overload 134

AUGUST 2016 £3

Kill the Clones
Problem code can hide in surprising places. We see how to detect software clones and uncover hidden dependencies.

Testing Propositions
Is testing propositions more important than having examples as exemplars?

C++ Antipatterns
"Pro-tips" to avoid the mistakes that crop up frequently in C++ code

Some Big-Os are Bigger Than Others
Exploring the limits of Big-O notation

Implementing Snaaake
Tales of implementing the classic game of Snake!

Afterwood
Removing the barriers that cause bottlenecks

A magazine of ACCU

ISSN: 1354-3172

JET
BRAINS

A Power Language Needs Power Tools

We at JetBrains have spent the last decade and a half helping developers code better faster, with intelligent products like IntelliJ IDEA, ReSharper and YouTrack. Finally, you too have a C++ development tool that you deserve:

- Rely on safe C++ code refactorings to have all usages updated throughout the whole code base
- Generate functions and constructors instantly
- Improve code quality with on-the-fly code analysis and quick-fixes

ReSharper C++

Visual Studio Extension for C++ developers

CLion

Cross-platform IDE for C and C++ developers

AppCode

IDE for iOS and OS X development

Find a C++ tool for you
jb.gg/cpp-accu

OVERLOAD 134

August 2016

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 135 should be submitted by 1st September 2016 and those for Overload 136 by 1st November 2016.

Overload is a publication of the ACCU

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

4 Some Big-Os are Bigger Than Others
Sergey Ignatchenko reminds us that Big-O applies to very large data sets.

8 Kill the Clones
Adam Tornhill demonstrates how to detect software clones and uncover hidden dependencies using Empear.

11 Implementing SNAAAKE
Thaddaeus Frogley shares a diary of how his implementation of Snake! developed.

14 C++ Antipatterns
Jonathan Wakely offers some pro-tips to help you avoid common errors in C++.

17 Testing Propositions
Russel Winder considers how to test propositions rather than specific examples.

28 Afterwood
Chris Oldwood considers varying approaches to gatekeeping.

The ACCU

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

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

Just a minute

Constraints can seem like an imposition. Frances Buontempo wonders if banning hesitation, deviation or repetition is a good thing.

The BBC Radio 4 comedy ‘Just a minute’ has been running since 1967. For those unfamiliar with it, the contestants are tasked to speak for one minute on a subject without hesitation, deviation or repetition. Points are awarded for speaking when the minute whistle blows, or for correctly spotting a failure to comply with the rules. “What does this have to do with an editorial for *Overload*?” I hear you ask. A good question. Having recently stopped commuting to work I am now at home in time to catch some of the 18:30 broadcasts of Radio 4 comedy programmes. Previously I could find them on the BBC iPlayer, but I prefer the set time and structure of certain little rituals. Having an editorial at the front of *Overload* is presumably a ritual some readers pine for, though as ever I hope I may be forgiven for deviating somewhat, sometimes repeating myself and definitely hesitating as I try to gather my thoughts every two months and yet again fail to write an editorial. Perhaps you will award me the occasional point for making you chuckle from time to time though.

Constraints, such as banning hesitation, repetition, or deviation might seem to make things harder; however, this can actually aid creativity. For example, writing a short story for a competition with a strict limit on the number of words, or filling in a personal statement for a role with a strict limit on the number of characters (spaces did count apparently) gives a structure and something to aim for. Adding a deadline may also provide some impetus. Constraints and structure can be a good thing. Writing a short story is a vague requirement, whereas writing one in under 500 words, using ‘elephant’, ‘Plank constant’ and ‘denial of service attack’ might focus the mind enough to get the imagination firing.

If you are self-employed or conducting long term research, perhaps a PhD, then a change from set working hours and a list of tasks to complete, or from timetabled lectures and weekly homework can be a shock to the system. On paper, it might sound liberating. Just imagine being free to do whatever you want, when you feel like it. In practice, a high level of discipline is required to ensure you do not get distracted, say by listening to comedy programs or doing a random walk through the internet, and instead keep focussed on the task in hand. Finding some structure, or setting a timetable for yourself, can be useful in such circumstances. Now working with a (very) small start-up, I have found it useful to start the day with a coffee, of course, and to use the agile style daily stand-up format, just briefly. What were we doing? What do we plan to do today? What’s getting in the way? It is far too easy to start by looking at emails and so on, only to find two hours have vanished once some emails have been read, others deleted, links to articles have been followed, proclaiming exciting new programming languages and tools, surveys have sucked you in, and other distractions, or deviations, if you will, have stolen your time. What was I saying? Oh yes. Try to stay on target in the face of distractions and lack of formal structure. It is all too easy to be tempted to stay up far too late if you are in the middle of something interesting, without the need to be in the office by 9am sharp the next day. As a student, I frequently pulled all-nighters. I love having an interesting problem to get my teeth into, and there is usually no one much around to distract you in the middle of the night. However, irregular sleep patterns are not a good idea. I have personally found it really important to try to keep regular bed times, avoiding reading maths books before putting the light out. If I don’t, then I find it harder to get a proper night’s sleep. The interesting problem will still be there in the morning. If you are pulling an all-nighter for work, the bug will still be there in the morning. In both cases, you might be more effective after a good night’s sleep. It seems sleep/wake homeostasis and the circadian biological clock are important [Body Clock]. There have been suggestions that irregular sleeping patterns, including shift work, can lead to health problems. J Harrington provides an overview in the BMJ’s *Occupational and Environmental Medicine* magazine [Harrington].

As programmers, hopefully avoiding burning the midnight oil, we schedule many things, often automatically. Aside from keeping meetings in regular timeslots, to avoid the disruption of flash-mob style meetings intruding on moments when you were trying to concentrate on a complex problem, we may schedule server reboots, or scripts to run when code is committed, from checking it compiles to checking tests pass or perhaps to deployment if everything is ok. We may automatically, or manually, check that coding standards, another form of constraint, have been adhered to. Hopefully the occasional deviation will be allowed, otherwise developers have a tendency to find back doors and workarounds where officious gatekeepers are perceived to stand in the way of productivity. A known timetable, such as a linux startup sequence; bios, mbr, grub, kernel, init then runlevel, means we know what will happen in which order. This constraint makes it easy to set things up and troubleshoot if required. Every linux installation doing things in a unique and unpredictable order would cause chaos.

A programming language could be seen as a sequence of constraints. Either a compiler or interpreter will enforce the syntax of the language and refuse to do any more if you try the coding equivalent of free-form jazz where a type-name was expected. Some languages are stricter than others. The Haskell type system is strongly revered, purporting to be able to give you code that is correct, rather than defensive. Originally introduced to solve problems concerning to equality and other numeric operations, its powerful type classes now allow reasoning about correctness [Hudak *et al*]. At the extreme, some languages, such as Prolog, are purely based on constraints. Where an imperative approach

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

would give step by step instructions and a declarative approach would declare what must be done, logic or constraint programming states constraints, or facts or rules, a solution must satisfy. This immediately lends itself to word puzzles like:

```
    SEND
 +  MORE
 -------
 = MONEY
```

where each unique letter corresponds to a unique digit, and the given sum is correct. Prolog and variants have successfully been used for expert systems and natural language programming (NLP). GOLEM is an expert system built on Prolog [Muggleton] which combines examples with rules supplied by experts that are combined into hypotheses. In turn these can then be transformed into rules, when validated against further data. NLP uses include representing a semantic web [Wielemaker *et al*], to combine disparate data from various sources allowing a 'semantic search' to find potentially related entries. Constraint programming can also be applied to planning problems readily. The constraints in these examples all form the basis of a problem that the language will attempt to solve. I suspect logic programming is rather more niche than functional programming. More generally, this little diversion has shown constraints can solve problems.

Forays into niche languages aside, unfamiliarity can be exciting and uncomfortable in varying measures. A holiday or a new job will offer novelty but the potential culture shock as you are re-orientated in unusual places is a reminder that the same-old run-of-the-mill routine had its benefits. Many C++ programmers, or users of any compiled language, find dynamic languages unnatural initially. Similarly going from the constraint of a strongly typed language to duck typing can be confusing. It is interesting to note that more freedom often tends to coincide with fear, to begin with. Eventually people tend to embrace these brave new worlds. As a kitten constantly looks out of the window, seemingly longing to explore, and then recoils in horror when the back door is finally opened after a few weeks, the constraints that we are used to provide a familiarity and sense of safety. This disruption, whether it be changing programming languages or paradigms, operating systems or going outside for the first time ever is a thing of note. Constraints help us to be creative, by knowing what to expect, catching mistakes early, communicating clearly, even if hesitating when we forget the exact keyword or syntax, but also can inspire deviation or disruption. What if C++ (03) didn't require so much boilerplate just to output the contents of a vector? What if you introduced structure, via functions or submodules, to a coding language rather than having to trace `goto`s around to see what happened when? What if everything was an object? Maybe that's a step too far. What if you didn't use braces to delineate blocks, but used spacing instead, perhaps inventing a new language in the process? Constraints can help creativity, but so can questioning them.

The start-up culture has readily adopted the buzz-word 'disruptive'. Clayton Christensen [Christensen] coined the phrase 'disruptive innovation' in 1995 to describe small new companies being able to release new services and products far more quickly than established companies thereby being able to take hold of the so-called 'bottom of the market' and eventually stir things up for the 'big boys'. We see this happening with apps on mobile phones to book taxis, rather than hailing traditional Black Cabs, with noise about crypto-currencies stealing a market share off big banks, Amazon starting to threaten bookshops, and so on. Things do change and often small companies do disrupt the normal course of things with innovative ideas. I notice people are tending to just say 'disruptive' rather than using the whole phrase 'disruptive innovation'. Something got lost in the meme sharing. Something like the punk anthem 'Smash it up' [The Damned] conjures up the sense of pure disruption for disruption's sake. Disruptive innovation, on the other hand, may create new markets, or sustain existing products and services, by making them more efficient.

Sometimes rebellion for rebellion's sake does hail a new beginning. Punk was a kick against the hippie movement or prog rock, giving short simple songs anyone could yell along to, or write themselves. It's tempting to draw analogies between the punk rock movement and disruptive start-ups, though like most ideas it's been done before. The Think Jar Collective offers such a parallel and reminds us that for all its disruption:

> The punk movement and the music ate itself within a few years of its adrenaline and narcotic inspired inception. But punk morphed itself into new wave and gradually added itself into the mainstream of music.[TJC]

Uber and Amazon and the like are now becoming mainstream. New things do sometimes become old-hat. The abnormal might morph the world around itself until it becomes the new normal. This usually happens because people like the new ideas, rather than through clever marketing schemes.

References

[Body Clock] https://sleepfoundation.org/sleep-topics/sleep-drive-and-your-body-clock

[Christensen] http://www.claytonchristensen.com/key-concepts/

[The Damned] https://www.youtube.com/watch?v=YZ76LC_l8yk

[Harrington] http://oem.bmj.com/content/58/1/68.full

[Hudak *et al*] 'A History of Haskell: Being Lazy With Class' Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, Pages 12-1–12-55, 2007 http://dl.acm.org/citation.cfm?id=1238856

[Muggleton] S. Muggleton and C. Feng, 'Efficient induction of logic programs', *Inductive Logic Programming*, Academic Press, pages 281–297, 1992.

[TJC] http://thinkjarcollective.com/articles/punk-rock-innovation/

[Wielemaker *et al*] 'Using Prolog as the fundament for applications on the semantic web' Jan Wielemaker , Michiel Hildebrand2 , and Jacco van Ossenbruggen in *Proceedings of the 2nd Workshop on Applicatiions of Logic Programming and to the web, Semantic Web and Semantic Web Services*, p84–98, 2007 http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-287/paper_1.pdf

Some Big-Os are Bigger Than Others

Big-O notation is often used to compare algorithms. Sergey Ignatchenko reminds us that asymptotic limits might not be generally applicable.

Often, when speaking about algorithms, we say things such as, "Hey, this algorithm is O(n^2); it is not as good as that one, which is O(n)". However, as with any bold and simple statement, there are certain very practical applicability limits, which define when the statement can be taken for granted, and where it cannot. Let's take a closer look at Big-O notation and at the real meaning behind it.

Definition

Mathematically, Big-O can be defined as follows:

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n > 0, \forall n \geq n0 : 0 \leq f(n) \leq cg(n)\}$$

In other words, $f(n) \in O(g(n))$ if and only if there exist positive constants c and $n0$ such that for all $n \geq n0$, the inequality $0 \leq g(n) \leq cg(n)$ is satisfied. We say that $f(n)$ is Big O of $g(n)$, or that $g(n)$ is an *asymptotic upper bound* for $f(n)$ [CMPS102].

In addition to O(n), there is a similar notation of Θ(n); technically, the difference between the two is that O(n) is an upper bound, and Θ(n) is a 'tight upper bound'. Actually, whenever we're speaking about Big-Os, we usually actually mean Big-Thetas.

For the purposes of this article, I prefer the following plain-English sorta-definition [Wiki.Big-O]:

1. '$T(n)$ is O(n^{100})' means $T(n)$ grows asymptotically no faster than n^{100}
2. '$T(n)$ is O(n^2)' means $T(n)$ grows asymptotically no faster than n^2
3. '$T(n)$ is Θ(n^2)' means $T(n)$ grows asymptotically as fast as n^2

Note that (as O(n) is just an upper bound), all three statements above can stand for the very same function $T(n)$; moreover, whenever statement #3 stands, both #1 and #2 also stand.

In our day-to-day use of O(n), we tend to use the tightest bound, i.e. for the function which satisfies all three statements above, usually we'd say "it is O(n^2)", while actually meaning Θ(n^2).

Admittedly, the preliminaries above are short and probably insufficient if you haven't dealt with Big-Os before; if you need an introduction into the world of complexities and Big-Os, you may want to refer, for example, to [Orr14].

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

Everything is O(1)

One interesting observation about O(n) notations is that:

> *Strictly speaking, for real-world computers, every algorithm which completes in a finite time can be said to be O(1)*

This observation follows from the very definition above. As all real-world computers have finite addressable data (2^{64}, 2^{256}, and the number of atoms in the observable universe are finite), then for any finite-time algorithm there is a finite upper bound of time MAXT (as there is only a finite number of states it can possibly go through without looping). As soon as we know this MAXT, we can say that for the purposes of our definition above, c is MAXT, and then the inequality in the definition stands, technically making ANY real-world finite-time algorithm an O(1).

It should be mentioned that this observation is more than one singular peculiarity. Instead, it demonstrates one very important property of the Big-O notation: *strictly speaking, all Big-Os apply ONLY to infinite-size input sets*. In practice, this can be relaxed, but we still need to note that

> *Big-O asymptotic makes sense ONLY for 'large enough' sets.*

Arguing about O(n) vs O(n^2) complexity for a set of size 1 is pretty pointless. But what about size 2? Size 64? Size 1048576? Or more generally:

> *What is the typical size when the difference between different Big-Os starts to dominate performance in the real world?*

Big-O notation itself is of no help in this regard (neither is it intended to be). To answer this question, we'll need to get our heads out of the mathematical sand, and deal with the much more complicated real world.

History: early CPUs

Ancient times

Historically, Big-O notation, as applied to algorithm complexity, goes back at least 50 years. At that time, most CPUs were very simple, and more importantly,

> *Memory access times were considered comparable to CPU operation times*

If we take the MIX machine [Knuth] (which assigns execution times 'typical of vintage-1970 computers'), we'll see that ADD, SUB, etc. – as well as all LOAD/STORES – are counted as 2 CPU clocks, MUL is counted as 10 CPU clocks, and the longest (besides I/O and floating-point) operation DIV is counted as 12 CPU clocks. At the same time, memory-copying MIX operation MOVE goes at the rate of two CPU clocks per word being moved. In short:

> *In 1970's computers – and in the literature of the time too – memory accesses were considered to have roughly the same CPU cost as calculations.*

As this was the time that most O(n) notation (as applied to computer science) was developed, it led to quite a number of assumptions and common practices. One of the most important assumptions (which has since become a 'silent assumption') is that we can estimate complexity by

This discrepancy in speed growth between CPU core and memory latencies has caused Very Significant changes to the CPU architecture

simply counting the number of operations. Back in ancient times, this worked pretty well; it follows both from the MIX timings above, and from data on emerging micro-CPUs such as the i8080 (which had R-R ops at 4 clocks, and R-M ops at 6 clocks). However, as CPUs were developed, the cost difference of different ops has increased A LOT, while the assumption (being a 'silent' assumption) has never really been revised.

