overload 98

AUGUST 2010 £3

Renovating a Legacy C++ Project

We investigate how a software team tamed the beast within a sprawling codebase

I'm a Business Analyst, Get Me Out of Here

How the Business Analyst role fits into an agile development process

Exceptions: The Worst Form of "Error" Handling

...except for all the others. A controversial look at error handling.

Debugging Run Time Memory Problems

We take a look at the consequences of "undefined behaviour", and how to take advantage of it to diagnose programming problems as early as possible

A magazine of ACCU

ISSN 1354-3172

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

OVERLOAD 98

August 2010

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

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

Copy deadlines

All articles intended for publication in Overload 99 should be submitted by 1st September 2010 and for Overload 100 by 1st November 2010.

ACCU

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

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

Copyrights and Trade Marks

A Little Learning Is A Dangerous Thing

An opportunity for nostalgia triggers some musings on the value of education. Ric Parkin goes back to school.

Roughly every decade, two years worth of students get invited back to my college, involving afternoon tea, a service in the chapel, and a drinks reception followed by a formal dinner in hall. In a few days time it's my year's turn, and it looks to be a good turnout, with around 130 out of around 200 possible signed up. (I know this because there's now a web page showing who's coming – a sure sign of the ubiquity of the internet!). The last one was great fun, and was fascinating seeing how people had changed (not much), and catching up with home and work life. This year won't be quite as surprising as many people are in touch via LinkedIn or Facebook (which now has an amazing 26m UK users [BBC], that's around 40% of the population). But anticipating seeing everyone (and rehearsing my answers to the inevitable 'what are you doing now' questions) got me thinking about how well my education had prepared me for my working life.

The first thing is to get some background about the computer industry, and education. I'd always been brought up with computers in the background, as my father worked as a computer engineer and later a trainer. Visiting the training facilities on the way home from school led to my first programming experience when I was around 10–11 using teletypes and VDUs attached to large VAX clusters. This interest led to me getting one of the first ZX Spectrums as part of the exploding home computing craze. Sure, I played lots of games, but I also did a lot of programming, not just using the built-in BASIC, but also using assembler, C, Forth and Pascal. I eventually sold it just before the peak, and didn't get another computer of my own until around 2000.

So how was I using computers during my education? Well, the answer is 'hardly at all'. When I was choosing which O levels to do, I was advised that I'd be able to pass Computing tomorrow, so I didn't take it. I did use a bit for my Control Technology project, which used a computer controlled gadget to read multiple choice question papers (amusingly on the morning of the assessment one of the photo receptors failed, but realising that using my test sheets I could infer its value from the others, I managed to tweak the program to work correctly. An early lesson in data redundancy and error correction). For my A levels we didn't need computers at all. And in my Maths degree, we had a couple of projects using BBC micros to solve Schrodinger's equation, and do numerical analysis finding iterative solutions keeping track of errors. While I did some really nice user interfaces and code structuring, the projects were marked for the mathematical conclusions you came to rather than the code. And that was it! The main things I took away were being able to think logically, and some idea of how to understand orders of magnitude (useful when it came to understanding the STL's complexity guarantees) and estimate numerical errors.

Thankfully I was getting some proper experience elsewhere – to top up my finances I'd got a sponsorship from the Ministry of Defence, and spent the summers in Malvern at the Royal Signals and Radar Establishment (now part of Qinetiq) doing statistical analysis of aircraft trajectories to better understand the consequences of a collision avoidance system [TCAS]. This was on VAXes using Pascal and various maths libraries, and LaTEX for creating the reports, and I learnt a lot about optimisation and code structuring. It was probably this experience that allowed me to get my first job in software.

So would I have been better prepared if I'd taken a Computing degree? Very likely, although at the time the course was much more theoretical as it was harder to get access to computing resources. Also, the whole idea of Software Engineering was relatively young. Nowadays it's pretty much a given that everyone has their own laptop, so courses are more hands-on and relevant (as an example, here's Cambridge's lecture and project list [Cambridge]), although it's still hard for people to get 'real-world' experience on projects of a decent size. This is inevitable and must be taken into account when hiring graduates, making sure you understand the strengths and weaknesses (basically, they've done a bit of everything but not much depth, and will not have much idea of the software life cycle nor projects of much size) so you can design their early projects and experiences to get them up to speed, perhaps using some mentoring/apprentice model.

So what sort of education, formal or otherwise, have I done over the years while I've been working? I've never had any formal training that lead to a qualification I could put on a CV. Some do exist, and some sectors are keener on seeing things like that than others. One of the problems is that computing evolves so quickly that putting together a formal syllabus can lead to it being out of date almost immediately. Quite often such qualifications are pushed by the larger companies as a way of promoting their technologies, such as MSCE [Microsoft] or SCP [Sun]. These tend to be more up to date, and can be useful if you need to specialise in that area. I have attended some training courses, and you do get some sort of certificates for attending, or even taking some sort of exam, but I'm a bit more dubious about the worth of many of them – a few days study will get you up to speed on the basics, but doesn't tell you much about how well you can put things into practice, and whether you've progressed to becoming proficient, or even expert. Those things takes time, experience, learning from the experiences of others.

It is this last area where organisations like ACCU really shine. By bringing together a disparate group of people all eager to learn you can get great insights that would have taken you years to have had (if at all). Even if it's just planting the seed of an idea, when a future problem needs tackling quite often these ideas will pop up and you have a possible avenue to

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

explore. Quite often though you learn something of immediate use, which will improve your ability to produce good quality and value software.

Of course this is just my experience, and I'm very aware that my career is very different from other people's. In particular I've tended to mainly work for small companies, often start-ups with small customer bases and a small amount of largely new code that has to be produced quickly without letting quality fall. In such an environment broad experience and adaptability tend to be prized. It would not surprise me that in other situations, perhaps something like financial modelling for a large bank, then in that case long term qualifications showing real depth in a specialized area would be favoured instead. And yet both extremes still have in common that continuously learning, whether via books, magazines, conferences, or formal courses, is the only way to improve and keep your skills current.

The wisdom of crowds?

A few issues ago I mentioned this book by James Surowiecki [Wisdom], which discusses how large groups of people can be better at certain tasks than individuals, even experts. He wisely also discusses when such ideas are not applicable, and can lead to sub-optimal solutions. Interestingly there have been some great examples recently, which quite often rely on computing, in particular social media. One odd one was BP asking for ideas to help cap the deep water oil leak in the Gulf. However, this wasn't as crazy an idea as it first seemed, as they were dealing with a problem that no one had had to face before, so casting around for ideas could inspire alternative approaches in case the main approaches failed. The main problem with it was more political – it makes it look like they didn't have any ideas of their own, and are desperate.

This is also a risk with one of the more recent ones launched by the new government for people to suggest which laws could be repealed [YourFreedom], and areas for spending cuts [SpendingChallenge]. A neat idea in theory, not that dissimilar to an online suggestions box, but the way it was implemented as a Web 2.0-style social forum meant that it tended to be the loudest with a grudge to dominate the discussions and suggestions, which has the danger of drowning out the more interesting ideas. This is almost inevitable – by making it so open, public, and interactive (for perfectly laudable reasons), it breaks some of the criteria to get a Wise Crowd. Hopefully someone will be taking the time to sift through all the ideas to find the interesting unexpected ones, rather than the obvious ones with a large populist backing. It does show that social webs are excellent at allowing people to seek out and interact with like minded people, but are not as good at getting a balance of opinion as they are all too easily susceptible to group-think and self-selection. While such uses of technology can be powerful, you do need to understand whether the dynamics of the resulting system match your needs.

References

[BBC] http://www.bbc.co.uk/news/technology-10713199

[Cambridge] http://www.cl.cam.ac.uk/teaching/0910/CST/

[Microsoft] http://www.microsoft.com/learning/en/us/certification/mcse.aspx

[SpendingChallenge] http://spendingchallenge.hm-treasury.gov.uk/

[Sun] http://en.wikipedia.org/wiki/Sun_Certified_Professional

[TCAS] http://en.wikipedia.org/wiki/Traffic_collision_avoidance_system

[Wisdom] http://en.wikipedia.org/wiki/The_Wisdom_of_Crowds

[YourFreedom] http://yourfreedom.hmg.gov.uk/

"I'm a Business Analyst – Get Me Out Of Here"

Some classic roles are omitted by Agile methodologies. Allan Kelly considers how a Business Analyst fits in.

Over the last couple of years I have noticed that there is a genuine desire in the Business Analysis community to know more about how Business Analysis and the BA role fit into Agile software development.

This is quite natural. Agile software development is changing the face of software development and BAs want to be part of the change while fulfilling their responsibilities to the best of their ability. While there is a lot of literature on the role of Software Developers on Agile teams the same is not true of the BA role.

This article will attempt to answer two questions:

- What is the role of a BA on an Agile software team? and,
- How does the BA role change between traditional (so called 'waterfall') development and Agile?

In the process I will also explain why there is more work for a BA to do on an Agile team than on a traditional team.

Naming the role

The first thing to point out is that Business Analysts do not exist in every organization. While they are common in corporate IT departments and external service providers (ESPs) they are usually absent from product development organizations. Instead of BAs companies like Adobe, Oracle and Autodesk have Product Managers or Technical Product Managers.

The Product Manager role is a first cousin of the BA and like the BA they are concerned with determining the needs the software is attempting to satisfy. Product Managers employ many of the same tools as the BA and at the heart of both roles is: analysis.

However the roles are different in one important aspect: while the BA role is inwardly focused (within the corporation, within the department or within the client company) the Product Manager role is externally focused. Figure 1 illustrates the general position.

For a BA the end product, that is the software, will be used by the people within the organization who have little choice over what they use. Conversely, Product Managers deals with customers who have a choice: they can buy the software or not, they can buy elsewhere.

This distinction is important to Agile working for two reasons. Firstly, as we will show when discussing changing to the BA role; BAs needs need develop the same commercial awareness as Product Managers.

Allan Kelly has held just about every job in the software world. Today he provides training and coaching to teams and companies in the use of Agile and Lean techniques to develop better software with better processes. He is the author of *Changing Software Development: Learning to become Agile*, numerous journal articles and is currently working on a book of Business Strategy Patterns. Contact him at http://www.allankelly.net.

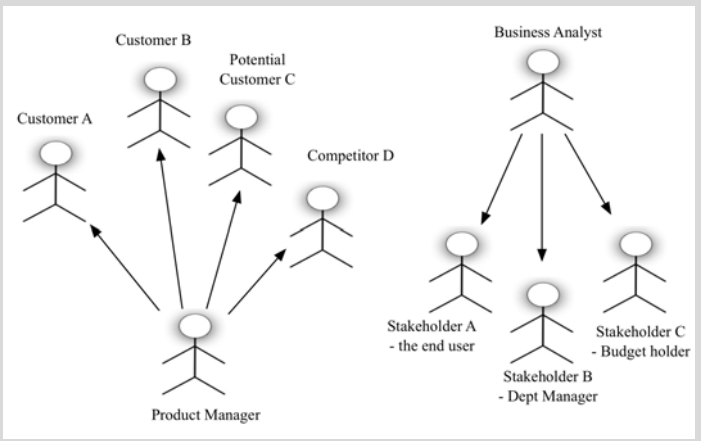


Figure 1

Secondly, the most popular Agile method, Scrum, contains a role called Product Owner which is largely based on the Product Manager role. Thus it is easier for Product Managers to understand their role on an Agile team than it is for BAs to.

Part of the confusion surrounding the BA role on Agile teams stems from the fact that neither Scrum, nor the other widely known Agile method, Extreme Programming (XP) describe a BA role. Both Scrum and XP place great emphasis on development teams working as closely as possible with someone who actually wants the final product. XP calls this person the 'customer' and Scrum, as already mentioned, calls it the 'product owner'.

In fact, the first XP project (the Chrysler C3 development) had a BA filling the customer role. Although the books and descriptions of the project call this role a 'customer' the two people who filled the role were BAs.

It is not always practical, or desirable, to have an actual customer work with a development team. Instead a proxy needs to play that role. Indeed, it is wrong to assume there is a single customer. If there is to be a single customer voice someone needs to amalgamate multiple voices.

This is where the BA fits in. The BA is a customer proxy. The BA is the person who listens to multiple 'customers' and speaks with a single voice to the team.

One can think of the term Product Owner as an alias used in Agile literature to mean 'the person who represents the customers needs'. In a software product company there is a Product Manager behind the alias, and in a corporate IT environment there is a BA behind the alias. (Unless otherwise stated from here on I assume the Product Owner role is filled by a BA.)

Agile is not a management free zone

To date most Agile methods have largely been developer centric. As a result the BA role has been underplayed. Adding to this neglect is the belief in some places that Agile does not require management. And since the BA role is normally a non-coding role it is perceived as a management role by many developers.

Someone needs to decide what is most important, what compromises are possible, and what doesn't get done

The management free message contained in some Agile texts, and espoused by some Agile advocates, comes from a belief that developers know best and self-organizing teams are the most productive way to work. While there is some truth in both arguments there is still a need for management. Self-organizing teams do not come into being spontaneously.

Even on a self-organizing team it helps if someone is focused on the question of what the customer wants, their needs and where the greatest value is to be found. It makes sense for someone with business analyst's skills and background to take on this role. While they may take on some other work within the team (e.g. administration, testing) one should not fall into the trap that other work necessarily includes coding. Not everyone on the team will possess the necessary skills to write Java or Python. Indeed, not having these skills can be a useful barrier to prevent the 'all hands to the pump, we need more code!' mentality taking hold. Such a mentality drives out other work in a blaze of naivety.

Historically Agile has underplayed this role. While XP describes what developers do in detail and Scrum describes how the team works together to achieve the project no popular method describes the how the Product Owner fills their role.

To put is another way: think of an Agile team members as actors performing a play. There is a BA playing the role of Product Owner or Customer. The script (Scrum, XP or some other method) describes what to do when on the stage: work with the team, prioritise, etc. But it doesn't describe what the actor does off stage.

It is the offstage work that allows the Product Owner to fulfil their onstage duties. While the onstage role is similar for a Product Manager or a BA playing the Product Owner, the offstage role is very different. To date Agile methods have had little to say about these activities, leaving BA to decide for themselves what needs doing.

The problem with Customers

On the face of it the original XP model seems ideal: find out what you want from an actual customer. However there are a number of reasons why a customer proxy might be better than an actual customer.

Firstly customers may tend to see what is in front of them immediately rather than looking at the longer term or strategic objectives. In an iterative development model this may result in a stream of requests to change the appearance of the software, or add minor features rather than driving towards the ultimate goal. While useful, even valuable, this can lead to small changes, perhaps only cosmetic, which do not justify the cost of the work.

This is particularly true when the development is intended to introduce change into an organization. IT systems are rarely developed and deployed to support the status quo, instead they are created to improve processes, reduce costs, enter new markets and so on. Using an actual customer, or end user, may focus efforts on the current environment and not on change.

There is a balance to be struck here. The message of Agile methods, incremental change and continual improvement, is as applicable in the wider business environment as it is in the software development one. However there needs to be a change driver.

There is another, more commonly cited, problem with using actual customers to source requirements: time. Customers have their own jobs to do, they may not want to leave their job to work with a development team or the company may not be prepared to spare them for the task. This is particularly true in development teams that work with highly valuable, highly paid, individuals like financial traders or doctors. Getting access to such experts is nearly impossible.

All of this assumes there is a single customer who can be identified and who knows what is required. While this is true in some cases it is far from universal. Many internal projects struggle with multiple customers and stakeholders, each with different expectations and needs for the development. At best the stakeholders' needs are complementary and combined will produce a better system. At worst these needs and stakeholders are in competition and not compatible.

Given the focus on delivering business need there is a real need to evaluate requests, compare them and turn some down. Where one of these customer stakeholders is elevated to the position of 'the customer' there is a danger that their own priorities may take precedence over others regardless of the business value.

Frequently there is not one single customer but several end-users, and if we further expand the list of those interested in the work to stakeholders the possibility of alternative needs conflicting becomes an almost certainty. Someone needs to decide what is most important, what compromises are possible, and what doesn't get done.

Taken together it becomes clear that the single actual customer model is too simplistic for many environments. The net result is that it is often better to employ a proxy in the customer role than an actual customer.

In short there is more to fulfilling the Product Owner role than knowing what one wants oneself. Someone needs to understand the deeper needs, the motivation and goals of undertaking the work, who the interested parties are and what their interests are.

Just like the original

I was recently involved with a project which had been started with the brief 'Make it exactly like the original'. Obviously there was no need for a BA here, all the development team had to do was copy a working product.

The original was built using PowerBuilder and while it was operating fine the client didn't want to risk being dependent on a dated product. So a project was started to re-write it as a web based Java application.

The users were denied the option of changes that would benefit their work and consequently demanded 'exactly the same'. The developers were told to copy the original, as a result they produced a web application that looked and felt like a PowerBuilder one.

With no BA involved, and no sense of business value, this potentially win-win turned into a lose-lose. Rather than have the improved application they

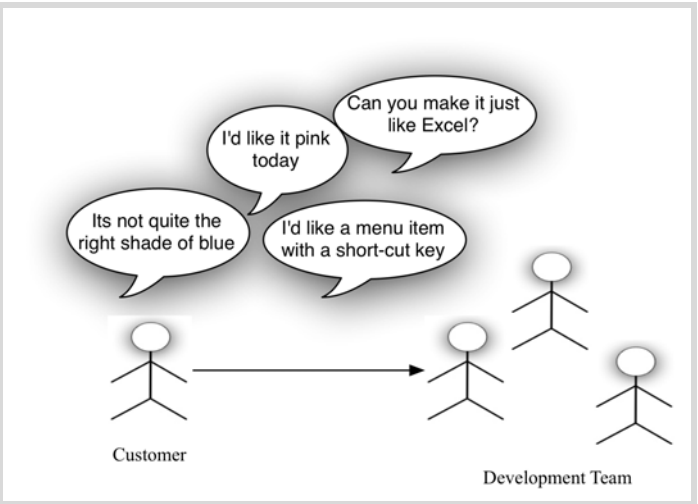


Figure 2

wanted the users were given an application which cost more to build than it needed to, probably with more features than they needed.

When you consider that as much as 80% of the features in bespoke software are not used, then in copying the original software the development team probably did five times more work than was actually required.

The problem(s) a BA can address

Many Agile adopters believe the best way to produce software a customer wants is to listen directly to the customer. This can work; this approach should not be dismissed, particularly if the customer is the person paying the bills.

However this approach is not guaranteed to bring success. There are a number of reasons why this approach either will not operate or will not result in the desired outcome. Consider for a moment a corporate situation where one set of 'users' will use the software created, but another set of customers, 'managers', are commissioning the software with view to changing how the users work, and a third set of customers are paying the bills. Who is the customer for the team?

Such situations where competing 'stakeholders' have different needs and desired outcomes for work is the bread-and-butter of business analysis and systems engineering. Of course the development team may nominate one of their own to understand these issues and make sense of the competing demands but in doing so they have created a business analysis role.

When this happens it is fair to ask: does the person filling the BA role have the right skills and experience? And: Is this an effective use of the skills and experience they do have? It seldom makes sense to stop a brilliant coder from coding and ask them to perform a role they are less effective at.

In order to understand where the BA fits on an Agile team it is worth considering a number of these scenarios that can, and do, occur.

Figure 2 illustrates the position in which the customer doesn't know what they want, perhaps they ask for trivial changes or oscillate between different requests. However, there is a more subtle but severe issue here. Namely the customer does not know what will result in the greatest value. Asking a development team to 'make it like' is a common way of describing a need but results in zero value to the business. If the customer wants a product 'like Excel' then why not have Excel?

Two problems will occur again and again: what should the team develop? And which requests will maximise business value?

