

Designing Observers in C++11

The Observer design pattern is an often-used classic. Contemporary C++ provides ways to implement it more elegantly.

Testing Drives the Need for Flexible Configuration

How to support multiple configurations for more flexible code

Ruminations on Self-Employment and Running a Business

The pros and cons of running your own business

Order Notation in Practice

Real life complexity measurement

Non-Superfluous People: Testers

The importance of professional testers

OVERLOAD 124**December 2014**

ISSN 1354-3172

EditorFrances Buontempo
overload@accu.org**Advisors**Matthew Jones
m@badcrumble.netMikael Kilpeläinen
mikael.kilpelainen@kolumbus.fiSteve Love
steve@arventech.comChris Oldwood
gort@cix.co.ukRoger Orr
rogero@howzatt.demon.co.ukJon Wakely
accu@kayari.orgAnthony Williams
anthony.ajw@gmail.com**Advertising enquiries**

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and designPete Goodliffe
pete@goodliffe.net**Copy deadlines**

All articles intended for publication in Overload 125 should be submitted by 1st January 2015 and those for Overload 126 by 1st March 2015.

The ACCU

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

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

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

4 Designing Observers in C++11

Alan Griffiths fits a venerable design pattern into a contemporary context.

6 Non-Superfluous People: Testers

Sergey Ignatchenko takes a look at the importance of professional testers.

9 Ruminations on Self Employment and Running a Business

Bob Schmidt looks at the pros and cons of running your own business.

14 Order Notation in Practice

Roger Orr revises complexity measurement and considers it in real situations.

21 People of the .Doc

Andrew Peck breaks down the rhetoric surrounding the role of a technical author.

22 Testing Drives the Need for Flexible Configuration

Chris Oldwood demonstrates how to support multiple configurations flexibly.

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.

Finding your muse

Lack of ideas and confidence can freeze an author. Frances Buontempo considers how to warm up.

“Yet again I seem to be stuck on what to write about for an editorial, so you will have to forgive me for yet another side-track. Part of the difficulty is being put on the spot and expected to think of a topic to write about. Even if I do think of something, there are other considerations such as would anyone be interested in reading this? Do I know enough to write about this? Surely everyone, or at least most people, know more about this than I do. Being expected to perform a godlike act of creation *ex nihilo* is a tall order, even without the nagging doubts about one’s qualifications and ability. As I am sure you are aware, *Overload* requires not only an editorial, but also articles so this problem will affect all ACCU members from time to time since these articles are written, in the main, by ACCU members. We do welcome submissions from non-members too, though. Previously we considered the importance of peer review in general and code reviews and assessing articles in particular [Buontempo]. This presupposes code or articles in the first place. Often code is written because someone in charge tells you to write it – perhaps you are on the treadmill and churn it out, perhaps you manage to sneakily delete some as you go. Sometimes you have an inspired idea and work on a personal project or find a better way to do things. Sometimes you want to try a new language feature or framework. Many of us are in this gig because we enjoy life-long learning. We frequently learn by reading things other people have written, either books, blogs or tutorials and documentation for new languages or frameworks others have invented. Given the wealth of things to learn about, how can you possibly be expected to come out with something new or original? This very question has been mused on in a previous *CVu* [Oldwood]:

If none of what I do is novel is it a surprise that I’d have nothing truly interesting to write about? After all, if everyone reads the same books and blogs, follows the same people on Twitter, etc. then they’ll already know everything I do; probably more. Who exactly would be listening?

Sometimes the knowledge that we seemingly know nothing can cause writer’s block. You may recall Socrates claimed he knew nothing and that didn’t stop him having a thing or two to say. Anecdotally, he is supposed to have said, “As for me, all I know is that I know nothing” [Plato] though this may be somewhat out of context and there are similar quotes pulled from other places. See this Wikipedia entry [Wiki] for example. Though mention is made of potential fear and upset when sharing, Oldwood’s article makes the important point that you do not need to write about something new in order to write – much writing offers a new viewpoint on an old topic. This is true of *Overload*, computing in general and in many other disciplines too. Indeed, I have recently acquired a DVD of *Rosencrantz and Guildenstern are dead* which weaves a tale around two minor characters in

Shakespeare’s *Hamlet*. Many other works of fiction are based around minor characters in a relatively well known story or historical account. The constraint gives a clear context and timeline of events and to an extent provides ready-made characters, though possibly only in outline comic form. It is ok to write around a well-known topic or existing body of knowledge. Many peer reviewed journals have a ‘state of the art’ review article from time to time which summarises the current players in a given area and digs into the advantages and disadvantages of each. For example, our penultimate issue had two articles considering various stands on test driven development.

This begs the question how do other people invent new features or frameworks or topics it’s ok to write about? Where do the new ideas come from? We could ask the same question of articles in this very journal. Sometimes they are inspired by problems people have faced and are a write up of emergent solutions or patterns if you will. Sometimes ‘hot-topics’ show up on online discussion forums or Twitter or similar. In both cases, the ideas are probably not completely new. This does not matter a jot – the important thing is they are interesting. Sometimes you may even find as you try to write up something you think you know well, for example a discussion on accu-general, you will inevitably find you need to investigate areas you hadn’t considered before and so learn even more. Perhaps it’s not possible to create anything new anyway – perhaps we are constantly rehashing and rediscovering old ideas. How often do you attend a talk or conference session to be presented with a quote from the 1960s? More generally, philosophical debates about whether anything, ranging from inventions to mathematics, are created or discovered have raged for a long time.

Pulling back from a grand digression into an abstract philosophical subject, let us return to our key point. How does one find inspiration for a topic to write about? Terry Prachett [Prachett] mentions inspiration particles

Particles of raw inspiration sleet through the universe all the time. Every once in a while one of them hits a receptive mind, which then invents DNA or the flute sonata form or a way of making light bulbs wear out in half the time. But most of them miss. Most people go through their lives without being hit by even one.

Newton’s gravity may have come from being directly hit by an apple rather than an inspiration particle. Evidence may suggest this was an outright lie, possibly never told by Newton himself [Cracked], and is not the only ‘Newtonian-ism’ which gets misunderstood, though it may give us something to think about. Stepping away from your books and desk for a while to go outside and get a breath of fresh air is the only way to get hit on the head by an apple and might make you receptive to inspiration particles. The second well-known quote from Newton is almost certainly ‘If I have seen further it is by standing on the shoulders of giants’. He was writing to Robert Hooke at the time, and it has been suggested this was



Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 12 years professionally, and learnt to program by reading the manual for her Dad’s BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

really a disparaging remark about Hooke's physique [Crease]. Nonetheless, its frequent quoting suggests drawing on the ideas and previous work of others is acceptable.

What actually causes the spark of an idea? I believe it is really important to find ways to free up your thinking in order to get the creative juices flowing, to coin a phrase. I have recently finished reading *What-If?* [Munroe]. The sheer ridiculousness of the questions was a delight, and left me for a while afterwards musing on various unconventional angles on almost everything I was faced with. Sadly the different perspective has now waned, but this reminded me how important it is to stop being so serious from time to time and allow oneself to ask seemingly crazy questions. The occasional book can act as a direct hit by an inspiration particle. There are some other ways to attempt to move into a creative space.

If you are learning something new, it is sensible to keep notes as you go. If you read back through them and asked focused questions – Does this make sense? What haven't I covered? Are there edge cases? – you can sharpen up your notes and form a 'How-to' article. If you look back at anything you have previously written you may have further ideas. Unfortunately, this requires you to have written something in the first place. We touched upon constraint to bring about creativity – either in the form of fitting around an existing story, or joining in a current debate on a hot topic, and perhaps writing a summary article. In contrast, it can be useful to drop constraints. Even if you have a specific task to do, it can be helpful to step back for a moment and allow a free-form doodle as a team on a white board during a design session or a time to 'brainstorm'. It is important to emphasise when doing this that no idea is too stupid otherwise most people will avoid speaking out. The aim is to get ideas flowing. If no one will speak out in your group, there are ways to get the ball rolling. For example, the 'McDonalds trick' – if a group of people are trying to decide where to go for a meal and no one will make the first suggestion, the theory goes saying "McDonalds" will encourage everyone to say "No, even [somewhere else] is better than that". Saying something, no matter how unsophisticated, or downright poor, can get the ball rolling. Many great ideas can come out of a stream of ridiculous ideas. Sometimes just brainstorming by yourself works – write down all the things you are thinking/reading about and see what emerges. Allow yourself some unstructured play time. Recently there have been increasing reports of children having too much structured leisure time – formal music or sports lessons, extra academic coaching and so on. Though these can be very useful, many are keen to point out the importance of 'Free-play' for normal development. "Jean Piaget conducted extensive research into play and concluded that play was a vital component to children's normal intellectual and social development." [Journal of Play] This article also touches on the importance of art and music for creativity too. If you do not feed your inner muse, she will die. Don't forget, adults need play time too. "All work and no play makes Jack a dull boy."

So, what makes you creative? You can simply summarise what you have been learning about – this will be your unique perspective on the topic and will at least be notes to your future self. You can meet-up with other people and try to throw some ideas around. You need to give your muse space and time. Sometimes just waiting doesn't make the proverbial apple fall down though. A brainstorming session may still not be quite enough to get things moving. The so-called Disney brainstorming method may help – in order to come out with and refine ideas, aim for three different roles; a dreamer, a realist, and a spoiler [Disney]. The dreamer is allowed, indeed, must come out with fantastical ideas. They can break the laws of physics, be completely impractical, apparently pointless. The next role grounds the ideas a bit – they are not dismissed but refined in order to be possible. The final stage rejects anything it's confronted with, shooting holes in it. The first two roles can defend/refine further and so on. In theory some ideas will survive the process. You can do this with other people or by yourself.

Though it is often used as a training exercise for entrepreneurship it can work in various other realms. Constraining yourself to only being a dreamer – coming out with new ideas – can set off a stream of creativity.

A final example of constraint leading to creativity is warfare. War is sometimes described as the mother of invention. There are many examples of manufacturing companies creating new products during various wars, such as vegetarian sausages [Sausages]. To some extent the difficulties cause workarounds to be discovered or invented; while to another extent governments often find sources of funding to solve specific problems. Sometimes disasters, rather than wars, also lead to inventions. John Harrison invented a timekeeping piece and thereby claimed the £20,000 reward which had been in place for almost 60 years since a disaster at sea killed over 2000 people, leading to calls for better means of navigation [Observatory]. Most people face the equivalent of disasters or wars at work, albeit on a smaller scale, and as a colleague keeps pointing out to me "No one died." The response, "This time" is possibly wearing thin though. Nonetheless, these problems can inspire new approaches to avoid repetitions of mistakes, which are always worth recording for posterity.

What have we discovered? Finding a topic to write about can be difficult, but just keeping notes on what you are doing day to day can be a fertile source of ideas. Deliberately meeting with others to brainstorm can lead to new ideas, but might need tempering with various techniques to get the ideas flowing. Walking away for a bit can help, allowing a chance to be hit by an inspiration particle, or apple. If you are brave and try writing you will learn to weather any storm of rotten fruit and there may be less than you expect. It's ok to do something that's been done before – you can bring a new twist and in turn set off a new train of thought for someone else. In order to try to find completely new ideas you need something to fire up your imagination. Whatever you decide to write about, hopefully the ACCU provides a supportive and shepherding environment and never stifles your creativity. Please feel free to submit articles, no matter how left-field you think they are. We do not promise to accept them but you never know 'til you try.



References

- [Buontempo] 'Peer Reviewed' Frances Buontempo *Overload* 123, October 2014
- [Cracked] http://www.cracked.com/article_16101_the-5-most-ridiculous-lies-you-were-taught-in-history-class.html
- [Crease] *The Great Equations: The hunt for cosmic beauty in numbers* Robert Crease, 2009
- [Disney] Various (contrary) links, but for example: <http://www.idea-sandbox.com/blog/disney-brainstorming-method-dreamer-realist-and-spoiler/>
- [Journal of Play] <http://www.journalofplay.org/sites/www.journalofplay.org/files/pdf-articles/1-3-article-childrens-pastimes-play-in-sixteen-nations.pdf>
- [Munroe] *What If? Serious scientific answers to absurd hypothetical questions* Randall Munroe, 2014
- [Observatory] <http://www.rmg.co.uk/about/history/royal-observatory>
- [Oldwood] Chris Oldwood. 'Being original' *CVu* 26(3):9, July 2014
- [Plato] *The Republic, Book 1*
- [Prachett] *Wyrd Sisters*
- [Sausages] <http://www.bbc.co.uk/news/magazine-26935867>
- [Wiki] http://en.wikipedia.org/wiki/I_know_that_I_know_nothing

Designing Observers in C++11

The observer pattern is over two decades old. Alan Griffiths fits a venerable design pattern into a contemporary context.