Modern CPUs: from 1 to 100 clocks and beyond

From about the 80s till now, the situation has changed in a very significant way. Since about the early 80s, micro-CPU clocks speeds grew from ~4MHz to ~4GHz (i.e. 1000-fold), while memory latencies only improved from about 250ns to about 10ns, i.e. only 25-fold[1] ☹. This discrepancy in speed growth between CPU core and memory latencies has caused Very Significant changes to CPU architecture; in particular, LOTS of caches were deployed between the core and main RAM, to make things work more smoothly.

Currently, a typical x64 CPU has three levels of cache, with typical access times being 4, 12 and 40 clocks for L1, L2 and L3 caches respectively. Main RAM access costs 100–150 clocks (and up to 300 clocks if we're speaking about multi-socket configurations with memory on a different NUMA node). At the same time, the costs of simple ALU operations (such as ADD etc.) have dropped below 1 CPU clock.[2]

This is a Big Fat Contrast with the assumptions used back in the 1970s; now the difference because of unfortunate 'jumps' over (uncached at the time) memory can lead to a 100×+ (!) performance difference. However, it is still O(1) and is rarely taken into account during performance analysis ☹.

Real world experiment

Theory is all very well, but what about a little practical experiment to support it? Let's take a look at a short and simple program (Listing 1).

Trivial, right? Now let's see how this program performs in two cases: first, when run 'as is', and second, when we replace **list<int>** in line **(*)** with **vector<int>**.

As we can see from Listing 1, between moments t0 and t1 we're only traversing our lists or vectors, and each inner iteration is O(1). Therefore, the whole thing between t1 and t0 is clearly O(N*M*P) = O(n) – both for list and vector. Now, let's see what is the *real-world* difference between these two algorithms with exactly the same Big-O asymptotic.

WARNING: VIEWER DISCRETION ADVISED. The results below can be unsuitable for some of the readers, and are intended for mature developer audiences only. Author, translator, and Overload *disclaim all the responsibility for all the effects resulting from reading further, including, but not limited to: broken jaws due to jaw dropping, popped eyes, and being bored to death because it is well-known to the reader.* (See Table 1.)

```
constexpr int N=100;
constexpr int M=5000;
constexpr int P=5000;

using Container = list<int>;//(*)

int main() {
  //creating containers
  Container c[M];
  srand(time(0));
  for(int i=0;i<M;++i) {
    for(int j=0;j<P;++j) {
      Container& cc = c[rand()%M];
      cc.push_back(rand());
    }
  }

  clock_t t0 = clock();
  //running over them N times
  int sum = 0;
  for(int i=0;i<N;++i) {
    for(int j=0;j<M;++j) {
      for(int it: c[j]) {
        sum += it;
      }
    }
  }

  clock_t t1 = clock();
  cout << sum << '\n';
  cout << "t=" << (t1-t0) << '\n';
  return 0;
}
```

Listing 1

As we can see, Big-O(n) for **list<>** *has been observed to be up to 780× bigger than intuitively the same Big-O(n) for* **vector<>**.

And that was NOT a specially constructed case of swapping or something – it was just an in-memory performance difference, pretty much without context switching or any other special scenarios; in short – this can easily happen in a pretty much any computing environment. *BTW, you should be able to reproduce similar results yourself on your own box, please just don't forget to use at least some optimization, as debug overhead tends to mask the performance difference; also note that drastically reducing M and/or P will lead to different cache patterns, with results being very different.*

In trying to explain this difference, we'll need to get into educated-guesswork area. For MSVC and gcc, the performance difference between **vector<>** and **list<>** is pretty much in line with the difference between typical cached access times (single-digit clocks) and typical uncached access times (100–150 clocks). As access patterns for

1. I don't want to get into discussion whether it's really 10× or 100× – in any case, it is MUCH less than 1000×.
2. Statistically, of course.

any evidence we've got represents only a small subset of all the possible experiments

Box 1	clang -O2 N=100,M=5000,P=5000 RESULT: **vector<> is 434x faster**	clang -O3 -march=native N=100,M=5000,P=5000 RESULT: **vector<> is 498x faster**	clang -O3 -march=native N=100,M=1000,P=1000 RESULT: **vector<> is 780x faster**[a]
Box 2	MSVC Release N=100,M=1000,P=1000 RESULT: **vector<> is 116x faster**	MSVC Release N=100,M=5000,P=5000 RESULT: **vector<> is 147x faster**	gcc -O2 N=100,M=1000,P=1000 RESULT: **vector<> is 120x faster**

a. This result is probably attributed to `vector<>` with M=1000 and P=1000 fitting into L3 cache on this box.

Table 1

`vector<>` are expected to use CPU prefetch fully and `list<>` under the patterns in Listing 1 is pretty much about random access to memory, which cannot be cached due to the size, this 100–150× difference in access times can be expected to translate into 100–150× difference in performance.

For clang, however, the extra gain observed is not that obvious. My somewhat-educated guess here is that clang manages to get more from parallelization over linear-accessible `vector<>`, while this optimization is inapplicable to `list<>`. In short – when going along the `vector<>`, the compiler and/or CPU 'know' where exactly the next `int` resides, so they can fetch it in advance, and can process it in parallel too. When going along the `list<>`, it is NOT possible to fetch the next item until the pointer is dereferenced (and under the circumstances, it takes a looooong while to dereference this pointer).

On the other hand, it should be noted that an exact explanation of the performance difference is not that important for the purposes of this article. The only thing which matters is that we've taken two 'reasonably good' (i.e. NOT deliberately poorly implemented) algorithms, both having exactly the same Big-O asymptotic, and easily got 100×-to-780× performance difference between the two.

Potential difference: 1000x as a starting point

As we can see, depending on the compiler, results above vary greatly; however, what is clear, is that

> *These days, real-world difference between two algorithms with exactly the same Big-O asymptotic behaviour, can easily exceed 500×*

In other words: we've got very practical evidence that the difference can be over 500×. On the other hand, any evidence we've got represents only a small subset of all the possible experiments. Still, lacking any further evidence at the moment, the following assumption looks fairly reasonable.

> *With modern CPUs, if we have two algorithms (neither of which being specially anti-optimized, and both intuitively perceived to have the same performance at least by some developers), the performance difference between them can be anywhere from 0.001× to 1000×.*

Consequences

Now let's take a looking at the same thing from a bit different perspective. The statement above can be rephrased into the following:

> *with modern CPUs, unknown constants (those traditionally ignored in Big-O notation) may easily reach 1000.*

In turn, this means that while O(n2) is still worse than O(n) *for large values of n*, for n's around 1000 or so we can easily end up in a situation when:

- algo1() is $O(n^2)$, but T(algo1(n)) is actually 1* n^2
- algo2() is O(n), but T(algo2(n)) is actually 1000*n
- then $\forall$ n < 1000, T(algo1(n)) = n^2 < 1000*n = T(algo2(n))

This observation is nothing new, and it was obviously obvious to the Founding Fathers back in the 1970s; what's new is the *real-world difference* in those constants, which has grown Very Significantly since that time, and can now reach 1000×. In other words,

> *the maximum size when $O(n^2)$ can be potentially faster than O(n), has grown to a very significant n~=1000*

If we're comparing two algos, one with complexity of O(n) and another with complexity of O(log n), then similar analysis will look as follows:

- algo1() is O(n), but T(algo1(n)) is actually 1* *n*
- algo2() is O(log n), but T(algo2(n)) is actually 1000*$\log_2$(n)
- then $\forall$ n < 13746, T(algo1(n)) = n < 1000*$\log_2$(n) = T(algo2(n))

In other words,

> *the maximum size when O(n) can be potentially faster than O(log n), is currently even larger, in the over-10K range*

Summary

All animals are equal, but some animals are more equal than others.
~ George Orwell, Animal Farm

To summarize the findings above:

- Big-O notation still stands ;-)
  - All Big-Os are Big, but some Big-Os are bigger than others

If in doubt – test it! Real-world results can be very different from intuitive expectations

  - On modern CPUs, the performance difference between two reasonably good algos with the same Big-O can easily reach over 500× (that's for usual scenarios, without swapping or context switch thrashing)
- Big-O asymptotic can still be used for *large enough* values of *n*. However, what qualifies as *large enough* for this purpose, has changed over the years
  - In particular, for modern CPUs, even if the asymptotic difference is 'large' (such as comparing O(n^2) to O(n)), then the advantage of O(n) SHOULD NOT be taken as granted for sizes < 1000 or so
  - If the asymptotic difference is 'smaller' (such as comparing O(n) to O(log n)), then the advantage of O(log n) SHOULD NOT be taken as granted for sizes < 10000 or so
  - Starting from n=10000, we can usually still expect that the better Big-O asymptotic will dominate performance in practice
  - If in doubt – test it! Real-world results can be Very Different from intuitive expectations (and still Very Different from estimates based on pure Big-O asymptotic).

References

[CMPS102] CMPS 102, 'Introduction to Analysis of Algorithms', University of California, https://classes.soe.ucsc.edu/cmps102/Spring04/TantaloAsymp.pdf

[Knuth] Donald E. Knuth, *The Art of Computer Programming*, Vol.1, section 1.3.1

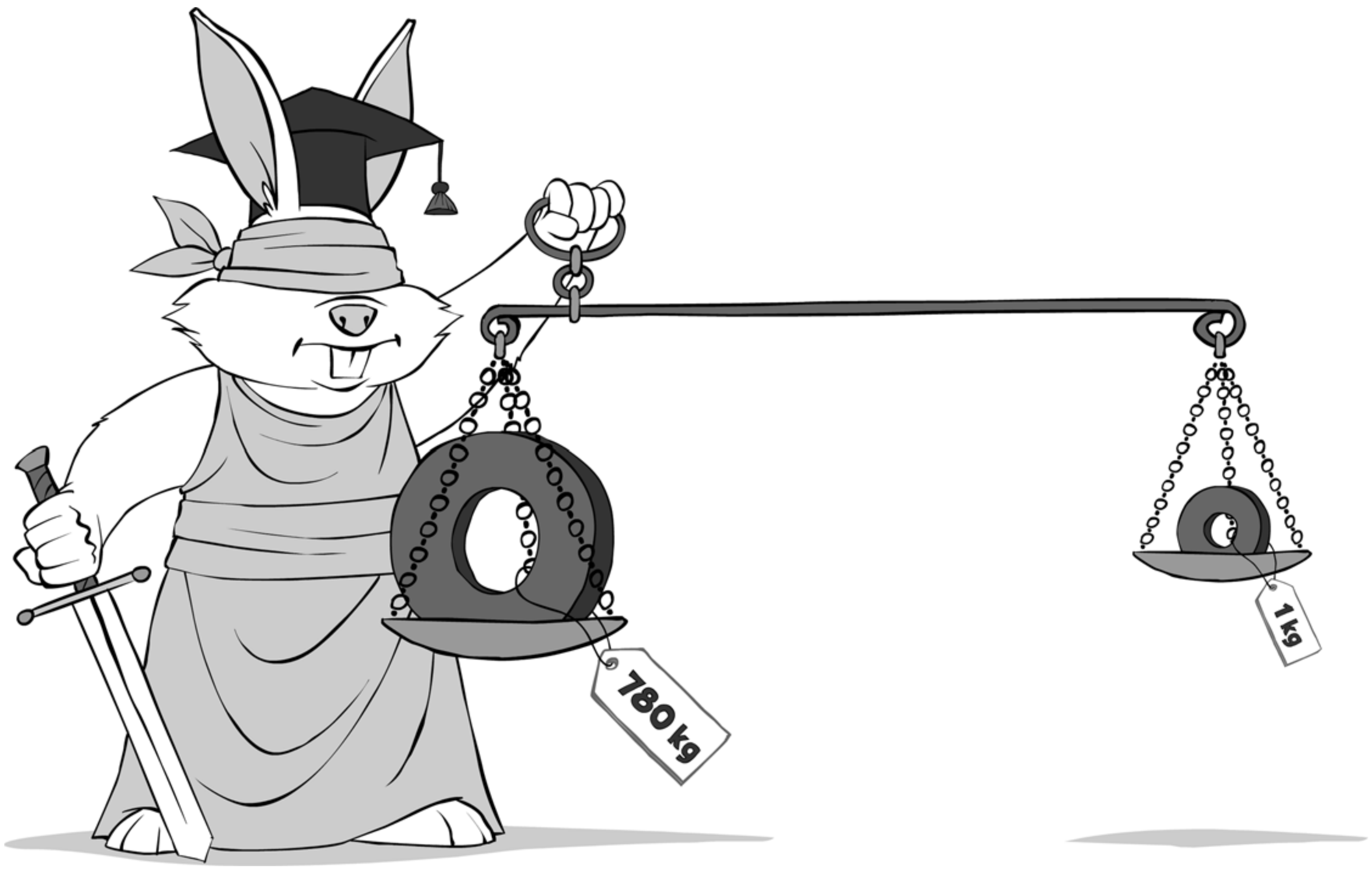
[Loganberry04] David 'Loganberry' Buttery, 'Frithaes! – an Introduction to Colloquial Lapine', http://bitsnbobstones.watershipdown.org/lapine/overview.html

[Orr14] Roger Orr, 'Order Notation in Practice', *Overload* #124

[Wiki.Big-O] https://en.wikipedia.org/wiki/Big_O_notation

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.



Kill the Clones

Problems in code can hide in surprising places. Adam Tornhill demonstrates how to detect software clones and uncover hidden dependencies.

Your Code as a Crime Scene [Tornhill15] presents around 15 different software analyses. Of all those analyses, *Temporal Coupling* is the most exciting one. In this article, we'll put the analysis to work on a real-world system under heavy development: Microsoft's ASP.NET MVC framework [ASP]. You'll see why and how Temporal Coupling helps us design better software as we detect a number of possible quality issues in ASP.NET MVC. In addition, you get a preview of Empear Developer [Empear], a new tool to automate software analyses.

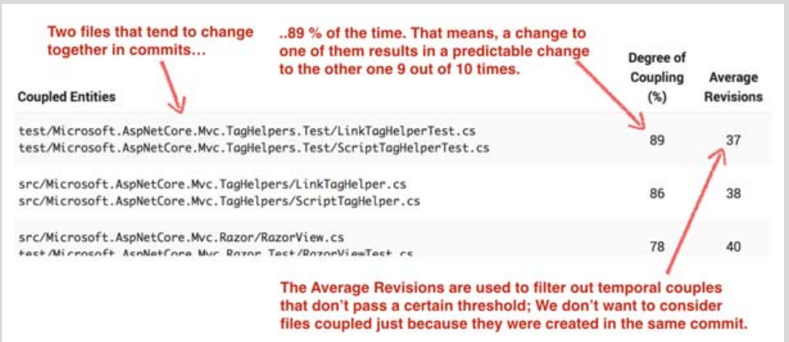
What's temporal coupling?

Temporal Coupling means that two (or more) modules change together over time. In this version, Empear Developer (see Figure 1) considers two modules coupled in time if they are modified in the same commit (in future versions you'll find other options too).

The fascinating thing with Temporal Coupling is that the more experience we get with the analysis, the more use cases there seem to be. For example, you can use Temporal Coupling results to:

1. Detect software clones (aka copy-paste code).
2. Evaluate the relevance of your unit tests.
3. Detect architectural decay.
4. Find hidden dependencies in your codebase.