Another common issue is demonstrated in Figure 3, that where there are multiple different 'customers' for a system each of whom has different requirements on the system and different expectations. Determining which requests will result in the greatest business value is even more complicated in these cases as each customer group will argue their own side.

Complicating matters further is the small matter of business strategy and company objectives. On occasions it may not be possible to demonstrate

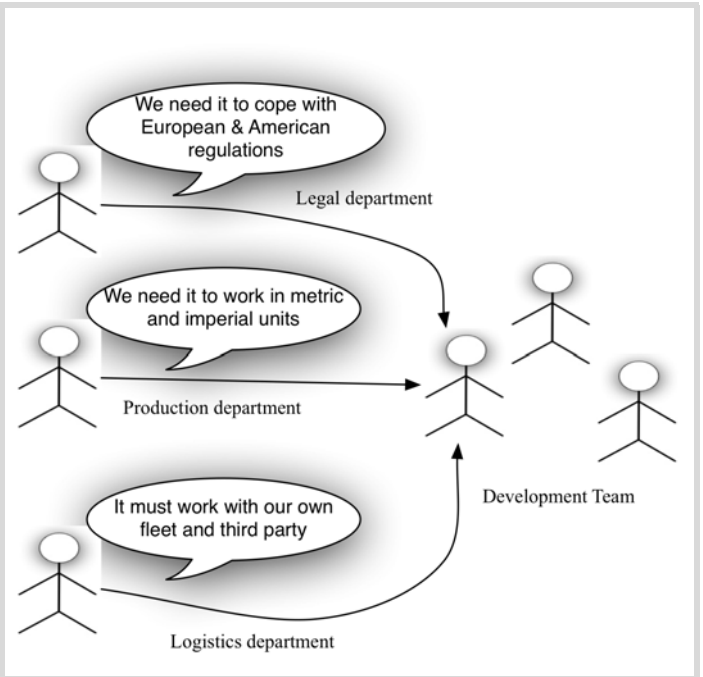


Figure 3

highest value for a particular feature but business strategy demands the feature.

For example, take the case illustrated; the software may be under development for a European company with no American operations. On the face of it there is no need to have the software work within US regulations. However, the board of the company may be looking to sell the company within the next two years. If they can interest American investors they may be able to extract a higher price from the buyers even if the final buyer is European.

This example starts to illuminate the difficulty in determining business value for IT developments. Research shows there is commonly a time lag between investment in IT and value returned [Brynjolfsson09]. Nor is this the only complicating factor. Without new processes, training and other changes the potential value delivered by an IT system may remain unrecognised. Consequently while a business unit may be able to demonstrate significant business value from new IT this value might not be recognisable if the unit concerned cannot make other changes.

Globalisation means that development teams are increasingly faced not only with multiple customers but also with customers spread

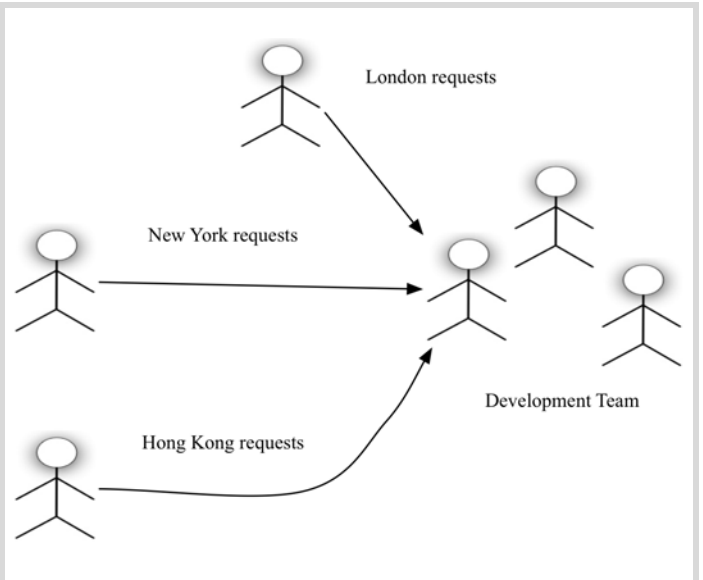


Figure 4

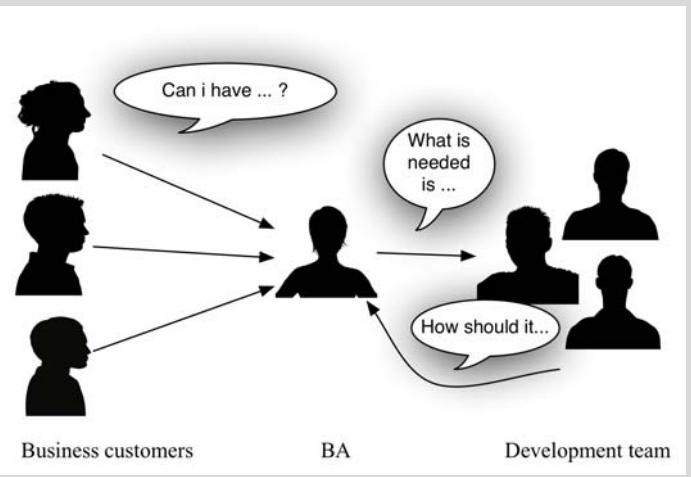


Figure 5

geographically. Determining which location gets what it wants, and which should be made to wait is no small matter. When a team is based in one location, say, London, requests from that location may win out over another, say, Hong Kong, not through logic but through acquaintance. The team and their customer eat in the same places, drink beer together and share the same space.

Not only do different geographic locations introduce subtle preferences and information asymmetries, they can slow down work. In a perfect world a question asked by the London development team could be answered by New York and Hong Kong overnight and the answer be waiting for the London team the when they arrive for work. Yet experience shows the reverse: increased distance between team members and customers results in slower communication.

The need to arrange conference calls, video and online meetings means spontaneous and informal meetings cannot occur. A question sent from London to Hong Kong on Monday afternoon (GMT) that is not answered by Hong Kong during their Tuesday cannot be chased until Tuesday in London, which may mean the answer does not appear until Wednesday.

The answer to all the problems outlined so far is to introduce a new role between the customers and developers – the age old 'add another layer of indirection' solution. This role gathers the requests, helps the requester with their logic, examines or creates the business case and value statement, and generally keeps the request pipeline ordered.

Which raises the question: who should fill this role? Often this person is some sort of manager. Project Managers are a popular choice for this role because they are professional organizers. But a close look at the skills required, and responsibilities inherent in this role suggests a Business Analyst would be more suitable – shown in Figure 5.

Project Management training is focused on the 'when' of work. They learn about work breakdown structures, contingency planning, risk logs, reporting and perhaps producing Gantt charts. In an Agile project much of this is irrelevant. Agile teams operate without Gantt charts and many of the traditional artefacts of Project Management.

Business Analysts on the other hand are trained in stakeholder identification and liaison, business and process analysis and requirements discovery. This makes them a better candidate for filling this role.

Yet every extra role that is placed between the development team and their customers introduces more potential gaps: messages need repeating, some get lost or distorted in the retelling, and more competing views and agendas are brought into to play. The more layers between the development team and the ultimate customer of the company the more the developers are isolated, and insulated from market forces and real customer focus. Each extra layer reduces the real Agility of the team and the business.

Sometimes the right answer is to remove layers. The difficulty lies in knowing when adding an extra layer will improve things, and when removing a layer is the right thing to do.

BAs and the other type of Product Managers

It is not only software companies that have Product Managers. Banks, telcos, FMCG and other companies have them however their responsibilities are to the customers of the real product: financial products, telephones, washing up liquid and so on. When these ultimate products contain a high degree of technology, and in this case software technology, it is better to have these Product Manager work directly with the developers.

In software companies Product Manager replace BAs because the software is the product. In such places a Product Manager who does not understand software, what it is, how it is created, the value it delivers and how it delivers value will have problems.

Yet when the product is not software the Product Manager is unlikely to understand software and IT in the way they need to work with the business team and it is necessary to introduce a BA.

Consider a traditional travel and holiday company. Products are sold through high street branches or through a call centre. Product Managers focus on customers' experience in the shop, on the holiday and stages in between. IT supports the experience but the Product Manager needs to understand holidays and customers, not IT. So they use a BA to make requests on IT.

But, imagine the company ditches its high street stores and closes the call centres. It offers the products online through a website. Until customers board the plane the experience is electronic and based on IT. Product Managers still need to understand customers and the travel market, but if they do not understand IT they are handicapped. The ultimate product now has a high IT content so the BA role should be removed and the Product Manager work directly with the technology team.

As more companies find their core products are delivered by software systems, the role those systems play in the product experience becomes more important. No longer is software a back-office operational issue; it is a part of the front-office environment customers engage with. Someone needs to represent the software by looking both externally to how customers engage and internally at how it drives the business. This role is part BA and part Product Manager, and I expect it to become more common and important.

The BA role on an Agile team

Most of the skills and experience a BA has can be carried directly from traditional projects to Agile projects. Gathering needs, talking to stakeholders, running workshops, writing business cases and so on are as important on Agile projects as any other. When these old techniques are carried forward they often occur in an accelerated fashion.

What will be new to some BAs is the need to step back from requirements and examine more closely what the business is trying to do and, more importantly, why. Rather than simply document some given need the BA needs to understand the motivation behind the request and the business value. Only with this information can the BA prioritise the work requests – something else that might be new.

Requirements

What BAs won't be doing any more of is writing long requirements documents or leaving a project when the coding has only just begun. Most BA work still occurs before code is written but the two phases overlap. The BA discovers a bit, the developers code a bit; and while the developers are coding the BA discovers a bit more so when the developers have completed the first bit there is a little bit more to do.

Business needs (i.e. requirements) are taken in bite-sized chunks rather than in thick binders. These small chunks of work have been called Minimally Marketable Features (MMF) or Business Value Increments (BVI). The emphasis is on just in time requirements that produce near term

business value rather than trying to discover everything that can possibly be known before asking for anything.

BAs need to be embedded in the team, in daily contact with developers, answering their questions and reviewing work as it is done. Simultaneously the BA needs to work slightly ahead of the development team, but only just ahead so they are ready for the next question, or to prioritise the next request.

The development team can only move fast if they are fed a constant stream of needs. But those needs are changing as they work – not least because work which is complete changes the view on what is needed next. The further the BA is ahead of the team the greater the possibility that the needs will change while they are waiting to be implemented.

The key BA skill, namely analysis, is to the fore. While developers are concerned with creating a solution, i.e. synthesis, it is the BA's role to work out what needs to be done.

Ensure business value

Research has repeatedly shown that as much as 80% of features or functionality in customer software development is unused [Poppendieck03]. (For commercial products the figure is usually 20% of the features used by 80% of the users.) Stemming the requests at source by understanding what is truly needed and what will actually be used can therefore reduce the team's workload by four fifths.

More importantly, Product Owners need to ensure these requests have business value attached. For a team to demonstrate its worth it needs to be producing valuable software. And if a team is going to turn down 80% of requests then it needs to ensure it does the 20% with the greatest value.

Gatekeeper and prioritiser

Product Owners on Agile teams are the gatekeepers to the development team. They decide what will be developed in the next iteration, what will be held until later and what will not get done at all. And when work is accepted into an iteration it is the Product Owner who sets the priorities for the team in the next development episode.

In order to make the best possible decisions about what functionality to develop and what priorities to set the Product Owner role needs to know about the value of the work being requested and overall objective. Sometimes they may be told the value but more often than not they will need to determine the value for themselves. This means they need understand what is being asked for, and how it aligns with the objective. They need to know about all the options and different requests being made on the system when they make the call.

They also need to know when the best option is to do nothing, and when to cease development. If value falls below cost then the greatest value comes from doing nothing.

While business value is the most obvious criteria for determining priorities there are others. Some may prefer to prioritise by risk, or others by 'juicy bits' – those features which will get the most attention.

Prioritisation criteria can, and do, change over time as work progresses. In the first few iterations a few 'juicy bits' may be picked off and delivered to demonstrate progress. The next few iterations might attack risk directly while later iterations use straight business value.

Go to

As if this weren't enough the Product Owner is the 'go to person' for two groups of people. The first group is those who want the software to fill some need, whether we consider this people customers, users, stakeholders or simply 'the business' all requests should travel through the BA (or the BA team). This is necessary both so requests can be validated and prioritised and in order to reduce disruption to the developers.

The second group for whom the BA is the 'go to person' is the developers. When a developer needs more information about a request or when some unforeseen question arises they need to ask the BA. In some cases developers might be able to go to the final customer or user with such a question but when this is not possible, or not appropriate, then the BA takes on this role.

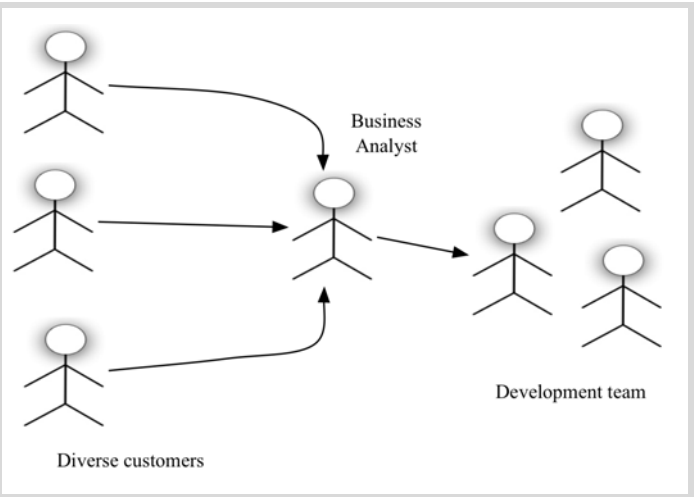


Figure 6

When the BA is also a subject matter expert (SME, sometimes called a domain expert) this they may be able to answer the question directly, other times the BA will need to know where to go to find an answer, or be prepared to make a judgement call.

Where delay occurs in answering a question development should not proceed. If a developer makes a guess there is every chance the guess will be wrong resulting in rework, delay and disruption. Alternatively a developer may work on another piece of work but this too results in disruption as one piece of work is laid to one side and another starts – and consequently tracking work is more difficult. It may well be better to have a developer do nothing rather than take their focus away from the work in hand.

In fulfilling the roles of gatekeeper, prioritiser and go to person the BA has to continually keep the delivery to the business at the front of their mind. The BA's primary responsibility is always to ensure the work being undertaken will produce business value. Ensuring business value is delivered in a timely fashion is not just a case of determining what needs to be done and asking for it. There are a multitude of options, possible actions and decisions which follow once need is determined. Maintaining a steady flow of deliveries means keeping sight of the overall objectives.

In closing

The lack of a clear BA role in Scrum and XP has created confusion about what, if anything, Business Analysts do on Agile teams. Yet there is an important role for them to play in keeping the corporate arteries clear; ensuring considered and valuable requirements reach Developers, and ensuing that Developers get the information they need, when they need it.

There is more to Agile software development than simply declaring your team 'Agile'. The Magic Agile Dust only works if teams and their management are prepared to make real changes. This includes changing the way needs are presented to the teams. The changes in the BA role are perhaps more subtle than the changes to the Developer role but are just as key in delivering genuine agility. ■

References

[Brynjolfsson09] Brynjolfsson, E. 2009. *Wired for Innovation*, MIT Press.

[Poppendieck03] Poppendieck, M. and Poppendieck, T. 2003. Lean *Software Development*, Addison-Wesley.

The Model Student: The ACCU 2010 Crypto Challenge

Electronic computers advanced cryptography enormously. Richard Harris sets a challenge, and finds a solution.

Once again I have had the good fortune to have been invited to contribute a cryptographic puzzle as part of the fund raising efforts for Bletchley Park and the Museum of Computing during the ACCU conference.

Last year, if you recall, I designed a puzzle based upon the Enigma machine which could be broken with pencil and paper in 15 to 30 minutes once the weaknesses in the rotor mechanism and the crib in the message had been spotted.

As was the case with that puzzle, this one has been designed to be possible to solve with pencil and paper alone and includes a bonus question that cannot be answered if the problem is simply brute-forced.

So that you too may enjoy the puzzle, it is repeated below followed by its historical justification, its solution and the names of the master cryptographers who solved it at the conference.

The challenge

Encoding

The enemy are using a 32 character alphabet encoded as 5 bit unsigned binary numbers. The # character is a control code indicating that the following character should be interpreted as its numerical value rather than as a letter or punctuation. The full table of character mappings is given below.

```
00000000001111111111222222222233
01234567890123456789012345678901
#abcdefghijklmnopqrstuvwxyz ?!.,
```

Encryption

The encryption scheme used by the enemy is a symmetric key stream cipher in which each plaintext number is bitwise exclusive or-d with the key to create the ciphertext number after which the plaintext number is added to the key (discarding any bits above the 5th) to create a new key.

An in-place C implementation of this scheme is given in Listing 1.

```
void
encrypt(unsigned char *s, size_t n,
   unsigned char key)
{
  while(n--)
  {
    unsigned char c = *s;
    *s++ ^= key;
    key += c;
    key &= 0x1f;
  }
}
```

Listing 1

Plaintext

We know that the enemy prefix their messages with a single 5 bit number (i.e. one character) representing the message's priority. We also know that they foolishly postfix their messages with the date in the form

```
DDMMM
```

where `DD` is the day and `MMM` is the month, represented by a number and a three character abbreviation respectively.

The enemy encrypt each message with a randomly chosen key. This message key is repeated twice in a 2 character header at the start of the plaintext which is itself encrypted using a daily key.

Example

The plaintext

```
kkbflee#dfeb
```

represents a priority 2 message of 'flee' sent on the 4th February that will be encrypted with the message key 'k' and the daily key using

```
assert(*plaintext == *(plaintext+1));

encrypt(plaintext+2, 10, *plaintext);
encrypt(plaintext,   2,   daily_key);
```

If the daily key is 'd' this yields the ciphertext

```
odik,zaimkvz
```

Ciphertext

Decrypting the following ciphertext will reveal the number of the box in which to deposit your ticket.

```
ynb#zuybzmot.vt!xbhaxh
```

Bonus question

If we add to the ciphertext and the information we have about the structure and content of the plaintext a set of guessed key and/or plaintext characters at specific locations in the plaintext, and if we assume that these guessed characters are added to this set in the worst possible order that is logically consistent with a cryptanalysis, how large must this set be to guarantee that we correctly deduce where to deposit the ticket?

The historical justification

After the electro-mechanical rotor-based cryptosystems like the Enigma, the next big development in commercial cryptography were the digital electronic cryptosystems of the 1970s.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

These ciphers draw their inspiration from the only provably unbreakable encryption algorithm: the one-time pad

Perhaps the most successful of these was the Data Encryption Standard, or DES, developed by IBM as a candidate for a government encryption standard proposed by the US National Bureau of Standards and not officially retired until 2005.

This was a block cipher, in which the plaintext is broken up into relatively large blocks, each of which is algorithmically combined with a fixed key.

In contrast to block ciphers are stream ciphers which bear a much greater resemblance to their forebears, being polyalphabetic substitution ciphers operating on single characters, or even on single bits, of the message at a time. In doing so, they are attractive for telecommunications, for which the size of the message can rarely be determined in advance and the hardware may have limited volatile memory in which to store the message data.

These ciphers draw their inspiration from the only provably unbreakable encryption algorithm: the one-time pad. To operate a one-time pad, the sender and receiver of encrypted messages must first exchange identical sequences of random characters. To encrypt a message the sender iterates through both its characters and those in the sequence of random characters, combining them; typically with x-or. To decrypt a message the receiver performs the same operation on the ciphertext. Crucially, each must discard the random characters as they are used; if they do not do so their messages can be broken with statistical analyses.

