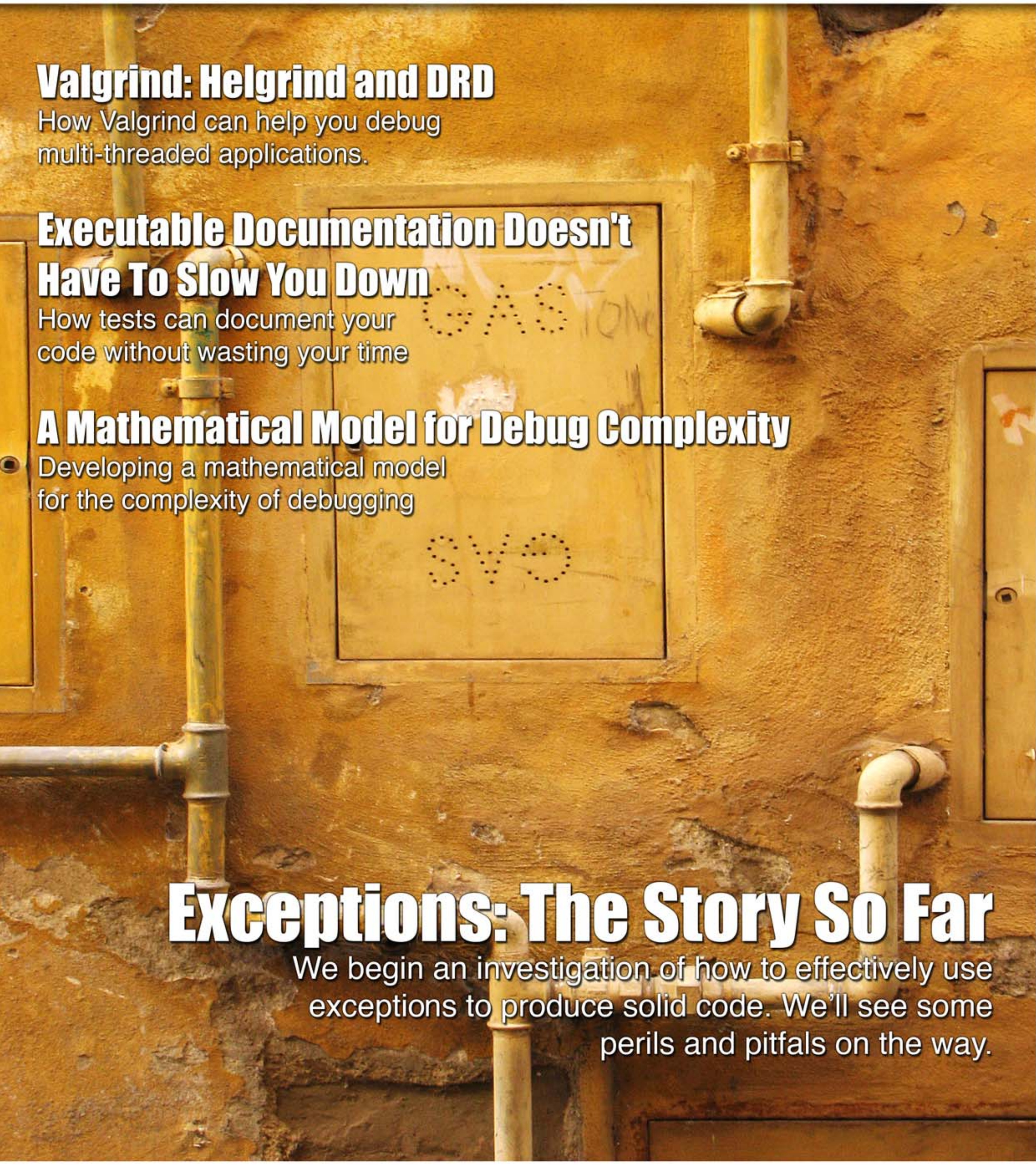
overload 114

APRIL 2013 £3



Valgrind: Helgrind and DRD
How Valgrind can help you debug multi-threaded applications.

Executable Documentation Doesn't Have To Slow You Down
How tests can document your code without wasting your time

A Mathematical Model for Debug Complexity
Developing a mathematical model for the complexity of debugging

Exceptions: The Story So Far
We begin an investigation of how to effectively use exceptions to produce solid code. We'll see some perils and pitfals on the way.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 114

April 2013

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@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 115 should be submitted by 1st May 2013 and for Overload 116 by 1st July 2013.

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

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.

Knitting Needles and Keyboards

Traditionally, both journals and developers have editors. Frances Buontempo considers the role of an editor, in another attempt to avoid writing an editorial.

A recent discussion started on accu general about editors, specifically asking for helpful top-tips for gvim or emacs, rather than starting yet another editors' war. This halted all noble thoughts of finally writing an *Overload* editorial, since I couldn't remember the first editor I used. The internet came to my rescue, and I whiled away minutes speed reading the BBC user manual. The keyboard was an integral part of the Beeb. It had a break key: "This key stops the computer no matter what it is doing. The computer forgets almost everything that it has been set to do" [BBCUserGuide]. Commands typed at a prompt were executed immediately whereas a series of numbered instructions formed a program. The user guide is littered with instructions on how to type, such as "If you want to get the + or * then you will have to press the SHIFT key as well as the key you want. It's rather like a typewriter: while holding the SHIFT key down, press the + sign once." [op cit]

Being able to type well seemed to be important back then, if slightly esoteric. After all, computers do often come with a keyboard, though not always. Can you imagine programming on your mobile phone using predictive text? Many years ago, people's first encounter with programming might have been via a programmable calculator, which posed similar editing problems. I'm sure most of us have seen rants from Jeff Atwood and Steve Yegge on typing, or rather many programmers' inability to type [Atwood]. Though typing is very important, I think my first interaction with a computing device was sliding round beads on an abacus, later followed by 'needle cards' [NeedleCards]. Allow me to explain – I dimly recall punching holes in a neat line along the top of cue cards, writing something on the cards, quite possibly names of polygons and polyhedra (ooh – I have just discovered polytopes, but I digress) and using each hole to indicate whether a property held true or held false for the item on the card. If the property were true, you cut the hole out to form a notch, and if false, you left the hole (or possibly vice versa). To discover which shapes involved, say, triangles you stuck a knitting needle in the pile of cards through the hole representing the property 'Triangle?', and gave the stack of cards a good shake. The relevant ones then fell in front of you (or stayed on the needle depending – it is a tad hazy). My introduction to knitting was therefore derailed and I still can't knit.

Historically inputs to a variety of machines involved holes in card or paper. Aside from the pianola, or autopiano, using perforated paper to mechanically move piano keys, it is often claimed that computers trace back to the Jacquard loom. According to one Wikipedia page, "The Jacquard loom was the first machine to use punched cards to control a sequence of operations" [Jacquard]. Eventually, computers used punched cards, for example Hollerith cards, prepared using key punch machines. On a trip to Bletchley Park a couple of years ago, I was surprised to learn that the paper tape input for Colossus (more holes in paper) was read by light, rather than mechanically, and could therefore achieve a comparatively high reading speed. Generating input in this forward-only format must have been quite time consuming. Without a backspace key, if you make a mistake, "You have to throw the card out and start the line all over again." [R-inferno] Aside from getting the hole in the punched cards correct and the cards themselves arranged in the correct order, they need carrying around: "One full box was pretty heavy; more than one became a load" [Fisk]. Not all machine inputs were holes in card or paper. Consider the 'Electronic Numerical Integrator and Computer', ENIAC. Though it would accept punched cards, its programming interface originally "required physical stamina, mental creativity and patience. The machine was enormous, with an estimated 18,000 vacuum tubes and 40 black 8-foot cables. The programmers used 3,000 switches and dozens of cables and digit trays to physically route the data and program pulses." [ENIAC]

Eventually random-access editing became possible, with punched cards being replaced by keyboards. On a typewriter, it was possible to wind the paper back and use Tipp-Ex to edit previous mistakes, though it would help if you wanted to replace the mistake with the same number of characters, or fewer. Tipp-Ex cannot change the topology of the paper you are typing on. "In 1962 IBM announced the Selectric typewriter" [Reilly]. This allowed proportional font and had a spherical 'golf-ball' typing head that could be swapped to change font. The ball would rotate and pivot in order to produce the correct character. These electronic typewriters eventually morphed into a machine with memory thereby allowing word-processing. They were also used as computer terminals after the addition of solenoids and switches, though other purpose built devices were available [Selectric]. A computer interface that allows editing changes the game. Emacs came on the scene in 1976, while Vim released in 1991, was based on Vi, which Wikipedia claims was also written in 1976 [vi]. Many editors allow syntax highlighting now, adding an extra dimension to moving backwards and forwards in the text. This requires the ability to parse the language being input which is leaps and bounds beyond making holes in card. Parsing the language on input also allows intellisense or auto-completion, though I tend to find combo-boxes popping up in front of what I am trying to type very off-putting. After a contract using C# with Resharper for a year, my typing speed has taken a nose-dive, and my spelling is now even worse. I tend to look at the screen rather than the keyboard since I can type, but when the screen is littered with pop-ups I stop watching the screen and look out of the window instead. If only that particular IDE came with a pop-up blocker.

In order to use these text editors, a keyboard obviously is required. QWERTY keyboards have become the de-facto standard. Why? "It makes no sense. It is awkward, inefficient and confusing. We've been saying that for 124 years. But there it remains. Those keys made their first appearance on a rickety, clumsy device marketed as the 'Type-Writer' in 1872." [QWERTY] This article debunks the myth that its

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 for over 12 years professionally, 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.

inventor Sholes deliberately arranged his keyboard to slow down fast typists who would otherwise jam up the keys. The arrangement seems rather to have been designed to avoid key jams directly rather than slowing down typists and hoping this avoided key jams. Surprisingly the keyboard in the patent has neither a zero nor a one. I am reliably told that the letters 'l' and 'O' would be used instead. Not only have typewriters left us with the key layout, but also archaic terms like carriage return, line feed and shift. There are other keyboard layouts, such as Dvorak, which you can switch to in order to confuse your colleagues or family silly. I am personally tempted to get a Das keyboard. From their marketing spiel, "Efficient typists don't look at their keyboards. So why do others insist on labelling the keys? Turns out you'll type faster than you ever dreamed on one of these blank babies. And that's not to mention its powers of intimidation." [DASKeyboard]

Research into computer interfaces has a surprisingly long history. "The principles for applying human factors to machine interfaces became the topic of intense applied research during the 1940s" [HCI] Human-computer interaction is still an active area of research today, covering many inter-disciplinary areas from psychology to engineering. Input methods have moved away from knitting needles and even keyboards. Starting with text editing, in the 1950s, through to the mouse (1968) and gesture recognition (1963), notice all in the 60s, to the first WYSIWYG editor-formatter Xerox PARC's Bravo (1974) [Meyers], new ways of telling a computer what to do are constantly being created. Perhaps we are moving closer to the realisation of a futuristic cyberpunk dream or virtual reality. "By coupling a motion of the user's head to changes in the images presented on a head-mounted display, the illusion of being surrounded by a world of computer-generated images or a virtual environment is created. Hand-mounted sensors allow the user to interact with these images as if they were real objects located in space surrounding him or her" [HCI] Mind you, Meyers tells us virtual reality was first worked on in 1965–1968, using head-mounted displays and 'data gloves' [op cit], so perhaps it is more a dystopian cyberpunk dream that is constantly re-iterated. Let's keep our eyes on the latest virtual reality, Google glass [Glass].

A variety of programming languages have sprung up now we can type words into machines, or click on words in intellisense. Some programming languages do seem to be easier to edit than others. APL springs to mind. APL had its own character set, and required a special typeball. "Some APL symbols, even with the APL characters on the typeball, still had to be typed in by over-striking two existing typeball characters. An example would be the 'grade up' character, which had to be made from a 'delta' (shift-H) and a 'Sheffer stroke' (shift-M). This was necessary because the APL character set was larger than the 88 characters allowed on the Selectric typeball." [APL] It seems likely that making input to a computer easier than punching cards or swapping cables and flipping switches gave rise to high-level programming languages. I wonder if any new ways of interacting with computers will further change the languages we use. Perhaps the growth of test-driven development, TDD, will one day be taken to its logical conclusion: humans will write all the tests then machines can generate the code they need to pass the tests. Genetic programming was introduced to perform exactly this task, possibly before TDD was dreamt of [GP]. If this became the norm, another form of human computer interaction we have not considered would become obsolete: compilers. They exist purely to turn high-level languages, understood by humans, into machine language. If programs are eventually all written by machines, there will be no need for a human to ever read another line of code again. Electronic wizards can automatically generate all the code we need; we are only required to get the tests correct.

We have considered a variety of ways of editing inputs for computers, so should step back and consider editors in the more usual sense of the word. An editor is "in charge of and determines the final content of a text", according to Google. It is striking that this is remarkably like the definition of a computer editor. Now, an editor needs something to edit, which ultimately needs the invention of printing. Originally, scribes would hand-copy texts self-editing as they went, but eventually books and pamphlets could be mass-produced. This allowed arguments about spelling and an insistence on accuracy, with such characters as Frederic Madden coming to the fore [Matthews]. Perhaps with the prevalence of blogs and other means of self-publishing, things have come full circle, leaving people tending to self-edit.

We have seen the historical forms and roles of editors, and glimpsed a fifty-year old dream of the future. I feel more prepared to attempt a proper editorial next time, but suspect I might need to learn to type properly first. Hoorah for spell checkers and the *Overload* review team.

References

[APL] http://en.wikipedia.org/wiki/APL_(programming_language)

[Atwood] http://www.codinghorror.com/blog/2008/11/we-are-typists-first-programmers-second.html

[BBCUserGuide] http://bbc.nvg.org/doc/BBCUserGuide-1.00.pdf

[DASKeyboard] http://www.daskeyboard.com/#ultimate

[ENIAC] http://abcnews.go.com/Technology/story?id=3951187&page=2

[Fisk] http://www.columbia.edu/cu/computinghistory/fisk.pdf

[Glass] http://www.google.com/glass/start/

[GP] http://www.genetic-programming.com/

[HCI] 'Human-Computer Interaction: Introduction and Overview' K Butler, R Jacob, B John, *ACM Transactions on Computer-Human Interaction,* 1999

[Jacquard] http://en.wikipedia.org/wiki/Jacquard_loom

[Matthews] *The Invention of Middle English: An Anthology of Primary Sources* David Matthews

[Meyers] *A Brief History of Human-Computer Interaction Technology*, Brad A. Myers

[NeedleCards] http://en.wikipedia.org/wiki/Needle_card

[QWERTY] http://home.earthlink.net/~dcrehr/whyqwert.html

[R-inferno] Partick Burns 2011 (http://www.burns-stat.com/pages/Tutor/R_inferno.pdf)

[Reilly] *Milestones in Computer Science and Information Technology*, Edwin D. Reilly 2003

[Selectric] http://en.wikipedia.org/wiki/IBM_Selectric_typewriter

[vi] http://en.wikipedia.org/wiki/Vi

A Model for Debug Complexity

Debugging any program can be difficult. Sergey Ignatchenko and Dmytro Ivanchykhin develop a mathematical model for its complexity.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translator or the Overload editor. Please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented providing an exact translation. In addition, both the translators and Overload expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Debugging programs is well-known to be a complicated task, which takes lots of time. Unfortunately, there is surprisingly little research on debugging complexity. One interesting field of research is related to constraint programming ([CP07][CAEPIA09]), but currently these methods are related to constraint-based debugging, and seem to be of limited use for analysis of debugging complexity of commonly used program representations. This article is a humble attempt to propose at least some mathematical models for the complexity of debugging. It is not the only possible or the best possible model; on the contrary, we are aware of the very approximate nature of the proposed model (and of all the assumptions which we've made to build it), and hope that its introduction (and following inevitable criticism) will become a catalyst for developing much more precise models in the field of debugging complexity (where, we feel, such models are sorely needed).