In this article we focus on detecting software clones and uncovering hidden dependencies.



Two files that tend to change together in commits...

..89 % of the time. That means, a change to one of them results in a predictable change to the other one 9 out of 10 times.

Coupled Entities	Degree of Coupling (%)	Average Revisions
test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/LinkTagHelperTest.cs test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs	89	37
src/Microsoft.AspNetCore.Mvc.TagHelpers/LinkTagHelper.cs src/Microsoft.AspNetCore.Mvc.TagHelpers/ScriptTagHelper.cs	86	38
src/Microsoft.AspNetCore.Mvc.Razor/RazorView.cs test/Microsoft.AspNetCore.Mvc.Razor.Test/RazorViewTest.cs	78	40

The Average Revisions are used to filter out temporal couples that don't pass a certain threshold; We don't want to consider files coupled just because they were created in the same commit.

Figure 1

Explore your physical couples

Let's fire-up Empear Developer and run an analysis of ASP.NET MVC. Since we're interested in exploring Temporal Coupling, we click on the corresponding button. Figure 2 shows what the result looks like.

Empear Developer presents us with multiple views so that we can investigate different aspects of the results. The default view above is based on a visualization technique called Hierarchical Edge Bundling. You see the names of all involved files as nodes with links between the ones that are coupled. If we hover over a node, its temporal couples will light-up in red.

So, why do two source code files change together over time? Well, the most common reason is that they have a dependency between them; One module is the client of the other.

Empear Developer

Empear Developer is a desktop application for programmers and software architects that gives you unique insights into your codebase:

- Use a Hotspot analysis to identify complicated code that you have to work with often.
- Discover Temporal Coupling between modules in your code and get deep design insights.
- Use Complexity Trends to supervise how your code evolves over time and react early to potential quality issues.

Empear Developer is a tool built on the ideas in the popular book *Your Code as a Crime Scene*. Empear Developer analyses data from your Git source code repository. This gives you a new perspective on your codebase and unique insights that you cannot get from the code alone. And it's information that's based on actual data from how you've worked with the code so far. You use that information to improve your software and make it easier to maintain.

Adam Tornhill Adam is a programmer who combines degrees in engineering and psychology. He's the author of Your Code as a Crime Scene, has written the popular 'Lisp for the Web' tutorial and self-published a book on Patterns in C. Adam also writes open-source software in a variety of programming languages. His other interests include modern history, music and martial arts.

What are the criteria for Temporal Coupling?

Empear Developer only looks for changes in the same commit. However, Empear's Enterprise version [Enterprise] presents alternative metrics where two, or more, files are coupled if:

- they are changed by the same programmer within a specific time frame, or
- the commits refer to the same Ticket ID (Jira, GitHub, etc).

The first metric is a heuristic while the second one delivers more accurate results based on the ticket system on top. That means you're able to find temporal coupling even if the files are changed by different programmers (e.g. front-end versus back-end teams) or when you have your code in multiple repositories.

You use this variant of temporal coupling to identify change patterns that ripple over repository boundaries. For example, I did a software analysis for one of our customers a while back. They used a microservices architecture and we identified some heavy temporal coupling between some of their services. That was a clear sign that their services weren't as autonomous as they should be. Not only is it hard to reason about system behaviour in that case; It's also expensive to maintain since you now need to deploy whole clusters of services at once.

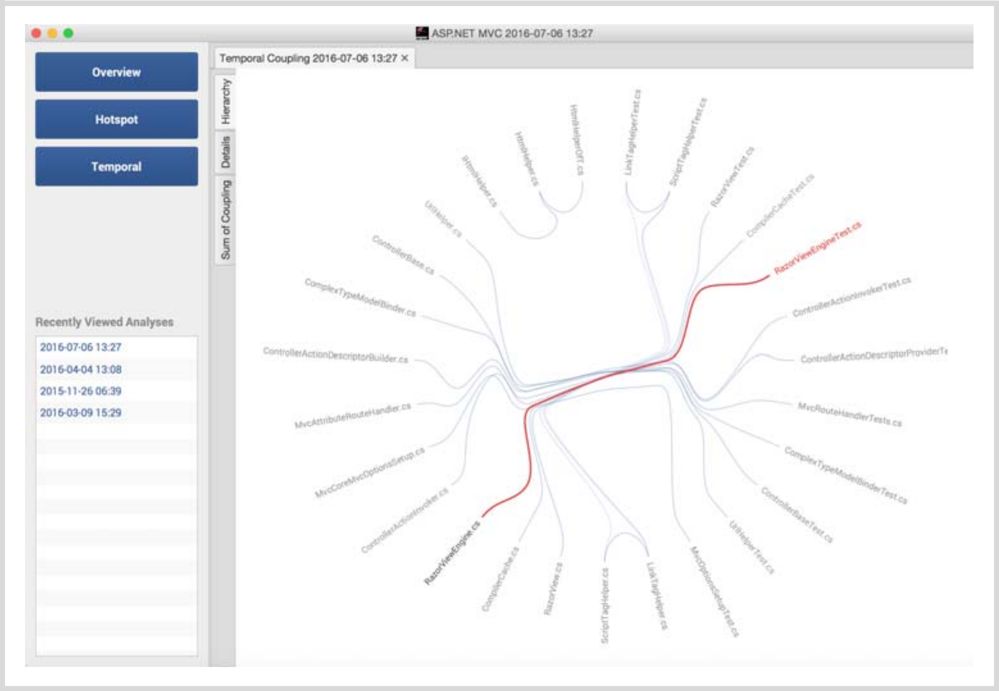


Figure 2

You see a few examples on physical coupling in the picture above, a unit test tends to change together with the code under test. This is expected. In fact, we'd be surprised if the temporal coupling was absent – that would be a warning sign since it indicates that your tests aren't being kept up to date or aren't relevant.

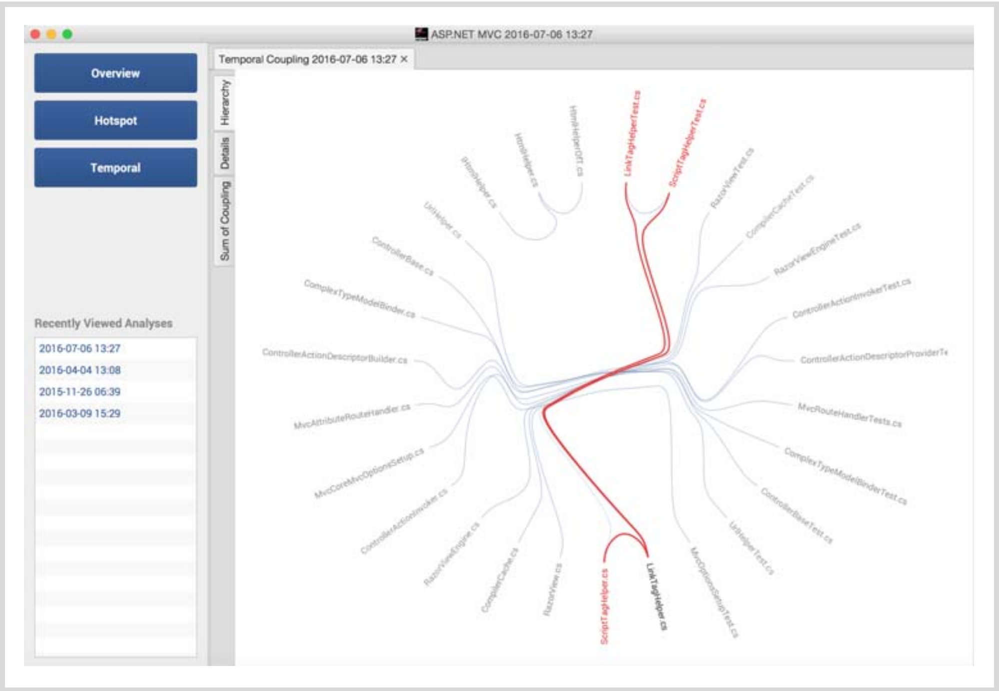


Figure 3

A physical dependency like this is something you can detect from the code alone. But remember that Temporal Coupling isn't measured from code; Temporal Coupling is measured from the evolution of the code. That means you'll sometimes make unexpected findings.

Look for the unexpected

There's one main heuristic to keep in mind as you analyze a software system: always look for the unexpected. Look for surprises, since a surprise in a software design is bound to be expensive from a cognitive perspective and therefore expensive to maintain.

As soon as you find a logical dependency that you cannot explain, make sure to investigate it. Let me clarify with an example from ASP.NET MVC (see Figure 3).

The visualization above shows temporal coupling between a `LinkTagHelper.cs` and a `ScriptTagHelper.cs`. You also see that their unit tests tend to be changed together.

While those two classes seem to solve related aspects of the same problem, there's no good reason why a change to one of them should imply that the other one has to be changed as well. Let's get more information on this potential problem by switching to the detailed view in Figure 4.

The data above confirms that there's a strong degree of coupling between the `LinkTagHelper.cs` and `ScriptTagHelper.cs`, but also between their unit tests. 9 out of 10 changes we do to one of the classes will result in a predictable change to its coupled peer.

When you find an unexpected change pattern like this you need to dig into the code and understand why. It's particularly interesting in this case since there is not any direct physical dependency between the logically coupled classes! Let's have a look at the `LinkTagHelper.cs` and the `ScriptTagHelper.cs` to see if we can spot any patterns (see Figure 5).

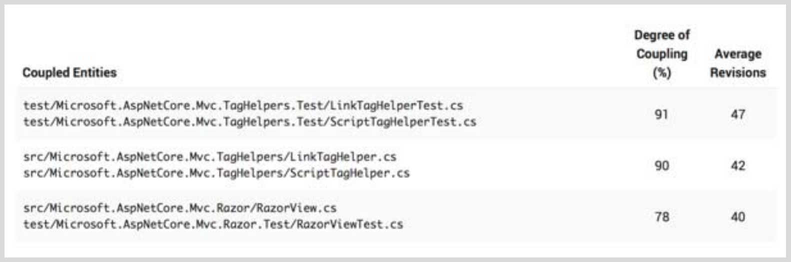
So you have the `ScriptTagHelper.cs` to the left and the `LinkTagHelper.cs` to your right. Do you see any pattern? Indeed, and this is what I tend to find on a regular basis as I inspect logical coupling – a dear old friend – copy-paste.

If you take a detailed look, you'll note something rare. The variable names and, even more rare, the comments have been updated so this is more like copy-paste with a gold plating (see Figure 6).

Break the logical dependencies

When you have an unexpected temporal dependency, you'll often find that there's some duplication of both code and knowledge. Extracting that common knowledge into a module of its own breaks the temporal coupling and makes your code a bit easier to maintain. You see, temporal coupling often suggests refactoring candidates.

At this point it's important to note that duplicated code in itself isn't always a problem; just because two pieces of code



Coupled Entities	Degree of Coupling (%)	Average Revisions
test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/LinkTagHelperTest.cs test/Microsoft.AspNetCore.Mvc.TagHelpers.Test/ScriptTagHelperTest.cs	91	47
src/Microsoft.AspNetCore.Mvc.TagHelpers/LinkTagHelper.cs src/Microsoft.AspNetCore.Mvc.TagHelpers/ScriptTagHelper.cs	90	42
src/Microsoft.AspNetCore.Mvc.Razor/RazorView.cs test/Microsoft.AspNetCore.Mvc.Razor.Test/RazorViewTest.cs	78	40

Figure 4

look the same, that doesn't mean they have to be refactored to use a shared abstraction. Instead we want to focus on what the code expresses.

In the case of ASP.NET MVC, it's clear that the two classes model the same process. So it is indeed a duplication of knowledge and it's likely that the code would benefit from a refactoring. This is even more important since, as the temporal coupling results indicate, we have the same amount of duplication between their corresponding unit tests. Avoiding expensive change patterns makes software maintenance much easier. Duplication of knowledge makes change much more expensive: it is easy to forget to update one of the copies when the business rules change or when a bug gets fixed.

Complement your intuition

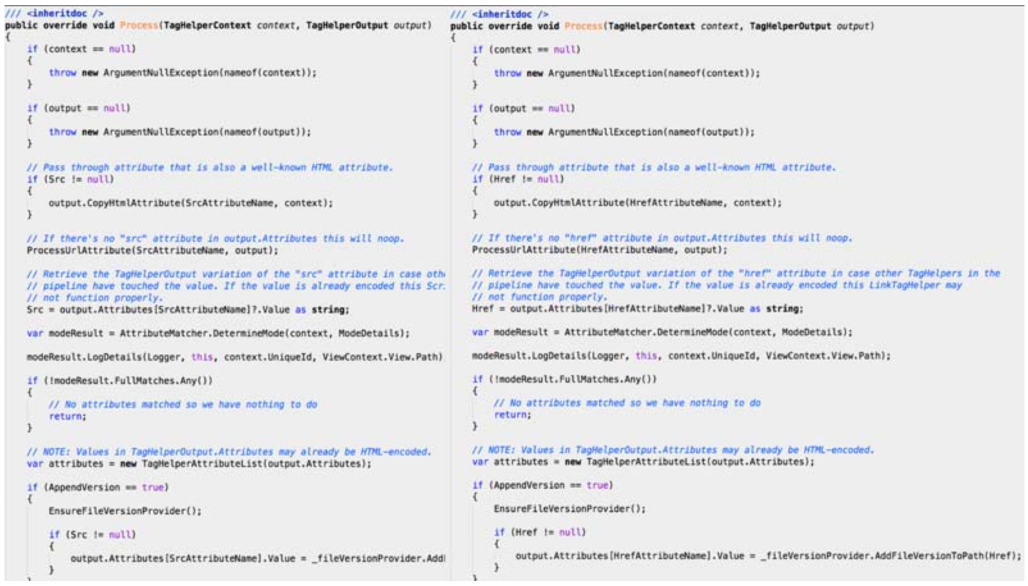
If you're an experienced developer who has contributed a lot of code to a particular project then you probably have a good feeling for where the most significant maintenance problems will show-up. You may still get surprised when you run an analysis, but in general several code analysis findings will match your intuitive guess. Temporal Coupling is different. We developers seem to completely lack all kind of intuitive sense when it comes to Temporal Coupling.

Conclusion

Exploring Temporal Coupling in our codebases often gives us deep and unexpected insights into how well our designs stand the test of time.

In this article we explored how Temporal Coupling detects a DRY violation in ASP.NET MVC. We ran an analysis with Empear Developer and looked for unexpected temporal dependencies to identify expensive change patterns in our code. Used that way, Temporal Coupling suggests both refactoring candidates and the need for new modular boundaries.