The difficulty in exchanging the random characters makes the one-time pad unattractive for almost all applications. It is therefore extremely tempting to replace the random sequence with a pseudo-random sequence so that the two parties need only exchange the much shorter seed. The resulting cryptosystem is no longer provably unbreakable, but may be practically unbreakable.

Provided the pseudo-random sequence of characters is not too predictable, that is.

The solution

To decrypt the message we apply the same algorithm, although since the plaintext character is recovered after the x-or we don't need to store it to add to the key. An in-place C implementation is given in Listing 2.

```
void
decrypt(unsigned char *s, size_t n,
   unsigned char key)
{
  while(n--)
  {
    *s ^= key;
    key += *s++;
    key &= 0x1f;
  }
}
```

Listing 2

Trivially, we could brute force the message and discover the plaintext with a worst case of 32 guessed characters. The real question is whether or not we can decode it with fewer.

Noting that the enemy postfix their messages with the date we know that the 5th character from the end must be `'#'`, represented by a `0`. Since the ciphertext is generated by x-oring the plaintext with the key we can deduce that the key for this character must be 'b'. Running the last 5 characters through the decryption function yields

```
#jmar
```

indicating that the message was sent on the 10th March.

We might assume that the final characters of the message represent the number of the box, in which case the 7th character from the end must also be a '`#`', meaning that the key must be `'!'`. Using this key to decrypt the last 7 characters of the message yields

```
#e#jmar
```

which is entirely consistent with what we already know about the plaintext. This is reasonably convincing evidence that the ticket should be deposited in box 5 and that we need just one guessed character.

Reasonably convincing, but wrong.

In last year's Enigma challenge, the predictable structure of the message's postfix signature was the crib that ultimately allowed its decryption. As such it was too tempting for me to resist using it as a red herring in this puzzle.

This time the crib is, in fact, the repeated message key.

The trick is to recognise that for single bits, addition is exactly the same as x-or. Specifically, for any pair of integers, we have

```
(x+y)&1 == (x^y)&1
```

Denoting the ith bits of the daily key, the message key and the jth character of the plaintext as d_i, m_i and $T_{j,i}$ respectively, we have

$$\begin{aligned} T_{0,0} &= d_0 \oplus m_0 \\ T_{1,0} &= (d_0 + m_0) \oplus m_0 \\ &= (d_0 \oplus m_0) \oplus m_0 \\ &= d_0 \oplus m_0 \oplus m_0 \\ &= d_0 \end{aligned}$$

From this we can deduce that the least significant bit of the message key is given by

$$m_0 = T_{1,0} \oplus T_{0,0}$$

We can perform a similar calculation for the second least significant bits, but must take into account the fact that there may have been a carry when the daily key was added to the message key.

Fortunately, we can retrieve the carry bit c_1 from the information we have already extracted from the message.

The difficulty in exchanging the random characters makes the one-time pad unattractive for almost all applications

$$c_1 = d_0 \wedge m_0$$

Denoting the *i*th carry bit as c_i we can generalise our formulae to bits other than the least significant.

$$\begin{aligned} c_1 &= d_{i-1} \wedge m_{i-1} \\ T_{0,i} &= d_i \oplus m_i \\ T_{1,i} &= (c_i + d_i + m_i) \oplus m_i \\ &= (c_i \oplus d_i \oplus m_i) \oplus m_i \\ &= c_i \oplus d_i \oplus m_i \oplus m_i \\ &= c_i \oplus d_i \end{aligned}$$

giving us a recursive procedure for extracting the bits of the keys.

$$c_i = \begin{cases} 0 & i = 0 \\ d_{i-1} \wedge m_i & i > 1 \end{cases}$$

$$\begin{aligned} d_i &= c_i \oplus T_{1,i} \\ m_i &= c_i \oplus T_{1,i} \oplus T_{0,i} \end{aligned}$$

The first two characters of the ciphertext are

```
y = 25 = 11001
n = 14 = 01110
```

Applying this procedure yields

$$\begin{aligned} c_0 &= 0 \\ d_0 &= c_0 \oplus T_{1,0} = 0 \oplus 0 = 0 \\ m_0 &= c_0 \oplus T_{1,0} \oplus T_{1,0} = 0 \oplus 0 \oplus 1 = 1 \\ c_1 &= d_0 \wedge m_0 = 0 \wedge 1 = 0 \\ d_1 &= c_1 \oplus T_{1,1} = 0 \oplus 1 = 1 \\ m_1 &= c_1 \oplus T_{1,1} \oplus T_{0,1} = 0 \oplus 1 \oplus 0 = 1 \\ c_2 &= d_1 \wedge m_1 = 1 \wedge 1 = 1 \\ d_2 &= c_2 \oplus T_{1,2} = 1 \oplus 1 = 0 \\ m_2 &= c_2 \oplus T_{1,2} \oplus T_{0,2} = 1 \oplus 1 \oplus 0 = 0 \\ c_3 &= d_2 \wedge m_2 = 0 \wedge 0 = 0 \\ d_3 &= c_3 \oplus T_{1,3} = 0 \oplus 1 = 1 \\ m_3 &= c_3 \oplus T_{1,3} \oplus T_{0,3} = 0 \oplus 1 \oplus 1 = 0 \\ c_4 &= d_3 \wedge m_3 = 1 \wedge 0 = 0 \\ d_4 &= c_4 \oplus T_{1,4} = 0 \oplus 0 = 0 \\ m_4 &= c_4 \oplus T_{1,4} \oplus T_{0,4} = 0 \oplus 0 \oplus 1 = 1 \end{aligned}$$

Giving us daily and message keys of

```
01010 = 10 = j
10011 = 19 = s
```

Using these to decrypt the message yields

```
ssqdrop no. three#jmar
```

meaning that the ticket should be placed in box number 3 and that it was not necessary to guess any characters.

And finally...

Congratulations to Duncan Gary Duke, Andrew Kemp, Per Liboriussen, Callum Piper and Martin Turnock who successfully cryptanalysed the Crypto Challenge.

To everyone who took part, and to everyone who donated to Bletchley and the Museum of Computing, we extend our deepest gratitude. ■

Renovating a Legacy C++ Project

Over time projects tend to become hard to maintain. Alan Griffiths describes how one was improved.

I've been using C++ for a long time, at first because it was the principle language available for developing desktop applications on OS/2 and Windows. More recently it has been chosen either for non-technical reasons or because it provides better control over resources than other popular languages. As a result of this and the efforts of others like me the world is now full of functionally rich, slow to build, hard to maintain, C++ systems. Some of these have been developed over long periods of time by many and varied hands.

These C++ systems continue to exist because they provide valuable functionality to the organisations that own them. To maximise this value it is necessary to provide interfaces to today's popular application development languages, and make it possible to continue to develop them in a responsive and effective manner.

The need to work with other languages comes from the forces that change our industry: as computers and development tools have got more powerful we have tackled bigger and more complex problems. To facilitate tackling each part of the problem in the most effective way, it is common for different parts of the system to be built using different programming languages. In recent years the projects I've worked on have included C++ with combinations of Java, C#, Python and Javascript.

The project

The project that I'm going to describe used C++ for a mixture of reasons – the non-technical reason was that the codebase has been developed over a couple of decades, originally in 'C With Objects' but more recently, after a port, in C++. The technical reasons for choosing C++ were the usual 'control over resources' ones – principally CPU and memory. There are man-centuries invested in the codebase so a rewrite in a fashionable language would be hard to justify.

The code in question is a 'Quantitative Analytics Library' – it does the numeric analysis that underlies the trades done by an investment bank. This isn't the place to explain these trades and valuation methods in any detail (even if I could). Briefly, investment banks trade in a range of 'assets': futures of commodities, derivatives of stocks, currencies, bonds and indexes, and base their valuation on the available information (mostly historical pricing information).

Alan Griffiths has long been a contributor to ACCU, including chairing the organisation, editing Overload, contributing articles to the magazines and presenting at conference. During that time he's delivered working software and development processes, written contributions for magazines and books, spoken at conferences and made many friends.

He has been developing software through many fashions in development processes, technologies, and programming languages. Firmly convinced that common sense is a rare and marketable commodity he's currently working as an independent. He can be contacted at alan@octopull.co.uk

An analytics library takes this data to assess the price needed to assure a likely profit from each trade. Among other things it builds multi-dimensional data structures of largely floating point numbers and processes these on a number of threads – small changes to layout and processing order can have big effects on performance. (And the resulting numbers!) Using C++ does indeed give some control over this – hence, while there are other plausible languages for this work, C++ remains a popular choice for such code.

There are a lot of applications in different areas of the bank that make use of this library to value the trades they are making. Most of the Linux based user applications are using Java, and most of the Windows ones are using C#. (It also exists as an Excel plugin – that's written entirely in C++.)

The codebase is monolithic – highly coupled, incohesive and with no agreed 'public interface', but it does have a suite of end-end tests that covers all the financial models supported in production and, at least in principle, any bug fixes do come with a corresponding test case.

The developers of these financial models are known as 'quants' (from 'quantitative analytics') and specialise in their domain knowledge rather than software engineering. I and a couple of other developers worked with them to redress this balance. (In this case the quants worked principally on Windows – of which more later.)

The library is supported on a range of platforms: primarily 32 bit Windows and 32 bit Linux, but 64 bit Linux is supported in development and planned for roll out later this year, and Win64 is under development – to be supported next year.

The legacy interface for users

Historically the users have been given a set of libraries (.sos or .dlls) and all the header files extracted from the codebase. Unsurprisingly the Java and C# users are not happy with this as the supported interface. Nor would any hypothetical C++ users be happy as things are forever changing (because there is no agreed public interface). It is also far from clear how a particular type of trade should be valued. Each application team therefore has to do work to map from its own representation of trades to the mechanics of configuring the corresponding analytic model for valuing it. (For example, it was the application team that constructed the correct choice of asset price and price volatility models for the trade.)

At some time in the past one such client group wrote a series of scripts to generate and build a Java interface to the library using SWIG [SWIG]. Other groups started using this interface and it is now shipped with the analytics library. This isn't ideal as, while there are multiple groups using this interface, there are no tests at all. Provided it compiles and links it will be shipped. When we started work the code wasn't even part of the main repository: it was a svn 'extern' to a repository owned by the group who once employed the original author – we didn't have commit access. (There was also a C# derivative of this work, but this will not be discussed separately.)

The lack of ownership of this SWIG interface generating code was particularly problematic as the same code was pulled in by all the active

With these issues to contend with it could take over three months to get a new type of trade and valuation model into production

branches. (There are typically a couple of branches in production and another in development – but this can increase occasionally.) The main problems occur if changes on one of the branches necessitate changes to the generating code – which, as it has lots of special case handling for particular methods and constructors, happens. (In particular SWIG isn't able to expose the C++ distinctions between const, references, pointers and smart pointers – this can, and does, lead to unintended duplication of method signatures. There are some sed scripts to hide these problematic functions from SWIG.)

One of the first changes we made was to 'adopt' this code into our repository so that we could fix problems for each of the active branches independently. This wasn't entirely satisfactory as we still had no tests or access to the client code to check it compiled against the generated interface – the best we achieved was to ensure that the SWIG library compiled and linked then wait for users to 'shout'. (We could have done more to ensure the quality of these releases, but as we wanted people to move off this 'legacy interface', we felt we'd get better returns on time invested elsewhere.)

The new interface for users

As mentioned earlier, the legacy interface caused problems for our users. The interface didn't reflect normal Java (or C#) conventions, wasn't stable and reflected the need of the implementer, not the user. Each client application's developers needed to understand not only the trades they were valuing but also the correct way to value them. In addition, they needed to get 'sign-off' of the valuation models being produced for each trade type they implemented.

With these issues to contend with it could take over three months to get a new type of trade and valuation model into production. For competitive reasons the business wanted to move faster than this.

To address this we built a new, more stable, public interface that directly supports Java and C# and incorporates the mapping between a trade definition and the valuation models. Thus the client would supply the price and volatility data and our API would know the appropriate way to model these based on the information supplied about the trade. This would be much easier for client applications as the developers need only to present the trade (and market data) in an agreed format and don't need to be concerned with the method of valuation. This comprised a number of components (see figure 1):

- one that supports a uniform data representation (this can be thought of as a subset of JSON – as that is its serialised form);
- another that maps a trade representation to a model representation (by passing it to a library of Javascript functions that implement the mappings);
- a third that uses the modelling library to value the model (actually we introduced support for a second modelling library – so there were two of these);
- a fourth that manages all of this; and

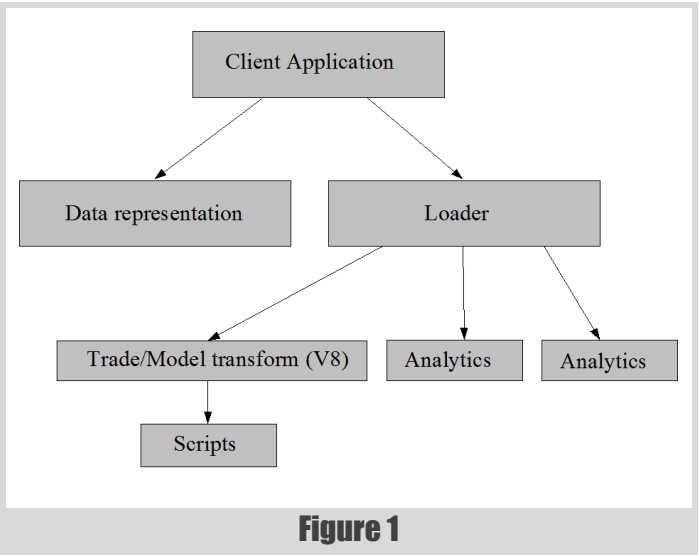


Figure 1

- native C# and Java APIs that provide access to all of this.

Much of this is implemented in C++, but there are obviously bits of C# and Java and the mappings between trades and models is implemented in Javascript.

We address the lack of an idiomatic API by creating a C# API using properties for the data representation and IDisposable for resource management (so that 'using' blocks correctly managed C++ resources). The corresponding Java API was designed using setters and getters and a `dispose()` method. In both languages errors were represented by exceptions (one to say 'your input doesn't make sense' another to say 'that valuation failed').

Initially we implemented the C# API using 'Platform Invoke' and the Java API using 'JNA'. These are roughly equivalent technologies based on reflection that, given an interface coded in the corresponding language implements the calls to the native code at runtime. (There is a runtime cost to this, but it does provide a quick way to implement access to native code.) Given the large amounts of processing by the analytics library we didn't expect the performance to be an issue.

We later found that one of our client application teams using Java had two problems with this implementation. Firstly, they were doing a lot of fast running valuations and that for them performance was an issue and secondly, they were doing some fancy custom `ClassLoader` tricks and these clashed with similar practices in the JNA implementation – switching to a JNI implementation of our interface removed these difficulties.

The client interfaces have remained relatively stable over time. During the first six months there were binary interface changes, and through the first year there were tweaks to the trade definitions to approach a more uniform naming style (and to co-ordinate with a global initiative to represent trade elements in the same way throughout the business). All of that is settling

a new type of trade was implemented by a client application in three weeks instead of the three months that would have previously been required

down and work now focusses on reflecting changes and enhancements to the valuation engine and providing mappings for additional trade types.

Naturally, we introduced some 'acceptance tests' for the supported trade types that ensured that they were validated correctly. This greatly simplified the task of application developers who now only need to sign off that they were presenting us with correct trade representations (our tests established the correctness of the results for all the application teams).

As a measure of success a new type of trade was implemented by a client application in three weeks instead of the three months that would have previously been required. They were also able to ditch several hundred thousand lines of code when migrating code to the new interface. (Although, as they were also removing a massive tangle of Spring, that work must share some of the credit.)

Building integration and release

When I joined the project much of the effort was expended 'firefighting' the build and release process. There were a number of problems:

'Clever' use of CruiseControl

CruiseControl [CruiseControl] was used to manage continuous integration for the supported release branches and integration on the trunk. Something unusual was clearly going on as it was managing builds for both Windows and Linux. Instead of the more common email based reporting on builds summary results were being published to a 'chat' channel the developers subscribed to. Definitely not a 'vanilla' CruiseControl setup. It took a while to figure out what was going on!

With understanding, things were not as good as first appearances had suggested: it turned out that despite appearances and natural assumptions, CruiseControl didn't actually control either the checkouts or build processes directly. What CruiseControl treated as a build was actually a shell script that wrote a token file to a shared directory – another shell script on the corresponding target platform polled for these tokens and checked out the current source and built it. This then wrote the build results back to the share for the first script to pick up. The results were confusing as HEAD often changed between CC scanning the repository and the build script checking things out. The result of this inventiveness was:

- No reporting on regression tests
- Incorrect change reporting
- Poor error reporting

Something had to be done. But there were more issues to deal with than that.

The Linux build

Even with good code the Linux build failed about half the time – this turned out to be a parallelism issue, one make rule created a directory, another wrote to it and there was no dependency between them. Some very fragmented makefiles (lots of included fragments) made this hard to spot.

A less frequent cause of build failures was that the quants worked on Windows and the code checked in was not always good for Linux. Two problems we saw after any significant commit were that the case was wrong on #includes and that it was often necessary to add standard headers (for content that was implicitly supplied by others on Windows).

The Windows build

The Windows build was also in a mess. No-one quite knew why but the Windows 'Release' build configuration would fail if the Windows 'Debug' build configuration wasn't built first. However, apart from being needed to create the 'Release' build the 'Debug' build 'wasn't used'!

The quant developers worked with a more useful 'Debug(DLL)' build configuration that allowed components to be worked on independently – the 'Release' and 'Debug' builds produced a single DLL which took an age to link. The upshot of this was that the build server built three build configurations in succession 'Debug', 'Release' and finally 'Debug(DLL)'. There was also a 'Quantify' build configuration but no-one maintained it.

Building all these configurations took a few hours.

The release build process

The above describes the integration build. You might be forgiven for assuming that the release process was based on the same mechanism. You'd be wrong.

In order that the code released could be built from a label (yes, I know that Subversion allows you to label what you built afterwards) there was a pre-release build (with the same steps as the integration build) that would label the sourcecode for release if it succeeded. Again, HEAD would often change after the pre-release build was requested. (And, as the build failed randomly half the time, it often took several attempts to build before a label was applied – with the codebase evolving all the time.) Once a build had succeeded and a label applied another script would be started to build the software from the label. (As before the build failed half the time, but at least the code wasn't changing).

Once the release binaries had been created it was time to push the binaries out to the development and production environments. (You weren't thinking 'testing' were you? 'Fortunately' this system was a library for other teams applications to use, and they did need testing before releasing a version of their application to production using our release.)

Back to pushing out the binaries to the development and production environments. This was also problematic. Apart from the inevitable organisational change control forms that needed to be completed, there was a sequence of scripts to be run in sequence. The various shell and perl scripts involved were fragile. They had hidden hardcoded interdependencies, poor error checking, needed to be run as specific accounts, and no documentation.

Oh, and no version control. Not only did this mean that one had to track changes manually, it also meant that the same script was used for building a 'patch release' for a year old release branch as for the current version.

What started as a slow to update component with a hard to use and unstable interface changed into a much more responsive and user friendly tool

This meant, for example, that adding components for the new interface wasn't just a matter of adding them to the scripts, but also surrounding these changes with tests for the versions in which they existed.

Slow and steady

Although these issues were a constant drain, fixing them was a background task – getting releases out consumed about one developer's productivity. On top of which there was the new library interface to be built. With a small team (two for a lot of the time) progress was slow.

Changing the build system was also time consuming – even with pairing to review changes, and trying to separate out bits that so that they could be tested independently, changes tended to break things in ways that were not detected until the next release was attempted.