Basic model for naïve debugging

Let's consider a simple linear program which is *N* lines of code in size. Let's see what happens if it doesn't produce the desired result. What will happen in the very simplistic case is that the developer will go through the code in a debugger, from the very beginning, line by line, and will see what is happening in the code. So, in this very simple case, and under all the assumptions we've made, we have that

$$T_{dbg} = N \times T_{line}$$

where T_{line} is the time necessary to check that after going through one line, everything is still fine.

So far so obvious (we'll analyze the possibility for using bisection a bit later).

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko and Dmytro Ivanchykhin using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

Dmytro Ivanchykhin has 10+ years of development experience, and has a strong mathematical background (in the past, he taught maths at NDSU in the United States). Dmytro can be contacted at d_ivanchykhin@yahoo.com

Now, let's try to analyze what T_{line} is about. Let's assume that our program has *M* variables. Then, after each line, to analyze if anything has gone wrong, we need to see if any of *M* variables has gone wrong. Now, let's assume that all our *M* variables are tightly coupled; moreover, let's assume that coupling is 'absolutely tight', i.e. that any variable affects the meaning of all other variables in a very strong way (so whenever a variable changes, the meaning of all other variables may be affected). It means that, strictly speaking, to fully check if everything is fine with the program, one would need to check not only that all *M* variables are correct, but also (because of 'absolutely tight' coupling) that all combinations of 2 variables (and there are *C*(*M*,2) such combinations) are correct, that all combinations of 3 variables (and there are *C*(*M*,3) such combinations) are correct, and so on. Therefore, we have that

$$T_{line} = \sum_{i=1}^{M} C(M,i) \times T_{singlecheck}$$

where

$$C(M,i) = \frac{M!}{i! \times (M-i)!}$$

and $T_{singlecheck}$ is the time necessary to perform a single validity check.

Therefore, if we have a very naïve developer who checks that everything is correct after each line of code, we'll have

$$T_{dbg} = N \times \sum_{i=1}^{M} C(M,i) \times T_{singlecheck}$$

which is obviously (see picture) equivalent to

$$T_{dbg} = N \times (2^M - 1) \times T_{singlecheck} \qquad *$$

Optimizations

Now consider what can be done (and what in the real-world is really done, compared to our hypothetical naïve developer) to reduce debugging time. Two things come to mind almost immediately. The first one is that if a certain line of code changes only one variable *X* (which is a very common case), then the developer doesn't need to check all 2^M combinations of variables, but needs to check only variable *X*, plus all 2-variable combinations which include *X* (there are *M*-1 of them), plus all 3-variable combinations which include *X* (there are *C*(*M*-1,2) of them), and so on. Therefore, assuming that every line of code changes only one variable (including potential aliasing), we'll have

$$T_{dbg} = N \times \left(\left(1 + \sum_{i=1}^{M-1} C(M-1,i) \right) \times T_{singlecheck} \right)$$

or

$$T_{dbg} = N \times 2^{M-1} \times T_{singlecheck}$$

Therefore, while this first optimization does provide significant benefits, the dependency on *M* is still exponential.

The second optimization to consider is using bisection. If our program is deterministic (and is therefore reproducible), then instead of going line-by-line, a developer can (and usually will) try one point around the middle of the program, check the whole state, and then will see if the bug has already happened, or if it hasn't happened yet. The process can then be repeated. In this case, we'll have

$$T_{dbg} = \log_2 N \times \left(2^M - 1\right) \times T_{singlecheck}$$

Note that we cannot easily combine our two optimizations because the first one is essentially incremental, which doesn't fit the second one. In practice, usually a 'hybrid' method is used (first the developer tries bisection, and then, when the span of a potential bug is small enough, goes incrementally), but we feel that it won't change the asymptotes of our findings.

Basic sanity checks

Now, when we have the model, we need to check how it corresponds to the real world. One can build a lot of internally non-contradictory theories, which become obviously invalid at the very first attempt to match them to the real world. We've already made enough assumptions to realize the need for sanity checks. We also realize that the list of our sanity checks is incomplete and that therefore it is perfectly possible that our model won't stand further scrutiny.

The first sanity check is against intuitive observations, that the longer the program, and the more variables it has, the more difficult it is to debug. Our formulae seem to pass this check. The next sanity check is against another intuitive observation that debugging complexity grows non-linearly with the program size; as usually both *N* and *M* grow with the program size, so our formulae seem to pass this check too.

Sanity check – effects of decoupling on debugging complexity

Now we want to consider a more complicated case. Since the 70s, one of the biggest improvements in tackling program (and debugging) complexity has been the introduction of OO with hidden data, well-defined interfaces, and reduced coupling. In other words, we all know that coupling is bad (and we feel that no complexity model can be reasonably accurate without taking this into account); let's see how our model deals with it. If we split *M* 'absolutely tightly' coupled variables into two bunches of uncoupled variables, each being *M*/2 in size, and each such group has only $M/2 \times K_{p2p}$ public member variables (K_{p2p} here stands for 'public-to-private ratio', and $0 <= K_{p2p} <= 1$), then our initial formula [1] (without optimizations) will become

$$T_{dbg} = N \times \left(T_{firstbunch} + T_{secondbunch} + T_{interbunch}\right)$$

where both $T_{firstbunch}$ and $T_{secondbunch}$ are equal to

$$\sum_{i=1}^{M/2} C\left(\frac{M}{2}, i\right) \times T_{singlecheck}$$

and

$$T_{interbunch} = \sum_{i=1}^{\frac{M}{2} \times K_{p2p} \times 2} C\left(\frac{M}{2} \times K_{p2p} \times 2, i\right) \times T_{singlecheck}$$

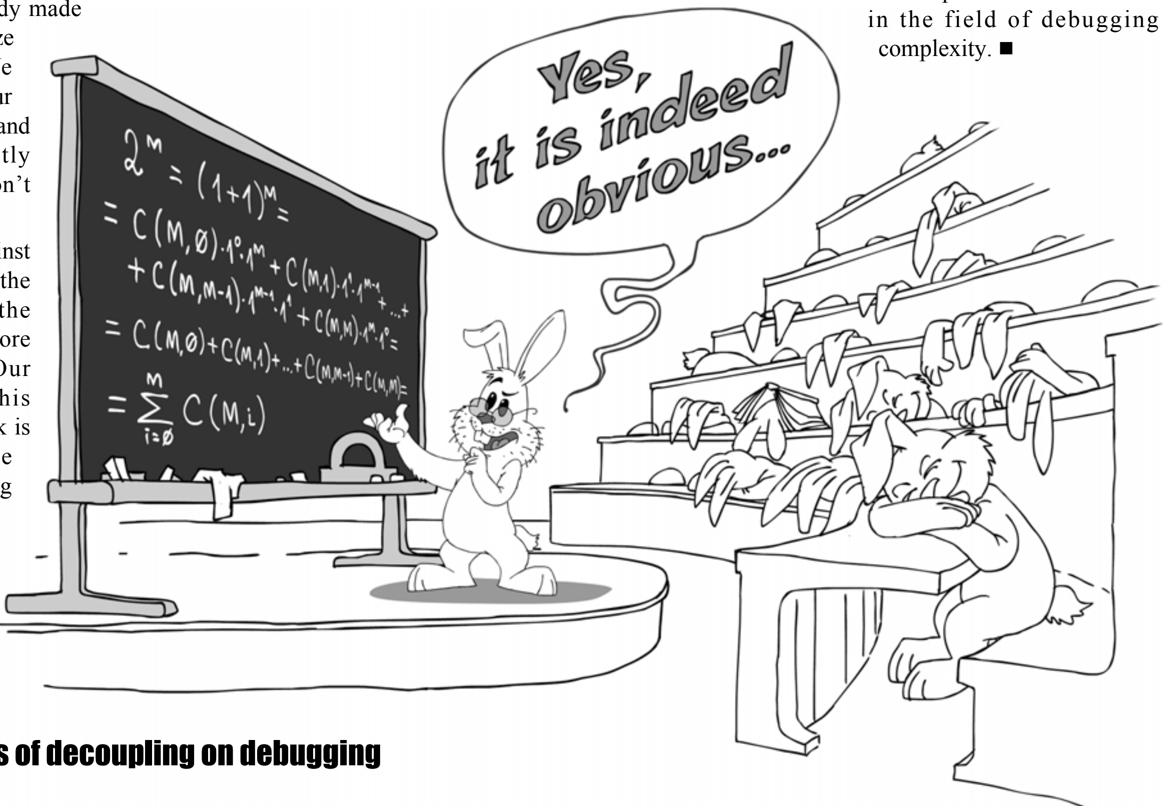
After some calculations, we'll get that

$$T_{dbg} = N \times \left(2 \times \left(2^{\frac{M}{2}} - 1\right) + \left(2^{M \times K_{p2p}} - 1\right)\right) \times T_{singlecheck} \qquad **$$

It means that if we have 10 variables, then splitting them into 2 bunches of 5 variables each, with only 2 public member variables exposed from each bunch (so K_{p2p}=0.4), will reduce debugging time from approx. $2^{10} \times T_{singlecheck}$ (according to *), to approx. $(2^6+2^4) \times T_{singlecheck}$ (according to **). So yes, indeed our model shows that coupling is a bad thing for debugging complexity, and we can consider this sanity check to be passed too.

Conclusion

In this article, we have built a mathematical model of debugging complexity. This model is based on many assumptions, and is far from being perfect. Still, we hope that it can either serve as a basis for building more accurate models, or that it at least will cause discussions, leading to the development of better models in the field of debugging complexity. ■


Yes, it is indeed obvious...
2^M = (1+1)^M =
= C(M,∅)·1^0·1^M + C(M,1)·1^1·1^(M-1) + ... + C(M,M-1)·1^(M-1)·1^1 + C(M,M)·1^M·1^0 =
= C(M,∅)+C(M,1)+...+C(M,M-1)+C(M,M) =
= Σ_(i=∅)^M C(M,i)

References

[CAEPIA09] On the complexity of program debugging using constraints for modeling the program's syntax and semantics, Franz Wotawa, Jörg Weber, Mihai Nica, Rafael Ceballos, Proceedings of the Current topics in artificial intelligence, and 13th conference on Spanish association for artificial intelligence.

[CP07] Exploring different constraint-based modelings for program verification, Hélène Collavizza, Michel Rueher , Proceedings of the 13th international conference on Principles and practice of constraint programming.

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

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

Valgrind Part 6 – Helgrind and DRD

Debugging multi-threaded code is hard. Paul Floyd uses Helgrind and DRD to find deadlocks and race conditions.

The marketing department has promised a multithreaded version of your application, which was designed ten years before threads were available in commercial OSes [No Bugs]. So you use Callgrind [callgrind] to identify your program's hotspots and set about retrofitting threads to the code. You know a bit about threads, and manage to get a decent speed up. Unfortunately, the results are intermittently incorrect and the program occasionally locks up. Because the problems seem non-deterministic, they are hard to debug. What to do? Valgrind to the rescue, again. Valgrind includes not one but two tools for diagnosing thread hazards, Helgrind and DRD. Why two? They perform substantially the same work, but Helgrind tends to produce output that is easier to interpret whilst DRD tends to have better performance. DRD means Data Race Detector and I'm not sure where the name Helgrind comes from, maybe it just fits in with the other -grind suffixes.

Both tools work with pthreads and Qt 4 threads (not yet Qt 5, but that should be in Valgrind 3.9 and I believe it is in the current development version of Valgrind under Subversion). In order to detect hazards they record memory access ordering and also thread locking. If you use some other threading library or write your own threading functions then you will need to write your own wrappers in order to be able to use these tools.

Since I've mentioned Qt, that leads me to a slight digression on C++ and dynamic libraries. In general, Valgrind tools intercept various C library calls like **`malloc`**. In the case of C++, that interception has to be done with the mangled symbols. Valgrind also needs to know the soname of the containing dynamic library (the soname is the library name registered within the library file). With Qt 5, `libQt4Core.so` became, unsurprisingly, `libQt5Core.so`.

I will assume that you have a working knowledge of programming with threads. See [threads references] if you need a refresher. In this article, I'll use simple examples of API misuse, race conditions and deadlock.

The basic mechanisms used for the tools are to record reads and writes at the machine code emulation level plus to intercept calls to the thread synchronization functions. The relative order of reads and writes on different threads can be analysed and if they are not within appropriate synchronization barriers, an error is detected. Locking is not inferred from the machine code, which is why interception is required.

Paul Floyd has been writing software, mostly in C++ and C, for over 20 years. He lives near Grenoble, on the edge of the French Alps, and works for Atrenta, Inc. developing tools to aid early design closure in electronic circuit conception. He can be contacted at pjfloyd@wanadoo.fr.

```
==2573== ---Thread-Announcement------------------------------------------
==2573==
==2573== Thread #1 is the program's root thread
==2573==
==2573== ----------------------------------------------------------------
==2573==
==2573== Thread #1 unlocked an invalid lock at 0x601040
==2573==    at 0x4C2EDBE: pthread_mutex_unlock (hg_intercepts.c:609)
==2573==    by 0x400659: main (noddy.cpp:6)
```

Figure 1

```
==2570== The object at address 0x601040 is not a mutex.
==2570==    at 0x4C3180E: pthread_mutex_unlock (drd_pthread_intercepts.c:703)
==2570==    by 0x400659: main (noddy.cpp:6)
```

Figure 2

I'll start with a noddy example of pthreads abuse (see Listing 1).

You can't get much easier than that, example wise. The command line is equally easy:

```
valgrind --tool=helgrind -v --log-file=hg.log
./noddy
```

The resulting log file starts with about 70 lines of information about Valgrind, shared libraries and symbols. I won't include it here. The important part is shown in Figure 1.

So we know which thread we're on, and we're informed of the misuse of **`pthread_mutex_unlock`**.

Let's run the same example with DRD (see Figure B):

```
valgrind --tool=drd -v --log-file=drd.log ./noddy
```

The output is a little bit terser (see Figure 2), but the same information is there.

Next up an example with a race condition. Basically this uses a usleep to pretend to do some work [with the nice property that Valgrind sleeps at the same speed as the application running at full speed]. However, there's a static variable in the third file that I accidentally 'forgot'. When I run the

```
// noddy.cpp
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main()
{
  pthread_mutex_unlock(&mutex);
}
```

Listing 1

```
// main.cpp
#include <pthread.h>
#include <iostream>

void* worker_thread(void* in);
int backdoor();

int main()
{
  pthread_t t1, t2;
  pthread_create(&t1, 0, worker_thread,
     reinterpret_cast<void*>(7));
  pthread_create(&t2, 0, worker_thread,
    reinterpret_cast<void*>(13));

  pthread_join(t1, 0);
  pthread_join(t2, 0);

  std::cout << "Forgotten static is "
            << backdoor() << " ought to be "
            << 1000000/7 + 1000000/13 + 2 << "\n";
}
```

Listing 2

```
// worker.cpp
void other_work(int n, int i);

void* worker_thread(void* in)
{
  int number = reinterpret_cast<long>(in);
  for (int i = 0; i < 1000000; ++i)
  {
    other_work(number, i);
  }
  return 0;
}
```

Listing 3

```
// other.cpp
#include <unistd.h>

static int forget_me_not;

void other_work(int n, int i)
{
  if (i % n == 0)
  {
    ++forget_me_not;
  }
  else
  {
    ::usleep(1);
  }
}

int backdoor()
{
  return forget_me_not;
}
```

Listing 4

application, I usually get the correct result, but about a quarter of the time the result is 1 or 2 too low (see `main.cpp` in Listing 2, `worker.cpp` in Listing 3 and `other.cpp` in Listing 4).

