overload 104

AUGUST 2011 £3

An Introduction to Test-Driven Development

An example-led foray into TDD, involving acceptance, integration, and unit tests

Over-Generic Use of Abstractions

A look at how using overly-general functionality can lead to sub-optimal outcomes

Integrating Testers on Agile Teams

We look at how testing and testers fit into an agile development process

Thread-Safe Access Guards

A simple but effective way to control access to data from multiple threads

Why <algorithm> Won't Cure Your Calculus Blues

We start a new series showing how to analyse numerical calculations. It's never as simple as you think!

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 104

August 2011

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

Copy deadlines

All articles intended for publication in Overload 105 should be submitted by 1st September 2011 and for Overload 106 by 1st November 2011.

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

4 Over-Generic Use of Abstractions as a Major Cause of Wasting Resources
Sergey Ignatchenko finds that good intentions can lead to performance problems.

7 Integrating Testers Into An Agile Team
Allan Kelly considers how to fit testing into your development process.

10 Thread-Safe Access Guards
Bjørn Reese develops a template to help with accessing shared data.

13 An Introduction to Test-Driven Development
Paul Grenyer gives a worked example showing the various types of tests.

21 Why [Insert Alogorithm Here] Won't Cure Your Calculus Blues
Richard Harris investigates how to use floating point calculations properly.

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

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

Rise of the Machines

Been having trouble with technology or simple devices? Ric Parkin fears for our future

I've not been having a good run of luck with machines. I almost feel like they're out to get me, but that is taking anthropomorphisation too far and they'd hate that.

First the oven broke. Fortunately it was just the heating element that needed replacing, but it still took the engineer three tries over a week or so to get the right part as there were several possible ones, all with subtly different attaching holes and all incompatible.

Then I was awoken at 3:30am by water dripping through the ceiling. Turns out the water tank was leaking, and after a few goes trying to fix, I was told that the whole cold water system needed replacing – apparently the ex-councellor who'd owned the house in the 1970s had used it to showcase local businesses, and all the piping was made by a local company ... but only for a year or so. It had never caught on as the acrylic pipes needed gluing together, which meant you couldn't test them for 24 hours, and were pretty much impossible to repair piecemeal. Plus, since then standard sizes have changed and you can't get compatible parts. Given the size of the job I'm getting the bathroom redone at the same time, and so am looking at options, in particular which shower to have put in.

Then this weekend I came home to find a washing machine full of undrained dirty water, with a couple of arbitary lights flashing.

What do these have in common? Well, I've got around to reading Donald A. Norman's classic book *The Design Of Everyday Things* [Norman] so have been thinking about how bad much of design is, including physical objects, electrical devices, and user interfaces. And I realised that my rebellious machines showed various aspects of his ideas, good and bad, and that many of them have lessons that can be applied to software.

So let's look at my examples in more depth.

An oven heating element is remarkably simple. It's just a heavy spiral of metal that attaches to the electrics, and which, due to its high resistance, heats up. It connect to a circuit controlled via a thermostat and the user's controls on the face panel, and is held in place with a central screw. Unfortunately the usual element for that make has *two* screw holes on either side, and so couldn't be attached. A note was made of the model, and after some searching online a new one was ordered – to find that it wasn't quite the right one, which didn't fit either. Finally the correct one was found, and installed in minutes.

What can be learnt here? First is to ask why on earth are there different fittings? It is surely simple to standardise on a single connector, or have a 'universal' bracket that has the right holes for any oven's need, perhaps with two outer holes as well as a central. Having said that, there could be a safety aspect – you wouldn't want to install an element that couldn't cope with the current the oven would deliver. A good solution to that is actually to have different connection types for the different power ratings so that you cannot possibly install the wrong type. But why did the wrong one get ordered? Well the types only differed in subtle ways that were missed when searching. So here is another lesson: make the differences **obvious**, perhaps via the number of rings, or the connection orientation, or even colour coding of the insulation. As it was they all looked the same.

As for the water system, one problem I found was knowing which bits were what, what each valve did, and where did all these pipes go as they usually disappeared under insulation or through the floor. What would have made things easier would be to have put simple labels on the pipes detailing their purpose, and perhaps even their measurements. When I get the system replaced I'll be adding them. Another idea I got from Stewart Brand's *How Buildings Learn* [Brand] is, before you hide the utilities behind walls, plaster etc, take photos and put them in a book. The book stays with the house, and forms a record of the hidden anatomy of the place, and will become invaluable to yourself and any future owners and maintainers. And of course, as the pipes and shower are going to be boxed in, I'll make sure there's some way of access for maintenence.

In contrast the washing machine is a bit more of a success story, but there are still lessons to be learnt. In contrast to most models with their enormous number of controls, dials and displays, which are so complicated that you tend to learn one or two common settings and never use the rest, this one has gone for a very simple but useful set. See Figure 1. It comprises an outer ring of five buttons – one power with embedded red 'locked' LED , and four wash types, all of which have simple logos and temperature, and are back illuminated; an inner ring of LEDs; and a central logo. To start a wash, press the power button and the LEDs will light up one by one and a short tune indicates it booting quickly. What next? It prompts by having the wash buttons' lights gently fading on and off. Then press your chosen wash button, and that's it! Two presses – that's all that's needed to start one of the six wash types. Ooops, yes you spotted it....four buttons, six types. Why did they spoil the simplicity of the single buttons? How can I remember how to do the others? They do redeem themselves partly by having an aide memoire of the types and the button presses needed on a little panel when you open the door to put the washing in. And one of the types – rinse only – is detected automatically by sensing detergent on the clothes at the start. But a mark lost for overloading the buttons for the last two.

As the cycle progresses, the LEDs light up one by one giving a simple indicator of how long is left, spoilt only by the possibility of multiple rinses making the last few significantly longer than the others – thankfully it's not as wildly inaccurate as the Windows progress dialog. Completion

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.

Figure 1

is indicated by a long beep, and in case you miss that the LEDs fade on and off to show it's finished, which really stands out in a dark room.

What if you want to cancel it? Just hold the power button till it beeps and starts to flash. Notice you have to hold it, otherwise you could cancel by accident. Would have been even better if it gave some feedback while you held it to show you were doing something correct, perhaps beeping repeatedly (but not rapidly – that might make you think it was an incorrect action), but holding a power button to turn off is a pretty common convention.

The only other thing that's wrong I spotted when the engineer came to look at it. If you noticed, the logo of the make is a play on the standard symbol for a power button, and as it's also curved to stand out, quite sensibly that was the first button he pressed to switch it on. An improvement might have been to actually make that the power button.

So here we have a fairly clean, simple user interface, with obvious options for the day-to-day things, a reminder of the less obvious, and useful feedback without information overload. Why can't more devices be made this useable?

And not just physical devices – these examples have their counterparts in the world of software. For example, the popular distributed source control system, git, seems to cause new users some confusion. The example, as I understand it, involves some of the defaults – when you do the basic **`git pull`** to get code, it gets only your current branch. But when you **`git push`**, it pushes all your changes, including ones on other branches, to the repository. Why the unexpected difference? It gets worse – if there is a conflict on a different branch, it prevents you pushing, in a hard to fathom way. Sure you can get used to these, but choosing good defaults would be better. What sort of defaults? Well, ones that do the most obvious thing given the rest of the context, and causes least damage if invoked unexpectedly. In this case I'd say the best is to push/pull only the current, and provide explicit options to provide a list, or all. This is a bit like having sockets compatible only for safe things – make it obvious to do the right, safe thing, but hard work (or impossible) to do anything stupid or unsafe.

This principle applies to tools, but also to API design. For example, consider a function that returns a raw pointer. What is its lifetime? Do you own it? How do you release it? You could document this, but that's the equivalent of those thick appliance manuals that you never read. Instead (or on top of the harder to use API), encapsulate the ownership rules in a smart pointer. Perhaps allow a 'checked' mode to detect incorrect usage, so long as performance doesn't suffer unduly. If there is an order of calls required, make the orders obvious, perhaps by passing a token or object back from one that needs to be passed to the next.

Similar ideas also apply to user interface design – make the obvious things easy, and provide ways to recover from unexpected things, perhaps with an undo facility. Also avoid mistakes in the first place by providing clear feedback and indicators of what can be done next.

And, topically, security can learn from some of these principles. For example, if you provide a voicemail facility, it could be a good idea to have it disabled by default. If it's on, by default only allow access from the phone itself. Or allow only from known authorised numbers. And if from anywhere, do not have a default password/pin, but force one, perhaps with checks to discourage obvious ones such as 1234 or part of the phone's own number. If such simple measures were in place, we might be a rather different world...

References

[Brand] http://en.wikipedia.org/wiki/How_Buildings_Learn

[Norman] http://www.jnd.org/books.html#33

Over-Generic Use of Abstractons as a Major Cause of Wasting Resources

We often find ways to hide complexity. Sergey Ignatchenko looks at how this can be sub-optimal.

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

A story to freeze the blood

Tharn (Lapine) – stupefied by terror

Here is the horror story told to me on a Christmas night by one of my fellow rabbits (later he agreed to me publishing it on condition that I will never reveal his name):

> I will never forget the night it happened. It was a quiet night and I was downloading a 2G ZIP file with Internet Explorer. Suddenly there was a loud beep. I sprang out of sleep and looked at the monitor. Nothing looked out of the ordinary except there was a message box saying that it needs 2G of free space on the drive. 'Whaaaaaat?' I thought, 'Last time I saw the progress bar it was at 90% downloaded, what does it need another 2G for?' Summoning my courage I copied 2G of files to an external drive and clicked the button to tell IE about it. All of a sudden IE started to copy the file (which had already been downloaded) from its cache to the location that I originally asked for. Positively tharn, I waited while this blood-curdling process has been completed. When it was over I thought that the worst was behind me. I couldn't have been more wrong. When I clicked 'Open' my computer froze: even the mouse cursor didn't move. After several long horrifying minutes, when my trembling hand was already reaching for the power button, I suddenly noticed a constantly lit HDD light. 'It is not a virus, the damn thing is just swapping!' flashed through my mind. Oh, it was a great relief. From that point it didn't take more than ten minutes (swapping the 1G of RAM I have is a rather long process) and half a dozen of clicks to unpack the whole ZIP file and reach what I needed. Interestingly enough, I didn't need to free any more space for the unpacked files.

Cold-blooded analysis

After listening to such a chilling story, we'll have to warm our blood a bit before we start analyzing this horrifying sequence of events in cold blood (suitable methods of heating are beyond the scope of this article). Putting on a deerstalker and using Holmesian deduction [Holmes], we can establish the following points rather quickly:

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams [Adams].

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

1. IE always downloads a file to a cache rather than to the final destination, even though it removes it from the cache right after the copying.
2. IE always copies the file from the cache to the final destination and then deletes the cached one, even if they are on the same partition so moving the file instead would be equivalent and orders of magnitude faster.
3. When IE copies file from the cache to the final destination it doesn't block Windows from caching in memory the file being copied even if the file is 2G in size and the total amount of RAM is 1G, which makes it nothing but cache poisoning. Windows, on the other hand, went to great lengths to perform this caching in the best possible way (from its own point of view), swapping out active programs to cache perfectly useless portions of the file.

None of the things above makes any sense in our case. But now we need to understand the motives behind this horrible crime against the end-user (who's the one we're all working for [NoBugs11]) and CO_2 emissions. Going further in our deduction exercise, we can easily notice that all three cases are related to using an abstraction (or abstractions) which is useful in many cases but fails in our specific case. For item 1, we can safely assume that it was a concept 'download everything to the cache' which has misfired. For item 2, it was the combination of two separate abstractions 'after downloading, copy files from the cache to the destination' and 'remove file from the cache when it is no longer needed', which became deadly inefficient when used together because of the lack of interaction between them. For item 3, it was an abstraction of 'cache files in RAM', which makes sense in general, but when used for huge files such an oversimplified approach has caused cache poisoning, and severely hit the overall process.

The combination of these misused abstractions has caused about 6G of data to be copied between RAM and the HDD without any real reason behind it: first, IE has copied the 2G file from its cache to the final destination (requiring 2G of data to be moved into memory, and then back to file for a total of 4G); second, Windows swapped out (roughly) 1G of RAM while the file was copied, and third, it swapped 1G of RAM back in. Still, even if all this outright unnecessary operations are eliminated, the process will still lack from end-user point of view: the user would still need to move 2G of data to another HDD to free the space, which is used only temporarily.

And one more point

4. If we think how the end-user would ideally like to see it, we notice that (given what is usually done with ZIP files) it is clearly possible and feasible for IE to ask the user (even before downloading) if they wants to extract the ZIP file 'on the fly'; and if user knows that s/he really only needs the extracted files then no extra copying will be necessary, saving another 8G of HDD operations (4G for freeing space, and 4G for unzipping).

We should note that the lack of implementation of item 4 can be attributed to a 'download file, then call application which handles it' abstraction

when you repeatedly allocate and deallocate memory from the common heap it becomes fragmented with interlaced used and unused blocks

which is very useful in general, but fails to address specific cases (for us, downloading large ZIP files).