But every time we fixed one of these problems life got easier. The race condition in the Makefile was the first fixed and made things a lot more predictable. Killing the dependency on 'Debug' and eliminating the 'Debug' and 'Quantify' build configurations also sped up integration and release builds. (In later days the 'Debug(DLL)' build was renamed 'Debug' to keep things simple.)

To fix the poor error reporting from integration we needed to build the version of the code that the CI tool was looking at. This could have been fixed with CruiseControl but, in practice, waited until we gained a team member with experience of TeamCity [TeamCity]. He replaced CruiseControl with TeamCity which has direct support for having 'build agents' on a variety of platforms and eliminated the scripting complexity used to achieve this with CruiseControl. (There are other continuous integration servers that might have served instead of TeamCity – Hudson [Hudson] is often suggested. Support for 'build agents' was one motivation for changing, another was the web interface for managing configuration which was a lot easier to learn than the XML files driving CruiseControl.)

TeamCity also integrates an artefact repository (in the past I've combined Ivy with CruiseControl to the same effect, but having it 'out of the box' was nice). This allowed us to eliminate the pre-release and release builds – our new release process (automated as Ant scripts run by TeamCity) took the binaries directly from the integration build, tagged the corresponding source revision in the repository, and then distributed the binaries to the development and production environments around the world.

It took around six months to address all the problems we'd seen initially and life was a lot easier – we could focus on the new library interface.

The benefits we saw

If integration was 'green' then a release could be requested of that code (and even if integration were broken, if a recent successful build contained the needed changes then that version could be released).

The build process was easy to change and enhance. We split out builds of separate components to give faster feedback.

We also added automated regression and validation tests as TeamCity 'build configurations' within the project. This was helpful to the application developers that were using our library through the 'new' interface – we had already generated much of the test evidence for the supported models.

Compared to the fragile collection of scripts we had been using copying build configurations in TeamCity is easy. This enabled us to set up parallel builds to test changes to compiler and library versions and extend support to 64bit Linux.

Copying projects in TeamCity is also easy – so when a stable release build was branched for maintenance it was easy to set up the integration build for it.

The ability to run multiple build agents meant that we could have a lot of building happening in parallel – which improved responsiveness to checkins.

Things were a lot more productive. Where a release once took a week and most of a developer's time it was now a few minutes of form filling (irreducible bureaucracy – although we did talk of scripting this) and a couple of button clicks. A couple of hours later the code was in production.

Conclusions

We managed to get buy-in from the management, clients and quants by being responsive to the current issues and regularly delivering incremental improvements (every week brought something visible). As releases became faster and more reliable, interfaces became cleaner and builds more reliable and faster, it became accepted that we could be trusted to fix the things brought to our attention.

What started as a slow to update component with a hard to use and unstable interface changed into a much more responsive and user friendly tool. The team and the client are happy with what we achieved and so are the client applications.

It is possible to make legacy C++ systems into projects fit for the third millennium! ■

References

[CruiseControl] http://cruisecontrol.sourceforge.net/

[Hudson] http://hudson-ci.org/

[SWIG] http://www.swig.org/

[TeamCity] http://www.jetbrains.com/teamcity/

Single-Threading: Back to the Future? (Part 2)

Multithreading can cause notoriously difficult bugs. Sergey Ignatchenko finds mitigating strategies for programs on servers.

As we have seen in the previous article [Ign10] which described the 1st half of the historical match between our 'No Multithreaded Bugs' Bunny and Multithreaded Gorrillazz, in most cases our Bunny has managed to reach the touchdown area without the need for heavily multi-threaded development. It means that on the client side the number of cases which are really calling for heavily multithreaded code (known to have numerous problems) is rather limited, and so in most cases heavy multi-threading can be avoided. Now it is time to take a look at server-side programs to see if our hero can avoid heavy multithreading there. As 'server-side' we will consider programs aimed to support many users simultaneously over the network. Web applications is one prominent example, but we will not limit our analysis only to them.

One common feature of server-side applications is that they almost always depend on some server-side storage, normally a database; therefore, we will assume that some database (not specifying if it is relational or not) is used by application. Even if the application uses plain files for storage, we still can consider it as a kind of (very primitive) database for the purposes of our analysis. There are a few exceptions to this database dependency, but as we will show below in 'No Database? No Problem' section, the most practically important of them can be handled using the same techniques as described here and in part1.

It should be noted that, as before, this is mostly a summary of practical experiences and cannot be used as strict instructions for 'how to avoid multithreading'. Still, we hope it will be useful as a bunch of 'rules of thumb'. We will also try to provide some very rough numbers to back up these ideas, but please take them with a good pinch of salt.

Quarter 3&4 line-up

As in the first half of the match, we have our 'No Multithreaded Bugs' Bunny standing on the left side of the field, with touchdown being his only chance to win. He faces a team of extremely strong 'Multithreaded Gorrillazz', and any single Gorrilla is strong enough to stop him forever. Fortunately they're rather slow, which leaves our hero a chance. As in the first half we will make a distinction between heavily multithreaded code all over the place (which results in perpetual debugging and a maintenance nightmare) and isolated multithreaded pieces (which are not exactly a picnic either, but can be dealt with with finite amount of effort; we will consider them acceptable if there are no better options).

To discuss server development, the very first thing we need to see is if our Bunny is writing a program based on standard interfaces and frameworks, or needs to develop his own framework. If a framework is already there (for example, he's developing a web application), it simplifies his task significantly. As the whole idea of the server-side application is to serve independent requests from many users, most existing frameworks (eg CGI/FastCGI, PHP, .NET, Java Servlets) do not require any interaction between requests. Sometimes avoiding interaction between threads within the framework requires some discipline (for example, static data in Java servlets can cause inadvertent interactions leading to problems [CWE-567]), but overall it is usually not rocket science to avoid it (unless it is dictated by application logic itself, which is discussed below).

Now, let us consider the scenario where standard interfaces are not good enough; while it is not so common, there are still several good reasons not to use standard web frameworks in some specific cases. Such reasons may include, for example, the inherently request-response nature of the HTTP protocol, which doesn't necessarily fit all application usage scenarios. The decision to write your own framework is always a tough one and obviously includes lots of work, and often such frameworks need to be multithreaded for performance reasons. But even when it is really necessary, the framework can still be written such that all multithreading stuff is kept within the framework itself and is never exposed to the application. It means that even if our hero has quite an unusual case when existing frameworks don't work, he can still confine multithreading to relatively small and (probably even more importantly) rarely changed area of the code.

Wazzup, doc?

Now, one way or another, our 'No Multithreaded Bugs' Bunny has a framework which handles multiprocessing and multithreading itself, without imposing that his application code be multithreaded. It doesn't mean he will be able to avoid multithreading in the end, it merely means that he hasn't been grabbed by any of Gorrillazz yet.

The next question to our 'No MT Bugs' Bunny is the very same 'Houston, do we have a problem?' question that he needed to answer for the client-side. The main reason for multithreading is performance, so if there are no performance problems there is no real need to do anything about it (and if multi-threading exists for any other reason, our 'No Bugs' Bunny should think about it twice, especially if threads were added only because it's 'cool' or because without them the program is 'so 1990-ish'). If there are any observable performance problems, the very first thing our Bunny should ask himself is 'Are you sure that the database has all the indexes it needs?' It is unbelievable how many cases can be drastically improved by simply adding one single index. In particular, developers and even DBAs often tend to forget that a 2-column index on columns A+B can be orders of magnitude faster than 2 separate indexes on column A and column B. The biggest improvement we've observed from adding a 2-column index, was 1000x; not a bad incentive to take a closer look at indexes. So, we want to re-iterate: when using databases, indexing is of paramount importance, and is the very first thing to be considered if there are any performance problems. No amount of multithreading/multi-coring will save your program if the database lacks appropriate indexes. Another thing to take a look at this stage is eliminating outright inefficient requests (there are usually at least a few in any application, and basic profiling using database-provided tools should be able to help).

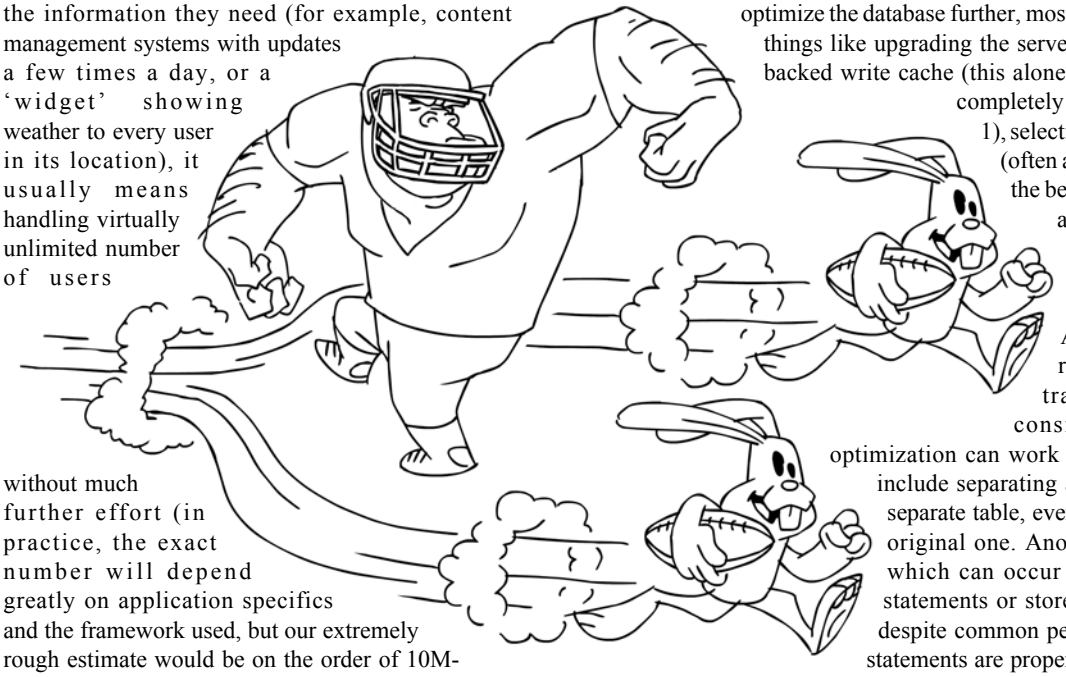
Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

where updates are rare but the number of read requests is huge, the next step is usually to see if it some kind of caching will help

If the database indexes are fine and there are still performance problems (for Internet applications it usually won't happen until about 1M-10M requests/day[1]), than the next question arises. Usually most applications of this kind can be divided into two wide categories. The first category of applications is 'data publishing' and have mostly read-only requests (represented by any kind of site which publishes information, including serving search requests). The second category makes many updates, but these updates are usually trivial and, after optimizations mentioned above, should take rather little time; reporting can still be ugly and have heavy and very heavy requests (this is a typical pattern of 'Online Transactional Processing', or OLTP, applications). At this point our Bunny should understand which category his application belongs to.

The clone bunny

For a 'data publishing' application where updates are rare but the number of read requests is huge, the next step is usually to see if it some kind of caching will help. Caching does introduce interactions between requests, but with a proper implementation (similar, for example, to memcached [Facebook08]) it can easily be used in a way which has nothing to do with multithreading. For applications/sites which can cache all the information they need (for example, content management systems with updates a few times a day, or a 'widget' showing weather to every user in its location), it usually means handling virtually unlimited number of users without much further effort (in practice, the exact number will depend greatly on application specifics and the framework used, but our extremely rough estimate would be on the order of 10M-100M requests per day per typical 2-socket 8-core 'workhorse' server, with an option to add more such servers as necessary). If, on the other hand, there are essential requests which cannot be handled from the cache (for example, ad-hoc searches) and even after caching everything we can performance is still not good enough, then things become more complicated. At this stage, our 'No Multithreaded Bugs' Bunny should consider creating a 'master' database which will handle all the updates, and multiple 'replica' databases which will handle the read-only requests.[2] This approach will allow scalability for read-only requests (with an extremely rough estimate of number of requests per single 'workhorse' server on the order of 1M-10M per day, though with proper optimization in some cases it can reach as high as 100M), so the only risk which remains is handling the update requests; usually it is not a problem for this kind of application, but if it is – our 'No MT Bugs' Bunny can approach them the same way as described below for typical OLTP applications.



Heavy-weights

So, what should our 'No MT Bugs' Bunny do if he faces an application which needs to handle lots of updates (more than a few million per day) and still experiences problems after all necessary indexes are present and the outright inefficient requests are eliminated? The next step is usually to optimize the database further, mostly at the physical level. It could include things like upgrading the server to a RAID controller with a battery-backed write cache (this alone can help a lot), moving DB logs to a completely separate set of HDDs (usually RAID-1), selecting an optimal RAID structure for tables (often a simple bunch of RAID-1 arrays works the best, and RAID-5/RAID-6 are usually not a good idea for heavily updated tables), separating tables with different behavior into separate bufferpools and onto separate physical disks, and so on. Additionally, moving most (or all) reports to 'uncommitted read' transaction isolation level could be considered; in some cases this simple optimization can work wonders. A related optimization can include separating a few frequently updated fields into a separate table, even if such a table has 1:1 relation to the original one. Another application-related optimization which can occur at this stage is moving to prepared statements or stored procedures. It is worth noting that despite common perception, on a DBMS where prepared statements are properly supported (last time we've checked

1. All numbers in the article are extremely rough estimates, your mileage may vary. Also we're assuming 'typical' Internet application with 'typical' distribution of requests over the day, with difference between minimum hour and peak hour not exceeding 2-5x. Still, while numbers are extremely rough, we feel that even such rough numbers can be of some value on initial stages of analysis.

2. Unfortunately, way too many RDBMS still experience problems under heavy load when replication is implemented using RDBMS-provided means. Heavy testing with comparable to production loads and data volumes is advised when trying to implement replication. As a workaround, custom application-level replication can be considered, but it is rather complicated and is beyond the scope of this article.

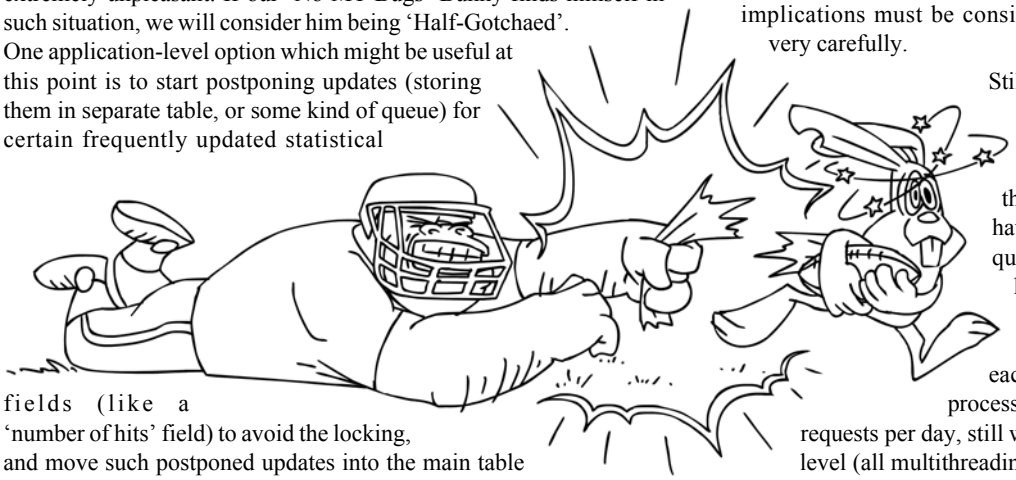
it still didn't include MySQL) they tend to provide almost the same performance as stored procedures, while requiring less code rewriting and keeping more potential for switching the DBMS if necessary.

Half-gotchaed?

What will happen if our 'No MT Bugs' Bunny did all the above optimizations, but his system or program still doesn't work efficiently enough (which we estimate shouldn't normally happen until 10M update requests/day is reached)? It is no picnic, but is still not as bad as heavy multithreading, yet. At this stage our hero can try to separate the operational updatable database from the read-only reporting database, making the reporting database a replica of the 'master' operational database, running on a separate server. The effect achieved by this step heavily depends on DBMS in use and types of load, but usually the heavier the load – the bigger the effect observed (removing inherently heavy reporting requests from an operational database reduces cache/bufferpool poisoning and disk contention, and these effects can hurt performance a lot).

If it doesn't help, our 'No Bugs' Bunny might need to take a closer look at inter-transaction interaction (including transaction isolation levels, `SELECT FOR UPDATE` statements and the order of obtained locks). We feel that if it goes as far as this, he is in quite big trouble. While inter-transaction communication is not exactly multithreading, it has many similar features. In particular, deadlocks or 'dirty reads' can easily occur, eliminating them can be really tricky, and debugging can become extremely unpleasant. If our 'No MT Bugs' Bunny finds himself in such situation, we will consider him being 'Half-Gotchaed'.

One application-level option which might be useful at this point is to start postponing updates (storing them in separate table, or some kind of queue) for certain frequently updated statistical fields (like a 'number of hits' field) to avoid the locking, and move such postponed updates into the main table later, time-permitting or in bulk, reducing locking.



Single connection: back to the future?

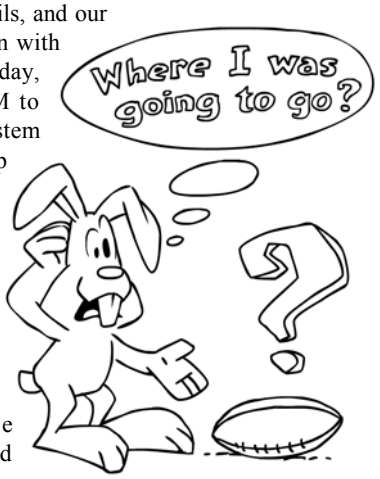
It is worth noting that there is an option to avoid this kind of multithread-like problems altogether, which is rarely considered. It is sometimes possible to move all update statements into a single DB connection (essentially to a single thread); while such approaches are often ostracized for lack of scalability, practice shows that in some environments (especially those where data integrity is paramount with no room for mistakes, for example, in financial areas), it is a perfectly viable approach – the biggest load which we have observed for such single-update-connection architecture was on the order of 30M update transactions per day for a single synchronous DB connection, and when it became not enough, it was (though with a substantial effort) separated into several databases with a single connection for updates to each one, reaching 100M+ update transactions per day (and with the option to go further if necessary).

Divide et impera

If after applying all the optimizations above our our 'No MT Bugs' Bunny still experiences performance problems, his next chance to escape fierce 'Multithreaded Gorrillazz' is to try to find out if the data he works with can be easily classified by certain criteria, and split the single monolithic database into several partitioned ones. For example, if his application needs to handle transactions in a thousand stores worldwide, but most transactions are in-store and only a few interactions between the stores (similar to the task defined in [TPC-C] benchmark), he has a chance of getting away with partitioning the database by store (one or several stores per database), achieving scalability this way. Methods of separation can vary from DBMS-supported partitioning (for example, [IBM06] and [Oracle07]) to application-level separation. Application-level separation can have many varieties (with many being extremely application-specific), and detailed discussion of such separation can easily take a few books, so we will not try to go into more details here.

In-memory state: case of multiple sclerosis

If everything described above fails, and our Bunny indeed has an application with 100M+ update transactions per day, he may need to resort to RAM to remember some parts of the system state, rather than to keep everything in the database. It is a fundamental change, and it won't be easy. One important implication is that all information held in memory only will be lost if system goes down or reboots; in some cases (like caches) it doesn't matter, but if going beyond caches, the implications must be considered very carefully.