This is a simple example of omission to protect some shared resource from threaded write access. In real world problems, the problem could also be inappropriate protection, such as a write within a section that is only protected by a read lock.

Now let's run Helgrind.

```
valgrind --tool=helgrind -v
  --log-file=helgrind.log ./test
```

Then it tells us about the number of threads created. Two, as expected (see Figure 3).

So far so good. Then it detects a possible data race (see Figure 4).

Now, when it says 'possible', don't fool yourself into thinking 'oh, that will never happen'. If the threads run for long enough simultaneously, the chances are that it will. However, there is some chance that you might be lucky. Since Valgrind uses an abstract model for the emulation, it isn't always identical to the underlying CPU. In particular, Intel/AMD perform less reordering than platforms like POWER which can perform stores out of order. So whilst there may be no error on Intel, the same code could could fail on POWER.

```
==2678== ---Thread-Announcement------------------------------------------
==2678==
==2678== Thread #3 was created
==2678==    at 0x594928E: clone (in /lib64/libc-2.15.so)
==2678==    by 0x4E3BFF4: do_clone.constprop.4 (in /lib64/libpthread-2.15.so)
==2678==    by 0x4E3D468: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-
2.15.so)
==2678==    by 0x4C2E27F: pthread_create_WRK (hg_intercepts.c:255)
==2678==    by 0x4C2E423: pthread_create@* (hg_intercepts.c:286)
==2678==    by 0x400A6A: main (main.cpp:11)
==2678==
==2678== ---Thread-Announcement------------------------------------------
==2678==
==2678== Thread #2 was created
==2678==    at 0x594928E: clone (in /lib64/libc-2.15.so)
==2678==    by 0x4E3BFF4: do_clone.constprop.4 (in /lib64/libpthread-2.15.so)
==2678==    by 0x4E3D468: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-
2.15.so)
==2678==    by 0x4C2E27F: pthread_create_WRK (hg_intercepts.c:255)
==2678==    by 0x4C2E423: pthread_create@* (hg_intercepts.c:286)
==2678==    by 0x400A4F: main (main.cpp:10)
==2678==
```

Figure 3

In this case, it is a read/write hazard. There is also a write/write hazard (I'll just show the top of the stack for brevity in Figure 5).

The log ends with a summary of the test.

Now the same test, but using DRD.

```
valgrind --tool=drd -v
--log-file=drd.log
./test
```

This generates similar header information and again detects the two read/write and write/write race conditions (only the read/write one shown in Figure 6).

There are two differences with Helgrind. Firstly, it tells us where the conflicting memory was allocated (in this case, BSS, a file static variable). The first location is given precisely, but the second one

```
==2678== ----------------------------------------------------------------
==2678==
==2678== Possible data race during read of size 4 at 0x602198 by thread #3
==2678== Locks held: none
==2678==    at 0x400B5B: other_work(int, int) (other.cpp:9)
==2678==    by 0x400BAE: worker_thread(void*) (worker.cpp:8)
==2678==    by 0x4C2E40C: mythread_wrapper (hg_intercepts.c:219)
==2678==    by 0x4E3CE0D: start_thread (in /lib64/libpthread-2.15.so)
==2678==    by 0x59492CC: clone (in /lib64/libc-2.15.so)
==2678==
==2678== This conflicts with a previous write of size 4 by thread #2
==2678== Locks held: none
==2678==    at 0x400B64: other_work(int, int) (other.cpp:9)
==2678==    by 0x400BAE: worker_thread(void*) (worker.cpp:8)
==2678==    by 0x4C2E40C: mythread_wrapper (hg_intercepts.c:219)
==2678==    by 0x4E3CE0D: start_thread (in /lib64/libpthread-2.15.so)
==2678==    by 0x59492CC: clone (in /lib64/libc-2.15.so)
```

Figure 4

```
==2678== Possible data race during write of size 4 at 0x602198 by thread #3
==2678== Locks held: none
==2678==    at 0x400B64: other_work(int, int) (other.cpp:9)
==2678==
==2678== This conflicts with a previous write of size 4 by thread #2
==2678== Locks held: none
==2678==    at 0x400B64: other_work(int, int) (other.cpp:9)
```

Figure 5

only gives a starting and ending range. In this case, between the start of the thread and line 13 of `other.cpp`.

For my final example which illustrates deadlock, I'll use Qt 4 this time. Deadlock occurs when you have cycles in the graph that represents your lock/unlock states, or conversely, if your lock/unlock states form a tree, you won't have any deadlocks. This is described in [correct parallel algorithms]. If you have two or more locks, then always lock/unlock them in the same order. You won't have any deadlocks, though you may get thread starvation, a lot of contention and poor performance.

There are several ways of creating threads with Qt and I'll use inheritance for simplicity. I'm using explicit locks and unlocks in order to make the deadlocking more obvious. In practice, I'd recommend using QmutexLocker as it will make your code shorter and safer (see Listing 5).

```
==2669== Thread 3:
==2669== Conflicting load by thread 3 at 0x00602198 size 4
==2669==    at 0x400B5B: other_work(int, int) (other.cpp:9)
==2669==    by 0x400BAE: worker_thread(void*) (worker.cpp:8)
==2669==    by 0x4C2E2A4: vgDrd_thread_wrapper (drd_pthread_intercepts.c:355)
==2669==    by 0x4E47E0D: start_thread (in /lib64/libpthread-2.15.so)
==2669==    by 0x59542CC: clone (in /lib64/libc-2.15.so)
==2669== Allocation context: BSS section of /home/paulf/vg_thread/test
==2669== Other segment start (thread 2)
==2669==    at 0x4C3180E: pthread_mutex_unlock (drd_pthread_intercepts.c:703)
==2669==    by 0x4C2E29E: vgDrd_thread_wrapper (drd_pthread_intercepts.c:236)
==2669==    by 0x4E47E0D: start_thread (in /lib64/libpthread-2.15.so)
==2669==    by 0x59542CC: clone (in /lib64/libc-2.15.so)
==2669== Other segment end (thread 2)
==2669==    at 0x5925CAD: ??? (in /lib64/libc-2.15.so)
==2669==    by 0x594E6B3: usleep (in /lib64/libc-2.15.so)
==2669==    by 0x400B75: other_work(int, int) (other.cpp:13)
==2669==    by 0x400BAE: worker_thread(void*) (worker.cpp:8)
==2669==    by 0x4C2E2A4: vgDrd_thread_wrapper (drd_pthread_intercepts.c:355)
==2669==    by 0x4E47E0D: start_thread (in /lib64/libpthread-2.15.so)
==2669==    by 0x59542CC: clone (in /lib64/libc-2.15.so)
```

Figure 6

Let's run that with Helgrind. I built it using Qt Creator, which made a funky directory for me

```
  valgrind
--tool=helgrind -v
--log-file=qthg.log
 ./QtExample-build-
desktop-
Qt_4_8_1_in_PATH__
System__Debug/QtExample
```

Figure 7 shows the output from Helgrind.

There is a first error that is a false positive. See https://bugs.kde.org/show_bug.cgi?id=307082 which says that pthread_cont_init is not handled, while the second error is the deadlock.

DRD does not detect any error with this example. Unfortunately I've run out of time to get an explanation for this article.

So, in summary, multithreaded programming is difficult to debug, but Valgrind can make it easier.

In this series I've covered the main tools that make up Valgrind. However, there's still more. Valgrind contains a few experimental tools like checking stack and global variables and a data access profiler. Even after that there is still more. Since Valgrind is open source, there are several third party extensions available, most notably including support for Wine and another variation on thread checking, ThreadSanitizer [thread sanitizer]. I have no experience with them – yet. ■

References

[callgrind] *Overload* 111, 'Valgrind Part 4: Cachegrind and Callgrind', Paul Floyd

[correct parallel algorithms] *Overload* 111, 'A DSEL for Addressing the Problems Posed by Parallel Architectures', Jason McGuiness and Colin Egan

[No Bugs] They didn't listen to 'No Bugs' bunny. *Overload* 97, 'Single-Threading: Back to the Future?' and *Overload* 98, 'Single-Threading: Back to the Future? Part 2', Sergey Ignatchenko

[thread sanitizer] http://code.google.com/p/data-race-test/wiki/ThreadSanitizer

[threads references]

1. *Multithreaded Programming With Pthreads*, (ISBN 10: 0136807291 / ISBN 13: 9780136807292), Lewis, Bil, Berg, Daniel J., Sun Microsystems Press
2. *Programming with POSIX Threads*, (ISBN-10: 0201633922 / ISBN-13: 978-0201633924), David R. Butenhof , Addison-Wesley Professional

```
// main.cpp
#include <QThread>
#include <QMutex>
#include <iostream>

QMutex m1;
static int resource1;

QMutex m2;
static int resource2;

class MyThread1 : public QThread
{
public:
  void run();
};

static int LOOPS = 10000;

void MyThread1::run()
{
  for (int i = 1; i < LOOPS; ++i)
  {
    m1.lock();
    usleep(10);
    m2.lock();
    if (resource1 % 3)
      ++resource2;
    usleep(10);
    m2.unlock();
    m1.unlock();
  }
}
```

Listing 5

```
class MyThread2 : public QThread
{
public:
  void run();
};

void MyThread2::run()
{
  for (int i = 1; i < LOOPS; ++i)
  {
    m2.lock();
    usleep(10);
    m1.lock();
    if (resource2 % 2 == 0)
      ++resource1;
    usleep(10);
    m1.unlock();
    m2.unlock();
  }
}

int main()
{
  MyThread1 t1;
  MyThread2 t2;

  t1.run();
  t2.run();

  t1.wait();
  t2.wait();

}
```

Listing 5 (cont'd)

```
==13045== Thread #1: lock order "0x602160 before 0x602168" violated
==13045==
==13045== Observed (incorrect) order is: acquisition of lock at 0x602168
==13045==    at 0x4C2CA24: QMutex_lock_WRK (hg_intercepts.c:2058)
==13045==    by 0x4C30715: QMutex::lock() (hg_intercepts.c:2067)
==13045==    by 0x401228: MyThread2::run() (main.cpp:44)
==13045==    by 0x4012C5: main (main.cpp:61)
==13045==
==13045==  followed by a later acquisition of lock at 0x602160
==13045==    at 0x4C2CA24: QMutex_lock_WRK (hg_intercepts.c:2058)
==13045==    by 0x4C30715: QMutex::lock() (hg_intercepts.c:2067)
==13045==    by 0x40123C: MyThread2::run() (main.cpp:46)
==13045==    by 0x4012C5: main (main.cpp:61)
==13045==
==13045== Required order was established by acquisition of lock at 0x602160
==13045==    at 0x4C2CA24: QMutex_lock_WRK (hg_intercepts.c:2058)
==13045==    by 0x4C30715: QMutex::lock() (hg_intercepts.c:2067)
==13045==    by 0x40118A: MyThread1::run() (main.cpp:23)
==13045==    by 0x4012B9: main (main.cpp:60)
==13045==
==13045==  followed by a later acquisition of lock at 0x602168
==13045==    at 0x4C2CA24: QMutex_lock_WRK (hg_intercepts.c:2058)
==13045==    by 0x4C30715: QMutex::lock() (hg_intercepts.c:2067)
==13045==    by 0x40119E: MyThread1::run() (main.cpp:25)
==13045==    by 0x4012B9: main (main.cpp:60)
```

Figure 7

Quality Matters #7 Exceptions: the story so far

Exception handling is difficult to get right. Matthew Wilson recaps the story so far.

This instalment is the long-awaited next instalment of the Quality Matters column in general and the exceptions series in particular. In it, I recap the previous thinking in regards to exceptions, update some information presented in the previous episode, and present some material that leads into the rest of the series.

Introduction

As the last three years have been extremely involved for me – I've been acting as an expert witness in a big software copyright case here in Australia – I've been pretty unreliable at meeting publishing deadlines: the Quality Matters column has suffered just as a much as my other article and book writing endeavours, not to mention my various open-source projects. (We've even had a new language standard in the interim!) Since finishing the latest big batch of work I've been 'reclaiming my life', including spending heaps of time riding my bike, during which the subject of software quality, particularly failure and exceptions, has kept my brain occupied while the roads and trails worked on the rest of me. I have, unsurprisingly, come up with even more questions about exceptions and therefore more material to cover, which will likely require an extension of the previously planned four instalments.

Given the extended break in this column – it's been 15 issues! – I thought it appropriate to start back with a recap of the thinking so far, rather than jump straight into what I'd previously intended to be an instalment on 'Exceptions for Recoverable Conditions', particularly to ensure that the vocabulary that's been introduced is fresh in your mind as we proceed. I also want to revisit material from the previous instalment – 'Exceptions for Practically-Unrecoverable Conditions' [QM-6] – insofar as it's affected by the release of new (open-source) libraries that simplify the definition of program entry points and top-level exception-handling boilerplate. And to give you some confidence that a lot of new material is on its way in the coming months, I will also present an examination of exception-handling in a recent Code Critique that provides an insight into some of the deficiencies of exceptions for precisely reporting failure conditions (whether that's for better contingent reporting or for use in determining recoverability), which will be covered in more detail as the series progresses.

Nomenclature

Although writing software is a highly creative process, it still behoves its practitioners to be precise in intent and implementation. In common with many, no doubt, I struggle greatly with the prevalent lack of precision in discussion about software development in general and in written form in particular: for example, common use of the terms 'error' and 'bug' are replete with ambiguity and overloading, making it very difficult to convey meaning precisely. Therefore, in order to be able to talk precisely about such involved concepts, I have found it necessary to provide precise definitions to some existing terms, and even to introduce some new terms.

In the first – 'Taking Exceptions, part 1: A New Vocabulary' [QM-5] – I suggested a new vocabulary for discussing programming conditions and actions, as follows (and as illustrated in Figure 1):

- *normative* conditions and actions are those that are the main purpose and predominant functioning of the software; and
- *non-normative* conditions and actions are everything else.

Non-normative conditions and actions split into two: *contingent* and *faulted* conditions:

- *Contingent* conditions and actions are associated with handling failures that are according to the design of the software. They further split into two types:
  - *practically-unrecoverable* conditions and actions are associated with failures that prevent the program from executing to completion (or executing indefinitely, if that is its purpose) in a normative manner. Examples might include out-of-memory, disk-full, no-network-connection; and
  - *recoverable* conditions and actions are associated with failures from which the program can recover and to completion (or executing indefinitely, if that is its purpose) in a normative manner. Examples might be user-entered-invalid-number, disk-

New Vocabulary : Is it worth it?

One of the esteemed ACCU reviewers (and a friend of mine) Chris Oldwood offered strong criticism of the vocabulary that I have devised and employed in this column. Chris (quite rightly) points out (i) that the language seems precise, possibly to the point of pedanticism, and (ii) that no-one else is (yet) writing about these issues in these terms. He is right on both.

The problem is hinted at in Chris' second objection: no-one else is (yet) writing about these issues in these terms. Indeed, as far as I have been able to research, no-one is writing about these issues. And that's the real problem. (Acknowledging that I'm not omniscient in any Google-like sense) I have not been able to find any (other) books or articles that dissect the notion of failure in as detailed a manner as I feel I am compelled to do in this column and, indeed, as I have felt compelled to do in my work ever since beginning this column.

So, gentle readers, I'm afraid the high-falutin' language stays, simply because there's no other way I know of to precisely impart what I have to say. If it puts you off, well, I'm sorry (and I guess you won't be alone), but I'm neither able nor willing to attempt to write about such important and nuanced material without having recourse to precisely defined terms, even if I have to invent half of them myself.

One thing I will do in moving towards Chris' position: I will ensure that definitions and exemplifying material for all the terms used are gathered and available in a single place on the Quality Matters website: http://www.quality-matters-to.us/.