Generalization about being overly generic

The analysis we've done has shown that while there were multiple reasons for such a deadly inefficient process, all the reasons which we were able to deduce were related to the single root cause – abstractions and/or concepts which are useful in general but lead to substantial inefficiencies in a certain specific case. Let's name this phenomenon an 'Over-Generic Use of Abstractions', or OGUA. It is important to note that while the primary responsibility for OGUA lies with developer who uses the abstraction without proper applicability analysis, the way abstractions are represented can themselves stimulate their misuse. One common example is when abstraction is represented with an API which simply lacks the functions necessary to avoid being over-generic. Still, we need to mention that the potential of avoiding being over-generic is not an excuse to include 'each and every function which might be needed by somebody in the future' into the API, and that the principle of 'No Bugs' Axe' [NoBugs10] should still be observed. In practice this means that by default such functions to avoid being over-generic should not be included into the API, but that whenever a concrete case is demonstrated which shows real benefits which cannot be achieved otherwise, then such functions should be added. Any of items 1–4 above are a very good example of such a concrete case which calls to be fixed (if necessary – by extending APIs of involved abstractions).

Clues in a heap

For a long time I was wondering: what are all the modern browsers doing with memory? If you work with a browser, keeping a few dozens of open tabs for several days, it tends to consume enormous amounts of RAM (a browser taking 1G for two dozens of tabs open is not uncommon, and while I won't start the speech 'in our day you could have a full-scale 3D game within 41K' again [NoBugs11a], even these days 1G is still an enormous amount of memory).

Recently I've run into an in interesting feature of Internet Explorer (at least IE8) which helps in researching this phenomenon. When you kill the IE process (for example, using Task Manager), Internet Explorer automatically re-launches the process and restores all the tabs you had open to the very same state as they looked before the kill. So, in a few seconds after killing the process we have the very same system, displaying the very same information, as right before killing the process. One should expect that nothing should have changed, right? Wrong – very often the total RAM usage of the system drops and often the difference is pretty large. Let's take a look at Fig. 1. It shows the total memory usage during the exercise I've described above. Originally memory usage was displayed as 1.04G, then we killed the first process, memory usage went down sharply, then was a small rebound (indicating what was really necessary to show this information). Then we killed another IE8 process (by default, IE8 opens one process per several tabs) and so on. As a result we've got the system which is showing the very same pages, but using about 200M

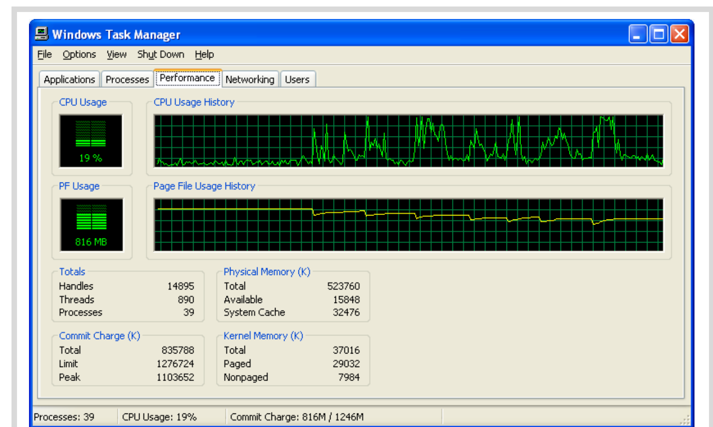

Figure 1

(!) less RAM. And this isn't the most impressive case – I've seen occurrences when RAM usage has dropped from 1.5G to 800M after such an exercise – still showing the very same pages (note to those who will try to reproduce this effect – for maximum observable results you need to spend a while in IE8, ideally several days, browsing lots of media-rich pages – Yahoo, CNN and similar do just nicely – within the same tabs, leaving the tabs open on the page which interests you).

Our findings indicate an obvious waste of resources, and in the search for a culprit our detective instincts tell us that it should be either memory leaks or a fragmented heap. As the problem of using enormous amounts of RAM is common for all the modern browsers, which use very different engines, it doesn't look likely that every browser has memory leaks. So, using Holmes' favorite adage of 'Once you eliminated the impossible, whatever remains, however improbable, must be the truth', we can point towards heap fragmentation as a most likely suspect.

After faithful Watson takes a look at Holmes' extensive library of bugs, we find that heap fragmentation is a nasty and way-too-often overlooked effect: when you repeatedly allocate and deallocate memory from the common heap it becomes fragmented with interlaced used and unused blocks, and often a new allocation cannot be satisfied by the heap even if the total free memory is enough: some space is available, but it is just allocated within chunks each of which are not big enough to allocate what we need. Another related, but separate, effect is that, as allocation of memory from the OS for the heap is typically based on CPU pages, it means that to release a page back to OS, all objects on the page must be freed. A typical page size these days is 4K and a typical object size is around 100 bytes, so if, for example, we have allocated 10000 objects and therefore got 256 pages from OS for them, and then we randomly deallocate 9900 of the objects – statistically we'll get only 173 pages back leaving us with 100 100-byte objects taking 83 pages (332K) – a 3220 % waste!

Now as we've identified our suspect, let's find how he could have do his scary job right under the very noses of Scotland Yard. Once again, it was

separating processes merely to address heap fragmentation looks like a huge overkill

inspired by an over-generalized abstraction: while the concept of a 'default heap' with functions like `malloc()` and `free()` conveniently free the developer from thinking 'where should this memory go', it turns out to be a curse in disguise for long-lasting applications which tend to suffer from heap fragmentation. In most cases, to mitigate the effects of heap fragmentation it is necessary to have separate heaps; some modern browsers (notably Chrome and IE) are achieving this by putting different pages into separate processes. This has other positive effects but what is obvious is that whenever a process is killed its heap will be freed with all the accumulated fragmentation. Still, separating processes merely to address heap fragmentation looks like a huge overkill (and a waste of other resources too). A much better way would be to take over control of memory allocation, for example using Boost pool allocators [Boost], but applications which do this are very few and far between.

Yet another case against OGUA

Another case which looks bad for the OGUA is the ubiquitous TCP protocol. It provides a nice and neat abstraction of a reliable network stream, and it works. Problems start when one needs to implement an interactive service over TCP, which needs to provide different priorities for different requests. TCP maintains rather large buffers on both sides of connection, and doesn't support priority itself (in theory, there is such thing as TCP Out-of-band data, but it is well-known to be unreliable in practice, see, for example, [TCPOOB]). It means that even if you implement your system with priorities in mind, your users will still be affected by delays introduced by TCP. These delays (unlike, for example, delays in transit) exist not because some law of physics or informatics says that delay is inevitable: this kind of delays is there for only one reason: TCP suffers from OGUA (and this time it is not the fault of developers who're using it – these days there is simply no way around TCP if you need to transfer data over the Internet, as even if you take big pains to implement your own protocol over UDP it will be blocked in many practically important cases).

Of course, there are ways to bypass these delays (quite a few people have managed to write interactive programs over TCP), but each of these techniques will cause a waste of some other resource. For example, the flag `TCP_NODELAY` is commonly use to mitigate this problem in real-world interactive systems, but will cause excessive network traffic; reducing send/receive windows will reduce throughput; and creating a second TCP connection with smaller send/receive windows will also cause excessive network traffic, not to mention the complexity of dual connections.

Full-scale discussion of TCP and interactivity is very complicated and can easily take the whole article, or even a whole book, but what is already clear is that this is also a case when a nice and neat abstraction (which works in many practical scenarios) causes a waste of resources when it faces certain tasks with specific requirements.

Elementary, my dear Watson!

We have analyzed several significant cases of criminal waste of resources, and have found that despite the resources being very different in each case – from HDD accesses to memory to network traffic – in all of them our main suspect was the over-generalized use of abstractions, the fiendish OGUA. We will see during an upcoming trial if the evidence we have gathered is enough to get a guilty verdict from a jury based on 'beyond reasonable doubt', and put this 'Napolean of Waste' behind bars for good. ■

References

[Adams] http://en.wikipedia.org/wiki/Lapine_language

[Boost] http://www.boost.org/doc/libs/1_46_1/libs/pool/doc/index.html

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

[Holmes] http://en.wikipedia.org/wiki/Sherlock_Holmes#Holmesian_deduction

[NoBugs10] 'From Occam's Razor to No Bugs' Axe', Sergey Ignatchenko, *Overload* 100, December 2010

[NoBugs11] 'The Guy We're All Working For', Sergey Ignatchenko, *Overload* 103, June 2011

[NoBugs11a] 'Overused Code Reuse', Sergey Ignatchenko, *Overload* 101, February 2011

[TCPOOB] http://en.wikipedia.org/wiki/Transmission_Control_Protocol#Out_of_band_data

Integrating Testers Into An Agile Team

Agile has usually concentrated on how to organise developers. Allan Kelly shows how testers fit in.

When a Software Tester first looks at Agile process descriptions they see something missing: Testers, and, for that matter: Testing.

It is not so much that Agile processes and teams ignore testing, it's more that they ignore Testers. This is for two reasons: originally Agile came from Developers. Hence it is very developer centric and does not pay enough attention to such roles as Tester, Product Managers and Business Analysis. Second, Developers – like many others in the software world – don't really want to admit the need for Testers. Developers like to think they can do it right and nobody needs to check on them. Call it arrogance if you like.

Yet testing itself is taken very, very, seriously. One of the tenets of Agile software development is a commitment to higher quality. If the thing Developers produce is of better quality then there is less rework to be done, so there is less disruption and overall productivity rises. This is the *Quality is Free* argument advanced by Philip Crosby in the 1980s [Crosby80].

If proof is required look at Capers Jones' book *Applied Software Measurement* [Jones08]. Jones describes how IBM first discovered in the early 1970s that the projects with the highest quality (lowest number of bugs, bugs found and removed earliest) also had the shortest delivery schedules and as a consequence the lowest costs. Jones' data collected over four decades supports this assertion.

Thus the Agile approach to testing is to inject more quality early in the process. Practices such as automated unit tests – whether we call it Test Driven or Test First Development, automated acceptance tests, pair-programming and continuous integration all serve to improve the initial quality of the software.

If these practices are not adopted teams have little or no chance of making a successful release at the end of an iteration. If quality is not built into the code from the start of the iteration it cannot be retrofitted at the end. In order to make regular scheduled release of software it is necessary to abolish the indeterminable test-fix-test cycle at the end of development. See Figure 1 for how we imagine testing fits with development, and Figure 2 for the reality.

Software developers, and in particular the managers of development teams, seem capable of a specific form of doublethink. When it first becomes clear that a project is going to be late they get optimistic about later phases. When something needs to be done the first place the schedule is changed is in the test phase. Optimism wins and the time allowed for testing is reduced – 'this time we'll be better' while the final date is held fixed.

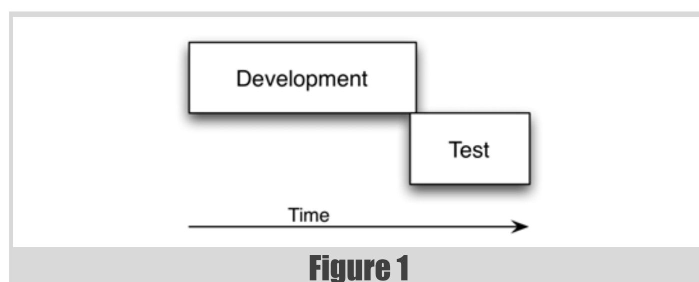

Figure 1

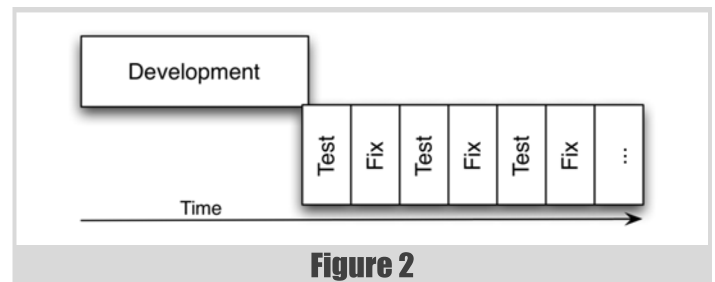

Figure 2

When a project is eventually closed and people look back they inevitably say 'We should have allowed more time for testing', yet a few weeks later when the next project starts to slip the first place they cut is testing again.

If, through Agile techniques, we manage to fix this problem we are left with the question: *where do Testers fit in*?

Furthermore, if we remove traditional requirements documents, then how do Testers know what to test? Let me answer the second question in the expectation that an answer to the first question will emerge.

How do I know what to test?

Dialogue over document is the norm for requirements in Agile teams. Rather than try and write everything down in some kind of binding contract, ideas are captured but details deferred until development. This just-in-time approach to requirements elaboration relies on conversations between the Product Owner and Developers.

