACTIVE OBJECT pattern and LOOP PARALLELISM too. He also used his own Just Thread library to show how you can build on the C++11 standard library to give the higher-order mechanisms for the actor and dataflow models.

The discussion of cohesion was particularly useful as it's a term that's often banded about but not well defined

The second round of lightning talks gave Phil Nash a chance to extend his keynote further by providing a heart-warming tale about one particular user of his C++ unit testing framework Catch. Matt Turner vented anger on fluent style interfaces whilst Charles Bailey tackled the classic 'Law of Three'. It wouldn't be an ACCU conference without Didier Verna pointing out the superiority of LISP and he kindly obliged.

The Xbox Connect Challenge provided an amusing backdrop to the evening's proceedings in the hotel bar as grown men (and women) were flinging themselves around in an attempt to outdo one another. Pete Goodliffe set a good early pace, but his crown was soon taken and the high-score rose to the heights of the more hardened gamers. There were definitely a few sore limbs the following morning for those who let the alcohol fuel their competitiveness.

Friday

Uncle Bob had the pleasure of entertaining us on the Friday morning with a 'Requiem for C'. After going to two sessions the previous day on the topic it was interesting to see what spin he would put on its demise. There was a fair amount of personal background to start with that finally lead up to his eventual inauguration to C. Then we had the downward slope as he tried to convince us that we are getting further and further away from the metal with technologies such as the JVM and so consequently C's value is diminishing rapidly. Some may have bought this, but from the lightning talks to follow others weren't so easily taken in. He also wore a head-cam during the keynote which should make for interesting viewing by others later.

'Devops, Infrastructure-as-code' by Gavin Heavyside was my first choice of the day. This was one of those thoroughly practical talks that gives you insight into how others teams do stuff; the stuff in question being continuous deployment and other devops related activities such as monitoring. He took a deeper look at Chef, which is used to manage deployments, and the DSL used in configuring/testing it and explained how they perform zero-downtime deployments. The use of tweets from DevOps Borat as a backdrop made this all the more enjoyable.

I find that I need to limit myself on the number of sessions by Kevlin Henney as he makes me think too damn hard by questioning the status quo, and with a title like SOLID Deconstruction you new this was going to do just that. Last year he turned SOLID into just SID and this year it became FLUID, although serious artistic licence was required for the D! The discussion of cohesion was particularly useful as it's a term that's often banded about but not well defined. One thing I found most interesting, but which was never highlighted, was the way that he named the test fixture and test methods in one of his examples.

Sadly I spent too long chatting over coffee to get into John Lakos' talk and so I had to go find another, and I'm really glad I did as Jurgen Appelo was a pleasure to listen to. He took us through the murky waters of 'Complexity Thinking and Systems Thinking' to look at its failed application to social systems such as software development. I thought this was going to be way out of my league as I had no idea who any of the 'experts' were, but fortunately he translated the concepts into something us mere mortals could understand and added a large dash of humour to keep the session from drying up. It's sessions like these that help break down the divide between programmers and managers.

The final round of 11 lightning talks opened with Tom Gilb following up last years attempt to quantify love by showing that music can be quantified. Diomidis Spinellis tried to convince us that we should alias the entire C++ library so that it follows Java naming conventions. Ed Sykes and Raj Singh explained how they had pushed pair programming to the next level with Posse Programming and Bernhard Merkle responded to Uncle Bob's keynote by showing that C, albeit in a more modern non-standard form, still matters.

Friday night plays host to the speakers' dinner where delegates get the chance to wine and dine with those that have preached to them during the last few days. Each course is met with a change of table to ensure there is a constant stream of new faces to chat with, and a new random element this year meant speakers could be forced to move too. Once again coffee was served whilst an auction raised money for Bletchley Park and the boat race made a welcome return too in the bar afterwards to battle out once and for all whether C was really dead or not…

Saturday

Even though the disruption caused by the volcano was well and truly behind us the conference has maintained the format of the lightning keynotes on the Saturday morning. These are 4 shorter keynotes hosted by a variety of speakers. Naturally, over indulgence from the previous night meant I missed the first two and so I started the day listening to Emily Bache's 'Geek Feminism'. Emily took a look into what differs between men and women to see what it might be about the way software development teams work that might be off putting to women. She made a particularly poignant observation that the Wisdom of Crowds is amplified by diversity and so being a minority is actually highly beneficial. The final keynote was from Jurgen Appelo that looked into what motivates us. He took various established sets of core values and munged them together to come up with his own variant that goes by the rather long-winded acronym CHAMPFROGS!

Giovanni Asproni has always been the conference chair when I've attended and so unshackled from these duties he had the opportunity to host a session instead. This was on API Usability and looked into what it means to design an API, whether that is for public or private consumption. His example of how the Single Responsibility Principle had been taken to the extreme was amusing but also enlightening. He is never one to pass up the opportunity to berate the SINGLETON pattern, and this occasion was no exception. Giovanni pulled together lots of little threads to provide one of those bread-and-butter talks that reminds you of all the little things you have to remember every day.

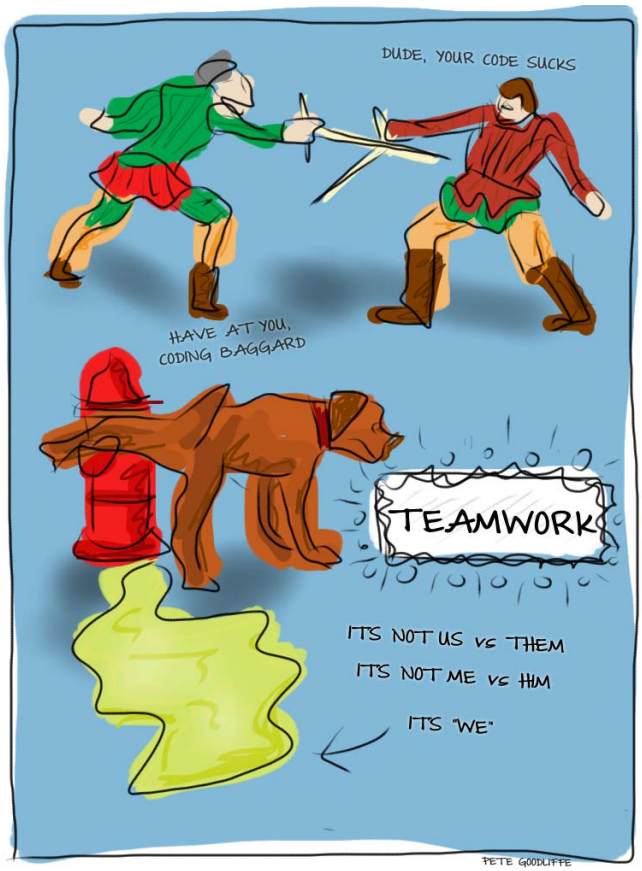
My final session of the conference was to be with Diomidis Spinellis about UML Graph, a tool to enable the generation of small UML diagrams using annotated source code. The premise of generating model diagrams from source code versus manual drawings was compelling and something I've been recently looking into with Doxygen. He then went on to discuss other tools for generating different sorts of diagrams and charts to give you plenty of options on how to programmatically create them; a very informative end to the conference.

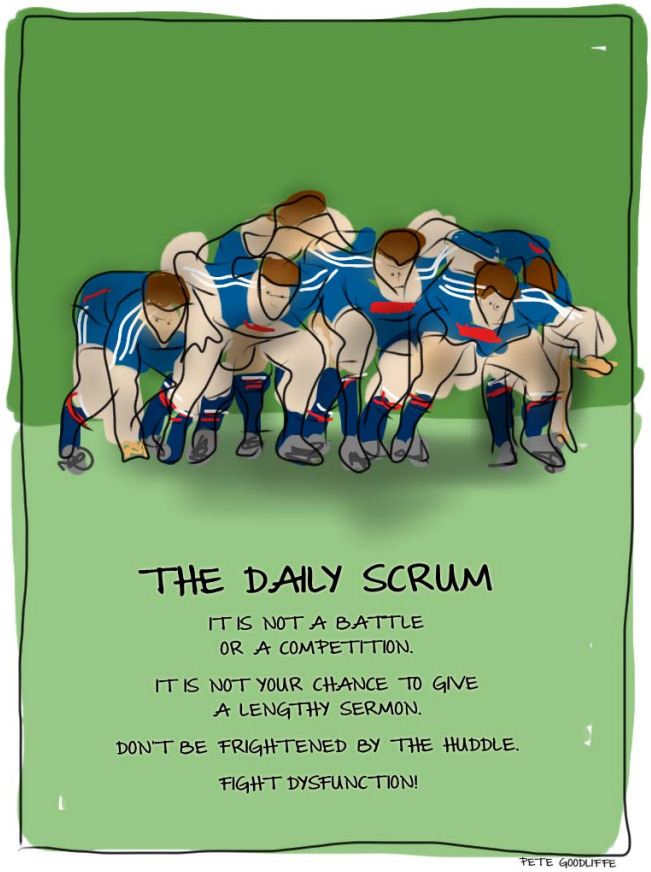
Epilogue

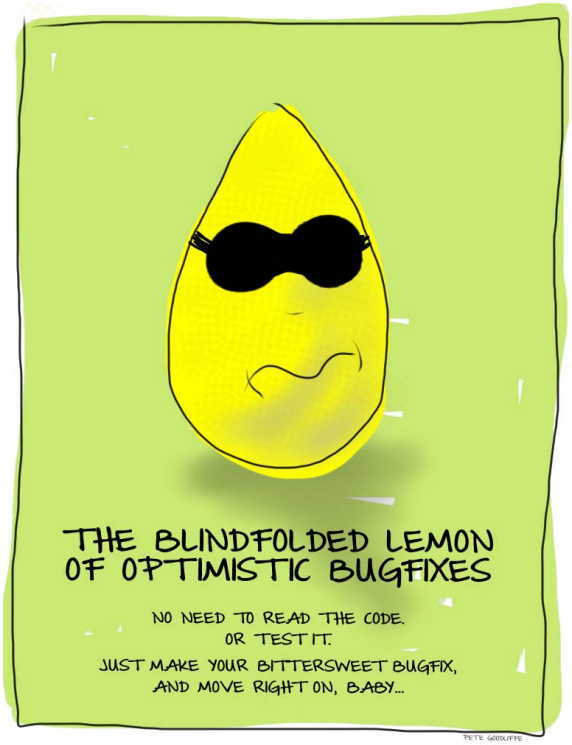
The hardest part about coming home from four days at the ACCU conference is integrating oneself back into society. It's an intense few days where you can literally eat, drink and sleep all things about programming. Depending on the kind of organisation you're going back to you might be bursting with energy just waiting to pass on all that valuable knowledge, or perhaps you're a small cog in a very big wheel with disinterested co-workers. Either way *your* participation means that you'll be a better programmer for having gone and ultimately that's what it's all about. ■

The Art of Software Development

Pete Goodliffe vents the modern developer angst.