Matthew Wilson is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.

diagnostic logging statements are often predominantly used for recording contingent events, but this not need be so

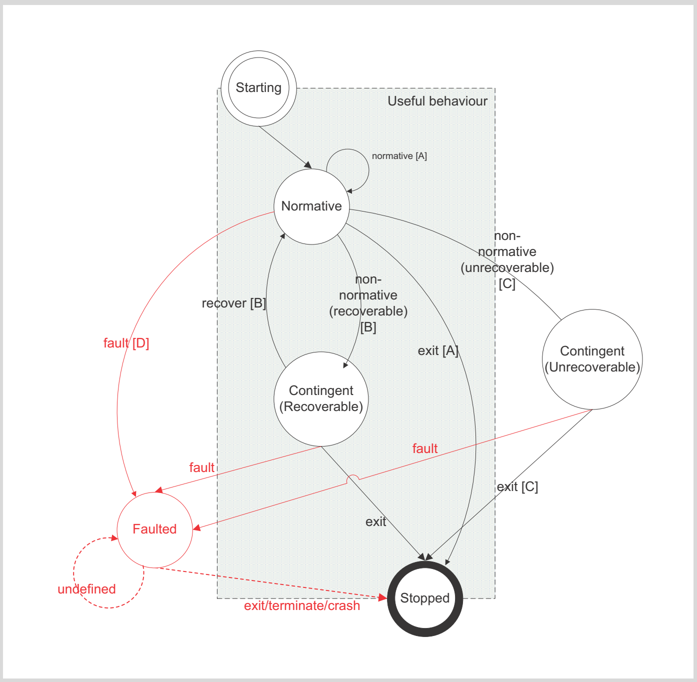


Figure 1

full (if a user/daemon is present and able to free/add additional capacity).

- *faulted* conditions and actions are associated with the program operating outside of its design: the classic 'undefined behaviour'.

These terms will be used (and their definitions assumed) in the remaining instalments of this series and, in all likelihood, most/all future topics of Quality Matters.

Essence of exceptions

Also in [QM-5] I discussed the various uses and meanings of exceptions – some desired; some not – and came to the conclusion that the only definitive thing we can say about exceptions are that they are (an alternative) flow-control mechanism. This might seem pointlessly pedantic, but it's a necessary guide to issues I'll get to in the coming instalments. Promise.

Stereotypes

Also in [QM-5] I discussed exception use stereotypes, such as **exceptions-are-evil**, **exceptions-for-exceptional-conditions**, and so on. I do not reference them directly in this instalment, and will assume you will man the browser if you want to (re)visit them in detail. One aspect of this discussion that does bear repetition here, however, is some of the obvious problems attendant with the use of exceptions, namely:

- Exceptions break code locality: they are invisible in the (normative) code, and create multiple (more) function exit points. Consequently, they significantly impact on code transparency. This occurs in both positive and negative ways, depending on conditions/actions, as we will see in later instalments; and
- Exceptions are quenchable: it is possible (with very few exception(!)s) to catch and not propagate those exceptions that are being used to indicate practically-unrecoverable conditions. (The same applies to exceptions that are used to indicated faulted conditions, though this is predominantly an unwise application of them.) This leaves higher levels of the program unaware of the situation, with obvious concomitant impacts on program reliability.

Reporting

In the second – 'Exceptions for Practically-Unrecoverable Conditions' [QM-6] – I considered the necessity for reporting to a human (or human-like entity) as part of contingent action, in two distinct forms: contingent reports, and diagnostic log statements.

Definition: A **contingent report** is a block of information output from a program to inform its controlling entity (human user, or spawning process) that it was unable to perform its normative behaviour. Contingent reports are a part of the program logic proper, and are not optional.

Definition: A **diagnostic logging statement** is a block of information output from a program to an optional observing entity (human user/administrator, or monitor process) that records what it is doing. Diagnostic logging statements are optional, subject to the principle of removability [QM-1], which states: "It must be possible to disable any log statement within correct software without changing the (well-functioning) behaviour."

Typical contingent reports include writing to the standard error stream, or opening a modal window containing a warning. They are almost always used for issuing important information about *recoverable* or *practically-unrecoverable* conditions.

Similarly, diagnostic logging statements are often predominantly used for recording contingent events, but this not need be so. In principle, all interesting events should be subject to diagnostic logging, to facilitate detailed historical tracing of the program flow. A good diagnostic logging library should allow for statements of different severities to be selectively enabled with minimal intrusion on performance when disabled.

Exception-hierarchies

Also in [QM-6] I discussed the conflation, by some compiler manufacturers, of the C++ exception-handling mechanism and operating-system 'exceptions'/faults, which can allow the quenching of unequivocally fatal conditions via a **`catch(...)`** clause! The consequence of this is that use of **`catch(...)`** should be eschewed in the broad, which, in combination with the fact that programs need a top-level exception-handler ([QM-6]; discussed in the next section), means that all thrown things should be derived from **`std::exception`**.

Recommendation: All thrown entities must (ultimately) be derived from **`std::exception`**.

Boilerplate handling

Also in [QM-6] I examined the inadequacy of the *hello world* introductory program totem: in all compiled languages examined (C, C++, C#/.NET, Java) the classic form is inadequate, and will indicate success even in some failure conditions! The starkly obvious conclusion is that programs written in all these languages require some explicit highest-level exception-handling. I then proffered as example a 'production-quality **`main()`**' used in several of the software analysis tools contemporaneously-(re)developed for the case I've been working on, within which a caller-supplied 'main' was executed within the scope of initialisation of some third-part libraries and subject to a rich set of catch clauses.

You may recall that the rough algorithm was as follows:

1. Catch **`std::bad_alloc`** (and any equivalent, e.g. **`CMemoryException*`**) first, and exit;
2. Catch **CLASP** – a command-line parsing library [ANATOMY-1] – exceptions. Though CLASP is a general-purpose (open-source) library, its use in the analysis tool suite is specific, so CLASP exceptions were caught within the context of expected program behaviour, albeit that the condition of an invalid command-line was in this case – and usually will be in the general case – deemed to be practically-unrecoverable;
3. Next were exceptions from the **recls** library. Again, recls is a general-purpose (open-source) library but its use within the analysis tool suite was specific and could be treated in the same manner as CLASP;
4. A 'standard catch-all' clause, catching **`std::exception`**; and
5. Optionally (i.e. under opt-in preprocessor control), a genuine catch all (**`catch(...)`**) clause.

In all cases, appropriate diagnostic log statements and contingent reports were made (or, at least, attempted), balancing the extremity of the situation (e.g. out-of-memory) with the requirement to provide sufficient diagnostic and user-elucidatory information.

I also discussed a header file supplied with the Pantheios diagnostic logging API library [PAN] that abstracted the boilerplate exception-handling for COM components.

Since that time, I've continued to work on the subject, and have now released two new libraries relevant to the subject:

- **Pantheios::Extras::xHelpers** (C++-only) – provides general exception-catching and translation-to-failure-code services for any components/libraries that need to exhibit a C-API, along with a COM-specific set implemented in terms of the general, and logs the conditions using Pantheios; and
- **Pantheios::Extras::Main** (C, C++) – provides both automatic (un-initialisation of the Pantheios library (otherwise required explicitly in C code) and, for C++, handling of exceptions as follows:
  - **Out-of-memory** exceptions – issues minimal appropriate contingent reports and diagnostic log statements;
  - '**Root**' exceptions (**`std::exception;`** for MFC-only **`CException*`**) – issues contingent reports and diagnostic log statements including exception message; and
  - **Catch-all** clause – issues minimal appropriate contingent reports and diagnostic log statements; not enabled by default (for the reasons explained in [QM-6]), but can be enabled under direction of pre-processor symbol definitions.
- (All other application/library-specific exceptions (such as those shown in Listing 1 of [QM-6]) are to be handled in a deeper, application-supplied handler, thereby achieving a clean(er) separation of the specific and the generic.)

I'll provide more details for these libraries (all downloadable from the Pantheios [PAN] project), at a future time, when their specific features and utility are worth mentioning explicitly. **Pantheios::Extras::Main** will certainly appear in the upcoming issues of CVu when I continue – hopefully next month – my series looking at program anatomies [ANATOMY-1], and again in this series, when I consider the benefits of a dumped core (pointed out to me by readers of [QM-5]).

Use of these libraries now changes significantly the previous production **`main()`**, discussed and listed in [QM-6], to that shown in Listings 1 and 2.

There were three significant problems with the previous 'production quality **`main()`**':

1. It was large and unwieldy;
2. It mixed library initialisation in with contingent action; and
3. Most seriously from a design perspective, it failed to distinguish, ascribe, or respect any abstraction levels of the sub-systems for which the contingencies were provided.

As I look back on the proffered **`main()`** from some distance in time, it is the third issue that strikes me as most grievous. Since that time I have done much work on program structure – discussed in one prior [ANATOMY-1] and several forthcoming articles about 'program anatomy' in CVu – supported by several new libraries, including those mentioned above.

Instead of a single initialisation utility function approach, I now layer the initialisation+failure-handling as follows.

First (i.e. outermost), comes the initialisation of the diagnostic logging facilities, and handling of otherwise-uncaught generic exceptions. This is achieved by the use of **Pantheios::Extras::Main**'s **`invoke()`**, in Listing 1's **`main()`**, which:

1. (un)initialises Pantheios, ensuring that the caller-supplied inner main is always executed within an environment where diagnostic logging facilities are available; and
2. catches those exceptions that are appropriately top-level:
   a. **`std::bad_alloc`** for out-of-memory conditions. This exception can and should be readily ignored by all inner-levels of the program, and represents a bona fide practically-unrecoverable condition; the exception to that will be discussed in – you guessed it! – a later instalment;
   b. **`std::exception`** for the necessary outer-catch behaviour required of us due to the latitude in the standard discussed in [QM-6]. It may be that such general catching at this outer level represents valid behaviour; I tend to think it will rather indicate a failure on the part of the programmer to account for all facets of behaviour in the design: i.e. the more specific (though still **`std::exception`**-derived) exceptions should have been caught in the inner, application-specific, logic levels of the program. (Once again, this being a rather huge and contentious issue, it'll have to wait for a further instalment for further exploration.); and
   c. **`catch(...)`**, but only if the preprocessor symbol **`PANTHEIOS_EXTRAS_MAIN_USE_CATCHALL`** is defined, because this should almost always be eschewed.

Next, I chose to employ another of Pantheios' child libraries: **Pantheios::Extras::DiagUtil**, to provide memory leak detection, in Listing 1's **`main1()`**. (The naming has to be worked on, I know!) Importantly, this happens *after* diagnostic logging is initialised and before it is uninitialised, so that (i) these facilities are available for logging the leaks, and (ii) there are no false-positives in the leak detection due to long-lived lazily-evaluated allocations in the diagnostic logging layer. If you don't want to do memory-leak tracing you can just leave this out (and wire up **`main()`** to **`main2()`**).

Last, comes the command-line handling, in the form of CLASP's child library **CLASP::Main**, in Listing 1's **`main2()`**.

```
// in common header

extern
int main_proper_outer(
  clasp::arguments_t const* args
);
extern clasp_alias_t const aliases[];

. . .

// in file main.cpp
//
// common entry point for all tools

static
int main2(int argc, char** argv)
{
 return clasp::main::invoke(argc, argv,
    main_proper_outer, NULL, aliases, 0, NULL);
}

static
int main1(int argc, char** argv)
{
 return ::pantheios::extras
            ::diagutil::main_leak_trace
                ::invoke(argc, argv, main2);
}

int main(int argc, char** argv)
{
 return ::pantheios::extras
            ::main::invoke(argc, argv, main1);
}
```

Listing 1

```
int
main_proper_outer(
   clasp::arguments_t const* args
)
{
  try
  {
    return main_proper_inner(args);
  }
  // 0
  catch(program_termination_exception& x)
  {
    pan::log_INFORMATIONAL(
     "terminating process under program direction;
      exit-code=", pan::i(x.ExitCode));
    return x.ExitCode;
  }
  // 1. out-of-memory failures now caught by
  //    Pantheios::Extras::Main
  // 2. CLASP failures now caught by CLASP::Main
  // 3. recls
  catch(recls::recls_exception& x)
  {
    pan::log_CRITICAL("exception: ", x);

    ff::fmtln(cerr, "{0}: {1}; path={2};
       patterns={3}", ST_TOOL_NAME, x,
       x.get_path(), x.get_patterns());
  }
  // 4. standard exception failures now caught by
  //    Pantheios::Extras::Main
  return EXIT_FAILURE;
}
```

Listing 2

Note that so far we've not seen any explicit mention of exceptions, even though all three libraries perform exception-handling within:

- **`pantheios::extras::main::invoke()`** catches **`std::bad_alloc`** (and **`CMemoryException*`** in the presence of MFC);
- **`pantheios::extras::diagutil::main_leak_trace::invoke()`** catches any exceptions emitted by its 'main' and issues a memory leak trace before re-throwing to its caller; and
- **`clasp::main::invoke()`** catches CLASP-specific exceptions, providing the pseudo-standard contingent reports along the lines of 'myprogram: invalid command-line: unrecognised argument: --oops=10' (and, of course, diagnostic library statements in the case where the CLASP code detects it is being compiled in the presence of Pantheios); all other exceptions are ignored, to be dealt with by the outer-layers.

Each of these is designed to be entirely independent of the others, and it's entirely up the programmer whether or not to incorporate them, and any similar layering facilities, in whatever combination is deemed appropriate. All that's required is to provide whatever declarative information is required – the example shown just provides 0s and NULLs for default behaviour – and rely on a sensible, predictable, and dependable behavioural foundation to the program.

With the (boring) boilerplate out of the way in this fashion, the programmer may then focus on what are the more interesting aspects of failure (as reported by exceptions) his/her program may encounter. In a simple standalone program, the next and final stage might be the 'main proper'. However, more sophisticated programs that depend on other libraries that may themselves throw exceptions to report practically-unrecoverable and recoverable conditions may elect to have shared, program-suite-specific 'outer main proper', as I have done so with the analysis tool suite, which looks something like that shown in Listing 2. (This illustrates use of a **`program_termination_exception`** class, which is yet another thing I'll discuss later. Prizes for the first reader to write in with why such an exception might be used in preference to a simple call to **`exit()`**.)

Code Critique #73

One of the things I did to keep my programming mojo idling over during my long engagement was to solve the 73rd Code Critique [CC-73].

The first thing to note is that I misread the program description, and so got slightly wrong requirements for my own implementation! The actual requirement was to read from standard input, whereas I misunderstood it to mean that it should read from a named file. Fortunately, the change is useful in illustrating the points I wish to make. Other than that, I *think* I have it right, as follows:

- obtain name of input text file from command-line (this is the requirement I misread);
- open source text file, and read lines, then for each line:
  - upon finding a line – *control line* – with the format (in regex parlance) **`/^----  (.+)  ----$/`**, close the previous output file, if any, and open a new output file with the name specified in (regex parlance again) group **`$1`**;
  - upon reading any other kind of line – *data line* – write it out to the currently-open output file;
- when all lines are done, exit.

The reason this example sparked my interest in cross-pollination with QM is that it exercises one of my interests with respect to software quality:

1. The impact of process-external interactions – in this case the file-system – on the software quality of a program;

along with two of my frequent preoccupations with C++:

2. When is it more appropriate to use another language to solve a problem; and
3. The IOStreams library, and the myriad reasons why I hate it.

```
int main(int argc, char** argv)
{
 std::ifstream ifs(argv[1]);
 std::ofstream ofs;
 std::string           line;

 for(; std::getline(ifs, line); )
 {
  if( line.find("--- ") == 0 &&
      line.find(" ---") == line.size() - 4)
  {
   std::string const path =
      line.substr (4, line.size() - 8);
   if(ofs.is_open())
   {
    ofs.close();
   }
   ofs.open(path.c_str());
  }
  else
  {
   ofs << line << "\n";
  }
 }
 return EXIT_SUCCESS;
}
```

Listing 3

When working with the file-system, we must care deeply about failure, since failure awaits us at every turn: capacity; hardware failures; input; output; search; user directions; and so on. I'll cover some of the relevant issues (as they pertain to failure detection and handling, and exceptions) in the remainder of this instalment. (Note that the steps are written for maximum *revisibility* [ANATOMY-1], and don't necessarily reflect how I would proceed normally.)