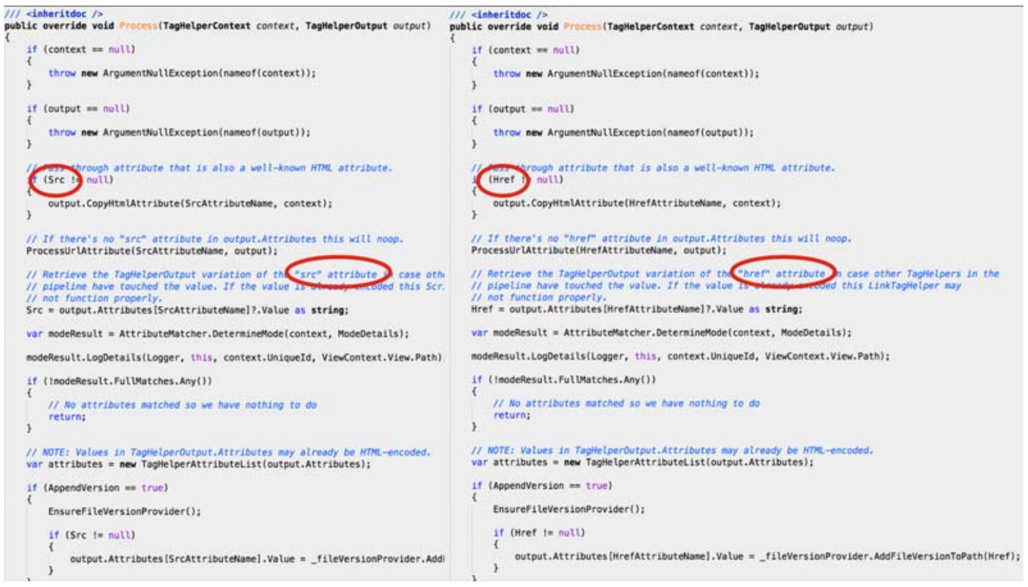
The same analysis principle also helps you catch architectural decay. All you have to do is to lookout for temporal dependencies that span architectural boundaries. Temporal Coupling is like bad weather – it gets worse with the distance you have to travel – and it makes a big difference if we need to modify two files located in the same package versus modifying files in different parts of the system. So make it a habit to investigate the temporal dependencies in your repository on a regular basis. Your code will thank you for it. ■



```
/// <inheritdoc />
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (output == null)
    {
        throw new ArgumentNullException(nameof(output));
    }

    // Pass through attribute that is also a well-known HTML attribute.
    if (Src != null)
    {
        output.CopyHtmlAttribute(SrcAttributeName, context);
    }

    // If there's no "src" attribute in output.Attributes this will noop.
    ProcessUrlAttribute(SrcAttributeName, output);

    // Retrieve the TagHelperOutput variation of the "src" attribute in case oth
    // pipeline have touched the value. If the value is already encoded this Scr.
    // not function properly.
    Src = output.Attributes[SrcAttributeName]?.Value as string;

    var modeResult = AttributeMatcher.DetermineMode(context, ModeDetails);

    modeResult.LogDetails(Logger, this, context.UniqueId, ViewContext.View.Path)

    if (!modeResult.FullMatches.Any())
    {
        // No attributes matched so we have nothing to do
        return;
    }

    // NOTE: Values in TagHelperOutput.Attributes may already be HTML-encoded.
    var attributes = new TagHelperAttributeList(output.Attributes);

    if (AppendVersion == true)
    {
        EnsureFileVersionProvider();

        if (Src != null)
        {
            output.Attributes[SrcAttributeName].Value = _fileVersionProvider.Addl
        }
```

```
/// <inheritdoc />
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (output == null)
    {
        throw new ArgumentNullException(nameof(output));
    }

    // Pass through attribute that is also a well-known HTML attribute.
    if (Href != null)
    {
        output.CopyHtmlAttribute(HrefAttributeName, context);
    }

    // If there's no "href" attribute in output.Attributes this will noop.
    ProcessUrlAttribute(HrefAttributeName, output);

    // Retrieve the TagHelperOutput variation of the "href" attribute in case other TagHelpers in the
    // pipeline have touched the value. If the value is already encoded this LinkTagHelper may
    // not function properly.
    Href = output.Attributes[HrefAttributeName]?.Value as string;

    var modeResult = AttributeMatcher.DetermineMode(context, ModeDetails);

    modeResult.LogDetails(Logger, this, context.UniqueId, ViewContext.View.Path);

    if (!modeResult.FullMatches.Any())
    {
        // No attributes matched so we have nothing to do
        return;
    }

    // NOTE: Values in TagHelperOutput.Attributes may already be HTML-encoded.
    var attributes = new TagHelperAttributeList(output.Attributes);

    if (AppendVersion == true)
    {
        EnsureFileVersionProvider();

        if (Href != null)
        {
            output.Attributes[HrefAttributeName].Value = _fileVersionProvider.AddFileVersionToPath(Href);
        }
```

Figure 5



```
/// <inheritdoc />
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (output == null)
    {
        throw new ArgumentNullException(nameof(output));
    }

    // Pass through attribute that is also a well-known HTML attribute.
    if (Src != null)
    {
        output.CopyHtmlAttribute(SrcAttributeName, context);
    }

    // If there's no "src" attribute in output.Attributes this will noop.
    ProcessUrlAttribute(SrcAttributeName, output);

    // Retrieve the TagHelperOutput variation of the "src" attribute in case oth
    // pipeline have touched the value. If the value is already encoded this Scr.
    // not function properly.
    Src = output.Attributes[SrcAttributeName]?.Value as string;

    var modeResult = AttributeMatcher.DetermineMode(context, ModeDetails);

    modeResult.LogDetails(Logger, this, context.UniqueId, ViewContext.View.Path)

    if (!modeResult.FullMatches.Any())
    {
        // No attributes matched so we have nothing to do
        return;
    }

    // NOTE: Values in TagHelperOutput.Attributes may already be HTML-encoded.
    var attributes = new TagHelperAttributeList(output.Attributes);

    if (AppendVersion == true)
    {
        EnsureFileVersionProvider();

        if (Src != null)
        {
            output.Attributes[SrcAttributeName].Value = _fileVersionProvider.Addl
        }
```

```
/// <inheritdoc />
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (output == null)
    {
        throw new ArgumentNullException(nameof(output));
    }

    // Pass through attribute that is also a well-known HTML attribute.
    if (Href != null)
    {
        output.CopyHtmlAttribute(HrefAttributeName, context);
    }

    // If there's no "href" attribute in output.Attributes this will noop.
    ProcessUrlAttribute(HrefAttributeName, output);

    // Retrieve the TagHelperOutput variation of the "href" attribute in case other TagHelpers in the
    // pipeline have touched the value. If the value is already encoded this LinkTagHelper may
    // not function properly.
    Href = output.Attributes[HrefAttributeName]?.Value as string;

    var modeResult = AttributeMatcher.DetermineMode(context, ModeDetails);

    modeResult.LogDetails(Logger, this, context.UniqueId, ViewContext.View.Path);

    if (!modeResult.FullMatches.Any())
    {
        // No attributes matched so we have nothing to do
        return;
    }

    // NOTE: Values in TagHelperOutput.Attributes may already be HTML-encoded.
    var attributes = new TagHelperAttributeList(output.Attributes);

    if (AppendVersion == true)
    {
        EnsureFileVersionProvider();

        if (Href != null)
        {
            output.Attributes[HrefAttributeName].Value = _fileVersionProvider.AddFileVersionToPath(Href);
        }
```

Figure 6

References

[ASP] https://github.com/aspnet/Mvc

[Empear] http://empear.com/offerings/developer-edition/

[Enterprise] http://empear.com/offerings/enterprise-edition/

[Tornhill15] Tornhill, Adam (2015) *Your Code as a Crime Scene*, Pragmatic Bookshelf, ISBN: 978-1-68050-038-7

Implementing SNAAAKE

Almost everyone knows the game Snake! Thaddaeus Frogley shares a diary of how his implementation grew over time.

Snake! Best practise...

Snake is a very old computer game, which appears to date back to an arcade game from 1976 called Blockade [Snake]. Some of our audience may recall playing a variant on a BBC machine or similar in the 1980s. Younger readers may have first encountered this game on an older Nokia mobile phone. You play by moving a dot (or similar shape) which gradually grows around the screen and lose when you run into yourself, edge of the screen or obstacle. There are many variants but I am used to one where 'eating' another object by running over it, makes the dots grow longer, thereby making the game more difficult as the snake grows.

Andy Balaam demonstrated how he'd implemented 'Snake!' in a variety of different languages at this year's ACCU conference, including Elm (Haskell for your browser) [Elm], a ZX spectrum emulator, Ruby, Python 3 with Qt5, Groovy, and Dart [Dart]. The idea of re-implementing something in several different ways is appealing. You learn by practising, and redoing the same thing a few times is a common learning technique. If you learn to play the piano, you practise your scales over and over. Some people try the same code kata over and over. Each time allows you to concentrate on improving at a different aspect of the task. Of course, if you implement a game, you can then play it afterwards. What could be better?

I am therefore inviting people to send their attempts at Snake!, or similar, to *Overload*. You had best get practising! In what follows, Thaddeus Frogley shares a diary of how his implementation in C++ using Emscripten developed. You can play it here: http://thad.frogley.info/snaaake/

Fran

My implementation of snake was a spare time project, which I worked on during the evenings after work. A day in the timeline represents anything from a few minutes to a few hours of work. Working on this, I made 77 commits over the course of 52 different days between Feb 11 and Dec 20. This is probably the equivalent of around 2 weeks worth of work at 'full time job' hours. Obviously, focus and flow impact productivity, so making comparisons like that is rather speculative.

The first 16 days worth of work is purely 'tech', none of what I'd call the 'game' is done until day 17, from then it's just 10 days until the game is basically done. From then until the end it's polish (fine tuning) and bug fixing, with no significant changes to how the game looks or feels.

That time breaks down roughly as first 30% on tech investment, then 20% on making the game and the last 50% on polish & bug fixing.

With unused source files removed, the dependency graph looks like Figure 1 (overleaf).

Development timeline

Day 1 (Tue Feb 11)
Setting up Emscripten [Emscripten], and getting a simple SDL [SDL] "hello world" program to compile and run in a browser.

Day 2 (Thu Feb 13)
Cobbled together from code copy-pasted from other projects, I set up an event loop, some math primitives, and classes for drawing simple shapes with OpenGL, to display some animated geometry to a window in the browser.

Day 3 (Sat Feb 15)
Switched from legacy immediate mode OpenGL to ES2 style rendering, using hard coded vertex and pixel shaders.

Day 4 (Mon Feb 17)
Added support for passing scale and translation through the shaders so that the shapes can be positioned on screen again. Made the shape classes hold a reference to their shader programs.

Day 5 (Tue Feb 18)
Added support for passing the colour used to draw into the shaders from the C++ code as a uniform [OpenGL].

Day 6 (Wed Feb 19)
Changed from passing position and scale as separate uniforms, to passing in a 4x4 matrix.

Day 7 (Thu Feb 20)
Moved all the math code copy-pasted into `src/geometry` on Day 2 into its own git repo, added that as a submodule in `libs/geometry` [Geometry].

Day 8 & 9 (Fri Feb 21, & Wed Feb 26)
Improvements to the matrix class interface in `src/geometry` to make it easier to integrate with the OpenGL code I'm writing. These changes introduce the first use of a c++11 feature, the **`initializer_list`**.

Day 10 (Sun Mar 2)
Creation of matrixes from separate translation, scale, and rotation values. The app now displays animated rotating stars and rings.

Day 11, 12, & 14 (Mon Mar 3, Sat Mar 22, & Sun Apr 27)
Improvements to the geometry code, mostly focused on making it easier to create matrixes for different uses.

Day 15 & 16 (Mon Apr 28, Tue Apr 29)
Added a 'quad' shape class, and set up a display grid of 84x48 to emulate the classic Nokia 3310 screen. Up until this point all the work has been preparatory, but now I have settled on making my snake clone a tribute to one of the original mobile phone versions. Using an array as a framebuffer, and then render it with single quad per pixel.

Day 17 (Wed Apr 30)
First gameplay work: Implemented the tracking of a position, moving in a direction writing to the framebuffer, and

Thaddeus Frogley Thaddeus started programming on the ZX81 when he was 7 years old, and has been hooked ever since. He has been working in the games industry for over 20 years. On Twitter he is @codemonkey_uk or reach him by email: thad@bossalien.com

A day in the timeline represents anything from a few minutes to a few hours of work ... focus and flow impact productivity

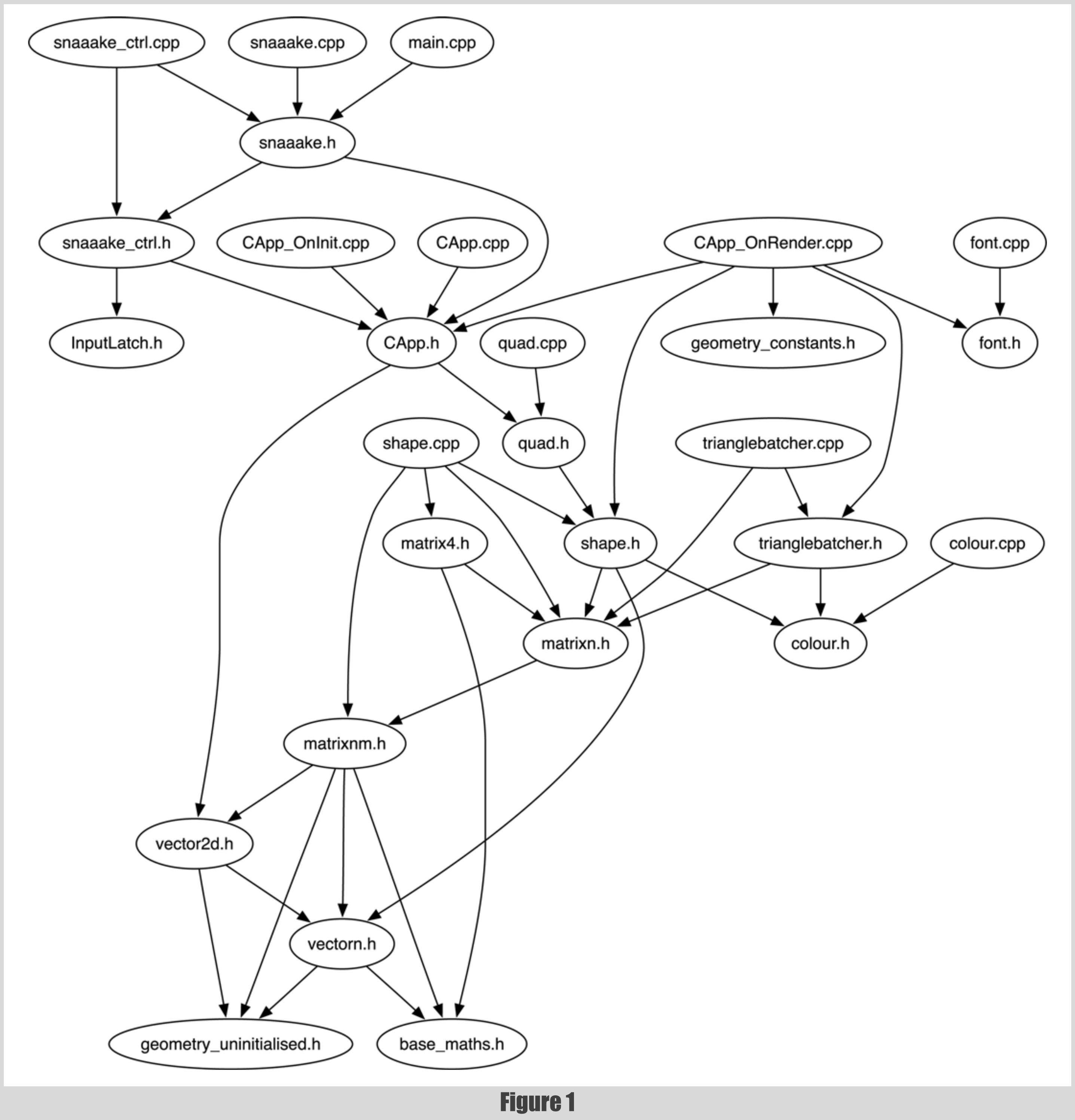


Figure 1

then detecting self-collisions and resetting the game. Add to that, the display of a score on screen, and I have a minimal snake game.

Day 18 & 19 (Thu May 1, Fri May 2)
The snake's tail now only grows when it eats a pick up. Snake wraps around the sides. Code rate of change slows as I spend more time testing the game-feel and thinking about making it fun.

Day 20 (Sun May 4)
Design a more compact score font. Each glyph in the font is defined as a simple array of 6 `const char*`, and the font is just an array of 10 glyphs.

Day 21 (Tue May 6)
Code and gameplay improvements around spawning food and poison pills.