Two decades ago the ‘Gang of Four’ popularised a pattern form and described twenty one patterns (and one anti-pattern). One of these is the OBSERVER PATTERN and the subject of this article.

I’ve been working on an open source project for the past couple of years and like many projects we found use for OBSERVERS. In order for you to appreciate the problems we encountered I first need to explain a little about the project. (I will be brief.)

The project is a library that allows the code that uses it to handle graphics and input devices in a manner that is portable across the Linux device drivers found on desktop and android devices. One area in which we used OBSERVER was to monitor changes to the ‘surfaces’ representing things that appear on the screen(s). These need to be monitored by other components such as the ones that composite these surfaces onto the screens and the one that routes input.

There are a number of threads running in the application as this makes it easy to partition work between applications updating their surfaces, input events and interacting with display devices such as monitors. As I will describe, this leads to a need to address some synchronization issues.

It is hopefully evident that the ‘consumer’ of observations (e.g. one representing a monitor) generally outlive the surfaces being observed. Mostly the consumer is part of the application infrastructure and not part of the dynamic state of applications being launched, opening and closing windows and exiting.

For this case it is simplest to create an ‘observer’ object (that calls notification methods on the consumer) and pass its ownership to the ‘subject’. The consumer can then forget about everything except handling the notifications.

To summarise the differences from the classical OBSERVER PATTERN context :

1. We’ve split the Observer role into Consumer and Observer
2. We’re not using garbage collection and so need to explicitly address the lifetime of the objects
3. We have multiple active threads

The subject maintains a collection of listeners and the naïve approach to synchronization is for the collection to be locked when being updated and when sending notifications. (In fact that is how we first implemented it.)

This works well until we hit one of two cases:

1. The consumer takes some action that generates a new notification
2. The consumer is destroyed (e.g. a monitor is unplugged) and needs to prevent further notifications to a dead object.

In case 1, if we have hold exclusive lock during notifications then we’ll get a deadlock.

Alan Griffiths has been developing software through many fashions in development processes, technologies and programming languages. During that time, he’s delivered working software and development processes to a range of organizations, written for a number of magazines, spoken at several conferences, and made many friends. He can be contacted at alan@octopull.co.uk.

In case 2, we expect the consumer to remove the observer from the subject’s collection after which notifications cease. And to ensure ‘after’ needs an ordering on removal and notifications we need something akin to the lock that causes deadlock in case 1.

One failed solution that we tried is to copy the collection before propagating a notification and release the lock before calling each of the observers. That doesn’t work with the above solution to case 2: after releasing the lock nothing prevents listeners being ‘removed’ on another thread before being notified through the copy.

Another solution we rejected was for the subject’s collection to be formed of `weak_ptr<>s` and the consumer to manage a collection of `shared_ptr<>s` to the observers it creates. Having an additional collection to manage in the consumer didn’t lead to any simplification. This might be different in other contexts but many of our consumers only need to register an observer and handle a few events without tracking either the subject or the observer.

Another (working) solution we tried was to hold a recursive lock during notifications and updates to the collection. That allowed other notifications to take place on the same thread and changes to the collection. Because the collection could change we took a copy of the collection and traversed that (to avoid iterators invalidating), but before invoking them also would check that objects exist in the ‘true’ collection. The disadvantage of this approach is that copying the collection is a lot of work to send each notification for collections that very rarely changed (your application might be updating the ‘surface’ at 60fps but adding and removing monitors might happen once a month).

You might think that we could avoid this trouble by sharing ownership of the consumer with the observer (so that the consumer could not ‘die’ while the observer exists). After all, that is exactly what would happen ‘automatically’ in a garbage collected language. The trouble is that it becomes extremely difficult to ensure that the consumer dies when one needs it to.

Eventually we came up with an ‘observer collection’ implementation that works for our specific circumstances. If you have massively parallel code or rapidly changing collections this is probably not going to work for you.

The code makes use of a `RecursiveReadWriteMutex` and associated `RecursiveReadLock` and `RecursiveWriteLock`. These work pretty much as one might expect – a read lock can be acquired provided there are no write locks and a write lock can be acquired unless another thread has a lock. (This isn’t a component of C++11, so we rolled our own – one day we will have shared and exclusive locks available to us in the standard library.)

Information about the observers is held in a singly linked list with atomic forward pointers that allow lock free traversal and expansion (Listing 1). (We don’t need contraction for our use case – we might end up with a few ‘free’ items in the list but not enough to be of concern.)

There are three member functions to go along with this. The easy one is `for_each()` which is used to traverse the collection and send notifications. Each node is read locked in turn, and the supplied functor

```

struct ListItem
{
    ListItem() {}
    RecursiveReadWriteMutex mutex;
    shared_ptr<Observer> observer;
    atomic<ListItem*> next{nullptr};
    ~ListItem() { delete next.load(); }
};

```

Listing 1

invoked. Note that we need to lock the node until we complete the notification to prevent a race with another thread removing the observer and deleting the consumer. No lock is needed for the traversal itself as we are assuming that no node is ever removed from the list. (Listing 2)

The second function is to add an item. This searches for a free node. If it finds one it tries to upgrade to a write lock and, if it is still free, uses it to store the supplied observer. Otherwise a new node is added to the end of the list using the C++11 atomic ‘compare exchange’. This code also assumes that the list never shrinks as `current_item` has to remain valid. (Listing 3) One ‘gotcha’ here (spotted by the *Overload* review team) is that `compare_exchange_weak()` can ‘fail spuriously’¹, so it is necessary to test that expected has changed before assigning it to `current_item`.

The final member function removes an observer by searching the list. The logic is very similar to that in `add()`. (Listing 4)

It is a deceptively simple solution to a problem that for a while seemed intractable. I hope you enjoy it. ■

References

The project is called ‘Mir’ and can be found (including the full version of the code) at <http://unity.ubuntu.com/mir/>

Acknowledgements

The Mir team (<https://launchpad.net/~mir-team/+members>) especially Alberto Aguirre and Robert Carr for encountering this design context and working through a series of proposed resolutions.

The *Overload* team for seeing the code and text presented here with a fresh eye and spotting a few problems with correctness and clarity that had been overlooked.

```

template<class Observer>
void BasicObservers<Observer>::add(
    shared_ptr<Observer> const& observer)
{
    ListItem* current_item = &head;

    do
    {
        // Note: we release the read lock to avoid two
        // threads calling add at the same time
        // mutually blocking the other's upgrade to
        // write lock.
        {
            RecursiveReadLock lock{current_item->mutex};
            if (current_item->observer) continue;
        }
        RecursiveWriteLock lock{current_item->mutex};

        if (!current_item->observer)
        {
            current_item->observer = observer;
            return;
        }
    } while (current_item->next &&
             (current_item = current_item->next));
    // No empty Items so append a new one
    auto new_item = new ListItem;
    new_item->observer = observer;

    for (ListItem* expected{nullptr};
         !current_item->next.compare_exchange_weak(
             expected, new_item);
         expected = nullptr)
    {
        if (expected)
            current_item = expected;
    }
}

```

Listing 3

```

template<class Observer>
void BasicObservers<Observer>::for_each(
    function<void(shared_ptr<Observer>
                 const& observer)> const& f)
{
    ListItem* current_item = &head;
    while (current_item)
    {
        RecursiveReadLock lock{current_item->mutex};
        // We need to take a copy in case we recursively
        // remove during call
        if (auto const copy_of_observer =
            current_item->observer)
            f(copy_of_observer);
        current_item = current_item->next;
    }
}

```

Listing 2

```

template<class Observer>
void BasicObservers<Observer>::remove(
    shared_ptr<Observer> const& observer)
{
    ListItem* current_item = &head;

    do
    {
        {
            RecursiveReadLock lock{current_item->mutex};
            if (current_item->observer != observer)
                continue;
        }
        RecursiveWriteLock lock{current_item->mutex};

        if (current_item->observer == observer)
        {
            current_item->observer.reset();
            return;
        }
    } while ((current_item = current_item->next));
}

```

Listing 4

1. http://www.cplusplus.com/reference/atomic/atomic/compare_exchange_weak/: “Unlike `compare_exchange_strong`, this *weak version* is allowed to *fail spuriously* by returning `false` even when *expected* indeed compares equal to the *contained object*. This may be acceptable behavior for certain looping algorithms, and may lead to significantly better performance on some platforms. On these *spurious failures*, the function returns `false` while not modifying *expected*.”

Non-Superfluous People: Testers

Software development needs a team of supporting players. Sergey Ignatchenko takes a look at the role of professional testers.

The superfluous man (Russian: ЛИШНИЙ ЧЕЛОВЕК, lishniy chelovek) is an 1840s and 1850s Russian literary concept derived from the Byronic hero. It refers to an individual, perhaps talented and capable, who does not fit into social norms.
~ Wikipedia

Disclaimer: as usual, opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translator or *Overload* editors; please also keep in mind that translation difficulties from Lapine (like those described in [LoganBerry04]) might have prevented 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.

This article intends to open a mini-series on the people who're often seen as 'superfluous' either by management or by developers (and often by both); this includes, but is not limited to, such people as testers, UX (User eXperience) specialists, and BA (Business Analysts). However, in practice, they are very useful – that is, if you can find a good person for the job (which admittedly can be difficult). This article tries to discuss the role of testers in the development flow.

Hey, we already have test-driven development, testers are so XX century!

Assumption is a mother of all screw-ups
~ Wethern's Law of Suspended Judgment

When raising a question about testers in modern developer circles, frequently the first reaction is surprise. As a next step, if developers are kind enough not to take me by my ears and throw me out immediately, they start to explain all the benefits of automated unit testing and/or test-driven development. I can assure all the readers that I know all the benefits of these concepts and sometimes I even use them myself.

Still, even if you have test-driven development (which is often a good thing, there is no argument about it), it still doesn't mean you've got a silver bullet which guarantees that you have zero bugs. Moreover, there are at least four really important reasons why we cannot expect that such tests can possibly cover the whole space of potential bugs.

First of all, we need to mention that usually, the most hard-to-find bugs are integration bugs. How often do you face the following situation: "Both modules are working fine separately, but together they fail; to make things worse, they fail only in 20% of common use cases."? I'm willing to bet all the carrots in the world that for any more-than-single-person-development such a situation is not just 'common', it is a thing which takes at least 50%

of the system debugging time. In the theory of test-driven development, it should be a responsibility of whoever combined these two modules to write test cases for all possible interactions, but it never happens if modules are not-too-trivial. Some tests (often as little as one, to have something to run) are written, sure, but all of them? No way. Why this is the case can be somewhat explained by the next two reasons.

The second reason is related to psychology. The mindset of a developer is usually 'to create something', not 'to break something'. It is especially true when you need to look for ways to break your own creation. 'To create a test to demonstrate that your creation is working' is one thing (and this is what happens in test-driven development), but 'to look for creative ways to break your own creation' is a *very* different story. Sometimes this second effect may be mitigated by having developer A write code and developer B write test cases for this code (and vice versa), but such a policy essentially means that each developer works as a half-time tester, that's it.

The third reason is related to the observation that when higher-level more-complicated modules are being integrated, the developer who's doing it simply does not have enough information to find all the relevant test cases. In practice, module documentation rarely goes beyond doxygen, and never ever describes all the relevant details (not that I'm saying it can be possibly done at all). It means that unless the developer who performs the integration wrote both modules being integrated, she doesn't have sufficient information to write an exhaustive set of test cases (not to mention that doing it may be prohibitively expensive). In theory, one shouldn't rely on anything which has not been tested in module unit tests, but even this is not realistic, and doesn't guarantee against integration issues.

The fourth reason is related to the same developer at the same time having too much information about the program. Often the developer 'knows' how it should work, and assumes (!) that it does. Which inevitably leads to test cases being omitted. On the other hand, the developer often doesn't know all the details about how the code is supposed to be used (and especially those scenarios which are not supposed to happen, but will arise in real life anyway).

In three out of these four cases listed above (#1,#2, and #4) a tester performing high-level testing has a clear advantage over a developer doing unit testing (there are other cases where unit tests have advantage over high-level testing, but this is beyond the scope now as we're not arguing about unit testing being unnecessary). Based on the reasoning above, and on many years of experience, IMNSHO testers *are* necessary in most projects developed by more than 1 or 2 people.

With this in mind, we've already answered the question, "Do we need testers if we have test-driven development?" and now can proceed to the next ones: "What are we trying to achieve?", "How to convince management that you need testers", and "How to organize a useful testing team?" (even if 'team' is as small as 1 tester).

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

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

as a zero-defect product is a non-existing beast, it is better to admit it and make it clear that team should actively look for bugs

Goals of the testing team

Each team should have well-defined goals. The goal of a testing team is to improve product quality (some may argue that it is to 'assure quality', but as a zero-defect product is a non-existing beast, it is better to admit it and make it clear that team should actively look for bugs, rather than passively say, "Everything is ok").