Still, even with in-memory state multithreading is not always necessary; it can be avoided either by techniques described in the previous article [Ign10], or by separating the system into a series of logical objects, each having its own in-memory state and incoming queue, and all the logical object input being limited to the processing of incoming messages, with all interaction between objects restricted to sending messages to each other. One of us has seen such a system processing over 1 billion (yes, this is nine zeros) of requests per day, still without any multithreading at the application level (all multithreading has been confined to a few thousand lines of specialized framework, which is 100% isolated from the application-level business logic and therefore is changed extremely rarely). If our Bunny is one of the few very lucky (or unlucky, depending on point of view) ones who really need to process more than 1e9 requests per day – there is a chance he will be gotchaed, but honestly – how many of us are really working on such systems? To set some kind of benchmark: NASDAQ is currently able to process 2e8 transactions per day [NASDAQ], so we can reasonably expect that there are relatively few systems which need more than 1e9. Still, it can happen and we have no choice other than to award a point to Gorrillazz in this case.

No database? No problem

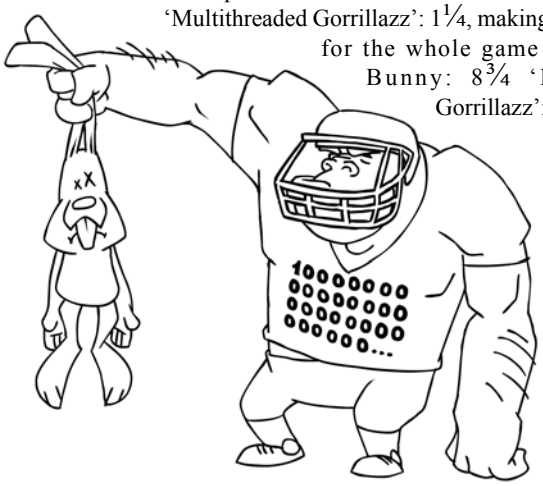
As promised at the very beginning of the article, now we will come back to discussing examples of server-side applications which don't use databases (or use them in a very limited way). One such example is music/video streaming server applications. While such applications don't need to rely on a database, they can be scaled easily enough similar to any other 'data publishing' application (see 'The Clone Bunny' above); in extreme cases where top performance is necessary, using non-blocking I/O techniques can be used to improve performance further.

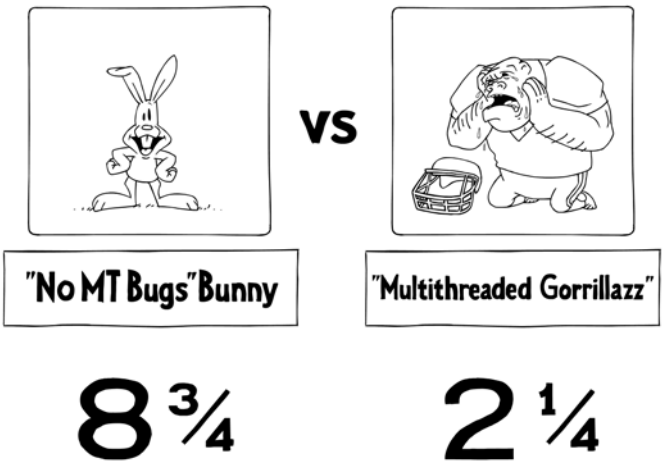
Another prominent example of server-side applications which don't really need to depend on the database, is game servers. While it is very difficult to generalize such a vast field as games in general, massive server-side games usually seem to fit under 'In-Memory State: Case of Multiple Sclerosis' described above, and our 'No MT Bugs' Bunny can try to handle

them using the very same techniques as described there and in previous article.

Quarter 3&4: 'No MT Bugs' Bunny: 4¾ 'Multithreaded Gorrillazz': 1¼

Now as the match between 'No MT Bugs' Bunny and 'Multithreaded Gorrillazz' has came to end, we're able to find out the final score of this magnificent game. As we've seen, similar to client-side, on the server-side there aren't too many cases for multithreading either. Our 'No MT Bugs' Bunny managed to make 9 home runs on the server side of the field, while being gotchaed only once, and being half-gotchaed once. Taking into account the relative weights of these runs, we conclude that quarters 3 & 4 have been completed with a score of 'No MT Bugs' Bunny: 4¾ 'Multithreaded Gorrillazz': 1¼, making the overall score for the whole game 'No MT Bugs' Bunny: 8¾ 'Multithreaded Gorrillazz': 2¼. ■





References

[CWE-567] CWE-567: Unsynchronized Access to Shared Data, Common Weakness Enumeration, http://cwe.mitre.org/data/definitions/567.html

[Facebook08] Scaling memcached at Facebook, Paul Saab, 2008, http://www.facebook.com/note.php?note_id=39391378919

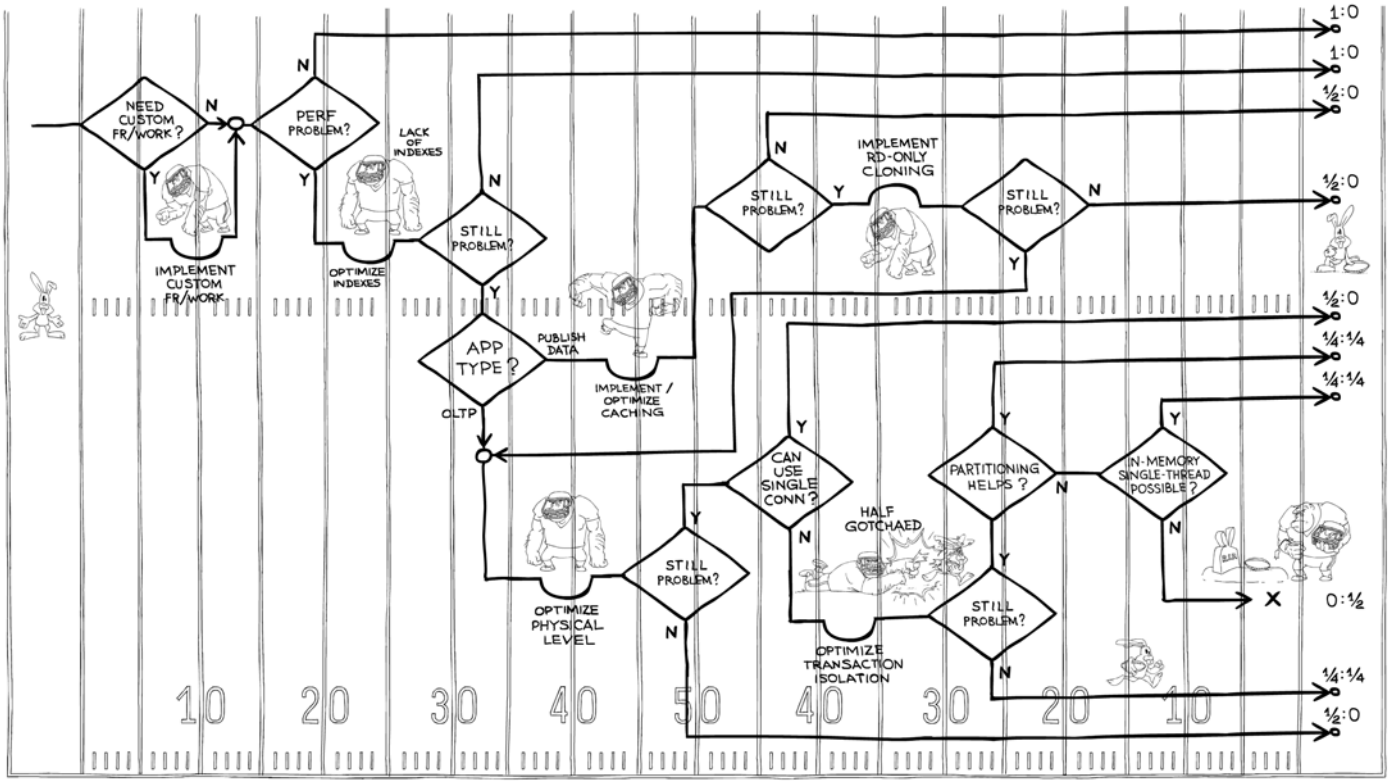
[IBM06] Introducing DB2 9, Part 2: Table partitioning in DB2 9, Rav Ahuja, 2006, http://www.ibm.com/developerworks/db2/library/techarticle/dm-0605ahuja2/

[Ign10] 'Single-Threading: Back to the Future?' Sergey Ignatchenko, *Overload* #97, June 2010

[NASDAQ] Technology Fast Facts, NASDAQ, http://www.nasdaq.com/newsroom/presskit/reports/NASDAQ_Technology_Worksheet.pdf

[Oracle07] Partitioning in Oracle Database 11g, Hermann Baer, 2007, http://www.oracle.com/technology/products/bi/db/11g/pdf/partitioning-11g-whitepaper.pdf

[TPC-C] *Overview of the TPC Benchmark C: The Order-Entry Benchmark*, Francois Raab, Walt Kohler, Amitabh Shah, http://www.tpc.org/tpcc/detail.asp



Debugging Run Time Memory Problems

The C++ Standard is often silent on what happens when things go wrong. Matthew Jones looks under the bonnet to take control.

This article explores some of the the grey areas between the C++ language, the OS and the CPU. All three come into play when memory problems arise, and usually in the form of the dreaded undefined behaviour (UB).

A lot goes on behind the scenes between a system booting, or an application starting in its fresh virtual memory space, and `main()` being called. Most people are happy to gloss over, or even ignore, this area. This is acceptable because it works perfectly almost all the time. Usually we only care that `main()` is called (somehow), memory is allocated when you ask for it (somehow), and everything works just fine. But what happens when things go wrong and the system starts to behave incorrectly? Will we even notice? Will it lead to a crash? How do we go about debugging these problems? Is it even feasible to debug dynamically, or do we give up and start inspecting the source code? We might, and in simple single threaded code this could be sufficient. But if we put some effort into taming, or even defining, our undefined behaviour, we can tell far more from a crash than we might think. With a bit of preparation, we can make UB work for us, rather than baffle us.

The details here are highly system specific, but so are crashes. Every architecture and application is different, but the principles should be widely applicable. Some techniques involve dipping into the C run time initialisation code, and will only be possible in the more embedded (i.e. roll your own) environments. We focus on C++, and some of its peculiar memory problems, but some of the techniques could apply to many other languages. YMWCV (Your Mileage Will Certainly Vary)!

Some of you might notice that a number of the tricks here are based on the Microsoft SDK C run time debug, in particular the memory debug and filling on initialisation [MSDN1]. Even if you are already using that it is useful to understand the details of how and why it works.

Program behaviour

The result of what we do as programmers is mostly defined and specified: we write source code according to a standard [C++], that a compiler converts into object code. The behaviour of that code is well defined and should hold no surprises.

There are a number of other kinds of behaviour, presented here in order of increasing 'danger' to the programmer.

Implementation defined behaviour

Standardised behaviour is not possible because of a dependency on the underlying platform. Word size, alignment and endianness are common cases. Given a known platform or architecture, however, behaviour is defined.

Matthew Jones has been programming for over 20 years, 15 of them professionally. He started with BBC BASIC, then C whilst at VI form. He moved on to C++ and has worked on a variety of large embedded systems. He can be contacted at m@badcrumble.net

Unspecified behaviour

Behaviour that is not specified by the standard, but it is still *good* behaviour. The standard can not make guesses about, or rely on, such behaviour because it depends on the system context. Examples are program performance; storage of data in memory (padding, ordering etc., but not alignment); or the exact instructions a compiler might generate for a given line of code. The actual behaviour can often be changed, for example by altering compiler settings. Even if we do not have control over this kind of behaviour we can at least discover what it is.

Undefined behaviour

When program behaviour depends on factors entirely beyond the control of the language specification, there is no way it can state what should happen outside 'normal' behaviour: it has to be left undefined by the language. For example some hardware will trap certain types of illegal memory access, but other hardware doesn't.

Undefined behaviour is traditionally, and rightly, the most feared kind. But fear usually stems from lack of knowledge or understanding, and we certainly can address these two deficiencies.

It was pointed out on ACCU-General recently [Henney] that on any particular system, with a particular context, UB is *repeatable*:

> With respect to the C and C++ standards, the use of certain constructs can be considered to result in undefined behaviour *with respect to* the relevant language standard. For a given execution of a given program on a given platform, however, such behaviour is normally very well defined.

When we know the system, and can see the assembly code generated by the compiler, we *do* know what will happen, even though the results are probably undesirable. So for a known context (the application we are developing, with our particular compiler, on one particular system) our UB *usually* becomes known, repeatable behaviour. There is still scope for other factors to influence predictability (e.g. multithreading), but in principle we can still attempt to understand and therefore control it.

The side effects of UB can often go unnoticed. Silent failures are very dangerous because the defect leading to the UB has therefore escaped test and will be released from development. It is often the case that a later change to the software or system then changes the context of the defect: the symptoms change for the worse and the failure is no longer silent. One of our objectives should be to ensure that UB *fails fast* [Fowler].

Because of the lack of definition, there is potential for us to step in and not only define the behaviour, but to ensure it is something that suits us. Since UB usually stems from incorrect program operation, any definition we impose should deal with, or be tolerant of, that incorrect operation. We have seen that UB is actually predictable and repeatable: if we choose to define an area of UB, it becomes *user defined*, *repeatable* behaviour triggered by incorrect operation: we can make it provide useful debug information, and turn it to our advantage. I hope to show that is feasible to replace UB with defensive measures that are sufficient to automatically diagnose most memory problems.

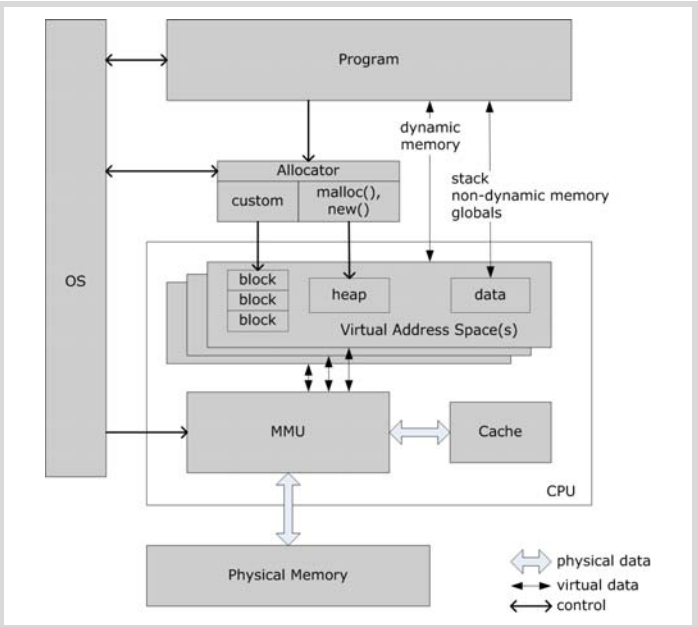


Figure 1

Background

I am assuming a fair degree of familiarity with how CPUs work, how memory is allocated, how the compiler represents objects, and so on. Before exploring the details, this section contains definitions and summaries of the most relevant terms. Most terms used here are explained in depth by Wikipedia [Wikipedia1].

Figure 1 shows the memory features of a generalised system. In the simplest scenarios the program runs within a virtual memory space, but is unaware of this. The built-in `new()` and `delete()` are used to get dynamic storage from the heap. These may or may not interact with the OS and MMU to allocate memory pages. More advanced scenarios might involve a custom allocator, which may use OS facilities to manage memory more directly. More about custom allocators later.

Crashes

The term *crash* is widely used but seldom defined. In general it means some kind of serious, often low level, error that is unrecoverable [Wikipedia2]. A crash is usually followed by the failed system ceasing to respond, or halting. A software application would usually have to be restarted. An embedded system might need to be power cycled. In this article it is used in a non-specific manner to mean incorrect operation leading directly to an abnormal, obvious, non-recoverable state.

MMUs and illegal access

Most systems have some kind of Memory Management Unit. It might not be called by that name, but somewhere in the system there will be a function responsible for tasks such as logical to physical address translation and controlling access. If you have an MMU it will probably signal to the application, or at least the OS, when exceptions such as illegal memory accesses occur. The definition of 'illegal' may vary, but usually includes

- any form of access to unmapped (i.e. invalid) addresses
- writing to program/protected/read only memory
- fetching instructions from non-program memory
- illegal access (application doesn't have permission)

Even if you don't have a dedicated MMU, most advanced/modern CPUs can be configured to convert illegal memory access into some sort of IRQ or CPU exception.

If you have a basic CPU it should still be possible to implement a rudimentary MMU, if you have space in an FPGA or PLD. It could be made to can raise an IRQ to the CPU, and provide a status register or two.

Details here are necessarily vague because this subject is entirely system dependent. But there is no getting away from the need to read and understand your programmer's reference manual. It is no coincidence that the list of illegal operations above is similar to the root causes of Windows GPF/BSoD [Wikipedia3],[Wikipedia4].

Where is everything?

When the linker takes your object files and links them into a library or executable, it can usually be commanded to output a *map file*. This lists symbols and their addresses. Details are specific to the tools used, but the map file or equivalent is an extremely useful tool when debugging memory problems. Given a word of memory, it is possible to deduce what it might be, by referring to the map file, and the system's memory map. The word might fall into one of a number of address ranges, and these indicate likely interpretations of the meaning of that value.

Memory range	Possible Use
ROM, .text, code segment, program area	vtable, function pointer
data segment, .data, .bss	global variable
task stack (*)	automatic variable
RAM, heap	dynamically allocated memory

(*) these might have been allocated from the heap so can be hard to distinguish. In a more static, embedded, system, they might be global data allocated at startup.

Note that if your code is relocatable, a shared library, DLL etc., the addresses in the map file will not be the final addresses. In this case map file addresses will simply be offsets. The final addresses are only known once the library has been loaded.

What is stored in memory?

If we want to decipher the clues left by a memory related crash, we have to know exactly what we're looking at. Memory is just bits, bytes and words, but it is given meaning when interpreted by code. Problems, and sometimes crashes, occur when the code interprets some piece of memory incorrectly. If we can understand *correct* operation, then when we are faced with a crash (incorrect operation) we can look for specific symptoms that might lead us back to a root cause based on *mis*interpretation.

The canonical example is interpreting a number as a pointer, with null pointer the most frequent case. The code is interpreting some memory as a valid pointer, but actually it is *mis*interpreting a number. Whether this was meant to be a valid pointer, or is some other stray value, is moot. If the number is zero, the outcome is at least partly predictable: memory at (or offset from) address 0 will be accessed. If the number being misinterpreted is some other (essentially random) value, the outcome depends on the value in question.

Raw data

When raw data is stored in memory, aside from decoding it, we can't tell much about it. Its appearance is governed by the source code, and there are few compiler generated clues to its nature. However, if it has been allocated we can find out when, by whom, and possibly why. This may give further clues about the content. See later for more on unused/deleted blocks.

Objects and object pointers

To exist at run time, an object is either allocated its own memory from a memory pool, or it is an automatic variable on the stack. Either way, it occupies an area of memory. But what is in that memory?

Every instance of any object must include at least the non-static, data members. These will appear in memory according to platform and compiler dependent rules on data size, ordering, and endianness rules. They will often be separated by padding to satisfy the data alignment rules of either the compiler or the processor.

If there is no polymorphism, the compiler knows the type of the object at compile time. No metadata is required to identify the type and only the

```
class BoringThing
{
public:
    int a;
    int b;
    BoringThing() : a(0), b(1) {}
    void Function () { a = 10; }
};
void boring (void)
{
    BoringThing *b = new BoringThing;
    b->Function();
    // resulting assembly code: note address
    // of function is absolute
    //  lwz r3,0x2C(r31) ; r3,44(r31)
    //  bl  0xA24C       ; BoringThing::Function
}
// Map file extract
  .gnu.linkonce.t._ZN11BoringThing8FunctionEv
     0x0000a24c    BoringThing::Function()
  .gnu.linkonce.t._ZN11BoringThingC1Ev
     0x0000a278    BoringThing::BoringThing()

// Contents of memory at 'b' (0d5267e0):
  _address_|_0_______4_______8_______C_______
   0D5267D0| 00000000 053635F8 00000008 FDFDFDFD
   0D5267E0| 00000000 00000001 FDFDFDFD
```

Listing 1

object's data members need be stored. When we find an object of such a class in memory, we typically see just its data members. It can be very hard to tell that the memory contains an object unless we know in advance. Often the pattern of data can be a clue but we can't assume that there will be anything definite to go on. Listing 1 shows an example of a trivial object and the resulting map and memory layout.