Day 22 (Thu May 8)
A very important change to the keyboard event handling so players can u-turn reliably. Actions are queued, and processed one per frame. Switched to using `std::mt19937` over `rand()`, as `rand` was giving inconsistent, and in some cases highly predictable, behaviour across browsers. Did code housekeeping around resetting and restarting.

Day 23 (Thu May 15)
Added ability to grow the snake by arbitrary length for each food pick up collected.

Day 24–28 (Tue May 20, Wed 21, Thurs 22 and Fri 30)
Some refactoring, followed by making the snake 2 pixels thick, instead of 1. Made the spawns fat as well. Cornering with the thick snake was surprisingly tricky to get right. Played it a lot, then fixed a subtle bug in it.

Day 29 & 30 (Sat May 31, Sun Jun 1)
Optimisation: Performance was not ideal on all browsers. Implemented pre-calculating the geometry, and then batching the draw calls. OpenGL and WebGL draw calls can be very expensive. Batching is a common solution to this problem – we gather things than can be drawn in one call and submit them together. This reduces the number of calls per frame from 4032 to 3.

Day 31 (Mon Jun 2)
Moved the text drawing code into its own header and translation unit. Added a system to remove the poison pills when the snake collects food. Definitely an improvement. Fun to play.

Day 32 (Tue Jun 3)
Added ability to pause, with on screen status display.

Day 33 (Thu Jun 12)
Added an on screen game title, and improved removal of poison pills so that single pixel pieces don't get left behind.

Day 34 (Mon Jun 16)
Minor refactor of score tracking.

Day 35 (Wed Jun 25)
Updated build scripts, and in-game title so that the game is now called SNAAAKE.

Day 36–37 (Thu Jun 26, Wed Jul 2)
HTML wrapper work. Getting the game ready to be put online.

Day 38 (Sun Jul 6)
Double thickness snake behaviour still wasn't quite right. Fixed a bug with edge wrapping.

Day 39 (Fri Jul 11)
Refactoring, and code tidy up, followed by restricting the spawns from appearing while the snake is growing to improve pacing.

Day 40 (Sat Jul 12)
Changed the food and poison spawning logic to keep them from spawning on top of each other. Started work on an OS X native build.

Day 41 (Sun Jul 13)
Working on OS X project, icon, etc to have a native OS X binary distributable.

Day 42 & 43 (Mon Jul 14, Tue Jul 15)
Added on-screen (in-game) instructions, CMD-Q handler to quit in the OS X build, then added a 'favicon' and download link, for the OS X version, to the HTML5 container.

Day 44 (Wed Jul 16)
Refactoring, bug fixing, and presentation polish including the player-praise header text.

Day 45 (Fri Jul 18)
Refactoring, bug fixing, and presentation polish.

Day 46 (Mon Jul 21)
Added centre alignment support to the text rendering code to further improve the presentation of instructions.

Day 47 (Fri Jul 25)
Refactoring: Moved non-game event handling out of the game specific event handler.

Day 48 (Sat Aug 2)
Refactoring: Separated 'Controller' from 'Model', then implemented an AI 'Controller' to control the snake for an attract mode, and improved the instructions.

Day 49 (Sun Aug 3)
Bug fix: Prevented the AI from taking high scores.

Day 50 (Mon Aug 4)
Improved the attract mode AI by making it more natural/fallible and less predictable.

Day 51 (Tue Sep 2)
Optimised the thumbnail for the website (optipng).

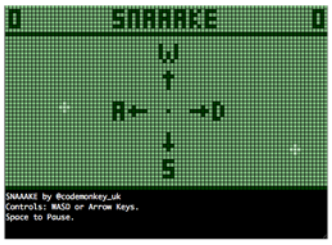
Day 52 (Sat Dec 20)
Added music (web version only).

Addendum, 2 years later …. **Day 53** (Sat May 11)
Finished and committed high score persistence and unused source file removal.

Conclusion

There is no conclusion. Spare time projects don't end, they just slow down until movement is undetectable. A space to practise and experiment is worthwhile for all programmers. But remember

- what's good for your personal projects isn't the same as what's good for team projects, and
- what's good for small projects isn't the same as what's good for large projects.



SNAAAKE by @codemonkey_uk
Controls: WASD or Arrow Keys.
Space to Pause.

References

[Dart] https://www.dartlang.org/

[Elm] http://elm-lang.org/

[Emscripten] https://github.com/kripken/emscripten

[Geometry] https://github.com/codemonkey-uk/geometry

[OpenGL] https://www.opengl.org/wiki/Uniform_(GLSL)

[SDL] https://www.libsdl.org

[Snake] https://en.wikipedia.org/wiki/Snake_(video_game)

The source code, with full revision history, can be found on github:
https://github.com/codemonkey-uk/snaaake

C++ Antipatterns

Certain mistakes crop up frequently in C++. Jonathan Wakely offers some pro-tips to help you avoid common errors.

This article documents some common mistakes that I see again and again in bug reports and requests for help on sites like StackOverflow.

Reading from an istream without checking the result

A very frequently asked question from someone learning C++ looks like this:

> I wrote this program but it doesn't work. It reads from the file correctly but does the calculation wrong.
>
> ```
> #include <iostream>
> #include <fstream>
>
> int calculate(int a, int b, int c)
> {
>   // blah blah blah complex calculation
>   return a + b + c;
> }
>
> int main()
> {
>   std::ifstream in("input.txt");
>   if (!in.is_open())
>   {
>     std::cerr << "Failed to open file\n";
>     return 1;
>   }
>
>   int i, j, k;
>   in >> i >> j >> k;
>   std::cout << calculate(i, j, k);
> }
> ```
>
> Why doesn't the calculation work?

In many, many cases the problem is that the `in >> ...` statement failed, so the variables contain garbage values and so the inputs to the calculation are garbage.

The program has no way to check the assumption 'it reads from the file correctly', so attempts to debug the problem are often just based on guesswork.

The solution is simple, but seems to be rarely taught to beginners: **always check your I/O operations**.

The improved version of the code in Listing 1 only calls `calculate(i, j, k)` if reading values into all three variables succeeds.

```
int i, j, k;

if (in >> i >> j >> k)
{
  std::cout << calculate(i, j, k);
}

else
{
  std::cerr <<
    "Failed to read values from the file!\n";
  throw std::runtime_error("Invalid input file");
}
```

Listing 1

Now if any of the input operations fails you don't get a garbage result, you get an error that makes the problem clear immediately. You can choose other forms of error handling rather than throwing an exception, the important bit is to check the I/O and not just keep going regardless when something fails.

Recommendation: always check that reading from an istream succeeds.

Locking and unlocking a std::mutex

This is **always** wrong:

```
std::mutex mtx;
void func()
{
  mtx.lock();
  // do things
  mtx.unlock();
}
```

It should **always** be done using one of the RAII scoped lock types such as `lock_guard` or `unique_lock` e.g.

```
std::mutex mtx;
void func()
{
  std::lock_guard<std::mutex> lock(mtx);
  // do things
}
```

Using a scoped lock is exception-safe, you cannot forget to unlock the mutex if you return early, and it takes fewer lines of code.

Recommendation: always use a scoped lock object to lock and unlock a mutex.

Jonathan Wakely Jonathan's interest in C++ and free software began at university and led to working in the tools team at Red Hat, via the market research and financial sectors. He works on GCC's C++ Standard Library and participates in the C++ standards committee. He can be reached at accu@kayari.org

The program has no way to check the assumption 'it reads from the file correctly', so attempts to debug the problem are often just based on guesswork

Be careful that you don't forget to give a scoped lock variable a name! This will compile, but doesn't do what you expect:

```
std::mutex mtx;
void func()
{
  std::unique_lock<std::mutex> (mtx);  // OOPS!
  // do things, but the mutex is not locked!
}
```

This default-constructs a **unique_lock** object called **mtx**, which has nothing to do with the global **mtx** object (the parentheses around **(mtx)** are redundant and so it's equivalent to simply **std::unique_lock<std::mutex> mtx;**).

A similar mistake can happen using braces instead of parentheses:

```
std::mutex mtx;
void func()
{
  std::unique_lock<std::mutex> {mtx};  // OOPS!
  // do things, but the mutex is not locked!
}
```

This *does* lock the global mutex **mtx**, but it does so in the constructor of a *temporary* **unique_lock**, which immediately goes away and unlocks the mutex again.

Testing for istream.eof() in a loop

A common mistake when using istreams is to try and use **eof()** to detect when there is no more input:

```
while (!in.eof())
{
  in >> x;
  process(x);
}
```

This doesn't work because the **eofbit** is only set *after* you try to read from a stream that has already reached EOF. When all the input has been read, the loop will run again, reading into **x** will fail, and then **process(x)** is called *even though nothing was read*.

The solution is to test whether the read succeeds, instead of testing for EOF:

```
while (in >> x)
{
  process(x);
}
```

You should never read from an istream without checking the result anyway, so doing that correctly avoids needing to test for EOF.

Recommendation: test for successful reads instead of testing for EOF

Inserting into a container of smart pointers with emplace_back(new X)

When appending to a **std::vector<std::unique_ptr<X>>**, you cannot just say **v.push_back(new X)**, because there is no implicit conversion from **X*** to **std::unique_ptr<X>**.

A popular solution is to use **v.emplace_back(new X)** because that compiles (**emplace_back** constructs an element in-place from the arguments, and so can use **explicit** constructors).

However, this is not safe. If the vector is full and needs to reallocate memory, that could fail and throw a **bad_alloc** exception, in which case the pointer will be lost and will never be deleted.

The safe solution is to create a temporary **unique_ptr** that takes ownership of the pointer before the vector might try to reallocate:

```
v.push_back(std::unique_ptr<X>(new X))
```

(You could replace **push_back** with **emplace_back** but there is no advantage here because the only conversion is explicit anyway, and **emplace_back** is more typing!)

In C++14 you should just use **std::make_unique** and it's a non-issue:

```
v.push_back(std::make_unique<X>())
```

Recommendation: do not prefer **emplace_back** just because it allows you to call an **explicit** constructor. There might be a good reason the class designer made the constructor **explicit** that you should think about and not just take a short cut around it.

(Scott Meyers discusses this point as part of Item 42 in *Effective Modern C++*.)

Defining 'less than' and other orderings correctly

When using custom keys in maps and sets, a common mistake is to define a 'less than' operation as in Listing 2.

This **operator<** does not define a valid ordering. Consider how it behaves for **X{1, 2}** and **X{2, 1}**:

```
X x1{1, 2};
X x2{2, 1};
assert( x1 < x2 );
assert( x2 < x1 );
```

The **operator<** defined above means that **x1** is less than **x2** but also that **x2** is less than **x1**, which should be impossible!

The problem is that the **operator<** says that **l** is less than **r** if *any* member of **l** is less than the corresponding member of **r**. That's like saying that 20 is less than 11 because when you compare the second digits '0' is less than '1' (or similarly, that the string "20" should be sorted before "11" because the second character '0' is less than the second character '1').

In technical terms the **operator<** above fails to define a *Strict Weak Ordering*.

Unordered containers require an equality operator, but that's harder to get wrong

```
struct X
{
  int a;
  int b;
};

inline bool operator<(const X& l, const X& r)
{
  if (l.a < r.a)
    return true;
  if (l.b < r.b)
    return true;
  return false;
}
```

Listing 2

Another way to define a bogus order is:

```
inline bool operator<(const X& l, const X& r)
{
  return l.a < r.a && l.b < r.b;
}
```

Where the first example gave the nonsensical result:

```
x1 < x2 && x2 < 1
```

this definition gives the result:

```
!(x1 < x2) && !(x1 < x2)
```

In other words, the two values are considered to be equivalent, and so only one of them could be inserted into a unique associative container such as a **`std::set`**. But then if you compare those values to **`X x3{1, 0}`**, you find that **`x1`** and **`x3`** are equivalent, but **`x1`** and **`x2`** are not. So depending which of **`x1`** and **`x2`** is in the **`std::set`** affects whether or not you can add **`x3`**!

An invalid order like the ones above will cause undefined behaviour if it is used where the Standard Library expects a correct order, e.g. in **`std::sort`**, **`std::set`**, or **`std::map`**. Trying to insert the **`x1`** and **`x2`** values above into a **`std::set<X>`** will give strange results, maybe even crashing the program, because the invariant that a set's elements are always sorted correctly is broken if the comparison function is incapable of sorting correctly.

This is discussed further in *Effective STL* by Scott Meyers, and in the *CERT C++ Coding Standard.*

A correct implementation of the function would only consider the second member when the first members are equal i.e.

```
inline bool operator<(const X& l, const X& r)
{
  if (l.a < r.a)
    return true;
  if (l.a == r.a && l.b < r.b)
    return true;
  return false;
}
```

Since C++11 defining an order correctly is trivial, just use **`std::tie`**:

```
inline bool operator<(const X& l, const X& r)
{
  return std::tie(l.a, l.b) < std::tie(r.a, r.b);
}
```

This creates a tuple referring to the members of **`l`** and a tuple referring to the members of **`r`**, then compares the tuples, which does the right thing (only comparing later elements when the earlier ones are equal).

Recommendation: When writing your own less-than operator make sure it defines a Strict Weak Ordering, and write tests to ensure that it doesn't give impossible results like **`(a < b && b < a)`** or **`(a < a)`**. Prefer using **`std::tie()`** to create tuples which can be compared, instead of writing your own error-prone comparison logic.

Consider whether defining a hash function and using either

- **`std::unordered_set`**
- **`std::unordered_map`**

would be better anyway than

- **`std::set`**
- **`std::map`**

Unordered containers require an equality operator, but that's harder to get wrong.

In summary

The only common theme to the items above is that I see these same mistakes made again and again. I hope documenting them here will help you to avoid them in your own code, and in code you review or comment on. If you know of other antipatterns that could be covered let me or the *Overload* team know about them, so they can be included in a follow-up piece (or write a follow up piece yourself !). ■

Testing Propositions

Is testing propositions more important than having examples as exemplars? Russel Winder considers this hypothesis.

With the rise of test-driven development (TDD) in the 1990s as part of the eXtreme Programming (XP) movement, the role of example-based testing became fixed into the culture of software development[1]. The original idea was to drive development of software products based on examples of usage of the product by end users. To support this Kent Beck and others at the centre of the XP community created test frameworks. They called them unit test frameworks presumably because they were being used to test the units of code that were being constructed. This all seemed to work very well for the people who had been in on the start of this new way of developing software. But then XP became well-known and fashionable: programmers other than the original cabal began to claim they were doing XP and its tool TDD. Some of them even bought the books written by Kent Beck and others at the centre of the XP community. Some of them even read said books.

Labels are very important. The test frameworks were labelled unit test frameworks. As all programmers know, units are functions, procedures, subroutines, classes, modules: the units of compilation. (Interpreted languages have much the same structure despite not being compiled per se.) Unit tests are thus about testing the units, and the tools for this are unit test frameworks. Somewhere along the line, connection between these tests and the end user scenarios got lost. Testing became an introvert thing. The whole notion of functional testing and 'end to end' testing seemed to get lost because the label for the frameworks were 'unit test'.

After a period of frustration with the lack of connection between end user scenarios and tests, some people developed the idea of acceptance testing so as to create frameworks and workflows. (Acceptance testing has been an integral part of most engineering disciplines for centuries; it took software development a while to regenerate the ideas.) FitNesse [FitNesse] and Robot [Robot] are examples of the sort of framework that came out of this period.

However the distance between acceptance testing and unit testing was still a yawning chasm[2]. Then we get a new entrant into the game, behaviour-driven development (BDD). This was an attempt by Dan North and others to recreate the way of using tests during development. The TDD of XP had lost its meaning to far too many programmers, so the testing frameworks for BDD were called JBehave, Cucumber, etc. and had no concept of unit even remotely associated with them.

Now whilst BDD reasserted the need for programmers and software developers to be aware of end user scenarios and at least pretend to care about user experience whilst implementing systems, we ended up with even more layers of tests and test frameworks.