Improving product quality is a very broad task, and includes at least such things as (with more details on some of these tasks provided below, under 'How to organize the testing team'):

- automated regression testing; sure, unit tests should be run by developers themselves, but integration testing should also include automated regression testing
- actively looking for ways to break the program. For example, one thing which a good tester usually does is to look for things like "Hey, what if a user presses this button now?"
- if applicable: monitoring end-user feedback and producing reproducible bug reports out of it
- if there is no UX team: reporting 'usability defects', also (if applicable) from end-user feedback

If you're still sure that you (as developer) want to do all these things yourself – I give up. For the rest of you, let's proceed to the next question: "How to convince management that you need testers"?

Testers from the management perspective

Admittedly, convincing management about testers can be really tough. After all, from an accounting perspective, testers are often interpreted as an expense, without producing anything tangible (developers are at least producing code, which can be seen as an asset, but testers, even good ones, don't produce any assets). The same logic, however, can be extended to say that as code is an asset, a bug in a code (which reduces code quality and user experience) is a liability (in extreme cases – it can be literally a liability in a sense too [Levy89] [Techdirt11]). Therefore, while testers indeed do not produce assets, they do remove liabilities, which has a positive impact on the bottom line of the company (provided that the testers are good, but this stands for all kinds of employees).

Throwing in such an all-important thing as improved end-user satisfaction, this should be enough to convince all but the most-stubborn managers to admit that testers are useful; the problem of convincing them that testers are needed 'right now' (and not "yes, sure, maybe, some time later") is left as an exercise for the reader.

How NOT to organize a testing team

At some point in my career, I was working for a really huge company (on the scale of "I'm not sure if there is anything larger out there"). There, the testing department for one of the projects consisted of several dozen people, who were given instructions like "press such and such button, you should be shown such and such screen, if it is not – report, if it is – go to the next step". This is certainly one way not to organize your testing team.

How to organize a testing team

As usual, organizing the development process is more art than science, and this applies to testing teams in spades. However, there are some common observations which may help on the way:

- never ever position testers as inferior to developers. First, they are not inferior (and if they are, you've done a poor job organizing the testing team). Second, it will hurt the process badly. Overall, it is usually better to position testers higher than developers (in a sense that developers should fix the bugs found by the testers instead of arguing that it is not a bug, but a feature); at the very least:
 - opened bugs should be processed according to tester-specified severity and priority
 - the process of closing bugs as WONTFIX shouldn't be too easy for developers, and should involve discussions and/or management approval
- never ever think of testers as inferior developers. Think of them as of developers with a different mindset. As discussed above, there are things which developers cannot do for objective reasons.
- automated testing tools is are an absolute must. Situations when testers are just pressing buttons according to instructions must be avoided at all costs. Whenever a bug is found and fixed, a test case must be included into a standard regression test set.
- whenever possible, consider writing automated self-testing tools at non-UI level; for example, if your application is a network one, one way to test is not by pressing buttons in the UI client, but by making network requests instead. This may allow testing the system a bit differently (which is always good) and under much heavier stress (which is even better). Such automated self-testing tools are normally written by developers and used by testers.
- it should be clearly stated that normally it is a tester's responsibility to make a bug reproducible (exceptions for intermittent bugs are possible, but they should stay as exceptions). Having irreproducible bugs in bug tracking is bad for several reasons, including the following two: first, it forces developer to do tester's job (with most of the argument for having separate testing team applicable); second, it greatly reduces enthusiasm for developers to fix the problem.
- if a product is already released, at least some portion of testers' time should be dedicated to go through user complaints, try to reproduce them and open bugs if bugs are confirmed. For a product with an active user forum, this can work wonders with regards to product quality. This approach risks starting to deal with singular user complaints (i.e. those which represent a problem only for one single user), but provided that user forum is active enough, simple filtering of "at least N complaints" usually does the trick (see also below about "we've already got two(!) complaints" approach – it did work in practice).

- one special area is ‘Usability defects’. Strictly speaking, it is better to delegate dealing with such defects to UX specialists (which represent another category of ‘Non-Superfluous People’ and will be hopefully be discussed in a separate article). However, if you do not have separate UX team (which you should, but probably don’t) – it should be a responsibility of testers to complain about ‘Usability defects’ (in other words, about ‘inconvenient/confusing UI’). At the very least, end-user complaints about ‘Usability defects’ should be taken really seriously and fixed whenever there are enough users complaining. After all, it is the end-user who’s ultimately paying for the development [NoBugs11]. In one company with millions of users, I’ve seen a policy of “Hey, we already have two (!) complaints from end-users – we should do something about it in the next release”. Believe it or not, such a policy did result in the company making the best software in the field (and making money out of it too).
- having a testing team does not imply in any way that unit tests and/or test-driven development are not necessary. Ideally, testing teams should work only with stuff which has already been unit-tested and concentrate on (a) automated regression testing; and (b) not-so-obvious ways to break the program.

Hey, where to find good testers?

Unfortunately, finding good testers is a big problem. However, if you stop thinking about testers as inferior people, and search for them not as an afterthought, but in the same way as you’re looking for developers, it usually becomes possible. Yes, finding a good tester is not easy; however, finding a good developer is also not easy, but every successful team does it (otherwise it won’t be successful). So, try and find at least one good tester to complement your good developers – the improved quality of your product will almost certainly raise user satisfaction (whether it will raise company profits and developer salaries is a different story, but things such as marketing are beyond the scope of both this article and *Overload* in general).

Testing metrics

When you do have a testing team, the question arises: how to measure its performance? In practice, I’ve seen two approaches (which can be combined). The first one is to measure how many bugs (and of which severity) were found. This approach has the problem that you’d need somebody (besides testers and developers) to judge severity of bugs; and if this is not done, the metric quickly deteriorates into something as meaningless as the ‘number of lines of code’ metric for developers. The second approach only works if you have strong end-user feedback. In such a case, you can estimate both end-user satisfaction, and the percentage of bugs which reach the end-user. While formalizing it further is also



not easy (and should be done on case-by-case basis), this approach provides a very reliable way to see how useful the testers’ job is for the product (and eventually for the bottom line of the company).

Summary

- If you think you don’t need a testing team – think again
- Test-driven development doesn’t mean you don’t need a testing team
- The testing team can be as small as one person
- You do not just need ‘any testing team’, you need a good one
- Organization of the testing team and its interaction with developers is all-important
- Finding good testers is as much of a challenge as finding good developers
- All these things are not easy, but do-able
- If done properly, it is worth the trouble and expense ■

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

References

[Levy89] Levy, L. B., & Bell, S. Y. (1989). *Software product liability: understanding and minimizing the risks*. High Tech. LJ, 5, 1.

[Loganberry04] David ‘Loganberry’, Frithaes! – an Introduction to Colloquial Lapine!, <http://bitsnobstones.watershipdown.org/lapine/overview.html>

[NoBugs11] Sergey Ignatchenko. The Guy We’re All Working For. *Overload* 103.

[Techdirt11] UK Court Says Software Company Can Be Liable For Buggy Software. <https://www.techdirt.com/blog/innovation/articles/20100513/0053499408.shtml>

Ruminations on Self Employment and Running a Business

Being self-employed has pros and cons. Bob Schmidt reviews what he has learnt about running your own business.

I've been self-employed as a contractor and consultant for 20 years. Being self-employed has both its positive and negative aspects. Being your own boss and being able to set your own hours is a positive; worrying about where the next contract is going to come from is a negative. Being able to pick assignments that interest you is a positive; having to take grunt work because nothing else is available at the moment is a big negative. In spite of the negatives I suspect I would have a hard time going back to being someone else's employee.

What follows are some of the things I've learned about being self-employed, in no particular order of importance, and at the end I'll mention one big thing I haven't yet figured out. Fair warning: I am neither an attorney (solicitor) nor an accountant, and nothing here should be construed to be legal or accounting advice. It is important that you consult both of these types of professionals when setting up your own business.

Give a little away

I am a strong believer in 'giving a little away'. In my experience, every time I have 'given a little away', I have gotten back more in the long run.

A couple of examples should help illustrate the concept.

I had a customer for whom I provided all software support on one of their plant computer systems. (This system was just replaced, after 32 years of service.) Their systems engineering department has an area called the bullpen, which has white boards on two walls, and a large table with seating for the whole group. It was the practice at the time to gather together to eat lunch in the bullpen every day. Some work got discussed, but mostly it was a social hour.

During one trip to the site there was quite a bit of information on the larger white board regarding a problem they were having with another of the plant computer systems. From the drawings on the board it appeared the problem had to do with serial communications, but if so there were several problems with their understanding of how asynchronous serial communications work. Over the next several days, during lunch, I started asking questions and discussing the problem with the two engineers responsible for the system. By the end of that two week trip I had 'given away' my lunch hour almost every day.

The result of my 'giving a little away' over the course of several lunch hours was four months of additional, paid work, on a system I had never worked on before. In the winter of 2012 we installed an upgrade to the software which fixed several major and a lot of minor software problems, which improved the mean-time-to-failure of the system by an order of magnitude.

Another example: In the early- to mid-1980s I worked for a company that built industrial control systems based around ModComp mini-computers. These systems had a proprietary, home-grown database sub-system, and historical transaction data that was stored using that database. The systems had a report generator that allowed for selecting and sorting records for reports, but it was slow and not very extendable. (Full disclosure: I wrote that report generator in 1982, and it was still in use at one plant up until November 2013, when the last system of its type was replaced.)

During a trip to one of the sites in the early 2000s, one of the system owners expressed an interest in the possibility of moving the transaction data to a Microsoft Access-based database, so that he could manipulate the data more easily, much faster, and at his desk. I decided to see if I could provide a product that would export the data from the Modcomps and import it to Access.

I don't have a Modcomp sitting around my office (that would be so cool). They require a lot of space – imagine a small closet (wardrobe) – and have power requirements that would result in my late 1960s house burning down when the aluminum wiring decided to sizzle. My wife puts up with a lot when it comes to the space my business requires, but being homeless was just not part of the deal.

Fortunately, I was able to strike a deal with the systems engineering supervisor at one of the other plants that still had a Modcomp system. He allowed me to use the development computer in their lab to develop the software that migrated the historical transaction data from the proprietary database to a Microsoft SQL server database. In return, I provided support for their systems when needed.

The little bit I gave away in the form of infrequent support for their systems gave me access to the resources I needed to develop a product. I was eventually able to sell three copies of the resulting software, one of which was to the plant which allowed me to use their system to develop it.

Don't promise to do what you don't know you can do

The large company for which I worked built in-plant and SCADA industrial control computer systems for public utilities. The systems were defined by a specification document developed by an architectural engineering firm. At that time those documents could easily be four or more inches thick, full of finely wrought details.

The contracts for these systems were awarded at the end of a bidding process. A company's bid was supposed to include a list of exceptions to the specification, to let the customer know what they weren't going to get. My company had a standard system that we would customize to meet the specification as closely as possible, but in general it wasn't possible to meet every jot and tittle of a specification.

I happened to overhear an assistant ask a bid manager for the exception list for a bid they were developing. He answered that there were no exceptions. The bid manager's response set in motion several long-term issues.

First, once the contract was awarded, the bid manager no longer had responsibility for the project. He was making promises he didn't have to keep – it would be the project team's responsibility to execute the project.

Bob Schmidt is president of Sandia Control Systems, Inc. in Albuquerque, New Mexico. In the software business for 33 years, he specializes in software for the process control and access control industries, and dabbles in the hardware side of the business whenever he has the chance. He can be contacted at bob@sandiacontrolsystems.com.

If I don't know that I can do something, I go off and try to do it. On my time. At my expense.

Second, it set the project up for schedule and financial failure, since the time and cost of implementing all of the features that should have been expected were never accounted for in the bid. (We used to joke that we lost money on every job but made up for it in volume.) Third, it reinforced the idea of 'win the contract now and fight it out in court later', which was all too prevalent in the company (and the industry in general) at that time.

I swore I would never run my business this way. As a small business I can't afford the money it would take to defend myself if something like this got to court, nor the hit my reputation would take.

A quick word about what I call 'science projects'. It is OK to take on work you don't know you can do, as long as you are up front and honest about it. In the first example in 'give a little away' above, I didn't know if I was going to be able to fix that customer's problems. I was going to be working on software I had never seen before, running on hardware with which I was unfamiliar, and could offer no guarantees of success. I made sure to emphasize these issues before taking on the work, so the customer knew what the possible outcomes of the work were.

A working prototype can sell itself

If I don't know that I can do something, I go off and try to do it. On my time. At my expense. That program I described earlier that exported data from a proprietary database to an SQL database is just such an example. When I started that project I didn't know Visual Studio .NET (which I used for the PC-side software and the GUI); I didn't know SQL server (the PC-side data store); I didn't know if it was going to be practical to move the data over a 19.2K baud serial communications channel – the fastest I had available. (One large 9-track magnetic tape took 8 to 12 hours to convert and transmit.) But once I had it working it was an easy sell.

