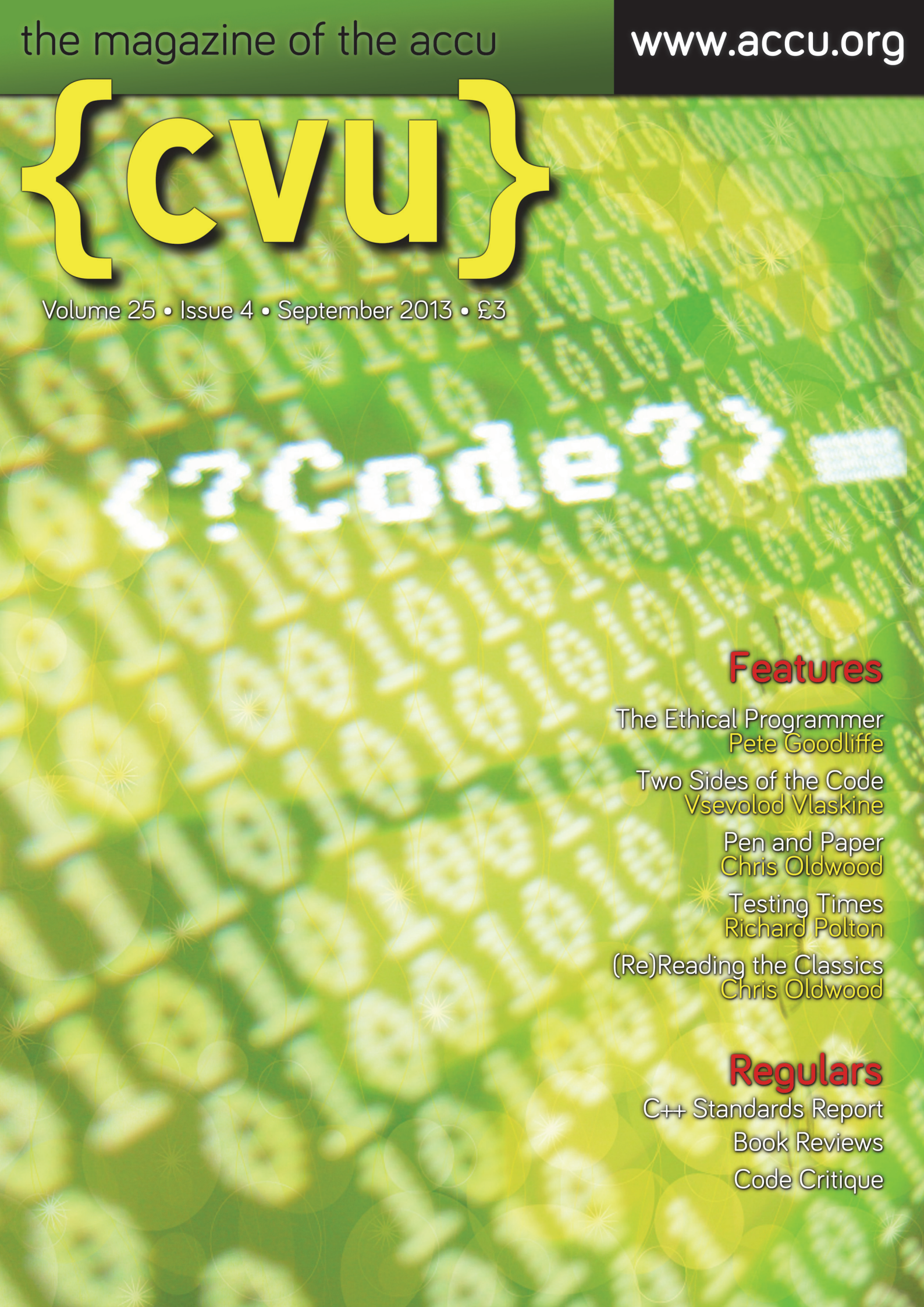
the magazine of the accu

www.accu.org

{cvu}

Volume 25 • Issue 4 • September 2013 • £3



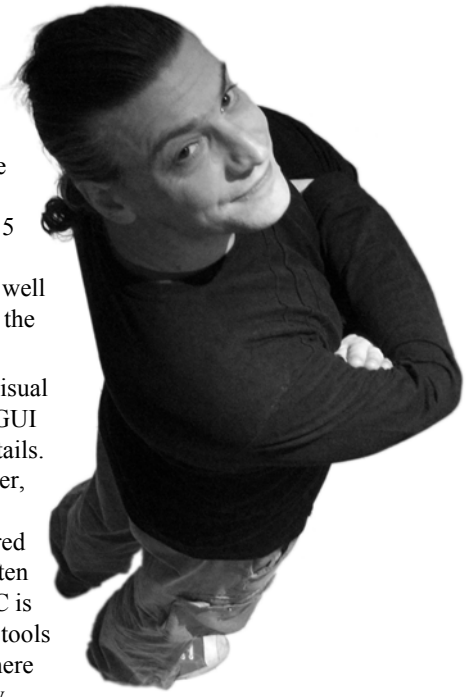
Features

Regulars

{cvu}
Volume 25 Issue 4
September 2013
ISSN 1354-3164
www.accu.org

accu

OSS Enterprise



I was struck the other day by a comment made about installing software. In particular, it was about the difference in experience between installing the latest GCC C++ compiler for Windows, and the latest Community Preview edition of Microsoft's Visual Studio 2013. The former is installed by extracting a compressed file, and then modifying the PATH environment variable to ensure the correct binary is picked up at the command line. It took me 5 minutes or so – including downloading it! VS2013 requires the most up-to-date Internet Explorer, took well over 2 hours to install, and then required a reboot at the end.

Ok so the GNU tools are command line only, and Visual Studio installs tools for C#, F#, web and Windows GUI toolkits, as well as the full IDE, with all that that entails. The truth is, however, I just wanted the C++ compiler, in order to hone my C++11 skills, and have at least two compilers to compare. Visual Studio is considered a full Enterprise Application, whilst GCC is very often considered to be a hobbyists tool. Nevertheless, GCC is used in enterprise environments; in fact, many such tools seem to be making their way into the Enterprise: where once upon a time ClearCase and Oracle were clearly dominant, Git and MySQL are making headway.

Time was an Enterprise System was a one-stop-shop for everything from the Operating System and hardware up to all the applications. The tail-end of the 20th century saw a move from mainframe to personal computers in use by big business, with the result that applications were provided by many people (although Windows and Microsoft tools had a clear lead). The 21st century seems to be showing a move from commercial, 'Enterprise-y' applications to increased dependence on OSS tools and custom environments. The rise of the web application developed by start-up companies has been a driving force for this, I believe. Perhaps 'The Enterprise' saw the benefits of using – and even contributing to – software that could be downloaded for free, and perhaps even perceived a value-for-money gap in the support contracts required for the Enterprise Solutions. I wonder what will be 'Enterprise-y' tomorrow?

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 25.5: 1st October 2013
C Vu 25.6: 1st December 2013

Overload 118: 1st November 2013
Overload 119: 1st January 2014

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

COPYRIGHTS AND TRADE MARKS

The Ethical Programmer

Pete Goodliffe follows his moral compass.

I might fairly reply to him, "You are mistaken, my friend, if you think that a man who is worth anything ought to spend his time weighing up the prospects of life and death. He has only one thing to consider in performing any action – that is, whether he is acting rightly or wrongly, like a good man or a bad one."
~ Socrates, *The Apology*

I often describe how the quality of a coder depends more on their *attitude* than their technical prowess. A recent conversation on this subject led me to consider the topic of *the ethical programmer*.

What does this mean? What does it look like? Do ethics even have a appreciable part to play in the programmer's life?

It's impossible to divorce the act of programming from any other part of the coder's human existence. So, naturally, ethical concerns govern what we, as programmers, do and how we relate to people professionally.

It stands to reason, then, that being an 'ethical programmer' is a worthwhile thing; at least as worthwhile as being an ethical *person*. You'd certainly worry about anyone who aspired to be an *unethical programmer*.

Many professions have specific ethical codes of conduct. The medical profession has the Hippocratic Oath, binding them to work for the benefit of their patients, and to not commit harm. Lawyers and engineers have their own professional bodies conferring chartered status, which require members to abide by certain rules of conduct. These ethical codes exist to protect their clients, to safeguard the practitioners, as well as to ensure the good name of the profession.

ethical codes exist to protect their clients, to safeguard the practitioners, as well as to ensure the good name of the profession

In software 'engineering' we have no such universal rules. There are few industry standards that we can be usefully accredited against. Various organisations publish their own crafted code of ethics, for example the ACM [1] and the BSI [2]. However, these have little legal standing, nor are they universally recognised.

The ethics of our work are largely guided by our own moral compass. There are certainly many great coders out there who work for the love of their craft and the advancement of the profession. There are also some shadier types who are playing the game predominantly for their own selfish gain. I've met both.

The subject of computer ethics was first coined by Walter Maner in the mid 1970s. Like other topics of ethical study, this is considered a branch of philosophy.

Working as an 'ethical' programmer has considerations in a number of areas: notably in our attitudes towards code, and towards people. There are also a bunch of legal issues that need to be understood. We'll look at these in the following sections.

Attitude to code

Do not write code that is deliberately hard to read, or designed in such a complex way that no one else can follow it.

We joke about this being a 'job security' scheme: writing code that only you can read will ensure you will never get fired! The ethical programmer knows that their job security lies in their talent, integrity, and value to a company, not in their ability to engineer the company to depend on them.

> Do not make yourself 'indispensable' by writing unreadable or unnecessary 'clever' code.

Do not 'fix' bugs by putting sticking-plaster workarounds or quick bodges in place, hiding one issue but leaving the door open for other variants of the problem to manifest. The ethical programmer finds the bug, understands it, and applies a proper, solid, tested fix. It's the 'professional' thing to do.

So, what happens if you're within a gnat's whisker of an unmovable deadline and you simply *have* to ship code, when you discover an awful, embarrassing, show-stopping bug? Is it ethical to apply a temporary quick-fix in order to rescue the imminent release? Perhaps. In this case, it may be a perfectly pragmatic solution. But the ethical programmer does not let it rest here: a new task is added to the work pool to track the 'technical debt' incurred and attempts to pay it off shortly after the software ships. These kinds of band-aid solution should not be left to fester any longer than necessary.

The ethical programmer aims to write the best code possible. At any point in time, work to the best of your ability. Employ the most appropriate tools and techniques that will lead to the best results, e.g.: use automated tests that ensure quality, pair programming and/or code review to catch mistakes and sharpen designs.

Legal issues

An ethical, professional programmer understands pertinent legal issues and ensures that they abide by the rules. Consider, for example, the thorny field of software licensing.

Do not use copyrighted code, like GPL [3] source, in proprietary code when the licence does not permit this.

> Honour software licences.