And then came QuickCheck [QuickCheck], and the world of test was really shaken up: the term 'property-based testing' became a thing.

1. Into the culture of cultured developers, anyway.
2. Yes there is integration testing and system testing as well as unit testing and acceptance testing, and all this has been around in software, in principle at least, for decades, but only acceptance testing and unit testing had frameworks to support them. OK, technically FitNesse is an integration testing framework, but that wasn't how it was being used, and not how it is now advertised and used.

QuickCheck [Hackage] first appeared in work by John Hughes and others in the early 2000s. It started life in the Haskell [Haskell] community but has during the 2010s spread rapidly into the milieus of any programming language that even remotely cares about having good tests.

Example required

Waffling on textually is all very well, but what we really need is code; examples are what exemplify the points, exemplars are what we need. At this point it seems entirely appropriate to make some reuse, which, as is sadly traditional in software development, is achieved by cut and paste. So I have cut and paste[3] the following from a previous article for *Overload* [Winder16]:

> For this we need some code that needs testing: code that is small enough to fit on the pages of this august journal, but which highlights some critical features of the test frameworks.
>
> We need an example that requires testing, but that gets out of the way of the testing code because it is so trivial.
>
> We need factorial.
>
> Factorial is a classic example usually of the imperative vs. functional way of programming, and so is beloved of teachers of first year undergraduate programming courses. I like this example though because it allows investigating techniques of testing, and allows comparison of test frameworks.
>
> Factorial is usually presented via the recurrence relation:

$$f_0 = 1$$
$$f_n = nf_{n-1}$$

This is a great example, not so much for showing software development or algorithms, but for showing testing[4], and the frameworks provided by each programming language.

Given the Haskell heritage of property-based testing, it seems only right, and proper, to use Haskell for the first example. (It is assumed that GHC 7.10 or later (or equivalent) is being used.)

Russel Winder Ex-theoretical physicist, ex-UNIX system programmer, ex-academic. Now an independent consultant, analyst, author, expert witness and trainer. Also doing startups. Interested in all things parallel and concurrent. And build. Actively involved with Groovy, GPars, GroovyFX, SCons, and Gant. Also Gradle, Ceylon, Kotlin, D and bit of Rust. And lots of Python especially Python-CSP.

3. Without the footnotes, so if you want those you'll have to check the original. We should note though that unlike that article of this august journal, this is an August august journal issue, so very august.
4. OK so in this case this is unit testing, but we are creating APIs which are just units so unit testing is acceptance testing for all intents and purposes.

it seems to be idiomatic to have the type signature ... as a check that the function implementation is consistent with the stated signature

```
module Factorial(iterative, naïveRecursive,
  tailRecursive) where
exceptionErrorMessage = "Factorial not defined for
negative integers."

iterative :: Integer -> Integer
iterative n
    | n < 0 = error exceptionErrorMessage
    | otherwise = product [1..n]

naïveRecursive :: Integer -> Integer
naïveRecursive n
    | n < 0 = error exceptionErrorMessage
    | n == 0 = 1
    | otherwise = n * naïveRecursive (n - 1)

tailRecursive :: Integer -> Integer
tailRecursive n
    | n < 0 = error exceptionErrorMessage
    | otherwise = iteration n 1
    where
      iteration 0 result = result
      iteration i result = iteration (i - 1)
        (result * i)
```

Listing 1

Haskell implementation...

There are many algorithms for realizing the Factorial function: iterative, naïve recursive, and tail recursive are the most obvious. So as we see in Listing 1 we have three realizations of the Factorial function. Each of the functions starts with a type signature followed by the implementation. The type signature is arguably redundant since the compiler deduces all types. However, it seems to be idiomatic to have the type signature, not only as documentation, but also as a check that the function implementation is consistent with the stated signature. Note that in Haskell there are no function call parentheses – parentheses are used to ensure correct evaluation of expressions as positional arguments to function calls. It is also important to note that in Haskell functions are always curried: a function of two parameters is actually a function of one parameter that returns a function of one parameter. Why do this? It makes it really easy to partially evaluate functions to create other functions. The code of Listing 1 doesn't make use of this, but we will be using this feature shortly.

The `iterative` and `naïveRecursive` implementations are just matches with an expression: each match starts with a `|` and is an expression of Boolean value then a `=` followed by the result expression to evaluate for that match expression. Matches are tried in order and `otherwise` is the 'catch all' "Boolean" that always succeeds; it should, of course, be the last in the sequence. The `error` function raises an exception to be handled elsewhere. The `tailRecursive` function has a match and also a 'where clause' which defines the function `iteration` by pattern matching on the parameters. The 'where clause' definitions are scoped to the function of definition[5,6].

...and example-based test

Kent Beck style TDD started in Smalltalk with sUnit[7] and then transferred to Java with JUnit[8]. A (thankfully fading) tradition seems to have grown that the first test framework in any language is constructed in the JUnit3 architecture – even if this architecture is entirely unsuitable, and indeed not idiomatic, for the programming language. Haskell seem to have neatly side-stepped the problem from the outset since although the name is HUnit [HUnit] as required by the tradition, the architecture is nothing at all like JUnit3. Trying to create the JUnit3 architecture in Haskell would have been hard and definitely not idiomatic, HUnit is definitely idiomatic Haskell.

Listing 2 shows the beginnings of a test using a table driven (aka data driven) approach. It seems silly to have to write a new function for each test case, hence the use of a table (`positiveData`) to hold the inputs and outputs and create all the tests with a generator (`testPositive`, a function of two parameters, the function to test and a string unique to the function so as to identify it). The function `test` takes a list argument with all the tests, here the list is being constructed with a list comprehension: the bit before the `|` is the value to calculate in each case (a fairly arcane expression, but lets not get too het up about it) and the expression after is the 'loop' that drives the creation of the different values, in this case create a list entry for each pair in the table. Then we have a sequence (thanks to the `do` expression[9]) of three calls to `runTestTT` (a function of one parameter) which actually runs all the tests.

Of course, anyone saying to themselves "but he hasn't tested negative values for the arguments of the Factorial functions", you are not being silly; you are being far from silly, very sensible in fact. I am avoiding this aspect of the testing here simply to avoid some Haskell code complexity[10] that adds nothing to the flow in this article. If I had used Python or Java

5. If you need a tutorial introduction to the Haskell programming language then http://learnyouahaskell.com/ and http://book.realworldhaskell.org/ are recommended.
6. If you work with the JVM and want to use Haskell, there is Frege; see http://www.frege-lang.org or https://github.com/Frege/frege Frege is a realization of Haskell on the JVM that allows a few extensions to Haskell so as to work harmoniously with the Java Platform.
7. The name really does give the game away that the framework was for unit testing.
8. Initially called JUnit, then when JUnit4 came out JUnit was renamed JUnit3 as by then it was at major version 3. Now of course we have JUnit5.
9. Yes it's a monad. Apparently monads are difficult to understand, and when you do understand them, they are impossible to explain. This is perhaps an indicator of why there are so many tutorials about monads on the Web.
10. Involving Monads. Did I mention about how once you understand monads, you cannot explain them?

The proposition of proposition-based testing is to make propositions about the code and then use random selection of values from the domain to check the propositions are not invalid

```
module Main where

import Test.HUnit

import Factorial

positiveData = [
  (0, 1),
  (1, 1),
  (2, 2),
  (3, 6),
  (4, 24),
  (5, 120),
  (6, 720),
  (7, 5040),
  (8, 40320),
  (9, 362880),
  (10, 3628800),
  (11, 39916800),
  (12, 479001600),
  (13, 6227020800),
  (14, 87178291200),
  (20, 2432902008176640000),
  (30, 265252859812191058636308480000000),
  (40,
815915283247897734345611269596115894272000000000)
]

testPositive function comment =
    test [comment ++ " " ++ show i ~: "" ~:
    expected ~=? function i |
    (i, expected) <- positiveData]

main = do
 runTestTT (testPositive Factorial.iterative
   "Iterative")
 runTestTT (testPositive Factorial.naïveRecursive
   "Naïve Recursive")
 runTestTT (testPositive Factorial.tailRecursive
   "Tail Recursive")
```

Listing 2

(or, indeed, almost any language other than Haskell) we would not have this issue. For those wishing to see the detail of a full test please see my Factorial repository on GitHub [Winder].

And the proposition is…

The code of Listing 2 nicely shows that what we are doing is selecting values from the domain of the function and ensuring the result of executing the function is the correct value from the image of the function[11]. This is really rather an important thing to do but are we doing it effectively?

Clearly to *prove* the implementation is correct we have to execute the code under test with every possible value of the domain. Given there are roughly 2^{64} (about 18,446,744,073,709,551,616) possible values to test on a 64-bit machine, we will almost certainly decide to give up immediately, or at least within just a few femtoseconds. The test code as shown in Listing 2 is sampling the domain in an attempt to give us confidence that our implementation is not wrong. Have we done that here? Are we satisfied? Possibly yes, but could we do more quickly and easily?

The proposition of proposition-based testing is to make propositions about the code and then use random selection of values from the domain to check the propositions are not invalid. In this case of testing the Factorial function, what are the propositions? Factorial is defined by a recurrence relation comprising two rules. These rules describe the property of the function that is Factorial with respect to the domain, the non-negative integers. If we encode the recurrence relation as a predicate (a Boolean valued function) we have a representation of the property that can be tested by random selection of non-negative integers.

Listing 3 shows a QuickCheck test of Factorial. The function **`f_p`** is the predicate representing the property being tested. It is a function of two parameters, a function to test and a value to test, with the result being whether the recurrence relation that defines Factorial is true for that value and that function: the predicate is an assertion of the property that any function claiming to implement the Factorial function must satisfy. Why is this not being used directly, but instead **`factorial_property`** is the predicate being tested by the calls to **`quickCheck`**? It is all about types and the fact that values are automatically generated for us based on the domain of the property being tested. **`f_p`** is a predicate dealing with **`Integer`**, the domain of the functions being tested, values of which can be negative. Factorial is undefined for negative values[12]. So the predicate called by **`quickCheck`**, **`factorial_property`**, is defined with **`Natural`** as the domain, i.e. for non-negative integers[13]. So when we execute **`quickCheck`** on the function under test, it is non-negative integer values that are generated: The predicate never needs to deal with negative values, it tests just the Factorial proposition and worries not about handling the exceptions that the implementations raise on being given a negative argument. Should we test for negative arguments and that an exception is generated? Probably. Did I mention ignoring this for now?

Earlier I mentioned currying and partial evaluation. In Listing 3, we are seeing this in action. The argument to each **`quickCheck`** call is an expression that partially evaluates **`factorial_property`**, binding a

11. Pages such as https://en.wikipedia.org/wiki/Domain_of_a_function and https://en.wikipedia.org/wiki/Image_(mathematics) may be handy if you are unused to the terminology used here.
12. And also non-integral types, do not forget this in real testing.
13. If you are thinking we should be setting up a property to check that all negative integers result in an error, you are thinking on the right lines.

Shrinking is such a boon ... that it is now seen as essential for any property-based testing framework

```
module Main where

import Numeric.Natural
import Test.QuickCheck

import Factorial

f_p :: (Integer -> Integer) -> Integer -> Bool
f_p f n
    | n == 0 = f n == 1
    | otherwise = f n == n * f (n - 1)

factorial_property :: (Integer -> Integer) ->
  Natural -> Bool
factorial_property f n = f_p f (fromIntegral n)

main :: IO()
main = do
  quickCheck (factorial_property iterative)
  quickCheck (factorial_property naïveRecursive)
  quickCheck (factorial_property tailRecursive)
```

Listing 3

particular implementation of `Factorial`, and returning a function that takes only a `Natural` value. This sort of partial evaluation is a typical and idiomatic technique of functional programming, and increasingly any language that supports functions as first class entities.

By default QuickCheck selects 100 values from the domain, so Listing 3 is actually 300 tests. In the case we have here there are no fails, all 300 tests pass. Somewhat splendidly, if there is a failure of a proposition, QuickCheck sets about 'shrinking' which means searching for the smallest value in the domain for which the proposition fails to hold. Many people are implementing some form of proposition testing in many languages. Any not having shrinking are generally seen as being not production ready. Shrinking is such a boon to taking the results of the tests and deducing (or more usually inferring) the cause of the problem, that it is now seen as essential for any property-based testing framework.

Figure 1 shows the result of running the two test programs: first the HUnit example based testing – 18 hand picked tests for each of the three implementations; and second the QuickCheck property-based testing – 100 tests for each case, all passing so no need for shrinking.

But who uses Haskell?

Well, quite a lot of people. However, one of the major goals of Haskell is to 'Avoid success at all costs'[14]. The point here is not un-sensible. Haskell is a language for exploring and extending ideas and principles of functional programming. The Haskell committee therefore needs to avoid having to worry about backward compatibility. This puts it a bit at odds with many commercial and industrial operations who feel that, once written, a line of code should compile (if that is appropriate) and execute exactly the same for all time without any change. Clearly this can be achieved easily in any language by never upgrading the toolchain. However, the organizations that demand code works for all time usually demand that toolchains are regularly updated. (Otherwise the language is considered dead and unusable. There is irony in here somewhere I believe.) There is no pleasing some people. Successful languages in the sense of having many users clearly have to deal with backward compatibility. Haskell doesn't. Thus Haskell, whilst being a very important language, doesn't really have much market traction.

Frege makes an entry

Frege [Frege] though is actually likely to get more traction than Haskell. Despite the potential for having to update codebases, using 'Haskell on the JVM' is an excellent way of creating JVM-based systems. And because the JVM is a polyglot platform, bits of systems can be in Java, Frege, Kotlin [Kotlin], Ceylon [Ceylon], Scala [Scala], Apache Groovy [Groovy], etc. For anyone out there using the Java Platform, I can strongly recommend at least trying Frege. To give you a taste, look at Listing 4, which shows three Frege implementations of the Factorial function, and that Frege really is Haskell. The tests (see Listing 5) are slightly different from the Haskell ones not because the languages are different but because the context is: instead of creating a standalone executable as happens with Haskell, Frege create a JVM class to be managed by a test runner. So instead of a `main` function calling the test executor, we just declare property instances for running using the `property` function, and assume the test runner will do the right thing when invoked. The three examples here show a different way of constraining the test domain to non-negative integers than we saw with Haskell. Function composition (`.` operator, must have spaces either side to distinguish it from member selection) of the property function (using partial evaluation) with a test data generator (`NonNegative.getNonNegative`; dot as selector not function composition) shows how easy all this can be. Instead of just using the default generator (which would be `Integer` for this property function `factorial_property`, we are providing an explicit generator so as to condition the values from the domain that get generated.

```
$ ./factorial_test_hunit
Cases: 18  Tried: 18  Errors: 0  Failures: 0
Cases: 18  Tried: 18  Errors: 0  Failures: 0
Cases: 18  Tried: 18  Errors: 0  Failures: 0

$ ./factorial_test_quickcheck
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Figure 1

14. A phrase initially spoken by Simon Peyton Jones a number of years ago that caught on in the Haskell community.

it behoves us to consider the proposition of proposition testing in one or more languages that have already gained real traction

```
module Factorial where

exceptionErrorMessage = "Factorial not defined for
negative integers."

iterative :: Integer -> Integer
iterative n
    | n < 0 = error exceptionErrorMessage
    | otherwise = product [1..n]

naïveRecursive :: Integer -> Integer
naïveRecursive n
    | n < 0 = error exceptionErrorMessage
    | n == 0 = 1
    | otherwise = n * naïveRecursive (n - 1)

tailRecursive :: Integer -> Integer
tailRecursive n
    | n < 0 = error exceptionErrorMessage
    | otherwise = iteration n  1
    where
      iteration 0 result = result
      iteration i result = iteration (i - 1)
        (result * i)
```

Listing 4