I had another customer who complained that a critical hardware subsystem was obsolete, breaking down frequently, and impossible to fix. They had asked another company for help. That vendor had sent them a sample of their replacement product, but my customer couldn't get it to work. The vendor wasn't very helpful, and my customer was getting frustrated.

I had just purchased a development kit for (what was at that time) a new microprocessor called the Rabbit 2000. I sketched up some circuit diagrams, and proceeded to develop a very ugly, hand soldered and wire-wrapped prototype for a replacement product that would work for my customer. I wrote the firmware, debugged the hardware, and when I was done called him up and said I had a working prototype, and would he have some time for me to visit the site and try it out? He did, I did, the prototype worked, and I eventually sold 150 production-quality units – the largest single project I had up to that point.

The customer is not always right

It is trendy to say that in business the customer is always right. I've found in this business the customer is often wrong. The tricky part is knowing what to do about it, and when.

Years ago, when I was working for the large corporation, I paid a visit to a customer site in order to perform an operating system upgrade. This was

a non-trivial activity, which required a simultaneous hardware upgrade, since major operating system versions required specific hardware revision levels to function properly.

The customer had procured the operating system upgrade and upgraded boards from the computer vendor. I was brought in to load the operating system and rebuild the application software (which was required because the run-time libraries were upgraded with the OS).

The customer wanted to make a major, system-wide change to one of the global linker options at the same time as the other work was done. Having experienced the joys of upgrading this particular computer system before, I refused. We were already making two large changes to the system – one more than I usually like to make, but necessary – and adding a third large change was just asking for trouble. It's hard enough to track down problems caused by one change. I believe my exact words were "You may be able to find someone in the office who is willing to do it your way, but it won't be me."

We ended up doing it my way, but not without a fight. I backed out the change to the linker options (the customer had already made that change to the build script), and we did the OS and hardware upgrade. I then researched the linker option change, and discovered that other architectural aspects of the system precluded what the customer wanted to do. Had we lumped all three changes together the system would have crashed, and we wouldn't have had a clear idea of where the error was located.

Here's another example; same customer, same system. The engineer discovered that in one of the peripheral cabinets we had routed a control signal through both sides of a double pole, double throw relay. (The outgoing signal went through one pole; the return through the other.) He wanted to rewire the cabinet because using both sides of the relay would decrease the mean-time-to-failure of the relay. Doing what he wanted would have meant incurring a huge cost to change the wiring, plus the cost of changing the drawings, and at the time it cost them \$1000.00 USD just to have a draftsman change the date on a drawing – all to protect against a supposed increase in the likelihood of the failure of a five dollar relay whose mean-time-to-failure was already much longer than the expected life of the system. An expletive was involved on my part (I was much younger then). Not very professional, and I regret my reaction to this day, but I got the point across. The system as delivered was in use for 15 years (more than twice its designed lifetime). Even if they had to replace that relay once a year, the total cost of repair was a fraction of what the redesign and rework would have cost (without even including the time value of money).

I don't recommend profanity, but I do believe that it is better to withdraw from a project or a task than to take part in something which you believe to be misguided, unethical, or illegal.

Pay attention to finances

It always amazes me when I hear about a small business owner who doesn't have any idea of the financial health of his or her company. Yes, I know we don't go into business to do the accounting (unless you're an

Have a place for everything, and file everything in its place... You are less likely to get in trouble for keeping more records than required.

accountant!), but it's important – you have to do it. It may not be practical or affordable to hire someone else to do it for you, and even if you can afford it, you need to have some understanding of the system and the numbers.

Rule Number One: Everyone gets paid before you. Everyone. Particularly taxing authorities. The last thing you need is to get crossways with a government tax agency, at any level. I file three monthly tax forms, three quarterly tax forms, and multiple end-of-year tax forms. Most of them have money due with them. File the reports and pay the tax due, on time. It's the best way to stay out of trouble.

Rule Number Two: Keep great records. Not just good, but great. We write software for a living, which requires an attention to detail required of few others (some – my wife – might call it anal retentiveness). Use that attention to detail to organize your records to a fare-thee-well. Have a place for everything, and file everything in its place. Keep all of the records and receipts required by your tax authorities, plus some. You are less likely to get in trouble for keeping more records than required.

Several years ago my company was audited by the state taxation division. A discrepancy was flagged between the income stated on our end-of-year tax returns and the income on which the company had paid gross receipts tax (similar to a VAT). The discrepancy was due to the fact that almost all of my business comes from out of state, which is not subject to the state GRT.

Audit day came, and if I could have been plucked like a string I would have rung a high C. I had pulled three years of company sales records, invoices, and check stubs. I had copies of all my contracts and purchase orders. I had all of my personal tax returns. I had an official letter from my attorney documenting the section of state law that exempted the out-of-state work from GRT. Everything was laid out on my dining room table (I work from home), in folders, labeled, in chronological order.

The audit I had dreaded took 45 minutes. The auditors told me they wished all of their audits were that easy. They put a letter in my file explaining the nature of my business so the next time I got flagged it would be available to that auditor. I haven't heard from them since.

Understand the concept of risk

The two most common contractual ways a contractor or consultant is hired are time and materials (T&M) and fixed cost. In a T&M contract you get paid by the hour, and are reimbursed for expenses and any materials you purchase to fulfill the contract. In a fixed cost contract you say you will do the work for a set amount of money, in a set amount of time, and you are responsible for paying for all of your expenses and materials. Your contract may also be a hybrid of these two approaches.

This is where the concept of risk comes in. In a T&M contract, most if not all of the risk is on your customer. They may specify a limit to how much they are willing to spend (a 'not to exceed' clause), but in the end if the work takes more time and money than originally estimated it is their problem. In a fixed cost contract all of the risk is on you. You have to

deliver a defined scope of work, in a fixed amount of time, and it doesn't matter to your customer how much it costs you to do the work.

If you are going to take on a fixed cost contract, be very sure you can do the scope of work, in the amount of time specified, and at or below the costs you estimate. (Don't promise to do something you don't know you can do.) I prefer T&M contracts, but have taken fixed cost contracts when the work is something I have done before. I also add risk to the quote, by increasing my hourly rate and over-estimating time and expenses. It is very important that the scope of work and time constraints be very well defined before signing the contract. Also keep in mind that you are on the hook for warranty work on fixed cost contracts, so you should figure extra money in for that, as well.

My T&M contracts usually invoice once a month. On fixed cost contracts I try to negotiate a payment schedule that includes small start-up payment (particularly if I have to buy hardware), regular payments based on partial deliverables, and large payment on delivery. I also usually let my customer keep five or ten percent of the total cost as a retention payment, payable upon completion of some sort of integration test. This shows that you are committed to sticking with the project to completion.

Several years ago I quoted a fixed cost contract for an interface to hardware that was new to my customer. Since the hardware was new to me, too, I wrote into my proposal that the quote was contingent on getting assurances from the hardware vendor that we would have their full support. When I learned that the hardware vendor wasn't going to provide any support, I was able to convert the contract to T&M, moving the risk from me to my customer. It turned out that because we had to work out the details of the protocol without help, the work took 35% longer than I had estimated (and my estimate had been very conservative to begin with).

On a related note – beware the 'budgetary estimate' trap. An estimate tends to become 'the price'. If you are asked for an estimate, always estimate high. You can always decrease the price, but it is very difficult to increase it once the estimate is out there.

Invest in your skills and your tools

As a reader of this magazine (and perhaps a participant in the yearly conference), you already appreciate the benefits of keeping your skills current. Your skill set is what you are selling to your customers. It pays to keep up to date. In addition to the ACCU, I belong to the IEEE, the Computer Society, and the ACM, and try to work my way through the magazines that come with the memberships. I attend one professional conference every year, and hope to bump that number up to two next year. (ACCU Bristol 2015, here I come!)

Good tools are also important. Since I design circuit boards as well as write software, over the years I've bought a quality soldering iron and desoldering station; a digital oscilloscope; and an inexpensive logic analyzer. (I have a better tool set than some of my customers.) There are at least six PCs in my office, of varying vintage, including two that I purchased to support work for just one client. Being willing to spend money on tools to

support a customer is an easy way to show your commitment to their project.

What about software? There is a lot of free software out there, and I use some of it, but most of my customers are Windows shops, so I have an MSDN subscription. I also have three subscriptions for PC-Lint – one for each of my main development computers – and two licenses for Windows Office tools (desktop and laptop). I also have a license for backup software. (Don't forget to do backups! And make sure you can actually recover backed up files.) Accounting software and end-of-year tax software round out the list.

Other 'tools' you may need, that may not be obvious, and may not be directly tied to your work: a desk, chair, filing cabinets (to store those excellent records), printers, internet access, phone line and/or mobile phone, and a dedicated FAX line.

I mention all of this because this stuff costs money and has to be budgeted from your income. Remember, everyone else gets paid first.

Insurance

As much as we may hate to admit it, we all make mistakes, and some mistakes are more costly than others. Insurance is a way to mitigate the risks associated with mistakes, and to protect you and your family from the costs of those mistakes.

My contracts typically require three types of insurance: errors and omissions (E&O), general liability, and automobile. E&O insurance is professional liability insurance, which pays off in the event of a professional mistake. General liability insurance covers non-professional liability (such as injury to another person). Automobile insurance covers accidents you may have while on a customer's property.

I work in the nuclear power industry, which makes it very difficult for me to find an insurance carrier, and makes the policies I can get more expensive than they otherwise might be. (A colleague pays one third of what I do because he didn't disclose that he works in the nuclear industry.) I could save a lot of money by not declaring my nuclear work, but that would likely result in my policy being invalidated the first time it is needed. Be honest about the industries in which you work. Insurance is about mitigating risks, so it makes no sense to risk the insurance.

Sometimes it is just business

It's hard to not take things personally when you're in business for yourself. Being told your services are no longer required, when you have done nothing to deserve getting fired, can be a blow to the ego. It is also hard to not get a contract or a job after spending time and money trying to earn it. It is important to recognize that businesses make decisions based on business requirements. It may not be personal to them – it shouldn't be personal to you.

It can help to be able to separate a person's corporate persona from their private persona. I have worked with several people who were great individuals, but terrible bosses. At work they were holy terrors; get them away from work and they were good friends, gracious hosts, and delightful to be around.

On the other hand, if you did something that deserved firing, you should take the time to reflect on what you did and why it led to the outcome it did.

Be scrupulously honest and ethical

As an individual, your reputation is one of your two most important attributes (the other being your technical competence). A good reputation opens doors to other work. Customers may recommend you to their peers, and give you good references when asked. A bad reputation can easily put you out of business, and once acquired can be devilishly difficult to turn around. The best way to earn and keep a good reputation is to run your business as honestly and ethically as possible.

Ethics is about how we act when no one else is looking. It is not that difficult to run your business in an ethical manner. Do the right thing. Act as a faithful agent for your customers [NSPE]. Generate accurate invoices. Pay your vendors and your taxes on time. Follow the law. Tell the truth.

Want more information on ethics? Look to the codes of conducts and ethics provided by our professional organizations for guidance. The BCS has its Code of Conduct [BCS], as does the IEEE [IEEE] and ACM [ACM]. If you are licensed by the BCS (in the U.K.) or one of the state boards of engineers (in the U.S.) these codes may have the force of law. (My professional engineering license requires that I take at least one hour of ethics training a year, and yes, it is unethical – and illegal – to not get the ethics training and say you did.)

Admit to your mistakes

We call them bugs, but they are really mistakes. When you make one, be willing and able to say "I made a mistake". I know I hate saying those four words, and the best way to keep from having to admit to a mistake is to work very hard to not make one, but none of us is perfect. (Fran – I made a mistake committing to this deadline. See, that wasn't too hard.)

Growing the business

At the beginning of this article I mentioned there is one aspect of being a business owner that I have never been able to figure out. (There are more than one, but this is the big one.) I remain a one-man shop, mostly because I've never been able to solve the chicken-and-egg problem. I can't afford to hire someone unless I have paying work for them to do, and I can't take on more work than I can handle because of the risk of not being able to find someone to do that work.

It seems that real entrepreneurs (read 'risk-takers') don't spend a lot of time worrying about those types of issues. I do. It goes back to not taking on work that I know (or at least reasonably expect) I can't do. If you have a solution to this problem, I'd like to discuss it with you.

Afterword

I didn't set out to be an entrepreneur – I just fell into it. My wife wanted to move closer to home, which was out of state, which meant I had to leave my job of 13 years at the large company.

At about the same time, the large company sold off the division for which I had worked. My adventure in self-employment began with a phone call from a long-time colleague at one of the newly created companies. "We have a couple of weeks of startup work at a water treatment plant in Modesto, California, starting next Monday. Are you interested?" "Sure," I replied, "I've got nothing else to do." And off I went.