When changing jobs, do not take source code or technology from a old company and transplant parts of it into a new company. Or even show parts of it in an interview with another company.

This is an interesting topic, as it leads to a large grey area: copying private intellectual property or code that has a clear copyright notice is clearly stealing. However, we hire programmers based on their prior experience; the things they have done in the past. Is re-writing the same kind of code from memory, without duplicating exact source lines ethical? Is re-implementing another version of a proprietary algorithm that conferred competitive advantage unethical, if you've hired the designer of that algorithm specifically for their experience?

Often code is published online with a very liberal licence, merely asking for attribution. The ethical programmer takes care to make sure attribution is given appropriately for such code.

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

Ensure appropriate credit is given for work you reuse in your codebase.

If you know that there are legal issues surrounding some technology you're using, for example: encryption/decryption algorithms that are encumbered by trade restrictions, you have to make sure your work does not violate these laws.

Do not steal software, or use pirated development tools. If you are given a copy of an IDE make sure that there is a valid licence for you to use it. Just as you would not pirate a movie, or share copyrighted music online, you should not use illegally copied technical books.

Do not hack or crack your way into computers or information stores for which you do not have access authority. If you realise it's possible to access such a system, let the administrators know so they can remedy the permissions.

Next time

In the next installment, we'll look at ethical approaches to the other people we interface with. We'll consider the other developers we work with, our users, our managers, and our employers. Even ourselves!

And I'll try to construct a *Hippocodic Oath*. Stay tuned. ■

Questions

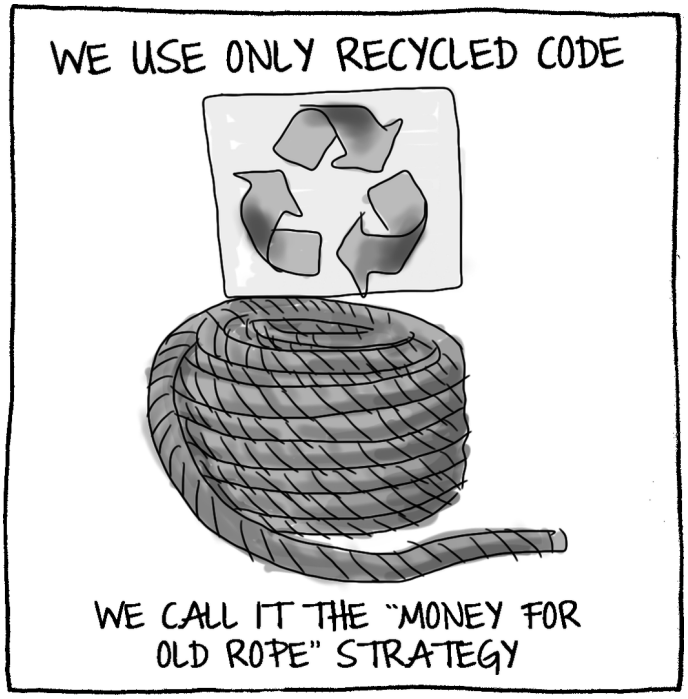
1. Do you consider yourself an 'ethical' programmer? Is there a difference between being an ethical programmer and an ethical person?
2. Do you agree or disagree with any of the observations above? Why?
3. Is it ethical to write software that makes bankers fabulously wealthy if the money they make comes at the expense of other people, who are not able to exploit the same computing power? Does it make a difference whether the trading practice is legal or not?
4. If your company is using GPL code in its proprietary products, but is not fulfilling the obligations of the licence terms (by withholding its own code), what should you do? Should you lobby for the licence terms to be met by open sourcing the company's code? Or should you ask for that GPL code to be replaced with a closed-source alternative? If the product has already shipped, should you be a 'whistle blower' and expose the licence violation? What if your job security depended on keeping quiet?

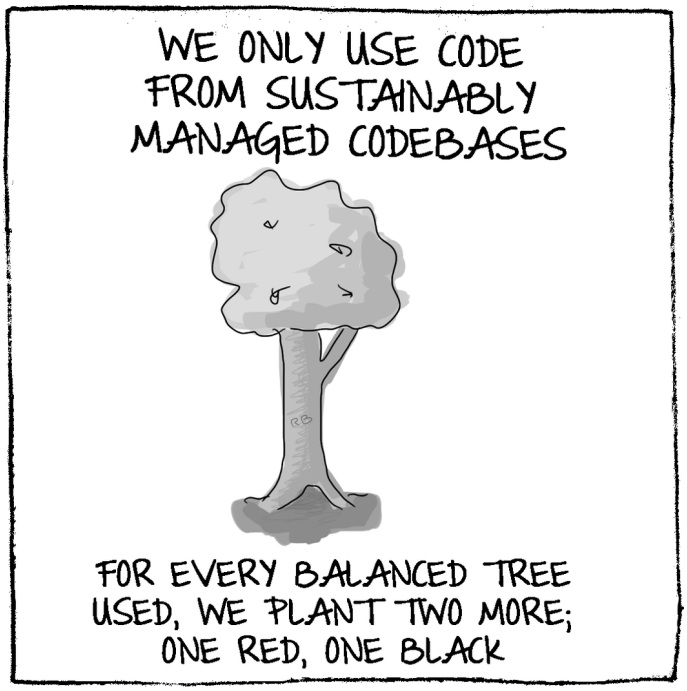
Acknowledgements

Many thanks to Steve Love and Emyr Williams for comments on drafts of this article.

References

[1] ACM Code of Ethics http://www.acm.org/about/code-of-ethics
[2] BSI code of Contact http://www.bcs.org/server.php?show=nav.6030
[3] The GNU Public License http://www.gnu.org/licenses/gpl.html





Two Sides of the Code

Vsevolod Vlaskine takes a critical look at the language of programming.

In this article, I try to characterize software engineering from the linguistic point of view. It is not a futile theoretical exercise, but a way to inform our daily programming practice with the help of simple and distilled concepts of modern language theory, which are rarely taught to software professionals.

Such a concept is a practice. Software code is the best illustration of it: by saying something it necessarily does something: any expression we write in python or C++ is meant to be executed.

Code as two texts

In human speech, John Austin calls such expressions performative [1], an idea he introduced in 1955. For example, by saying "I declare you man and wife", a priest actually does make a couple husband and wife. He performs an action just by saying the phrase.

Without going into details, many things we say just describe the situation, e.g. "it rains". By contrast, an expression in C or Java is written to perform an action. Even the data definitions, although seemingly descriptive, would be useless unless they eventually form 'words' in the vocabulary of executable expressions.

The code has two meanings: what it says and what it actually does. Code is two texts at the same time: one instructs the computer to do something, the other expresses what it is. The former is performative (it performs, does things), the latter expressive (it says, expresses things).

When a programmer writes `++n`, the code says that `n` is incremented by 1. It is correct, but meaningless. One will need to read through the whole function or `for`-loop to understand the purpose of incrementing `n`. It could possibly be improved by writing `++count` but there is still is a question: 'count' of what? But at least we are saying what we are doing: counting.

To make things worse, if one writes something like this:

```
int increment( int n ) { return n - 1; }
```

the code says one thing, but does another.

These examples are trivial, but they illustrate the common experience that each disconnection between expression and action makes the code slip into chaos at least at a polynomial rate, since the mess isn't additive, but multiplicative.

Laplace's daemon – software version

"It is all in my code. Don't ask me questions, just read it and you will see how it works." Throwing code over the fence is still typical in software teams.

Coding rules like "functions should not be longer than 10 (or whatever number of) lines" do not address the problem: the function works, but it does not say what it does. Or rather, it says: "My meaning is exactly the sum of my instructions. If you go through them one by one, you will know what I do." It is like saying that the structure of a building is a sum of its bricks on top of each other: as true as useless.

In 1814, Laplace suggested a thought-experiment: if someone (e.g. a superhuman daemon) knew the exact location and speed of all the particles in the world at a given moment, they would be able to exactly calculate any future state of the universe.

The determinism in physics expressed in Laplace's daemon argument would not work due to the laws of thermodynamics and quantum theory. However, in software engineering there still persists an illusion of determinism that when you know every single detail of a program, you will know what the system is doing. It is the software flavour of Laplace's daemon.

Many companies go through it: the founder feverishly produces the first cut of the system to get it out of the window. Then, if the start-up picks up, the founder becomes busy with the super-nova expansion and is hardly available for questions. Then, a bunch of engineers literally spend months or years digging through, reverse-engineering, and becoming experts in the twists and dark corners of the original code.

On a smaller scale, engineers often stop too early at decomposing their code. They give up when they split the functionality into components until they say: fine, even a child could understand it. However, it is not about size, but about semantics decomposition. If there still is functional coupling, or especially the need to look into implementation details to understand what the code is doing, the design job is not finished.

The reasons Laplace's daemon does not work in software engineering are in a way opposite to the ones in physics: it is not entropy or the uncertainty principle, but the complexity of language playing against the simplicity of mathematics or logic.

It is not all maths

It is not universal, but common to hear software engineers saying "software development is essentially a mathematical thing; projects are troubled and work frustrating, because management tries to apply business logic to maths, as well as because fresh graduates are not taught maths properly anymore."

These laments are true to an extent; indeed, engineers often struggle with basic mathematical concepts. However, the expectation that if the whole software enterprise could be organized like mathematical theories, it would bring rigour, efficiency, and quality into the trade is a big mistake. Maths is a large part, but not everything in software engineering.

Symbolic systems

Programming and software design essentially are a language enterprise.

In Louis Hjelmslev's terms [2], mathematics operates with symbolic systems, whereas programming deals with semiotic ones, and the latter cannot be reduced to the former.

The distinction is based on the fact that the reference of a sign (e.g. of a word) and the meaning of the sign are two different things.

In a symbolic system, symbols interact in the same way as what they symbolize (their references). For example, numbers symbolize countable objects (say, apples).

Symbols are like text labels in the zoo: the label says 'elephant' and we see an elephant inside of the cage. Instead of operating on the actual animals (e.g. counting them), we can use the labels on the cages.

That's how mathematics can be so powerfully applied: if it is reasonable to assume that elements in the 'real world' interact in the same way as the

VSEVOLOD VLASKINE

Vsevolod Vlaskine has over 15 years of programming experience. Currently, he leads a software team at the Australian Centre for Field Robotics, University of Sydney. He can be contacted at vsevolod.vlaskine@gmail.com

elements of a mathematical theory, then we could expect that all theorems of that theory will also be true in the 'real world'.

For example, natural numbers and their relationships map one-to-one into piles of apples and therefore all the theorems of number theory will apply to operations on piles of apples.

Semiotic systems

Unlike mathematics, which is based on symbols, semiotic systems (e.g. natural language) are based on signs (in this article, we use Hjelmslev's terms [3]).

Sign is a unity of two radically different things: its expressive part (for example, the way a word sounds) and its content [4]. The expressive system of a language has essentially different relationships between its parts from the system of its meanings.