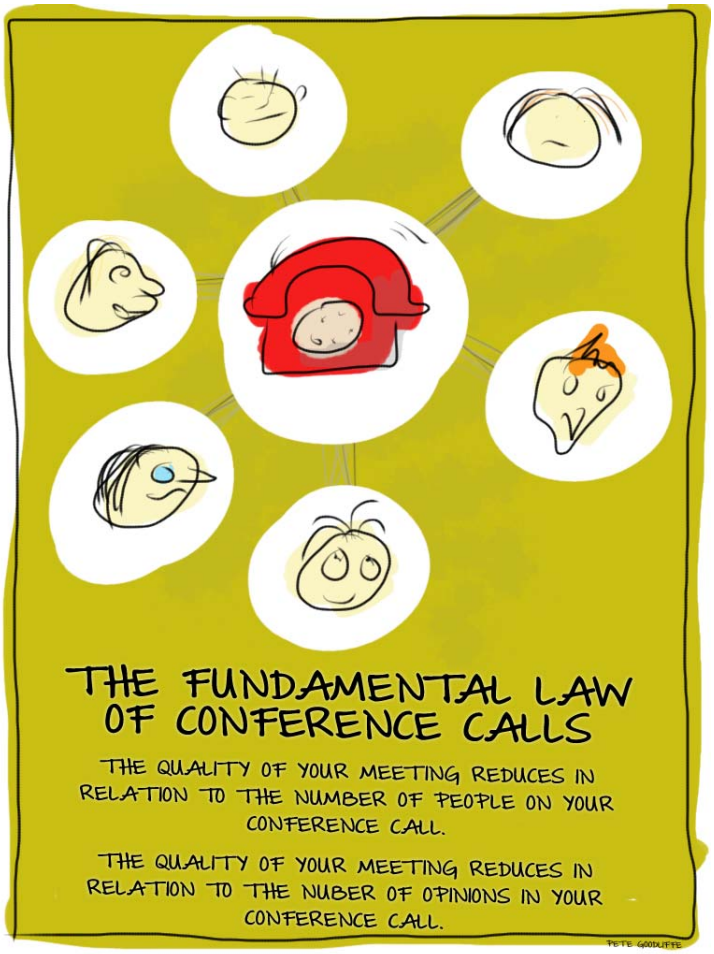


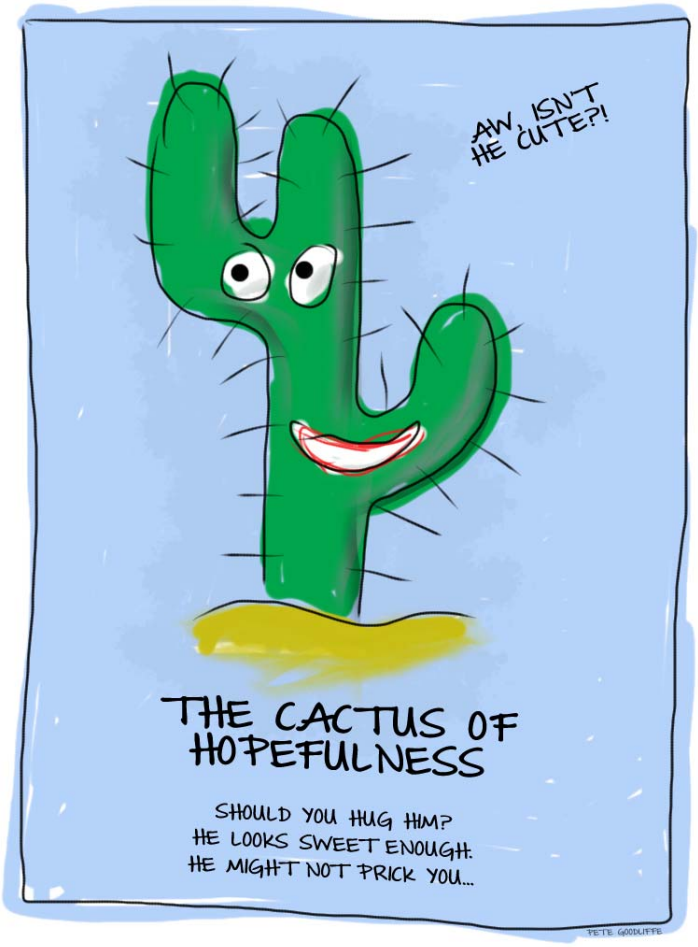


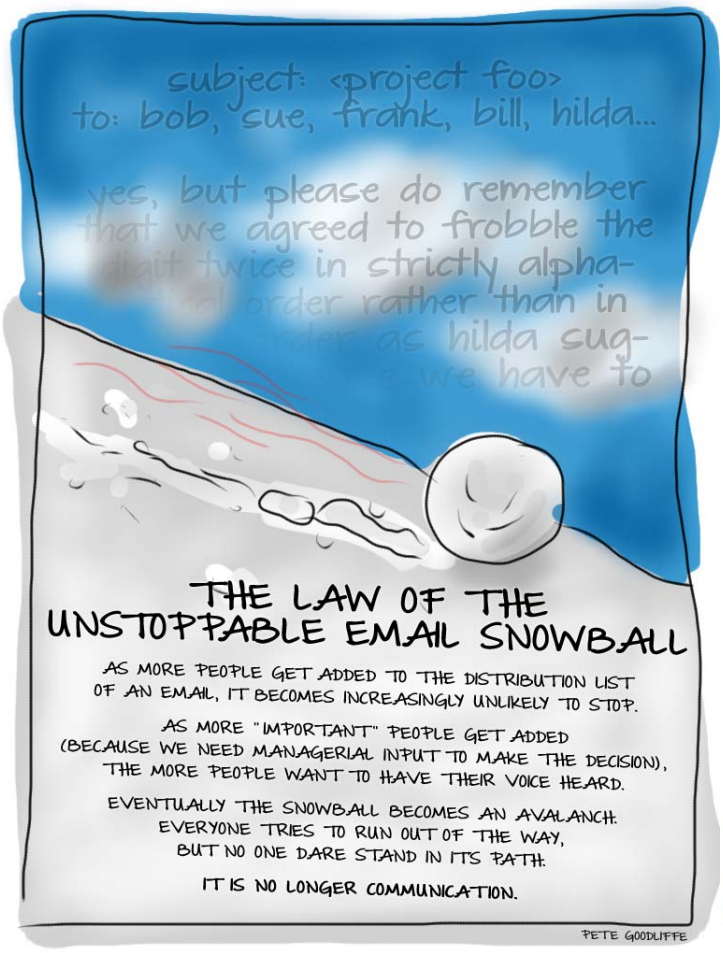


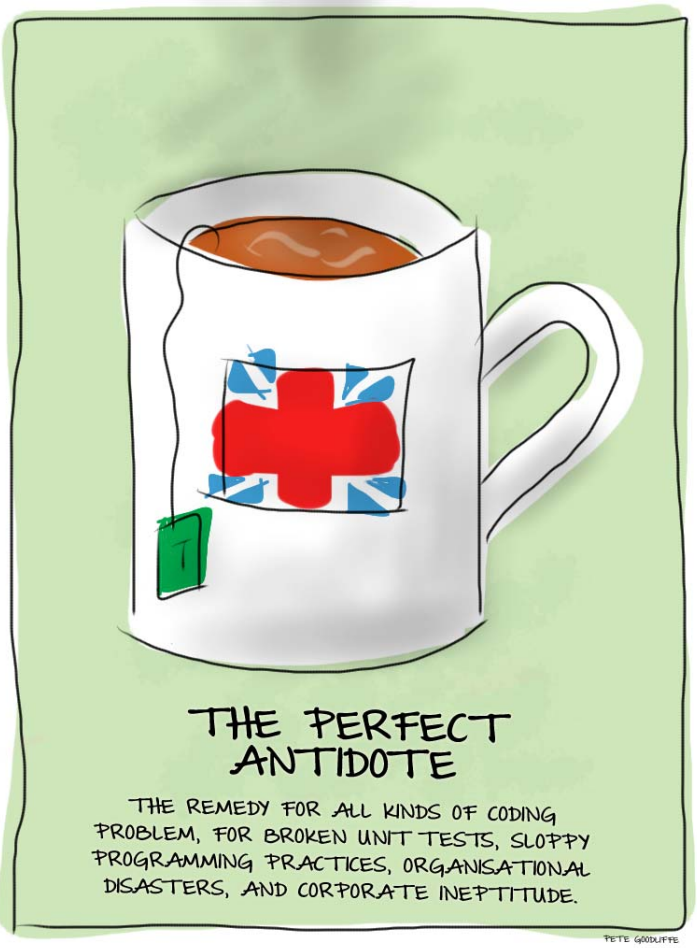
PETE GOODLIFFE

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


THE FUNDAMENTAL LAW OF CONFERENCE CALLS
THE QUALITY OF YOUR MEETING REDUCES IN RELATION TO THE NUMBER OF PEOPLE ON YOUR CONFERENCE CALL.
THE QUALITY OF YOUR MEETING REDUCES IN RELATION TO THE NUBER OF OPINIONS IN YOUR CONFERENCE CALL.
PETE GOODLIFFE