Before those two weeks were up I got another phone call. "We've got two weeks of startup work at a water treatment plant in Norfolk, Virginia, next week. Are you available?" (For those of you keeping score, those cities are on opposite coasts.) "Sure", I replied, "I've got nothing else to do." And I've had nothing else to do ever since. ■

Acknowledgments

Thanks to Fran and the reviewers for their making this a better article.

References

- [ACM] Association for Computing Machinery, Code of Ethics <http://www.acm.org/about/code-of-ethics>
- [BCS] BCS, The Chartered Institute for IT, Code of Conduct <http://www.bcs.org/category/6030>
- [IEEE] Institute of Electrical and Electronics Engineers, Code of Conduct http://www.ieee.org/about/ieee_code_of_conduct.pdf
- [NSPE] National Society of Professional Engineers, Code of Ethics <http://www.nspe.org/resources/ethics/code-ethics>

Dear Santa Claus,
but how is Mrs Claus? well about  my presents for
then a bucket of candy, awesum clothes, really
with a totally coool dance CD collectshun
don't forget a big party  with lotsa fwienDs an'
could  then watch t.v. forever, while I got
an' a huge monster truck for ~~me~~ my friend
hoping an' Hoping an' wishing for no more school
Maybe you'll get my family a present, but
to get ME a lot an' lot of SNOW!
HOPE FULLY YOURS,

For help with your very important documents, get in touch.

We can help you with product manuals, user guides, online help, training materials (including e-learning), bids and proposals...

Order Notation in Practice

What does complexity measurement mean?

Roger Orr reminds us of the academic definition and looks at some real life situations.

Most computer programmers have heard of Order Notation – if you have studied computer science then it's almost certain you'll have studied this at some point during the course.

The notation is a way of describing how the **number of operations** performed by an algorithm varies by the **size of the problem** as the size increases.

But why do we care? Almost no-one is actually interested *directly* in this measure – but many people do care greatly about the performance of a function or algorithm. The complexity measure of an algorithm will *affect* the performance of a function implementing it, but it is by no means the only factor.

There are a number of different ways to measure the performance of a function, with overlap, or at least strong correlation, between many of them. Examples of common performance measures include:

- Wall clock time
- CPU clock cycles
- Memory use
- I/O usage (disk, network, etc)
- Power consumption

Complexity measurement is (normally) used to approximate the number of operations performed and this is then used as a proxy for CPU clock cycles and hence performance (or at least one of the measures of performance). However, it is a simplification of the overall algorithm; it may be a measure of only one of the operations involved and it may ignore other factors, such as memory access costs that have become increasingly important in recent years.

Introduction, or re-introduction, to order notation

Order Notation is a classification of algorithms by how they respond to changes in size.

It uses a big O (also called Landau's symbol, after the number theoretician Edmund Landau who invented the notation). We write $f(x) = O(g(x))$ to mean:

There exists a constant C and a value N such that

$$|f(x)| < C|g(x)| \quad \forall x > N$$

There may be a variety of possible ways of picking C and N . For example, consider the functions: $f1(x) = 2x^2 + 3x + 4$ and $f2(x) = x^2 + 345678x + 456789$.

For $f1$ we notice that $3x + 4$ is less than x^2 when x is bigger than four. Hence $f1(x) < 3x^2$, for $x > 4$. So we get $C = 3$, $g(x) = x^2$ and $N = 4$.

For $f2$ the numbers are a little larger but a similar method leads to seeing that $f2(x) < 2x^2$ for all values of $x > 345678$ and hence get $C = 2$, $g(x) = x^2$

Roger Orr has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

and $N = 345678$; or we might start with a larger value for C , say 4000, and then have $f2(x) < 4000x^2$ for all values of $x > 87$ hence get $C = 4000$, $g(x) = x^2$ and $N = 87$.

For the purposes of order notation it doesn't matter what C and N are nor how large or small they are – they are constants; the important item is the function $g()$ – in this example both $f1$ and $f2$ are the *same* complexity, $O(x^2)$. In simple polynomial functions like these you don't need to do the full analysis to find C and N as the complexity is simply the biggest power of x .

With more complex formulae it can be much harder to come up with appropriate constants and expressions; fortunately the complexity for many common algorithms is well-known (as we shall illustrate below).

Note too that for the O measure of complexity the function g may not be the smallest value needed. $f3(x) = 16$ would usually be described as $O(1)$ but can also be described as $O(x^2)$ (with $C = 1$ and $N = 4$) but like any estimate it's normally more useful the 'closer' it is and this function. This matters more for non-trivial algorithms where it may be very hard to exactly specify the best complexity function but much easier to specify a slightly larger one.

There are other symbols used in order notation such as the little-o symbol and the big-theta symbol. For example if both $f(x) = O(g(x))$ and $g(x) = O(f(x))$ then we can write $f(x) = \Theta(g(x))$. These are used in the mathematical theory of complexity but are generally less common in computer science.

Some common orders

Here are some common orders, with the slower growing functions first:

- $O(1)$ – constant
- $O(\log(x))$ – logarithmic
- $O(x)$ – linear
- $O(x^2)$ – quadratic
- $O(x^n)$ – polynomial
- $O(e^x)$ – exponential

Order arithmetic

When two functions are combined the order of the resulting function can (usually) be inferred quite simply from the orders of the original functions. When adding functions, you simply take the biggest order.

eg. $O(1) + O(n) = O(n)$

When multiplying functions, you multiply the orders

eg. $O(n) * O(n) = O(n^2)$

So, more generally, when a function makes a *sequence* of function calls the overall order of the function is the same as the highest order of the called functions.

```
void f(int n) {
    g(n); // O(n.log(n))
    h(n); // O(n)
}
```

Order Notation is a classification of algorithms by how they respond to changes in size

In this example $f(n) = O(n \log(n))$. This highest order is sometimes called the *dominant* complexity since as the number of items increases this value will dominate the overall complexity of the whole calculation.

For a function using a loop the order is the product of the order of the *value* of the loop count and the loop body

```
void f(int n) {
    int count = g(n); // where the value of count
                      // is O(log(n))
    for (int i = 0; i != count; ++i) {
        h(n); // O(n)
    }
}
```

Hence in this example too $f(n) = O(n \log(n))$

Many standard algorithms have a well-understood order. One of the best known non-trivial examples is probably quicksort, which ‘everyone knows’ is $O(n \log(n))$. Except when it isn’t, of course! On *average* it is $O(n \log(n))$ but the worst-case complexity, for particularly unhelpful input values, is $O(n^2)$.

Also, this is the computational cost, in terms of the number of **comparison** operations, not necessarily all operations or the memory cost.

The C++ standard mandates the complexity of many algorithms, using various different operation counts.

For example, `container::size`:

Complexity: constant.

and `std::list::push_back`:

Complexity: Insertion of a single element into a list takes constant time and exactly one call to a constructor of T.

There are also various flavours of sorting. For example, `std::sort`:

Complexity: $O(N \log(N))$ comparisons.

and `std::stable_sort`:

Complexity: It does at most $N \log^2(N)$ comparisons; if enough extra memory is available, it is $N \log(N)$. (The standard is silent on what ‘enough’ means!)

and `std::list::sort`:

Complexity: Approximately $N \log(N)$ comparisons

The .Net documentation provides complexity for (some) algorithms. For example, `List<T>.Sort`:

On average, this method is an $O(n \log n)$ operation, where n is Count; in the worst case it is an $O(n^2)$ operation.

Java too provides complexity measures for some algorithms. For example, `Arrays.sort`:

This implementation is a stable, adaptive, iterative mergesort that requires far fewer than $n \lg(n)$ comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered...

(The Java spec uses \lg rather than \log – but all logarithms have the *same* complexity so it is immaterial!)

```
int strlen(char *s) /* source: K&R */
{
    int n;

    for(n = 0; *s != '\0'; s++)
    {
        n++;
    }
    return n;
}
```

Listing 1

However, unlike C++, neither .Net nor Java seem to provide much detail for the cost of *other* operations with containers. This makes it harder to reason about the performance impact of the choice of container and the methods used: not everything is dominated by the cost of sorting alone!

So that’s the theory; what happens when we try some of these out in an actual program on real hardware? Your own figures may vary because of machine and operating system differences (different clock speeds, varying amounts of memory, different speeds of memory access and cache sizes and different choices of memory allocation strategies).

strlen()

This seems like a straightforward function and at first sight measuring its complexity should be simple enough: $O(n)$ where n is the number of bytes in the string. You may even have read some example source code for `strlen()` if and when you first learned C (see Listing 1).

Have you looked inside `strlen()` recently? Things have got much more complicated than this in practice! Here’s an extract from an implementation of the function on x64 – probably rather more than you wanted to know... (see Listing 2).

However, despite the re-write in assembly language and the tricks to enable checking 64 bits at once in the main loop this code is still $O(n)$.

Naïvely we write some code that calculates the elapsed time for a call to `strlen()` like this:

```
timer.start();
strlen(data1);
timer.stop();
```

However, on most compilers in release mode the call appears to take **no time at all**

The reason for this is that calls to functions like `strlen()` can be optimised away completely if the return value is not used.

It’s vitally important with performance measuring to check you’re measuring what you *think* you’re measuring!

So we change the code to use the return value of `strlen()` and set up a couple of strings to test against:

```
char const data1[] = "1";
char const data2[] = "12345...67890...";
```

It's vitally important to check you're measuring what you think you're measuring!

```

strlen:
    mov     rax,rcx      ; rax -> string
    neg     rcx
    test   rax,7        ; test if 64 bit aligned
    je     main_loop
    ; ...
    ; loop until aligned (or end of string found)
    ; ...
main_loop:
    mov     r8,7EFEFEFEFEFEFFh
    mov     r11,8101010101010100h
    mov     rdx,qword ptr [rax]      ; read 8
bytes
    mov     r9,r8
    add     rax,8
    add     r9,rdx
    not     rdx
    xor     rdx,r9
    and     rdx,r11
    je     main_loop
    mov     rdx,qword ptr [rax-8]
                ; found zero byte in the loop

    test   dl,dl
    je     byte_0      ; is it byte 0?
    test   dh,dh
    je     byte_1      ; is it byte 1?
    shr    rdx,10h
    ; ... and the rest
byte_0:
    lea    rax,[rcx+rax-8]
    ret
    
```

Listing 2

```

Compare time for
    v1 = strlen(data1)
against
    v2 = strlen(data2)
    
```

Once again, you may get a bit of a surprise, depending on which compiler and flags you're using, as a call to `strlen()` of a constant string can be evaluated at *compile* time and hence is $O(1)$ (also known as constant time.) I repeat – it's vitally important to check you're measuring what you *think* you're measuring!

So for our third attempt we set up the string at runtime and now we get the graph we were expecting that demonstrates `strlen()` is $O(n)$. (Figure 1.) Series 1 and Series 2 are two separate runs over the same range of lengths, and demonstrate how repeatable the results are. However, this nice simple straight line graph breaks if we make the length a little larger. (Figure 2.) The graph is no longer linear and also no longer as consistent between runs. The reason is the the machine I used for this test has 2814Mb of RAM and

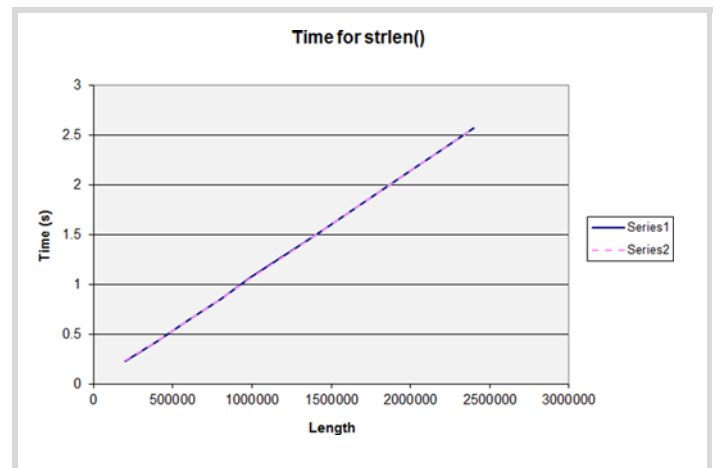


Figure 1

the operating system starts swapping memory to disk as the size of the string gets near to this value.

There is another anomaly with smaller strings too: this is present in the first graph above but not obvious to the naked eye. If we change the graph to display the average time for each byte we can see a jump at around 600,000 bytes. (See Figure 3.) Again, figures may differ on different hardware – **this** machine has 64K L1 + 512K L2 cache per core, ie. 589,524 bytes. So here we can see the effect of the cache size of this test.