Let's start with a fully working version – in the sense that it addresses the normative requirements outlined above – and then see how it behaves when presented with an imperfect execution environment. This is given in Listing 3. When operated with the input file shown in Listing 4 – where ¶ denotes an end-of-line sequence, missing on the last line – it experiences normative-only conditions and produces the intended two output files (`outputfile-a.txt` and `outputfile b.txt`).

However, we don't have to try very hard to break it. Simply omitting the input-file command-line argument will cause a dereference of **`NULL`** (**`argv[1]`**), and undefined behaviour (in the form of a segmentation fault / access violation => crash). This is easily fixed by the addition of a test in Step 1 (Listing 5), causing issue of a contingent report and return of **`EXIT_FAILURE`**; I'm omitting diagnostic logging from these examples for brevity.

It's not just omitting the input-file command-line argument. Equally wrong is to supply two arguments. This is easily addressed (Step 2; listing not shown) by changing the test expression from:

```
  if(NULL == argv[1])
```

```
--- outputfile-a.txt ---¶
abc    ¶
--- abc¶
def    ¶
¶
--- outputfile b.txt ---¶
ghi¶
jklm¶
¶
nop
```

Listing 4

```
int main(int argc, char** argv)
{
 if(NULL == argv[1])
 {
  std::cerr << "step1 : <inputfile>" << std::endl;
  return EXIT_FAILURE;
 }

 std::ifstream ifs(argv[1]);

 . . . // as before

 return EXIT_SUCCESS;
}
```

Listing 5

to:

```
  if(2 != argc)
```

Furthermore, we learned last time that any production-quality program requires an outer try-catch, since otherwise any exception thrown by the program (including **`std::bad_alloc`**, which is the only one that may be thrown by the versions so far) will cause an unexplained – by dint of a lack of a *guaranteed* (i.e. standard-prescribed) contingent report – **`abort()`** (via **`std::terminate()`**). Fulfilling this gives Step 3, shown in Listing 6; all subsequent steps will assume **`main_inner()`**.

With just a modicum of consideration, I'm able to come up with eight potential problems that would result in failure of the program, including the first two:

```
static
int main_inner(int, char**);

int main(int argc, char** argv)
{
 try
 {
  return main_inner(argc, argv);
 }
 catch(std::bad_alloc&)
 {
  fputs("step3 : out of memory\n", stderr);
 }
 catch(std::exception& x)
 {
  fprintf(
    stderr
  , "step3 : %s\n"
  , x.what()
  );
 }
 return EXIT_FAILURE;
}

int main_inner(int argc, char** argv)
{
 if(2 != argc)
 {
  std::cerr << "step3 : <inputfile>" << std::endl;
  return EXIT_FAILURE;
 }

 . . . // as before

 return EXIT_SUCCESS;
}
```

Listing 6

```
int main_inner(int argc, char** argv)
{
 if(2 != argc)
 {
  std::cerr << "step4: <inputfile>" << std::endl;
  return EXIT_FAILURE;
 }
 char const* const inputPath = argv[1];
 std::ifstream ifs(inputPath);
 std::ofstream ofs;
 std::string line;
 std::string path;
 for(; std::getline(ifs, line); )
 {
  if( line.find("--- ") == 0 &&
      line.find(" ---") == line.size() - 4)
  {
   path = line.substr(4, line.size() - 8);
   if(ofs.is_open())
   {
    ofs.close();
   }
   ofs.open(path.c_str());
  }
  else
  {
   if(path.empty())
   {
    std::cerr << "step4: invalid input file '"
         << inputPath
         << "': data line(s) before control line"
         << std::endl;
    return EXIT_FAILURE;
   }
   ofs << line << "\n";
  }
 }
 return EXIT_SUCCESS;
}
```

Listing 7

1. Input file path argument not specified by user;
2. 2+ arguments specified by user;
3. Input file contains data line before the first control line;
4. Input file does not exist;
5. Input file cannot be accessed;
6. Control line specifies output path that cannot exist (because directory component does not exist);
7. Control line specifies output path that cannot exist (because path contains invalid characters); and
8. Control line specifies output path of file that exists and is read-only.

I'll measure the normative version, and the other versions I'll introduce in response, against these eight problems. Note that all eight are eminently reasonable runtime conditions: none is caused by conditions, or should invoke responses, that are (or should be) outside the design. In our parlance: none of these should result in a faulted condition.

The first two problems we've already encountered; the last five pertain to the file-system, and will be the main points of interest in this section. Before that, however, I must address a key piece of functionality, which is to handle the case where the input file specifies data lines before control lines (problem #3). Let's assume an input file, `inputfile-3.txt`, that is the same as `inputfile-0.txt` with a single data line "xyz" before the first control line. The current version, Step 3 (Listing 6), silently ignores it, 'successfully' creating the two (other) output files. This has to be caught, as shown in Step 4 (Listing 7): the 'un-**`const`**-ising' (forgive me!) of **`path`**, somewhat regrettable in its own terms, and moving it outside allows of the loop allows it to be used as an indicator as to whether an output file is open, with the corresponding contingent report and

```
int main_inner(int argc, char** argv)
{
 . . .
 char const* const inputPath = argv[1];
 std::ifstream ifs(inputPath);
 std::ofstream ofs;
 std::string line;
 std::string path;
 if(ifs.fail())
 {
  std::cerr << "step5: could not open '"
            << inputPath << "'" << std::endl;
  return EXIT_FAILURE;
 }
 for(; std::getline(ifs, line); )
 {
  . . .
}
```

Listing 8

`EXIT_FAILURE` if not. Note the precise and sufficiently-rich explanation provided in the contingent report, which will allow a user (who knows how the program is supposed to work) to correct the problem with the input file.

Let's now turn our attention to problems 4 and 5. As discussed in [QM-6], the IOStreams do not, by default, throw exceptions on failure conditions. In this mode, to detect whether an input file does not exist or cannot be accessed, the programmer must explicitly test state member functions. Absent such checks, as in Step 4 (Listing 7), the program will silently fail (and indicate success via **`EXIT_SUCCESS`**!).

We have two choices: either test the state of the streams at the appropriate point or cause the input stream to throw exceptions (and catch them). Step 5 (Listing 8) takes the former approach. While this detects the failure event – the fact that the input file cannot be open – it does not provide any reason as to what caused the failure: for our purposes there is no discrimination between problems 4 and 5. At a minium, we would want to include some information in our contingent report that indicates to the user which one it is.

We might consider the following logic: All C++ standard library implementations (that I know of) are built atop the C standard library's Streams library (**`FILE*`**, **`fprintf()`**, etc.), which uses **`errno`**, so I can use **`errno`** and **`strerror()`** to include contingent reporting. This might take us to Step 6 (Listing 9). This appears to work for me with VC++ for both Problem 4:

```
step6: could not open 'not-exist.txt': No such
file or directory
```

```
#include <fstream>
#include <iostream>
#include <string>
#include <cerrno>
#include <cstdlib>
#include <cstring>

 . . .

 if(ifs.fail())
 {
  int const e = errno;
  std::cerr << "step5: could not open '"
            << inputPath << "': "
            << strerror(e) << std::endl;
  return EXIT_FAILURE;
 }

 . . .
```

Listing 9

and Problem 5:

```
step6: could not open 'inputfile-0-NO-READ.txt':
Permission denied
```

The same is to be had from the CodeWarrior 8 compiler (on Windows):

```
step6: could not open 'not-exist.txt': No such
file or directory
```

and:

```
step6: could not open 'inputfile-0-NO-READ.txt':
Operation not permitted
```

The problem is that nowhere in the C++ (03) standard, or nowhere that I could find anyway, does it mandate that the IOStreams will be implemented in terms of C's Streams, nor that it will faithfully maintain and propagate **errno**. (All that is mandated is that the standard input, output, and error streams will be synchronisable with C++'s IOStreams' **cin**, **cout**, and **cerr**.) Compiling Step 6 with Digital Mars compiler results in:

```
step6: could not open 'not-exist.txt': No error
```

and:

```
step6: could not open 'inputfile-0-NO-READ.txt':
No error
```

indicating that **errno** is **0** in those cases.

Clearly this is not a portable strategy for propagation of failure reason to the user via the contingent report (or via diagnostic logging). Nor is it a reliable mechanism for making programmatic decisions regarding what contingent actions to take. In one of the next instalments I'm going to show just how much rubbishy hassle is attendant with this problem, particularly (though perhaps not exclusively) in .NET.

Let's now consider the alternative approach of using exceptions, as in Step 7 (Listing 10). At face value, it appears that this is an improvement: we've certainly improved the transparency (with respect to the normative behaviour of the program).

However, when we run this (with VC++) we get the following output:

```
step7 : ios::failbit set
```

and:

```
step7 : ios::failbit set
```

That's pretty depressing. By setting the stream to emit exceptions we've gained the automatic and unignorable indication of failure, which is good. But we've failed to gain any useful qualifying information. Even to a programmer, being told "ios::failbit set" is pretty useless; to a user it's arguably less than that. Furthermore, since the standard defines **std::ios::failure** to have only **constructor**, **destructor**, and **what()** overload – there's no **underlying_failure_code()** accessor method or any equivalent – the exception cannot provide any information that could be used programmatically.

```
int main_inner(int argc, char** argv)
{
 . . .
 char const* const inputPath = argv[1];
 std::ifstream ifs(inputPath);
 std::ofstream ofs;
 std::string line;
 std::string path;

 ifs.exceptions(std::ios::failbit);

 for(; std::getline(ifs, line); )
 {
  . . .
 }
 return EXIT_SUCCESS;
}
```

Listing 10

```
int main_inner(int argc, char** argv)
{
 . . .
 char const* const inputPath = argv[1];
 . . .
 ifs.exceptions(std::ios::failbit);

 try
 {
  for(; std::getline(ifs, line); )
  {
   . . .
  }
 }
 catch(std::ios::failure&)
 {
  if(!ifs.eof())
  {
   throw;
  }
 }

 return EXIT_SUCCESS;
}
```

Listing 11

But hold on. It gets (much) worse. If we now supply a *valid* input file we still get the same contingent report, but after it has produced the requisite output file(s). What gives?

Sadly, the (C++03) standard prescribes (via a convoluted logic including clauses 27.6.1.1.2;2 and 21.3.7.9;6) that when **std::getline()** encounters the end-of-file condition it throws an exception if the stream is marked to throw on **std::ios::failbit**, even when, as in our case, it is not marked to throw on **std::ios::eofbit**. I've little doubt that there's a necessary backwards-compatibility reason for this, but in my opinion this just leaves us with ludicrous behaviour. (Ignoring the almost equally ludicrous set-exceptions-after-construction anti-idiom), what could be more transparent than the following:

```
std::ifstream ifs(path);
ifs.exceptions(std::ios::failbit);
for(std::string line; std::getline(ifs, line); )
{
  . . . // do something with line
}
. . . // do post-read stuff
```

You don't get to write that. At 'best', you must write something like the following (and in Step 8, Listing 11):

```
std::ifstream ifs(path);
ifs.exceptions(std::ios::failbit);
try
{
  for(std::string line; std::getline(ifs, line);)
  {
    . . . // do something with line
  }
}
catch(std::ios::failure&)
{
  if(!ifs.eof())
  {
    throw;
  }
}
. . . // do post-read stuff
```

Are we having fun yet?

Let's now consider the final three problems (6–8), all of which pertain to the ability to create the specified output file. If we do this manually, it'll

```
int main_inner(int argc, char** argv)
{
 if(2 != argc)
 {
  std::cerr << "step9: <inputfile>" << std::endl;

  return EXIT_FAILURE;
 }

 char const* const inputPath = argv[1];
 std::ifstream  ifs(inputPath);
 std::ofstream  ofs;
 std::string   line;
 std::string   path;

 ifs.exceptions(std::ios::failbit);
 ofs.exceptions(std::ios::failbit);

 try
 {
  for(; std::getline(ifs, line); )
  {
   if( line.find("--- ") == 0 &&
    line.find(" ---") == line.size() - 4)
   {
    path = line.substr(4, line.size() - 8);
    if(ofs.is_open())
    {
     ofs.close();
    }
    try
    {
     ofs.open(path.c_str());
    }
```

Listing 12

involve testing **`ofs.fail()`** after the call to **`ofs.open()`** inside the loop. (For completeness we'd probably also want to test after the insertion of **`line`** into **`ofs`**, but I'm trying to be brief here …) But as we've seen with the input file/stream, we're still out of luck in discriminating which of the actual conditions 6–8 (or others) is responsible for the failure – and even if we could, there are only two (C++03) standard-prescribed **`errno`** values, **`EDOM`** and **`ERANGE`**, neither of which are applicable here – and might not even get a representative (albeit platform/compiler-specific) 'error' string from **`strerror()`**.

Similarly, if we call **`exceptions()`** on **`ofs`** without more fine-grained catching, we have two obvious problems:

- loss of precision as to location of exception, and, hence, the cause; and
- no more informative (**i.e. not!**) messages than we've seen provided with input file/stream failure.

There's nothing we can do about the latter, but we *can* address the former, by catching with greater granularity to afford identification of the failure locations, as in Listing 12. I hope you agree with me that the implementation as it now appears is patently inferior: we're polluting the pretty straight-forward functionality of this simple program with too much failure-handling, a clear loss of transparency. Furthermore, there is no clear delineation between application-specific failure – e.g. the presence of data prior to output-file-name in an input file – and general (in this case file-system) failure. This is hardly the promise of clarity in failure-handling that exception proponents might have us believe.

This conflict is something I will be exploring further next time, including considerations of precision and accuracy of identifying and reporting (sufficient information about) the failure, use of state, issues of coupling between throw and call sites, and further conflicts in software quality characteristics arising from the use, non-use, and misuse of exceptions. ■

```
    catch(std::ios::failure&)
    {
     std::cerr
        << "step9: could not open output file '"
        << path << "'" << std::endl;

     return EXIT_FAILURE;
    }
   }
   else
   {
    if(path.empty())
    {
     std::cerr << "step9: invalid input file '"
        << inputPath
        << "': data line(s) before control line"
        << std::endl;

     return EXIT_FAILURE;
    }

    try
    {
     ofs << line << "\n";
    }
    catch(std::ios::failure&)
    {
     std::cerr
      << "step9: could not write to output file '"
      << path << "'" << std::endl;

     return EXIT_FAILURE;
    }
   }
  }
 }
 catch(std::ios::failure&)
 {
  if(!ifs.eof())
  {
   throw;
  }
 }

 return EXIT_SUCCESS;
}
```

Listing 12 (cont'd)

References

[ANATOMY-1] 'Anatomy of a CLI Program Written in C', Matthew Wilson, *CVu* volume 24, issue 4

[CC-73] *CVu* vol 23, issue 6

[PAN] http://www.pantheios.org/

[QM-1] 'Quality Matters 1: Introductions and Nomenclature,' Matthew Wilson, *Overload* 92, August 2009

[QM-5] 'Quality Matters 5: Exceptions: The Worst Form of 'Error' Handling, Apart from all the Others', Matthew Wilson, *Overload* 98, August 2010

[QM-6] 'Quality Matters 6: Exceptions for Practically-Unrecoverable Conditions', Matthew Wilson, *Overload* 99, October 2010

Causality – Relating Distributed Diagnostic Contexts

Supporting a system with many moving parts can be hard. Chris Oldwood demonstrates one way to add tags to log information to aid diagnosis.