AW, ISN'T HE CUTE?!
THE CACTUS OF HOPEFULNESS
SHOULD YOU HUG HIM? HE LOOKS SWEET ENOUGH. HE MIGHT NOT PRICK YOU...
PETE GOODLIFFE


subject: <project foo>
to: bob, sue, frank, bill, hilda...
yes, but please do remember that we agreed to frobble the
twice in strictly alpha-
order rather than in
as hilda sug-
we have to
THE LAW OF THE UNSTOPPABLE EMAIL SNOWBALL
AS MORE PEOPLE GET ADDED TO THE DISTRIBUTION LIST OF AN EMAIL, IT BECOMES INCREASINGLY UNLIKELY TO STOP.
AS MORE "IMPORTANT" PEOPLE GET ADDED (BECAUSE WE NEED MANAGERIAL INPUT TO MAKE THE DECISION), THE MORE PEOPLE WANT TO HAVE THEIR VOICE HEARD.
EVENTUALLY THE SNOWBALL BECOMES AN AVALANCH. EVERYONE TRIES TO RUN OUT OF THE WAY, BUT NO ONE DARE STAND IN ITS PATH.
IT IS NO LONGER COMMUNICATION.
PETE GOODLIFFE


THE PERFECT ANTIDOTE
THE REMEDY FOR ALL KINDS OF CODING PROBLEM, FOR BROKEN UNIT TESTS, SLOPPY PROGRAMMING PRACTICES, ORGANISATIONAL DISASTERS, AND CORPORATE INEPTITUDE.
PETE GOODLIFFE

Patterns and Anti-patterns – Factorisation

Richard Polton shows how redundant code can be removed by factoring to a functional style.

When I came to it, my inherited codebase was very procedural and imperative and contained a lot of classic bugs, by which I mean commonly-occurring types of bugs, which often occurred multiply because of the temptations of the Evil Demon Cut-And-Paste. This prompted a thorough trawl through the codebase looking for certain patterns. The plan, such as it was, was to factorise the patterns (having corrected them where necessary) and to migrate the bulk of the imperative code to a declarative style. Factorisation also has the pleasant benefit of reducing wordcount, albeit sometimes at the expense of an increase in density. This article describes one of the patterns.

First, however, let me briefly comment on declarative programming. I suppose I mean that declarative style indicates intent, ie 'this' is defined to take 'that' value, where 'that' is probably, but not necessarily, the results of a function call, unless of course you live in a world where everything is a function, in which case 'that' is a function too!

Phew!

The declarative style is a Good Thing (tm) because it reduces side-effects, although in my code I should point out that it directly led to one side-effect, namely far fewer overnight support calls! I call that a Good Thing (tm) too. Anyway, reduction of side-effects is a Good Thing (tm) because that means that the code exhibits reproducible behaviour. That's nice to have because it is easier to reason about reproducible behaviour than seemingly-random behaviour. Being able to reason about behaviour is good because it implies structure.

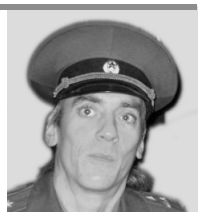
Structure implies order. Order opposes chaos. QED ;-)

... and I am a definite and fully signed-up fan of 'immutability' too. I should have mentioned that :-D

And so to the meat of the matter

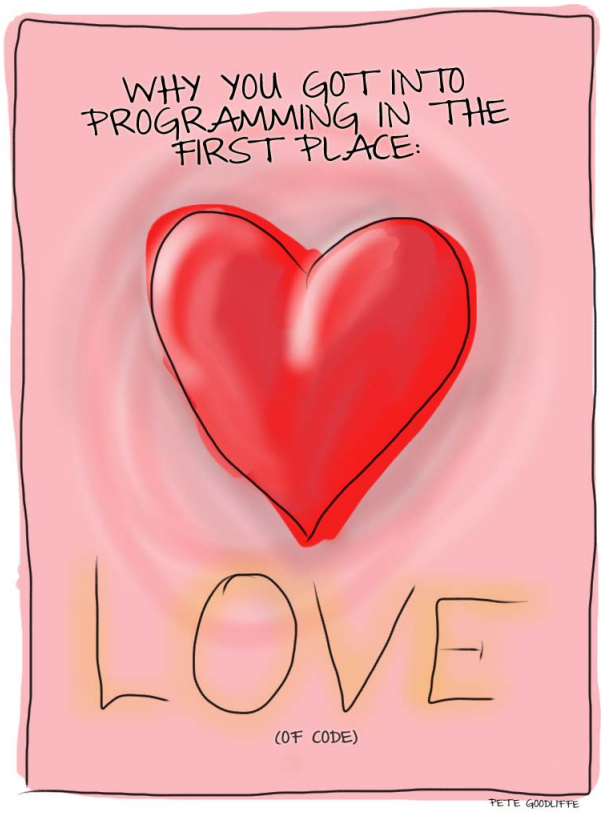
One construct that I have stumbled across frequently in my current inherited codebase is the long, and sometimes complex, `if` statement. Invariably, in its simpler form, it is structured as a sequence of predicates joined by either a logical 'and' or a logical 'or'. Oftimes these predicates all take the same argument as each other. For example, one pattern which was seen to occur frequently is given by

RICHARD POLTON



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

The Art of Software Development (continued)



```
ComplexObject a = ComplexObject.CreateInstance();
//do something with a
if(a.GetParameter().Contains("A") ||
  a.GetParameter().Contains("B") ||
  a.GetParameter().Contains("C") ||
  a.GetParameter().Contains("D"))
  ; // do something positive
else
 ; // do something else
```

which suggests to me that, at the very least, `a.GetParameter()` is a prime candidate for factorisation. Possibly, `Contains` could be factorised as well. Additionally, it can be seen that the logical 'or' is also a potential candidate for factorisation.

But back to the code. As presented the above snippet is a fair example of what could be seen throughout. The first question to be considered is whether `GetParameter()` or `Contains()` have side-effects. If not, then the call to one or both of `GetParameter` and `Contains` can be factorised. On the other hand, if there are side-effects, then we have a design problem as well as a code problem. I intend to deal with only the code problem here.

So, after a first pass of factorisation, we have

```
ComplexObject a = ComplexObject.CreateInstance();
// do something with a
string p = a.GetParameter();
if(p.Contains("A") || p.Contains("B") ||
  p.Contains("C") || p.Contains("D"))
  ; // do something positive
else
  ; // do something else
```

Well, so far so good, but this is pretty standard stuff.

At this point, however, we conclude that there is at least one more feature that can be factorised; the `Contains` function call. This is where things can become interesting.

Factorising `Contains` out of the expression leaves us with, conceptually, this:

```
if( p.Contains ("A" || "B" || "C" || "D"))
```

Okay, so I admit that this is not valid C# code but it is heading in the right direction. With more recent versions of the .NET Framework, certainly in 3.5, it is possible to write

```
if( new[]{"A","B",
          "C","D"}.Any(str=>p.Contains(str)))
```

which has the extra benefit of factorising the logical 'or' operators as well, replacing them with `Any`. The equivalent replacement for logical 'and' is `All`, both of which are defined in the `System.Linq` namespace. For those who are unfamiliar with the syntax, the `=>` operator denotes a lambda function, the LHS of which being the local variables and the RHS being the function body. The `Any` function is an `Extension` method – why did Microsoft create `Extension` methods instead of using static functions? – which operates on `IEnumerable<T>` and applies the lambda predicate until its return value is `true`, ie proper short-circuit evaluation. Similarly, `All` operates on `IEnumerable<T>` and applies the lambda predicate while its return value is `true`, terminating early (short-circuit, again) if any of the predicates return `false`.

Similarly, if the array of strings being searched is not composed of constant strings, we can still use this pattern. For example,

```
new[]{"A",GetB(),
      "C",GetD()}.Any(str=>p.Contains(str))
```

However, this has the downside of constructing the array before evaluating the lambda function, and this means that `GetB` and `GetD` will be evaluated even if the `p.Contains("A")` is `true`. This is the case because C# is an immediate language. That is to say that values are evaluated at their point of definition. The converse is a lazy language in which values are evaluated at their point of use. Therefore, each of the elements of the array will have been created before the `Any` function is applied to the array.

Listing 1

```
public bool Equals(ChangeObject x,
                   ChangeObject y)
{
  // Check whether the compared objects reference
  // the same data.
  if (Object.ReferenceEquals(x, y)) return true;
  // Check whether any of the compared objects
  // is null.
  if (Object.ReferenceEquals(x, null) ||
      Object.ReferenceEquals(y, null))
    return false;
  // Check whether the products' properties
  // are equal.
  return x.Quantity == y.Quantity
    && x.Price == y.Price
    && x.Identifier == y.Identifier
    && x.FXFixing == y.FXFixing
    && x.Fixing == y.Fixing
    && x.Spread == y.Spread
    && x.TradeDate == y.TradeDate
    && x.TradeCurrency == y.TradeCurrency
    && x.ValueDate == y.ValueDate
    && x.UnderlyingIdentifier ==
       y.UnderlyingIdentifier
    && x.BuySell == y.BuySell
    && x.Client == y.Client
    && x.ClientBroker == y.ClientBroker
    && x.Spread == y.Spread
    && x.Book == y.Book
    && x.Name == y.Name;
}
```

This can be avoided by delaying the construction of the individual elements of the array until they are required, although this has the disadvantage of further complicating the code. This is both disappointing and to be demonstrated later.