If polymorphism is involved, when the code is running and wishes to access an object, it has to find out the actual type of the object. The code therefore has to store in memory with the data members some information that identifies the object's type, initialised when the object is created. When virtual functions are called, the object is accessed and the code first reads this key to determine what type it is, and therefore exactly how to access it. Without this the code doesn't know what methods could be called, or even what data members the object might have. To meet this requirement gcc [gcc] and many other compilers use the address of the object's virtual function table (vtable) as this identifier, and it is stored in memory with the data members. Every class that has virtual functions has its own vtable, and conversely the vtable uniquely identifies the class. The vtable is essential for the compiler: it defines the specific overridden function to call when the code calls a virtual function on objects of that class.

The fact that the vtable pointer is stored with the object is extremely useful: given any pointer to memory, if we know that it is a (polymorphic) object, we simply find the vtable pointer, then look in the map file produced by the linker, and thereby divine the run time type of the object. If you are debugging memory problems, get familiar with your map file! It tells you most of what you need to know.

Given an unknown chunk of memory, if we spot a word that appears to be an address in program memory, that word really ought to be a pointer to a function, either global or a member of some class; or a pointer to a vtable. If we see the latter we are probably looking at an object, and can quickly confirm this by looking in the map file. Note that this relies on 'program memory' being contiguous, and being an identifiable region of memory. See the later section about controlling your memory map.

Listing 2 shows a trivial class hierarchy and a function to exercise instances of the classes. Listing 3 shows the resulting map file, Listing 4 the virtual function tables, and Listing 5 the contents of the pointers in Listing 2.

```
class Thing
{
public:
    int a;
    int b;
    Thing (): a(0), b(0) {}
    virtual ~Thing() {}
    void Apple ();
    virtual void Banana () { a = 1; }
    virtual void Cherry () { b = 2; }
};
class DerivedThing : public Thing
{
public:
    virtual void Banana () { a = 2; }
};
class MoreDerivedThing : public DerivedThing
{
public:
    virtual void Cherry () { b = 3; }
};
void foo (void)
{
    Thing * t = new Thing;
    DerivedThing * dt = new DerivedThing;
    MoreDerivedThing * mdt
       = new MoreDerivedThing;
    Thing * t1 = new Thing;
    t1->Banana();
    t1->Cherry();
    Thing * t2 = new DerivedThing;
    t2->Banana();
    t2->Cherry();
    Thing * t3 = new MoreDerivedThing;
    t3->Banana();
    t3->Cherry();
}
```

Listing 2

Note that the memory blocks shown in Listing 5 have some of the debug features described later: guard blocks (`FDFDFDFD`), inter-block filling (`0x31`), and the word before the start block is the block size. Each pointer points to a block that starts with the object's vtable. It is interesting to compare the type of the vtable with actual type of each pointer. Note that due to the mysterious ways of gcc 3.4.3, each vtable starts with two empty words, then two distinct (but identical) destructors. The pointer stored with each object instance is always to the first destructor, not the start of the table.

Common types of memory fault

In this section a number of familiar memory faults are introduced. One of the simplest memory faults is to leak allocated memory. This is such a widely experienced problem that it does not warrant its own section here.

Later we will discuss a number of techniques to defend against these faults, or to analyse their aftermath.

Null pointer access

Because null pointers are usually represented by the value zero, they inadvertently point to the memory at address 0. They are not meant to be dereferenced, but when they are, the code inevitably reads or writes to the memory at or near address 0.

When a piece of code dereferences a null pointer, the first thing that will happen is the compiler generated assembly code will read a word from address 0 onwards. If this in itself does not trigger an exception, the word will be interpreted.

If the pointer is to raw data, the outcome is application specific. If the pointer is to an object, the contents of memory at 0 will be assumed to be

```
.gnu.linkonce.t._ZN5Thing5AppleEv
  0x0000a470    Thing::Apple()
.gnu.linkonce.t._ZN16MoreDerivedThingC1Ev
  0x0000a490
  MoreDerivedThing::MoreDerivedThing()
.gnu.linkonce.t._ZN12DerivedThingC2Ev
  0x0000a4d8    DerivedThing::DerivedThing()
.gnu.linkonce.t._ZN5ThingC2Ev
  0x0000a520    Thing::Thing()
.gnu.linkonce.t._ZN12DerivedThingC1Ev
  0x0000a568    DerivedThing::DerivedThing()
.gnu.linkonce.t._ZN5ThingC1Ev
  0x0000a5b0    Thing::Thing()

.gnu.linkonce.t._ZN5Thing6BananaEv
  0x0000b28c    Thing::Banana()
.gnu.linkonce.t._ZN5ThingD0Ev
  0x0000b2b8    Thing::~Thing()
.gnu.linkonce.t._ZN5ThingD1Ev
  0x0000b314    Thing::~Thing()
.gnu.linkonce.t._ZN5Thing6CherryEv
  0x0000b370    Thing::Cherry()
.gnu.linkonce.t._ZN12DerivedThingD0Ev
  0x0000b39c    DerivedThing::~DerivedThing()
.gnu.linkonce.t._ZN5ThingD2Ev
  0x0000b400    Thing::~Thing()
.gnu.linkonce.t._ZN12DerivedThingD1Ev
  0x0000b45c    DerivedThing::~DerivedThing()
.gnu.linkonce.t._ZN16MoreDerivedThing6CherryEv
  0x0000b4c0    MoreDerivedThing::Cherry()
.gnu.linkonce.t._ZN12DerivedThing6BananaEv
  0x0000b4ec    DerivedThing::Banana()

.gnu.linkonce.t._ZN16MoreDerivedThingD1Ev
  0x0000ba90
  MoreDerivedThing::~MoreDerivedThing()
.gnu.linkonce.t._ZN12DerivedThingD2Ev
 0x0000baf4    DerivedThing::~DerivedThing()
.gnu.linkonce.t._ZN16MoreDerivedThingD0Ev
  0x0000bb58
  MoreDerivedThing::~MoreDerivedThing()

.gnu.linkonce.r._ZTV16MoreDerivedThing
  0x00391e88    vtable for MoreDerivedThing
.gnu.linkonce.r._ZTV12DerivedThing
  0x00391ea0    vtable for DerivedThing
.gnu.linkonce.r._ZTV5Thing
  0x00391eb8    vtable for Thing
```

Listing 3

the pointer to the vtable for the object. Depending on which virtual function the code intends calling, the bogus vtable will be indexed into and the function pointer in question will be read, and then jumped to. Depending on the data, the address might be valid, data that looks like a valid address, or completely invalid. If you are lucky, the jump will be to an illegal address and the MMU will step in.

If the pointer is being used to access object data, the memory near address 0 will be read and misinterpreted.

Address 0 is the start of the *zero page* [Wikipedia5]. The zero page has special meaning in some systems, and might not be 'normal' memory. What is at address 0 in your system? You should familiarise yourself with it, because it defines what happens when null pointers are accessed. Often its some sort of interrupt or exception table. It might be an invalid address, and the MMU will already be catching illegal access. If reading from/near address 0 does get past the MMU, what are the symptoms? If we are reading ROM (fixed data, code, etc.) we get very obscure, but *completely repeatable* tell-tale values. It took me a long time to understand that when

```
0x00391e88    vtable for MoreDerivedThing

_address_|__________
 00391E88| 00000000
 00391E8C| 00000000
 00391E90| 0000BA90
MoreDerivedThing::~MoreDerivedThing()
 00391E94| 0000BB58
MoreDerivedThing::~MoreDerivedThing()
 00391E98| 0000B4EC    DerivedThing::Banana()
 00391E9C| 0000B4C0    MoreDerivedThing::Cherry()


0x00391ea0    vtable for DerivedThing

_address_|__________
 00391EA0| 00000000
 00391EA4| 00000000
 00391EA8| 0000B45C
DerivedThing::~DerivedThing()
 00391EAC| 0000B39C
DerivedThing::~DerivedThing()
 00391EB0| 0000B4EC    DerivedThing::Banana()
 00391EB4| 0000B370    Thing::Cherry()


0x00391eb8    vtable for Thing

_address_|__________
 00391EB8| 00000000
 00391EBC| 00000000
 00391EC0| 0000B314    Thing::~Thing()
 00391EC4| 0000B2B8    Thing::~Thing()
 00391EC8| 0000B28C    Thing::Banana()
 00391ECC| 0000B370    Thing::Cherry()
```

Listing 4

```
t (0D5267E0):
_address_|_0_______4_______8_______C_______
 0D5267D0| 00000000 053635F8 0000000C FDFDFDFD
 0D5267E0| 00391EC0 00000000 00000000 FDFDFDFD

dt (0D5267AC):
_address_|_0_______4_______8_______C_______
 0D5267A0| 053635F8 0000000C FDFDFDFD 00391EA8
 0D5267B0| 00000000 00000000 FDFDFDFD 31313131

mdt (0D526778):
_address_|_0_______4_______8_______C_______
 0D526770| 0000000C FDFDFDFD 00391E90 00000000
 0D526780| 00000000 FDFDFDFD 31313131 31313131

t1 (0D526744):
_address_|_0_______4_______8_______C_______
 0D526740| FDFDFDFD 00391EC0 00000001 00000002
 0D526750| FDFDFDFD 31313131 31313131 31313131

t2 (0D526710):
_address_|_0_______4_______8_______C_______
 0D526700| 00000000 053635F8 0000000C FDFDFDFD
 0D526710| 00391EA8 00000002 00000002 FDFDFDFD

t3 (0D5266DC):
_address_|_0_______4_______8_______C_______
 0D5266D0| 053635F8 0000000C FDFDFDFD 00391E90
 0D5266E0| 00000002 00000003 FDFDFDFD 31313131
```

Listing 5

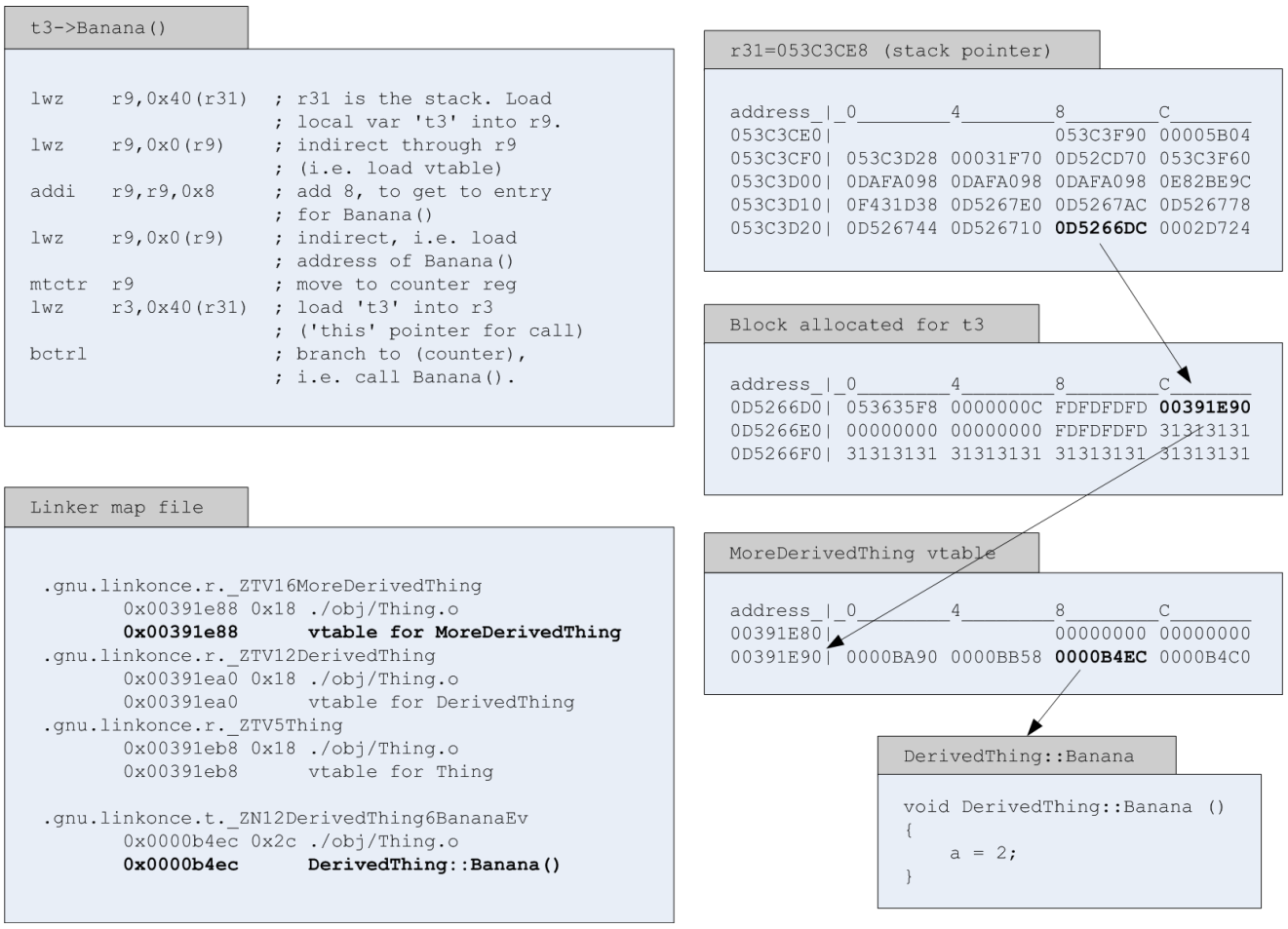


Figure 2

I *kept on seeing* an exception saying that an instruction was being read from an invalid address, **0x9561FFFC**, it was because of this:

```
_address_|_0_____4_____8_____C_____
 00000000| 3821FFF8 91810000 9561FFFC 9541FFFC
 00000010| 9521FFFC 7D9A02A6 9581FFFC 7D9B02A6
 00000020| 9581FFFC 7D9A0AA6 9581FFFC 7D9B0AA6
 00000030| 9581FFFC 39600000 9561FFFC 7D4902A6
 00000040| 3D200002 61292EEC 7D2903A6 4E800420
 00000050| 3821FFF8 91810000 9561FFFC 9541FFFC
```

The seemingly random value was actually one of the words near address 0. Always check suspect values against the memory contents near address 0. See later for a way to prevent this.

To understand the details of a null object pointer, let's look at the sequence of events during a normal virtual function call. If we consider a single line of Listing 2:

```
t3->Banana();
```

then the compiled assembly code and its interpretation are shown in Figure 2.

Accessing deleted memory

Accessing deleted memory is a very grey area. It depends not only on the compiler, but on the memory allocation strategy and the dynamics of the system.

The least harmful case is that the memory has not changed since de-allocation, and is therefore effectively valid. This will pass unnoticed. In some respects a silent failure is the worst possible outcome, because we don't fail fast.

If the memory has been re-allocated it begins a new life that the original code is unaware of. If the memory is written to by the new owner, and the original code reads it, this new data will be misinterpreted. Then depending on the nature of the new data, and what the old code does with it, it might lead to a crash or less severe erroneous behaviour. It is equally disastrous if the original code writes to the memory, and the new owner then misinterprets it. Again, the worst possible outcome is that this goes unnoticed. If the side effects appear much later, the original code might have moved on from abusing the re-allocated block, and the trail will have gone cold when the problem is investigated.

In some systems de-allocated blocks, or the memory pages containing them, are given back to the OS, to be protected by the MMU. In this case access should immediately trigger an MMU exception. Here the architecture is already helping us and there is nothing to add.

Note that if the deleted memory contained a polymorphic object, the vtable pointer will have been changed to that of the base class when **delete()** worked its way through the hierarchy of destructors. If the memory is unaltered, using the vtable as a clue about the type of the object could be misleading.

Off the end of an array

The effect of writing off the end of an array depends on memory alignment, memory allocation strategies, the nature of the array, and the purpose of the memory surrounding the array. For example if the array is an odd number of bytes in length, and memory is allocated on word boundaries, there will be a small number of empty padding bytes after the array. Writing over these (and then reading them back) has no noticeable effect, and will be a silent failure. If we write far off the array, or it adjoins another allocated area, then other data will be overwritten. If this is accessed later, it might be misinterpreted.

```
    char * x = new (10);

For block size 18 (0x12):

_offset__|__0__1__2__3__4__5__6__7_
00000000 | GG GG GG GG  00 00 00 00
00000008 | 00 00 00 00  00 00 GG GG
00000010 | GG GG
```

Listing 6

Since the side effects depends on so many factors, they can be very hard to spot, and will often fail silently. We will look at ways to make if fail fast later.

Defence

Once we understand the nature of problems that stem from memory abuse, we can start to defend ourselves against them. Here are a variety of tried and tested techniques.

Tools

You should always start by attacking any problem with tools. If you are not using some form of static analysis tool as you write your code, you are leaving yourself open to the full spectrum of human fallibility. Lint will spot obvious memory leaks, null pointer use and so on, the moment you compile your code. Most systems come with some form of run time analysis tool: perfmon, UMDH (windows); valgrind (*nix).

Some of the techniques described next might replace or even interfere with mechanisms employed by COTS tools. You must investigate, understand, and experiment with your environment before committing to using them.

Block allocator

Anyone can write a block allocator, or get one off the web. A quick Google for 'memory block allocator' or 'memory pool allocator' will start you off.

When a block allocator is available, it is easy to override the global operator `new()` and `delete()` so that it is used in preference to whatever heap allocator is provided by the language support library. A further improvement is to provide dedicated allocators for, and override the allocation operators of, specific classes that are in some way interesting (frequently allocated, troublesome etc.).

Once memory allocation is under our control (rather than part of the hidden, built in, language support) we can start to instrument it, and add defensive features.

If you can build the language support library yourself, it is possible to add everything discussed here to the built in `new()` and `delete()`.

Guard blocks

The most common addition is a pair of guard words (or bookends, goalposts, etc.). When the application asks for a block of memory we allocate more than was asked for, and place a guard word at the start and end. The guard words surround the exact number of bytes asked for, not the entire block, because this may well be larger than was asked for. The pointer returned to the caller is to the memory just after the first guard (highlighted in Listing 6).

Now we can check the block for corruption: if the application writes off the end of the block (or off the front), the guard word is altered, and this can be detected. Normally the guard blocks would be checked in operator `delete()`, i.e. when the block is de-allocated, but this is often too late. To be more proactive, the list of currently allocated blocks can be periodically checked by a background task or on demand when certain conditions arise. The value of the guard block is not particularly important, but it would be foolish to use 00! It is worth ensuring that it can not be misconstrued as a valid address, just in case faulty code reads it as real data.

If we add a guard area at the end of our block, we can catch the most common error, off by one, without it damaging anything else.

Metadata

It is often useful to know more than just the address of an allocated memory block. If we reserve extra space in the block, we can store metadata such as:

- time of allocation (not usually very interesting in itself)
- allocating thread/task/process
- size asked for (this may be different from the size allocated if the blocks are fixed sizes)
- source file and line of code.[1]

This data will normally be ignored, but when something goes wrong, post mortem analysis (whether manual or automatic) can put it to good use. It can't be added after the incident, so you must pay the price of putting it in place for every block, every time.