These relationships cannot be reduced to a symbolic system: there is no direct symbol-referent mapping, since meanings operate in a system of relationships different from expressions: meaning of words vs sound of words; meaning of a novel vs its expressive elements (metaphors, text structures, etc); meaning of a software library (what it does) vs. its API (how it expresses it).

Object-oriented design is another example of a failure to tell symbolic and semiotic apart

By the way, it is a very common problem with researchers who become programmers: being used to symbolic systems from their scientific experience, in their code, they do not program, they model. They end up with highly inflexible closed implementations either endlessly parametrized or semantically rigid, which is the same at the end of the day. Something that may look like an immense number of degrees of freedom becomes almost a definition of coupling: if you need to set up and fine-tune dozens or hundreds of parameters to make your system work, it means that the system is semantically extremely hard to use. And since the meaning of the software system is its use (see 'Meaning is use'), 'hard to use' means 'meaningless', devoid of semantic flexibility, meaning only one thing: itself.

The rise and fall of object-oriented design

Object-oriented design is another example of a failure to tell symbolic and semiotic apart: in software engineering it has often been perceived as a sufficient model of the world, creating an illusion that any problem domain can be expressed purely in terms of classes, objects, and their relationships. Due to this mix-up of meaning and reference, OOD gets mistaken for a universal language (i.e. a semiotic system), whereas it was a symbolic one.

Essentially it is saying: class `elephant` or `car` in a Java or C++ program corresponds to elephants in a real zoo or cars on a real road. Then, to design any software system that can deal with elephants or cars, we simply express the relationships between the real things through the relationships between the corresponding classes. This makes OOD essentially a symbolic system.

However, the limitation of OOD is that we do not deal just with cars. In a single software system, a car could be a means of transportation, a physical mechanism, an asset, an obstacle, a consumer item, a registered entity, etc, all at once. How do we apply OOD? By having one over-sized class car? Or by having many interrelated classes belonging to various domains? Clearly, the latter.

But here we are: the sign 'car' interacts with other signs differently depending on what 'car' means in various contexts or problem domains. The system we want to build is semiotic, not symbolic.

Engineers were forced into OO designs which did not acknowledge the shortcomings of the method. Wherever the naive OOD went beyond its limitations (trying to solve 'semiotic' problems), it caused damage: over-engineered architectures; brainwashed teams; OO tools, largely abandoned now, that consumed huge resourses to be developed and sucked millions of dollars from the users in licence fees. I still think UML was a waste of time. Scores of ugly, coupled software systems were spawned, which either failed or, worse, companies are still trying to unscramble.

Software as text in artificial languages – and software development as collaboration in natural languages – are (implicitly or explicitly) based not only on maths, but heavily on the methods coming from linguistics and cannot be reduced to mathematics or another unified system.

Meaning is use

Continuing our example, what do we mean by 'car', when we design, say an automated vehicle? There are multiple domains of its meaning: car in the context of its mechanics, of fuel consumption, of route planning, of obstacle avoidance, of asset tracking, etc.

We would like to better understand the relationship between those meanings on one hand, as well as classes, libraries, and other software artifacts on the other.

To design a software system, we need to be clear about what we mean, when the system requirement document says 'car' or 'road', etc.

Ludwig Wittgenstein has influenced most of the modern philosophy and linguistics, arguing that generally in any language the meaning of its words or sentences is their use, and they do not have any meaning beyond that [5]. It turned a great deal of all traditional philosophy, logic, or psychology upside down.

Luckily, unlike humanities in general, in software engineering we can safely say the most (if not the only) important thing about a software system is how it will be used.

We can state for all the practical reasons, the meaning of any software system is its use. And therefore, Wittgenstein's ideas should be highly relevant to software.

Besides, earlier we figured out that a software system is a semiotic system, which has two sides, expressive and performative: what a piece of code says and what it does.

But what exactly does the code do?

Consider the following code snippet:

```
db::client client( host, port );
db::response response = client.send
  ( "SELECT name FROM address_book" );
```

We could say: the client class opens a socket, performs checks, sends the requests, polls the socket for the reponse, etc.

Or we could say: the code creates a database client, sends a request, and receives response.

The latter is what matters for us when we use the code. After all, the code is always an instrument, a means to some ends. For those who use it, it is not important what it does, but rather what they can do with it. And this is exactly the meaning of 'the code is its use' in Wittgenstein's terms.

On one hand, exposing implementation details destroys its meaning as its usage. Say, if after construction we had to call `client.connect()`, it would likely give out implementation details that are meaningless in the context of usage.

However, high-level usage is not always a hallmark of meaningful code, either. If it were true, encapsulation would solve all the problems, and the latter has been seriously overused in the old-style object-oriented design.

It is the same problem as saying that there is only one 'car' class. The actual car as an entity made of metal and plastic lures us into thinking so. However, 'car' can belong to multiple meaning/use domains (car as a vehicle, as an obstacle, as an asset, etc).

The meaning of a class is its use, i.e. the meaning is outside of the class and, thus, a strong force against encapsulation. This leads to the conclusion

Pen and Paper

Chris Oldwood finds uses for old-fashioned tools.

...when people say "bring a notebook" they mean "bring a laptop", not "bring a moleskine"
~ @natpryce

"Which is your favourite UML tool?" I swear by Crayola
~ @jasongorman

First thing in the morning at work, after turning on my machine and taking a few gulps of coffee, the next thing I do is get out my notepad and pen. Despite all the advances in hardware and software and the plethora of apps on the market designed to aid in every office task, I still regularly use the old fashioned method of pen and paper.

TODOs

By far the main use of my notebook is a list of things that either needs to be done or have caught my attention and so possibly deserve further thought or scrutiny at a later date. Sadly, when it comes to remembering what I'm (supposed to be) doing I'm like the proverbial goldfish – I find it all too easy to head off on a tangent and then lose focus on what really matters at that point. Although I have the ability to remember significant amounts of code I have written in my programming career I still can't make it to the top of the stairs at home without forgetting what it was I went up there… oh, look, shiny thing!

The sorts of things I end up making a note of are the questions (or quite possibly the WTFs [1]) that form in my mind as I'm working with both my own, and other people's code. I have yet to work in an environment where code reviews happen formally, regularly and cover 100% of the codebase. Instead I tend to see what's changed when I update my local copy, mostly out of self interest to see where any instability might arise, but also because I'm just a nosy person who likes to know what's going on.

Most times the kind of things that raise questions are fairly isolated and so can often be resolved pretty quickly, especially if the other developer is happy to take it up there and then. The alternative is a wider issue that involves looking across the entire codebase for similar cases. For example, catching every type of exception, e.g. `catch(…)` in C++ or `catch(Exception)` in C#, is a code smell for me when not done on a module boundary and so I'd reserve a little time to review the codebase and look for similar occurrences. I probably wouldn't want to take that on immediately, but I also don't want to forget it either.

Naturally when trying to work in an agile kind of way one is often mindful of eliminating any waste, and these little nuggets of work are no exception. My notebook acts as a temporary store to record my observations, but at a later date they can be handled in a variety of ways. It could be that someone else has picked up on it too and it's all resolved so I can just scribble out my note. Or perhaps what I initially thought was a smell turned out to be insignificant after all. At the other end of the scale are the items that have since mushroomed into a much larger task and so needs to be placed on the backlog as a formal piece of work – subject to further prioritisation of course. In between these two are the little things that aren't really important enough to distract me now but I'm not quite sure whether there is any real substance to them – superficially they might seem to have value, but perhaps the cost of change could only be justified as part of some other work.

CHRIS OLDWOOD

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

Two Sides of the Code (continued)

that there should be as little encapsulation as possible. One could also say: the class should be interpenetrable in as many semantic directions as possible. Excessive encapsulation blocks exactly this transparency. This blockage is the price of encapsulation.

Consider encapsulating connectivity, e.g. by using singletons or inversion of control, as handy as it may look, would be too much:

```
db::client client;
```

It hides the connectivity aspect of the client, which makes testing or changing the connection parameters very difficult. There is too much encapsulation.

The following expression:

```
db::client client( host, port );
```

hints that the client connects over a TCP socket. However, it exposes too much implementation detail.

A more meaningful way of fully exposing the aspects of the client (connectivity and database interface) might be:

```
db::client client( tcp::endpoint( host, port ) );
```

Conclusions

Linguistic concepts offer criteria to judge and distill design, code, and software process in general, looking at the software design of any scale as a semiotic system rather than symbolic one, and the code as a unity of its expression and use. This imposes strong requirements of semantic flexibility and transparency on design and code.

The clear cut between expressive and performative sides allows us to speak about the code more efficiently during the design and code reviews, plugging natural language into code-writing.

To put it simply: the craft of software engineering is to make the code to do what it says and to say what it does. ■

References

[1] Austin, J.L. *How to Do Things with Words*. Oxford, Clarendon Press, 1962

[2] Hjelmslev L., *Prolegomena to a Theory of Language.* The University of Wisconsin Press, Madison, 1963.

[3] Hjelmslev L., *Prolegomena to a Theory of Language.* The University of Wisconsin Press, Madison, 1963, pp. 111–114

[4] Saussure F. de, *Course in General Linguistics*.

[5] Wittgenstein L., *Philosophical Investigations*. Basil Blackwell, 1953

There was an interesting question raised on accu-general a few weeks back about tracking new tests that spring to mind whilst writing existing ones. In this particular instance I will use the code to record my thoughts, perhaps as a simple comment or more likely as a stub test case that shows up 'inconclusive' to make it more visible. If the test is for another class outside the scope of the current feature it goes in the notebook as that's probably a can of worms just waiting to be opened.

In essence the moment when I commit my changes is a natural barrier for me because it signifies the publishing of a single atomic, tested change in the codebase. If something needs to happen before the next commit I might as well just get on and do it. Otherwise it goes in the book and I can recalibrate my workload again afterwards. Assuming I don't break the build of course…

Diagrams

When I first discovered UML and started playing with a couple of Enterprise UML tools I, like many others, thought this was going to be a Big Thing. Of course it hasn't really panned out that way, but what I have got from it is a common vocabulary to use when knocking up diagrams. Martin Fowler uses the term 'sketches' in his *UML Distilled* book which I prefer as it takes a lot of the formality out of their use and recasts them as what they most usefully are – a conduit for further discussion.

What I particularly like about diagrams drawn in my notebook is that they also contain all the little amendments and annotations that occurred during the discussion. While the end result might be a nice snapshot of the destination, the journey often says far more about the contention and ideas that were explored along the way. Seeing the furiously drawn circles and underlines again is a great reminder on where the points of contention were and consequently where future confusion may lie.

Then there is just the immediacy of being able to flick back a few pages to the diagram in the middle of a discussion. Nothing kills the flow more than sitting, idly waiting for someone to hunt around in their emails looking for a picture. You could of course just draw it again, perhaps slightly differently this time so that the perspective changes. That in itself is a smell – constantly referring to or drawing the same picture – as it might mean the design is unintuitive.