So, at this point it looks as if we have completely factorised our expression and can move on to other pieces of code, remembering to return so that we can 'deal with' the array.

Next, another frequent construct in my codebase that is similar to the above example, except that the complex object is now the function parameter instead of the calling object, something like this.

```
ComplexObject a = ComplexObject.CreateInstance();
// do something with a
if(Utils.Prepare(a) && Utils.CalcWith(a) &&
   Utils.OutputResultsFrom(a))
  ; // do something positive
else
  ; // do something else
```

There is still factorisation to be done here, except this time the common component is the `ComplexObject a`. Factorising `a` out of the expression leaves us with an array of predicates, which we can use as follows

```
if(new Predicate<ComplexObject>[]{
    c=>Utils.Prepare(c),
    c=>Utils.CalcWith(c),
    c=>Utils.OutputResultsFrom(c)}.All(f=>f(a)))
```

where `Predicate<T>` is defined in the .NET Framework as equivalent to `Func<T,bool>`. Even though the compiler can deduce the type of non-function types, for some reason known only to themselves, the C# team have required us to specify the type of the lambda functions, at least in v3.5. We can, however, make use of so-called Method Groups to reduce the typing, and thus comprehension, burden. And so we have

```
if(new Predicate<ComplexObject>[]{
    Utils.Prepare,
    Utils.CalcWith,
    Utils.OutputResultsFrom}.All(f=>f(a)))
```

Listing 2

```
var i1 = new Func<ChangeObject,int>[]
  {c=>c.Quantity};
var i2 = new Func<ChangeObject,string>[]
  {c=>c.Identifier,c=>c.TradeDate,
   c=>c.TradeCurrency,c=>c.ValueDate,
   c=>c.UnderlyingIdentifier,c=>c.BuySell,
   c=>c.Client,c=>c.ClientBroker,
   c=>c.Book,c=>c.Name};
var i3 = new Func<ChangeObject,double>[]
  {c=>c.Price,c=>c.FXFixing,
   c=>c.Fixing,c=>c.Spread};
```

I think you will agree that we have achieved brevity here and I hope that you will agree that writing only the code we need, without repetition, makes it easier to comprehend. This is as far as we can take this particular direction.

Should we wish to proceed with the removal of the type specifier from the definition of the anonymous array, it is necessary to split this code across multiple lines, which is somewhat contrary to the spirit of the exercise. This is a shame, and someone should mention it to the C# team the next time they pop over for a cup of a tea and a biscuit. The compiler ought to be able to deduce function types as well as non-function types; after all, we can ;-) So, we will temporarily disregard our desire for brevity and (visual) simplicity, and divert along a different path. So, in long, we have

```
Predicate<ComplexObject> prepare = Utils.Prepare;
Predicate<ComplexObject> calcWith = Utils.CalcWith;
Predicate<ComplexObject> output =
   Utils.OutputResultsFrom;
if(new []{prepare,calcWith,output}.All(f=>f(a)))
```

Recall to mind the comment made about 'immediate' versus 'lazy' languages and the fact that C# falls into the former camp. This means that **`prepare`**, for example, has been defined to be **`Utils.Prepare`**. Had our definition been

```
var prepare = Utils.Prepare(something);
```

then **`prepare`** would hold the return value of the **`Utils.Prepare`** function. But here, **`prepare`** is defined as the function itself.

Armed with the above, we observe that we can delay the construction of our earlier array, remember the one which contained **`GetB`** and **`GetD`**? comme ca:

```
Action<string> getA = () => "A";
Action<string> getB = () => GetB();
Action<string> getC = () => "C";
Action<string> get/8D = () => GetD();
if(new []{getA,getB,
          getC,getD}.Any(str=>p.Contains(str())))
```

This time, **`GetB`** and **`GetD`** are not evaluated when we create the anonymous array because the array is an array of functions to be evaluated instead of an array of results of functions which have been evaluated. It's all in the tense :-)

And so now we do not evaluate **`GetB`** and **`GetD`** unless they are actually required because of the short-circuit evaluation behaviour of **`Any`**.

At this point, having seen that we can factorise functions as well as data, it would be reasonable to ask whether we can improve on this.

Again, in our codebase, we have a number of **`IEqualityComparer<T>`** classes and, invariably, their **`Equals`** and **`GetHashCode`** functions are remarkably similar. Additionally the **`Equals`** function is almost always a boiler-plate piece of code which compares all the relevant properties of the **`T`**. For example, see Listing 1.

This is actually snipped directly from the codebase and one source of error may be immediately apparent – this code was constucted using the tried-and-tested cut-and-paste operation and, as a result, **`Spread`** occurs twice! The code for **`GetHashCode`** uses the same set of properties, even down to the duplication of **`Spread`**. There are a number of observations that can be made about the final **`return`** statement. It is composed from a number of equality-testing predicates each of which is constructed by repeating the function on the LHS and RHS of the equality. Also, the results of the predicates are combined in a logical **`and`**, which we have already seen is equivalent to the **`Extension`** method **`All`**. Finally, although not shown here, **`GetHashCode`** uses the same set of functions. All of this duplication is a possible source of error as demonstrated in the example code.

Unfortunately, it is at this point that things become difficult. This is because the functions need to have the same type in order to hold them all in the same container. This is a feature of the .NET generic containers which accept only values of a given type, or possibly related type.

In our case, some functions return a string, some return an integer and the remaining return a double. One approach is to group them into three arrays (see Listing 2).

Recalling **`x`** and **`y`** are **`ChangeObjects`**, we can combine these three arrays using the **`Extension`** method.

```
return i1.All(f=>f(x)==f(y)) &&
       i2.All(f=>f(x)==f(y)) &&
       i3.All(f=>f(x)==f(y));
```

This has the advantage that **`i1`**, **`i2`** and **`i3`** can be re-used in the **`GetHashCode`** implementation if they are moved into class scope. However, this implementation is not ideal. I would like to be able to ignore the intermediate return value type of the properties and factorise the equals method too, but presently the equals operators, although appearing similar, are three different functions: **`int.Equals`**, **`double.Equals`** and **`string.Equals`**.

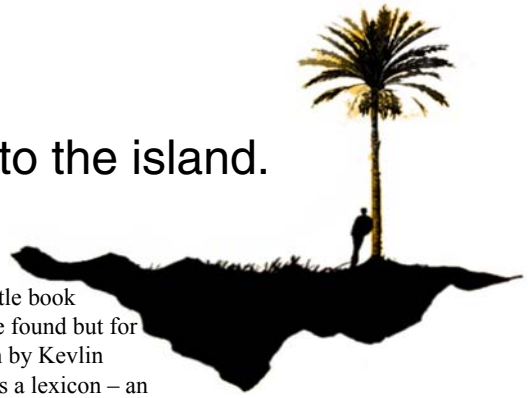
At this juncture, I have not solved this problem. ■

Next time...

Next time we will investigate ...

Desert Island Books

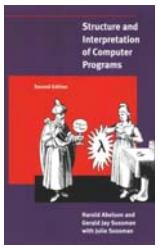
Mick Brooks shares what he will take to the island.



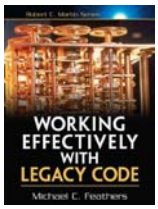
I haven't met up with Mick half as often as I would have liked too, but we've often exchanged emails. I first encountered Mick on one of the early Mentored Developers projects. He's one of the quiet men of the ACCU, but he's there, often in the background doing a solid job as membership secretary.

Mick Brooks

So, my colleagues finally formed an ostracism committee and arranged for me to be left on an island? I can't say I blame them: it's probably for my own good, and it's very nice of them to let me take some books.

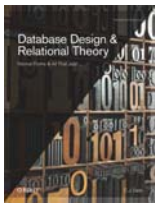


Choosing the first book to pack is easy: I have unfinished business with *Structure and Interpretation of Computer Programs* by Harold Ableson and Gerald Jay Sussman. Our reading group at work took this on recently, and we made good progress through the first couple of chapters. By chapter three I'd had to accept that I had no hope of doing all of the exercises and keeping up with our modest pace, but by the middle of chapter four we'd all run out of steam. Each section took only minutes to read, but seemed to require more and more time wrestling with the exercises before I could claim to get it. There are so many important ideas in this book, and the exercises and footnotes provide so many jumping off points. Time alone on the island to tackle it all would be a good thing, but I'm worried that a lack of reference material might drive me insane. I'll have to take my chances.

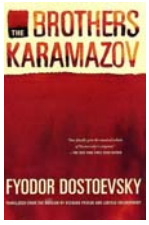


Next I'd grab *Working Effectively with Legacy Code* by Michael Feathers. I raced through this when I first picked it up – it was a very reassuring read. I'd been bitten by the automated testing bug while I was a student, but was finding it difficult to apply in my first programming job. This book seemed to be written by a friendly uncle who'd seen the same frustrations (proving I wasn't an idiot!), and even better, knew what to do about them. In hindsight, I don't think I made the best of his advice. The aspiration for testable code stuck, and I think improved the new code I wrote, but I never really put the techniques for working with not-new code into practice. I definitely owe it a revisit.

