overload 122
AUGUST 2014 £3

TDD Discussions and Disagreements
Summarising the differing standpoints on the benefits and pitfalls of TDD

Musings on Python - by a C++ Developer
Python and C++ are very different languages; we see how

KISSing SOLID Goodbye
Distilling SOLID down to two easy to remember principles

Activatable Objects
Using locks will slow down threaded code, an Activatable Object can reduce blocking

Does Test-Driven Development Harm Clarity?
Is clarity the key thing to aim for when writing software and does TDD harm clarity?

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 122

August 2014

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 123 should be submitted by 1st September 2014 and those for Overload 124 by 1st November 2014.

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

Shop 'til you Drop

Deciding which tools to acquire can be a difficult decision. Frances Buontempo muses on a failed shopping trip.

Having had a birthday recently I am now the proud owner of some tokens for a high street 'fashion' store. I did try very hard to spend them – I spent something like an hour wandering round a branch of the shop in question looking at almost everything. Possibly longer. I did see quite a nice pair of black trousers, but they matched the ones I was wearing at the time and having two identical pairs seemed a bit pointless. I suspect this drew attention to me, since at least three people who worked there came up to me and asked if I wanted help. I was going to ask if they'd swap the vouchers for cash, but didn't want to push my luck. This time consuming activity might be the best excuse I've got for not writing an editorial once more.

Shopping isn't just restricted to buying, or failing to buy, clothes, shoes or even socks. Some of us may be lucky enough to choose the tool chain at work. If you use a compiled language, which compiler will you use? This, of course, may be dictated by the platform the code will eventually run on, and may not be in your control. You may need it to run on more than one platform, so might have to end up using the lowest common denominator of an unfortunate combination, such as Microsoft's VC6 and the latest gcc. Such a combination may be deeply unpleasant, and might drive a hardware upgrade so you could avoid the older compiler, and hence more shopping. You may be writing web pages, and just tell your customers "It doesn't work on your browser, use a different one" if this happens to be the case.

What tool chain would you have in place, if you had free control? Would you dictate IDEs or editors to the rest of your team, or allow a consensus to evolve or let anarchy reign and each person individually do their own thing? Would you use Excel as a bug tracking system, write your own, use an open source application, or pay for something? Or cover your monitor in post-it notes? I have been in the unfortunate position of being asked to use spreadsheets to track bugs at work a couple of times, and adding screenshots of various GUI bugs was less than perfect. Trying to use a data filter to figure out who was doing what and what was going to happen when was also a tad difficult. How do you decide which language to use? This can be based on expertise of the team, though if everyone knows FORTRAN or Cobol you are restricting potential future new hires. Going to the other extreme and constantly rewriting your software in the latest trendy language will cause a different set of problems. How many people actually know them? Can you afford the time to train new people? You may give the impression the code will continue to be rewritten each time a new language comes on the block. Perhaps you could go even further and write your own in-house language, though it may be worth considering how highly transferrable skills tend to be rated in many quarters. Of course, DSLs have a place and the ability to pick up new skills easily and deal with whatever gets thrown at you is at least as important as having experience of specific libraries, frameworks and even languages.

Do you choose to enforce coding standards? Might these include doxygen style comments, even if the parameters have no description and in general it's only half done thereby causing 1,500 warnings hiding several import ones? Turning off these specific warnings is always possible, though it could be argued that this defeats the object of demanding the doxygen style comments in the first place. You could always go the other way and entirely ban comments. If you have coding standards do you enforce them by paying for software to help with code reviews, or do you let the troops just use pen and paper or pair programming? Perhaps you could settle for some pre-commit checks in your set up. Or use a policing system, like FxCop [FxCop] after the fact (i.e. a commit) and just leave any offenders to buy donuts in the case of a blunder. Either way, it involves shopping. Or, worse, talking to each other. Do you even have coding standards? You could simply say follow the Good Book [Sutter and Alexandrescu] or follow what Google do [Google]. There are many possible approaches.

Even having decided which language to use, you still need to decide which compiler or interpreter, which build tool, how you will support your applications, how you will deploy them, which platforms you will support. Some of these decisions may be out of your hands. How many places are still running Windows XP due to the perceived risk of upgrading the operating system? Some businesses have only recently moved to Windows 7, and there have been rumours that the next Visual Studio '14' may not work on Windows 7, though certainly it is suggested that you only run the community technology preview on a virtual machine rather than side by side [VS]. Does an OS service pack force a compiler upgrade [SO], and if so are you comfortable changing things?

How do you decide between other supporting tools, like libraries, frameworks and so on? Do you find something open source which hasn't had a bug report in years? This either means it is completely stable and just works, or no-one uses it, possibly with good reason. As people who code, we may tend to have a tendency to assume all software has bugs so be suspicious of something that hasn't changed for years. Nonetheless, typing 'ls' at a prompt probably wouldn't even cause a moment's pause for thought. Perhaps bug free software is possible, but just unlikely. How do you choose continuous integration software? If you would rather avoid manipulating xml directly that will close down at least one choice (naming no names). Personal taste and previous experience will tend to have an influence in these situations. If you are selecting open source middleware, seeing the number of forks for ZMQs .Net port may be off-putting. Nuget [Nuget] currently finds 25 packages and github is showing 85 forks [netmq]. Some companies stick with tooling from one place, maybe

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.

Microsoft or Atlassian and so on. Others may stick with free software, or even insist on using open source software. These decisions can be based on previous choices, making it difficult to swap tooling. Alternatively, a perception that one has better support or quicker bug fixes than the other may hold sway. Ideally, it would be nice to be able to swap from one tool chain and set of libraries to another easily, or at least be able to upgrade compilers without several months of code changes.

Stepping back, observe that choice of language is frequently claimed to affect thought [Language]. For example if a number system just has 1, 2 and many this might appear to affect what you are capable of thinking about. How many colours are there in a rainbow? This varies from culture to culture too. Do choices about programming languages or even tool chain affect your programming behaviour and ultimately the code your write? If you think something like Resharper [Resharper] will show you code that can be improved do you just tap it out and just accept the recommendation afterwards? It can be too easy to do this, without thinking about the suggestions and their consequences. Clearly some may be good and some not. This is, of course, configurable, which requires further choices. Some setups are easier to debug than others, which can mean it becomes tempting to write more code and just step through in a debugger when it doesn't work. If something needs deploying to an embedded system, or controls a spaceship then just running it to see what happens might not be so easy. The context in which you are experiencing your development process is shaped by your language choices, tool-chain and hardware.