What we have demonstrated here is that the runtime execution time of `strlen()` is $O(n)$ to a very good approximation when n is between cache size and available memory. The complexity in terms of the number of access operations on the string *is* still linear for larger strings than this but the effects of swapping dwarf this. It is likely that the cost of swapping is

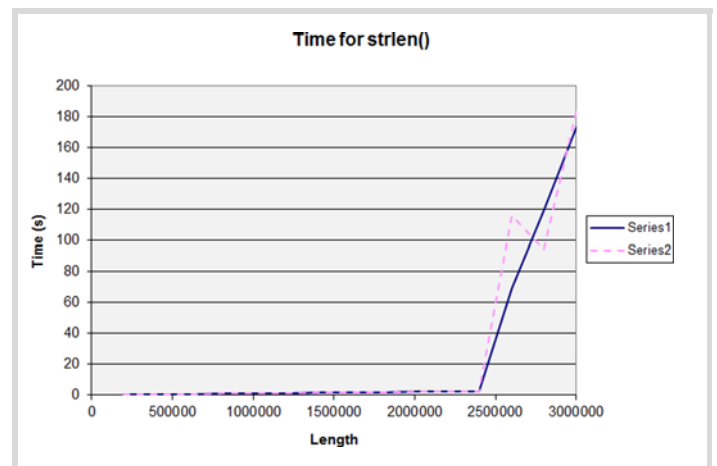


Figure 2

Make sure you are testing against similar data sets to those you will experience in real executions!

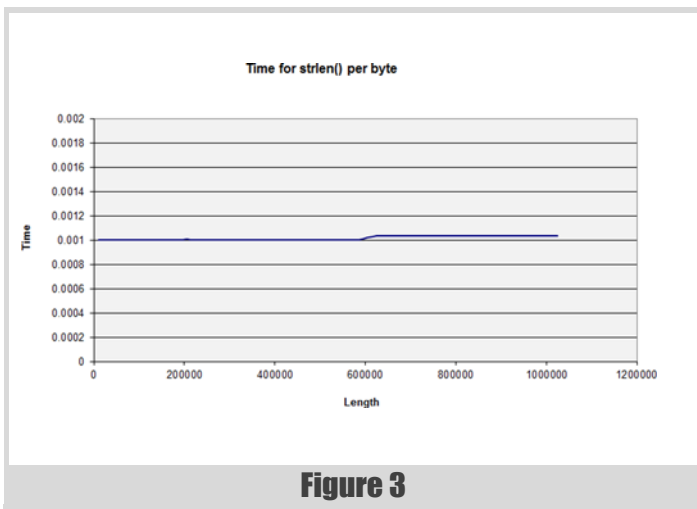


Figure 3

also $O(n)$, but the ‘scaling’ factor C is much bigger (perhaps 250–300 times bigger in this case on this hardware).

Let us see what happens if we try the same sort of operation but a slightly more generic algorithm by swapping over from using `strlen()` to using `string::find('\0')`

We expect this will behave like `strlen()` and indeed it does – consistently slightly slower (Figure 4).

Sorting

We now turn our attention to various sorting algorithms and how they behave under various conditions. We start with a (deterministic) `bogo` sort (see Listing 3).

```
template <typename T>
void bogo_sort(T begin, T end)
{
    do
    {
        std::next_permutation(begin, end);
    } while (!std::is_sorted(begin, end));
}
```

Listing 3

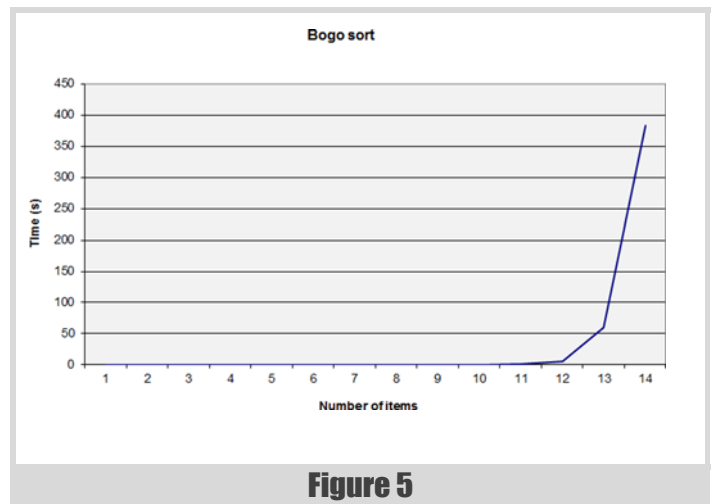


Figure 5

This is not a sort you ever want to use in production code as it has $O(n \times n!)$ comparisons. Except when it doesn't – here are some timings.

- 10,000 items: 1.13ms
- 20,000 items: 2.32ms
- 30,000 items: 3.55ms
- 40,000 items: 4.72ms

This appears to be $O(n)$ – but ... how? I ‘cheated’ and set the initial state carefully. When measuring the performances of sorting you must be very careful about the best and worst cases. Make sure you are testing against similar data sets to those you will experience in real executions! In this case, I changed the generation of the data sets to use a randomised collection and then I obtained the expected sort of graph (Figure 5).

I didn't do any runs with more than 14 items as the time taken was so long! You can see that we appear to hit a ‘wall’ at 13 or 14 items. But appearances can be deceptive – if we take the graph after eight items we get a similar ‘wall’ effect. (Figure 6.)

While graphs can make some things easy to visualise they can also slightly mislead the eye: the wall effect seen here depends on the vertical scale of the graph.

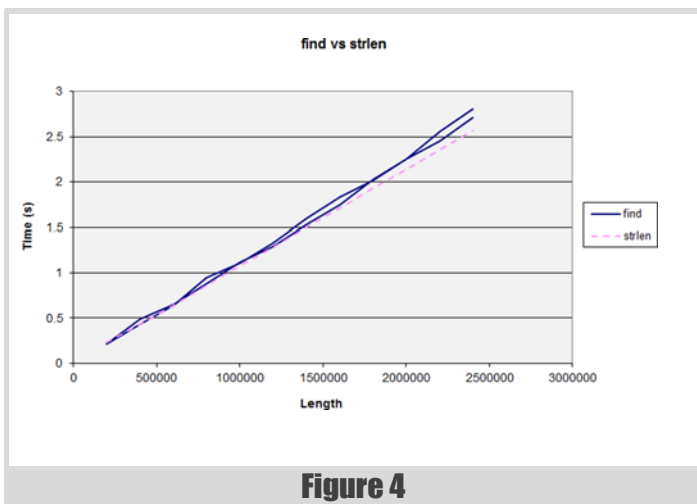


Figure 4

std::sort is faster than qsort which can come as a surprise to those who assume C is always faster than C++

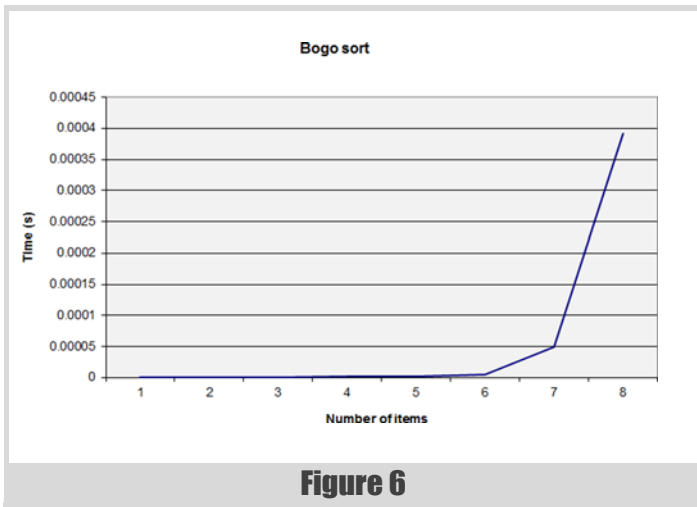


Figure 6

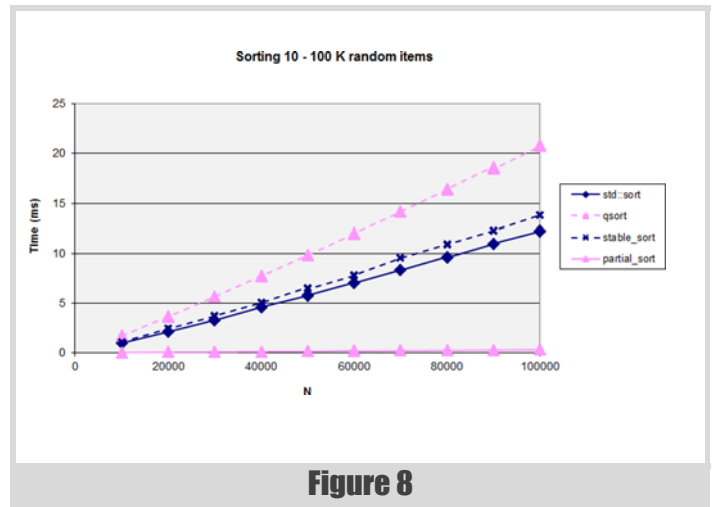


Figure 8

What this means in practice is that the point at which the increasing complexity cost of a poor algorithm significantly affects the overall performance of the whole function or program will depend on what the relative timings are of the algorithm and the whole thing.

Let us leave the quaint bogo sort behind and try out some more performant flavours of sorting. `std::sort` which is the commonest used in C++, `qsort` the equivalent for C, `bubble_sort` which is easy to explain and demonstrate, `stable_sort` which retains the order of equivalent items and `partial_sort` which sorts the first *m* items from *n* (in this test I sorted the 'top ten' items).

(If you want to visualise some of these sort algorithms in practice I must mention AlgoRythmics – illustrating sort algorithms with Hungarian folk dance: <https://www.youtube.com/watch?v=ywWBy6J5gz8>)

The dances do help to give some idea of how the algorithm works – they also show the importance of the multiplier *C* in the formula). (Figure 7).

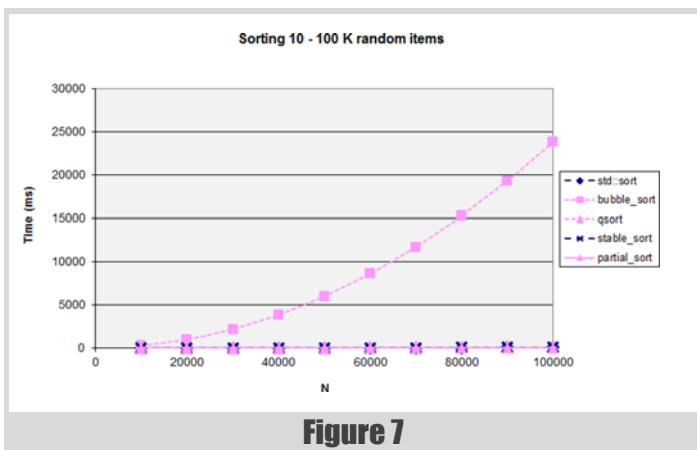


Figure 7

This graph might help to explain a quote from Andrei Alexandrescu: “I’d like to go back in time and kill the inventor of bubblesort”.

Removing this sort algorithm the graph now reveals the differences between the others (Figure 8).

Notice that `std::sort` is faster than `qsort` which can come as a surprise to those who assume C is always faster than C++. It also shows that you do seem to pay a small cost for the stability of `stable_sort`. However, the real surprise for many people may be the excellent performance of `partial_sort` which is considerably faster than all of the other algorithms that sort the entire data set. When confronted with a sorting problem it is worth asking whether or not you need the full set sorted – if you only need a small number of the top (or bottom) items then `partial_sort` may prove to be a more performant solution.

However, that was with *randomised* input data – in practice a lot of real data is not randomly sorted. When sorting *nearly* sorted data the `bubble_sort` algorithm can perform surprisingly well. (Figure 9.)

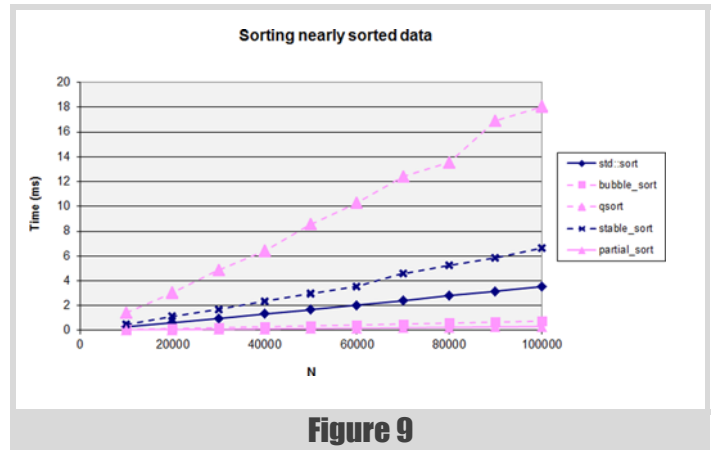


Figure 9

modern computers perform very much better on data with good locality of reference

This demonstrates how important it is to test performance in an environment as similar as possible to the expected target – many algorithms are sensitive to the input data set and if the test data set has different characteristics than the production data you may make a non-optimal choice.