If you're going to knock up a sketch it's better still if you have a couple of different coloured pens to hand too. Even just using black and red gives you a way to call out the more important aspects. Given the wage of an average programmer it costs very little to buy your own set of pens. You might need to keep them in a locked drawer though if you ever want them to remain in your possession for longer than a few days!

there is just the immediacy of being able to flick back a few pages to the diagram in the middle of a discussion

Support

Outside the usual run-of-the-mill development notes are the support ones. When in the heat of battle as I'm trying to resolve a production issue I need somewhere to jot down my observations. Often these are events that come from log files or specific data items like a unique customer ID. They often exist only transiently, appearing perhaps within the command line in my shell or a SQL query window. After the dust has settled I like to write-up a short post-mortem that explains the problem and contains the key data items in case a similar problem occurs again.

Anything that happens out of the ordinary in production has the hallmark of an edge case that will likely feed into some later development, such as a fix that requires a specific test case. Whilst a post-mortem email is readily searchable the margins of my notebook are often a quicker source when looking for some 'interesting data' to put a test case together.

Post script

My need to write things down was borne out of the frustration of forgetting things which had the undesirable consequence that I'd go off at a tangent and therefore disregard the real priority. Sadly it took me a very long time to realise that the act of writing down notes was what mattered, not that I had actually written them. It also probably explains why I can remember much of the code I have written. This turns out to be a well known phenomenon [2] that is possibly even more effective when handwritten than typed.

The most important part of this discovery though affects my personal life. Where before I used to rely, and fail miserably, on trying to remember the hints my wife would drop for her birthday and Christmas presents; now I surreptitiously jot them down on my smart phone under the guise of sending another banal tweet. The old fashioned approach definitely works better for me in the office but in the great outdoors I'm happy to bow down to the convenience of modern technology. ■

References

[1] http://thedailywtf.com/

[2] http://lifehacker.com/5738093/why-you-learn-more-effectively-by-writing-than-typing

Testing Times

Richard Polton looks at unit tests from a different perspective.

I'm going to make a wild stab in the dark here and suggest that, based on the title of this article, you're thinking "Yawn! Another pseud writing yet another article about TDD." Let me put your mind at rest. Whilst I advocate and practice TDD, this is not actually that song, this is just a tribute [1].

Before we start I ought to be clear that this is a speculative article and so there's not going to be any real code here that you can enter and run. It's a thought experiment for now that may at some future point become Real. What I want to talk about is automatically generating code from unit tests. I'm sure we've all seen the projects which claim to generate your unit test code for you automatically. There's a StackOverflow question [2] about this very subject. It may be true that it's a handy way of establishing a baseline in the case of an inherited codebase with no tests but that is not what we're here to talk about.

the aim of the exercise isn't building the parser, it's automatically generating the code

This article is going to concern itself with running the process in the other direction – from test to code. You may ask yourself why anyone would want to do that. For my part, it seems like a natural extension of TDD. First of all we decide what it is we wish to do (that is, implement code that satisfies the functional requirements passed to us), then we write a test which (eventually) leads to a minimal set of code that satisfies the test. So, in my mind, what I am imagining is a piece of test code that demonstrates some aspect of the system we aim to produce but which does not currently succeed. Then I am imagining some process that parses that code and extracts interfaces, classes, functions, maxima and minima and ranges of values, and so on. This automatic process could be built in to the IDE or exist as a very early step in the build process.

There are existing methods which go some of the way to achieve this aim but they are not especially common nor are they easy to use. The language Z [4], for example, exists in this domain but lies outside almost everyone's ken. I know I was interested in learning more about Z back in the day but much time has passed since then and I just haven't read any more about it, let alone learnt how to use it correctly. I would surmise that most people fall into the same group as me here, that is the group of people who have not had the time (or possibly inclination) to learn Z or another formal specification language.

Instead, then, let us consider what we might like an automated tool to do for us. Let us take a very simple unit test as an example.

```
[Test]
public void ThisIsATest1()
{
  var a = Mock<AnInterface>();
  Assert.IsTrue(a.DoIt());
}
```

This test expects two things, an interface and a specific function on that interface. The test will succeed when the function returns **`true`**. So if we were to build a parser to extract these pieces of information we would be looking for mocked interfaces, or possibly mocked abstract classes, and function calls from those interfaces. Conceptually, we should be able to extract these data from the test. We could parse the source code as it is presented to the compiler but why do all the heavy lifting ourselves? Let's suppose that the unit tests have been compiled and run at least once. In this case we have assemblies containing the compiled code which we can load into our generator and parse. Using reflection therefore, we could load the assembly that contains the tests and then parse each test looking for mocked interfaces. I imagine building up a dictionary of interfaces names and the variable names associated with them, a symbol table if you will. We would also be looking for function calls from the variable. In addition to the function name we would need to extract the expected return type and the types in the parameter list. I imagine that much of this could be made available by the Microsoft Compiler project (Roslyn [3]) but if it is then that will be the subject of a future article; one in which we present actual working code. Indeed, this work does seem to be the sort of thing that a compiler might do.

Okay, so let us suppose we can associate variable names and interface names. Also let us suppose we have some dynamic object that represents each of the functions with the names, return types and parameter list stored appropriately which we then associate in some manner with each interface name. One other piece of information we can infer from the test is the expected relationship between input data and output data, i.e. what data we need to provide in order to cause the test either to succeed (as we have an **`IsTrue`** assertion in our example) or fail. In the example above we have a very simple relationship: no input data are required at all and the result is always, insofar as we can determine at this point, true.

Conceptually we probably have enough understanding of the form the data might take but, at this stage, we don't have a parser and writing and presenting one would detract from the time and space available to talk about our code generator.

Practically, to help with the thought experiment, we might consider annotating the test code to help us out. After all, the aim of the exercise isn't building the parser, it's automatically generating the code. Let's use attributes **`Interface`** and **`Function`** which we expect to behave in the following manner

We would like the `Function` attribute to lead to a code template of the form

```
<ReturnType> <FunctionName>(<ParameterList>)
{
  return <ExpectedType>;
}
```

and for the functions thus produced to be contained within an interface or class (depending on whether we are producing declarations or function bodies). The **`Interface`** attribute, therefore, should produce an interface declaration such as

```
public interface <InterfaceName>
{
}
```

and, presumably, a class that implements it and contains the function body described above. If we now rewrite our test code using these attributes we have Listing 1.

Then we extract the attributes from the test code and use them to build our program code. I think that **`Receives`** and **`Returns`** are probably

RICHARD POLTON

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

Listing 1

```
[Test]
public void ThisIsATest1()
{
  [Interface("AnInterface")]
  var a = Mock<AnInterface>();

  [Function("DoIt"),Receives(),
     Returns(Boolean),Expects(true)]
  Assert.IsTrue(a.DoIt());
}
```

Listing 2

```
public interface AnInterface
{
  Boolean DoIt();
}

public class AClassThatImplements : AnInterface
{
  public Boolean DoIt()
  {
    return true;
  }
}
```

superfluous because the types should be obtainable from the **Expects** attribute but we'll keep them both for the nonce. We would like our generator to take the above test code and to produce interface and class definitions like Listing 2.

This is nice and simple and also highlights a point which I'm sure we all come across regularly. Tests are of little value if they only verify the success criteria. If we produced code which only asserted that a function returned **true** then we might as well just have a function that does no more than return **true**. It satisifies the functional requirements (that the Expectation is **true**) and passes the tests after all.

Okay, so that's simple enough. Now we can make it a bit more involved by passing a parameter into the function to be tested. Let's suppose that the function is a simple predicate – give it one input value and expect **true** anything else returns **false**. How should we specify the expected output of **Verify** though? In a declarative model, I propose that we would specify the rule as the expectation. Therefore, we will write an Expression as the **Expects** attribute value with the understanding that the number, position and type of the parameters corresponds to those of the function under test. (See Listing 3.)

Our generator will then produce something like Listing 4.

Can we define something more complex? Why not? How about passing more than one parameter into **Verify**? Or caching some state in the constructor which is used in **Verify**?

Passing more parameters into our test function shouldn't be too different from the preceding work. We change the **Receives** attribute to take additional types, for example,

```
[Function("Verify2"),Receives(int,int),
  Returns(Boolean),Expects((a,b)=>a==b)]
Assert.IsTrue(a.Verify2(1,1));
Assert.IsFalse(a.Verify2(1,0));
```

Note, however, that the rule used in the **Expects** attribute is not entirely satisified by the assertions. Again, this is not uncommon in practice and is something that we should strive to avoid. Our two assertions prove that the function **Verify2** only works as specified in two cases and is something I would expect a code coverage tool to bring to our attention. Let's look at the code our generator might produce for the above test case.

```
public Boolean Verify2(int a, int b)
{
  return a==b;
}
```

That is well and good for now, because we specified the functionality expected in our attributes, but in the ideal implementation of this process, we would expect the application itself to determine the range of values that result in a truth response. Had we presented only the assertions it would not be possible for a generator to deduce anything beyond **Verify2(1,1)** should return **true** and **Verify2(1,0)** should return **false**. In other words, the tests presented do not describe the system under test.

Let us consider some other assertions. If our unit test contained **Assert.IsGreaterThan** or **Assert.IsLessThan** then our generator ought to be able to produce program code to suit. That is,

```
[Function("GetValue"),Receives(),
  Returns(Boolean),Expects(a=>10<a && a<100)]
Assert.IsGreaterThan(10, a.GetValue());
Assert.IsLessThan(100, a.GetValue());
```

Listing 3

```
[Test]
public void ThisIsATest2()
{
  [Interface("AnInterface")]
  var a = Mock<AnInterface>();

  [Function("Verify"),Receives(int),
    Returns(Boolean),Expects(a=>a==1)]
  Assert.IsTrue(a.Verify(1));
  Assert.IsFalse(a.Verify(2));
}
```

Listing 4

```
public interface AnInterface
{
  Boolean Verify(int a);
}

public AClassThatImplements : AnInterface
{
  public Boolean Verify(int a)
  {
    return a==1;
  }
}
```

Listing 5

```
[Test]
public void ThisIsATest2()
{
  [Interface("AnInterface")]
  var a = Mock<AnInterface>(10);
  [Function("Verify"),Receives(int),
    Returns(Boolean),Expects(a=>a==State1)]
  Assert.IsTrue(a.Verify(10));
  Assert.IsFalse(a.Verify(2));
}
```

Listing 6

```
public class AClassThatImplements : AnInterface
{
  public int State {get; private set;}
  public AClassThatImplements(int a)
  {
    State = a;
  }
  public Boolean Verify(int a)
  {
    return a==state;
  }
}
```

(Re)Reading the Classics

Chris Oldwood looks back at his favourite literature.