Testers need to be involved in the same dialogue as the Developers and Product Owner. Problems that remain are not so much because there is no document but rather because there is a problem with the requirements, or because Testers are cut out of the loop altogether. Either way, there is a problem that needs to be addressed and documentation is not necessarily the answer.

(I'm using the term Product Owner here. In most organizations this person goes by the name of is a Business Analyst or Product Manager. The term Product Owner originates from Scrum. What ever they are called, this is the person who decides what should be in, and how it should be.)

Traditionally, team members focused on the document because, despite its imperfections, it is the only thing that is available. The deeper problem is often that nobody pro-actively owns the business need. All too often it is

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.

Testers should have an automated acceptance test ready in by the time developers finish their coding

Acceptance criteria v. acceptance tests

In a recent e-mail to Paul Grenyer and me, Rachel Davies pointed out that there is an often overlooked difference between Acceptance Criteria and Acceptance Tests:

Acceptance criteria are part of what you agree the story should cover. These can be simply notes about what scenarios are agreed to be in scope for the iteration. These are what need to be reviewed and agreed before committing to delivering a story in the next iteration.

Acceptance tests are scripts – manual or automated – that detail specific steps to test a feature. I suggest write these in the same iteration as the code, before or in parallel with developing the code. I do not suggest writing them in a planning meeting. I'm not suggesting that you write all the acceptance tests for all the stories at the beginning of the iteration. Just that when work starts on a story, the first thing to do is to get really clear about what tests should pass by following an Acceptance Test Driven Development way – this can be by writing one acceptance test at a time or the whole set for the story.

assumed that *developers know best*. The document was expected to answer three questions:

1. What needs testing?
2. How should the functionality perform?
3. How should the functionality be tested?

Tests need to be created and run on a just-in-time basis. While tests need to be created before a task is marked as done – otherwise how do you know it is done? – they should not be done too far in advance.

When tests are created too far in advance two problems emerge: work may be removed or postponed. Creating tests for work that is removed is a waste of time. Secondly, only when the work is about to begin is it clearly defined. In fact, at any time until the work begins information may arrive which changes that which is to be done. So any tests there are written in advance may need changing.

The mechanism which tells Developers what to work on next is the same mechanism that should tell Testers what needs testing. This could be your iteration planning meeting, or it could be the visual tracking system used by the team. Either way, it should be clear that code is about to be cut, therefore tests need to be created. It should also be clear that the code has been developed and the tests are ready to run.

Before the code progresses to automated acceptance tests Testers may be involved in keeping developers honest. They may ask the developer for evidence that the work has been performed in a quality manner: Were unit tests written? Was the code developed using pair programming? Or has a second developer reviewed the code?

Testers should have an automated acceptance test ready by the time developers finish their coding. This means Testers, like Developers, need to understand what the Product Owner expects from the software. Again the dialogue over document principle applies.

Testers need to be part of the dialogue, they need to part of the conversation when the Product Owner and Developer discuss the work so they can understand what is required at the end. If the work is contentious, or poorly understood, they may take notes and confirm these with the Product Owner later. They can then devise tests to tell when these requirements are met.

In this way Testers can understand what needs testing and how it should perform, they then use their professional skills to understand how to go about testing it.

It is worth adding here: automate, automate, automate. Tests which are not automated are expensive to run and therefore don't get run as often as they could. Automated tests are cheap to run and so maintain the quality into the future.

The testers' role

The testers' role is to close the loop, following from the Product Owner role, not the Developers role. They need to understand the problem the Product Owner has and which the developer is solving so they can check the problem is solved.

If Testers are having a problem knowing how something should be then it is probably a question of Product Ownership. When this role is understaffed or done badly, people take short-cuts and then it becomes difficult to know how things should be. One of the reasons many people like a signed-off document is that it ensures this process is done – or at least, it ensures something is done. But freezing things far too early, and for too long, reduces flexibility and inhibits change.

As software quality improves there may well be less need for Testers. However the need will never go away completely. There are some things that still require human validation – perhaps a GUI design, or a page layout, or printer alignment.

Three levels

There is a recurring model of testing found in many organizations and shown in Figure 3.

Many more organizations have a similar model but with one, or even two, levels missing. Instead they agonize about the level they 'should' do.

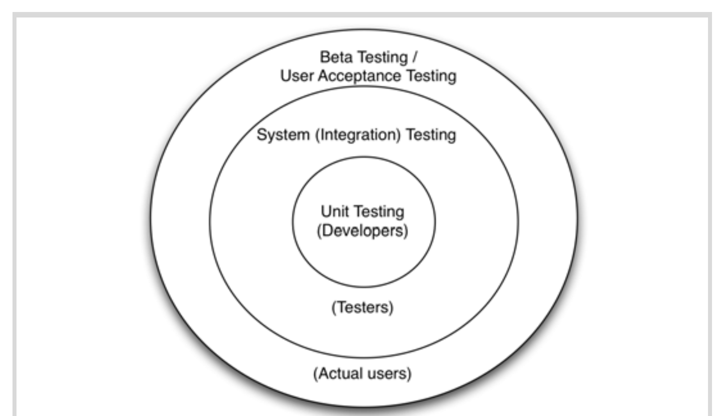

Figure 3

> **Removing the need for testing may well be an aspirational goal for a team but it is going to be a while before the goal is met**

- The innermost ring is Unit Testing. This is the developers' responsibility, the testers role here is largely to keep developers honest, and ensure they have written and performed the tests. Unfortunately this level is the one that is most commonly absent, which is a shame because it is probably the most important ring of the three.
- The second ring is variously called System Testing, Integration Testing, System Integration Testing or just SIT. This is the realm of the professional tester. Like Unit Testing the ideal is that this ring is completely automated. When this is possible Testers can write their tests before developers begin work, and the tests then become the developers completion criteria.
- The outermost ring differs depending on the nature of the customers. When customs are outside the organization, companies conduct Beta Test Programmes; when they are inside it is usually called User Acceptance Tests. As the name implies this is where actual users get involved.

  Sometimes, when quality is very low the UAT/Beta cycle becomes effectively a second SIT loop to catch more of the bugs which slipped through the first.

  Because this ring is about customers it is more difficult to automate. Testing is only part of the objective: one of the less spoken aims of this ring is to win over customers acceptance of the product, to make them feel involved and give them a voice. Unfortunately, this voice is often heard too late to make any real difference.

  Another problem with this loop is that actual users – particularly in corporations – are either too busy or do not value this loop. Therefore the loop ends up being run by professional testers again – in the extreme these are testers attached to the user group not the development group. When this happens the secondary objective of the loop is entirely lost because actual users are not involved.

Personally I believe it should be possible to merge the two outer loops. Firstly, UAT should not be used to catch bugs which slip through SIT: quality should be high enough to stop them at source or catch them the first time. Secondly, customer/user voices should be heard earlier, and repeatedly, in the development process at a time when they can influence the process.

Capers Jones, by the way, says that on average each round of testing removes 30% of all defects, which probably explains the three tier model. If one ring is to be removed then defect detection and removal rates need to improve in the other layers.

The future

In the short run there is a lot of software out there and quality isn't going to improve overnight. Removing the need for testing, and reducing the number of testers, may well be an aspirational goal for a team but it is going to be a while before the goal is met in the majority of cases. Even then someone will still need to write the acceptance tests.

If this weren't enough, the success of Agile should make more work for everyone. Agile teams are more productive and add more business value. Therefore the organization will succeed, therefore there will be more business, and thus there will be more work to do.

Over time as quality increases Testers' roles will change. The role will be less test centric and more quality assurance centric. Instead of bashing a keyboard to check the software doesn't crash Testers will increasingly close the loop with customers and Product Owners to ensure that what is delivered satisfies their need. ■

References

[Crosby80] Crosby, P. B. 1980. *Quality is free : the art of making quality certain*, New American Library.

[Jones08] Jones, C. 2008. *Applied Software Measurement*, McGraw Hill.

cqf.com

Expand Your Mind and Career

Designed by quant expert Dr Paul Wilmott, the CQF is a practical six month-part time course that covers every gamut of quantitative finance, including derivatives, development, quantitative trading and risk management.

Find out more at **cqf.com**.

ENGINEERED FOR THE FINANCIAL MARKETS

Thread-Safe Access Guards

Ensuring safe access to shared data can be cumbersome and error-prone. Bjørn Reese presents a technique to help.

This article describes a C++ template solution for ensuring synchronized access to variables in multi-threaded applications.

When doing multi-threaded programming in C++, we frequently come across this pattern seen in Listing 1.

Every time we want to access the member variable, we must make sure that we lock the mutex. Unfortunately this pattern is error-prone. The compiler cannot alert us when we have forgotten to lock the variable before we use it.

We really would like kind of some mechanism that made sure that access to the member variables always is synchronized, and where we can control the locking scope.

The basic idea is to have two classes. An accessor that automatically locks and unlocks the mutex, and a variable wrapper that contains the mutex and the variable we wish to protect with the mutex. The wrapper prevents any access to the variable except through the accessor.

We will start with the wrapper (Listing 2).

Here we have a class that can be initialized with a value, but does not allow anybody to access its value. The templated constructors are used to avoid temporary objects during intialization if **`T`** is a struct.

The accessor is a very simple smart pointer and will then look like Listing 3.

```
template<typename T>
class unique_access_guard : noncopyable
{
public:
  unique_access_guard()
    : value() {}
  unique_access_guard(const T& value)
    : value(value) {}

  template<typename A1>
  unique_access_guard(const A1& arg1)
    : value(arg1) {}
  template<typename A1, typename A2>
  unique_access_guard(const A1& arg1,
                      const A2& arg2)
    : value(arg1, arg2) {}
  // Add constructors with more arguments,
  // or use C++0x variadic templates
private:
  friend class unique_access<T>;
  std::mutex mutex;
  T value;
};
```

Listing 2

```
class Person
{
public:
  std::string GetName() const
  {
    std::unique_lock<std::mutex> lock(mutex);
    return name;
  }
  void SetName(const std::string& newName)
  {
    std::unique_lock<std::mutex> lock(mutex);
    name = newName;
  }
private:
  std::mutex mutex;
  std::string name;
};
```

Listing 1

```
template<typename T>
class unique_access : noncopyable
{
public:
  unique_access(unique_access_guard& guard)
    : lock(guard.mutex),
      valueRef(guard.value)
  {}

  T& operator* () { return valueRef; }
  T* operator-> () { return &valueRef; }

private:
  std::unique_lock<std::mutex> lock;
  T& valueRef;
};
```

Listing 3

Bjørn Reese is a software engineer at Bang & Olufsen and has been developing C++ programs for more than a decade. He can be contacted at breese@users.sourceforge.net.

Let us continue with an example of how this works. We will assume that we also have defined a **`const_unique_access`** class. This can be implemented using the slicing technique. (Listing 4.)

Compared to the **`Person`** class, we cannot access the **`name`** member variable without locking it, so we cannot forget the lock when we are extending the class with a new member function.

there are several of the less common use cases for locking that they do not support

```
class SafePerson
{
public:
  std::string GetName() const
  {
    const_unique_access<std::string>
       name(nameGuard);
    return *name;
  }

  void SetName(const std::string& newName)
  {
    unique_access<std::string> name(nameGuard);
    *name = newName;
  }

private:
  unique_access_guard<std::string> nameGuard;
};
```

Listing 4

What happens if our class has many member variables? That depends on our needs. If the member variables are independent from each other, then we can declare more access guards and use separate accessors for them. A more common scenario is when we want to lock all member variables when we access one or more of them. In that case we can place them in a struct. This struct can be nested in the class, as shown in Listing 5.

Another common scenario is the use of private helper member functions. With the normal mutex and lock method we must make sure that the member variables are locked before we call the helper function. The usual way to ensure this is to document the precondition in a comment.

With the access guards we can make this precondition explicit. There are two solutions. The first, shown in the **HelperPerson** class, is to pass the **const_unique_access** object by reference to the helper function. Now it is not possible to call the helper function unless we have an accessor and hence a lock. (See Listing 6.)

The other solution, which is shown in the **MemberHelperPerson** class, is to let the helper be a member function of the **Member** struct. As we only can deference the **Member** struct when we have an accessor, we are guaranteed that any member function in the **Member** struct only runs when we have a lock. (Listing 7)

All of the above has been illustrated with unique access to member variables. The framework can be easily extended to encompass shared access as well. We need a **shared_access_guard** that embeds a **std::shared_mutex**, and a shared_access accessor to gain shared access to the member variables. If we want unique access to these sharable member variables, then we can use the **unique_access** accessor on the **shared_access_guard**.

```
class SafeMemberPerson
{
public:
  SafeMemberPerson(unsigned int age)
     : memberGuard(age) {}

  std::string GetName() const
  {
    const_unique_access<Member>
       member(memberGuard);
    return member->name;
  }

  void SetName(const std::string& newName)
  {
    unique_access<Member> member(memberGuard);
    member->name = newName;
  }

private:
  struct Member
  {
    Member(unsigned int age) : age(age) {}

    std::string name;
    unsigned int age;
  };
  unique_access_guard<Member> memberGuard;
};
```

Listing 5

Const correctness is used to ensure that **const_unique_access** and **shared_access** can only read member variables, and only **unique_access** can write them.