Fill values

A surprisingly useful trick is to fill every block with a tell tale value before giving it to the application. The fill value should be obvious, and as ever, should not look like a valid address when consecutive bytes are taken to be a pointer. Although this impacts performance, the price is usually worth paying due to the obvious benefits:

- Using uninitialised data. If the block is not filled, the previous contents, or fresh 00s, will be read, resulting in random behaviour, and possibly silent errors. The best culprit here is uninitialised pointers: if the fill value is an invalid address, as soon as the pointer is dereferenced, the MMU will step in and raise an exception.
- Unused blocks. If the fill values are still there in `delete()` the block was not used. Why?
- Under-used blocks. If a 'tide mark' is visible in the block, why is the application using less of the block than it asked for?

When the block is deallocated, it should be filled with a second, distinct, value. This allows access of deleted data to be caught (until the block is reallocated). Again the fill value should not look like a valid address, so that the MMU can catch illegal pointer dereferencing.

Note that the Microsoft Developer Studio C++ compiler does this in Debug mode [MSDN1].

Tie down the memory map

If your environment allows you control of the linker script and/or memory map, it is possible to define your memory map in the most defensive manner possible. This ensures maximum scope for help from the MMU.

Arrange program sections with different access criteria (read, write, execute) to be grouped together, and to be on boundaries and/or page sizes that the MMU can control independently. The aim is to restrict access as far as possible, and to be as strict as you can.

Once the program has been loaded into RAM and the data section is initialised, but before `main()` is called; make the pages containing the program and data sections effectively ROM by restricting memory access to execute only and read only respectively.

Ensure that the zero page has no write access, regardless of its contents, so that writing to null pointers is caught. If possible, make the zero page completely inaccessible, at the MMU level, to catch all null pointer action.

The ideal system would have access rights as follows:

	r/w/x (read write execute)
ROM	r
.rodata	r
.text	x
.data	r
.bss	r/w
RAM	r/w

1. This will involve turning `new()` into an evil macro involving `__FILE__` and `__LINE__`

Fill the zero page with known illegal values

We have already seen that null pointers are intimately connected with the contents of memory at address 0, and the confusion that arises from misinterpreting it. If the context allows it, reserve the first 256 bytes or so, and fill it with with a tell-tale value which, when misinterpreted via a null pointer, is an illegal addresses. The aim is to trigger an MMU exception as soon as the pointer is dereferenced. This is what mine looks like:

```
_address_|_0________4________8________C________
 00000000| BAD00000 BAD00000 BAD00000 BAD00000
 00000010| BAD00000 BAD00000 BAD00000 BAD00000
 ...
 000000E0| BAD00000 BAD00000 BAD00000 BAD00000
 000000F0| BAD00000 BAD00000 BAD00000 BAD00000
 00000100| 3821FFF8 91810000 9561FFFC 9541FFFC
 00000110| 9521FFFC 7D9A02A6 9581FFFC 7D9B02A6
```

Fill the zero page with pointers to a debug function

An extension to the previous technique is to fill memory after 0 with *valid pointers*, pointing to a debug function which raises an exception within the application. At the very least it should print "null pointer" and call `exit()`.

The drawback with this is that reading raw data from near zero will now read a valid address and possibly fail silently. The address of the debug function, although generated by the linker, is still a tell-tale value and should be kept in the back of the mind when investigating memory faults.

MMU exceptions should dump all pertinent registers

Most MMUs have a number of status registers which can be read when an access violation is detected. One of them will be the address which triggered the exception. All pertinent registers should be saved and made accessible to debug code, the user, etc. Armed with this data it is possible to produce very useful messages to the user. There are some examples later.

Run time checks

If you have instrumented or protected your memory blocks as described earlier, a background task can periodically analyse all blocks for integrity. This introduces the possibility of finding errors while objects are still alive, shortly after the fault occurred; rather than when they are deleted, which may be far too late, or even never.

Pool analysis

If a memory leak has evaded static analysis, it must be caught at run time. It is feasible to stop the program and analyse the heap, either by inspecting the memory directly (e.g. using a debugger), or by calling a dedicated function. Microsoft's Developer Studio IDE performs memory block analysis to detect memory leaks in Debug mode [MSDN2].

Now that we know that any block of memory containing an object will probably start with a pointer to that object's vtable, one approach is to simply walk the list of allocated blocks, summing the instances of each vtable. By counting these, we are counting instances of each object type. If we are leaking objects, then over time one object count will have an obvious growth trend.

A more involved technique is to use a series of fixed size block allocators and iteratively tune the block sizes to narrow down the size of object, or piece of data, being leaked. Eventually this will lead to a short list of objects which can be debugged directly.

Analysing your memory pool usage can be very illuminating, whether in response to a problem, or simply as a background activity. The analysis can be as detailed as your imagination allows, but bear in mind any intrusion on a running system. Analysis is possible with any memory allocation strategy, although it is easier if you use an allocator that you control. If you want to analyse the built-in C or C++ heap you must first find out how it works.

Recalling the earlier section about how objects are stored in memory, a very interesting analysis is to examine the first word of every block. This gives us a picture of what is going on in our block. Below is a real (but slightly contrived) example, where **`FDFDFDFD`** is the memory block guard, **`AA`** is the allocate fill value, and **`DD`** is the delete fill value.

```
Heap at 0D3B003C:
  Data      Size    Count
  ----------------------
  00391E90     C    27760(1)
  00391EC0     C     1236(2)
  00391EA8     C      502(3)
  003C2FC0    1C      176(4)
  FDFDFDFD     0       54(5)
  42537472    14        2(6)
  ...
  AAFDFDFD     1        1(7)
  ...
  AAAAAAAA    20        1(8)
  ...
  00FDFDFD     1        1(9)
  ...
```

1. A lot of `00391E90`'s. If we look back to the virtual function call example above, we'll see that this is `0x8` into the vtable for `MoreDerivedThing`, so we have 27760 instances of this object.
2. By the same analysis, 1236 instances of `Thing`.
3. And 502 `DerivedThings`.
4. 176 instances of some other class, whose vtable is near `003C2FC0`.
5. Why are we allocating 54 arrays of size 0? Something to investigate.
6. Looks like ASCII, or some other data: its certainly not an object because interpreting that word as an address results in an area of memory that is not the program (i.e. it is not within the linker `.text` section).
7. A chunk of data, of size 1, that's been allocated but not used yet. We can see one byte of `AA` (allocated marker) and 3 of the 4 bytes of the trailing guard area (`FDFDFD`).
8. The first word of a chunk of data, size 0x20, that hasn't been written to yet.
9. Very probably a zero length string: 1 byte has been requested, and already set to 0, the string terminator. Worth investigating why the application is asking for 0 length strings.

Diagnostic library

This set of diagnostic tools can be gathered together into a library allowing it to be selectively linked (or dynamically loaded) with the application only when needed. This avoids the performance impact of some techniques (the block allocator in particular), but means the ability to diagnose problems is a conscious decision. With unrepeatable faults it may be too late.

Examples

This section shows a number of real world examples. They are typical failure scenarios, and all have been automatically caught by one of the techniques described here. Each example shows the output from my own MMU exception handler. All five of the CPU's exception registers are dumped just in case, but SRR0 (Save/Restore Register) and DEAR (Data Exception Address Register) are usually the only two that are directly relevant. In later examples only the important lines are shown.

Accessing deleted data

```
>>> Unhandled Data Storage IRQ   <<<
>>> current task: 'PrxS_1', time: 2010 JUN 17
 17:54:15   <<<

SRR0  : 0010F9FC <- likely instruction which
caused the problem
SRR1  : 00028000
CSRR0 : 00029EB4
CSRR1 : 00000000
DEAR  : DDDDDDDD <- data exception address
```

Here we can see that the instruction at `0010F9FC` attempted to read or write data at address `DDDDDDDD` (our fill value for deleted memory). The

data address has not been modified, so this is probably a pointer to data, not an object.

```
>>> Unhandled Data Storage IRQ   <<<
 SRR0   : 002BDF08
 DEAR   : DDDDDDE1
```

In this variation the data address is `DDDDDDDD + 4`, so we are probably looking at code that is indexing from a deleted data or object pointer. Examination of the assembly code at `002BDF08` would confirm this.

Corrupt pointer

```
>>> Unhandled Data Storage IRQ   <<<
 SRR0   : 00381154
 DEAR   : 4E495077
```

This shows what is effectively a random data address, but one that luckily *was* invalid, causing an MMU exception. We would suspect a pointer that has been scribbled over by other data. We can look at the code at address `00381154` and work out which pointer, and what it was trying to do.

Null pointer read

```
>>> Unhandled Data Storage IRQ   <<<
 SRR0   : 0035312C
 DEAR   : BAD00030
```

Here we can see that the fill value for the zero page (`BAD00000`) has caused the null pointer to indirect to an invalid address, and `0x30` was added to the pointer before indirection.

Null pointer write

```
>>> Unhandled Data Storage IRQ <<<
 SRR0 : 00018D58
 DEAR : 00000000
```

In this example a null pointer was written to. In fact the bug was that `memcpy()` had been passed a null pointer. So the tell-tale values at address 0 were never read: the code simply tried to write to address 0, triggering an MMU exception.

Corrupt function pointer

```
>>> Unhandled Instruction Storage IRQ   <<<
 SRR0   : 08E12C40
 DEAR   : 0E0C0E8F
```

Note that the exception type is instruction, not data. This is evidence of an invalid function pointer, which was still pointing to valid memory, just not to a section of code. The instruction address that the CPU attempted to read was in an MMU page that was not marked as executable. This immediately triggered an instruction storage exception.

Bad pointer leading to an illegal instruction

```
>>> Unhandled Program IRQ   <<<
 SRR0   : 00BAD000
 DEAR   : 7D35CF83
*(SRR0): FFFFFFFF
```

Again, note the different exception type. This exception is triggered by an illegal instruction, in this case `FFFFFFFF`. The tell-tale value in `SRR0` looks like it came from the zero page, but is not correctly aligned. This implies some bad pointer manipulation, possibly unchecked C style casting in the code. An invalid function pointer has been created somehow, which itself was a valid address, but pointing to program memory that did not contain valid code.

Corrupt memory block

This example shows how operator `delete()` can be used to check the integrity of every returned block. The diagnostic output dumps the head and tail of the block for immediate analysis. We ca n see that the block size was `0xBC`, and when we look for the guard block at this offset, we find `30FDFDFD` rather than `FDFDFDFD`. Therefore the code has written off the end of the block. Due to blocks being allocated on word boundaries, this error would not normally have been found.

```
>>> AllocateError(): Delete() failed due to heap
corruption. Allocator = Block5, param = 161B1FAC
<<<

161B1FA0 0DB01F4C 000000BC FDFDFDFD 47432110
161B1FB0 0001F0B5 4C612C20 4C612C20 4C612C20
161B1FC0 4C612C20 4C612C20 4C612C20 4C612C20
...
161B2050 4C612C20 4C612C20 4C612C20 4C612C20
161B2060 32333435 36373839 30FDFDFD DDDDDDDD
161B2070 DDDDDDDD DDDDFDFD FDFD0000 00000000
161B2080 00000000 00000000 00000000 00000000
```

It is interesting to note that the line starting at `161B2070` shows archaeological evidence of an earlier use of the block: we can see another terminating guard block that was not corrupted, and the deleted block fill value, `0xDD`.

Conclusion

The overriding theme of this article is to take control of as much of your UB as your system allows. If 'U' is replaced by 'D(efined)', the 'magic stuff' under the bonnet can be tamed and turned to your advantage.

With careful preparation of your memory map you can force the run time system to tell you more about crashes than it normally would. If you can control the MMU (or equivalent), you can force access errors to be flagged rather than hidden. Subtle problems which might go unnoticed can be made to fail fast, and crashes can even diagnose themselves.

By taking control of, and instrumenting, memory allocation, you can prepare yourself for a number of failure scenarios, and provide very useful run time analysis of memory usage and leaks.

Know your enemy (memory contents) and, like a good boy scout, be prepared.

This article is a summary of personal experience: I have used nearly every technique that I describe. It is my toolbox. It is not a survey of the state of the art. If you have comments or better techniques, please get in touch, start a discussion on accu-general, or better still, write a follow-up article. ■

Acknowledgements

Thanks to my colleagues (myself included) for producing a selection of interesting bugs which lead to the development of these techniques. Thanks also to Ric 'Overlord' Parkin for considerable editorial help.

References

[C++] C++ standard: ISO/IEC (2003). ISO/IEC 14882:2003(E): Programming Languages – C++

[Fowler] http://www.martinfowler.com/ieeeSoftware/failFast.pdf

[gcc] http://gcc.gnu.org/

[Henney] Kevlin Henney, ACCU general: http://lists.accu.org/mailman/private/accu-general/2010-June/021847.html

[MSDN1] http://msdn.microsoft.com/en-us/library/Aa260966 (table 1)

[MSDN2] http://msdn.microsoft.com/en-us/library/e5ewb1h3%28VS.80%29.aspx

[Wikipedia1] http://en.wikipedia.org/

[Wikipedia2] http://en.wikipedia.org/wiki/Crash_%28computing%29

[Wikipedia3] http://en.wikipedia.org/wiki/GPF

[Wikipedia4] http://en.wikipedia.org/wiki/BSoD

[Wikipedia5] http://en.wikipedia.org/wiki/Zero_page

Quality Matters: The Worst Form of 'Error' Handling Except For All The Others

Dealing with errors is a vital part of good programming. Matthew Wilson specifies a taxonomy.

The next two instalments of *Quality Matters* document another deconstructionist taxonomic journey into the essence of software development, the subject being exception handling. Along the way I consider arguments from both sides of the exception debate – those that think they're a curse, and those that think they're a godsend – and come to the not-entirely-unpredictable conclusion that exceptions are a curse and a godsend. To paraphrase Winston Churchill, *exceptions are the worst form of 'error' handling, except for all the others*.

I shall postulate that much of what is seen as wrong with exceptions arises from them being used for things for which they're unsuitable, albeit some of these misuses are necessary given the syntactic limitations imposed by some languages. I identify four types of actions/circumstances in which exceptions are used, suggest that only two of these are appropriate, and argue that we could all get a little more godsend and a little less curse if programmers, particularly language and library designers, would be more circumspect in their application of what is a very big hammer.

I consider some effects of exceptions on the intrinsic characteristics of software quality, in particular with respect to correctness/robustness/reliability [QM-2], and discoverability and transparency [QM-1]. I'll consider how exceptions affect other aspects – the (removable) diagnostic measures and applied assurance measures – in later instalments.

As by-products of this latest expedition into nomenclatural contrarianism, I rant lyrical about the abject mess that is .NET's near-unworkable definition and classification of exceptions, lament quizzical about Python's iterable mechanism, and wax nostalgic for the simplicity of C's (near) total absence of hidden flow-control mechanisms. I also have a go at the logical contradiction of C++'s `logic_error` exceptions, and, surprisingly – most surprising to me! – largely change my mind about Java's checked exceptions.

Introduction

Having dedicated all my scripting language attention to Ruby over the last five years, I'm currently refreshing my Python knowledge. I'm reading a nice book on Python 3, by an accomplished Python expert, published by a quality publisher. I'm not naming it, however, since I'm going to be criticising specific aspects of the book, rather than providing a proper review, and I don't want to negatively affect the sales of what is otherwise a good book. (It's blessed with a breadth and concision I've never achieved in any of my books so far!) Furthermore, some of the criticisms are of the language, rather than the book's examples or philosophy.

The problems I'm having with the book's content, and with Python itself, include:

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

```
import sys
try:
  n = sys.argv[1]
  y = int(n)
except IndexError:
  print 'USAGE: Listing1.py <number>'
except ValueError, x:
  print x, "in", n
```

Figure 1

- the use of subscript-out-of-range exceptions as a 'convenient' way of avoiding an explicit bounds check when accessing `sys.argv[1]` (see Listing 1, which is a much-simplified Python 2 equivalent)
- the use of exceptions for 'regular' flow-control, e.g. synthesising three-level break semantics (in a case where a worker function would have been simpler and clearer)
- that iterators indicate (to the Python runtime) that they are complete by throwing an instance of `StopIteration`.

There's actually nothing unequivocally *wrong* per se with the program in Listing 1. But I would ***never*** write code like that. (It may be that the author was being pedagogical and would never write code like that either, but that's not indicated in the text, so I am compelled to take him at face value as advocating this coding style.)

The code relies on `IndexError` being thrown if the user fails to specify a command-line argument, whereby `sys.argv` contains only the program name. I have two problems with this. First, such a situation could hardly be called an exceptional condition. It's not only anticipated, it's overwhelmingly likely to happen many times in the lifetime of such a program. Handling it as an exception at once muddies the waters from truly exceptional conditions (such as failing to be able to create a file), and also moves the handling of such an eventuality far away, in terms of the logical flow of a program, from where it is to be detected and, in my opinion, should be handled.

In a larger program, this would mean a large amount of code between the access(es) of `sys.argv` and the requisite `catch` clause. That would not be transparent, and would be very fragile to maintenance change. In a non-trivial program, you might imagine, as would I, that there would be many causes to issue a `USAGE` message. In which case, one would be likely to implement a separate consolidating `usage()` function (which may also call `exit()`). In such a case, it's hard to argue that catching `IndexError` is preferable to simply testing `len(sys.argv)`.

The second issue is of much more moment. Exceptions such as `IndexError` are (if not wholly, then at least in part) used to detect mistakes on the part of the programmer. Coupled with the fact that almost any non-trivial program will contain multiple array indexed accesses, interpreting all `IndexErrors` to mean that the user has forgotten their command-line arguments is drawing a very long bow. (Strangely, the full example in the book does indeed contain multiple array accesses!)

I suggest that we need a **new vocabulary**, and am (humbly?) proposing one here

If, as I hope, you agree that even this small example reveals, and is representative of, a troubling misapprehension as to the use of exceptions in programming, read on. If you don't, well, read on anyway, as I plan to change your mind.

In this instalment (and all subsequent instalments), I will be eschewing the use of 'error'. It is a term, like 'bug', that has been overloaded to the point of being more hindrance than help for discussing software behaviour. (There is a useful definition that will be introduced when we discuss *contract programming*, but it's probably not the one that tallies with most programmers' understanding. Until that time, any use of the term 'error' will be presented in quotes as a reminder that it's vague, unspecific, and largely unhelpful.)

Groundwork

In my own work I've experienced more than one bout of doubt about exceptions. In the next instalment, I'll describe three concrete cases to act as material for the discussion. Before that, though, I need to establish new terminology to deal with our 'error' issue. And before that, I'm going to establish some stereotypical positions that one encounters – some more than others – in the wider development community. And before that … nope; just kidding.

Execution states and actions

Every program has a purpose. Much of the intention and, usually, much of the programming effort is spent to define the program to achieve that purpose. It is reasonable to say that the purpose should accord with the intentions, and the expectations, of its users.

Consider the following four use scenarios of a word-processing program.

Scenario A: A user kicks off an instance of the program – the process – and waits for it to be loaded by the operating environment. Once it's loaded and ready to use, he/she writes some text, formatting as necessary until a point at which he/she decides to save or discard the work. We'll assume the choice is to save. The save is successful. The user closes the process. All that was intended to be done has been achieved. All is right with the world.

Scenario B: When the user elects to save the file-system rejects their request, due to perfectly reasonable causes: the disk is full, or they do not have the requisite permissions, or the network is down, and so on. What should the process do? Presumably we would like it to detect that the file cannot be written and take some action, rather than giving the tacit impression that all that was intended to be done has been achieved when that is not the case. That action might be to clear out any temporary files it was using to cache information, and retry. Alternatively, it might be to open a modal dialog and request that the user attempts to save to a different location. Either way, the process is not performing the program's main purpose, but it is reacting to anticipated non-main conditions in a planned fashion, in an attempt to get back to performing its main purpose.

Question 1: Does the inability to save the file constitute an 'error'?