I am a product of the 1980s. Like some (many?) other ACCU members I cut my teeth on the 8-bit and 16-bit home micros that appeared during that decade. After an initial period learning BASIC from the bundled manual I soon realised the only true way to get anything decent out of these machines was to drop down to machine code. A lack of tooling at the time (probably due more to a lack of pocket money than products) meant hand-writing machine code and then poking it into place via a loader written in BASIC. The computer manuals of the time were very functional in nature, not in the paradigm sense of the word, but in the more traditional – they told you what the keywords did and perhaps gave you an example of how to use them. Any nods to Structured Programming were, I suspect, strictly for the professional programmer.

I spent seven years messing around with computers at home and school before eventually going to university to study Electronic Systems Engineering. During that time I did both an 'O' Level and an 'A' Level in 'Computer Science'. Despite my ability to program fast, smooth sprite-style animations in machine code, my grades were distinctly average. Apparently 'computers' are about more than just programming and so to get a decent grade you need to know a whole load of other stuff about batch processing, punched cards, governance and a host of other dreary topics. The fact that my sister, who knew practically zilch about programming, got an A in both only went to show that Computer Studies (sic), was a pointless waste of time. The only things I remember learning in those classes were the cool tricks from the rich kids whose parents could afford to buy them a BBC Micro. Oh, and Pascal, well, enough of it to do my A-Level project, but it seemed a neat language in a quaint sort of way.

As I mentioned recently [1], my decision to study analogue electronics turned out to be an incredibly poor choice. I had far more success with digital electronics because I could just stitch the building blocks together the same way that I would compose functions together in code. The course contained a few options on Software Engineering, which I selected, but to be honest C++ just seemed like a massively bloated extension to C. At least *that* seemed like a slightly more efficient way to write assembler. We covered Object-Orientation too, but it took watching the lab assistant turn our simple text-based 'shapes' C++ assignment into an X Window based version of Asteroids to really grasp the power of some of the concepts. However, no one appeared to doing any System Programming in C++, so it all felt quite literally academic.

Of course the real education starts when you leave academia, get a job and have to deliver real code to real customers. Fortunately my first job was with a small company that produced PC based desktop publishing software where quality was high on the agenda, even if the magazine reviewers didn't appreciate it. That meant continuous learning was an established practice brought about, in part, by subscriptions to various publications, such as *Dr Dobbs Journal*, *Windows Developer Journal*, *Microsoft Systems Journal* and *C/C++ Users Journal*. Certain key books, such as Steve McConnell's *Code Complete* and Steve Maguire's *Writing Solid Code* became mandatory reading as they tried to deal with the complexity of a codebase that had evolved over many years into different products and changed platforms along the way too.

the real education starts when you leave academia, get a job and have to deliver real code to real customers

Once I started freelancing I had to replace all that myself. But the necessity allowed me to discover *Application Development Advisor* where I first encountered the writings of one Kevlin Henney. *C++ Report* surfaced on my radar too, just before its demise! At least some of those articles found their way into book form under the guise of *C++ Gems*. Around the same time Scott Meyers had kicked off the *Effective C++* series which led to the style of books where advice was broken down into a number of 'items' – most notably the *Exceptional C++* series (Herb Sutter) and *C++ Common Knowledge* (Stephen Dewhurst). They were great books because they started to highlight the difference between a program that was perhaps technically correct, but ultimately not maintainable due to the dark corners of the language that may have been unintentionally exploited.

The other two sorts of books I lapped up were the *Design Patterns* series and the 'Haynes Manuals' of systems programming. The former, made famous by the Gang of Four's Design Patterns, were the two big series:

CHRIS OLDWOOD

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

Testing Times (continued)

How about state? If, in a similar way to the tokens in the `Expects` Expression, we use a magic token `State1` in our attributes which we interpret to be the first parameter that we pass to the constructor then we could write Listing 5, which could lead to program code of the form of Listing 6.

Although the goal of this article was to ramble on about this code generation idea I had, as the kinesis became taxis [5] it seems that the attributes we have introduced to make the parsing easier might be useful in another setting. Perhaps preconditions, postconditions and invariant conditions could also be implemented using this attribute scheme. That, however, I shall leave as an exercise for the reader. ■

References

[1] Tenacious D – Tribute http://www.youtube.com/watch?v=_lK4cX5xGiQ

[2] StackOverflow - auto-generation of .NET unit tests http://stackoverflow.com/questions/142481/auto-generation-of-net-unit-tests

[3] Roslyn http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx

[4] Z Formal Language – Specification of Computer Programmes http://en.wikipedia.org/wiki/Z_notation

[5] Taxis or Kinesis (I always loved these terms and what a splendid opportunity to use them ;-)) http://en.wikipedia.org/wiki/Kinesis_(biology)

Pattern Orientated Software Design (POSA) and *Pattern Languages of Program Design* (PLoPD). The latter were the various 'Undocumented' and 'Internals' books, such as *Undocumented Windows* (Andrew Schulman) and *Windows Internals* (Mark Russinovich). I always found myself working on projects where squeezing every last ounce from a box was the norm, despite the relative costs of developer versus hardware suggesting otherwise, and so I'd spent a lot of time learning everything I could about the platform and language I had always worked with.

And then in 2007 I joined ACCU [2].

Whilst I was clearly comfortable with the mechanics of producing software for my clients, I found myself beginning to question the more 'computer science-y' side. For instance I realised that I didn't really know what the differences were between classes and types, or functions, procedures and methods, or lambdas and closures, or aggregation and composition, etc. I had previously been throwing terms around without really knowing if I was using the correct one. There were also a variety laws and principles, such the Law of Demeter and SOLID that I knew what they stood for, and had an appreciation of what they were getting at but was sure I didn't fully understand all the forces involved. For example, The Liskov Substitution Principle (the L in SOLID) talks about types and not classes: does that matter?

I thought I knew what an Abstract Data Type was until I read his paper on them

A discussion on the ACCU channel at work led me to ask what the state of the art was in books on OO. I had a copy of *Object-Orientated Modelling & Design* by James Rumbaugh (1991) that I had rescued from a skip when moving offices around the turn of the millennium, but I had only skim read it. Somewhat to my surprise there wasn't really anything new. The books of the same era, such as Grady Booch's *Object Orientated Design* and Bertrand Meyer's *Object-Orientated Software Construction* were still the recommended reading. One slightly newer book (1996) that came out of the same discussion that I posted on accu-general was Arthur J. Riel's *Object-Orientated Design Heuristics*. The great thing about old books like these is that they're ten-a-penny in the online second hand book shops so it's not going to break the bank to own them all.

As I started reading them I also began to look up the references that they were citing, which in turn led me to even older books and published papers. Whilst I was aware that technology had obviously changed rapidly over the years, in contrast, the principles hadn't. There is always a danger though that those principles can become redundant without you realising it, such as the Law of the Big Three in C++ which has largely become a minor-issue since the invention of the reference-counted smart-pointer. It's also common for people to recite phrases such as 'premature optimisation is the root of all evil' without knowing the context in which it was framed. But do you need to know that, isn't it just 'obvious'? I don't think so, and I believe it's the reason Herb Sutter has the item on Premature Pessimization in his *C++ Coding Standards* book to highlight the effects of just glibly trotting out so called 'best practices'.

One name that came up frequently (apart from Edsger Dijkstra of course) was David Parnas, the man credited with identifying one of the three pillars of OO – encapsulation. Whilst many papers and articles of that earlier era are locked behind subscription based services I managed to uncover a book with a collection of his papers – *Software Fundamentals* – and it was going for about five quid. Parnas is a great writer and putting aside the actual content itself, which is gold dust, the style he writes in is worthy of attention alone. I thought I knew what an Abstract Data Type was until I read his paper on them. Also, reading about what is essentially exception handling in a paper dated back to the early 1970s is a great reminder of how timeless a lot of this stuff can be.

You can't go to an ACCU conference without hearing at least a dozen times about why we should be designing our code to exhibit low coupling and high cohesion. Tony Barrett-Powell went one further in 2010 [3] and did an extensive session all about it and many of the other related principles too. What was noticeable again was that the works he was referencing were also the older ones – for example books about Structured Programming. I found out that Meilir Page-Jones covers the subjects of coupling and cohesion pretty extensively in *Practical Guide to Structured Systems Design*, a book initially published in 1980! Kevlin Henney's more recent ACCU conference attempt to recast 'SOLID' as 'FLUID' just goes to show there's still so much more to unearth.

As I suggested earlier you have to be a little careful with what you read in some of these very old books as there is definitely advice that is a little dubious 30 to 40 years on. Whilst the programming side is still possibly valid, it's more the development process that you may find yourself wincing at. That said, the definitive classic book of software development is probably *The Mythical Man Month* by Fred Brooks. His 'No Silver Bullet' article must be pretty near the top of most referenced works. For me, though, 'The Tar Pit', which is the first chapter, is a great way to remind yourself why we get up in the morning to follow our chosen path. Meeting and chatting to Tim Lister at the ACCU conference two years ago instantly elevated his masterpiece, *Peopleware*, to the top of my reading pile. And with good reason because understanding the environment in which we work can have a significant effect on our productivity as well.

So, what of the output of modern day authors? Are there any books that I can see others discovering in a decade's time or that I'll find myself returning to again and again to see what new insights I can glean? One author that immediately springs to mind is Kent Beck. His books on *Extreme Programming*, *TDD by Example* and *Implementation Patterns* have already become a constant source of reference and, more importantly, inspiration, with the latter being one of my favourite programming books of all time. Whilst his books are short on pages they are jam packed with goodness and all written in a particularly comforting style.