More generally, your context affects your perceptions. Kant claimed that we never have direct experience of the 'noumenal' world, but rather experience 'phenomena' as conveyed by our senses [Kant]. This differs slightly from Plato's cave [Plato], where prisoners in a cave see shadows on a wall rather than directly experience what is real, in that it does not presuppose a prototypical universe where there is one true Form for everything e.g. one true sort algorithm. Rather, for Kant what we experience is through the filter of our senses and, if you will allow me, our preconceptions. We see events happening in space and time and so may conclude one event, which always comes first, causes another. The theory of relativity may seem to upset this view somewhat, though is not incompatible with the idea that our perspective or standpoint influences what we perceive. As Nelson probably never said "I see no ships" [Nelson]. Or was that "bugs", or "shops"? It just depends on your standpoint. It is possible to trace the history of ideas from Kant to Einstein, but this would be something of a digression.

Having got your tool chain in place, how will your development proceed? Is there 'One, true way' or just what currently works, given the targeted hardware, teams competency, current trend, legal requirements and so on? Does it matter if some team members write tests after they write the code, rather than before? Or perhaps having any form of tests at all seems like pure luxury. How will you deploy any software you write? If you have a continuous delivery setup then a failing test or a broken build may seem less dramatic than a continuous integration set up with a top rule: don't check-in on a broken build [Thoughtworks]. Furthermore, broken tests overnight might not work for a distributed team. Do you have a prod environment, or even a testing environment, or can you just release new features straight to you customers and fix things as you go? Again, the context will matter. What works for free mobile phones applications may not be appropriate for air traffic control software.

Even having decide to buy or acquire libraries, tool chains, compilers, hardware or more generally other digital items such as music or films, will they end up cluttering up your house or work space? Many things are running on virtual machines or moving off to the cloud. There is a noticeable trend towards having e-books, rather than dead tree versions and further, people will now pay for entertainment to be streamed from the internet rather than own a hard copy. What will happen when the lights go off? Clearly this is a topic for another day. For now, let me conclude by saying I am now the proud owner of far too many socks, the reasons for which I will not bore you with, but I still have several high street store vouchers left, and probably need to buy a new chest of drawers to accommodate the new socks. Unless I find a way to stream socks. Once you attempt to shop, you have started on a slippery slope, I assure you. I will try to avoid attempting to buy anything before the next issue of *Overload* and hope normal service will finally be resumed by then.

References

[FxCop] http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx

[Google] https://code.google.com/p/google-styleguide/

[Kant] *Critique of Pure Reason*, 1781

[Language] http://en.wikipedia.org/wiki/Language_and_thought

[Nelson] http://en.wikiquote.org/wiki/Talk:Horatio_Nelson

[netmq] https://github.com/zeromq/netmq

[Nuget] https://www.nuget.org/packages?q=zeromq

[Plato] *The Republic* (or see https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Allegory_of_the_cave.html)

[Resharper] http://www.jetbrains.com/resharper/

[SO] http://stackoverflow.com/questions/16782409/visual-studio-2012-and-program-compatibility-assistant

[Sutter and Alexandrescu] *C++ Coding Standards: 101 Rules, Guidelines and Best Practises*, Herb Sutter and Andrei Alexandrescu, Addison Wesley, 2004

[Thoughtworks] http://www.thoughtworks.com/continuous-integration

[VS] http://www.visualstudio.com/en-us/downloads/visual-studio-14-ctp-vs.aspx

Does Test-Driven Development Harm Clarity?

Is clarity the key thing to aim for when writing software? Andy Balaam considers if TDD harms clarity and if this matters.

In a recent keynote at RailsConf called 'Writing Software'[1], David Heinemeier Hansson [Hansson] argues that test-driven development (TDD) can harm the clarity of our code, and that clarity is the key thing we should be aiming for when writing software.

It's an excellent talk, and I would highly recommend watching it, especially if you are convinced (like me) that TDD is a good thing.

I was inspired by watching the video. Clarity certainly is extremely important, and the name he coins, Software Writer, sits better with me than Software Engineer or Software Developer. I have often felt Programmer was the best name for what I am, but maybe I will adopt Software Writer.

The real goal

I would argue that clarity is not our goal in writing software. I think our goal is:

> Working, modifiable software

Clarity helps us feel confident that our software works because we can read the code and understand what it does.

Clarity helps us modify our software because we can understand what needs to be changed and are less likely to make mistakes when we change it.

A good set of full-system-level tests helps us feel confident that our software works because they prove it works in certain well-defined scenarios. A good set of component-level and unit tests gives us confidence that various parts work, but as David points out, confidence in these separate parts does not give us much meaningful confidence that the whole system works.

Good sets of tests at all levels help us modify our software because we are free to refactor (or re-draft as David puts it). Unit and component tests give us confidence that structural changes we are making don't modify the external behaviour of the part we are re-structuring. Once we have made enabling structural changes, the changes we make that actually modify the system's behaviour are smaller and easier to deal with. The tests that break or must be written when we modify the system's behaviour help us understand and explain the behaviour changes we are making.

So both clarity and tests at all levels can contribute to our goal of writing working, modifiable software.

But David wasn't talking about tests – he was talking about TDD – driving the design of software by writing tests.

Andy Balaam is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

How TDD encourages clarity

Before I talk about how we should accept some of what David is saying, let's first remind ourselves of some counter-points. TDD is explicitly intended to improve our code.

I agree with David when he defines good code as clear code, so how does TDD encourage clarity?

TDD encourages us to break code into smaller parts, with names. Smaller, named classes and functions are clearer to me than large blocks of code containing sub-blocks that do specific jobs but are not named. When I write in a TDD style I find it encourages me to break code into smaller parts.

TDD encourages us to write code that works at a single level of abstraction. Code that mixes different levels is less clear than code at a single level. I find that using TDD helps me resist the temptation to mix levels because it encourages me to compose two pieces that deal separately with each level, rather than linking them together.

It is vital to point out here that TDD doesn't push you towards clarity unless you already wanted to go there. I have seen TDD code that is unclear, stuffed full of boilerplate, formed by copy-paste repetition, and is generally awful. As a minimal counter-example, TDD doesn't stop you using terrible names for things.

But, when you care about clarity, and have an eye for it, I feel TDD can help you achieve it.

How TDD hurts clarity

David's argument against TDD is that it makes us write code that is less clear. His main argument, as I understand it, is:

TDD forces us to use unnecessary layers of abstraction. Because we must never depend on 'the world', TDD forces us to inject dependencies at every level. This makes our code more complex and less clear.

At its core, we must acknowledge that this argument is true. Where TDD causes us to inject dependencies that we otherwise would not inject, we are making our code more complex.

However, there are elements of a straw man here too. Whenever I can, I allow TDD to drive me towards systems with fewer dependencies, not injected dependencies. When I see a system with fewer dependencies, I almost always find it clearer.