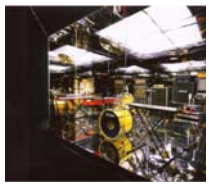
Third comes *Software Requirements & Specifications – a lexicon of practice, principles and prejudices* by Michael Jackson. This is a lovely little book that I'd never have found but for a recommendation by Kevlin Henney. It really is a lexicon – an alphabetical list of phrases related to analysis, specification and design, each followed by a short explanation or example of the concept. It's a book to dip into, so I can't tell you what proportion of it I've read: definitely not all of it, but I've read some parts many times. It's not a tutorial, in fact I'm not sure what it is, or how it's useful. It is entertaining though, and it would be perfect for idle island pondering.



Last of the techie stuff is a new book that I've been planning to read soon: *Database Design & Relational Theory* by C.J. Date. I've had quite a bit of practice with databases, but I've a yearning to fill in the theoretical foundations. The bits of theory that I've picked up tend to come with an 'aha'-moment as the formalism helps clear up some woolly thinking. I also nod along whenever I hear how SQL and most database technologies don't live up to the relational ideal. I've a suspicion that the NoSQL movement is at least partially a reaction to the limitations of existing implementations, rather than of the theory, and that the theory will be an important tool for finding our way around all the new products and approaches.



Well, looking at what I've packed so far, I've given myself a pretty hard time on the tech side. I don't see why my choice of novel should be any different. I'm not a big reader of fiction: I've never trusted my own judgement of what's good, and so I tend to stick to classics. I read a few Dostoyevsky novels and really enjoyed them, but got horribly stuck on his *The Brothers Karamazov*. Supposedly his best work, I've read the first third to a half twice now, but both times had to abandon it until I stopped feeling so morose. I quite like being miserable, but it wasn't fair on my family and friends. No worries about that on the island, so I should be able to wallow in it. And if it all gets too grim, I'll have the weekly design review sessions with my team of coconut-for-a-head developers to look forward to. Will they have index cards on the island, or should I try and smuggle some with me?



Finally some music. Here I will try and lighten the mood. My first choice is *Mirrored* by Battles. An amazing 'math-rock' album that always makes me smile, and usually has me doing strange hand-dancing. Then *69 Love Songs* by The Magnetic Fields. This is a compendium of short songs in a variety of styles. There are some properly catchy tunes in there, and the lyrics have a combination of cleverness and silliness that I really love.

Right, they're here, so it must be time to go. Do I have to wear that jacket? Okay. And the sleeves go this way? Oh right, I see. Can you just get that bit at the back...

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 us: cvu@accu.org.

After four years, Paul Grenyer is stepping aside from Desert Island Books. Thanks Paul! We want to continue introducing ACCU members, so if there's someone you think would make a good candidate for marooning on the island, let us know at cvu@accu.org. Or even better, introduce them yourself!

Code Critique Competition 76

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

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

Last issue's code

I've 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 1: `callback.h`
- Listing 2: `cb.h`
- Listing 3: `counter.h`
- Listing 4: `callbackTest.cpp`

Listing 1

```
#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 2

```
#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 3

```
#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 4

```
#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;
}
```

ROGER ORR

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

Listing 4 (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();
};
```

Critiques

Brook <jingyu_chen@hotmail.com>

In invoking **registerCB**:

```
CB() : registered(true) {
  ::registerCB(&fn, this);
}
```

the second **this** argument logically should point to a **Counter** object, but it's actually the pointer to a **CB** object; so we get the parents, giving this modified **class CB**:

```
template <typename T>
class CB
{
public:
  CB() : registered(true) {
    regist(fn); // << change >>
  }
  static void fn(void* arg)
  {
    static_cast<T*>(arg)->callback();
    //(T*)(arg)->callback();
  }
  virtual ~CB() {
    if (registered) {
      unregisterCB(&fn, this);
    }
  }
private:
  // << add >>
  virtual void regist(void (*fn)(void *)) {
    ::registerCB(fn, this);
  }
  // ...
};
```

Paul Floyd < Paul_Floyd@mentor.com>

Just glancing at the code and one thing jumps out

```
static_cast<T*>(arg)->callback();
```

Without even compiling the code, this looks suspicious. The other thing that I spotted was the semicolon after the closing curly brace of **main**.

Lastly, I didn't like the use of difficult pointer to function syntax without using a **typedef**, which would have made things much easier.

When I do compile the code, I get a lot of warnings.

```
CC -library=stlport4 +w2 callbackTest.cpp
"callbackTest.cpp", line 13: Warning: function
  void(void(*)(void*),void*) overloads extern
  "C" void(extern "C" void(*)(void*),void*)
because of different language linkages.
"callbackTest.cpp", line 11: Warning: fn hides
  the same name in an outer scope.
"callbackTest.cpp", line 12: Warning: arg hides
  the same name in an outer scope.
"callbackTest.cpp", line 21: Warning: function
  void(void(*)(void*),void*) overloads extern
  "C" void(extern "C" void(*)(void*),void*)
  because of different language linkages.
"callbackTest.cpp", line 19: Warning: fn hides
  the same name in an outer scope.
"callbackTest.cpp", line 20: Warning: arg
  hides the same name in an outer scope.
"cb.h", line 12: Warning: arg hides the same
  name in an outer scope. 7 Warning(s) detected.
```

I get as output

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

Looks a bit more reasonable than the junk numbers in the original Critique, but still wrong.

Next, let's try some debugging. The first thing that I notice is that if I watch the global **arg**, it doesn't have the same value as **this** in the first call to callback. It is the same when I step through **exercise**. The global **arg** is being assigned the **this** pointer of type **CB<Counter>** not that of type **Counter**.

Now for the fixes. Firstly, lets clean up the warnings and add a **typedef**.

```
typedef void (*fn_t)(void*);
```

then it's possible to write

```
extern "C"
void unregisterCB(fn_t fn_, void *arg_)
```

and avoid the 'hiding' warnings by adding an underscore _ to the argument names, e.g.

```
static void fn(void* arg_)
```

Now to the crux of the matter. Changing the **CB** constructor to

```
explicit CB(T *pthis) : registered(true) {
  ::registerCB(&fn, pthis);
}
```

and **Counter**'s to

```
Counter() : CB<Counter>(this), counter(0) {}
```

looks like it sets the global **arg** counter to the correct **this**.

From a design perspective, it's not pretty (global variables are rarely elegant, **void*** ones even less so). However, as I've done things like this in the past, I won't get all sanctimonious. If you are forced to use C, then either you can only use a very limited subset of C++, or you have to resort to dirty hacks like this. Personally I'd prefer to get rid of the C code, but that isn't always possible.

Helmut Wais <helmut.wais@gmx.net>

The main problem in the code I see is an invalid cast from pointer-to-object to pointer-to-void and then back to a different pointer-to-object.

The C++ standard (98 version) says in 5.2.9/10 (it's 5.2.9/13 in the C++11 standard):

> An rvalue of type "pointer to cv void" can be explicitly converted to a pointer to object type. A value of type pointer to object converted to "pointer to cv void" and back to the original pointer type will have its original value.

The 'back to the original pointer type' condition is not satisfied here. In the **CB** constructor, **this** is of type pointer-to-CB. It is implicitly converted to pointer-to-void in the call to **::registerCB()**. Later, in **CB::fn()** that pointer-to-void is cast to pointer-to-Counter, which is not the 'original pointer type'.

My first suggestion is to drop that C-library. In case you absolutely have to do that cast from **void*** to **Counter***, then my second suggestion is to do it via an intermediate **CB** cast (**CB** has to be a **public** base for that):

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

Another problem is the use of a pointer to a static member where a pointer to an extern 'C' is expected. My compiler (gcc47) allows that, but others may not.

Implementation details:

> gcc puts the vtable at the beginning of the object. Since the base **CB<Counter>** has no vtable, but the derived **Counter** does, the CB-subobject is not at the same address as the **Counter** object. The cast to pointer-to-void erases all type information: the compiler cannot adjust the pointer in the cast to pointer-to-Counter.

Simplified code to demonstrate the problem:

```
#include <iostream>

struct A {
  int i; // prevent empty-base optimization
};
struct B : A { virtual ~B() {} };

//   vtable  A.i
// |.......|....| // 8+4 byte on my 64bit machine
// ^       ^
// |       |
// |       |
// |       +------ A-subobject at: 0x7fff5fbff608
// +-------------- B-object at:    0x7fff5fbff600

int main()
{
  using namespace std;
  B b;
  cout << &b << endl;
  // 0x7fff5fbff600 on my machine

  A * pa = &b;
  // 0x7fff5fbff608, ok: pointer adjusted
  void * pv = pa;
  // 0x7fff5fbff608, ok

  cout << static_cast<B*>(pv) << endl;
  // 0x7fff5fbff608, nok: void->derived
    cout << static_cast<B*>(pa) << endl;
    // 0x7fff5fbff600, ok: base->derived
}
```

Balog Pál <pasa@lib.hu>

This is a very good sample. I imagine the scenario, someone presenting it as 'See, I have this callback-registry. I wrote some unit tests, and it works fine. I know it has some issues, but do you see any actual problem rather than cosmetic issue? I really want to check it in right now, we'll deal with beautifying later...' I think it is pretty easy to dismiss the problem, and let it go.

Certainly here we see the misbehaviour, but it's all too easy to use some different class in a test that 'works fine'.

Knowing that there is a problem made me read the code differently, and spot the root right on. Mostly because I had to deal with other manifestations of the same root cause in production. Hunting actual crashes after slight and apparently unrelated changes in the system...