The overhead of using accessor guards is minimal. The construction of an accessor is the same as a **std::unique_lock** plus the assignment of a reference. The subsequent use of an accessor is the extra level of indirection from **operator->** and **operator***. Some of this will be removed by the optimizer though.

The advantages of using access guards is:

- Guarded member variables can only be accessed when they are locked. The compiler will alert us if we attempt to do otherwise.
- The precondition on helper functions becomes explicit, and the compiler will alert us if we use it incorrectly.
- The mental model is simple – if we have the accessor then we have the lock – so developers are less likely to make errors with it.
- The guards also serves as documentation for which member variables are protected by what mutex. If a class has more than one access guard then that could indicate that the class has too many responsibilities and thus is a good candidate for refactoring.

The access guards have been designed to be simple. This means that there are several of the less common use cases for locking that they do not support.

Their limitations are:

- A variable can only be protected by at most one guard. It is not possible to have one guard to directly protect, say, the variables alpha and bravo, and another guard to protect bravo and charlie.
- The accessors are not full smart pointers, and should be made non-copyable, so we cannot pass them around, except by reference as shown in the `HelperPerson` class. This omission is a deliberate trade-off to avoid the added complexity needed to ensure that the accessor does not live longer than the guard it is accessing.
- There is no support for deferred locking (`std::defer_lock_t`) so we cannot use the `std::lock()` algorithm to avoid potential deadlocks. Hence if we have two or more access guards in a class, then we must make sure that they are always locked in the same order to avoid deadlocks.
- There is no support for early unlocking (like `std::unique_lock<T>::unlock()`) so we cannot have partially overlapping locks. For instance, if we have a class that contains a callback function, then we may want to lock the member variables with one mutex, and the execution of the callback with another mutex to allow that the callback can call functions that accesses the member variables. In this case we need to (1) lock the member variables, (2) use the member variables, (3) lock the callback, (4) unlock the member variables, (5) execute the callback, (6) unlock the callback, and (7) exit from the member function.
- There is no support for a couple of more advanced locking mechanisms, such as timed mutexes, recursive mutexes, tentative locking (`try_lock()`), or upgrading ownership.

The technique described here has some commonality with the `SynchronizedValue::Updater` [Dobbs].

The main differences are that the guards do not allow access to the variables, as `SynchronizedValue` does, and the accessors can used with different guards, thus separating the type of access (exclusive or shared) which is a property of the accessor, not the guard. ■

Reference

[Dobbs] http://www.drdobbs.com/cpp/225200269

```
class HelperPerson
{
public:
  HelperPerson(unsigned int age)
    : memberGuard(age)
    {}

  std::string GetName() const
  {
    const_unique_access<Member>
       member(memberGuard);
    Invariant(member);
    return member->name;
  }

  void SetName(const std::string& newName)
  {

    unique_access<Member> member(memberGuard);
    Invariant(member);
    member->name = newName;
  }

private:
  void Invariant(
     const_unique_access<Member>& member) const
  {
    if (member->age < 0)
       throw std::runtime_error(
       "Age cannot be negative");
  }

  struct Member
  {
    Member(unsigned int age) : age(age) {}

    std::string name;
    unsigned int age;
  };
  unique_access_guard<Member> memberGuard;
};
```

Listing 6

```
class MemberHelperPerson
{
public:
  MemberHelperPerson(unsigned int age)
    : memberGuard(age) {}

  std::string GetName() const
  {
    const_unique_access<std::string>
       member(memberGuard);
    member->Invariant();
    return member->name;
  }

  void SetName(const std::string& newName)
  {
    unique_access<std::string>
       member(memberGuard);
    member->Invariant();
    member->name = newName;
  }

private:
  struct Member
  {
    Member(unsigned int age) : age(age) {}

    void Invariant() const
    {
     // We always have unique access to the member
      // variables here
      if (age < 0)
        throw std::runtime_error(
           "Age cannot be negative");
    }

    std::string name;
    unsigned int age;
  };
  unique_access_guard<Member> memberGuard;
};
```

Listing 7

An Introduction to Test Driven Development

TDD is too often thought to be all about Unit Tests. Paul Grenyer works through an example to show the bigger picture.

There are a *lot* of introductory articles for Test Driven Development (TDD). So why, might you ask, am I writing yet another? The simple answer is because I was asked to by Allan Kelly, who wanted a piece for a small book he gives to his clients. I was happy to oblige, but writing about TDD is difficult. In fact if Allan hadn't wanted an original piece he could print as part of his book, I would have suggested he just get a copy of *Test Driven Development* by Kent Beck. The main difficulty is coming up with a suitably concise, yet meaningful, example and Beck has already done this.

Allan was also quite keen for me to publish elsewhere, so I chatted the idea over with Steve Love, the editor of the ACCU's *C Vu* magazine to see if he thought the readers would be interested in yet another TDD article. He said they probably would be as long as I thought carefully about how I wrote it. I thought this over for a long while. The majority of introductory TDD articles, at least the ones I have read, focus on unit testing. A recently completed ACCU Mentored Developers project read through *Growing Object Orientated Software Guided by Tests* (known as GOOS) by Freeman & Pryce [Freeman]. They focus on starting a simple application with acceptance tests and only writing unit and integration tests when the level of detail requires it or the acceptance tests become too cumbersome. However, it is a big book, so I decided to try and condense what I saw as the most important points into an introductory article and this is what you see before you. I hope you find it as useful and fun to read as I did to write.

What?

Test Driven Development is a way of developing software by writing tests before you write any production code. Hence it is sometimes referred to as Test First Development. The first and definitive book on the subject is *Test Driven Development* by Kent Beck [Beck02], which focuses on Unit Testing. However, as Steve Freeman and Nat Pryce tell us in GOOS, TDD can and should be employed at all levels, including integration and acceptance tests.

Unit tests

Unit tests are automated tests that test a single unit, such as a class or method. Usually the class under test is instantiated with any dependent collaborators mocked out (I'll describe mock objects later) and then tests are run against it. In some ways it is easier to describe what a unit test is not, and Michael Feathers has done this very well:

> A test is not a unit test if:
>
> - It talks to the database
> - It communicates across the network
> - It touches the file system
> - It can't run at the same time as any of your other unit tests
> - You have to do special things to your environment (such as editing config files) to run it.

decoupled code for just this reason, collaborators can be mocked out when a class that uses them is tested.

Integration tests

Integration tests are automated tests that test the interaction between at least two different parts of a system. For example a test which checks that a data access layer correctly loads data from a database is an integration test.

Acceptance tests

Rachel Davies describes acceptance tests as '...scripts – manual or automated – that detail specific steps to test a feature.' [Davies] Generally acceptance tests run against a deployed system to check that it behaves in a way that will be accepted by as a complete or partially complete feature.

Why?

In *Working Effectively With Legacy Code*, Michael Feathers [Feathers] tells us that:

> Code without tests is bad code. It doesn't matter how well written it is; how pretty or object orientated or well encapsulated it is. With tests we can change the behaviour of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.

There are many reasons why you should develop code with TDD, but this is the most important. The way I like to describe it is with the analogy of nailing down a floor in the dark. You've done one end of a plank and now you're nailing down the other end. As it's dark you can't see that the first end is slowly being prised up as you bang in the nails on the second end. Code without tests is the same. Without tests you cannot see how a new change to the code is affecting existing code. As soon as a test passes it instantly becomes a regression test that guards against breaking changes. In our analogy it's like turning the light on or getting a friend to watch the other end of plank:so you know if the floorboard was being prised up.

Another advantage of test driven development, especially with unit testing, is that you tend to develop highly decoupled code. When writing unit tests, you generally want to exercise a very small section of code, often a single class or single method, to run the tests against. It becomes very difficult if you have to instantiate a whole slew of other classes or access a file system or database that your class under test relies on (this is slightly different when writing integration tests that often do talk to a file system or database). There are a number of techniques for reducing dependencies on collaborators, mostly including the use of interfaces and Mock Objects. Mock objects are implementations of interfaces for testing purposes whose behaviour is under control of the test, so it can focus solely on the behaviour of the code under test. Writing against interfaces promotes

Paul Grenyer An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com.

stop developing once the test passes and then refactor to remove any duplication

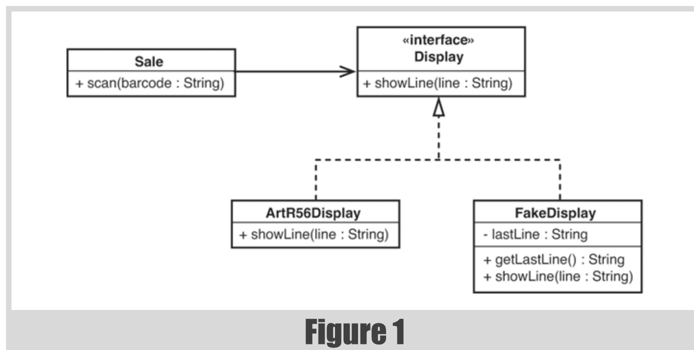

Figure 1

In *Working Effectively with Legacy Code*, Michael Feathers has an excellent example of mocking a collaborator.

Figure 1 shows a point-of-sale class, called `Sale`, that has a single method, `scan`, which accepts a barcode. Whenever `scan` is called the item relating to the barcode must be displayed, along with its price on a cash register display. If the API for displaying the information is buried deep in the `Sale` class, it is going to be very difficult to test, as we'd have to find some way of hooking into the API at runtime. Luckily the `Sale` class takes a `Display` interface which it uses to display the information. In production an implementation of the interface that talks to the cash register API is used. When testing, a `Fake` object (I'll talk more about fake objects later) is used to sense what the `Sale` object is passing to the display interface.

If you write a test for production code *before* the production code itself, stop developing once the test passes and then refactor to remove any duplication, you end up with just enough code to implement the required feature and no more. It is very important to take time to refactor, thus ensuring that complexity and repetition is reduced. Again if you come to change code in the future and it has been refactored to remove duplication, the chances are you will only have to change the code in one place, rather than several. Code that is developed with TDD, including refactoring, generally yields a better design which stands up well in the face of change.

There have been a number of case studies measuring the benefits of TDD, including 'Realizing quality improvement through test driven development' by Nagappan *et al* [Nagappen]. The key point is that following a study of two teams at Microsoft, one using TDD and one not, it was shown that the team following TDD achieved 75% fewer defects for only 15% more effort. It is true that TDD can take a little longer initially, but generally time is saved by the next release as there are fewer bugs to fix and those that do creep in are easier to find and resolve. In *Software Engineering Economics* [Boehm81] Barry Boehm tells us that that the cost of fixing a defect increases very quickly the later it is found. This often leads to the paradoxical observation that going 'faster' takes longer to get to release as there's a much longer spell of fixing the bugs. By avoiding them being there in the first place you save that effort with a very quick payback. Developers who use TDD will also find that they spend significantly less time in the debugger as their tests highlight where the bugs are.

The best way to find out why TDD is so good is to try it.

How?

The fundamental steps in test driven development are:

0. Decide what to test
1. Write a test
2. Run all the tests and see the new one fail
3. Write some code
4. Run all the tests and see the new one pass
5. Refactor
6. Run all the tests and see them pass

Repeat.

The first step isn't strictly a TDD step, but it's very important to know what you are going to test before write a test. For acceptance tests this will come from the user story or specification that you have. For integration tests you'll be looking at how two things, such as your data access layer and a database, interact with each other. For unit tests you want to test that the interface does what is expected.

The first real step is to write a test for the new feature you want to implement. Initially the test will not pass. In a lot of cases it wont even compile as you haven't written the code it is going to test yet.

The next step is to write the code to make the test compile, run all the tests and see the new one fail. Then write the code to make the test pass. Write the simplest and most straight forward code you can to make the test pass. Write no more code than it takes to make the test pass. If you write more code than is required you will have code that can change without breaking a test, so you wouldn't know when a change, or in a lot of cases a bug, is introduced, which defeats the goal of TDD. Run the tests again and see them all pass.

The next step is to examine both the test and the production code for duplication and remove it. The final step is to make sure all the tests still pass. Refactoring is about removing duplication and should not change what the code does. Then go back to step 1 and *repeat*. This is also known as the Red/Green/Refactor cycle in reference to the red (tests failed) and green (tests pass) bar that is a feature of many testing frameworks.

Many people think that TDD is only applicable to greenfield projects. In his blog post 'Implications of the Power Law' [Kelly], Allen Kelly tells us that this is not the case:

> The power law [PowerLaw] explains this: because most changes happen in a small amount of code, putting tests around that code now will benefit you very soon. Most of the system is not changing so it doesn't need tests. Of the 1 million lines in a project perhaps only 10,000 changes regularly. As soon as you have 1,000 covered you see the pay back.