Test against the real database?

David frequently repeats his example of testing without hitting the database. He points out that allowing this increases complexity, and that the resulting tests do not have anything like as much value as tests that do use the database.

This hurts, because I think his point is highly valid. I have seen lots of bugs, throughout systems (not just in code close to the database) that came from

1. Caution – contains swearing

wrong assumptions about how the database would behave. Testing a wide range of functionality against the real database seems to be the only answer to this problem. Even testing against a real system that is faster (e.g. an in-memory database) will not help your discover all of these bugs because the faster database will have different behaviour from the production one.

On the other hand, tests that run against a real database will be too slow to run frequently during development, slowing everything down and reducing the positive effects of TDD.

I don't know what the answer is, but part of it has got to be to write tests at component level, testing all the behaviour that is driven by database behaviour, but not requiring huge amounts of other systems to be spun up (e.g. the web server, LDAP, other HTTP endpoints) and run these against the real database as often as possible. If they only take about 5 minutes maybe it's reasonable to ask developers to run them before they commit code.

But my gut tells me that running tests at this level should not absolve us from abstracting our code from the production database. It just feels Right to write code that works with different storage back ends. We are very likely to have to change the specific database we use several times in the history of our code, and we may well need to change the paradigm e.g. NoSQL → SQL.

In a component that is based on the database, I think you should unit test the logic in the standard TDD way, unit test the database code (e.g. code that generates SQL statements) against a fast fake database, AND comprehensively test the behaviour of the component as a whole against a real database. I admit this looks like a lot of tests, but if you avoid "dumb" unit tests that e.g. check getters and setters, I think these 3 levels have 3 different purposes, and all have value.[2]

Injecting the database as a dependency gives us the advantage of our code having two consumers, which is one of the strongest arguments put forward by proponents of TDD that it gives us better code. All programmers know that there are only three numbers: 0, 1 and more. By having 'more' users of our database code, we (hopefully) end up with code that speaks at a single level e.g. there are no SQL statements peppered around code which has no business talking direct to the database.

Clarity of tests

In order for tests to drive good APIs in our production code, and for them to serve as documentation, they must be clear. I see a lot of test code that is full of repetition and long methods, and for me this makes it much less useful.

If our tests are complex and poorly factored, they won't drive good practice in our production code. If we view unclear tests as a smell, the fixes we make will often encourage us to improve the clarity of our production code.

If our tests resemble (or actually are) automatically-generated dumb callers of each method, we will have high coverage and almost value from them. If we try to change code that is tested in this way, we will be thwarted at every turn.

If, on the other hand, we write tests that are clear and simple expressions of behaviour we expect, we will find them easy to understand and maintain, they will drive clear code in production, and sometimes we may realise we are writing them at a higher level than unit tests. When this happens, we should 'float freely' with David and embrace that. They are testing more. That is good.

2. Writing the logic TDD encourages smaller units of logic, and gives confidence that it is correct. Testing the database code against a fake database gives confidence that our syntax is right, and testing the whole component against the real database gives us confidence that our behaviour really works.

Higher-level tests

David (with the support of Jim Coplien [Coplien]) encourages us to test at a coarser grain than the unit. I strongly agree that we need more emphasis on testing at the component and system levels, and sometimes less unit testing, since we don't want write two tests that test the same thing.

However, there are some problems with larger tests.

First, larger tests make it difficult to identify what caused a problem. When a good unit test fails, the problem we have introduced (or the legitimate change in behaviour) is obvious. When a component or system test fails, often all we know is that something is wrong, and the dreaded debugging process must begin. In my experience, this is not just a myth. One of the pieces of code that I am most proud of in my career was a testing framework allowed you to write concise and clear tests of a horrible tangled mess of a component. The tests ran fairly quickly, with no external dependencies being used, but if one of them failed your heart sank.

Second, large tests can be fragile. Sometimes they fail because the database or network is down. Sometimes they fail because your threads did stuff in an unexpected order. Threading is a really good example: when I want to debug a weird threading issue I want to write an unthreaded test that throws stuff at a component in a well-defined but unusual order. When I can get that test to fail I know I've found a real problem, and I can fix it. If the code is not already unit tested (with dependencies being injected, sometimes) then writing that test can be really difficult.

TDD makes our code testable, and while testability (like clarity) is not an end in itself, it can be darn useful.

Conclusion

Good things

- Clarity is good because it supports working, modifiable code.
- Tests are good because they support working, modifiable code.
- Testability is good because it supports tests, especially debugging tests.
- TDD is good when it supports clarity, testability and tests.

  TDD is bad when it hurts clarity.

If you throw out TDD, try not to throw out tests or testability. You will regret it.

What to do

- Write tests at the right level. Don't worry if the clearest level is not the unit level.
- Use tests to improve clarity.
- If your tests are unclear, there is something wrong. Fix it.

Steps to success

1. Get addicted to TDD.
2. Wean yourself off TDD and start to look at the minimal set of tests you can write to feel that sweet, sweet drug of confidence.
3. Do not skip step 1. ■

References

[Coplien] Jim Coplien – https://sites.google.com/a/gertrudandcope.com/www/jimcoplien

[Hansson] 'Writing Software' – keynote at RailsConf 2014. http://www.confreaks.com/videos/3315-railsconf-keynote

First published

This article was first published at: http://www.artificialworlds.net/blog/2014/05/09/does-test-driven-development-harm-clarity/

Musings on Python – by a C++ Developer

Python and C++ are very different languages. Sergey Ignatchenko walks through things in Python that can confuse a C++ programmer.

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

During my vacation in Watership Down warren, I met a fellow rabbit developer who's got some experience with developing in Python, after spending quite a while worshiping C++. Below is my humble attempt to express his feelings about Python in a more or less literary form. It doesn't aim to be a comprehensive analysis of the subject, but rather a set of things the guy himself has run into (YMMV).

The good

Ad-hoc typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

~ James Whitcomb Riley

One good thing about Python is its ad-hoc typing system (which is known in Python world as 'duck typing'). I've observed that it does speed up initial development quite a bit.

In any language, it is common to write something specific, and then to generalize it. In C++, it is doable, but difficulties related to generalization are quite substantial. In fact, you can either generalize via making a function virtual (relying on common base class), or making it a template. I won't discuss the advantages and disadvantages of each of the approaches here, but in any case you're expected to spend some time performing this generalization. If you prefer (or need, as it routinely happens with containers) the C++ template route, the necessary textual changes are massive (even when they're mostly mechanical), and debugging of the generalized program requires quite an effort (to put it mildly); in fact, it is such a big effort that many developers won't do it at all, and those who will, will think twice before going the template way. If going down the virtualization route, changes are not that massive (though are still substantial), but you're introducing a common base class, which is essentially a dependency which often leads to strange problems down the road (like multiple inheritance with virtual base classes etc.); while these problems can always be solved, solving them takes time, and this is my point here.