```
module Factorial_Test where

import Test.QuickCheck(quickCheck, property)
import Test.QuickCheckModifiers(NonNegative)

import Factorial(iterative, naïveRecursive,
  tailRecursive)

factorial_property :: (Integer -> Integer)
  -> Integer -> Bool
factorial_property f n
    | n == 0 = f n == 1
    | otherwise = f n == n * f (n - 1)

factorial_iterative_property =
  property ((factorial_property iterative)
  . NonNegative.getNonNegative)
factorial_naïveRecursive_property =
  property ((factorial_property naïveRecursive)
  . NonNegative.getNonNegative)
factorial_tailRecursive_property =
  property ((factorial_property tailRecursive)
  . NonNegative.getNonNegative)
```

Listing 5

```
Factorial_Test.factorial_tailRecursive_property:
  +++ OK, passed 100 tests.
Factorial_Test.factorial_iterative_property:
  +++ OK, passed 100 tests.
Factorial_Test.factorial_naïveRecursive_property:
  +++ OK, passed 100 tests.
Properties passed: 3, failed: 0
```

Figure 2

The result of executing the Frege QuickCheck property-based tests are seen in Figure 2. As with the Haskell, 100 samples for each test with no fails and so no shrinking.

But...

With Haskell trying not to have a user base reliant on backward compatibility, and Frege not yet having quite enough traction as yet to be deemed popular, it behoves us to consider the proposition of proposition testing in one or more languages that have already gained real traction.

First off let us consider…Python.

Let’s hypothesize Python

Python [Python_1] has been around since the late 1980s and early 1990s. During the 2000s it rapidly gained in popularity. And then there was the ‘Python 2 / Python 3 Schism’.[15] After Python 3.3 was released, there were no excuses for staying with Python 2. (Well, except two – and I leave it as an exercise for the reader to ascertain what these two genuine reasons are for not immediately moving your Python 2 code to Python 3.) For myself, I use Python 3.5 because Python now has function signature type checking [Python_2][16].

Listing 6 shows four implementations of the Factorial function. Note that the function signatures are advisory not strong type checking. Using the MyPy [MyPy] program the types will be checked, but on execution it is just standard Python as people have known for decades.

I suspect the Python code here is sufficiently straightforward that almost all programmers[17] will be able to deduce or infer any meanings that are not immediately clear in the code. But a few comments to help: the `range` function generates a range ‘from up to but not including’. The `if` expression is of the form:

15. We will leave any form of description and commentary on the schism to historians. As Python programmers, we use Python 3 and get on with programming.
16. This isn’t actually correct: Python allows function signatures as of 3.5 but doesn’t check them. You have to have a separate parser-type-checker such as MyPy. This is annoying, Python should be doing the checking.
17. We will resist the temptation to make some facetious, and likely offensive, comment about some programmers who use only one programming language and refuse to look at any others. “Resistance is futile.” Seven of Nine.

not only are we testing non-negative and negative integers, we also test other forms of error that are possible in Python

```
  <true-value> if <boolean-expression>
    else <false-value>
```

The nested function iterate in **`tail_recursive`** is scoped to the else block.

But are these implementations ‘correct’? To test them let’s use PyTest [Pytest]. The test framework that comes as standard with Python (unittest, aka PyUnit) could do the job, but PyTest is just better[18]. PyTest provides an excellent base for testing but it does not have property-based testing.

```
from functools import reduce
from operator import mul

def _validate(x: int) -> None:
  if not isinstance(x, int):
    raise TypeError('Argument must be an 
integer.')
    if x < 0:
      raise ValueError('Argument must be a 
non-negative integer.')

def iterative(x: int) ->int:
  _validate(x)
  if x < 2:
    return 1
  total = 1
  for i in range(2, x + 1):
    total *= i
  return total

def recursive(x: int) -> int:
  _validate(x)
  return 1 if x < 2 else x * recursive(x - 1)

def tail_recursive(x: int) -> int:
  _validate(x)
  if x < 2:
    return 1
  else:
    def iterate(i: int, result: int=1):
      return result if i < 2 else iterate(i - 1,
        result * i)
    return iterate(x)

def using_reduce(x: int) -> int:
    _validate(x)
    return 1 if x < 2 else reduce(mul,
      range(2, x + 1))
```

Listing 6

18. For reasons that may, or may not, become apparent in this article, but relate to PyUnit following JUnit3 architecture – remember the fading tradition – and PyTest being Pythonic.

For this we will use Hypothesis [Hypothesis] (which can be used with PyUnit as easily as with PyTest, but PyTest is just better).

Listing 7 shows a fairly comprehensive test – not only are we testing non-negative and negative integers, we also test other forms of error that are possible in Python. Tests are functions with the first four characters of the name being t, e, s, t. Very JUnit3, and yet these are module-level

```
from pytest import mark, raises

from hypothesis import given
from hypothesis.strategies import (integers,
  floats, text)

from factorial import (iterative, recursive,
  tail_recursive, using_reduce)

algorithms = (iterative, using_reduce, recursive,
  tail_recursive)

@mark.parametrize('a', algorithms)
@given(integers(min_value=0, max_value=900))
def test_with_non_negative_integer (a, x):
  assert a(x) == (1 if x == 0 else x * a(x - 1))

@mark.parametrize('a', algorithms)
@given(integers(max_value=-1))
def test_negative_integer_causes_ValueError(a, x):
  with raises(ValueError):
    a(x)

@mark.parametrize('a', algorithms)
@given(floats())
def test_float_causes_TypeError(a, x):
  with raises(TypeError):
    a(x)

@mark.parametrize('a', algorithms)
def test_none_causes_TypeError(a):
  with raises(TypeError):
    a(None)

@mark.parametrize('a', algorithms)
@given(text())
def test_string_causes_TypeError(a, x):
    with raises(TypeError):
        a(x)

if __name__ == '__main__':
  from pytest import main
  main()
```

Listing 7

automated data generation is at the heart of property-based testing

```
============================= test session starts ==============================
platform linux -- Python 3.5.1, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /home/users/russel/Docs/Papers/ACCU/Draft/TestingPropositions/SourceCode/Python, inifile:
plugins: hypothesis-3.4.0, cov-2.2.1
collected 20 items

test_factorial.py ....................

========================== 20 passed in 2.32 seconds ===========================
```

Figure 3

functions. There are no classes or inheritance in sight: that would be the PyUnit way. The PyTest way is to dispense with the classes as necessary infrastructure, swapping them for needing some infrastructure to be imported in some way or other. (This is all handled behind the scenes when pytest.main executes.) PyTest is in so many ways more Pythonic[19] than PyUnit.

PyTest has the **`@mark.parametrize`** decorator that rewrites your code so as to have one test per item of data in an iterable. In all the cases here, it is being used to generate tests for each algorithm[20].

The **`@given`** decorator, which comes from Hypothesis, does not rewrite functions to create new test functions. Instead it generates code to run the function it decorates with a number (the default is 100) of randomly chosen values using the generator given as argument to the decorator, recording the results to report back. This automated data generation is at the heart of property-based testing, and Hypothesis, via the supporting functions such as **`integers`**, **`floats`**, and **`text`** (for generating integers, floats, and string respectively), does this very well. Notice how it is so easy to generate just negative integers or just non-negative integers. Also note the use of the 'with statement'[21] and the **`raises`** function for testing that code does, in fact, raise an exception.

All the test functions have a parameter **`a`** that gets bound by the action of the **`@mark.parametrize`** decorator, and a parameter **`x`** that gets bound by the action of the **`@given`** decorator. This is all very different from the partial evaluation used in Haskell and Frege: different language features lead to different idioms to achieve the same goal. What is Pythonic is not Haskellic/Fregic, and vice versa. At least not necessarily.

The **`pytest.main`** function, when executed, causes all the decorators to undertake their work and executes the result. The output from an execution will look very much as in Figure 3. You may find when you try this that the last line is green.[22]

Doing the C++ thing

There are many other example languages we could present here to show the almost complete coverage of property-based testing in the world: Kotlin [Kotlin], Ceylon [Ceylon], Scala [Scala], Apache Groovy [Groovy], Rust [Rust], D [D], Go [Go],… However, given this is an August[23] ACCU journal and, historically at least, ACCU members have had a strong interest in C++, we should perhaps look at C++. Clearly people could just use Haskell and QuickCheck to test their C++ code, but let's be realistic here, that isn't going to happen[24]. So what about QuickCheck in C++? There are a number of implementations, for example CppQuickCheck [QuickCheck_2] and QuickCheck++ [QuickCheck_3]. I am, though, going to use RapidCheck [RapidCheck] here because it seems like the most sophisticated and simplest to use of the ones I have looked at to date[25].

There is one thing we have to note straight away: Factorial values are big[26]. Factorial of 30 is a number bigger than can be stored in a 64-bit integer. So all the implementations of Factorial used in books and first year student exercises are a bit of a farce because they are shown using hardware integers: the implementations work for arguments [0..20] and then things get worrisome. "But this is true for all languages and we didn't raise this issue for Haskell, Frege and Python." you say. Well for Haskell (and Frege, since Frege is just Haskell on the JVM) the **`Int`** type is a hardware number but **`Integer`**, the type used in the Haskell and Frege code, is an integer type the values of which can be effectively arbitrary size. There is a limit, but then in the end even the universe is finite[27]. What about Python? The Python[28] **`int`** type uses hardware when it can or an unbounded (albeit finite[27]) integer when it cannot. What about C++? Well

19. See http://docs.python-guide.org/en/latest/writing/style/
20. There are ways of parameterizing tests in PyUnit (aka unittest), but it is left as an exercise for the reader to look for these. PyTest and **`@pytest.mark.parametrize`** are the way this author chooses to do parameterized tests in Python.
21. Context managers and the 'with statement' are Python's way of doing RAII (resource acquisition is initialization, https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization) amongst other great things.
22. Whilst this is an August august journal (and so very august), it is monochrome. So you will have to imagine the greenness of the test output. Either that or actually try the code out for yourself and observe the greenness first hand.
23. Or should that be august. Well actually it has to be both.
24. Not least because Haskell's avowed aim is never to be successful.
25. Also it uses Catch [Catch] for its tests.
26. Factorials are big like space is big, think big in *Hitchhiker's Guide to the Galaxy* terms: "Space," it says, "is big. Really big. You just won't believe how vastly, hugely, mindbogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space. Listen…?"
https://en.wikiquote.org/wiki/The_Hitchhiker%27s_Guide_to_the_Galaxy

Space may be big but the universe (space being the same thing as the universe as far as we know) is finite

```
#include <gmpxx.h>

namespace Factorial {

mpz_class iterative(mpz_class const n);
mpz_class iterative(long const n);
mpz_class reductive(mpz_class const n);
mpz_class reductive(long const n);
mpz_class naive_recursive(mpz_class const n);
mpz_class naive_recursive(long const n);
mpz_class tail_recursive(mpz_class const n);
mpz_class tail_recursive(long const n);

} // namespace Factorial
```

Listing 8

the language and standard library have only hardware-based types, which could be taken as rather restricting. GNU has however conveniently created a C library for unbounded (albeit finite[27]) integers, and it has a rather splendid C++ binding [GNU].

So using the GMP C++ API, we can construct implementations of the Factorial function that are not restricted to arguments in the range [0..20] but are more generally useful. Listing 8 shows the functions being exported by the `Factorial` namespace. We could dispense with the `long` overloads, but it seems more programmer friendly to offer them.

Listing 9 presents the implementations. I suspect that unless you already know C++ (this code is C++14) you have already moved on. So any form of explanatory note is effectively useless here.[29] We will note though that

```
#include "factorial.hpp"

#include <functional>
#include <iterator>
#include <numeric>

namespace Factorial {
static void validate(mpz_class const n) {
  if (n < 0) {
    throw std::invalid_argument("Parameter must be
a non-negative integer."); }
}
auto const one = mpz_class(1);
auto const two = mpz_class(2);
```

Listing 9

```
mpz_class iterative(mpz_class const n) {
  validate(n);
  mpz_class total {1};
  for (unsigned int i = 2; i <= n; ++i) {
    total *= i; }
  return total;
}
mpz_class iterative(long const n) {
  return iterative(mpz_class(n)); }

class mpz_class_iterator:
  std::iterator<std::input_iterator_tag,
  mpz_class> {
  private:
    mpz_class value;
  public:
    mpz_class_iterator(mpz_class const v) :
      value(v) { }
    mpz_class_iterator& operator++() {
      value += 1; return *this; }
    mpz_class_iterator operator++(int) {
      mpz_class_iterator tmp {
        *this}; ++*this; return tmp; }
      bool operator==(mpz_class_iterator const &
        other) const {
        return value == other.value; }
      bool operator!=(mpz_class_iterator const &
        other) const {
        return value != other.value; }
    mpz_class operator*() const { return  value; }
    mpz_class const * operator->() const {
      return &value; }
};

mpz_class reductive(mpz_class const n) {
  validate(n);
  return (n < 2)
  ? one
  : std::accumulate(mpz_class_iterator(two),
    mpz_class_iterator(n + 1), one,
      std::multiplies<>());}
mpz_class reductive(long const n) {
  return reductive(mpz_class(n)); }
mpz_class naive_recursive(mpz_class const n) {
  validate(n);
  return (n < 2) ? one :
    n * naive_recursive(n - 1);
}
```

Listing 9 (cont'd)

27. Space may be big (see above) but the universe (space being the same thing as the universe as far as we know) is finite – assuming the current theories are correct.
28. Python 3 anyway. Python 2 has effectively the same behaviour, but with more types. It is left as an exercise for the reader whether to worry about this.
29. There was some thought of introducing the acronym RTFC (read the fine code), but this temptation was resisted. “Resistance is futile.” Seven of Nine.

You can have **any number** of parameters **– zero has been chosen here**, which might seem a bit strange at first

```
mpz_class naive_recursive(long const n) {
  return naive_recursive(mpz_class(n)); }

static mpz_class tail_recursive_iterate
  (mpz_class const n, mpz_class const result) {
  return (n < 2) ? result :
    tail_recursive_iterate(n - 1, result * n);
}

mpz_class tail_recursive(mpz_class const n) {
  validate(n);
  return (n < 2) ? one : tail_recursive_iterate(n,
    one);
}
mpz_class tail_recursive(long const n) {
  return tail_recursive(mpz_class(n)); }
} // namespace Factorial
```

Listing 9 (cont'd)

there is a class defined in there as well as implementations of the Factorial function.

Listing 10 presents the RapidCheck-based test code for the Factorial functions. There is a vector of function pointers[30] so that we can easily iterate over the different implementations. Within the loop we have a sequence of the propositions. Each check has a descriptive string and a lambda function. The type of variables to the lambda function will cause (by default 100) values of that type to be created and the lambda executed for each of them. You can have any number of parameters – zero has been chosen here, which might seem a bit strange at first, but think generating random integers. Some of them are negative and some non-negative and we have to be careful to separate these cases as the propositions are so very different. Also some of the calculation for non-negative integers will result in big values. The factorial of a big number is stonkingly big. Evaluation will take a while… a long while… a very long while… so long we will have read *War and Peace*… a large number of times. So we restrict the integers of the domain sample by using an explicit generator. In this case for the non-negative integers we sample from [0..900]. For the negative integers we sample from a wider range as there should only ever be a very rapid exception raised, there should never actually be a calculation.

So that is the Factorial functions themselves tested. I trust you agree that what we have here is a very quick, easy, and providing good coverage test. But, you ask, what about that class? Should we test the class? An interesting question. Many would say "No" because it is internal stuff, not exposed as part of the API. This works for me: why test anything that is not observable from outside. Others will say "Yes" mostly because it cannot hurt. For this article I say "Yes" because it provides another example of proposition-based testing. We do not test any examples, we test only properties of the class and its member functions. See Listing 11.

30. Well, actually pairs, with the first being the function pointer and the second being a descriptive string.