Causality: The relationship between cause and effect [OED].

Supporting a Distributed System can be hard. When the system has many moving parts it is often difficult to piece together exactly what happened, at what time, and within which component or service. Despite advances in programming languages and runtime libraries the humble text format log file is still a mainstream technique for recording significant events within a system whilst it operates. However, logging libraries generally only concern themselves with ensuring that you can piece together the events from a single process; the moment you start to invoke remote services and pass messages around the context is often lost, or has to be manually reconstructed. If the recipient of a remote call is a busy multi-threaded service then you also have to start picking the context of interest out of all the other contexts before you can even start to analyse the remote flow.

This article will show one mechanism for overcoming this problem by borrowing a hidden feature of DCOM and then exposing it using an old design pattern from Neil Harrison.

Manually stitching events together

The humble text log file is still the preferred format for capturing diagnostic information. Although some attempts have been made to try and use richer encodings such as XML, a simple one line per event/fixed width fields format is still favoured by many [Nygard].

For a single-process/single-threaded application you can get away with just a timestamp, perhaps a severity level and the message content, e.g.

```
2013-01-01 17:23:46 INF Starting something rather
important
```

Once the number of processes starts to rack up, along with the number of threads you need to start including a Process ID (PID) and Thread ID (TID) too, either in the log file name, within the log entries themselves, or maybe even in both, e.g.

```
2013-01-01 17:23:46 1234 002 INF Starting
something rather important
```

Even if you are mixing single-threaded engine processes with multi-threaded services it is still desirable to maintain a consistent log file format to make searching and parsing easier. For the sake of this article though, which is bound by the constraints of print based publishing, I'm going to drop some of the fields to make the log output shorter. The date, PID and severity are all tangential to most of what I'm going to show and so will be dropped leaving just the time, TID and message, e.g.

```
17:23:46 002 Starting something rather important
```

Assuming you can locate the correct log file to start with, you then need to be able to home-in on the temporal set of events that you're interested in. One common technique for dealing with this has been to manually annotate log lines with the salient attributes of the task inputs, e.g.

```
17:23:45 002 Handling request from 192.168.1.1
for oldwoodc
17:23:46 002 Doing other stuff now
17:23:47 002 Now doing even more stuff
. . .
17:23:59 002 Finished handling request from
192.168.1.1
```

If your process is single-threaded you can probably get away with putting the key context details on just the first and last lines, and then just assume that everything in between belongs to the same context. Alternatively you can try and 'remember' to include the salient attributes in every log message you write.

```
17:23:45 002 Handling request from 192.168.1.1
17:23:46 002 Doing other stuff now (192.168.1.1)
17:23:47 002 [192.168.1.1] Now doing even more
stuff
. . .
17:23:59 002 Finished handling from 192.168.1.1
```

Either way there is too much manual jiggery-pokery going on and as you can see from the last example you have to rely on all developers using a consistent style if you want a fighting chance of filtering the context successfully later.

Diagnostic contexts

The first problem we want to solve is how to 'tag' the current context (i.e. a single thread/call stack in the first instance) so that whenever we go to render a log message we can automatically annotate the message with the key details (so we can then grep for them later). More importantly, we'd like to do this in such a way that any code that is not already aware of our higher-level business goals remains blissfully unaware of them too.

In *Pattern Languages of Program Design Vol. 3*, Neil Harrison presents a number of logging related design patterns [Harrison], one of which is called DIAGNOSTIC CONTEXT. In it he describes a technique for associating arbitrary data with what he calls a 'transaction'. The term transaction is often heavily associated with databases these days, but the transactions we are concerned with here are on a much larger scale, e.g. a single user's 'session' on a web site.

A distributed system would therefore have many diagnostic contexts which are related somehow. The connection between these could be viewed as a parent/child relationship (or perhaps global/local). There is no reason why a context couldn't store different 'categories' of tags (such as problem domain and technical domain), in which case the term namespace might be more appropriate. However this article is not so much concerned with the various scopes or namespaces that you might create to partition your contexts but more about how you go about *relating* them. As you will see later it is a specific subset of the tags that interests us most.

Although you could conceivably maintain one context per task that acquires more and more tags as you traverse each service layer, you would

Chris Oldwood started out as a bedroom coder in the 80s, writing assembler on 8-bit micros. These days it's C++ and C# on Windows in big plush corporate offices. He is also the commentator for the Godmanchester Gala Day Duck Race and can be contacted via gort@cix.co.uk or @chrisoldwood

```
public void ProcessRequest(Request request)
{
  using(new DiagnosticContextTag("ID",
                                 request.Id))
  using(new DiagnosticContextTag("HOST",
                                 request.Host))
  {
    // Do some funky stuff with the request.
    . . .
  }
}
```

Listing 1

in effect be creating a Big Ball of Mud. However, the more tags you create the more you'll have to marshal and ultimately the more you'll have to write to your log file and then read again when searching. Although the I/O costs should be borne in mind, the readability of your logs is paramount if you're to use them effectively when the time comes. And so multiple smaller contexts are preferred, with thread and service boundaries providing natural limits.

Implementing a simple diagnostic context

The implementation for a DIAGNOSTIC CONTEXT can be as simple as a map (in C++) or a Dictionary (in C#) which stores a set of string key/value pairs (a tag) that relates to the current operation. The container will almost certainly utilise thread-local storage to allow multiple contexts to exist simultaneously for the multiple threads within the same process.

It should be noted that some 3rd party logging frameworks already have support for diagnostic contexts built-in. However, they may not be usable in the way this article suggests and so you may still need an implementation like the simple one shown below.

At the entry point to our 'transaction' processing code we can push the relevant tags into the container for use later. By leveraging the RAII idiom in C++ or the DISPOSE pattern in C# we can make the attaching and detaching of tags automatic, even in the face of exceptions. For example in C# we could write the code in Listing 1.

Behind the scenes the constructor adds the tag to the underlying container and removes it again in the destructor/DISPOSE method. The need for the code to be exception safe is important as we don't want the tags of one context to 'leak' by accident and infect the parent context because it would cause unnecessary pollution later when we are searching.

As Harrison's original pattern suggests we can create contexts-within-contexts by using stack-like push/pop operations instead of just a simple add/remove. However you still want to be careful you don't overload the meaning of any tag (e.g. 'ID') that will be used across *related* scopes as, once again, it will only create confusion later.

When the time finally comes to render a log message we can extract the set of tags that relate to this thread context, format them up nicely and append them to the caller's message behind the scenes, as in Listing 2.

The example above would generate a log line like this:

```
17:23:46 002 Doing other stuff now [ID=1234]
```

The statement `Context.Format();` hopefully shows that I've chosen here to implement the diagnostic context as a static Façade. This is the same façade that the constructor and destructor of `DiagnosticContextTag` would have used earlier to attach and detach the attributes. In C# the diagnostic context could be implemented like Listing 3.

```
public void WriteLogMessage(string message)
{
  string timestamp = FormatTimestamp();
  string threadId = FormatThreadId();
  string context = Context.Format(); // "ID=1234"

  Console.WriteLine("{0} {1} {2} [{3}]",
     timestamp, threadId, message, context);
}
```

Listing 2

```
public static class Context
{
  internal static void Attach(string key,
                              string value)
  {
    s_tags.Add(key, value);
  }
  internal static void Detach(string key)
  {
    s_tags.Remove(key);
  }
  public static string Format()
  {
    var builder = new StringBuilder();
    foreach(var tag in s_tags)
    {
      builder.AppendFormat("{0}={1}",
                           tag.Key, tag.Value);
    }
    return builder.ToString();
  }
  [ThreadLocal]
  private static IDictionary<string, string>
     s_tags = new Dictionary<string, string>();
}
```

Listing 3

The `Attach`/`Detach` methods here have been marked `internal` to show that tags should only be manipulated via the public `DiagnosticContextTag` helper class. (See Listing 4.)

Distributed COM/COM+

The second aspect of this mechanism comes from DCOM/COM+. Each call-chain in DCOM is assigned a unique ID (a GUID in this case) called the Causality ID [Ewald]. This plays the role of the *Logical* Thread ID as the function calls move across threads, outside the process to other local processes and possibly even across the wire to remote hosts (i.e. RPC). In DCOM this unique ID is required to stop a call from deadlocking with itself when the logical call-chain suddenly becomes re-entrant. For example Component A might call Component B (across the wire) which locally calls C which then calls all the way back across the wire into A again. From A's perspective it might seem like a new function call but via the Causality ID it can determine that it's actually just an extension of the original one.

This Causality ID is allocated by the COM infrastructure and passed around transparently – the programmer is largely unaware of it.

```
public class DiagnosticContextTag : IDispose
{
  public DiagnosticContextTag(string key,
                              string value)
  {
    Context.Attach(key, value);
    m_key = key;
  }
  public void Dispose()
  {
    Context.Detach(m_key);
  }
  private string m_key;
}
```

Listing 4

```
public void ProcessRequest(Request request)
{
  using(Causality.Attach("RequestId",
                         request.Id))
  {
    foreach(var task in request.Tasks)
    {
      Log.WriteLogMessage("Starting request");
      ProcessTask(task);
      Log.WriteLogMessage ("Request completed");
    }
  }
}
public void ProcessTask(Task task)
{
  using(Causality.Attach("TaskId", task.Id))
  {
    Log.WriteLogMessage ("Starting task");
    . . .
    Log.WriteLogMessage ("Task completed");
  }
}
```

Listing 5

The primary causality

The Causality mechanism is therefore nothing more than a combination of these two ideas. It is about capturing the primary tags used to describe a task, action, operation, etc. and allowing them to be passed around, both within the same process and across the wire to remote services in such a way that it is mostly transparent to the business logic.

As discussed earlier, the reason for distilling the entire context down into one or more simple values is that it reduces the noise as the processing of the task starts to acquire more and more ancillary tags as you move from service to service. The local diagnostic context will be useful in answering questions within a specific component, but the primary causality will allow you to relate the various distributed contexts to one another and navigate between them.

A GUID may be an obvious choice for a unique causality ID (as DCOM does), and failing any alternatives it might just have to do. But they are not very pleasing to the eye when browsing log data. If the request is tracked within a database via an Identity column then that could provide a succinct integral value, but it's still not easy to eyeball.

A better choice might be to use some textual data from the request itself, perhaps in combination with an ID, such as the name of the customer/user invoking it. The primary causality could be a single compound tag with a separator, e.g. 1234/Oldwood/192.168.1.1 or it could be stored as separate tags, e.g. ID=1234, LOGIN=Oldwood, IP=192.168.1. Ultimately it's down to grep-ability but the human visual system is good at spotting patterns too and if it's possible to utilise that as well it's a bonus.

Putting all this together so far, along with a static helper, **`Causality.Attach()`**, to try and reduce the client-side noise, we could write the single-threaded, multi-step workflow (a top-level request that invokes various sub-tasks) in Listing 5.

This could generate the output in Figure 1.

The decision on whether to extend the primary causality with the **`TaskId`** or just let it remain part of the local context will depend on how easy it is for you to piece together your workflow as it crosses the service boundaries.

```
17:50:01 001 Starting request [RequestId=1234]
17:50:02 001 Starting task [RequestId=1234;TaskId=1]
17:50:02 001 Doing user stuff [RequestId=1234;TaskId=1;User=Chris]
. . .
17:50:03 001 Task completed [RequestId=1234;TaskId=1]
17:50:02 001 Starting task [RequestId=1234;TaskId=2]
17:50:02 001 Doing payment stuff
[RequestId=1234;TaskId=2;Type=Card]
. . .
17:50:03 001 Task completed [RequestId=1234;TaskId=2]
. . .
17:50:01 001 Request completed [RequestId=1234]
```

Figure 1

```
interface IBugTrackerService
{
  Bug FetchBug(int id);
}
class BugTrackerProxy : IBugTrackerService
{
  public Bug FetchBug(int id)
  {
    . . .
  }
}
```

Listing 6

Marshalling the primary causality across the wire

We've removed much of the tedium associated with logging the context for a single-threaded operation, but that leaves the obvious question – how do you pass that across the wire to another service? We don't usually have the luxury of owning the infrastructure used to implement our choice of transports but there is nothing to stop us differentiating between the *logical* interface used to make a request and the *wire-level* interface used to implement it. The wire-level interface may well already be different if we know of a more efficient way to transport the data (e.g. compression) when serializing. If we do separate these two concerns we can place our plumbing right on the boundary inside the proxy where the client can remain unaware of it, just as they are probably already unaware there is an RPC call in the first place.

The *logical* interface in Listing 6 describes the client side of an example service to request the details of a 'bug' in an imaginary bug tracking system.

The client would use it like so:

```
// somewhere in main()
var service = BugTrackerProxyFactory.Create();
. . .
// somewhere in the client processing
var bug = service.FetchBug(bugId);
```

What we need to do when passing the request over the wire is to tag our causality data on the end of the existing parameters. To achieve this we have a separate *wire-level* interface that 'extends' the methods of the logical one:

```
interface IRemoteBugTrackerService
{
  Bug FetchBug(int id, List<Tag> causality);
}
```

Then, inside the client proxy we can hoist the primary causality out of the diagnostic context container and pass it across the wire to the service's remote stub (Listing 7).

We then need to do the reverse (inject the primary causality into the new call stack) inside the remote stub on the service side (Listing 8).

In this example the client proxy (**`BugTrackerProxy`**) and service stub (**`RemoteBugTrackerService`**) classes merely provide the mechanism

```
class BugTrackerProxy : IBugTrackerService
{
  public Bug FetchBug(int id)
  {
    var causality =
       Causality.GetPrimaryCausality();
    return m_remoteService.FetchBug(id,
                                causality);
  }
  private
     IRemoteBugTrackerService m_remoteService;
}
```

Listing 7

```
class BugTrackerServiceImpl : IBugTrackerService
{
. . .
}
class RemoteBugTrackerService
   : IRemoteBugTrackerService
{
  public Bug FetchBug(int id, List<Tag> causality)
  {
    using
      (Causality.SetPrimaryCausality(causality))
    {
      return m_service.FetchBug(id);
    }
  }
  private BugTrackerServiceImpl m_service;
}
```

Listing 8

for dealing with the non-functional data. Neither the caller nor the service implementation class (`BugTrackerServiceImpl`) are aware of what's going on behind their backs.

In fact, as a double check that concerns are correctly separated, we should be able to invoke the real service implementation directly instead of the client proxy and still get the same primary causality appearing in our log output:

```
//var service = BugTrackerClientFactory.Create();
  var service = new BugTrackerServiceImpl();
  . . .
  var bug = service.FetchBug(bugId);
```

Marshalling the primary causality to other threads

Marshalling the primary causality from one thread to another can be done in a similar manner as the remote case. The main difference is that you'll likely already be using your language and runtime library in some way to hide some of the grunge, e.g. by using a delegate/lambda. You may need to give this up slightly and provide the proverbial 'extra level of indirection' by wrapping the underlying infrastructure so that you can retrieve and inject the causality around the invocation of the business logic. Your calling code should still look fairly similar to before:

```
Job.Run(() =>
{
  backgroundTask.Execute();
});
```

However instead of directly using `Thread.QueueUserWorkItem` we have another static façade (`Job`) that will marshal the causality behind the delegate's back (Listing 9).