First I tried the code in the compiler: clean compile and the result produces similar bug effect. Great. (I mean it. We have a 99.9% clue for undefined behaviour, and for those chance is pretty high to observe some good behaviour that hardly helps.) I change the odd 'virtual' that was the only dark-red flag on the superficial reading – I mean making **~CB()** virtual – and presto, the behavior now meets the original expectations.

Certainly it is not the root cause of our problems. Destructor of a class meant as base class should better be virtual, I enforce it in guidelines except for corner cases like COM classes, but it is not mandatory. And using such class as base must work fine, unless we use delete improperly. It this example nothing wrong is done in that respect. As a matter of fact, this change should not have any visible impact on this code at all. Or any other code aware of the just mentioned problem, unless it queries **sizeof()** of the classes.

This change can (and in practice likely does) have an effect on the layout of the classes. We can see it in MSVC using some undocumented magic, the switches **/d1reportSingleClassLayout<classname>** or **/d1reportAllClassLayout**. I've set packing to zero to get rid of confusing alignments...

Originally it is:

```
class ?$CB@VCounter@@    size(1):
    +---
 0  | registered
    +---

class Countersize(9):
    +---
 0  | {vfptr}
    | +--- (base class ?$CB@VCounter@@)
 4  | | registered
    | +---
 5  | counter
    +---
```

After adding virtual to dtor:

```
class ?$CB@VCounter@@    size(5):
    +---
 0  | {vfptr}
 4  | registered
    +---

class Counter    size(9):
    +---
    | +--- (base class ?$CB@VCounter@@)
 0  | | {vfptr}
 4  | | registered
    | +---
 5  | counter
    +---
```

The important thing to look at is the placement of the base class within the derived. After the change they start at the same place. A pointer to the virtual method table (VMT) is at the start – it is present in any class with at least one virtual function. This part is never duplicated. In the derived class the new data member (counter) is just put at the next location. Before it we see the base embedded.

In the original version the compiler wants the VMT pointer at the start, but as the base class didn't have it, so it is placed at offset 4. We can say 'in the middle' of the layout.

Knowing all that it shall not be hard to find the problem place. We just need to follow conversions between pointers to those structures. One of them shouts at us loudly as **static_cast<> in CB::fn()**. It's

counterpart lives in **CB**'s **ctor** (and **dtor**). It's not so visible, as conversion toward **void*** is implicit.

So let's see the types in conversions:

- in **CB::CB** we have **this**, that has type **CB<T>***, and convert it to **void*** and store that.
- in **CB::fn()** we have that **void*** and cast it to **T***.

We use **static_cast** that is supposedly safe and sound, but in reality an **X* → void* → Y*** chain is identical to a **reinterpret_cast<Y*>(X*)**. For the supposedly obvious reason that **void*** carries no type info. It's a common question on forums whether in this case we should spell static or **reinterpret_cast**, and have all kinds of answers, some say use static, as it works, and we're supposed to use the weakest thing possible. Others say use reinterpret, as that is what happens, and it better be visible. This case probably supports the second camp.

Well, having **reinterpret_cast** in the code is not the end of the world, we just must be sure to obey the round-rules: we can use whatever many reinterprets on the pointer, but the last one, where we intend to use the pointer, must have the first type we started. In other words we must get back to the pointer's original type. And here that is not happening. To correct the situation, in **CB::fn()** we shall use **reinterpret_cast** (or **static_cast**) to **CB***, and only then **static_cast** further to **T***. Like

```
static_cast<T*>(static_cast<CB*>(arg))
  ->callback();
```

As soon as we do that, we get the desired behaviour, no matter how we juggle with the virtuals, or use other ways to challenge the layout. The situations where both **static_cast** and **reinterpret_cast** between two pointers both work but produce different results are pretty rare in real life, I'm aware of only two cases, one is the late-VMT as here and the other is multiple inheritance. No wonder it may catch programmers by surprise. But the consequence is undefined behaviour that in practice manifests in corrupting memory, as we writing through a pointer that missed an offset-correction.

Before going for other issues with the example, let's look at the big picture. On a code review the first questions are normally 'what is this class' or 'how does it work', or even jumping right to some detail. The better first question is to ask 'why did you write this class (package, library, ...)?' As we (should) know, the best code is the one you don't write. Think you need to implement another vector, string, smart pointer, quicksort, logger, ... ? Think again. Really. And explain why the zillion of existing ones is not good enough, or how yours will be better. The question applies here too. Why we need another homegrown, buggy, half-thought attempt at a callback wrapper? What is wrong with using **Boost::function** and **bind**, or those in the **g*** world the sigc++ library, or something else? Callbacks are pretty common, and I'd bet any framework would come with some support. And if it's a very special project forbidden to include a component it's still more productive to import sources.

Having said that, let's look for other problems, without ordering:

- We see include guards in some headers, that is good, but not all of them!
- The include guard uses lowercase. **#define** symbols are supposed to be all-uppercase almost anywhere. Also the symbols should be more unique.
- Headers shall be self-contained. Here `counter.h` includes `cb.h`, good, but `cb.h` just hopes **registerCB** is declared somehow before its inclusion. Similar problem that `counter.h` uses iostream.
- In `callback.h` functions take pointer-to-function as argument. For such beasts we should use a **typedef**, both for better readability and avoiding duplication.
- Documentation in `callback.h` is insufficient: for a registry/database operation we shall document the primary key! Can I register the same function with multiple different args and have them all? Can I register the same pair multiple times? If so, unregister removes all of them or keeps count?
- I expected to see functions of `callback.h` implemented in `callback.cpp` or some other file, that belongs to the library itself. Not in the unit test. (though it might be the 'mock' and that was just omitted as irrelevant.)
- The callback registry looks 'too global', and too generic. The callback systems I worked with normally had either more collections to observe or provided extra info used as event filter.
- With **CB/Counter** using CRTP I see what is happening, but not really get the point that why it is good, and what is the benefit. Unfortunately the documentation here is completely missing, and having just `cb.h` should I guess the intended use is to derive from **CB<T>** and have a member function called **callback()**?
- The already mentioned lack of virtual dtor in CB that is intended to serve as base class.
- CB has a member **registered** that has no usage. It can be scrapped until there is interface to fiddle with registration explicitly (or the **registerCB** function gains return value).
- Name hiding throughout, mitigated by use of **::**. It's good we can use qualified names when name clash happens out of our control. It does not mean we shall fight to invent ways to use it. It's not that hard to find another identifier for our arguments.
- The test code is not very impressive, though the ones I write also have a tendency to look awful, yet they do the job well. Without specification of the tested module we can't evaluate the goodness of the test anyway.
- There's an excess semicolon at end of **main()**.

Commentary

The critique was inspired by some code of mine that broke when the class hierarchy changed by the removal of a virtual method. The problem turned out to be an incorrect cast – as in this case – but it took a while to track it down. The trouble comes from the way C++ builds up an object and the different addresses involved. In this case the most derived object, test, is an instance of the **Counter** class which is derived from **CB<Counter>**. In the base class constructor the object's **this** pointer is passed as the 2nd argument to the **registerCB** function, implicitly casting it to **void***. When the callback fires, the argument **arg** is turned back into an object pointer in the function **fn** using a **static_cast** to a pointer to the **Counter** class. Unfortunately this cast, while it compiles, does not do the right thing.

One way to demonstrate this is to add code to print out the address of **this** into both the constructors. When I do this I get the following output:

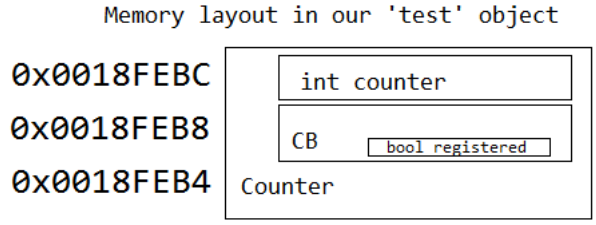
```
CB: 0018FEB8
Counter: 0018FEB4
Counter: 1
Counter: 1638153
Counter: 1638154
Counter: 1638155
```

We can see that the two addresses are different. In this case the base class, **CB**, portion of **Counter** is based 4 bytes into the overall object. It is this address which is passed to the C API, and casting this address to a Counter results in an address out by 4 bytes.

Figure 1 is a diagram of the memory layout in this particular example. The overall **Counter** class contains within itself the CB sub-object at an offset of 4 bytes and the counter value at an offset of 8 bytes. When we cast the address of the **CB** sub-object to a Counter pointer we have the wrong base address, and so when we reference the counter value at offset 8 we are reading the first integer beyond the end of the object.

One solution is to ensure we do a static cast that is the exact opposite of the implicit cast we first used, to get back the address of the **CB** sub-object, and then cast this pointer to a Counter pointer.

Figure 1




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

Unfortunately, performing this static cast via `CB` requires that `T` is publicly derived from `CB<T>`. At this point I'd make the dtor of `CB` protected to avoid trying to delete via pointer-to-base.

The Winner of CC 75

All four critiques identified that the problem was with the casting. The danger with this sort of example is that the bug is caused by undefined behaviour, so simply making the bug go away without understanding *why* the problem occurs runs the risk of the program *still* invoking undefined behaviour – but apparently working fine.

So Brook's proposed change, calling a (virtual) helper function in the ctor, and adding virtual to the dtor, does result in code that works on many compilers – but sadly it *still* invokes undefined behaviour. The change works because the layouts of the two classes now overlap (on most implementations, anyway).

Paul's solution, passing a pointer to the full derived class to the base class constructor, does fix the problem as the implicit cast to `void*` does now exactly mirror the `static_cast` to `T*`. However I am a little unhappy with the unnecessary extra argument begin passed to the ctor.