Michael Feathers has written a whole book (the already mentioned *Working Effectively with Legacy Code*), and it's a big book, of techniques

Write the simplest and most straight forward code you can to make the test pass

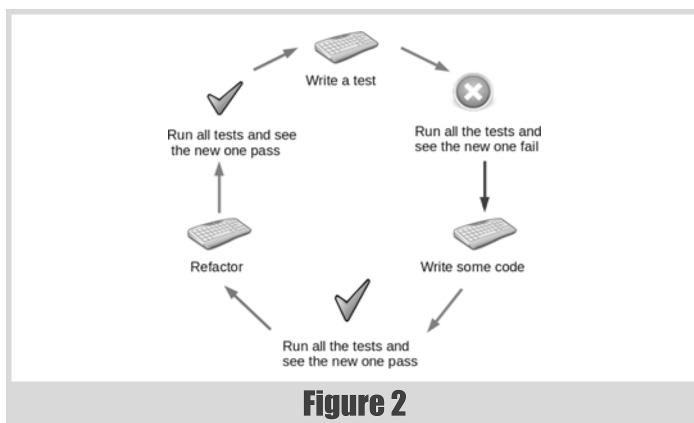

Figure 2

for getting existing code under test. Again, try it for yourself and you will soon see the benefits.

An example

I am going to use a simple program, called DirList, that generates a formatted directory listing for an example of how to develop a project test first, using TDD. One of the prime reasons for choosing it is to show how to deal with programs that rely on external resources and have external effects and so are traditionally hard to test. Here the external resource is the filesystem and the external effect is writing to standard out.

Here's a simple set of stories to describe how the application works:

- When DirList is executed with no arguments, the current directory will be the first message displayed in the output, preceded by **`"Directory: "`**.
- When DirList is executed and a path is specified as an argument, it will be the first message displayed in the output, preceded by **`"Directory: "`**.
- When DirList is executed without any arguments it will list all the files in the current directory.
- When DirList is executed and a path is specified as an argument it will list all the files in the specified directory.
- When DirList is executed each file will be listed on a separate line along with size, date created and time created.
  - The file name must be left justified and the other details right justified with a single space in between each detail.
  - If the file name would push the whole display line to greater than 40 characters, it must be truncated leaving a single space before the first detail.
  - The date format is dd/mm/yyyy
  - The time format is HH:MM

This is a good example project as it is relatively easy to write acceptance tests that invoke an executable with varying arguments and to examine the output. There is also scope for unit and integration tests, as well as the use of mock objects.

Every new project should start with acceptance tests. The easiest way to start is with a small simple acceptance test that exercises the 'system'. In DirList's case the system is an executable.

Step 1: Write a test

Here is a test for the first story, written in Python[1]:

```
class DirListAcceptanceTests(unittest.TestCase):
def
testThatCurrentDirectoryIsFirstMessageInOutputIfNoA
rgumentsSupplied(self):
  output = Execute(DIRLIST_EXE_PATH,'')
  self.assertEquals(0, output.find("Directory: " +
os.getcwd() + "\n" ), output)
```

In this test **`DIRLIST_EXE_PATH`** is a variable holding the path to the DirList executable and **`Execute`** is a method that calls the executable with the command line arguments passed in as the second parameter and returns the output. The single **`assert`** checks that the output *begins* with **`"Directory: "`** followed by the current working directory and a line feed. The third parameter, **`output`**, is there so that if the test fails the actual output from the executable is displayed for debugging.

Step 2: Run all the tests and see the new one fail

`Execute` throws an exception if the executable does not exist, so although this test compiles, it fails at runtime as the DirList executable hasn't been built yet.

Step 3: Write some code

DirList is a normal C# command line application:

```
namespace DirList
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

Building the code into an executable and setting **`DIRLIST_EXE_PATH`** to its path (e.g. `<bin dir>/DirList.exe`) will get the test a little further. It is important to rerun tests after every change, no matter how small. **`Execute`** can now find the executable and execute it, but there is no output so the assert fails. The simplest way to get the test to pass is to send the

1. If you're not familiar with unit testing in Python, *Dive Into Python* by Mark Pilgrim [Pilgrim] provides an excellent introduction.

current working directory to standard out with `"Directory: "` prepended and a carriage return appended:

```
static void Main(string[] args)
{
  Console.WriteLine(
  string.Format("Directory: {0}",
     Directory.GetCurrentDirectory()));
}
```

Step 4: Run all the tests and see the new one pass

Now when the executable is built and the test run, it passes and we have our first passing acceptance test that tests our little system from end-to-end.

Step 5: Refactor

Normally at this point you would go back and examine the test and the production code and remove any duplication. However, at this early stage there isn't any.

Step 6: Run all the tests and see them pass

As we haven't had to refactor, there's no need to run the tests again. However, it's always good to get that green bar feeling, so you can run them again anyway if you want to.

Repeat...

Step 1: Write a test

So we move on to write the next test:

```
def
testThatSuppliedDirectoryIsFirstMessageInOutputIfPa
thSupplied(self):
    dir = tempfile.mkdtemp()
    try:
      output = Execute(DIRLIST_EXE_PATH,dir)
      self.assertEquals(0,
         output.find("Directory: " + dir + "\n" ),
         output)
    finally:
      os.rmdir(dir)
```

This test is for the second story. It creates a temporary directory, passes its path as a command line argument to `Execute` and then asserts that the directory path is the first message in the output. The temporary directory must be cleaned up by the test so, as asserts throw when the condition they test for fails, the statement to remove it goes in a finally block. If the temporary directory wasn't cleaned up, running the tests would eventually start to fill your disk up. Creating and cleaning up a specific directory for each test, as opposed to using the same directory each time, means that the tests could be easily run in parallel.

Step 2: Run all the tests and see the new one fail

Of course, this new test fails as DirList sends the current directory to standard output. So we must look at the DirList `Main` method again to find the smallest and simplest change we can make to get the test to pass without breaking the existing tests.

Step 3: Write some code

```
static void Main(string[] args)
{
  if (args.Length == 1)
    Console.WriteLine(string.Format(
      "Directory: {0}", args[0]));
  else
    Console.WriteLine(
      string.Format("Directory: {0}",
         Directory.GetCurrentDirectory()));
}
```

Here the number of arguments passed in is examined and if there is only one it is assumed to be the path to use, otherwise the current directory is used. This really is the simplest implementation to get the test to pass. We're not taking any error handling into account or allowing for the first argument not to be the path. How DirList behaves under error conditions might be included in future stories.

Step 4: Run all the tests and see the new one pass

After DirList is compiled, the tests must be run and they all pass.

Step 5: Refactor

Stop! We're not finished yet. Is there some duplication we can refactor away? Yes, there is. The format of the output is repeated in two places. That's easily refactored:

```
private static string getDirToList(string[] args)
{
  if (args.Length == 1)
    return args[0];
  else
    return Directory.GetCurrentDirectory();
}
static void Main(string[] args)
{
  var dirToList = getDirToList(args);
  Console.WriteLine(string.Format("Directory: {0}",
     dirToList));
}
```

The duplication is refactored by moving the code that determines which path to use to a new method and formatting the result.

Step 6: Run all the tests and see them pass

The tests will tell us if a mistake has been made during the refactor, that's what they are for, and running them again reassures us that nothing has broken.

This is an example of the complete TDD cycle. We added a test, watched it fail, wrote some code, watched the test pass, and then refactored to remove duplication.

Repeat...

The third story is concerned with making sure all the files in the directory being listed are present in the output from DirList. What we need is another test.

Step 1: Write a test

```
def
testThatAllFilesInCurrentDirectoryAreIncludedInOutp
utIfNoArgumentsSupplied(self):
  output = Execute(DIRLIST_EXE_PATH,'')
  dirList = os.listdir(os.getcwd())
  for fname in dirList:
    if os.path.isfile(os.path.join(os.getcwd(),
       fname)):
      self.assertTrue(fname in output,
         "Missing file: " + fname)
```

Again, DirList is invoked without any command line arguments. Then a list of all the directories and files in the current directory is iterated through, the files are identified and and their presence checked for in the output.

Step 2: Run all the tests and see the new one fail

Naturally, the new test fails. To make it pass we need to modify `Main` again.

Step 3: Write some code

```
static void Main(string[] args)
{
```

```
  var dirToList = getDirToList(args);
  Console.WriteLine(string.Format("Directory: {0}",
     dirToList));
  var dirInfo = new DirectoryInfo(dirToList);
  foreach (var fileInfo in dirInfo.GetFiles())
    Console.WriteLine(fileInfo.Name);
}
```

Step 4: Run all the tests and see the new one pass

Run the tests, which pass.

Step 5: Refactor

Is there any duplication that can be refactored away? Yes, there is a little. There are two calls to `Console.WriteLine`. The underlying runtime will probably buffer, but that shouldn't be relied upon. The code could be refactored to use a single call with a `StringBuilder`. However this is premature optimisation and could cause DirList to use more memory that it really needs.

Step 6: Run all the tests and see them pass

Following the refactor, rerun the tests to make sure nothing has been broken. When carrying out a significant change, like changing the way text is formatted or output, it is easy to make a simple mistake.

Repeat...

The next step is to write a test that lists the files in a directory specified as a command line argument. To do this, as well as creating a temporary directory, some temporary files need to be created and checked against the output from DirList. I'll leave that as an exercise for the reader. If you do write the test you should find that you do not need to add any code to make it pass. This should make you question whether it is really necessary. In truth it is. The test describes and checks for some behaviour that may get broken by a change in the future. Without the test you wouldn't know when the code broke.

You may be thinking that the next step is to return to the acceptance tests and start adding tests for the next story, which describes the format of the output. You could do that, but it would be extremely cumbersome in terms of keeping the tests in line with the changing files in the current directory or creating temporary files of the correct size and creation date. You would end up with quite brittle tests as every time a file was added to the directory, removed from the directory or its size was modified the test would break. It's far easier to write a set of unit tests instead. That way you can fake the file system and have complete control over the 'files'.

The formatting story describes one of several possible ways of formatting the output of the directory listing. It is of course possible that other formatting styles may be required in the future, so the obvious implementation is the STRATEGY pattern [Strategy]. At the base of the strategy pattern is an interface that provides a point of variation that can be used to customise the format :

```
public interface Formatter
{
  string Format(IEnumerable<FileInfoMapper> files);
}
```

Another advantage in using an interface is that it makes the formatting strategy easy to mock should we need to for future tests (I'll discuss Mock Objects shortly). Implementations of `Formatter` take a collection of `FileInfoMapper` objects and return a formatted string. You may have been expecting it to take collection of `FileInfo` objects. To write unit tests for the format strategies you need to be able to create a collection of `FileInfo` objects and set the name, size, creation date and creation time. A quick look at the documentation for `FileInfo` [FileInfo] tells us that we can give a `FileInfo` object a name, but we cannot set the size, creation date or creation time. These properties are set when a `FileInfo` object is populated by examining a real file on the disk. We could write an integration test (I'll describe integration tests shortly too) that creates lots of different files of different sizes and somehow fix the creation date and time, but this would be cumbersome. A better solution is to create an interface that provides the properties we want from the `FileInfo` object (see Listing 1) and then write an adapter [Adapter] that extracts the properties from the `FileInfo` object in production and a fake [Fake] object, that implements the same interface, which can be used in testing. Before we look at the formatter strategy implementations, unit tests and fake, we need to satisfy ourselves that an adapter will work. The best way to do this is to write one.