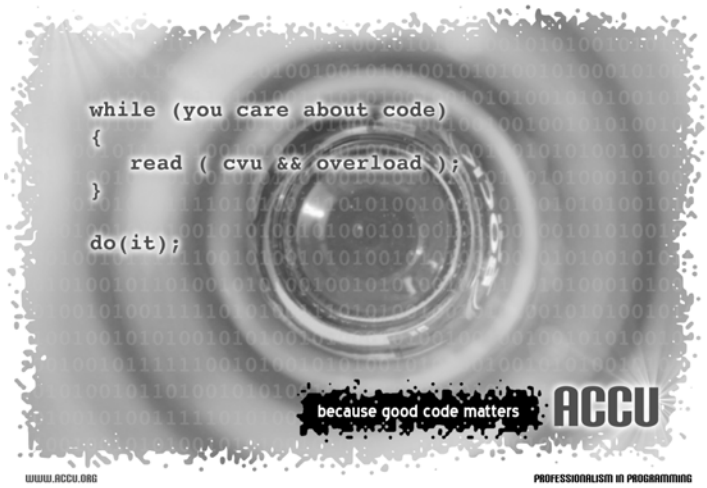
As I become more experienced and therefore more comfortable with being able to make software that 'works' I can devote more time to what it means to 'make it right'. Once upon a time I wanted books to get inside the heart of the machine or compiler to understand it, now I want books to get inside the head of the experts to understand what they 'see' and what makes them better programmers. ■

References

[1] 'Passionate About Programming or Passionate About Life?' – *C Vu,* July 2013

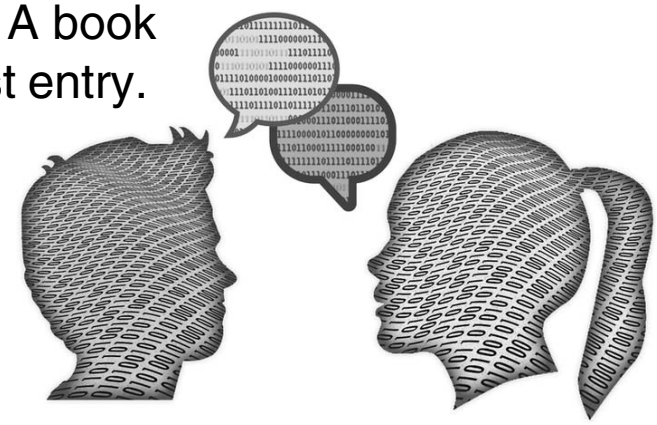
[2] 'The Downs and Ups of Being an ACCU Member' – *C Vu*, May 2013

[3] http://accu.org/content/conf2010/accu2010Coupling_TonyBarrettPowell.pdf



Code Critique Competition 83

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

This is a slightly different critique from the usual. The following code was presented to a number of people at the recent ISO C++ standards meeting and we were asked to work out what it would print. Please do so yourself – and *then* compile and run it. The critique part is to reflect on what happens, why it happens, and how to deal with it."

(My thanks to Alan Talbot for this interesting example.)

The code is in Listing 1.

Critiques

Paul Floyd <paulf@free.fr>

Indeed very interesting. I'm sure I've seen similar examples, but nonetheless I fell straight into the trap.

On reading the code, what I wanted it to do was

```
l l(42);
```

to construct an instance of `l` on the stack with the user defined constructor and to initialise `l`'s private member `x` to `42` in the constructor init list.

```
l.load();
```

Obviously this calls `l`'s load member function.

Listing 1

```
#include <iostream>

int x;
struct i
{
  i() { x = 0; }
  i(int i) { x = i; }
};

class l
{
public:
  l(int i) : x(i) {}
  void load() {
    i(x);
  }

private:
  int x;
};

int main()
{
  l l(42);
  l.load();
  std::cout << x << std::endl;
}
```

```
i(x);
```

Well, that looks like it's constructing an instance of `i` on the stack and passing it the member `x`. The `i` constructor that takes an `int` uses it to set the global `x`. Since `l.x` was constructed to be 42, that's what `cout` displays.

Of course, this is wrong. Let's compile. GCC 4.2 and 4.6 g++ say nothing. MS VC++ 2010 just makes a lot of noise abut the standard headers when compiling with /Wall. clang++ is more helpful:

```
/usr/bin/clang++ -Wall -Wextra -Weffc++ -o cc82
cc82.cpp -g
cc82.cpp:19:6: warning: private field 'x' is not
used [-Wunused-private-field]
 int x;
     ^
1 warning generated.
```

Eh? Oracle Solaris Studio also gives a hint as to what is up.

```
CC +w2 -g -o cc82 cc82.cpp
"cc82.cpp", line 16: Warning: x hides l::x.
```

Debugging the code, I saw that it stepped in to the constructor with `i` that takes no arguments. The light dawned.

```
i(x);
```

is not a constructor taking `x`. It's declaring a local variable `x` of type `i`, the same as `i x;`. Curse the C and C++ declaration syntax! This uses the `i` constructor without arguments

```
i() { x = 0; }
```

which sets global `x` to 0, which is what `cout` displays.

So we have 3 `x`'s. Global `x`, private `x` and local `x`. This explains the compiler warnings. `private x` is initialised but never read, and local `x` hides `private x;`

How to deal with it? Well, my first though on seeing global `x` being assigned values from `i`'s constructor was 'yuck'. Not doing that would be a good start (and by that I mean using global variables and using a constructor in a way that has side effects other than constructing). `x` is crying out to be a member of `i` (regular or static, depending on the needs). Next, as noted, using more than one compiler can be helpful. GCC wasn't much use. The next thing to do would be to avoid having three different

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

`x`'s. Let's imagine that we want to keep the broken design and just want it to compile and output 42. How to make it 'just do it'?

What we want for that is

```
i local_x(private_x);
```

So let's see what

```
i x(x);
```

does.

```
cc82.cpp:16:8: warning: variable 'x' is
uninitialized when used within its own
      initialization [-Wuninitialized]
   i x(x);
     ~ ^
cc82.cpp:19:6: warning: private field 'x' is not
used [-Wunused-private-field]
 int x;
     ^
```

Obviously not what we wanted.

Let's try being a bit more explicit.

```
i x(this->x);
```

No compiler warnings and the output is 42.

Let's try avoiding the name collisions with just

```
i foo(x);
```

Also compiles and outputs 42.

In summary, this example shows the importance of being able to understand C and C++ declaration syntax and the rules used for symbol resolution.

PS I wanted to have a play with cdecl. This is a tool, originally written I believe by Peter van der Linden (author of *Expert C Programming*), that takes a C declaration and converts it into a more or less human readable form. There's a C++ version as well, but I couldn't find an online version of that. There seem to be several online versions of cdecl, for instance, http://cdecl.org.

If I give it `i(x)` then it says it is a syntax error. On the other hand, if I simply substitute `i` for something that it knows, say `int`, then `int(x)` gives

```
declare x as int
```

Substituting `i` back into `int`, as it were, means that `i(x)` means 'declare `x` as `i`'.

Phil Nash <phil.nash@cibc.co.uk>

I was initially stumped too, until I started playing around with it to reveal the cracks.

Obviously it prints 0, where you might expect it to print 42.

This is because the expression:

```
i(x);
```

is not what you might think it is.

It looks like it's constructing an anonymous instance of type `i` and calling the constructor from `int` on it, passing the value of the member variable, `x`, right?

Wrong.

It's declaring a variable of type `i` ... and giving it the name `x` (which is a local name that shadows the `x` used as a member variable).

It becomes a little more obvious if you add a space in there:

```
i (x);
```

The parentheses are redundant here and this is the same as writing:

```
i x;
```

Peter Sommerlad <peter.sommerlad@hsr.ch>

This is a classic and one of the reasons I am such a fan of the C++11 (almost) uniform initializer syntax.

In my C++ class (before C++11) the same problem happened often when I asked to initialize a standard library container from a pair of `istream` iterators as follows. A default-constructed `std::istream_iterator` is used as the end iterator and a programmer might try to fill a vector from `std::cin` like this:

```
using namespace std;
vector<int> v(istream_iterator<int>(cin),
              istream_iterator<int>());
```

However, that doesn't define a vector variable `v`, but....

...declares a function named `v` returning a `vector<int>` and taking two `istream_iterator<int>` as arguments (the first one named `cin`, but not really relevant in a function declaration). WTF... (as Russel Winder posted on accu-general).

When learning C or C++, we might have met the old friend function pointer and that required parentheses in its declaration to bind the asterisk to the name of the pointer instead of the function call parentheses, because the latter have a higher priority, e.g., `void (*fp)();`

This leads me to the explanation that parentheses for grouping are not only allowed in regular expressions like `a*(b+c)`, but also in declarators. The corresponding grammar rules are given in chapter 8 [dcl.decl] of the C++ standard (2011) in paragraph 4 (excerpt):

> declarator: ptr-declarator
>
> ptr-declarator: noptr-declarator
>
> noptr-declarator: (ptr-declarator)

This means we can put potentially superfluous parentheses around every declarator we see in a program.

The second grammar aspect of the ambiguity between expressions and declarations is explained in chapter 6.8[stmt.ambig] of the standard:

> An expression-statement with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a (.

and it resolves it in favour of the declaration:

> In those cases the statement is a declaration.

Now let us get back to the example of the code critique. If we look at member function `l::load()` we see:

```
void load() {
  i(x);
}
```

For a naive first time reader this looks like a conversion expression constructing an object of type `i` from the member variable `x` that was initialized in `main` with 42. However, what the compiler really reads is something we could have written as

```
i x;
```

by removing the superfluous parentheses and this is obviously defining a local variable of type `i` using its default constructor and naming it `x`. And because of the above mentioned rules the default constructor of `i` sets the global variable `x` to zero, not to 42 as many first thought would be the result.

Now using braces for initializing allows us to construct what might have been intended by the author without any syntactical ambiguity:

```
void load() {
  i{x};
}
```

or using my example from above:

```
using namespace std;
vector<int> v{istream_iterator<int>{cin},
              istream_iterator<int>{}};
```

However, this usage of uniform initialization syntax also has its 'gotchas'. For example, `std::vector<int>` can be initialized with `n` elements of a given value, but

```
vector<int> tenfourtytwos{10,42};
```

will not give us ten elements intialized with 42, but a vector with two elements initialized with 10 and 42, because the `initializer_list` constructor takes precedence over other overloads when the values given between the curly braces can be interpreted as an initializer list. In these cases we have to step back to use the (potentially ambiguous) parentheses syntax for initializing our vector:

```
vector<int> tenfourtytwos(10,42);
```

If you can not use C++11 syntax, you can guarantee the interpretation as an expression by putting your intended value creation into an expression, i.e., using a comma operator:

```
void load() {
  0,i(x);
}
```

but that is ugly.

So what can we remember from it:

Use C++11 uniform initialization syntax whenever possible, avoid constructing values that you will not use and sidestep side effects (aka Monads in Haskell) whenever possible.

In general one can add the typical SCC blaming that the given code uses many bad practices:

- bad names for types and variables. I guess they have been intentionally obfuscated.
- global variable and using side-effects of object construction to manipulate it
- not thread safe, because of side-effects and the global variable
- unsuspecting side effect, because of the declaration of local variable `x` in `load()`

Mirko Stocker <me@misto.ch>

I found this issue of the code critique hanging by the coffee machine in our office. Cleverly placed at this strategic location by Peter Sommerlad, it didn't take long to catch my attention. Now, my C++ is a bit rusty, but after staring at it for a few minutes, I came to the conclusion that the output one would expect is 42. But there must be something else going on, why else would it be worthy of the attention of the fine ladies and gentlemen of the C++ committee?

A second cup of coffee didn't reveal any more insights, so I took the code to my favourite C++ IDE – Eclipse CDT – spiced up with some of our plug-ins. As expected with this kind of obfuscated code, Linticator [1] shows several warnings about shadowed declarations (see screenshot below).