In contrast, in Python you don't need to do anything special to make your code generic. In fact, each and every piece of code becomes as generic as possible at the very moment it is written. For example, with code such as

```
def f(x,y):
  return x*y
```

you don't care about types of `x` and `y`, as long as they support multiplication. While in C++ it can be written as a template quite easily, the amount of textual changes necessary when converting a function `f` from `int f(int x, int y)` to its template counterpart will be quite substantial (and if we consider more complicated functions, the complexity will rise further).

It should be noted that in Python you can (and should) use classes more or less like in C++. However, in Python you have an option not to do so (in trivial cases) – and this flexibility often saves quite a lot of development time.

Overall, it is not about 'what you can do' in Python and in C++ (whatever you can do in Python, you can do in C++), but more of 'what you can do *faster*'. This matters, because the more time you need to spend on technicalities related to your programming language, the less time you have left for the task in hand; in a sense, it is similar to an argument between assembler and C developers 40 or so years ago (I don't want to say that C++ will follow the fate of assembler, at least not yet).

A word for those who have arguments about advantages of strong typing – I will tell a bit about these advantages too, so please keep reading until you reach 'The bad' section :-).

Garbage collection with RAII support

One thing which I like about Python is that while it is garbage collected, it has explicit support for Resource Allocation Is Initialization (RAII). Garbage collection IMHO does speed development up (though contrary to common belief you still need to be careful to avoid memory leaks [Ignatchenko12]). On the other hand, some garbage-collected programming languages (notably Java, at least at the time I last saw it) have a problem that freeing resources becomes really cumbersome and error-prone.

Let's consider the C++ class `File`, which opens a file in constructor, and closes it in the destructor. It means that even if there was an exception, then when my object of class `File` goes out of scope, the file is closed and the resource is freed. Good, but we don't have garbage collection in C++.

The same class `File` in Java won't be able to have a real destructor (there are no destructors in Java). In Java, to guarantee that you always close all the relevant files, you have three and a half options. Option 1 is to find all places where you have instantiated `File`, enclose them in `try-finally` blocks, and close file manually in each finally block. Horrible. Option 1a is a variation of Option 1, based on the 'execute around' pattern. Basically, you're declaring a function wrapper which allocates resource, then calls

A philosphical approach

From a philosophical perspective, one can think of classical pre-template C++ (which relies on virtual functions) as having an 'it matters who you are' paradigm, while C++ templates and Python in general, relying on a quite different 'it matters what you can do' approach.

'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

One good thing about Python is its ad-hoc typing system (which is known in Python world as 'duck typing')

whatever function you need via an interface (doing it within **try-finally** block), and then frees allocated resource. As long as you can make sure that class **File** is used only within such a wrapper – it is not 'horrible' anymore, just 'very cumbersome'.

Option 2 looks a bit better on the surface – in Java you can define a **finalize()** function, which looks like 'almost a destructor'. Unfortunately, this 'almost' kills the whole idea: due to the very nature of garbage collection, Java cannot guarantee when exactly **finalize()** will be called; it means all kinds of trouble, including the program passing all the tests but failing in production. For example, you have **file.close()** in **finalize()**, and then re-open the same file somewhere down the road. It just so happens during the tests that **finalize()** is called before re-opening, and all tests pass, but in production **finalize()** is sometimes called later than re-opening the file, and therefore re-opening the file fails (to make things worse, it will invariably fail intermittently and at the very worst time to make debugging even more complicated). Overall, there is pretty much a consensus that **finalize()** should not be used for a generic resource cleanup. Ouch. In fact, this 'how to guarantee that resources are always freed when they're not necessary anymore' problem has always been my biggest complaint about Java.

Option 3 (thanks to Roger Orr for pointing it out): if you're lucky enough to run Java 7, you may implement the **java.lang.AutoCloseable** interface and then write code such as:

```
try (MyClass x = new MyClass(/*...*/))
  //'try-with-resources' statement
  {
    x.method("this might throw");
  } // x.close() is called in any case
```

Not bad – and we can say that Java 7 does support both RAII and garbage collection.

In a manner which is quite similar to Java 7, Python provides a neat way of expressing RAII. In Python, you can declare your class with special functions like **__entry__()** and **__exit__()** (and many of Python's own objects such as the file object, implement them too). Then, you can write something like:

```
with open("myfile.txt","r") as f:
  #work with f
  #more work with f
#at this point, f.__exit__() will be called
```

For me, it solves all my resource allocation concerns (and Python has garbage collection too). Oh, and while we're on the subject of garbage collection and finalizers in Python – a word of advice: never declare Python finalizers (**__del__() functions**) unless you really know what it means (Python **__del__()** causes very different behavior from the Java **finalize()**).

Usable Lambdas

I didn't think that I would ever be able to write anything good about lambda functions for any practical purpose, but here it is: lambda functions in Python are surprisingly readable and useful. They have a very simple syntax, and they're limited, but they're very readable. Compare:

Plain C++11:

```
1 sort( myVector.begin(), myVector.end(),
2     []( const MyClass& a, const MyClass& b )
3           { return a.x < b.x; }
4     );
```

Python:

```
sort( myList, key=lambda a: a.x )
```

I have never been a fan of one-line expressions just for the sake of being one-line, but the Python version is not only a one-line, it is obvious from the very first glance, while the C++ version requires quite a lot of time to parse when reading.

It should be noted that the point of the example above is not about **begin()** and **end()** in C++ line 1 or comparison in C++ line 3; as we're discussing lambdas, the difference under consideration is about C++ line 2 (and inevitable curly brackets from line 3).