Comparing and contrasting list and vector

The C++ collection classes, in common with some other languages, has a number of standard collection classes with slightly different interface and implementation. Two of these are `std::list<T>` and `std::vector<T>`. Both contain an ordered collection of values of type `T`, in the one case the underlying implementation is a doubly-linked list of nodes and in the other it is a contiguous array of objects. The C++ algorithm `std::sort` can be used to sort the vector, but not the list (since the list does not provide a random-access iterator). However, there is a member function `sort` in `std::list`. The complexity measure of `std::sort` is the same as `std::list::sort` – so what's the difference in practice?

In terms of implementation, sorting a vector must actually copy the objects around inside the underlying array; whereas sorting a list can simply swap around the forward and back links without needing to move the payloads.

So let's try it. Figure 10 plots two things at once: on the left hand axis we have the time to sort the collection and on the right hand axis the actual number of comparison operations performed (since the complexity is stated in terms of the number of comparisons).

There are several points to note in this graph. Firstly, sorting the vector involves performing nearly twice as many comparisons as for the list when sorting the **same** data set, so while both are of the same theoretical complexity ($n \cdot \log(n)$) the scaling of this (C from the formula at the start of this article) is different. However, even though list does far fewer comparisons it is consistently slower than vector and gets more so as the number of items sorted increases.

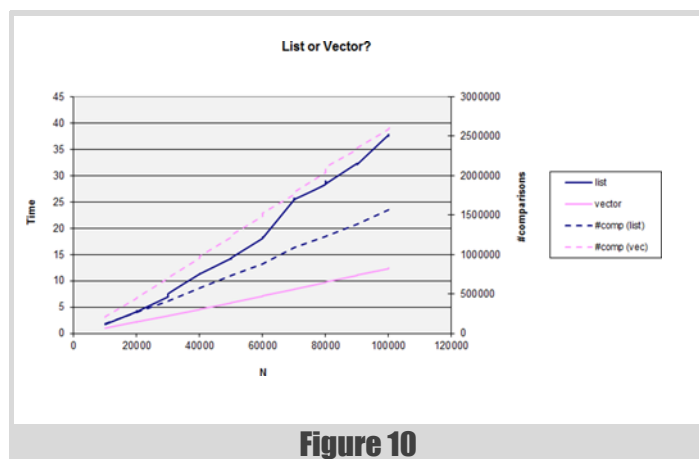


Figure 10

This is a 'worst case' example as the object I used for this example merely wraps an integer, and so it is actually quicker to move the payload (1 machine word) than to swap the pointers in the list (2 machine words)!

The obvious question then is what happens as the size of the payload increases. If we retain the original payload as the 'key' comparison value then number of comparisons will remain exactly the same, the only change will be in the amount of data moved (for the vector). The list will continue to swap pointers and doesn't even need to access the whole object.

I repeated the test above with gradually increasing sizes of payload for both vector and list. As expected, as the payload increased, the performance of the vector dropped until it eventually approximately equalled that of the list and then lagged behind it. However, I was surprised how large the payload size needed to be before this approximately equal performance was achieved: for *this* test on *my* hardware it was at around **100 bytes**. I would have expected the increasing cost of copying to have had its effect more quickly.

One of the reasons is that, as we are all gradually coming to understand, modern computers perform very much better on data with good locality of reference. A vector is about as good as you can get in the regard – the objects in the vector are contiguous in memory and there are no additional control structures involved inside the data. While the specifics vary, the principle of locality is important and if it is *multiplicative* with the algorithmic complexity it can change the complexity measure of the overall function.

The performance of the list test is instructive in this regard: as discussed above when using the same value for the comparison the sort is doing **exactly** the same sequence of comparisons and link swaps. I found that sorting a list with a 1Kb payload took between two and three times as long as sorting a list of integers. At first I thought there might be a simple relation to the cache line size and that once the object payload exceeded the cache size (64b on my hardware) there would be no further effect on performance; but this did not seem to be the case. See below, the graph of time against the number of items sorted and the log of the object size. (Figure 11.)

Perhaps we should be measuring the complexity of sort algorithms in other terms than just the number of comparisons?

Cost of inserting

Suppose we need to insert data into a collection and performance is an issue. Looking at the various standard containers we might be using, what might be the differences between using: `std::list`, `std::vector`, `std::deque`, `std::set`, or `std::multiset`?

To refresh your memory, the cost of inserting for each of these is:

- `std::list` 'constant time insert and erase operations anywhere within the sequence'
- `std::vector` 'linear in distance to end of vector'
- `std::deque` 'linear in distance to nearer end'
- `std::set` and `std::multiset` 'logarithmic'

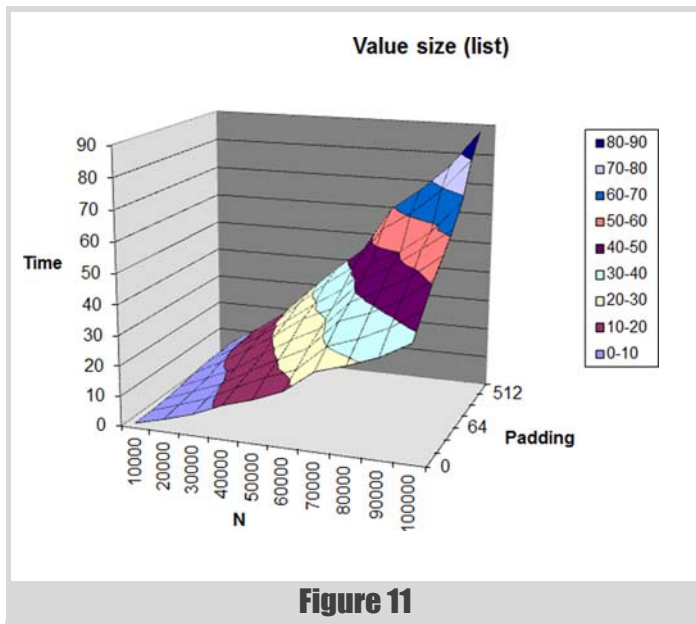


Figure 11

We are also affected by the time to find the insert point.

I tested randomly inserting 10,000 items into the various collections, with the following results:

- `std::list` ~600ms
very slow – cost of finding the insertion point in the list
- `std::vector` ~37ms
Much faster than list even though we're copying each time we insert
- `std::deque` ~310ms
Surprisingly poor – spilling between buckets
- `std::set` ~2.6ms – our winner!

It can be significantly faster to use a helper collection if the target collection type desired is costly to create. In this example, if I use a `std::set` as the helper object and then construct a `std::list` on completion of the inserts then the overall time to create the sorted list drops to ~4ms. The use of a helper collection will obviously increase the overall memory use of the program at the point of converting the source to the target collection, but the performance gains can be considerable.

If we change the insertion order from a random one to inserting 10,000 already sorted items, then the performance characteristics change again (there are two choices of sort order to select whether items are added to the front or to the back of the collection):

- `std::list` ~0.88ms
Fast insertion (at known insert point)
- `std::vector` ~0.85ms (end) / 60ms (start)
Much faster when appending
- `std::deque` ~3ms
Roughly equal cost at either end; a bit slower than a `vector`
- `std::set` ~2ms (between `vector` and `deque`)

This article is about order notation, so what happens to these numbers if we change the number of items? Let us try using ten times as many items:

- `std::list` ~600s (1000×)
- `std::vector` ~3.7s (100×)
- `std::deque` ~33s (100×)
- `std::set` ~66ms (33×)

The cost of finding the insertion point for `std::list` dwarfs the insert cost. It is easy to overlook parts of an algorithm to discover later they have added significant hidden complexity.

Can we beat `std::set`?

C++11 has some additional associative collection classes that use hashing for improved performance (at the expense of removing the natural sort order). If we try a naïve use of `std::unordered_set` we find it is very slightly slower at 10K (~2.8ms vs ~2.6ms) but does out-perform `std::set` better at 100K items (~46ms vs ~66ms)

However, we may have additional knowledge about our value set and so can use a different hash function – as is the case in my test program where a trivial, and fast, identity hash function can be used. This enabled `std::unordered_set` to achieve times of ~2.3ms (10K) and ~38ms (100K).

Conclusion

The algorithm we choose is obviously important for the overall performance of the operation (measured as elapsed time). As data sizes increase we eventually hit the limits of the machine; the best algorithms are those that involve least swapping. For smaller data sizes the characteristics of the cache will have some effect on the performance.

While complexity measure is a good tool we must bear in mind:

- What are N (the relevant size) and C (the multiplier)?
- Have we identified the function with the dominant complexity?
- Can we re-define the problem to reduce the cost?

Making it faster

We've seen a few examples already of making things faster.

- Compile-time evaluation of `strlen()` turns $O(n)$ into $O(1)$
- Can you pre-process (or cache) key values?
- Swapping setup cost or memory use for runtime cost
- Don't calculate what you don't need (We saw that, if you only need the top n , `partial_sort` is typically much faster than a full sort)
- If you know something about the characteristics of the data then a more specific algorithm might perform better – for example `strlen()` vs `find()`, sorting nearly sorted data, or a bespoke hash function.

Pick the best algorithm to work with memory hardware

- Prefer sequential access to memory over random access
- Smaller is better
- Splitting compute-intensive data items from the rest can help – at a slight cost in the complexity of the program logic and in memory use. ■

Acknowledgements

Many thanks to the *Overload* reviewers for their suggestions and corrections which have helped to improve this article.

Further reading

Ulrich Drepper 'What Every Programmer Should Know About Memory':
<http://people.redhat.com/drepper/cpumemory.pdf>

Scott Meyers at ACCU 'CPU caches':
http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

Bjarne Stroustrup's vector vs list test (especially slides 43–47):
<http://bulldozer00.com/2012/02/09/vectors-and-lists/>

Herb Sutter's experiments with containers:
<http://www.gotw.ca/gotw/054.htm>
and looking at memory use:
<http://www.gotw.ca/publications/mill14.htm>

Baptiste Wicht's list vs vector benchmarks:
<http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/>

People of the .Doc

Technical communication is often misunderstood by the world at large. Andrew Peck breaks down the rhetoric from a technical author's perspective.

We are sometimes invited to 'see ourselves as others see us'. This article takes the reverse viewpoint, seeing a group of people who often work alongside us through their eyes, not ours.

Do you find that your work is treated by friends and family as being a form of witchcraft? That's probably because they're in sway to Clarke's 3rd law, which states that 'Any sufficiently advanced technology is indistinguishable from magic.' [Wikipedia]. Computers, IT professionals and programmers are often treated with a reverence once reserved for doctors and before that priests and shamans. The mystery is two fold: first any modern device or application is – from the perspective of the user – a black box into which they place input to gain a result. They have no knowledge of the intricately crafted logic that allows the most aesthetically simple device to function, and so they adopt the same attitudes we might associate with throwing coins in a wishing well... only the wishing well of IT often throws something back.

It's not just the users and outsiders to blame for this of course. Computer systems are a mystery to the world at large in part because the majority of those who do understand them spend their working life surrounded by others who use the same jargon and do similar things, and so a closed community is created with knowledgeable insiders and unwitting outsiders.

I suppose this is where the technical communicator comes in. We are the deacons and evangelists of the church of high technology. Whilst linking to a blog post of mine, the *Guardian* technology team [Guardian] described us for the uninitiated as "the hapless folk who have to write the manual that you never read but which explains how it actually works".

Let's consider the accuracy of this definition and see if we can suggest an appropriate and approved alternative. Who knows, we may even get an amendment similar to those sometimes found in the cheaper tabloids when they get a footballer's deviance à la mode wrong!

The myth

Having regularly endured a myriad of Christmas movies featuring animated and/or over-acted depictions of Santa Claus, when I read the description of the 'hapless folk', I'm put in mind of the elf who's a little bit 'different', the one who is given some kind of make-work task because he can't be trusted with anything that might do lasting harm if inserted up a nostril. I'm a little disappointed that the popular view of technical writing is of something that happens under duress, for ungrateful disinterested end users. There is also the implication that our writing is somehow pointless, as if the only thing this profession produces is badly translated hand-outs to go with cheap electronics.

The reality

Technical communication can be outwardly very dull, but it's that way for a reason. I feel that as a general rule the more exciting, world changing and expensive the product, the more structured and precise any accompanying documentation becomes (imagine the precision needed in the manual for an anti-tank munitions). The reason for this of course is that

the more fantastical the product, the greater the cost and damage done if something goes wrong and that is essentially where we come in. If 'tech-support' is the cure, we are the prevention that is so much sweeter. It is frustrating to have to have to use the same lexical chunks within a piece of writing, but we are shoeing the technological horse, and florid patterns aren't really of much use to users, translators or localisation teams. (We can save these for other types of writing... in this article alone you'll find anglicised French, Latin and a parody of Islamic theology – none of which would be encouraged in software documentation.)