Scenario C: Consider next that the user was able to save the document, but that in doing so it filled all but a few hundred kilobytes of the only disk available on the system. Assume that each instance of the program needs several megabytes of working storage available on disk to fulfil its undo, background-printing and other 'neat' features. The next time the user invokes a new process, which now finds itself unable to allocate the requisite disk space, what should it do? Presumably, we would like it to detect the condition and its cause, and inform the user of both, since he/she may/should be in a position to amend the situation, something that the process cannot be expected to do. It would be unsatisfactory to have the process silently fail, as the user would be left clicking new instances into life until the end of time (or at least until the end of his/her rag). It would also be unsatisfactory to have the process kick into seemingly good life, only to fail to save (in any form, anywhere) the user's work.

Question 2: Does the inability to allocate the working storage constitute an 'error'?

Scenario D: Consider, finally, that the program's author(s) failed to anticipate a certain condition, such as the user specifying a negative font size, whereupon the process's stack becomes corrupted, the process embarks on 'undefined behaviour', and the execution ends up who-knows-where (including, potentially, the document's disk file being overwritten with garbage and all work being lost).

Question 3: Does the failure to anticipate the negative font, and the consequent undefined behaviour, crash, and loss of work constitute an 'error'?

In this instalment I'm actually not interested in which, if any, of the three questions gets a Yes. We'll deal with that in a later instalment. (For the record, it's Question 3.) What I hope the distinct natures of the three questions illustrate is that answering Yes to any more than one of them – as many programmers would if asked each in isolation – does not make sense. Only one of them can get a Yes. And that's why the term 'error' is so problematic.

Additionally, we have a problem with describing the four types of behaviour of the program. There's the *main-purpose* bit, the *can't-save-but-specify-another-place-and-carry-on-working* bit, the *can't-start-up* bit, and the *undefined-behaviour* bit.

A new vocabulary

I suggest that we need a new vocabulary, and am (humbly?) proposing one here. With some help from the folks on the ACCU general mailing list, I propose the following precisely-defined terms: *contingent*; *defective*; *faulted*; *normative*; *non-normative*; *recoverable*; *practically-*

once a process has entered the realm of **undefined behaviour**, there's no way to get home

Condition	Normative	Defective	Recoverable	Designed
normative (A)	Yes	No	–	Yes
contingent ***recoverable*** (B)	No		Yes	
practically-unrecoverable (C)			No	
faulted (D)		Yes		No

Table 1

unrecoverable. Each is derived in the following series of reductions, and the relationships are illustrated in the state diagram shown in Figure 1.

Normative behaviour is the main business of any piece of software, its raison d'être, represented in Figure 1 by the transition from the *Normative* state to *Normative* state, via the *normative* action. Excepting the transitions from *Starting* and to *Stopped*, everything in scenario A is encapsulated within that single state and transition.

Rather obviously, everything that is not normative is ***non-normative***. To properly understand non-normative action, and to use exceptions appropriately in accordance, we need two further reductions, delineating on the criteria of recoverability and defectiveness. All the reductions are shown in Table 1.

The difference between scenario A and the others is that, in scenarios B, C, and D the user encountered a non-normative condition, one that is not a (desired) part of the functionality that is the main-business of the program.

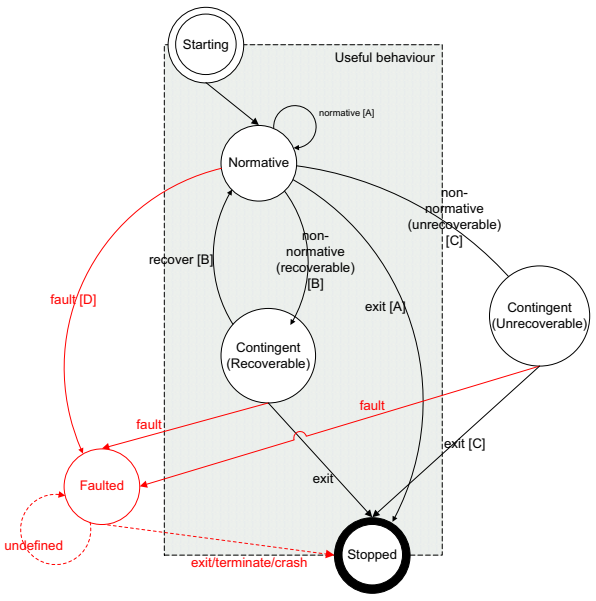


Figure 1

Certainly, if systems could be created in which the saving of a file could never fail, or file-systems were perfectly reliable and had infinite storage capacity, that would be great. But since such things are impossible in practice, it's necessary to deal with contingencies (where possible). Experiencing such conditions is an inevitable part of just about every process, so the detection and handling of them is a vitally important part of software development.

The difference between scenarios B and C is that, in scenario B, the non-normative action could be handled in a way – for example, the user may select a different disk, or may delete some unwanted files to make space – that allowed the process to return to normative behaviour. In scenario C, it was not possible to achieve normative behaviour, and the process could not (be allowed to) continue.

I suggest that non-normative conditions that are recoverable (i.e. from which it is possible to get back to normative conditions) and the actions (such as opening a dialog, or deleting some temporary storage) that allow such recovery be termed ***recoverable***. In Figure 1, this is represented by the transitions between the *Normative* and *Contingent (Recoverable)* states.

That just leaves us with discriminating between the behaviour of scenarios C and D, for which we need to consider the notion of defectiveness. As you will remember from part 2 [QM-2], an executing software entity can exist in three states: *correct*, *defective*, *unknown*. If a software entity exhibits behaviour that contradicts its design, then it makes the transition from correct or unknown to defective. What's not yet been established (but will be in a forthcoming instalment) is that the transition to defective is a one-way trip, according to the *principle of irrecoverability*:

> The ***principle of irrecoverability*** states: It is not possible for a software system to operate in accordance with its design if any component part of that system has violated its design.

Sticking with terminology that C/C++ programmers like, once a process has entered the realm of undefined behaviour, there's no way to get home. In Figure 1, this is evinced by only solid arrows leading into the *Faulted* state. The only arrows to emerge from the *Faulted* state, whence there's no guarantee of anything at all, not even of an eventual exit, are denoted by being dashed. I expect most of you, gentle readers, will have pertinent experiences as users and as programmers that attest to this uncertainty.

the failure to achieve a useful outcome is a result of the state of the operating environment

So, the crucial difference between the behaviour of scenarios C and D is not in terms of what the user is able to achieve with the process. In both cases the user cannot achieve anything useful with the process. (Or at least, we must hope that's true, since D can do *anything*, which could include appearing to behave in correct manner while silently deleting all the files on your system.) Rather, the difference is in what causes the unusability of the process.

In scenario C, this is caused by a runtime condition that can be experienced by a program operating in accordance with its design. In such cases, the failure to achieve a useful outcome is a result of the state of the operating environment, itself operating according to its design. There does not have to be anything wrong with either program or operating environment, it's merely a practical issue resulting from the confluence of their respective states and behaviours. After long, but ultimately fruitless, search for a better term, I suggest that this kind of non-normative condition be termed ***practically-unrecoverable***. (I wanted to use the term *fatal*, but I fear it too has been associated with an ambiguous cloud of meanings pertaining to things stopping for a variety reasons, including faults. In any case, the chosen term reinforces the fact that it's a practical matter, usually involving a balance between exhaustion of finite resources and the amount of development effort to work around the finitude.)

It will be useful to refer to recoverable and practically-unrecoverable collectively. Since both involve contingent action (even if some of that is provided by the language runtime), I suggest the term ***contingent***.

By contrast to scenario C, the unusability in scenario D arises from mistake(s) on the part of the software's author(s), either in the design, or in the reification of that design in the coding process. Since software does exactly what it's been programmed to do, if it's been programmed wrong, it is wrong. As I've already mentioned, I'll be dealing with the issue of software contracts, design violations, undefined behaviour, and so forth in a future instalment. For now, I'll restrict myself to simply defining the state of being defective as ***faulted***. Whilst it's possibly easier to say/read, I rejected the term *defective* because it already has a meaning, and because it's possible for a process to be defective but never fault, if the defective part does not affect a given execution (in which case one might never know that it's defective).

I want to stress the point that in both recoverable and practically-unrecoverable execution, the program is reacting to conditions that have been anticipated in the design. The better the anticipation, and the more detailed and sophisticated the response, the higher the quality of the software when viewed from the perspective of the user-in-an-emergency. It is vital to recognise the significant point is not that such anticipation/response *should* be made, but that it *can* be made. Both recoverable and practically-unrecoverable conditions can be anticipated.

It's almost a tautology that for a condition to be recoverable, it must have been anticipated. But the anticipation does not have to be precise. In scenario B, it's possible to handle the general case of fail-to-save and use the associated information provided with the condition by the software component asked to perform the file operation, so long as appropriate information is provided. (This is where exceptions can have a significant advantage over return codes, if used well.)

In some cases where the condition has not been explicitly anticipated by the programmer, the language runtime may do that on his/her behalf. An obvious example is C++'s `new` operator throwing `bad_alloc` if it cannot fulfil the allocation request, which, if not caught, results in `terminate()` begin invoked, which causes the process to be terminated. Thus, the set of practically-unrecoverable conditions can comprise those that have been anticipated by the programmer of the software as well as those that have been anticipated by the language/library designer.

By contrast, conditions (marked 'fault' in Figure 1) which lead to the faulted state are those that do not constitute part of, and in fact contravene, the design. To hammer the point home I've denoted it in the 'Designed'column in Table 1.

Whose concern(s)?

Normative behaviour is exhibited when your program encounters circumstances which your requirements and design designated as usual. Non-normative behaviour is exhibited when your program encounters circumstances which your requirements and design designated as unusual (or for which the requirements and design failed to account).

For obvious reasons, normative behaviour attracts the most focus. It is the prime concern of users, as is proper. It's also the major concern of project managers, requirements gatherers, business analysts, architects, and programmers, albeit that is a little less proper. When these disparate souls who band together to produce the software concern themselves exclusively, or even disproportionately, with normative behaviour, however, it's quite improper.

Non-normative actions, like insurance policies, the police, and a healthy immune system, are of little interest to most users until they find themselves in need of them. When they are needed, however, their worth multiplies enormously. (Philosophically speaking, this is one of the major causes of failure in software projects, though that's outside the scope of this instalment. I think it also explains what's wrong with democracy, but I'm way outside my remit here …)

It is often the case that the judgement of the quality of a software product is disproportionately biased in favour of how well it deals with non-normative situations. (The reason I love FireFox so much is that, when it crashes – which it does a fair bit, though usually only when a certain Jobs'-vilified multimedia plug-in is getting a time slice – it knows the exact state of every tab in every window – and I have lots of tabs in lots of windows! – and restores them faithfully when next invoked.) Consequently, when viewed from the perspective of the programmer, the non-normative actions are very important indeed. In some senses, they're more important than the normative. But just as they are important, so they are also not well understood, nor well discriminated, particularly when it comes to the application of exceptions. Addressing this situation is the main thesis of these instalments.

Exception use stereotypes

Before we start looking at code, or picking at any particular language/library scabs, let's set the scene by stipulating some exception use stereotypes.

The **exceptions-are-evil** faction holds that exceptions are merely a slight, and inadequate, step-up from using goto for the handling of non-normative behaviour. One of the group's most famous members is Joel Spolsky, who (in)famously declared his colours in [SPOLSKY]. Though he copped cartloads of opprobrium – some suggesting he'd jumped the shark – for having the guts to put on record what concerns many, Spolsky identified two significant problems with the use of exceptions:

- they are invisible in the source code
- they create too many possible exit points

In my opinion, the two problems indicated amount to the same thing: exceptions break code locality. One of the attractions of C programming [C^C++] is that, absent any use of `longjmp()` or any capricious use of `fork()` or `exit()`/`_Exit()`/`abort()`/`raise()`, it's possible to look at a sample of C code and understand its flow and, within the limitations of the given level of abstraction, its semantics for both normative and non-normative execution. Such is not the case in C++, C#, D, Java, Python, Ruby, or any of the other host of languages that rely on exceptions. When looking at code in such languages that is implemented in terms of other components, it is often difficult, even impossible, to understand the flow of control. This is a substantial detraction from transparency.

For my part, there is another important problem with the use of exceptions, which tends to get overlooked by most commentary:

- exceptions are quenchable

Regardless of the importance of a given exception, it is possible for any part of the calling code to catch it (and not rethrow it), thereby quenching what might have been a vital reporting of an unrecoverable condition. (The only exception of which I'm aware is .NET's `ThreadAbortException`.)

Spolsky's policy for exceptions is:

- never throw an exception
- catch every possible exception that might be thrown by a component in the immediate client code and deal with it

As with his objections to the use of exceptions, this policy contains some valuable nuggets of useful thought encapsulated within a horrifyingly-simplistic whole.

The **exceptions-for-exceptional-conditions** faction (as espoused in Kernighan and Pike's excellent *The Practice of Programming* [TPoP]) holds that exceptions should be used to indicate exceptional conditions (that fall within its design). In our terminology, this means that they should be used to indicate contingent conditions, but not for normative action.

The **exceptions-for-normative-execution** faction simply uses exceptions for any kind of programming they feel like. An example from the aforementioned Python book does exactly this. In order to effect a three-level break, a custom exception class is used.

The **exceptions-for-fault-reporting** faction holds that exceptions can be used for reporting contract violations (in addition to the other purposes already mentioned). Indeed, a number of languages and standard libraries adhere to this, including those of the C++ (`logic_error`), and Java (`AssertionError`) languages. Unfortunately, they are all wrong! Alas, once again I'm trespassing on the forthcoming contract programming instalment and trying your patience by teasing you with a controversial assertion without accompanying argument, but it can't be helped for now, as I'm running out of space and off the point.

The **exceptions-are-broken** faction don't trust their compiler to do the right thing when exceptions are thrown. Sometimes, they have a point.

The **exceptions-are-slow** faction can't use exceptions, due to performance (time and/or space) reasons, on real-time and/or embedded systems. They may be influenced in their views by the costs associated with the wholesale use of exceptions by the **exceptions-for-normative-execution** faction.

And the **exceptions-won't-clean-up-after-me** faction, populated by folks who don't know the techniques, like RAII, that can manage their resources expressively, with exception-safety, *and* make their code clearer.

Naturally enough, none of these positions is unalloyed wisdom, but that's enough groundwork. In the next instalment, I'll attempt to put flesh on the bones of the taxonomy, establish evidence for the recommendations I'm going to make, and illustrate the problems with each of the stereotypes, by discussing three of my own projects' exceptional issues.

What *are* exceptions?

All of the foregoing might have you wondering exactly what is an exception. (It did me.) Here's my parting thought on which you may cogitate until we meet again:

> **Exceptions**: Exceptions are not an 'error'-handling mechanism. They are an execution flow-control mechanism.

We can say this because:

- the term 'error', if meaningful at all, does not represent the things for which exceptions are predominantly used under the nebulous umbrella of 'error-handling'
- it's possible, even if it's not wise, to use exceptions for flow-control of normative execution, something that no-one could ever claim to be 'error'

In this regard, exceptions are the same as returning status codes. Just a little less ignorable. ■

Acknowledgements

I'd like to thank Ric Parkin, the Overload editor, for his customary patience in the face of my chronic lateness and for small-but-essential suggestions of improvement, and my friends and, on occasion, righteously-skeptical review panel resident experts, Garth Lancaster and Chris Oldwood.

References

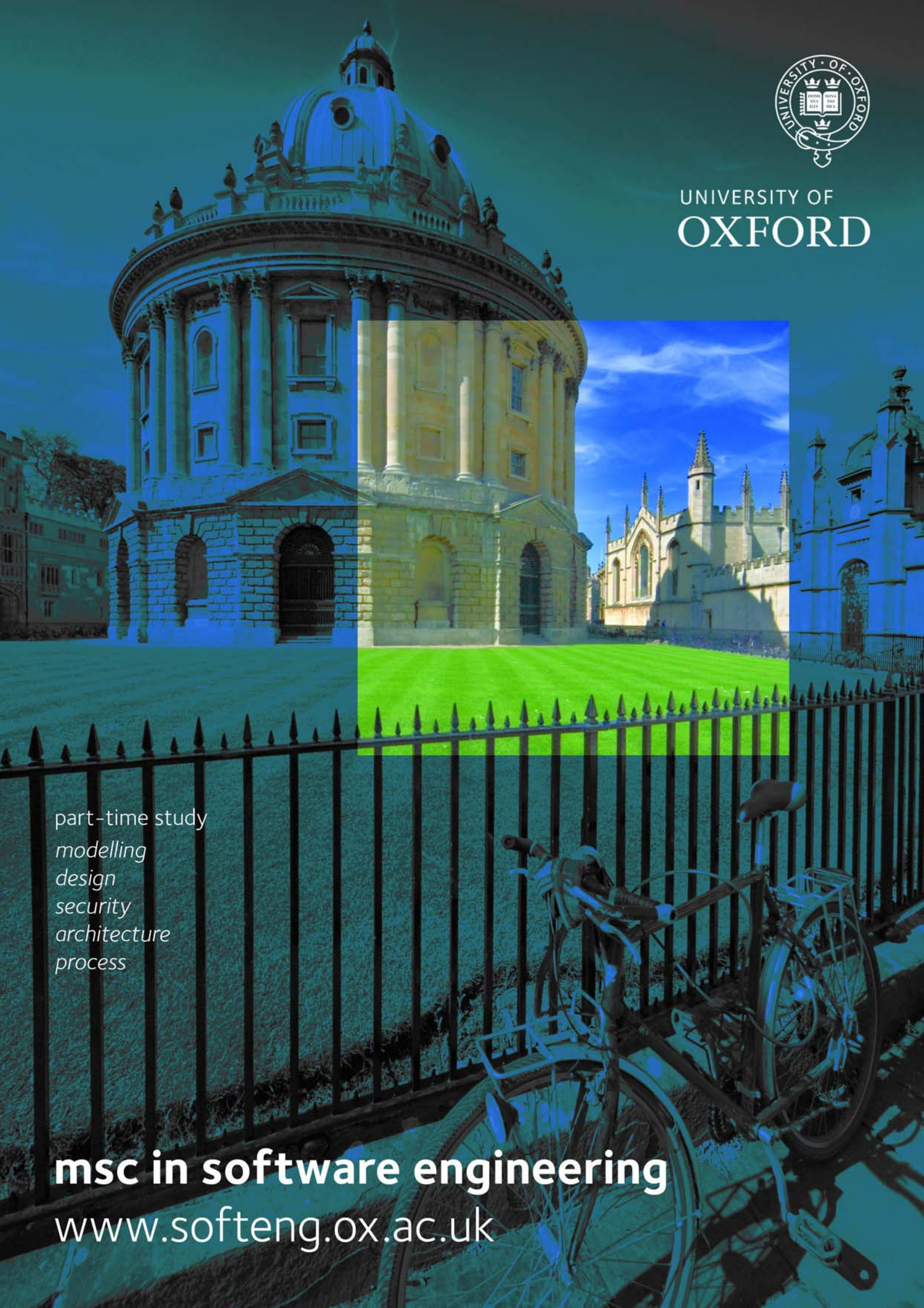
[C^C++] '!(C ^ C++)', Matthew Wilson, *CVu*, November 2008

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

[QM-2] 'Quality Matters: Correctness, Robustness, and Reliability', Matthew Wilson, *Overload* 93, October 2009

[SPOLSKY] http://www.joelonsoftware.com/items/2003/10/13.html

[TPoP] *The Practice of Programming*, Kernighan and Pike, Addison-Wesley


UNIVERSITY OF
OXFORD
part-time study
modelling
design
security
architecture
process
msc in software engineering
www.softeng.ox.ac.uk