As it was pointed out by Jens Auer in accu-general, the **boost::lambda** library (BLL) allows much shorter way of writing it.

boost::lambda library:

```
sort( myVector.begin(), myVector.end(),
      _1.x < _2.x );
```

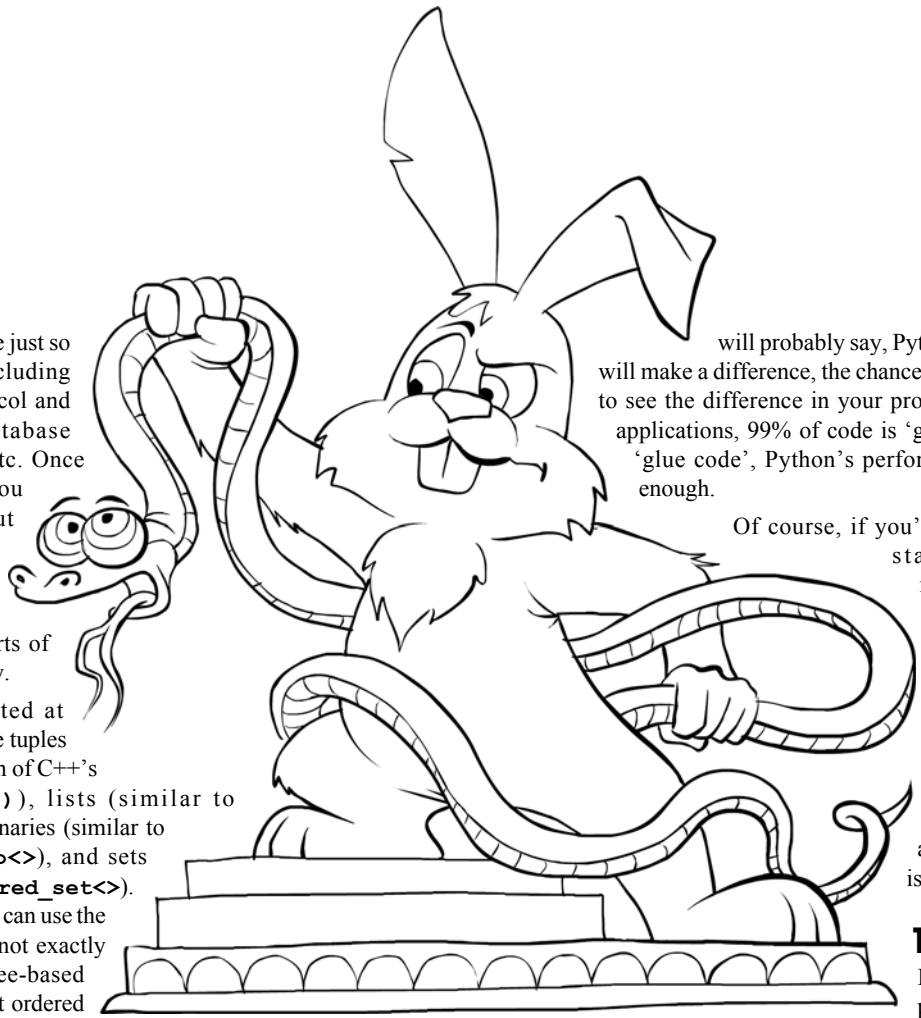
Still, I'd argue that while certainly shorter than plain C++, it is not exactly readable compared to Python version – first, numbered parameters are definitely worse than Python's named ones, and second, unless you know about BLL (and most developers don't as of now), such code becomes extremely confusing. Honestly, for a C++ project with more developers than just me I don't know which way I'd use – cumbersome plain C++ or a much shorter BLL with a comment for each such lambda saying `/* boost lambda */`, so an unaware reader knows how to Google it (with Python syntax, it is quite self-documented).

NB: Obviously, it is possible to write a non-lambda wrapper for a specific task of sorting a vector, but this won't get us any closer to having usable and readable lambdas, which this section is about. While it is perfectly possible to write code without lambdas at all, usable and readable lambdas do simplify development (not by much, but every bit counts), and having to write a non-lambda wrapper for each scenario where lambdas are useful, defeats the whole purpose of lambdas.

NB #2: there is a caveat related to lambdas in Python, please see 'The ugly'section

Standard library

The standard library in Python is huge and is very-well organized. It includes 90% of the things one may want from an application-level library; overall, having pretty much everything included into the standard Python library is often referred to with the 'Python. Batteries Included' phrase. With all due respect to the enormous efforts of **boost::** folks, matching functionality with the Python library isn't going to happen (and probably



is not aimed for) – there are just so many things in there, including cryptography, wide protocol and file format support, database interface libraries, etc. etc. Once again, it is not about 'you cannot do it in C++', but about 'how long it will take to do'.

Let us now consider some of the most important parts of the Python standard library.

Collections are supported at language level, and include tuples (somewhat similar to return of C++'s `std::make_tuple()`), lists (similar to `std::vector<>`), dictionaries (similar to `std::unordered_map<>`), and sets (similar to `std::unordered_set<>`). Notably missing (well, you can use the bintrees package but it is not exactly 'standard library') are tree-based maps/sets which allow fast ordered iterations over large datasets.

Assertions are first-class citizens and are recognized at language level, which is a good thing. They also allow specification of the message to report in case of assertion failure – if you feel like it.

Furthermore, the packages **profile** and **cProfile** provide a rather convenient built-in means of profiling of your program.

Regular expressions in Python are very efficient and are aided by 'raw string' literals. 'Raw string' literals are useful because the escaping rules for '\' in default C++ strings and default Python strings tend to make regexps quite cumbersome and poorly readable. In Python (as well as in C++11), there is an elegant way around it: whenever you prefix string literal with 'r' (such as `r"\([0-9]*\)"`), the escaping rules for backslashes will be different, which allows you to write regular expressions in quite a natural way.

Built-in unit testing framework

Having a built-in unit testing framework is a good thing in any language. For weakly typed languages such as Python, integrated unit testing (especially automated regression testing) becomes an absolute must. Fortunately, Python has support for it too (for details, see the unittest package).

Performance

For those who want to write Python off due to performance issues, I have a word of advice: don't rush. While it is perfectly possible to find an application where Python's performance (or as some C/C++ developers will probably say, Python's lack of performance) will make a difference, the chances are that you won't be able to see the difference in your program. In 99% of business applications, 99% of code is 'glue code', and for 99% of 'glue code', Python's performance will be more than enough.

Of course, if you're developing some non-standard computation-intensive stuff such as a video decoder, you will probably be out of luck. However, if you will run into situation where you need to write certain parts in C/C++, Python will provide a way to call your DLLs/.so's (the appropriate Python package is **ctypes**).

The bad

If you've read until this point, you may think that I'm a Python missionary on a quest to convert as many people as possible. Don't worry, I will mention bad sides of Python too.

Ad-hoc typing

While ad-hoc typing does have its advantages (as was discussed above), it has a big problem too, and this is a lack of scalability. Let me elaborate a bit. If you're creating an ad-hoc object such as `(1,2,3)` (similar to `std::make_tuple(1,2,3)`), it works very well for those cases where you need just to pass it from one point to another point, without going into hassle of declaring things. However, ad-hoc typing doesn't really scale – as soon as you're using the same ad-hoc type in 10 places, and it does need to be the same in all 10 places, code maintenance becomes a nightmare.