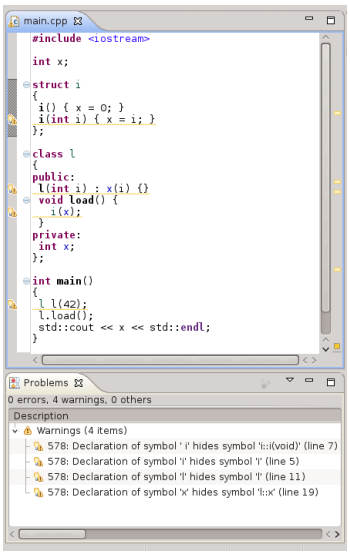


Figure 1

The last one makes me pause: declaration of symbol 'x' hides symbol 'l::x'. Declaration? Of course, that's the solution! 'i(x)' is a variable declaration of type 'i' named 'x', and not a constructor call. This will set the value of 'x' to 0 and not 42.

How can we deal with these pitfalls in the language? By using state of the art tools like Linticator and other static analysis software that make our lives a little bit easier.

Reference

[1] http://linticator.com

Balog Pal <pasa@lib.hu>

The provided code presses hard to abuse name lookup, reusing names with the same identifier i and l hoping it is still valid due to hiding rules. It actually succeeds in some places, say exploiting that the name of a function argument hides properly. (Interesting that we even missed one usual trick, constructor of l could name the argument x and in the initializer list x(x) would have the proper behavior in isolation...)

However it went one step too far. In the standard we have this rule:

> 3.3.7 Class scope [basic.scope.class]
>
> The following rules describe the scope of names declared in classes.
>
> 1) The potential scope of a name declared in a class consists not only of the declarative region following the name's point of declaration, but also of all function bodies, default arguments, exception-specifications, and brace-or-equal-initializers of non-static data members in that class (including such things in nested classes).
>
> 2) A name N used in a class S shall refer to the same declaration in its context and when re-evaluated in the completed scope of S. No diagnostic is required for a violation of this rule.
>
> 3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program is ill-formed, no diagnostic is required.

We have `x` at the global scope and also as member in l. At definition of `l::load` `x` is used and fits description of bullets 2) and 3). So this program is ill-formed, but the compiler is allowed to not even tell about it. [Ed: I don't believe these rules quite apply to this case.]

The answer to 'what it would print' then is kind-of moot. A better quality compiler would still issue the diagnostic but not produce anything to execute. For the rest, as behavior is only defined for well-formed programs, anything can happen, including printing something or nothing, or even posting the correct answer to the Code Critique problem.

Dealing with it is simple: don't abuse name lookup. Actually the main issue is trivial to mitigate with some naming convention, like naming global/namespace variables with a `g_` prefix and forbidding that form for anything else. Even so we can still construct a problematic case with anonymous namespaces or global types clashing with data members, and carefully choosing expressions that still work (maybe adding `decltype()` to the mix), but the problem potential will get too low to worry about.

Commentary

I think most of those who read this critique and tried to work out what it would print got a surprise. As most people realised the trouble is the declaration of the local variable `x` inside the `load()` function provides a superfluous pair of brackets, which to the reader – but *not* the compiler – make the call look like a constructor call. This is a relation of the 'most vexing parse' Peter refers to where the writer intends a variable declaration but produces a function declaration.

The fundamental problem we face here is realising that what the code *actually* means is not what we *think* it should. (I'm not sure whether the failure of many members of the ISO C++ standards body to recognise the problem with the breaking code makes things seem better or worse.)

This is where static analysis wins out … except when it doesn't. The difficulty with this code is deciding *what* is actually wrong with it. The

Two Pence Worth

An opportunity to share your pearls of wisdom with us.

One of the marvellous things about being part of an organisation like ACCU is that people are always willing to help out and put in their two-pence-worth of advice. In this new section of CVu, we'll capture some of those gems and print the best ones. If you have your own 2p to add to the collective wisdom of the group, send it to cvu@accu.org.

"Avoid spelling mistakes in your method names by using operator overloading instead."

"Reduce cyclomatic complexity by writing all your code as LINQ expressions."

"Make debugging easier by adding an implicit conversion to `const char*` to all your classes."

"Eliminate compiler errors by writing your code in a scripting language instead."

"Make sure you build want the customer wants by asking them what they want."

"Avoid building the wrong thing by gathering the requirements up-front and documenting them."

"Improve your chances of getting a job programming by listing every technology you've heard of on your CV."

"Avoid subtle performance problems caused by False Sharing by adding 64 bytes of padding between each data member."

"If you're going to add a singleton, please think about it as long as it takes to change your mind." rzeh, Chicago

Code Critique Competition 83 (continued)

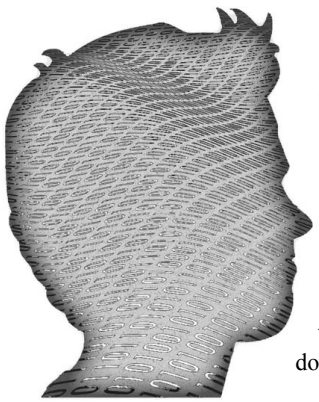
variable is unused, but since it is a user-defined type whose constructor has side-effects most tools seem to assume this is why the variable was declared in the first place. If this were to cause a warning many uses of, for instance, RAII helper objects would also cause (erroneous) warnings.

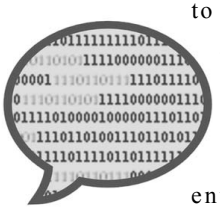
Clang warns because the private field **x** is not used – in a bigger example this might no longer be the case. Oracle CC warns about the local variable hiding the private member, which is the most useful compiler-generated warning anyone seems to have found. Perhaps Linticator deserves the prize for the most useful warnings?

Peter Sommerlad's advice to prefer brace-initialisation works well for writing solely in C++11 (and beyond) but sadly many C++ programmers cannot yet adopt the new language standard unreservedly in their working environment.

The winner of CC 82

The various entries covered pretty well everything I could think of about this code. I think on balance Paul Floyd covered the topic best – I found his explanation clear and thought his use of multiple compilers was a sensible suggestion. I also liked his reference to cdecl since C++ inherited the parse problem from C (although since C lacks constructors it is harder to come up with plausible breaking examples…)





Thanks to all who sent in entries and commented on accu-general. I hope no-one has given up on C++ (or even programming at all) after their initial reaction on seeing the code and trying to understand why it does what it does!

Listing 2

```
import random
x=range(1,23,1)
y=random.sample(x,11)
y=sorted(y)
for i in range(2,11,1):
    if y[i]-y[i-2]==2 or y[i]-y[i-1]>3 or \
       y[1]-y[0]>3:continue
print y
```

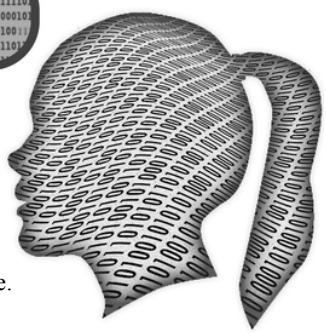
Code critique 83

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

After demolishing everyone's confidence of their knowledge of C++ in CC82 I thought it was time for some Python….

I want to split the set {1,2,3,...,21,22} into two disjoint subsets of 11 numbers such that neither subset contains three consecutive numbers. The following was an attempt to produce one possible subset. Having got an ordered sample of 11 from {1,2,3,...,22} I check that the sample does not contain three consecutive numbers (y[i]-y[i-2] is never 2), nor are there three consecutive numbers **not** in the sample (y[i]-y[i-1] is never greater than 3). But it produces samples meeting *neither* condition. Help!" (Yes, the problem with the *code* is fairly easy – but how best can you help the *programmer*?)

The code is in Listing 2.



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.

Standards Report

Mark Radford reports on the latest from C++14 Standardisation.

This is the time in the C++ standards process that has a fairly quiet feel to it. The Bristol ISO meeting seems to be a long time ago (well, four months, anyway), and the next meeting, in Chicago, is not until late September. However, things do happen when there are no ISO meetings: the interim mailing is available [1], the BSI Panel met on Monday 29th July, and SG1 (Concurrency and Parallelism) met on July 25th and 26th in Santa Clara, CA, in the US.

The interim mailing contains considerably fewer technical papers than usual, but it does contain the committee draft for C++14 (N3690). Another paper of note in the interim mailing is the working draft of the Technical Specification (TS) for the File System (N3693). I talked a bit about the role of the TS in my post-Bristol report when I talked about Constraints a.k.a. 'Concepts Lite', because that is what SG8 (Concepts) are working to produce, and SG3 (File System) are working towards the same product i.e. a TS. Note that this is the first appearance of the file system TS draft. N3693 says it revises N3505, however N3505 was a library proposal, whereas N3693 moves the paper forward to be the working draft of the TS.

The committee draft (CD) is the draft on which national bodies are balloted. Further, national bodies get to make comments on the CD, so it is the first point at which the standards committee gets feedback on its work. The ballot is a yes/no vote, but a yes vote can also be accompanied by comments. The C++14 CD was what the BSI Panel were concerned with (at the 29th July meeting). The UK's current intention is to vote yes to the CD, but with accompanying comments. The idea is that the changes indicated by our comments will be incorporated into the next draft. Voting yes at this stage means that our comments do not hold up progress to the next stage. If we make a comment that is not acted upon, and we feel strongly about it, we can vote no to the draft at the next stage. However, it is unlikely that a comment will be ignored, at least without satisfactory justification. In passing, the ISO C++ web site has a page explaining ISO procedures including stages and ballots [2].

The meeting on July 29th was taken up with going through, and reviewing, comments submitted by BSI Panel members. Following the review, a comment may or may not be accepted (by the BSI Panel as a whole) for being formally put forward to ISO (via the BSI channels). As it happened, the meeting also resulted in a few more comments, as a result of the discussion arising from those already submitted.

One topic that just won't go away is that of `std::future`'s destructor potentially blocking, if the future was returned from `std::async`. That is, the destructor blocks until the underlying thread behind the `std::async` call completes. This causes problems that I talked about in my previous report, and at the Bristol meeting there was a move to introduce changes to fix things (N3637). However, the proposed changes would break existing code and therefore some controversy has surrounded the N3637 proposal. At the end of the Bristol meeting N3637 was voted on by all the national bodies represented to ISO, the result being that there was no clear consensus. The motion (to incorporate the N3637 proposals into the working paper) was therefore not carried. The UK has made a ballot comment asking for several notes to be added to the standard, clarifying the current behaviour.

This topic was also discussed by SG1 at their Santa Clara meeting. I see from the minutes that SG1 still cares very much about finding a solution to this problem. Various options were discussed: one idea that was put forward is just to deprecate `std::async` as early as C++14, and introduce a replacement that doesn't have the problems associated with `std::async`. I'm assuming the motivation (for deprecation) is to allow time for code to be migrated to the replacement. Actually, deprecated features frequently never get removed (or at least not for a long time). The point really, is that deprecation sends a clear message that the feature should not be used in any new code. It is important to understand, that for the time being this is just discussion, and no decision has been taken (the next opportunity to make a firm decision is the Chicago ISO meeting).