Both Helmut and Pál gave similar solutions. Helmut provided the link to where in the standard it defines what is and is not defined behaviour when using `static_cast` with `void*` and both provided explanations of the actual memory layout in a specific implementation (and thank you to Pal for the information about the undocumented MSVC compile switches!) Helmut also noted that the proposed solution requires that the inheritance from `CB<T>` becomes `public` – sadly MSVC accepts the code without this (even with extensions disabled).

The resultant code, as Helmut noted, still has one piece of undefined behaviour: using the address of a static member function where a pointer to an extern 'C' function is required. This is in practice rarely an issue, but would be worth fixing for a truly portable solution.

I found it a difficult call to pick a winner (as Pál provided additional critiques of the code), but I liked the definitive nature of Helmut's solution so I have awarded Helmut this issue's prize.

Code Critique 76

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

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

The code:

- Listing 5 is `exception.h`
- Listing 6 is `test_exception.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 5

```
#include <iostream>
#include <vector>

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

Listing 6

```
#include <iostream>
#include <iterator>
#include "exception.h"

int func(int i)
{
  try
  {
    if (i <= 0)
    {
      throw exception("i must be positive");
    }
    int result(i*i);
    if (result < i)
    {
      throw exception("overflow");
    }
    return result;
  }
```

Standards Report

Mark Radford reports on the latest developments for the C++ standard.

You may notice a change of name on the standards report this time around. I've now taken over the role of Standards Officer, Lois Goldthwaite having now stepped down after several years in the role. Therefore, it seems appropriate to say a bit about what qualifies me to take over this role. I have been a member of the C++ Standards Committee for thirteen years. I have participated at ISO level and I regularly attend BSI Panel meetings. Although my interest has historically been in C++ standardisation, I have no intention of limiting the scope of my reports. Therefore, would anyone reading this, who is involved in a relevant standards' process, please make themselves known to me?

Before I go any further, there are two people I would like to say thank you to. Firstly, I would like to say a huge thank you to Lois, for all her work as standards officer. Happily, this is not a complete exit from the standards arena for Lois, as she remains convenor of the BSI C++ Panel. Secondly, I would like to thank Roger Orr for filling in when Lois was unable to write these reports, and also for his encouragement in getting me started.

UK To Host April 2013 C++ Standards Meeting

You may already have heard that the UK will be hosting the ISO C++ Standards Committee in the week following the ACCU conference, specifically 15th – 20th April 2013. The location is still to be decided, but will be the same as the ACCU conference, either in Oxford or Bristol. At the time of writing, this still needs £8,000 worth of sponsorship. If any readers have contacts that could help with this, please can you get in touch? Feel free to contact me initially.

Towards C++ 2017

The next update to the C++ Standard (nicknamed C++1y) is planned for 2017. C++ software developers have been kept waiting thirteen years between the original standard (1998) and C++2011 (the 2003 release was an interim release to fix various issues, but did not present an update to the standard). There is a strong desire in the standards committee that future updates should be much more frequent – about every five years.

So, what might be in C++2017? Well, one topic that has been discussed recently at BSI Panel meetings, is that of scheduling pieces of work for processing. Typically this involves scheduling the work onto a thread pool for concurrent execution, but it might also involve scheduling for sequential execution on a particular thread. This all started with a proposal from Google, which can be read in full in full [1].

This is just one example of a proposal that has been put forward. Sorry, but because of space and time limitations, I can't go into this any more at this point. However, it is something to look at again in a future column.

One last thing before I finish: the call for C++2017 library proposals is very much still out there. If you want to know more, there is a paper containing more information [2].

References

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

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

MARK RADFORD

Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

Code Critique Competition (continued)

Listing 6 (cont'd)

```
  catch (exception & ex)
  {
    ex("in func");
  }
  std::cout << "Shouldn't get here"
            << std::endl;
}
int mid(int i)
{
  try
  {
    return func(1) * func(i);
  }
  catch (exception & ex)
  {
    ex("in mid");
  }
  std::cout << "Shouldn't get here"
            << std::endl;
}
```

Listing 6 (cont'd)

```
void test(int i)
{
  try
  {
    std::cout << "mid() => " << mid(i)
              << std::endl;
  }
  catch (exception & ex)
  {
    std::cout << "Caught exception"
              << std::endl;
    ex.print();
  }
}