Many Python developers seem to realize the problem, and several workarounds have been created. In particular, I've found `namedtuple` package to be quite useful (in a sense, it is a close cousin of C++ struct):

C++:

```
struct X { int i; string s; }
```

Python:

```
X = namedtuple('X', ['i','s'])
```

On the other hand, more recent development of the Python abstract base classes (package `abc`) feels like a *contradictio in adjecto*: it is like writing in Python using C++ paradigms, which defeats the advantages of one while not providing benefits of the other one.

An ideal IMHO would be an environment where I could write ad-hoc types without declaring them (while they are still small), and then, whenever I feel that they became too large to be ad-hoc, to change them (just by adding

declarations where necessary, and not changing the actual code(!)) to strict typing. I have some ideas in this regard, though it is a bit too early to describe them.

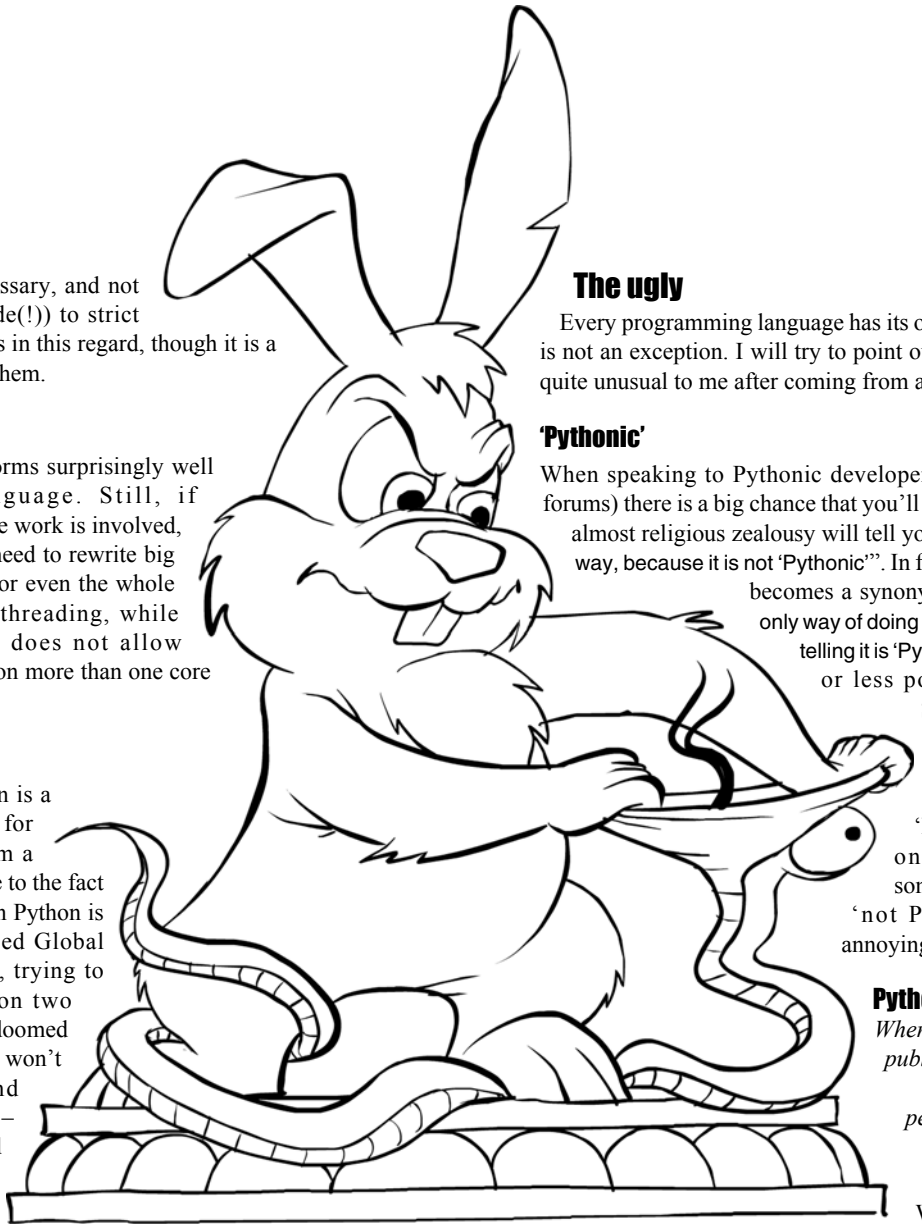
Performance

In general, Python performs surprisingly well for a scripting language. Still, if computationally intensive work is involved, one may end up with a need to rewrite big chunks of the program (or even the whole program). Also, multithreading, while technically possible, does not allow performing calculations on more than one core (see below).

Multithreading

Multithreading in Python is a joke; well, it is at least for those of us coming from a non-Pythonic world. Due to the fact that all data processing in Python is made under the so-called Global Interpreter Lock (GIL), trying to perform calculations on two cores in two threads is doomed (well, it will work, but it won't work any faster – and probably a tad slower – than single-threaded code). It limits the usage of multithreading to the cases when a thread is blocked due to I/O wait. Technically speaking, while the default Python distribution uses cPython which does have a GIL, GIL is not a restriction of Python as such, so you may be able to get away with using something like Jython or IronPython (I didn't try it myself though).

If you need to perform computational-intensive calculations in parallel while using the default cPython, there is still an option to spawn another process (which will have its own GIL so you will be able to calculate things in parallel). An appropriate Python package is `multiprocessing`, and it is quite convenient (in fact, it has an interface which is very similar to that of the `threading` package). However, you should keep in mind that under the hood it relies on marshaling/unmarshaling of all the parameters passed to the working process (and of all the values returned back), so if your parameters and/or return values are large you can easily get quite a performance hit. Which in turn can be overcome (at least in theory) by using shared memory, but this has a caveat too – shared memory cannot contain anything but very simple data. Overall, you can end up with a scenario where you'll essentially be forced to write the computational code in C/C++.



The ugly

Every programming language has its own peculiarities, and Python is not an exception. I will try to point out a few items which looked quite unusual to me after coming from a mainly C++ world.

'Pythonic'

When speaking to Pythonic developers (whether in person or in forums) there is a big chance that you'll run into somebody who with almost religious zealousy will tell you, "You shouldn't write it this way, because it is not 'Pythonic'". In fact, way too often 'Pythonic' becomes a synonym to "I believe that it is the only way of doing it; I cannot explain why, so I'm telling it is 'Pythonic'". Fortunately, in more or less populated forums (such as StackOverflow), usually there are enough people who make sure that whatever is called 'Pythonic' makes sense. Still, ongoing arguments about something being 'Pythonic' (or 'not Pythonic') can be rather annoying.

Python 3

Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs.