```
public interface FileInfoMapper
{
  string Name { get; }
  long Length { get; }
  DateTime CreationTime {get;}
}
```

Listing 1

At this point I am switching from writing acceptance tests in Python to writing an integration test in C# with NUnit [NUnit]. Integration tests usually run in the same process as the code they are testing, so using the same runtime, even if not the same language, is helpful. NUnit is a testing framework for .Net, which is usually used for unit testing but is also ideal for integration testing. An integration test tests the the interaction between one or more units. Usually a unit is a class, or the interaction between a unit and an external resource such as a database, file system, network etc. In the next test we're testing the interaction between the `FileInfo` class and the `FileInfoAdapter` class. It just so happens that this also requires interaction with the file system. Integration tests can be developed test first, just like acceptance tests.

I hope that by now you understand the 6 steps of TDD, so from now on I will assume you're following them as the code develops. (See Listing 2.)

This NUnit integration test fixture has a pair of setup and tear down methods and a test method. A test fixture is a C# class denoted by the

```
[TestFixture]
public class FileInfoAdapterTest
{
  private string tempFile;

  [SetUp]
  public void SetUp()
  {
    tempFile = Path.GetTempFileName();
    var file = new StreamWriter(tempFile);
    file.WriteLine("Test Driven Development!");
    file.Close();
  }

  [TearDown]
  public void TearDown()
  {
    File.Delete(tempFile);
  }

  [Test]
  public void testFileInfoMappings()
  {
    var fileInfo = new FileInfo(tempFile);
    var adapter = new FileInfoAdapter(fileInfo);
    Assert.That(adapter.Name,
       Is.EqualTo(fileInfo.Name));
    Assert.That(adapter.Length,
       Is.EqualTo(fileInfo.Length));
    Assert.That(adapter.CreationTime,
       Is.EqualTo(fileInfo.CreationTime));
  }
}
```

Listing 2

```
public class FileInfoAdapter : FileInfoMapper
{
  private readonly FileInfo fileInfo;

  public FileInfoAdapter(FileInfo fileInfo)
  {
    this.fileInfo = fileInfo;
  }

  public string Name
  {
    get { return fileInfo.Name; }
  }

  public long Length
  {
    get { return fileInfo.Length; }
  }

  public DateTime CreationTime
  {
    get { return fileInfo.CreationTime; }
  }
}
```

Listing 3

[TestFixture] attribute. When NUnit sees the **[TestFixture]** attribute on a class it knows it contains tests that it can run. Setup methods, denoted by the **[SetUp]** attribute, are run before every test method in the test fixture. Test methods are denoted by the **[Test]** attribute and contain tests. Tear down methods, denoted by the **[TearDown]** attribute are run after every test method.

FileInfoAdapterTest creates a temporary file, with some content, in the setup method before each test. It uses the temporary file to create a **FileInfo** object and uses it to create a **FileInfoAdapter** and then asserts each of the properties against the **FileInfo** object in the test method. Finally it deletes the temporary file in the tear down method. I have said a couple of times now that creating files on disk for the purposes of testing is cumbersome, and it is. However, in this scenario we only need a single file that is easily created and destroyed. This test won't even compile, let alone pass, so we need to write some code to make it pass.

As it was so simple I went ahead and wrote the final implementation of **FileInfoAdapter** straight away, rather than in small steps (Listing 3).

This is perfectly acceptable. You do not have to go through painstakingly slow steps of implementation. If you can see the solution and you have the tests written, just implement it.

Now we're ready to write the unit tests for the formatters. As was hinted at earlier, a unit test is a test that tests a single unit that does not interact with other units or resources. It may, however, interact with mocked dependencies as we'll see shortly. The first test is in Listing 4.

This test simply checks that all the required details are present in the output. The only part of the test that is not shown is the **BuildFileInfoMapper** method which is defined shown in Listing 5.

It's just a method that creates a **FakeFileInfoMapper** from the arguments supplied (Listing 6).

A fake is a type of mock object that implements a dependency for testing purposes. A mock object is a test class that replaces a production class for testing purposes. A mock object is usually a simpler version of the production class used to help break a dependency of the class under test or used to sense how the class under test interacts with the production version of the mock. In this case **FakeFileFinfoMapper** is a **FileInfoMapper** that allows us to specify **FileInfo** details without having to create a real **FileInfo** object or a file on disk. You'll also notice that the asserts in the test all have messages (See Listing 7)..

```
[Test]
public void
testThatAllDetailsArePresentInFormat()
{
  const string NAME = "file.txt";
  const long LENGTH = 10;
  var NOW = DateTime.Now;
  var NowDate = NOW.ToShortDateString();
  var NowTime = NOW.ToShortTImeString();
  var files = new List<FileInfoMapper> {
    BuildFileInfoMapper(NAME, LENGTH, NOW) };
  var formatter = new DetailsFormatter();
  var output = formatter.Format(files);

  Assert.That(output,
    Is.StringContaining(NAME),
    "Name");
  Assert.That(output,
     Is.StringContaining(LENGTH.ToString()),
     "Length");
  Assert.That(output,
     Is.StringContaining(NowDate),
     "Date");
  Assert.That(output,
     Is.StringContaining(NowTime),
     "Time");
}
```

Listing 4

```
private static FileInfoMapper
BuildFileInfoMapper(
  string name, long length,
  DateTime creationTime)
{
  return new FakeFileInfoMapper(name, length,
                                creationTime);
}
```

Listing 5

Some people believe that a test method should only have a single assert. In a lot of cases this is a very good idea as it helps track down exactly where any failure is more easily. It could easily be argued, for that very reason, that you should have separate test methods for asserting that name, length, date and time are all present in the output string. In most cases I feel that

```
private class FakeFileInfoMapper : FileInfoMapper
{
  private readonly string name;
  private readonly long length;
  private readonly DateTime creationTime;

  public FakeFileInfoMapper(string name,
     long length, DateTime creationTime)
  {
    this.name = name;
    this.length = length;
    this.creationTime = creationTime;
  }

  public string Name
    { get { return name; } }
  public long Length
    { get { return length; } }
  public DateTime CreationTime
    { get { return creationTime; } }
}
```

Listing 6

```
Assert.That(output,
   Is.StringContaining(NAME), "Name");
Assert.That(output,
   Is.StringContaining(LENGTH.ToString()),
   "Length");
Assert.That(output,
   Is.StringContaining(NOW.ToShortDateString()),
   "Date");
Assert.That(output,
   Is.StringContaining(NOW.ToShortTimeString()),
   "Time");
```

Listing 7

assert messages should not be used either. They're like comments and we all know that people write (usually unnecessary) comments describing code, later the code gets changed, but the comment is not updated making it invalid and often misleading. That risk is present here, but the advantage of having a single concise, clear test with multiple asserts that are easily identified in failure by the short message, outweighs the risk. My general rule is to have each test method only test one thing, except in circumstances like this where pinpointing the cause of a failure can be easily identified with a message. Currently the test won't build, let alone pass as there is no `DetailsFormatter` class, so we need to write it:

```
public class DetailsFormatter : Formatter
{
  public string Format(
     IEnumerable<FileInfoMapper> files)
  {
    return "";
  }
}
```

The test will now build, but the test will not pass as the `Format` method returns a blank string and the first assert is expecting a string with a file name in it. The simplest implementation to get it to pass looks like Listing 8.

Looking at the test you could argue that creating a `StringBuilder` and iterating through all the `FileInfoMapper` objects is not the simplest way to get the test to pass. The alternative is to just get the first `FileInfoMapper` from the enumerable, as we know there is only one, and return a string built from that. However the code to get the first item from an enumerable is quite verbose and it's easier to write the `foreach` expression, especially knowing that we'll need it later. If we return from the `foreach` we need a second return at the end of the method to keep the compiler happy, so it's just as easy to use a `StringBuilder`, again knowing that we'll need it later when we have multiple `FileInfoMapper` objects. This thinking ahead and implementing slightly more functionality than is strictly necessary could be considered 'gold plating' and goes against the TDD principle of not implementing any more functionality than is needed to make the test pass. However, TDD is not a straight jacket and in a few cases a little more functionality is ok. The second test asserts that the output is of the correct length (Listing 9).

```
public string Format(
   IEnumerable<FileInfoMapper> files)
{
  var output = new StringBuilder();

  foreach(var file in files)
    output.Append(
      string.Format("{0} {1} {2} {3}",
        file.Name,
        file.Length,
        file.CreationTime.ToShortDateString(),
        file.CreationTime.ToShortTimeString()));
  return output.ToString();
}
```

Listing 8

```
[Test]
public void testThatFormatIsCorrectLength()
{
  const string NAME = "file.txt";
  const long LENGTH = 10;
  var NOW = DateTime.Now;

  var files = new List<FileInfoMapper> {
    BuildFileInfoMapper(NAME, LENGTH, NOW) };
  var formatter = new DetailsFormatter();
  var output = formatter.Format(files);

  Assert.That(output.Length, Is.EqualTo(40));
}
```

Listing 9

The test builds, but it does not pass as the string returned from the `Format` method is not long enough. Also there is a lot of duplication with the previous test. Duplication in test code is bad for the same reason it's bad in production code. If you have to change the duplicated code, you have to change it everywhere it's duplicated. If you've refactored the duplication away, you only have to make the change in one place. We'll look at that in a moment once we've got the test to pass. The easiest way to get the test to pass is just to pad the output:

```
  return output.ToString().PadRight(40);
```

Of course this doesn't give the the final format that we want, but it's enough to get *this* test to pass. I don't like magic numbers. Magic numbers are numbers or string literals in the code that don't clearly explain what they are. Here the number `40` represents the row length. It could be changed to a static member variable, but it makes more sense to pass it in through the constructor in case it's ever changed in the future. It also makes the test more expressive (Listing 10).

Now that the test is passing we can look at removing duplication by introducing a setup method (see Listing 11).

```
public class DetailsFormatter : Formatter
{
  private readonly int rowLength;

  public DetailsFormatter(int rowLength)
  {
    this.rowLength = rowLength;
  }
  public string Format(
     IEnumerable<FileInfoMapper> files)
  {
    ...
    return output.ToString().PadRight(rowLength);
  }
}
private const int ROW_LENGTH = 40;
...

[Test]
public void testThatFormatIsCorrectLength()
{
  ...
  Assert.That(output.Length,
     Is.EqualTo(ROW_LENGTH));
}
```

Listing 10

Before we go any further the tests must be run again to make sure nothing has been broken. Then it's back to the tests, of which there are at least three more to write.

```
testThatFileNameAtBeginningAndDetailsAtEnd
testThatLongFileNamesAreTruncatedToFitIntoRowLength
testThatMultipleFilesAreDisplayedOnSeperateLines
```

They all involve writing a new failing test, writing the code to make it pass and then refactoring away duplication. These are all steps that we have seen a number of times already, so I'll leave them as exercises for the reader. Therefore all that remains is to integrate the `DetailsFormatter` into `Main` (Listing 12) and rerun the acceptance tests.

They pass! Although they didn't pass the first time I ran them. One of the files in the current directory was so long its name got truncated and appeared to be missing, therefore failing one of the tests. This was easily fixed by increasing `ROW_LENGTH`, rebuilding and running the acceptance tests again. It could be argued that this breaks the original specification and that the test should be updated instead. However, I prefer to look at this as the TDD process highlighting a flaw in the original specification and making a change to it. Agreed by the relevant stakeholder of course.

Finally

In this whirlwind tour of Test Driven Development I have discussed the what, why and hows. We've been through a very simple, yet all encompassing example of developing an application test first from the outside in. From acceptance tests that run outside the system to unit and integration tests that run on the internal units of the system. I hope it has encouraged you to try TDD for yourself and to read the books I have mentioned, and others, for an deeper look. ■

```
private const int ROW_LENGTH = 40;
private const string NAME = "file.txt";
private const long LENGTH = 10;
private DateTime NOW = DateTime.Now;
private List<FileInfoMapper> files;
private Formatter formatter;
[SetUp]
public void setUp()
{
  files = new List<FileInfoMapper> {
    BuildFileInfoMapper(NAME, LENGTH, NOW) };
  formatter = new DetailsFormatter(ROW_LENGTH);
  }
[Test]
public void
  testThatAllDetailsArePresentInFormat()
{
  var output = formatter.Format(files);
  Assert.That(output, Is.StringContaining(NAME),
     "Name");
  Assert.That(output,
     Is.StringContaining(LENGTH.ToString()),
     "Length");
  Assert.That(output,
     Is.StringContaining(NowDate),
     "Date");
  Assert.That(output,
     Is.StringContaining(NowTime),
     "Time");
}
[Test]
public void testThatFormatIsCorrectLength()
{
  var output = formatter.Format(files);
  Assert.That(output.Length,
     Is.EqualTo(ROW_LENGTH));
}
```

Listing 11

Acknowledgments

My thanks go to Allan Kelly for suggesting that I write a piece on Test Driven Development. I've thoroughly enjoyed it. To Chris O'Dell for thorough review and not being scared to tell me when she thinks I'm wrong and to Steve Love, Roger Orr, Matthew Jones and Ric Parkin for review and encouragement. To Rachel Davies for the description of acceptance testing and to Caroline Hargreaves for the diagrams.

References and further reading

[Adapter] 'Adapter Design Pattern': http://en.wikipedia.org/wiki/Adapter_pattern

[Beck02] *Test Driven Development* by Kent Beck. 2002. Addison Wesley. ISBN: 978-0321146533

[Boehm81] *Software Engineering Economics* by Barry W. Boehm. 1981 Prentice Hall. ISBN: 978-0138221225

[Davies] 'When To Write Story Tests' by Rachel Davies: http://agilecoach.typepad.com/agile-coaching/2011/07/when-to-write-story-tests.html

[Facade] 'The Facade Design Pattern': http://en.wikipedia.org/wiki/Facade_pattern

[Fake] 'Fake Objects': http://en.wikipedia.org/wiki/Mock_object

[Feathers] *Working Effectively with Legacy Code* by Michael Feathers. Prentice Hall. ISBN: 978-0131177055

[FileInfo] FileInfo documentation: http://msdn.microsoft.com/en-us/library/system.io.fileinfo.aspx

[Freeman] *Growing Object Orientated Software Guided by Tests* by Steve Freeman & Nat Pryce. Addison Wesley: ISBN: 978-0321146533

[Kelly] 'Implications of the Power Law' by Allan Kelly: http://allankelly.blogspot.com/2008/03/implications-of-power-law.html

[Nagappen] 'Realizing quality improvement through test driven development: results and experiences of four industrial teams' by Nachiappan Nagappan & E. Michael Maximilien & Thirumalesh Bhat & Laurie Williams: http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf

[NUnit] 'NUnit .Net Testing Framework': http://www.nunit.org/

[Pilgrim] 'Dive Into Python': http://diveintopython.org/

[PowerLaw] 'The Power Law': http://en.wikipedia.org/wiki/Power_law

[Strategy] 'Strategy Design Pattern': http://en.wikipedia.org/wiki/Strategy_pattern

Source code: http://paulgrenyer.net/dnld/DirList-Abridged-0.0.1.zip