```
#include "rapidcheck.h"

#include <string>
#include <utility>

#include "factorial.hpp"

std::vector<std::pair<mpz_class (*)(long const),
std::string>> const algorithms {
  {Factorial::iterative, "iterative"},
  {Factorial::reductive, "reductive"},
  {Factorial::naive_recursive, "naïve recursive"},
  {Factorial::tail_recursive, "tail recursive"}
};

int main() {
  for (auto && a: algorithms) {
    auto f = a.first;

    rc::check(a.second + " applied to non-negative
      integer argument obeys the recurrence
      relation.", [f]() {
      auto i = *rc::gen::inRange(0, 900);
      RC_ASSERT(f(i) == ((i == 0) ? mpz_class(1) :
        i * f(i - 1)));
    });

    rc::check(a.second + " applied to negative
       integer raises an exception.", [f]() {
      auto i = *rc::gen::inRange(-100000, -1);
      RC_ASSERT_THROWS_AS(f(i),
        std::invalid_argument);
    });
  }
  return 0;
}
```

Listing 10

By testing the properties, we are getting as close to proving the implementation not wrong as it is possible to get in an easily maintainable way. QED.

And to prove that point, see Figure 4, which shows the Factorial tests and class test executed. So many useful (passing) tests, so little effort.

The message

Example-based testing of a sample from the domain tells us we are calculating the correct value(s). Proposition-based testing tells us that our code realizes the relationships that should exist between different values from the domain. They actually tell us slightly different things and so arguably good tests do both, not one or the other. However if we have chosen the properties to test correctly then zero, one, or two examples are property-based testing (with as few examples as needed) is the future of testing

```
#include "rapidcheck.h"
#include "factorial.cpp"

int main() {
  using namespace Factorial;

  rc::check("value of operator delivers the right
    value", [](int i) {
      RC_ASSERT(*mpz_class_iterator{i} == i);
    });

  rc::check("pointer operator delivers the right
    value", [](int i) {
      RC_ASSERT(mpz_class_iterator{i}->get_si()
        == i);
    });

  rc::check("equality is value not identity.",
    [](int i) {
      RC_ASSERT(mpz_class_iterator{i}
        == mpz_class_iterator{i});
    });

  rc::check("inequality is value not identity.",
    [](int i, int j) {
      RC_PRE(j != 0);
      RC_ASSERT(mpz_class_iterator{i}
        != mpz_class_iterator{i + j});
    });
```

Listing 11

```
  rc::check("preincrement does in fact increment",
    [](int i) {
      RC_ASSERT(++mpz_class_iterator{i}
        == mpz_class_iterator{i + 1});
    });

  rc::check("postincrement does in fact
    increment", [](int i) {
      RC_ASSERT(mpz_class_iterator{i}++
        == mpz_class_iterator{i});
    });

  rc::check("value of preincrement returns correct
    value",  [](int i) {
      RC_ASSERT(*++mpz_class_iterator{i}
        == i + 1);
  });

  rc::check("value of postincrement returns
    correct value",  [](int i) {
      RC_ASSERT(*mpz_class_iterator{i}++ == i);
  });
}
```

Listing 11 (cont'd)

likely to be sufficient to 'prove' the code not incorrect. Hypothesis, for example, provides an `@example` decorator for adding those few examples. For other frameworks in other languages we can just add one or two example-based tests to the property-based tests.

But, some will say, don't (example-based) tests provide examples of use? Well yes, sort of. I suggest that these examples of use should be in the documentation, that users should not have to descend to reading the tests. So for me property-based testing (with as few examples as needed) is the future of testing. Examples and exemplars should be in the documentation. You do write documentation, don't you…

An apology

Having just ranted about documentation, you may think I am being hypocritical since the code presented here has no comments. A priori, code without comments, at least documentation comments[31], is a Bad Thing™ – all code should be properly documentation commented. All the code in the GitHub repository that holds the originals from which the code presented here were extracted is. So if you want to see the properly commented versions, feel free to visit https://github.com/russel/Factorial. If you find any improperly commented code, please feel free to nudge me about it and I will fix it post haste[32].

Acknowledgements

Thanks to Fran Buontempo for being the editor of this august[33] journal, especially this August august journal[34], and letting me submit a wee bit late.

Thanks to Jonathan Wakely for not laughing too much when I showed him the original C++ code, and for making suggestions that made the code far more sensible.

Thanks to the unnamed reviewers who pointed out some infelicities of presentation as well as syntax. Almost all the syntactic changes have been made – I disagreed with a few. Hopefully the changes made to the content has fully addressed the presentation issues that were raised.

Thanks to all those people working on programming languages and test frameworks, and especially for those working on property-based testing features, without whom this article would have been a very great deal shorter. ■

31. Debating the usefulness or otherwise of non-documentation comments is left as an exercise for the readership.
32. And request Doctor Who or someone to perform appropriate time travel with the corrections so that the situation has never been the case.
33. And, indeed, August.
34. "This joke is getting silly, stop this joke immediately." The Colonel.

Thanks to all those people working on programming languages and test frameworks, and especially for those working on property-based testing features

```
$ ./test_factorial
Using configuration: seed=10731500115167123548

- iterative applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- iterative applied to negative integer raises an
exception.
OK, passed 100 tests

- reductive applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- reductive applied to negative integer raises an
exception.
OK, passed 100 tests

- naïve recursive applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- naïve recursive applied to negative integer
raises an exception.
OK, passed 100 tests

- tail recursive applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- tail recursive applied to negative integer
raises an exception.
OK, passed 100 tests

$ ./test_mpz_class_iterator
Using configuration: seed=9168594634932513587

- value of operator delivers the right value
OK, passed 100 tests

- pointer operator delivers the right value
OK, passed 100 tests

- equality is value not identity.
OK, passed 100 tests

- inequality is value not identity.
OK, passed 100 tests

- preincrement does in fact increment
OK, passed 100 tests
```

Figure 4

References

[Catch] https://github.com/philsquared/Catch

[Ceylon] http://ceylon-lang.org/

[D] http://dlang.org/

[FitNesse] http://www.fitnesse.org/

[Frege] http://www.frege-lang.org or https://github.com/Frege/frege

[GNU] https://gmplib.org/, https://gmplib.org/manual/C_002b_002b-Interface-General.html

[Go] https://golang.org/

[Groovy] http://www.groovy-lang.org/

[Hackage] https://hackage.haskell.org/package/QuickCheck

[Haskell] https://www.haskell.org/

[HUnit] https://github.com/hspec/HUnit

[Hypothesis] http://hypothesis.works/, https://hypothesis.readthedocs.io/en/latest/, https://github.com/HypothesisWorks/hypothesis-python

[Kotlin] http://kotlinlang.org/

[MyPy] http://www.mypy-lang.org/

[Pytest] http://pytest.org/latest/

[Python_1] https://www.python.org/

[Python_2] https://www.python.org/dev/peps/pep-3107/, https://www.python.org/dev/peps/pep-0484/

[QuickCheck] https://en.wikipedia.org/wiki/QuickCheck

[QuickCheck_2] https://github.com/grogers0/CppQuickCheck

[QuickCheck_3] http://software.legiasoft.com/quickcheck/

[RapidCheck] https://github.com/emil-e/rapidcheck

[Robot] http://robotframework.org/

[Rust] https://www.rust-lang.org/

[Scala] http://www.scala-lang.org/

[Winder] The full Haskell example can be found at https://github.com/russel/Factorial/tree/master/Haskell.

[Winder16] *Overload*, 24(131):26–32, February 2016. There are PDF (http://accu.org/var/uploads/journals/Overload131.pdf#page=27) or HTML (http://accu.org/index.php/journals/2203) versions available.

Afterwood

Barriers can cause bottlenecks. Chris Oldwood considers varying approaches to gatekeeping.

Every morning, I struggle to get ready on time. No matter how hard I try I always seem to be leaving the house at 07:22 (+/- 30 seconds) and cycling like mad to the train station. As a consequence, I then have about a minute and a half to lock my bike up and run around to the front of the station, pass through the ticket barrier, and then skip to the last carriage where I will find a seat waiting for me.

Possibly the least predictable part of this entire journey is passing through the ticket barrier inside the station. What makes this all the more tense is that I can see my train on the platform just the other side and the clock ticking overhead on the departure board, in my head I hear the clock tick hugely amplified as if I'm Jack Bauer in an episode of 24. I've put my ticket in, it's been scanned and I know the back-end must have authorised me all in the blink of an eye but the barrier is mechanical and therefore subject to the laws of physics. I lift my arms to make myself as small as possible and eventually it opens wide enough for me to squeeze through and I'm out onto the platform before another barrier, this time the train doors, shuts me out.

Let's wind the clock back a few years to when I was fortunate enough to be able to travel to Japan to spend the week with a couple of friends at the Tokyo Game Show. Pretty much all my holidays up to that point had been within the European continent and so I was expecting to feel like the proverbial 'duck out of water' as I struggled in a country that used a completely different language for communicating, let alone the difference in culture and customs. Of course, they drive on the same side of the road so that was one less thing to worry about, not that we would be driving.

Naturally, to get around Tokyo we relied heavily on walking and public transport, most notably the rail system. There were many notable differences here too, such as the different little tune they play at each station, at which point a whole bunch of seemingly fast-asleep passengers lurch for the door before the train departs. The other interesting difference I noticed was the barriers they used at the entrance to the platforms – they were open by default. Of course initially we applied our usual British Rail mentality and just tried walking straight through on the assumption that they clearly must be broken or the ticket inspector is outside having a smoke or gone for an extended toilet break. As I entered the barrier a small pair of doors closed swiftly in front of us! At first I was confused, when I stepped back the doors opened, but as I stepped forward they closed again. How strange… Taking a moment to watch the natives it soon became apparent where we needed to touch the machine with our smart card so that the doors would remain open and we could pass through without any further disruption.

Aside from the psychological differences, which I'll come to in a moment, these two approaches have a very clear operational difference too. In the British case the barrier has to open and close for every single passenger, whereas the Japanese approach only requires the barrier to close when the passenger has forgotten to validate their ticket. This former's need to continuously operate the barrier means that it likely requires more energy to run, has a far shorter maintenance cycle and potentially a higher mean time-to-fail. The Japanese approach is more optimal simply by doing less work.

But what raises my ire the most about the British barrier is the implicit assumption that I am a fare dodger until I have proven myself innocent by validating my ticket. Far from welcoming me to their facilities my initial experience is one of confrontation as I am challenged to make myself worthy. In stark contrast the Japanese barrier welcomes me (literally) with open arms and invites me to proceed, only barring my entry if I should accidentally make a mistake. You can almost hear the apology from the gates as they shut, sorry that they've had to temporarily disrupt your journey. The fact that they're only knee high and therefore present no real obstacle means they're really just a speed-bump rather than a barrier.

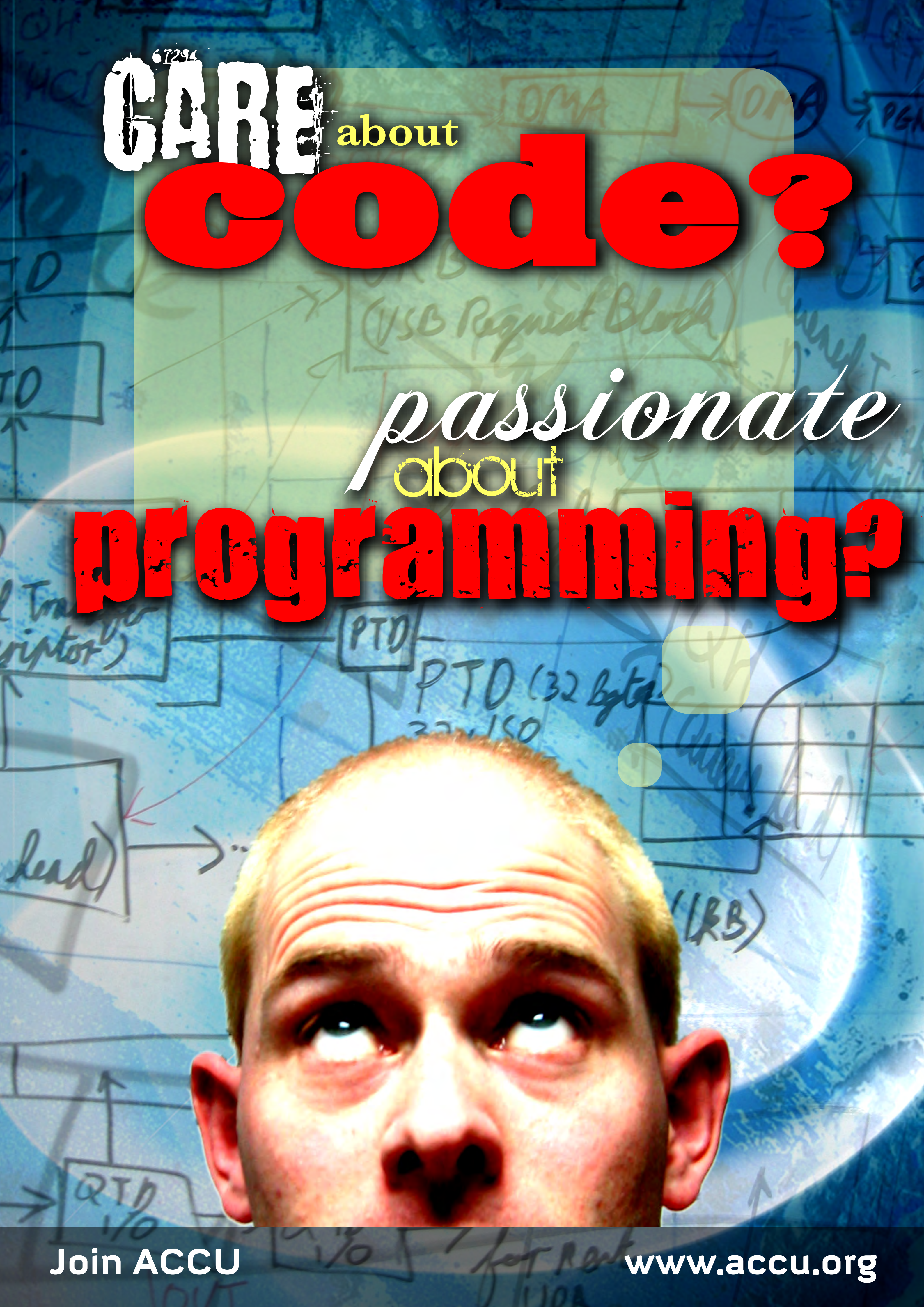
If only these kinds of barriers were limited to train stations.

Sadly we bump into the more metaphorical kind every day in the office. Instead of the culture making it easy to fall into the pit of success we find ourselves stumbling at every hurdle laid out in our path. The British-style barrier is the norm in most established organisations – you start from a position of being disallowed whatever it is you are after until you have collated enough 'evidence' to justify your right to continue with your intended course of action. In essence the tactic is one of pessimism – make it hard to do anything and the chance of mistakes happening will be reduced.

So what's the alternative? The Japanese-style barrier starts from a position of trust – it assumes that you are probably trying to do the right thing. This is one of optimism. But, crucially, it is not a naïve stance; an act of verification still has to occur. For the purposes of this analogy it would have been better if the Tokyo barriers had actually scanned my card after passing through the barrier, but I'll have to settle for a leakier metaphor.

The saying 'trust, but verify' has its roots in the Cold War where two vast nations were trying to avoid all-out nuclear war. Surely an organisation can start from a position of trusting its employees given that their mistakes will be far less costly? ■

Chris Oldwood Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood


CARE about code?
passionate about programming?
Join ACCU
www.accu.org

Software
Elite Reseller

ACCELERATE

MORE POWERFUL DATA ANALYSIS

Take your results to the next level with screaming-fast code.

Intel® Parallel Studio XE

www.qbssoftware.com/parallelstudio

020 8733 7101 | sales@qbssoftware.com

qbs SOFTWARE