~ Jonathan Swift, circa 1726

With all due respect to Guido van Rossum, I strongly believe that the approach taken with Python 3 is a huge mortgage-crisis-sized mistake. What has happened with Python 3 is that developers were told that Python 3 will be incompatible with Python 2. No smooth migration, no gradual deprecation, just 'all or nothing# migration path (well, with a helper '2to3' tool which 'sorta' converts Python 2 source code to Python 3). Moreover, certain constructs which are allowed in both Python 2 and Python 3, have a subtly different meaning in Python 3 (one such example is `dict.items()`). This has lead to enormous confusion and significant reluctance to move towards Python 3 (in fact, the adoption rate of Python 3 was reported to be as low as 2% 5 years after it has been introduced [Hiltmon14]).

Without going into Blefuscian-Lilliputian discussions of "What is better – to suffer from imperfections of Python 2 in Python 3 or to have better but incompatible Python 3?", I'll try to summarize the current situation:

- the official position of Guido and the Python core team is that all new development SHOULD be done in Python 3

- however, if you have Python as a part of a 3rd-party application (which tend to use Python 2, as they need to support older scripts written in Python 2) – you're pretty much doomed to Python 2
- moreover, as there is a '2to3' tool which 'sorta' converts your code from Python 2 to Python 3 (and there is no tool which converts code back – from Python 3 to Python 2), one way to have code which supports both Python 2 and Python 3, is to keep your codebase in Python 2. Alternatively, you may write in a dialect known as Polyglot (which works in both Python 2 and Python 3), though it has been argued that Polyglot is the worst language out of Python 2, Python 3, and Polyglot [Faassen14].

Phew, this is ugly indeed. To make it even uglier, there were even suggestions to stop supporting Python 2 to force migration to Python 3 [Faassen2014]. One thing I wonder about is how those people would stop a huge Python 2 community from creating an unofficial fork with ongoing support for Python 2 (it is open source, after all)?

Semantics of whitespace

Python is a quite unusual language in that it relies on whitespace to provide semantic data (or in other words – changing whitespace can change semantics of the program in Python). For example,

```
if a < b:
    x = 1
    y = 2
```

and

```
if a < b:
    x = 1
y = 2
```

are two different programs producing different results.

The Python approach has both advantages and disadvantages. On the positive side, it enforces code readability. On the negative side, it has several issues (in practice, rather minor if you are careful):

- you need to be careful when switching between windows using Alt-Tab: there is a substantial chance that an accidentally added tab can go unnoticed but will break your code, ouch
- you need to make sure that 'diff' tool which you're using with your source control system, does not ignore whitespace
- instead of an endless C/C++/C#/Java debate of "Where is the right way to put curly brackets?" it leads to another endless debate of "What is the right thing to use – tabs or spaces?" As it doesn't matter any more than which end of the egg is broken, the only thing which matters is consistency. And as such, I prefer to stick to the (widely accepted) recommendation from [PEP8]: use spaces, with 4 spaces per indentation level. Why? Just for the sake of consistency.

Lambdas within loops

With all the good things said about lambdas in Python, there is one thing to keep in mind: if you try to use lambda which captures variables within a loop, it won't work as you might have expected. For details, refer, for example, to [StackOverflow-1]. The best workaround I was able to find is a direct replacement with a function object. Instead of not-working-as-expected:

```
for i in range(10):
    a[i] = lambda x: x+i
        #every a[i] function will use the same i=10
```

you can use, for example, an almost equivalent but working as you (or at least me) would intuitively expect:

```
class MyLambda:
    def __init__(self,i):
        self.i = i
    def __call__(self):
        return x+self.i
#...
for i in range(10):
    a[i] = MyLambda(i)
```

There are other alternatives too, see, for example, [StackOverflow-1] and [StackOverflow-2] for details.

Overall, this is rather annoying, but is not that a big deal when you know about it.

Optimizing performance

Optimizing the performance of a Python program is very different from optimizing a C or C++ counterpart. For a Python program, instead of an 'I can write it myself' approach, one should look for highly optimized (a.k.a. 'written in C') functions from the Python standard library. Just one example: when we've needed to read a multi-megabyte text file accounting for 'universal line endings' (either `\r` or `\n`), the standard Python library did rather a poor job. Rewriting it to byte-by-byte processing (which would help in C/C++) has only made things worse, as more work for Python bytecode is rarely a good thing. However, when we became creative and started reading the file in chunks (each chunk being several kilobytes in size, so it usually contained multiple lines), pre-creating a regular expression pattern

```
eol_pattern = re.compile( r'([^\r\n]*)([\r\n])' )
```

and using

```
line = eol_pattern.match( chunk,
current_pos_within_chunk )
```

on the chunk to extract the next line (with an appropriate handling of double-symbol line endings), we got about a 2x speed improvement over the standard Python library universal line handling (which apparently was pure Python and was not that creative in this regard). The reason for this is quite obvious: the regular expression library is a heavily optimized C code, and when we pushed most of the processing there instead of doing it in Python, we got quite an improvement.

Conclusion

The guy who told me this story, is of the opinion (which I may or may not share) that Python is by far the best language available for writing 'glue' code. Yes, it has its quirks, but for most of the business-level code it is clearly 'good enough', and whenever top performance is necessary, C/C++ code can be integrated rather easily.

From my perspective, I would say that as both C++ and Python are Turing-complete,: you can always implement any practical program in both of them (well, assuming that Church-Turing thesis stands). In practice, of course, there are restrictions such as, 'Will we live long enough to write the program?' (an argument for C++ over asm and for Python over C++) and 'Will we live long enough for the program to execute?' (an argument in the opposite direction). As usual, it is all about choosing right tool for the job. ■

References

[Faassen14] 'The Gravity of Python 2', Martijn Faassen, http://blog.startifact.com/posts/python-2-gravity.html

[Hiltmon14] 'Python: It's a Trap', Hilton Lipschitz, http://hiltmon.com/blog/2014/01/04/python-its-a-trap/

[Ignatchenko12] 'Memory Leaks and Memory Leaks', Sergey Ignatchenko, *Overload* 107, February 2012

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

[PEP8] http://legacy.python.org/dev/peps/pep-0008

[StackOverflow-1] http://stackoverflow.com/questions/1841268/generating-functions-inside-loop-with-lambda-expression-in-python

[StackOverflow-2] http://stackoverflow.com/questions/938429/scope-of-python-lambda-functions-and-their-parameters

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

Activatable Object

Using locks will slow down threaded code. Len Holgate demonstrates how an Activatable Object can reduce the time spent blocked.

An Activatable Object is a variation on the ACTIVE OBJECT PATTERN [Wikipedia-1] in which the object does not run on a thread of its own but instead borrows one of the calling threads to process operations. Activatable Objects can be used to replace objects that can be accessed by multiple-threads and that have state that can only be manipulated by one thread at a time. They play nicely with threads in thread pools and allow for scaling to vast numbers without burdening the process with an excessive number of threads.