Marshalling via exceptions

In the previous two sections the marshalling was very much one-way because you want to unwind the diagnostic context as you return from each scope. But there is another way to marshal the causality, which is via exceptions. Just as an exception in .Net carries around a call stack for the point of the throw and any inner exceptions, it could also carry the causality too. This allows you to avoid one common (anti) pattern which is the 'log and re-throw' (Listing 10).

```
public class Job : IRunnable
{
  public static void Run(Action action)
  {
    var causality =
       Causality.GetPrimaryCausality();
    ThreadPool.QueueUserWorkItem((o) =>
    {
      using
        (Causality.SetPrimaryCausality(causality))
      {
        action();
      }
    });
  }
}
```

Listing 9

```
try
{
  // Read a file
}
catch (Exception e)
{
  Log.Error("Failed to read file '{0}'",
            filename);
  throw;
}
```

Listing 10

```
public class CustomException : Exception
{
  public CustomException(string message)
    : base(message)
  {
    m_causality = Causality.GetPrimaryCausality();
  }
  private List<Tag> m_causality;
}
```

Listing 11

The only reason the try/catch block exists is to allow you to log some aspect of the current operation because you know that once the call stack unwinds the context will be gone. However, if the exception captured the causality (or even the entire diagnostic context) in its constructor at the point of being thrown this wouldn't be necessary. You also won't have a 'spurious' error message either when the caller manages to completely recover from the exception using other means. (See Listing 11.)

Naturally this only works with your own exception classes, and so you might end up catching native exceptions anyway and re-throwing your own custom exception types just to capture the causality. However, you've avoided the potentially spurious log message though so it's still a small net gain.

If the exception flows all the way back to the point where the transaction started you can then log the captured causality with the final exception message. In some cases this might be enough to allow you to diagnose the problem without having to find the local context where the action took place.

```
public void ProcessAdditionRequest(Request
request)
{
  try
  {
    using (Causality.Attach
        ("Request", request.Id))
    using (Causality.Attach
        ("User", request.Login))
    using (Causality.Attach("Host", request.Host))
    {
      using (Instrument.MeasureElapsedTime
          ("Addition"))
      {
        request.Answer =
           request.Left + request.Right;
      }
    }
  }
  catch (MyException e)
  {
    // Recover from known problem
  }
  catch (Exception e)
  {
    // Recover from unknown problem
  }
}
```

Listing 12

```
public void ProcessAdditionRequest(Request
request)
{
  HandleRequest(request, () =>
  {
    request.Answer = request.Left + request.Right;
  });
}
private void HandleRequest(Request request,
Action action)
{
  try
  {
    using (Causality.Attach("Request",
           request.Id))
    {
      action();
    }
  }
  catch (MyException e)
  {
    // Recover from known problem
  }
  catch (Exception e)
  {
    // Recover from unknown problem
  }
}
```

Listing 13

Tag types

So far we've restricted ourselves to simple string based tags. But there is no reason why you couldn't store references to the actual business objects and use runtime type identification (RTTI) to acquire an interface for handling causality serialization and formatting. If all you're doing is rendering to a simple log file though this might be adding an extra responsibility to your domain types that you could do without.

This is one area where I've found Extension Methods in C# particularly useful because they only rely on the public state of an object and you can keep them with the infrastructure side of the codebase. The calling code can then look like this:

```
using (customer.TagCausality())
{
  // Do something with customer
}
```

The extension method can then hide the 'magic' tag key string:

```
public static class CustomerExtensions
{
  public Tag TagCausality(this Customer customer)
  {
    return Causality.Attach("Customer",
                            customer.Id);
  }
}
```

Keeping the noise down

Earlier I suggested that it's worth differentiating between the primary and ancillary tags to keep the noise level down in your logs as you traverse the various layers within the system. This could be achieved either by keeping the tags in a separate container which are then merged during formatting, or marking them with a special flag. The same suggestion applies to your context interface/façade - separate method names or an additional flag, e.g.

```
using (Causality.AttachPrimary("ID", Id))
```

versus…

```
using (Causality.Attach("ID", Id,
       Causality.Primary))
```

versus…

```
using (Causality.Attach("ID", Id))
using (Causality.MarkPrimary("ID"))
```

Whatever you decide it will probably be the choice that helps you keep the noise level down in your code too. Just as we wanted to keep the marshalling logic away from our business logic, we might also choose to keep our diagnostic code separate too. If you're using other tangential patterns, such as the Big Outer Try Block [Longshaw], or measuring everything you can afford to [Oldwood], you'll find weaving this aspect into your code as well might only succeed in helping you to further bury the functional part (see Listing 12).

Most of the boilerplate code can be parcelled up into a helper method that takes a delegate/lambda so that the underlying functionality shines through again, as in Listing 13.

Testing causality interactions

Due to the simplistic nature of the way the context is implemented it is an orthogonal concern to any business logic you might be testing. As the example implementation shows it is also entirely stateful and so there are no mocking concerns unless you want to explicitly test that the context itself is being correctly manipulated. Given that the modus operandi of the diagnostic context is to allow you to extract the tags for your own use the public API should already provide everything you need. This assumes of course that the operation you're invoking provides you with a "seam" [Feathers] through which you can observe the effects (for example, see Listing 14).

Summary

This article demonstrated a fairly unobtrusive way to associate arbitrary tags with a logical thread of execution to aid in the diagnosis and support of system issues via log files. It then illustrated a mechanism to pass the primary tags to other threads and remote processes so that multiple distributed scopes could be related to one another. The latter part contained some ideas on ways to reduce the noise in the client code and finished with a brief comment on what effects the mechanism has on unit testing. ■

```
public class TestService : IService
{
  public void DoSomething()
  {
      m_causality =
         Causality.GetPrimaryCausality();
  }
  public
  List<KeyValuePair<string, string>> m_causality;
}
[Test]
public void RequestShouldSetPrimaryCausality()
{
  var service = new TestService();
  var request = new Request(service);
  request.ProcessIt();
  Assert.That(service.m_causality.Count,
              Is.EqualTo(1));
  Assert.That(service.m_causality[0].Key,
              Is.EqualTo("ID"));
  Assert.That(service.m_causality[0].Value,
              Is.EqualTo("1234"));
}
```

Listing 14

Acknowledgements

A large dollop of thanks goes to Tim Barrass for providing me with the impetus to write this up in the first place and for forgoing sleep to review it too. Steve Love also picked up many problems both big and small. Finally the *Overload* team (Frances Buontempo, Roger Orr and Matthew Jones) helped to add some spit and polish.

References

[Ewald] *Transactional COM+: Building Scalable Applications*, Tim Ewald

[Feathers] *Working Effectively With Legacy Code*, Michael Feathers

[Harrison] *Pattern Languages of Program Design 3*, edited by Robert C. Martin, Dirk Riehle and Frank Buschmann.

[Longshaw] 'The Generation, Management and Handling of Errors (Part 2)', Andy Longshaw and Eoin Woods, *Overload* 93

[Nygard] Michael Nygard, Release IT!

[OED] *Oxford English Dictionary*

[Oldwood] Chris Oldwood, Instrument Everything You Can Afford To, http://chrisoldwood.blogspot.co.uk/2012/10/instrument-everything-you-can-afford-to.html

ACCU Conference Auction for Bletchley Park

We are very pleased to announce that the ACCU 2013 Conference will be continuing the now-traditional Charity Auction at this year's Speakers Dinner. While this is a bit of fun, it also raises vital funds for both The Bletchley Park Trust and The National Museum of Computing, helping them to continue to preserve our computing heritage. The Charity Auction can also be a fabulous opportunity to snag a piece of computing history for yourself!

The auction lots are not yet finalised but we do have a partial list. Take a look, have a think, then come to the Speakers Dinner ready to bid. After all, it *is* for charity!

If you can't be there in person, contact accu@archer-yates.co.uk to place a bid by email.

Lot 1 Bletchley Park family tour, and undercover look at Bletchley Park.

Lot 2 Hotel voucher (by Marriots).

Lot 3 Bombe Team tour, provisionally parrt of the day with a Bombe operator from the 1940s and a working operator in 2013.

Lot 4 Colossus Team tour of the Bletchley site culminating in a visit to the first semi-programmable electric computer and the new Computer Museum.

Lot 5 Colossus valve as used in Colossus mk1 & mk11

Lot 6 Group tour of Bletchley Park. A full day tour, with tea and coffee on arrival, and possible chat with the director prior to departure. Voucher valid for 14 months.

Lot 7 Amateur Radio Society of Bletchley Park day, with separate day in the National Radio Museum with Morse Code operators, talking to the world. An option on Saturday/Sunday for an afternoon tour of the listening/ wireless exhibition, and time with a lively retired MI8 agent (shhhh, it's secret... but she is willing).

Lot 8 Bletchley Park family pack, which includes unlimited visits for 1 year (including free car parking) and a Bletchley Secret Book.

Lot 11 Sculpture – by Steven Kettle.

Lot 13. At Bletchley Park stand from 10–12 April, this is the **star lot** of an ENIGMA machine replica. In kit form this costs £450, but this

assembled item has a reserve of £1000. The box it is housed in makes an amazing piece of furniture. It measures approximately 225mm × 300mm × 120mm deep.

Lot 16. Final clearance of Bletcheley Park papers. Due to a rebrand, the Bletchley Park retail manager is selling off their own published specialist papers.There are some 22 papers, each with its own ISDN number, covering specialist subjects from 'shark' to 'Turing' to the first computer, 'Colossus'.

Executable Documentation Doesn't Have To Slow You Down

Comprehensibility of end-to-end scenarios and quick feedback of unit tests are competing goals. Seb Rose introduces Cucumber with tags to meet both needs.

I was talking to a senior developer at a conference recently who was very supportive of TDD. However, he couldn't see why anyone would want to automate examples that were written in a way that was understandable by business folk – analysts, customers, product owners and so on. His experience was that the business people never participated in creating the examples and weren't interested in looking at them once they were automated. He also believed that the examples were frequently duplicated in unit tests and that since the unit tests ran so much faster, there was nothing to be gained from automating at the business level. In this article I want to examine this position and try to describe a way out of the impasse.

Business engagement

First, we need to address the lack of engagement of the business. The regular complaint is that the business people are too busy to spend time working on the examples with the development team, and the only way round this is to point out that unless someone who understands the business's needs is involved in the product evolution, then it's unlikely that the end result is going to be satisfactory. There are lots of ways to arrange business involvement, from having a fully empowered Product Owner co-located with the development team, through to scheduling regular times that a business person will be available to work with the team. The key point is that whoever works with the team needs to be authorised to make decisions regarding the development of the product – if all questions lead to an "I'll get back to you on that" then there will be problems.

A less satisfactory solution is for the development team to produce all the examples and then have them reviewed and signed-off. Many of the benefits of deriving the examples collaboratively are lost if you work this way:

- Deliberate discovery [Keogh12] – by having representatives of diverse stakeholder communities working collaboratively you can efficiently drive out more hidden issues than having them work separately. This harnesses the observed 'wisdom of crowds' phenomenon.
- Ubiquitous language [Fowler] – communication is about shared understanding. It's all too common for a term to have subtly different meanings to different communities, and the most effective way to combat this is to have those communities develop the organisation's vocabulary together.
- Immediate feedback – any process that involves hand offs between disparate groups is introducing delays, which affect the ability of the organisation to deliver value and respond to change.

Notice that I'm not talking about automation at all. No tools have been mentioned. Nor have I introduced any of the techy acronyms (BDD [North], ATDD [Hendrickson], SbE [Adzic11], EDD, TDD). I'm talking about collaboration pure and simple. This style of collaboration has been popularised by the Agile Manifesto [Agile], one of whose principles states:

> The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

You don't have to be agile to work like this (most organisations that think they are 'agile' don't work like this), but working like this will improve communication in any organisation and remove many of the hurdles from the real job of delivering something that the customer wants.

Automated examples

At this point you can decide to go no further, but there is great value to be had by considering automating some (or all) of the examples that were collaboratively authored:

- Living documentation – consider how often you have seen documentation that is out of date. By writing examples that demonstrate the behaviour of the system by actually executing against it, your documentation will always be up to date (as long as the examples all pass).
- Regression pack – since the examples describe the behaviour of the system, if that behaviour changes unexpectedly then some examples will fail.
- Feedback speed – because the examples run automatically against the system, they can be run frequently. The limiting factor for feedback speed is how quickly they run, but this will always be quicker than running an equivalent suite of manual tests.

Once you have decided to automate, you'll need to choose your approach. For the purposes of this article I will use examples written in Gherkin [Cucumber] (which interprets examples for Cucumber [Cucumber]). I am a fan of Cucumber/Gherkin, but there are many other tools available, most of which will automate the execution of examples.

Let's assume we have the example below, that deals with registering at some website:

```
Feature: Sign Up
    Scenario: New user redirected to their own page
        When I sign up for a new account
        Then I should be taken to my feeds page
        And I should see a greeting message
```

When Cucumber runs this scenario (example) it tries to match each step (introduced by the keywords Given, When, Then, And, But) with some glue code that exercises the system under test. The glue code can be written in various languages, depending on which port of Cucumber you are using, but for the purposes of this article I will limit myself to Java and so will be using Cucumber-JVM [Hellesøy12]. Each method in the glue code is annotated with a regular expression, against which Cucumber tries to match the text of each step:

- if there is no match, Cucumber generates an error
- if there are multiple matches Cucumber generates an error

Seb Rose is an independent software developer, trainer and coach based in the UK. He specialises in working with teams adopting and refining their agile practices, with a particular focus on delivering software through the use of examples. He can be contacted at seb@claysnow.co.uk

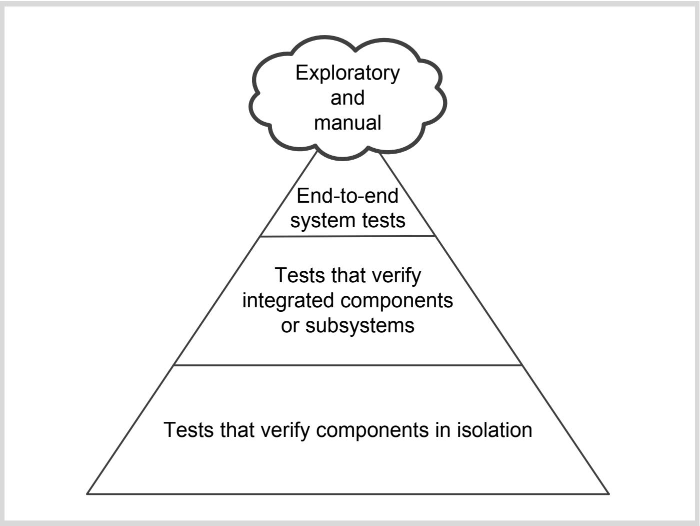

Exploratory and manual
End-to-end system tests
Tests that verify integrated components or subsystems
Tests that verify components in isolation

Figure 1

- if there is exactly one match, Cucumber executes the method in the glue code
  - if the method returns without raising an exception the step passes
  - if an exception propagates out of the method the step fails

How you implement the step definitions is up to you, and depends on the nature of your system. The example above might:

- fire up a browser, navigate to the relevant URL, enter specific data, click the submit button and check the contents of the page that the browser is redirected to
- call a method on a Registration object and check that the expected event is fired, with the correct textual payload
- or anything else that makes sense (e.g. using smart card authentication or retina scanning).

The point is that the text in the example describes the behaviour, while the step definitions (the glue code) specify how to exercise the system. An example glue method would be:

```
@When("I sign up for a new account")
public void I_sign_up_for_new_account() {
// Do whatever it takes to sign up for a new account
}
```

Examples everywhere

Newcomers to this style of working often adopt a style in which every example is executed as an end-to-end test. End-to-end tests mimic the behaviour of the entire system and create an example's context by interacting directly with the UI, and the full application stack is involved throughout (databases, app servers etc.). This sort of test is very useful for verifying that an application has deployed correctly, but can become quite a bottleneck if you use it for validating every behaviour of the system. The Testing Pyramid (see Figure 1) [Fowler12] was created to give a visual hint about the relative number of 'thick' end-to-end tests and 'thin' unit tests. In the middle are the component/integration tests that verify interactions within a subset of the entire system. (The 'Exploratory and Manual' cloud at the top is a reminder that not all tests can be automated, and that the amount of effort needed here is very system dependent.)