```
private const int ROW_LENGTH = 60;

static void Main(string[] args)
{
  var dirToList = getDirToList(args);
  var output = new StringBuilder(string.Format(
     "Directory: {0}\n", dirToList));
  var files = toMappers(dirToList);
  var formatter = new DetailsFormatter(
     ROW_LENGTH);
  output.Append(formatter.Format(files));
  Console.Write(output.ToString());
}

private static
IList<FileInfoMapper> toMappers(string path)
{
  var dirInfo = new DirectoryInfo(path);
  var files = new List<FileInfoMapper>();
  foreach (var fileInfo in dirInfo.GetFiles())
     files.Add(new FileInfoAdapter(fileInfo));
  return files;
}
```

Listing 12

Why [Insert Algorithm Here] Won't Cure Your Calculus Blues

We now know that floating point arithmetic is the best we can do. Richard Harris goes back to school ready to show how to use it.

We have travelled far and wide in the fair land of computer number representation and have seen the unmistakable scorch marks that betray the presence of the dragon of numerical error. No matter where we travel we are forced to keep our wits about us if we fear his fiery wrath.

Those who take sport in the forests of IEEE 754 floating point arithmetic have long since learnt when and how the dragon may be tamed and, if we seek to perform mathematical computation, we would be wise to take their counsel.

In this second half of this series of articles we shall take it as read that floating point arithmetic is our most effective weapon for such computation and we shall learn that, if we wish to wield it effectively, we are going to have to think!

To illustrate the development and analysis of algorithms for mathematical computation we shall continue to use the example of numerical differentiation with which we seek to approximate as accurately as possible the derivative of a function, primarily because it is the simplest non-trivial numerical calculation that I know of.

Before we do so, it will be useful to discuss exactly what we mean by differentiation and the tools that we shall exploit in the development and analysis of our algorithms.

On the differential calculus

The aim of the differential calculus is the calculation of the instantaneous rate of change of functions or, equivalently, the slopes of their graphs at any given point.

Credit for its discovery is often given to the 17th century mathematicians Newton and Leibniz, despite the fact that many of the concepts were discovered centuries before by the Greeks, the Indians and the Persians. That said, the contributions of Newton and Leibniz were not insignificant and it is their notations that are still used today: $\dot{y}$ from Newton and dy/dx from Leibniz.

The central idea of the differential calculus was that of the infinitesimals; hypothetical entities that have a smaller absolute value than any real number other than zero.

To see how infinitesimals are used to calculate the slope of a graph, first consider the slope of a straight line connecting two points on it, say (x_0,y_0) and (x_1,y_1)

$$\frac{y_1 - y_0}{x_1 - x_0}$$

To compute its slope at a point x we should ideally want to set both x_0 and x_1 equal to x but unfortunately this yields the meaningless quantity 0/0.

Instead we shall set x_0 equal to x and x_1 equal to x plus some infinitesimal quantity dx. Since this is closer to x than any real number, it should yield a result closer to the slope at x than any real number. The actual slope can therefore be recovered by discarding any infinitesimals in that result.

For example, consider the slope of the function x^2.

$$\begin{aligned}\frac{(x+dx)^2 - x^2}{(x+dx)-x} &= \frac{x^2 + 2\times x\times dx + dx^2 - x^2}{dx}\\ &= \frac{2\times x\times dx + dx^2}{dx}\\ &= 2x + dx \approx 2x\end{aligned}$$

where the wavy equals sign means approximately equal to in the sense that no real number is closer to the result.

We define a function to be continuous if an infinitesimal change in its argument yields an infinitesimal change of the same or higher order in its value. If the same can be said of its derivative, we say that the function is smooth.

For a smooth function f we therefore have $f(x+dx)=f(x)+df(x)$, where $df(x)$ is an infinitesimal of at least the same order as dx. Given this we can obtain Leibniz' notation

$$\frac{f(x+dx)-f(x)}{(x+dx)-x} = \frac{f(x)+df(x)-f(x)}{x+dx-x} = \frac{df(x)}{dx} = \frac{d}{dx}f(x)$$

That we require the function to be smooth rather than simply continuous might come as a bit of a surprise, but stems from the fact that a function does not have a well defined value at a discontinuity.

Treating d/dx as an operator as above, we recover the notation for repeated differentiation. We square it if we differentiate twice, cube it if thrice, and so forth, to obtain

$$2^{\text{nd}}\text{ derivate: } \left(\frac{d}{dx}\right)^2 f(x) = \frac{d^2 f(x)}{dx^2}$$

$$3^{\text{rd}}\text{ derivate: } \left(\frac{d}{dx}\right)^3 f(x) = \frac{d^3 f(x)}{dx^3}$$

$$n^{\text{th}}\text{ derivate: } \left(\frac{d}{dx}\right)^n f(x) = \frac{d^n f(x)}{dx^n}$$

We can recover the various identities of the differential calculus using infinitesimals. In derivation 1, for example, we prove the product rule for the derivative of the product of two functions f and g.

$$\frac{d\left(f(x)\times g(x)\right)}{dx} = f(x)\times\frac{dg(x)}{dx} + \frac{df(x)}{dx}\times g(x)$$

Given constant c, the exponential function e^x, its inverse $\ln x$ and functions f and g such that $y=f(x)$ and $z=g(y)$ some further useful identities are

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.

Proving the product rule with infinitesimals

Given smooth continuous functions f and g:

$$\begin{aligned}
\frac{d\left(f(x)\times g(x)\right)}{dx} &= \frac{f(x+dx)\times g(x+dx)-f(x)\times g(x)}{(x+dx)-x} \\
&= \frac{\left(f(x)+df(x)\right)\times\left(g(x)+dg(x)\right)-f(x)\times g(x)}{dx} \\
&= \frac{f(x)\times g(x)+f(x)\times dg(x)+df(x)\times g(x)+df(x)\times dg(x)-f(x)\times g(x)}{dx} \\
&= \frac{f(x)\times dg(x)+df(x)\times g(x)+df(x)\times dg(x)}{dx} \\
&= f(x)\times\frac{dg(x)}{dx}+\frac{df(x)}{dx}\times g(x)+\frac{df(x)\times dg(x)}{dx} \\
&\approx f(x)\times\frac{dg(x)}{dx}+\frac{df(x)}{dx}\times g(x)
\end{aligned}$$

Derivation 1

$$\frac{dc}{dx}=0 \qquad \frac{dx^n}{dx}=nx^{n-1}$$

$$\frac{de^x}{dx}=e^x \qquad \frac{d\ln x}{dx}=\frac{1}{x}$$

$$\frac{d\sin x}{dx}=\cos x \qquad \frac{d\cos x}{dx}=-\sin x$$

$$\frac{dz}{dx}=\frac{dz}{dy}\times\frac{dy}{dx} \qquad \frac{dx}{dy}=1\Big/\frac{dy}{dx}$$

Whilst infinitesimals provide a reasonably intuitive framework for the differential calculus it is not a particularly rigorous one. What exactly does it mean to say that an infinitesimal is smaller in magnitude than any real number other than zero? Given any real number, we can halve it to give us a number that is closer to zero. Halving again yields a number closer still, as does halving a third time. Repeating this process over and over again yields a sequence of numbers that shrink arbitrarily close to zero, so where exactly are the infinitesimals to be found?

This lack of rigour did not escape Newton and Leibniz' contemporaries; the philosopher George Berkeley [Berkeley34], for example, criticised the calculus for its 'fallacious way of proceeding to a certain Point on the Supposition of an Increment, and then at once shifting your Supposition to that of no Increment' and derided the infinitesimals as the 'ghosts of departed quantities'.

Despite these objections, many mathematicians were unwilling to dismiss the differential calculus due to its incredible usefulness. For example, consider the equation governing the straight line distance s travelled in a time t by a frictionless body from a standing start under a constant acceleration a

$$s=\tfrac{1}{2}at^2$$

If we take the derivative of this with respect to time, we recover the equation governing the speed of that body after the time t

$$v=\frac{ds}{dt}=at$$

That equations of motion such as these could be experimentally verified was rather strong evidence that the differential calculus was valid. It was not until some 150 years later that this was conclusively demonstrated, however.

On analysis

It was Cauchy who made the great leap forward in setting the differential calculus on a secure foundation. He did it not by giving the infinitesimals a rigorous definition but by doing away with them entirely.

His idea was to define the derivative as the limit of a sequence of ever more accurate approximations to it. Specifically, that

$$\frac{df(x)}{dx}$$

is the limit of

$$\frac{f(x+\Delta x)-f(x)}{\Delta x}$$

as Δx tends to 0

or in conventional notation

$$\frac{df(x)}{dx}=\lim_{\Delta x\to 0}\frac{f(x+\Delta x)-f(x)}{\Delta x}$$

For example, consider again the derivative of x^2

$$\begin{aligned}
\frac{(x+\Delta x)^2-x^2}{(x+\Delta x)-x} &= \frac{x^2+2\times x\times\Delta x+\Delta x^2-x^2}{\Delta x} \\
&= \frac{2\times x\times\Delta x+\Delta x^2}{\Delta x} \\
&= 2x+\Delta x \underset{\Delta x\to 0}{=} 2x
\end{aligned}$$

where the final equals sign means *equals as Δx tends to zero*.

Now this is how we defined the derivative in the first article in this series but, whilst it's certainly a step in the right direction, it's not yet quite enough. What exactly do we mean when we say Δx tends to zero? Do we repeatedly halve it? Do we start with a positive value less than 1 and square it, then cube it, then raise it to the 4th power and so on? Does it actually *matter*?

Cauchy's great achievement was to rigorously define the limit of a sequence and, in doing so, discover analysis: the mathematics of limits. We say that the limit of a function $f(x)$ as x tends to c is equal to l if and only if for any given positive ε, no matter how small, we can find a positive δ such that the absolute difference between $f(x)$ and l is always less than ε if the absolute difference between x and c is less than δ. In mathematical notation we write this as

$$\forall\varepsilon>0\ \exists\delta>0\left(0<|x-c|<\delta\Rightarrow|f(x)-l|<\varepsilon\right)$$

where the upside down A should be read as *for all*, the backwards E as *there exists*, and the arrow as *implies that*. The vertical bars represent the absolute value of the expressions bracketed by them.

With this definition we are not dependant upon the manner in which we approach a limit, only upon the size of its terms.

We now define the derivative $df(x)/dx$ with

$$\forall\varepsilon>0\ \exists\delta>0\left(0<|\Delta x|<\delta\Rightarrow\left|\frac{f(x+\Delta x)-f(x)}{\Delta x}-\frac{df(x)}{dx}\right|<\varepsilon\right)$$

If such a limit exists, we say that the function is differentiable at x.

The derivative of x^2 is $2x$ since for all x

$$|\Delta x|<\varepsilon\Rightarrow\left|\frac{(x+\Delta x)^2-x^2}{\Delta x}-2x\right|=|2x+\Delta x-2x|=|\Delta x|<\varepsilon$$

As we did with infinitesimals, we can derive the various identities of the differential calculus with this rigorous definition of a limit. Derivation 2 provides a proof of the product rule in these terms, for example.

That this proof is so much longer and more difficult than the one using infinitesimals is something that mathematicians have had to learn to live

Proving the product rule with limits

Given a differentiable function *f* we have

$$\forall \varepsilon_f > 0 \; \exists \delta > 0 \; \forall \Delta x \left(0 < |\Delta x| < \delta \Rightarrow \left| \frac{f(x+\Delta x) - f(x)}{\Delta x} - \frac{df(x)}{dx} \right| < \varepsilon_f \right)$$

$$0 < |\Delta x| < \delta \Rightarrow \left| f(x+\Delta x) - f(x) - \Delta x \times \frac{df(x)}{dx} \right| < |\Delta x| \times \varepsilon_f$$

$$0 < |\Delta x| < \delta \Rightarrow f(x) + \Delta x \times \frac{df(x)}{dx} - |\Delta x| \times \varepsilon_f < f(x+\Delta x)$$

$$< f(x) + \Delta x \times \frac{df(x)}{dx} + |\Delta x| \times \varepsilon_f$$

Now, given a second differentiable function *g* we have

$$\left(f(x) + \Delta x \times \frac{df(x)}{dx} + |\Delta x| \times \varepsilon_f \right) \times \left(g(x) + \Delta x \times \frac{dg(x)}{dx} + |\Delta x| \times \varepsilon_g \right)$$