The minutes of the Santa Clara meeting have not yet been published in a mailing, as the interim mailing (see above) came out before the meeting; I'm assuming these minutes will be in the next mailing (but I don't know if there will be another interim mailing, so the next one may be pre-Chicago).

References

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/#mailing2013-07

[2] http://isocpp.org/std/iso-iec-jtc1-procedures

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

The Reward

Richard Polton writes in suggesting how we might encourage good code.

I am a great believer in patterns of reuse, and thereby code reduction, as anyone who has seen my code could testify. Similarly, I am also a strong believer in test-first development. I preach both of these to anyone who will listen and probably too much ;-)

We find ourselves in an environment which often fosters a 'more is more' attitude and maintains that bigger is better. Even nowadays, when we have corporate recognition of Open Source code, I still hear development managers talking about lines of code as if it were a measure of success. Rewards are handed out accordingly. As time marches on, we make additive changes to our system and so they become more complex and, ultimately, literally incomprehensible.

I had a particularly bad experience of this nature at one place of employment some years back. They had a system that was critical and yet so dire that I was explicitly instructed on joining that the team's *modus operandi* when confronted with bugs was to fix them with a date condition so that the original behaviour was retained (in the event of a system re-run for a past date). The end result was a system that was utterly unmaintainable. This is a classic vicious circle.

I think that a paradigm shift in the thinking of development managers, and consequently the developers, is required. There is a learnt mindset to overcome, the mindset that copying is acceptable, that the rush-to-code is the correct approach. I have heard, and find it easy to believe given my personal experience, that significantly more time is spent in supporting the product and fixing bugs (or features, depending on your perspective, and that again is part of the problem) after the release than in development so that in itself should be enough to show the flaws in the mindset. That is, there must be a better way.

Planning to write, and then writing, reusable code is difficult. Maintaining high standards in the face of disinterest is difficult. There is an element of altruism involved, as well as pride in one's work, but that can be eroded quickly when powers-that-be have different perspectives, e.g. development and support budgets being separate and distinct and, possibly, owned by different people or departments. So it's all about education. This is exactly the reason why I have written my articles.

So allow me a moment to spew some key points which I think development managers should be encouraged to repeat.

- Copy-and-paste is evil. This *should* go without saying.
- Engage brain before mouth, or in this case, take the time to think before writing (code or prose, the same applies).
- Constantly re-evaluate your work; re-read what you have written looking for ways by which you can improve it.
- Always look for patterns. These can become shared, generic code.
- Ask whether someone has solved this problem before (and made the solution available).

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

This note came about because of a comment at work. There was a discussion of developers' rewards and how to ensure that the behaviours we wish to encourage are rewarded. The specific point that I identified with is 'Reward your junior developers by lines of code written. Reward your senior developers by lines of code removed.'

Personally, I think that the biggest reward is seeing your code in use. This is even more true if it is being (re-)used in someone else's application. This will never happen while you don't write reusable code and while the developer who might reuse your code doesn't know anything about it or has no incentive to seek it out. So, perhaps the solution lies in teaching the junior developers to seek out reusable code and, in that way, perhaps they will aspire to create their own. Therefore, I finally knuckled down and made my own toolkit available on both Google Code and NuGet. Search for functional-utils-csharp or ask me but don't expect any documentation yet; I'm not perfect (and I already have to juggle my limited spare time between making guitar noise and writing articles for this very magazine).

Richard Polton

Bookcase

The latest roundup of book reviews.

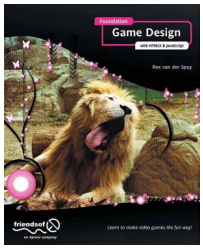
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

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-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

Foundation Game Design with HTML5 and JavaScript



By Rex van der Spuy. ISBN-13: 978-1-4302-4716-6

Reviewed by Stefan Turalski

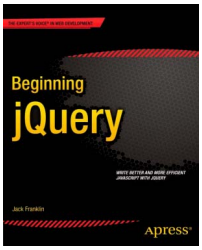
I must admit that whilst choosing the *Foundation Game Design with HTML5 and JavaScript* for a review I was not aware that I was signing up for a 750-pages long paperback brick. In retrospective, I shouldn't have been surprised as the subject domain is vast! It seems that we have not seen those killer HTML5 apps, because since late 2011 most experts (well, mostly experts) were busy churning books on and around the topic. Last time I checked ca.320 titles were published, ranging from beginner's overviews, through definitive guides, various cookbooks and recipes, focusing on performance, mobile development, usage of particular JavaScript frameworks or IDEs, toping the pile with whole book(let)s dedicated to a specific HTML5 tags and APIs.

Within such variety the Rex van der Spuy's work holds, in my opinion, a strong position in the entry level tier. It will not help you to write next single-page, WebGL based 3D game, utilise WebSockets, local storage nor geolocation, but it will definitely assist in establishing solid basic knowledge. The clear narrative, helpful illustrations and thought-through code samples help reader to stay engaged and find himself fully immersed and comfortable in the sea of HTML5 tags, JavaScripts and CSS code. Rex does not set prerequisites, thus even a total beginner, who never closed a HTML tag, will learn enough basics whilst going over first ~300 pages. A reader whishing to jump straight into learning about HTML5's specific techniques, may need to scan through the first half of the book, as each subsequent chapter is based on preceding discussion. However, there is nothing wrong with that and it is well worth to brush up on modern web development tricks.

I must admit that getting the code samples to work triggered this peculiar joy of seeing pixels moving around a green screen (note to editor: green screen should be crossed in final text) browser window, more or less along the intentions expressed in code. For that reason only I would recommend this book to everyone trying to pass her love of coding to a next generation of programmers.

P.S. Please be careful whilst ordering print edition, it seems that some copies are seriously mis-formated.

Beginning jQuery



By Jack Franklin, published by Apress, ISBN: 9781430249320, 181 pages.

Reviewed by Alex Paterson

There can be little doubt that jQuery has made a significant contribution to web development since its first release in 2006. It has provided higher-order constructs for HTML/CSS programming that allow complex dynamic content to be displayed and manipulated in more and more impressive ways.

As a C++ programmer with limited exposure to Javascript, I was hopeful that the book, *Beginning jQuery* by Jack Franklin (Apress, 2013), would give me a head-start on a new web project.

The first two chapters are clearly aimed at those with little programming experience. An overview of Javascript is given and the basic syntax of jQuery is introduced mostly through practical code examples.

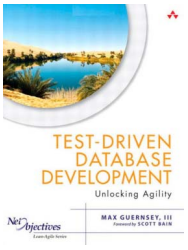
Chapters three and four look at how jQuery can be used to manipulate the DOM and alter the content of a web page. Again, code examples are used to try and explain the concept to the reader.

Chapters five and six give a very brief introduction to the power of functions in Javascript and there are further chapters on animation, ajax, jQuery plug-ins. The book rounds off with a worked example containing code elements seen throughout the chapters.

The author has commendably tried to cover a wide range of topics in a slim volume, however the presentation clearly lacks some polish. The most disappointing aspect is the almost inexcusable lack of diagrams; there are many areas where the author's explanation of a concept should have been accompanied by a diagram to aid the reader's understanding. There are some images in the book, but these are mostly browser output, jQuery API website screenshot and kittens. Yes, kittens. Another area that gave disappointment was the index, which looks incomplete and only provides entries for seven letters of the alphabet.

In summary, despite the interesting subject matter, the readability and presentation issues made the book almost inaccessible for this reader.

Test-Driven Database Development – Unlocking Agility



By Max Guernsey, III - Published by Addison Wesley, 2013 ISBN 13: 987-0-321-78412-4 / ISBN 10: 0-321-78412-X, 315 pages including index (no bibliography)

Review by Bob Corrick – Test-driven to distraction

In a recent Agile Testing Survey [2], 37% say that adopting test-driven development (TDD) is the most difficult challenge. As Martin Fowler explains, there are two main benefits of writing a test (in code) before the functional code: you get the repeatable tests as well as the code, and your functional code is easier to call because you think about how to call it as you write each test. The main challenge is to maintain the discipline of refactoring the code, to improve the design, while continuing to pass the tests [1].

The book starts by walking through an analogy with object oriented applications. Chapter 2 ends with the first specific recommendation: build and modify each database instance (development, test, production) to the required

View from the Chair

Alan Griffiths
chair@accu.org

Summer is vacation season and many things slow down or come to a stop – and the ACCU committee has been participating in this behaviour. There have been no committee meetings since my last report and very little that the committee has needed to deal with.

Next year is going to see some changes in the ACCU committee. Both Giovanni Asproni (as Secretary) and I (as Chair) intend to step down. As yet no-one on the current committee has shown an interest in taking either post.

In addition, as my last report explained we already need to find a new Membership Secretary. While the committee has co-opted last year's Membership Secretary (Mick Brooks) until a more permanent solution is found he, and we, would appreciate it if a volunteer stepped forward with a view to standing for election next year.

This means that it is likely that three key posts will not have current incumbents standing at the AGM next year. I personally think that it would be a good idea that any candidates have experience of the way that the committee works beforehand – and if anyone is interested in standing for election to these post next year I would urge them to make themselves known and to seek co-option to the current committee.

There is no need for committee members to reside in the UK as committee business is largely conducted over the Internet.

Bookcase (continued)

state, by applying the same version-numbered scripts in strict sequence. So far, not so bad.

The test-first examples are all for SQL Server, and the very first one tests whether the database exists. I prefer not to get in at the deep end, and don't have SQL Server, so I copied the scripts (SQL) out of the NUnit and C# code, and ran them directly in database sessions (Oracle, MySQL). One of the early examples is an elaborate test, using a C# exception, for adding a unique constraint. Here a simpler script (inserting two identical emails, and then failing on adding the database constraint) could be used to the same end, and could also show how the test-first technique can be adapted for databases.

This tendency to complexity is distracting, if not infuriating. In a worked example for a cross-reference of street intersections, a simple problem description is followed by two full pages of 'big design up front' as a deliberate demonstration of how not to do it. When you do get to the test scripts, they insert some data and test for a connection depth of 3, whatever that means. A simple query on a view of street addresses, listing where other streets intersect, would be more effective.

You can find an example of refactoring in the chapter 'Safely Changing Design', which is about checking that data is preserved during a change. That's good advice, but the discipline of refactoring is not treated explicitly until chapter 11. That chapter begins 'This book isn't directly about refactoring', revealing an outlook that reflects the challenge identified by Martin Fowler.

You won't find any advice on developing business logic in the database ("you should never do this", page 97), or on developing for 'NoSQL' databases (despite chapters 14 'Variations' and 15 'Other Applications'). This book is not for me, and I can't recommend it.

Downloadable code is available, all for Microsoft products as far as I could see: similar open source tools are described at datacentricity.net/tsqlt/

References

[1] Martin Fowler online martinfowler.com/bliki/TestDrivenDevelopment.html

[2] Agile Testing Survey November 2012, Ambysoft www.ambysoft.com/surveys/agileTesting201211.html