It may be reasonable to use the example scenario above as a 'Happy Path' end-to-end test, demonstrating that the whole application is hanging together. However, there are some other situations that emerged when this feature was discussed, some of which were:

- what happens if the user already exists?
- what happens if the credentials provided are unacceptable?
- how will errors be communicated to the user?

These questions are still independent of how the system is actually going to be implemented, but we can start fleshing out some examples:

```
Scenario: Duplicate user registration
    Given I already have an account
    When I sign up for a new account
    Then I should see the “User already exists” error message

Scenario: Unacceptable credentials at signup
    Given my credentials are unacceptable
    When I sign up for a new account
    Then I should see the “Unacceptable credentials” error message
```

These extra examples could be implemented using the whole application stack, but then the runtime of the example suite begins to rise as we execute more end-to-end tests. Instead, we could decompose these examples into:

- examples that demonstrate the correct feedback is given to the user in various circumstances
- examples that exercise the validation components

```
Scenario Outline: Display correct error message
    When the registration component returns an <error>
    Then the correct <message> should be returned

Examples:
| error                                 | message                    |
| error-code-user-already-exists        | "User already exists"      |
| error-code-unacceptable-credentials   | "Unacceptable credentials" |

Scenario: Detect duplicate user
    Given user already exists
    When the registration component tries to create the user
    Then it will return error-code-user-already-exists

Scenario: Unacceptable credentials at signup
    Given the credentials are unacceptable
    When the registration component tries to create the user
    Then it will return error-code-unacceptable-credentials
```

Speed, completeness and comprehensibility

These examples should run a lot faster, but are no longer written in business language (if you want an explanation of Scenario Outline look at the Cucumber documentation). They have lost some of their benefit and have become technical tests, mainly of interest to the development team. If we choose to 'push them down' into the unit test suite, where they seem to belong, then we will have lost some important documentation that is important to the business stakeholders.

This demonstrates the conflict between keeping the examples in a form that is consumable by non-technical team members and managing the runtime of the executable examples. Teams that have ignored this issue and allowed their example suite to grow have seen runtimes that are counted in hours rather than minutes. Clearly this limits how quickly feedback can be obtained, and has led teams to try different solution approaches, none of which are ideal:

- partition the example suite and only run small subsets regularly
- speed up execution through improved hardware or parallel execution
- push some tests into the technical (unit test) suite

In a recent blog post I introduced the Testing Iceberg (see Figure 2) [Rose], which takes the traditional Testing Pyramid and introduces a readability waterline. This graphically shows that some technical tests can be made visible to the business, while there are some end-to-end tests that the business are not interested in. We want to implement our business examples in such a way that they:

- document everything relevant to the business
- do not duplicate technical tests
- minimise the execution time of the examples

Using Cucumber

There are a few features of Cucumber that I need to introduce before describing the technique I use to keep my examples consumable by the business without sacrificing performance of the suite.

Tags

Any Cucumber scenario can have one or more free text tags applied to it:

```
@this_is_a_tag
@a_different_tag
@regression
Scenario: What just happened?
    When I do something
    Then something should happen
```

When invoking Cucumber you can pass in tags as arguments to identify which scenarios should be executed. This is useful when trying to partition the example suite, to build a regression suite or a smoke test suite, for example.

Hooks

Cucumber also allows you to provide setup/teardown hooks in your glue code that are run before and after each example.

```
@Before
public void beforeScenario() {
  // Do some setup work
}
```

Tagged hooks

And finally, Cucumber allows you to write tagged hooks, which are only run before scenarios that have matching tags (matching can use complex logic – see the documentation).

```
@Before("@regression")
public void beforeScenario() {
  // Do something specific to the "regression" tag.
}
```

Putting it all together

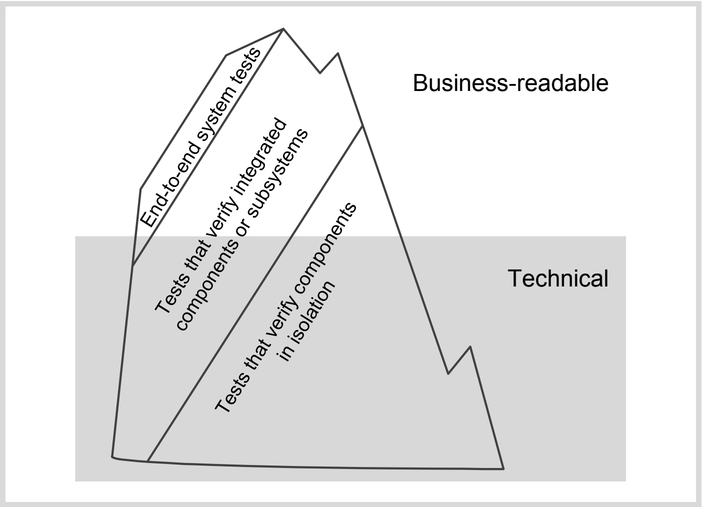
```
#sign_up.feature
@without_ui

Scenario: Duplicate user registration
    Given I already have an account
    When I sign up for a new account
    Then I should see the "User already exists" error message
```

```
// RegistrationSteps.java
class RegistrationSteps {
  private boolean without_ui = false;
  @Before("@without_ui")
  public void beforeScenario() {
    without_ui = true;
  }
  @When("I sign up for a new account")
  public void I_sign_up_for_new_account() {
    if (without_ui){
      // Send information directly to
      // registration component
    } else {
    // Drive UI directly using Selenium [Selenium]
    // or similar.
    }
  }
}
```

The benefits of working like this are:

- we can write our examples from a user perspective (which makes it easy for the business to understand)



End-to-end system tests
Tests that verify integrated components or subsystems
Tests that verify components in isolation
Business-readable
Technical

Figure 2

- we can execute the examples as thinner component or unit style tests (which keeps the runtime down)
- we can avoid duplication by using the glue to delegate directly to the unit tests where appropriate
- we can run the examples using the whole application stack and begin to thin down the stack using tags once we have built some trust in our initial implementation.

It is the business who should prioritise how to evolve a product, based on their understanding of the customers needs. Face to face communication between the business and the development team can help develop a ubiquitous language that can be used to document the behaviour of the system in a manner that is clear and unambiguous to all concerned. The examples that are produced during these conversations can then be automated, but there is an ongoing tension between the comprehensibility of end-to-end scenarios and the quick feedback of unit tests. Using Cucumber and tags it is possible to write the examples in an end-to-end style, but modify how they are executed (and hence their runtime costs) by applying or removing tags, without adversely affecting the comprehensibility of the example itself. ■

References

[Adzic11] http://specificationbyexample.com/key_ideas.html

[Agile] http://agilemanifesto.org

[Cucumber] http://cukes.info

[Fowler] http://martinfowler.com/bliki/UbiquitousLanguage.html

[Fowler12] http://martinfowler.com/bliki/TestPyramid.html

[Hellesøy12] http://aslakhellesoy.com/post/20006051268/cucumber-jvm-1-0-0

[Hendrickson] http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview/

[Keogh12] http://lizkeogh.com/2012/06/01/bdd-in-the-large/

[North] http://dannorth.net/introducing-bdd/

[Rose] http://claysnow.co.uk/?p=175315341

[Selenium] http://docs.seleniumhq.org

Why Dysfunctional Programming Matters

Function progamming is all the rage. Teedy Deigh considers how it offers many opportunities for the serious programmer.

Functional programming has, of late, taken its flight path across the radar of the industry, awaking from its paradigmatic slumber to threaten the livelihoods and codebases of hordes of OO half adopters and procedural laggards. How far should it be followed to keep one's job and reputation secure? What opportunities does it offer for programmer idiosyncrasy and awkwardness? This is clearly a matter that warrants further study.

As with any serious research effort, the dictionary is our first stop. It offers the following possibilities:

functional, *adjective*

- having a function
- working or operating
- useful, utilitarian
- designed to be practical rather than attractive.

For the paid programmer this sends out a mixed message. The first point is certainly met by much code. Indeed, there is an implication that one function is sufficient, thus cramming it all into **`main`** should be enough to meet this requirement. The outlook is good for many programmers who rely on such extreme encapsulation techniques as a means to differentiate their code from so-called clean code. But the second definition is not so welcoming for anyone whose expertise and day-to-day practices and heroism find expression in bugs and the debugger. Similarly, the joy of much programming is in considering it an art rather than some demeaning utilitarian endeavour, where *stakeholder value* is the only dull reward. The last definition offers mixed possibilities: that the code need not be aesthetically pleasing can be considered a good thing; being practical misses the point of much coding effort and advice. For example, a large part of the investment in OO is based on the premise of reuse rather than use, which is the perfect get-out for anything that may be criticised as not at first appearing useful.

There is a certain air of elitism that surrounds functional programming, which for some programmers allows them to retain a mystique and priest-like status. This is to be applauded, especially if the code is similarly shrouded in mystery. Yet at the same time there is something more hoi polloi about functional programming that threatens to wrest programming from the grasp of the few and deliver into the hands of the many:

Excel is the world's most popular functional language.
~ Simon Peyton Jones

It is important to remember that for a programmer:

The needs of the one outweigh the needs of the many.
~ James Tiberius Kirk

Thus some kind of distance from the common world of Excel, any kind of secret knowledge or obscure technique that can be brought into play, is needed if functional programming is to be taken seriously by real programmers.

Fortunately, the close association between mathematics and functional programming looks set to provide such separation. For those less comfortable with mathematics, but who revel and dwell in the darker corners of procedural coding, there is also hope:

Haskell is, first and foremost, a functional *language. Nevertheless, I think that it is also the world's most beautiful* imperative *language.*
~ Simon Peyton Jones

While beauty may not be a selling point – the very word *functional* suggests this is not a credible consideration – the imperative opportunity is in the imperative support:

The determined Real Programmer can write FORTRAN programs in any language.
~ Ed Post

Given that functional programming can be coaxed into something more familiar, it makes sense to probe a little further. And how better to understand FP than to appreciate the problems for which it is ideally suited? For example, OO programmers are drawn to bank accounts and stacks, enterprise architectures meet the needs of pet stores everywhere and TDD satisfies a collective need to understand Roman numerals and the rules of ten pin bowling. What then is the killer app for functional programming? There are many, but one that deserves our special attention is the factorial function – a pressing need for which seems to exist in teaching texts everywhere.

Where a modern C programmer might be satisfied with the following, with its exemplary use of postfix decrement, discreet use of the ternary operator and nod to design by contract (along with the "To **`NDEBUG`** or not to **`NDEBUG`**?" question it leaves in its wake):

```
int factorial(int n)
{
  assert(n >= 0);
  int result = n ? n : 1;
  while(n-- > 1)
      result *= n;
  return result;
}
```

The true functional programmer understands that:

To iterate is human, to recurse divine.
~ L Peter Deutsch

Teedy Deight dabbles in programming language design in the way a cat dabbles with a trapped mouse. You never know the details, but the outcome is rarely good for the mouse. The remains will turn up and surprise you where and when you're least expecting it, leaving you wondering how to dispose of it.

which gives us:

```
int factorial(int n)
{
  assert(n >= 0);
  if(n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

To achieve full divinity, however, requires the following:

```
int factorial(int n)
{
  assert(n >= 0);
  return n == 0 ? 1 : n * factorial(n - 1);
}
```

Enough to make many newbie programmers and maintainers mutter "God" under their breath. Their awe mingled with a lack of appreciation of the qualities the conditional operator can bring to a codebase when employed extensively and without mercy. It is worth noting that conditional expressions are the norm in functional programming, although they often lack the brevity of C's ternary operator.

That said, however, it is considered good functional style to express programs as transformations expressed through other functions, often those found in a standard library. While any good programmer makes some use of libraries, there is always a balance to be struck, always a sense that it is the programmer who should be writing the reusable code rather than actually reusing it.

The following version of factorial is implemented in Haskell, a pure functional language whose name is the outcome of a word association game where the chain of connections took in one of the fundamental programmer food types and the logician Haskell Curry:

```
fact n = product [1..n]
```

The definition of the **`fact`** function is brief and clear, with clarity being perhaps the major objection to adopting this approach in legacy-wannabe code. That said, the brevity of identifiers popular in functional programming is a welcome relief to anyone whose fingers have laboured over the enterprisey **`FactorialCalculationFunction`** (with a further liberal assortment of **`Manager`**, **`Controller`**, etc. suffixes waiting in line for inclusion), but at the same time offers sufficient scope for challenge and humour. Instead of the painfully obvious **`factorial`**, programmers can choose from:

- the boldly assertive **`fact`**;
- the chatty **`fac`**;
- the wry **`faq`** – factorial is, after all one of the most frequently asked-for programming examples;
- the vowel-deprived **`fct`** – a puzzle for its readers to solve or an opportunity for vowel-injected humour;
- the enigmatic discretion of **`f`**.

It is worth learning the lessons and orthodoxy of the Haskell version, while keeping in mind the following:

The code is more what you'd call guidelines *than actual rules.*
~ Captain Hector Barbossa

Thus, the optimal solution for factorial is one that combines considered naming with artisan-crafted logic.

Turning to larger programming problems, one of the major challenges – and therefore one of the major opportunities for programmers to inject themselves into projects as dependencies – is the question of state change.

In general, functional programming shies away from state change. It is easy to see that if state change is ignored, code achieves many non-functional qualities – interactions with databases don't function, user interaction doesn't function, etc. Indeed, any I/O or interaction with the physical world become completely non-functional. Sadly, for all the purity that this offers and peace that it brings – databases and users being a prime source of annoyance and bugs – a complete lack of state change is likely to attract few sponsors. To get around this, functional programming languages generally adopt one of three approaches – pragmatism, actors or monads:

- Pragmatism is often another way of saying, "We know this could be better, but that looks a bit tricky and, quite frankly, we can't be bothered." When used as a prefix, *pragmatic* is often shorthand for *not*, as illustrated by, for example, pragmatic Agile, pragmatic TDD, pragmatic OOP. Thus the question of functional I/O can addressed with pragmatic side-effect-free code.
- The actor model of computation is based on role playing and pretence. Instead of actually doing the I/O yourself, you employ an actor to do it for you. Actors pretend they're not doing anything – as the name suggests, it's all an act – but they're really doing I/O, and probably quite a lot of it. In common with their high-profile real-world namesakes, if you ask actors if they're doing anything questionable they'll deny it, which spawns a whole network of gossip (i.e., messages passed discreetly from actor to agent to journalist).
- It is perhaps fitting that monad rhymes with gonad, reflecting the common response to the masterful way in which I/O with monads is passed off as being free of side-effects by burying it so deeply in category theory – wrapped in a mystery inside an enigma spliced in a riddle encrypted with AES – that it becomes almost impossible to determine whether there is any I/O. Simply trying to work out what we mean by 'I/O' and what we mean by 'what we mean by' is usually enough to distract practitioners and theoreticians alike from any I/O that may (or may not) be happening, while also noting that *maybe* is itself a monad.

Thus, functional languages largely manage to achieve state change through a process of full-scale regime change.

As in other walks of programming, FP is unsettled on certain key issues, such as whether to favour dynamic typing or static typing, so the programmer can feel right at home and just as entrenched. Matters of syntax also offer scope for heated and vibrant discussion, with handwaving use of *elegance* often used to sweep under the carpet the frequent similarity between functional programs and `/etc/termcap` files. Hybrid languages provide a certain postmodernism relief to the clean lines of much neoclassical or modernist functional programming.

Although the claims of clarity, brevity and purity are enough to put many programmers off, we can see now that functional programming offers many opportunities for the serious programmer. ■