$$= f(x) \times g(x) + \Delta x \times f(x) \times \frac{dg(x)}{dx} + \Delta x \times \frac{df(x)}{dx} \times g(x)$$

$$+ \Delta x^2 \times \frac{df(x)}{dx} \times \frac{dg(x)}{dx} \quad + |\Delta x| \times \varepsilon_f \times \left(g(x) + \Delta x \times \frac{dg(x)}{dx} + |\Delta x| \times \varepsilon_g \right)$$

$$+ |\Delta x| \times \varepsilon_g \times \left(f(x) + \Delta x \times \frac{df(x)}{dx} + |\Delta x| \times \varepsilon_f \right)$$

The bounds on the product of $f(x+\Delta x)$ and $g(x+\Delta x)$ are the minimum and maximum of the four possible products of their bounds of which, if we denote them by *L* and *U*, we can be sure that

$$\iota = |\Delta x| \times \left| \frac{df(x)}{dx} \times \frac{dg(x)}{dx} \right| + \varepsilon_f \times \left| g(x) + \Delta x \times \frac{dg(x)}{dx} + |\Delta x| \times \varepsilon_g \right|$$

$$+ \varepsilon_g \times \left| f(x) + \Delta x \times \frac{df(x)}{dx} + |\Delta x| \times \varepsilon_f \right|$$

$$L \geq f(x) \times g(x) + \Delta x \times f(x) \times \frac{dg(x)}{dx} + \Delta x \times \frac{df(x)}{dx} \times g(x) - |\Delta x| \times \iota$$

$$U \leq f(x) \times g(x) + \Delta x \times f(x) \times \frac{dg(x)}{dx} + \Delta x \times \frac{df(x)}{dx} \times g(x) + |\Delta x| \times \iota$$

and hence that

$$\left| \frac{f(x+\Delta x) \times g(x+\Delta x) - f(x) \times g(x)}{\Delta x} - \left(f(x) \times \frac{dg(x)}{dx} + \frac{df(x)}{dx} \times g(x) \right) \right| \leq \iota$$

For any given positive ε we can, for example, choose positive δ such that

$$\delta \leq \frac{\varepsilon}{3\left| \frac{df(x)}{dx} \times \frac{dg(x)}{dx} \right| + 1}$$

$$\varepsilon_f \leq \frac{\varepsilon}{3\left(|g(x)| + \delta \times \left| \frac{dg(x)}{dx} \right| + \delta \times \varepsilon_g \right) + 1}$$

$$\varepsilon_g \leq \frac{\varepsilon}{3\left(|f(x)| + \delta \times \left| \frac{df(x)}{dx} \right| + \delta \times \varepsilon_f \right) + 1}$$

and thus $\iota < \varepsilon$ whenever $|\Delta x| < \delta$, as required.

Derivation 2

with; the price of rigour is generally paid in ink. The rigour that Cauchy brought to the differential calculus was part of a great revolution in mathematical thinking that took place during the 19th century. Indeed, almost all of the techniques of modern mathematics were developed during this period.

Infinitesimals two point 0

In the latter half of the 20th century the infinitesimals enjoyed something of a renaissance. Both Robinson, with his non-standard numbers [Robinson74], and Conway, with his surreal numbers [Knuth74], developed consistent number systems in which infinitesimals could be given a rigorous definition.

Their approach was to embed something akin to Cauchy's limits into the very idea of a number. Loosely speaking, Robinson defined a non-standard number as an infinite sequence of real numbers with arithmetic operations and functions applied on an element by element basis. For example

$$x = (x_0, x_1, x_2, x_3, ...)$$
$$y = (y_0, y_1, y_2, y_3, ...)$$
$$x + y = (x_0 + y_0, x_1 + y_1, x_2 + y_2, x_3 + y_3, ...)$$

Loosely speaking, two non-standard numbers *x* and *y* are considered equal if for all but a finite set of indices i, $x_i = y_i$ or, in limit notation, that

$$\exists n \; \forall i > n \; (x_i = y_i)$$

The remaining comparison operators are similarly defined and the real numbers are the subset of the non-standard numbers whose elements are identical for all but a finite set of indices. Now consider the standard number whose elements are a strictly decreasing sequence of positive numbers. For example

$$\delta = \left(1, \tfrac{1}{2}, \tfrac{1}{4}, \tfrac{1}{8}, ...\right)$$

By the rules of non-standard arithmetic this number is greater than zero since every element in it is greater than zero. Furthermore, it is smaller than any positive real number *x* since there will be some *n* for which $1/2^n$ is smaller than *x* and hence so will be the infinite sequence of elements of δ after the nth. So here we have a genuine, bona fide infinitesimal with all of the properties of those in Newton and Leibniz' differential calculus!

If a non-standard number *z* can be represented by a real number plus an infinitesimal non-standard number, we call that real number the standard part of *z*, or st(*z*). We can thus define the derivative of a function *f* as

$$\text{st}\left(\frac{f(x+\delta) - f(x)}{\delta} \right)$$

for a standard real *x* and all infinitesimal non-standard δ, if this value exists. For example, let's consider the derivative of x^2 a third time.

$$\delta = (\delta_0, \delta_1, \delta_2, \delta_3, ...)$$
$$x + \delta = (x + \delta_0, x + \delta_1, x + \delta_2, x + \delta_3, ...)$$
$$(x+\delta)^2 = \left((x+\delta_0)^2, (x+\delta_1)^2, (x+\delta_2)^2, (x+\delta_3)^2, ... \right)$$

$$\frac{(x+\delta)^2 - x^2}{\delta}$$
$$= \frac{\left((x+\delta_0)^2 - x^2, (x+\delta_1)^2 - x^2, (x+\delta_2)^2 - x^2, (x+\delta_3)^2 - x^2, ... \right)}{(\delta_0, \delta_1, \delta_2, \delta_3, ...)}$$
$$= \frac{\left(2 \times x \times \delta_0 + \delta_0^2, 2 \times x \times \delta_1 + \delta_1^2, 2 \times x \times \delta_2 + \delta_2^2, 2 \times x \times \delta_3 + \delta_3^2, ... \right)}{(\delta_0, \delta_1, \delta_2, \delta_3, ...)}$$
$$= (2 \times x + \delta_0, 2 \times x + \delta_1, 2 \times x + \delta_2, 2 \times x + \delta_3, ...)$$

$$\text{st}\left(\frac{(x+\delta)^2 - x^2}{\delta} \right) = 2x$$

Note that at no point did we rely upon the value of δ, just that it was an infinitesimal, and hence the result stands for all infinitesimals.

The problem with this loose definition of the non-standard numbers is that the vast majority of them are comprised of oscillating or random sequences which cannot meaningfully be ordered by the less than and greater than operators. Fortunately, the formal definition addresses this deficiency by demonstrating that it is possible to define rules by which we can do so, albeit ones which we cannot ever hope to write down in full.

Very roughly speaking, these rules unambiguously equate oscillating/random sequences with convergent or divergent sequences. By magic.

Robinson proved that the non-standard numbers do not lead to any logical contradictions other than those, if any, that are consequences of the standard reals and that his infinitesimals have exactly the properties of Newton's.

We can therefore dispense with the full limit notation and simply go back to using our original infinitesimals.

Now that we know exactly how the differential calculus is defined we are nearly ready to begin analysing numerical differentiation algorithms.

Before we can start, however, there is one last piece of mathematics that we shall need.

Taylor's theorem

For those who seek to develop numerical algorithms, Taylor's theorem is perhaps the single most useful result in mathematics.

It demonstrates how a sufficiently differentiable function can be approximated within some region to within some error by a polynomial. Specifically, it shows that

$$f(x+\delta) = f(x) + \delta \times f'(x) + \tfrac{1}{2}\delta^2 \times f''(x) + \ldots + \tfrac{1}{n!}\delta^n \times f^{(n)}(x) + O(\delta^{n+1})$$

where we are using the notational convention of $f'(x)$ for the derivative of f evaluated at x, $f''(x)$ for the second derivative and $f^{(n)}(x)$ for the nth derivative. The exclamation mark is the factorial of, or the product of every integer from 1 up to and including, the value preceding it. $O(\delta^{n+1})$ is an error term of order δ^{n+1} or, in other words, that for any given f and x is equal to some constant multiple of δ^{n+1}.

By sufficiently differentiable we mean that all of the derivatives of f up to the $n+1^{\text{th}}$ must exist, and that all of the derivatives of f up to the nth must be continuous, between x and $x+\delta$, inclusive of the bounds in the latter case but not in the former.

In fact, the error term has exact bounds given by a more accurate statement of Taylor's theorem

$$f(x+\delta) = f(x) + \delta \times f'(x) + \tfrac{1}{2}\delta^2 \times f''(x) + \ldots + \tfrac{1}{n!}\delta^n \times f^{(n)}(x) + R_n$$

$$\min\left(\tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(x+\theta\delta)\right) \le R_n \le \max\left(\tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(x+\theta\delta)\right) \quad \text{for } 0 \le \theta \le 1$$

or, equivalently, that for some y between x and $x+dx$

$$R_n = \tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(y)$$

If we put no limit upon n we recover the Taylor series of a function f in the region of x

$$f(x+\delta) = f(x) + \delta \times f'(x) + \tfrac{1}{2}\delta^2 \times f''(x) + \ldots + \tfrac{1}{n!}\delta^n \times f^{(n)}(x) + \ldots = \sum_{i>=0} \tfrac{1}{i!}\delta^i f^{(i)}(x)$$

where the capital sigma stands for the sum of the expression to its right for every unique value of i that satisfies the inequality beneath it and with the factorial of 0 being 1 and the 0th derivative of a function being the function itself. Note that this identity holds if and only if the sum has a well defined limit under Cauchy's definition.

We can use Taylor series about 0, also known as Maclaurin series, to prove the surprising relationship between the value of the exponential function at 1, e, the ratio of the circumference of a circle to its diameter, π, the square root of -1, i, and -1

$$e^{i\pi} = -1$$

We do this by examining the Maclaurin series of the exponential function, the sine function and the cosine function

$$e^x = 1 + x + \tfrac{1}{2}x^2 + \tfrac{1}{6}x^3 + \ldots + \tfrac{1}{n!}x^n + \ldots$$

$$\sin x = x - \tfrac{1}{6}x^3 + \tfrac{1}{120}x^5 + \ldots + \frac{(-1)^n}{(2n+1)!}x^{2n+1} + \ldots$$

$$\cos x = 1 - \tfrac{1}{2}x^2 + \tfrac{1}{24}x^4 + \ldots + \frac{(-1)^n}{(2n)!}x^{2n} + \ldots$$

Note that all three of these series converge for *any* value of x and can be extended to the complex numbers that are the sums of real numbers and multiples of i.

So let's now consider e^{ix}

$$\begin{aligned} e^{ix} &= 1 + ix + \tfrac{1}{2}i^2x^2 + \tfrac{1}{6}i^3x^3 + \ldots + \tfrac{1}{n!}i^nx^n + \ldots \\ &= 1 + ix - \tfrac{1}{2}x^2 - \tfrac{1}{6}ix^3 + \ldots + \frac{(-1)^n}{(2n)!}x^{2n} + \frac{(-1)^n}{(2n+1)!}ix^{2n+1} + \ldots \\ &= 1 - \tfrac{1}{2}x^2 + \ldots + \frac{(-1)^n}{(2n)!}x^{2n} + \ldots + ix - \tfrac{1}{6}ix^3 + \ldots + \frac{(-1)^n}{(2n+1)!}ix^{2n+1} + \ldots \\ &= \cos x + i \sin x \end{aligned}$$

This is known as Euler's formula and yields that surprising relationship when we set $x=\pi$.

As fascinating and profound as this undoubtedly is, it is not the reason that Taylor's theorem is of such utility in numerical computing. Rather, it is that Taylor's theorem provides us with an explicit formula for approximating a function with a polynomial and bounds on the error that results from doing so.

Such polynomials are very easy to mathematically and numerically manipulate and thus can dramatically simplify many mathematical computations; they are used to very great effect in Physics, for example.

Furthermore, it gives us an explicit formula for the error in the value of a function that results from an error in its argument, such as might occur through floating point rounding for example

$$|f(x+\delta) - f(x)| \le \left| \delta \times f'(x) + \tfrac{1}{2}\delta^2 \times f''(x) + \ldots + \tfrac{1}{n!}\delta^n \times f^{(n)}(x) + \tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(y) \right|$$

for some y between x and $x+\delta$ that maximises the right hand side of the equation.

So, now we have a thorough grasp of the differential calculus and are equipped with the numerical power tool of Taylor's theorem, we are ready to scrutinise some of the numerical algorithms for approximating the derivatives of functions.

I'm afraid we shall have to wait until next time before we do so, however. ■

References and further reading

[Berkeley34] Berkeley, G., *The Analyst; Or, A Discourse Addressed to an Infidel Mathematician*, Printed for J. Tonson, 1734.

[Knuth74] Knuth, D., *Surreal Numbers; How Two Ex-Students Turned on to Pure Mathematics and Found Total Happiness*, Addison-Wesley, 1974.

[Robinson74] Robinson, A., *Non-Standard Analysis*, Elsevier, 1974.