That's not to say that we're in any way less skilled than our counterparts who write in different ways for different purposes. The novelist or journalist may get away with 'typing', but we are master-users of desktop publishing, word processing and authoring software. I haven't used the buttons in Word's ribbon for 'bold' and 'italic' in a decade, and even the keyboard shortcuts find their outings cut short due to the catalogue of carefully constructed and balanced styles that have documents parading past a client's eyes like an old school soviet military parade.

Based on the above, the definition that I'd like to see in the public domain would be something along the lines of 'the professional specialists who make complex products and procedures clear and accessible to the rest of us'. Accessible documentation is as important as clarity, people should be able to find what they need to know when they need to know it.

Our responsibility ends once a high quality message is out there; if people choose not to read the manual, and as a result shut down a stock exchange, shoot themselves in the foot or put their furniture together upside down there's not a lot we can do about it.

The dream

The above definition is quite accurate, and I'd encourage anyone who's every wondered 'is this a career for me?' to think very carefully about the unique set of skills and traits they'll need to develop. As a reward, I can promise that no one is going to wrap fish and chips in what you write.

If there is a *Deus ex machina* (a term from literature meaning 'God from the machine') we technical communicators are the prophets, scribes and high priests of the 'People of the Doc'. ■

References

[Guardian] <http://www.theguardian.com/technology/blog/2013/jan/07/technology-links-newsbucket>

[Wikipedia] http://en.wikipedia.org/wiki/Clarke%27s_three_laws

Acknowledgements

This article is based on one previously published in *Communicator* (Spring 2013), the journal of the ISTC (www.istc.org.uk).

Andrew Peck is a technical author working for Clearly Stated Ltd near Nottingham. His background is as a Higher Education lecturer and military language trainer. He is a Member of the Institute of Scientific and Technical Communicators (ISTC).

Testing Drives the Need for Flexible Configuration

Inflexible configuration will cause problems. Chris Oldwood demonstrates how to support multiple configurations flexibly.

If you look at a system's production configuration settings you could be fooled into thinking that we only need a simple configuration mechanism that supports a single configuration file. In production it's often easier because things have settled down – the correct hardware has been provisioned, security accounts created, monitoring services installed, etc. But during development and testing is when the flexibility of your configuration mechanism really comes into play.

I work on distributed systems which naturally have quite a few moving parts and one of the biggest hurdles to development and maintenance in the past has been because the various components cannot be independently configured so that you can cherry-pick which services you run locally and which you draw from your integration/system test environment. Local (as in 'on your desktop') integration testing puts the biggest strain on your configuration mechanism as you probably can only afford to run a few of the services that you might need unless your company also provides fairly meaty developer workstations too.

In the past I've found the need to override component settings using a variety of criteria and the following article is definitely not exhaustive, but gives the most common reasons I have encountered for needing to configure something differently. It also goes into a little more detail about how you might support such multiple configurations whilst minimising the potential for duplication.

Per-environment

The most obvious candidate is environmental as there is usually a need to have multiple copies of the system running for different reasons. I would hazard a guess that most teams generally have separate DEV, TEST & PROD environments to cover each aspect of the classic software lifecycle. For small systems, or systems with a top-notch build pipeline and test coverage, the DEV & TEST environments may serve the same purpose. Conversely I have worked on a team that had 7 DEV environments (one per development stream [Oldwood14]), a couple of TEST environments and a number of other special environments used for regulatory purposes, all in addition to the single production instance.

What often distinguishes these environments are the instances of the external services that you will depend on. It is common for all production environments to be ring-fenced so that you only have PROD talking to PROD to ensure isolation. In some cases you may be lucky enough to have UAT talking to some read-only PROD services, perhaps to support parallel running. But DEV environments are often in a sorry state and highly distrusted so are ring-fenced for the same reason as PROD, but this time for the stability of everyone else's systems.

Where possible I prefer the non-production environments to be a true mirror of the production one, with the minimum changes required to work around environmental differences. Ideally we'd have infinite hardware so that we could deploy every continuous build to multiple environments configured for different purposes, such as stress testing, fault injection, DR failover etc. But we don't. So we often have to settle for continuous deployment to DEV to run through some basic scenarios, followed by promotion to UAT to provide some stability testing, and thence to PROD.

Where sharing of production input sources is possible this means is that our inputs are often the same as for production, but naturally our outputs have to be different. But you don't want to have to configure each output folder separately, so you need some variable-based mechanism to keep it manageable so that most settings are then derived, e.g. only the root folder name changes, the relative child structure stays the same and therefore does not require explicit configuration.

The Disaster Recovery (DR) environment is an interesting special case because it should look and smell just like production. A common technique for minimising configuration changes during a failover is to use DNS Common Names (CNAMEs) for the important servers, but that isn't always foolproof. Whilst this means that in theory you should be able to switch to DR solely through network infrastructure re-configuration, in practice you will find not every system you depend on will be quite so diligent.

Per-machine

Next up are machine specific settings. Even in a homogenous Windows environment you often have a mix of 64-bit and 32-bit hardware, slightly different hard disk partitioning, amount of memory, CPUs, etc. or different performance characteristics for different services. Big corporations love their 'standard builds', which largely helps minimise the impact, but even those change over time as the hardware and OS changes – just look at where user data has been stored in Windows over the various releases. The ever changing security landscape also means that best practices change and these will, on occasion, have a knock-on effect on your system's set-up.

By far the biggest use for per-machine overrides I've found, though, is during development, i.e. when running on a developer's workstation. While unit testing makes a significant contribution to the overall testing process you still need the ability to easily cobble together a local sandbox in which you can do some integration testing. I've discovered the hard way what happens when the DEV environment becomes a free-for-all – it gets broken and then left to fester. I've found treating it with almost the same respect as production pays dividends because, if the DEV environment is stable (and running the latest code) you can often reduce the setup time for your local integration testing sandbox by drawing on the DEV services instead of having to run them all locally.

Per-process-type

Virtually all processes in the system will probably share the same basic configuration, but certain processes will have specific tasks to do and so they may need to be reconfigured to work around transient problems. One

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

environment variables are one technique I wouldn't use by default to configure services on Windows ... because they are inherited from the parent process

of the reasons for using lots of processes (that share logic via libraries) is exactly to make configuration easier because you can use the process name as a 'configuration variable'.

The command line is probably the default mechanism most people think of when you want to control the behaviour of a process, but I've found it's useful to distinguish between task specific parameters, which you'll likely always be providing, and background parameters that remain largely static. This means that when you use the `--help` switch you are not inundated with pages of options. For example a process that always needs an input file will likely take that on the command line, as it might an (optional) output folder; but the database that provides all the background data could well be defaulted using, say, an `.ini` file.

Per-user

The final category is down to the user (or more commonly the service account) under which the process runs. I'm not talking about client-side behaviour which could well be entirely dynamic, but server-side where you often run all your services under one or more special accounts. There is often an element of crossover here with the environment as there may be separate DEV, TEST and PROD service accounts to help with isolation. Support is another scenario where the user account can come into play as I may want to enable/disable certain features to help avoid tainting the environment I'm inspecting, such as using a different logging configuration.

Getting permissions granted is one of those tasks that often gets forgotten until the last minute (unless DEV is treated liked PROD which drives the requirement out early). Before you know it you switch from DEV (where everyone has way too many rights) to UAT and you suddenly find things don't work. A number of times in the past I've worked on systems where a developer's account has been temporarily used to run a process in DEV or UAT to keep things moving whilst the underlying change requests bounce around the organisation. Naturally security is taken pretty seriously and so permissions changes always seem to need three times as many signatures as other requests; in the meantime though we are expected to keep development and testing moving along.

Hierarchical configuration

Although most configuration differences I've encountered tend to fall into one specific category per setting, there are some occasions where I've had cause to need to override the same setting based on two categories, say, environment and machine (or user and process). However, because the hardware and software is itself naturally partitioned (i.e. environment/user) it's usually been the same as only needing to override on the latter (i.e. machine/process). For example if a few UAT and PROD servers had half the RAM of the others, then the override could be applied at machine-level on just those boxes because the servers are physically separated (the environment) as UAT and PROD services are never installed on the same host.

What this has all naturally lead to is a hierarchical configuration mechanism, something like what .Net provides, but where `<machine>`

does not necessarily mean all software on that host, just my system components. It may also take in multiple configuration providers, such as a database, `.ini` files, the registry, command line, etc. With something like a database the problem of chickens-and-eggs rears its head and so it can't be a source for bootstrapping settings as you need somewhere to configure a connection string to access it.

As an aside environment variables are one technique I wouldn't use by default to configure services on Windows. This is because they are inherited from the parent process – `Services.exe` – and so any change to the system environment variables requires it to be restarted, which is essentially a reboot [KB].

Hierarchical files

The default file-based configuration mechanism that .Net uses has only two levels of `.config` file, but it's possible to leverage the underlying technology and create your own chain of configuration files. In the past I have exploited this mechanism so that on start-up each process will go looking for these files in the assembly folder in the following order:

```
System.Global.config
System.<environment>.config
System.<machine>.config
System.<process>.config
System.<user>.config
```

Yes, this means that every process will hit the file-system looking for up to 5 `.config` files, but in the grand scheme of things the hit is minimal. In the past I have also allowed config settings and the bootstrapping config filename to be overridden on the command line by using a hierarchical command line handler that can process common settings. This has been invaluable when you want to run the same process side-by-side during support or debugging and you need slightly different configurations, such as forcing them to write to different output folders.

Use sensible defaults

It might appear from this article that I'm a configuration nut. On the contrary, I like the ability to override settings when it's appropriate, but I don't want to be forced to provide settings that have an obvious default. I see little point in large configuration files full of defaulted settings just because someone may need to tweak it, one day – that's what the source code and documentation is for.

I once worked on a system where all configuration settings were explicit. This was intentional according to the lead developer because you then knew what settings were being used without having to rummage around the source code or find some (probably out-of-date) documentation. I understand this desire but it made testing so much harder as there was a single massive configuration object to bootstrap before any testable code could run. It became a burden needing to provide a valid setting for some obscure business rule when all I was trying to test were changes to the low-level messaging layer.

When there are many defaults and only a few overrides, following the hierarchical nature right through to the deployment means that it's easier to see what is overridden

```
[Feeds]
SystemX=\\Server\PROD\Imports\SystemX
SystemY=\\Server\PROD\Imports\SystemY
SystemZ=\\Server\PROD\Imports\SystemZ
```

Listing 1

Configuration file formats

I have a preference for simple string key/value pairs for the configuration settings – the old fashioned Windows .ini file format still provides one of the simplest formats. Yes, XML may be more flexible but it's also considerably more verbose. Also, once you get into hierarchical configurations (such as .Net XML style .config files), its behaviour becomes unintuitive as you begin to question whether blocks of settings are merged at the section level, or as individual entries within each section. These little things just add to the burden of any integration/systems testing.

I mentioned configuration variables earlier and they make a big difference during testing. You could specify, say, all your input folders individually as absolute paths, but when they're related that's a pain when it comes to environmental changes (see Listing 1 for an example).

One option would be to generate the final configuration files from some sort of template, such as with a tool like SlowCheetah [SlowCheetah], which could be done at compile time, package time or deployment time. The source files could then be hierarchical in nature but flattened down to a single deployable file.

When there are many defaults and only a few overrides, following the hierarchical nature right through to the deployment means that it's easier to see what is overridden because its file only lists exceptions. You can then use variables in the core settings and define them in the list of exceptions (for an example, see Listing 2).

The set of variables don't just have to be custom ones, you can also chain onto the underlying environment variables collection so that you can use standard paths such as %TEMP% and %ProgramFiles% when necessary.

```
<System.Global.config>
[Variables]
FeedsRoot=%SharedData%\Imports
```

```
[Feeds]
SystemX=%FeedsRoot%\SystemX
SystemY=%FeedsRoot%\SystemY
SystemZ=%FeedsRoot%\SystemZ
```

```
<System.PROD.config>
[Variables]
SharedData=\\Server\PROD
```

Listing 2

Summary

This article took a look at the differences between configuring a complex system for use in production and the many other environments in which it needs to operate, such as development and testing. We identified a number of patterns that help describe why we might need to configure the system in different ways and formulated a hierarchy that can be used to refine settings in a consistent manner. Finally we looked at how variables can be used to exploit the commonality across settings to further reduce the points of configuration to a bare minimum. ■

Acknowledgements

Thanks as always goes to the *Overload* advisors for watching my back.

References

- [Oldwood14] 'Branching Strategies', Chris Oldwood, *Overload* 121
- [KB] <http://support2.microsoft.com/kb/821761>
- [SlowCheetah] <https://www.nuget.org/packages/SlowCheetah/>