int main()
{
  test(0);          // should fail first test
  test(0x7fffffff); // should fail second
}
```

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)

The Geek Manifesto : Why Science Matters

By Mark Henderson, published by Bantam Press, ISBN:978-0593068236

Reviewed by Ian Bruntlett

This is a good book. Unfortunately it is a hardback and I'm trying not to buy hardbacks any more due to lack of shelf space and one of my book cases is on its way out. I bought it from Amazon UK where it was reasonably priced so I bought it anyway :)

In the past I, and probably other I.T. workers, have taken the stance that 'I don't want to get involved with politics – I just want to get things done'. This book takes the position that science belongs throughout our culture instead of being isolated in a ghetto. A manifesto is 'a public declaration of policy and aims, especially one issued before an election by a political party or candidate'.

Previously when asked to explain science and how it reacts to change I've referred people to *The Structure of Scientific Revolutions* by Thomas S. Kuhn. This book is making me more politically aware. It describs the use of the Office for Budget Responsibility to 'scrutinise the state of the public finances and the Treasury's economic forecasts' and suggests that a similar body be established – 'An Office for Scientific Responsibility could scrutinise the data advanced in support of every new government policy and report on whether it really passes muster'.

An OSR would be well placed to prevent the abuse of scientific evidence in politics. In fact, there are so many ways to abuse scientific evidence that the Henderson categorises some of them: Evidence shopping, Imaginary evidence, Fixing the evidence, Clairvoyant evidence, Mishandling the evidence and Secret evidence.

After nearly 250 pages of discussion, the book concludes with the chapter 'Geeks of the World unite'. It is a very short chapter and suggests the establishment of a geek movement and how geeks should be political animals. Fitting for a book advocating an evidence based approach to matters, after the final chapter there are roughly 60 pages of references, 4 pages of acknowledgements and a 10 page index.

GNU Make

By Richard M. Stallman, Roland McGrath, Paul D. Smith, published by GNU Press, ISBN:1-882114-83-3

Reviewed by Ian Bruntlett

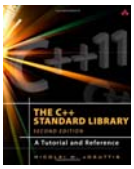
What is Make? It's a special language to build software applications. A file, typically "makefile" or "Makefile" lists the related modules that go into the making of an application using timestamps so it knows which modules are up to date and do not need to be recompiled. Sometimes an IDE will provide a GUI that sits on top of a Makefile. Users of old Linux applications will be familiar with the 3 commands to install an app – ./configure; make; make install but that is less common these days.

Some years ago I bought some stuff from the GNU people. One of those things was the manual for GNU Make. As well as being available in paper format, the most up to date version can be downloaded free and printed from http://www.gnu.org/software/make/manual/

My exposure to Make started when I got hold of C68 on the Sinclair QL. Then I got a job at LiBRiS which, at that time, was moving from Borland Turbo C++ to Watcom C. I was given a disk of source code and told to get it running. To cut a long story short – I got it running, I created Makefiles for families of LiBRiS products using Watcom's own Make – wmake supported by a basic source code control system called SCHOLAR.

This book is *very* detailed. Its authors recommend that first time readers concentrate on the first chapter and just read the initial pages of the remaining chapters. While it provides plenty of example snippets while describing various commands, I feel that more example makefiles would have been helpful.

The C++ Standard Library 2nd Edition



By Nicolai Josuttis, published by Addison Wesley Longman, ISBN: 978-0-321-62321-8

Reviewed by Francis Glassborow

There are very few programming books that remain in the top ten for as long as the first edition of this book. For over a decade the first edition of this book has remained as one of the essential reference books for any serious C++ programmer.

During that time the C++ Standard Library has undergone considerable change. The most important of these was when the first *Library Technical Report* was published. At that time the author was deeply involved in other programming languages and I suspect, rebuffed any overtures from his publishers to update his book. Nonetheless the first edition continued to sell and continued to be the first point of reference for programmers trying to get to grips with the C++ Standard Library,

However C++ has now moved on. Not only has its library been massively increased in size, but the original has undergone substantial revision to leverage on the changes made to the core of the C++ Language. The author has been seduced by these changes into renewing his acquaintance with C++. This is much to the benefit of the rest of us.

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

The author has brought his clear understanding of C++ to the latest version and written a second edition that will be an essential reference for the new, enhanced and completely revised C++ Library.

After a couple of unusually short chapters ('About this Book', and 'Introduction to C++ and the Standard Library') chapter 3 gives the reader a brief but very valuable overview of the new C++ language features. This chapter is important reading for anyone who is already familiar with C++ as it was prior to the new Standard.

Chapter 4 covers general concepts and should be read by everyone using this book as a reference (which, in this reviewer's opinion, is its most important and enduring use).

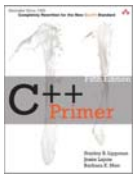
The rest of the book is an invaluable reference to much of the library. Unfortunately the C++ library has grown to a size where the kind of comprehensive tutorial/reference that the first edition provided is no longer possible. The author has had to be rather more selective and focus on the things that are important to the vast majority of C++ programmers whilst bypassing some of the more specialist parts. Even so the page count has now gone past 1000. I suspect that a truly comprehensive reference/tutorial would need over 2000 pages.

The text has been thoroughly checked by a large number of C++ library experts. How do I know that? Well I know most of the reviewers and have faith in their pride in doing such review work correctly. Perhaps one or two things may have crept past all the eagle eyes but none that I have noticed.

Now I must touch on the one negative aspect of this book. It has clearly not been copy edited by a native English speaker. The English syntax jars me far too often. This is a great pity because it spoils what is otherwise an outstanding work. The author's English is more than competent (and orders of magnitude better than my German) but it is the English of someone thinking in German and then making an excellent translation. Were it spoken you would probably not notice, but when written down it shows. I hope that when the author gets to produce a third edition he employs someone to copy edit the English.

If you are serious about writing C++ you need this book. If you already have the first edition you will know the quality of the author's writing and ability to provide that essential explanation that will assist you in getting the best from the C++ Standard Library. Those people will, if they are sensible, update their bookshelf by getting a copy of the 2nd edition. The rest of you should hurry quickly to your book supplier and purchase this edition it will save you a great deal of time.

C++ Primer 5th Edition



By Stanley Lippman, Josée Lajoie and Barbara Moo, published by Addison Wesley, ISBN:978-0321714114

Reviewed by Francis Glassborow

About 20 years ago Stan Lippman wrote the first edition of this book. At the time of writing it was among the best introductions to C++ published. A few years later the author produced a 2nd edition which tracked changes that were happening to the C++ language as it was being standardised. Soon after the C++ Standard was finalised the 3rd edition was published with a surprise second author, Josée Lajoie (well it was a surprise to me). Josée is a gifted teacher who at that time was working for IBM. She is a French Canadian but her English fluency would shame many for whom it is their first language. I know this from personal experience because she effectively mentored me during my first five years of active participation in WG21, giving freely of her insights and being very patient with this jumped up amateur. Her skills took a good book and turned it into an authoritative one.

Soon after the TC that updated C++ in 2003 a 4th edition appeared. This had acquired yet another new author, Barbara Moo, who has many years of programming experience and a real gift for writing about the most obtuse technical points with great clarity. She co-authored *Ruminations on C++* with Andy Koenig (her husband, and author of what must be the longest running programming book without amendments, *C Traps and Pitfalls*, as relevant today as it was when it was first published in 1989). *Ruminations on C++* is still a delight to read and is a model of how to write about technical things in a readable style. (If you have never read *Ruminations on C++*, get a copy and enjoy it.)

Barbara completely rewrote *The C++ Primer* (I do not know how much input came from Stan Lippman, and I guess none from Josée as she had moved on to academia and, as far as I know, was no longer programming in C++). I guess that Andy Koenig could not resist reading over her shoulder and making the odd suggestion now and again. The resulting book was even better than the 3rd edition (much better in my opinion, but that is not to belittle the previous editions).

And now C++ has undergone a major overhaul with 'making it easier to teach' being one of the criteria and so it is clearly time for yet another edition.

This time two things have been done, the text has been revised. However well you write the first time you think of better ways to express yourself when you come back to it after a few years. Comparing the 4th and 5th edition side by side shows that the authors (again I suspect that is mostly Barbara) have taken the text of the 4th edition and reworked it. The second thing is that the code and content has been completely revised to make use of the changes that C++11 introduced.

C++ is a vast language and any author writing an introduction must select what they intend to cover. I believe that the authors have made sensible decisions as to what to cover and what to omit. The only area that I cannot find (I am working from a draft without an index) that I think I would have covered is lambda functions. Those are at least as useful, and arguably simpler, as parameter packs and variadic templates which are covered.

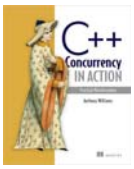
There is the additional problem with writing a book so soon after the release of the new standard; many compilers have not fully caught up with all the new bits but this is rapidly changing.

The authors write on the assumption that readers would be better off using a command line. That is a point with which I disagree. I much prefer to have novices use an IDE (such as Code::Blocks) with as up-to-date a version of g++ as I can get (currently I am using 4.7). However this does not really matter, just use whatever you are comfortable with.

If you are in the target readership, those with either a talent for programming or with some prior programming experience in another language (or possible C++ some years ago), then I can confidently say that this book will introduce you to C++ and set your feet firmly on the road to mastering the language.

This is not a book for those who are already relatively fluent in C++ as it was and wanting to update themselves to the latest version. Those people will need to look elsewhere. But if friends, colleagues or relatives want to learn heavy duty C++ and have some prior programming experience point them at this book and warn them that they need this edition and not an earlier one.

C++ Concurrency in Action



By Anthony Williams, published by Manning, ISBN:978-1-933-98877-1

Reviewed by Francis Glassborow

The author will be familiar to members of ACCU and those attending ACCU Conferences. Indeed anyone who attended his talk at the recent ACCU 2012 conference will probably already have bought a copy of this book.

C++11 (AKA C++0x) introduces a new memory model for C++ along with substantial support for concurrent programming. Some programmers (indeed originally quite a few members of WG21) think that multi-threaded programming is just a matter of using pthreads or some derivative of it. This is not the case. Many of our multi-threaded programs only scrape by because they have been running on a single processor. Indeed a great deal of 1990s thinking about multiple threads was based on that situation. We did not have to concern ourselves with the problems that arise when two processors simultaneously want access to the same data. It didn't happen, one or other got there first. Of course the indeterminacy of the

View From the Chair

Alan Griffiths
chair@accu.org

I wrote my last 'From the Chair' in March 2003 – nine years ago! I really expected it to be my last – but now there will be at least six more starting with this one. Looking for inspiration for this report I looked back at that one and was struck by how much is still relevant:

> This will be last 'From the Chair' and seems like a good opportunity for reflecting on where ACCU is going. It is my belief that the age-old imperative 'grow or die' applies and that what we need to seek is growth. What form that growth might take is up to all of us, but there are some lessons from the past.
>
> When I first became involved with the 'C Users Group (UK)' (as ACCU was then known) I was one of a few mavericks that were interested in an obscure C based dialect called 'C++' – and that interest largely because of the cross-platform promises made by the 'CommonView' class library.
>
> Well, CommonView probably pre-dates many of you, but the lesson that I want to draw is that this was accepted as a valid area of interest – and not outside the area interest of the C Users Group (UK). The area of interest grew to such an extent that C++ is now the lingua-franca of the organisation.
>
> The areas of interest accepted within ACCU keep expanding: my current responsibilities at work have almost no connection with C or C++ (very occasionally I get involved in resolving problems with JNI – an interface between our Java and C++ components). They have little to do with any specific programming language: they are to do with development processes and system design. But there doesn't seem to be any reason to seek out another organisation to discuss them: ACCU members are just as interested in change control strategies, Extreme Programming and testing as they are in the C programming language.
>
> In the journals and mailing lists there is a very healthy diversity of subject matter covered: design, version control, C++, XML, SQL, Java, Python, interview (and interviewing) techniques, C and probably more that don't spring to mind immediately. This reflects the range of knowledge that professional software developer comes into contact with during the course of his or her career.
>
> I take the fact that the membership of ACCU has been rising slowly during my term as an indication that we have been doing something right. And the strategy has been simple: if there are members willing to do something that sounds reasonable then give them the authority to do it. (Actually, I apply the same strategy in software development teams too.)
>
> What happens next will not be up to me, it will be up to you. If there is something that you think the ACCU should be doing, then do something about it...

Well, we haven't grown and we haven't died. One thing that has changed is that my current responsibilities are very much to do with C++ (in addition to development processes, system design and TDD). But I still see ACCU as a community interested in a wide spectrum of development topics. While I'm going back to C++ for a while that isn't the only thing that interests me and the ACCU members I know.

I've said it before, but it bears repeating: What happens next will not be up to me, it will be up to you. If there is something that you think the ACCU should be doing, then do something about it.

Bookcase (continued)

order might cause problems but that is not the same as genuine concurrency. Over the last few years we have moved on to a world where even our mobile phones have multiple cores (I can still remember the shock of discovering that C++ on the Symbian OS back in 2004 had to handle at least dual core processors in mobiles). Our programs need to be able to make use of the hardware. There is no point in running a program that needs good performance on a 16-core processor if our code has a single thread of execution.

So many of us know that we need to write concurrent code. Up until now the problem has been that doing so has meant using proprietary libraries and extensions. As of C++11 that has changed. It is now possible to write portable multi-threaded code.

Being able to do so is not the same as doing it. In addition there are various considerations that guide us to the most effective code for our purposes.

We need to consider issues of sharing data, synchronising operations, lock based versus lock-free and how to debug concurrent code.

Anthony covers all these and more in his book.

I found this book a pleasure to read and surprised myself with how quickly it helped me to progress in an area that I had previously avoided like the plague.

If you already use multiple threads you may think that you can simply dive straight into C++11. Well you may be right but my guess is that, in doing so, you will miss many of the tools that those who spent 8 long years working on concurrency for C++11 have provided. The library largely masks all the hard work by hiding its use in the internals of various classes but knowing these classes exist and what they offer will be a great help both to the newcomer to programming for concurrency and for the old hand.

Unless you have been as involved with the development of concurrency in C++ as the author has been, you will find this book a great help in bringing you up to speed on this important area of C++11. It is written in clear English and neither patronises nor assumes knowledge that some readers will not have. This is a book that you owe it to yourself to read, and read diligently and thoughtfully. Not to be read in the bath (well a few bits can be) but to be read alongside a computer running an up-to-date C++ implementation. In order to write this review I had to find a copy of MinGW with g++ 4.7 and discover how to use it from Code::Blocks. That in itself was an excellent piece of learning and sent me away with a warm fuzzy feeling of still being able to learn new things despite being in sight of my biblical three score years and ten. I felt even better as I worked through the text and discovered that I could actually manage it. Very timely as my publishers are showing interest in a new edition of my book, *You Can Do It!*, for inquisitive lay people (i.e. those who want to know what programming is about without intending to make a living from it).

Thank you Anthony for writing this book, I hope that many others choose to benefit from it as I have.