Background

I work on high performance networking systems on Windows platforms and these deal with many thousands of concurrent connections each of which is represented by a connection object with complex state. Things such as SSL and asynchronous flow control requirements [ServerFramework] mean that when I/O events occur on a connection it's often necessary to ensure that only one thread is manipulating the connection object's state at a given time. Since multiple I/O events can occur at the same time for a given connection and since 'user code' could also be calling into the connection object to initiate new I/O requests it is often necessary to protect these objects with locks. Unfortunately using locks in this way meant that threads are often blocked by other threads whilst operations are processed on the connection. See Figure 1.

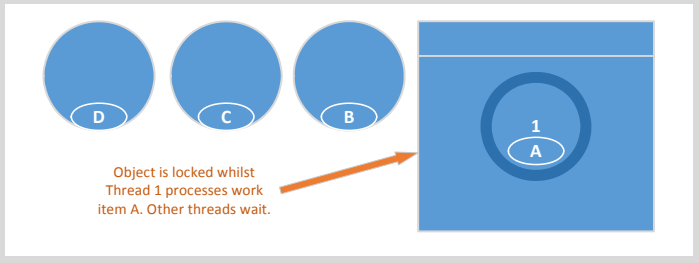


Figure 1

In Figure 1, we have a thread-safe connection object whose internal state is protected by a lock. Given the threads 1, 2, 3 & 4 with the four work items A, B, C & D, the threads are serialised and block until each can process its own operation on the object. This is bad for performance, bad for contention and bad for locality of reference as the thread-safe object can be bounced back and forth between different CPU caches as different threads enter the object to perform work. It also means that a lock is held whilst the threads do their work on the object. Holding a lock like this makes it more complicated to call back into 'user code' as it's easy to cause lock inversions if you hold locks whilst calling into code that you don't own [DrDobbs].

Active Objects, where the object itself contains a thread and where operations are passed to that thread via a work queue, can't help here as we do not wish to use one thread per connection as this does not scale [Kegel]. The preferred method of high performance I/O on Windows is to use I/O Completion Ports and this method can scale to tens of thousands of concurrent connections with but a small number of threads. In this kind of system a small number of threads reside in a thread pool and perform work on the connections as I/O operations complete.

The Activatable Object that I describe in the rest of this article enables efficient command queuing from multiple threads where commands are only ever processed on one thread at a time, minimal thread blocking occurs, no locks are held whilst processing and an object tends to stay on the same thread whilst there's work to be done with it.

Reducing thread blocking

The amount of time that a thread needs to block on an object can be reduced by adding a per-object queue of operations to be performed. If we state that all operations must be placed into this queue before being processed and only one thread can ever own the right to remove and process operations then we have a design whereby a single thread will process an operation and any additional operations that occur whilst the thread is processing will be queued for processing by that same thread.

Obviously the per-object operation queue needs to be thread-safe and so we need to lock before accessing it. We also need a way to indicate that a thread has ownership of the queue for processing. The important thing is that the processing thread does not hold the queue lock whilst it is processing and so threads with operations for this object are only blocked by other threads with operations for this object whilst one of them is holding the lock to add its operation to the queue rather than for the duration of all operations that are ahead of it in the queue.

In general, it should be cheaper to marshal the command and command data into the work queue and onto to the processing thread than it is to move the object to the thread that is issuing the command, both in terms of the enqueue/dequeue process that is required and in terms of locality of reference [Wikipedia-2]. This is generally true for my use cases as the connection objects are usually pretty heavy weight and the commands are generally marshalling pointers to memory buffers. Any marshalling costs are also mitigated by the fact that the alternative thread-safe object would block the calling thread until all previous operations have been completed on the object whereas the Activatable Object only blocks the thread if the work queue is locked by another thread.

In the diagram below three operations for an Activatable Object occur simultaneously and the first thread, thread 1, enters the work queue lock, adds its operation to the queue and sees if any other threads are processing. During this time the other two threads must wait. See Figure 2.

Note that it's possible to optimise the processing of an operation if the work queue is empty and there is currently no thread processing by avoiding the

Len Holgate has been programming for far too long and currently specialises in server-side C++ on Windows platforms. He worked for investment banks in London for several years before switching to working for clients directly from his office in Guildford, Surrey. You can contact him at len.holgate@jetbyte.com

it should be cheaper to marshal the command and command data into the work queue and onto to the processing thread than it is to move the object to the thread that is issuing the command

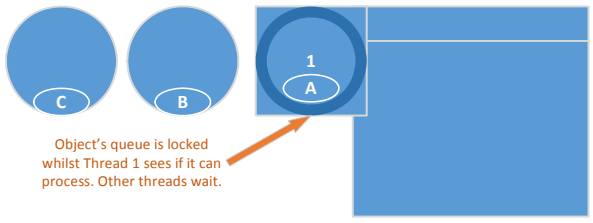


Figure 2

enqueue/dequeue steps and allowing thread 1 to become the processing thread and process the operation directly.

Once thread 1 is processing it releases the lock on the queue and enters the object to process the operation. Thread 2 can then obtain the lock on the queue and enters to add its operation to the queue and see if any other thread is processing. See Figure 3.

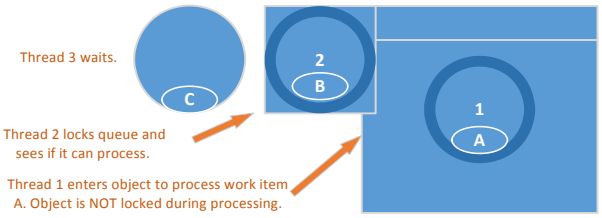


Figure 3

Since thread 1 is processing, thread 2 simply enqueues its operation and returns to do other work. Thread 3 can then enter the lock that protects the queue. See Figure 4.

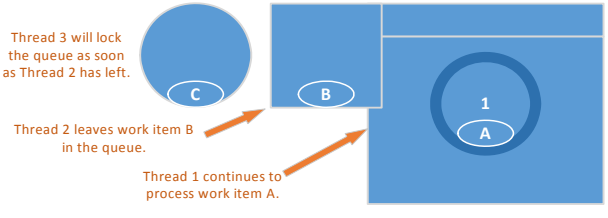


Figure 4

As thread 1 is still processing, thread 3 also simply enqueues its operation and returns. See Figure 5.

When thread 1 has completed processing its operation it needs to give up the right to be the 'processing thread'. To ensure that operations are processed efficiently it must first check the queue for more operations to process and process those before leaving the object to do other work. If

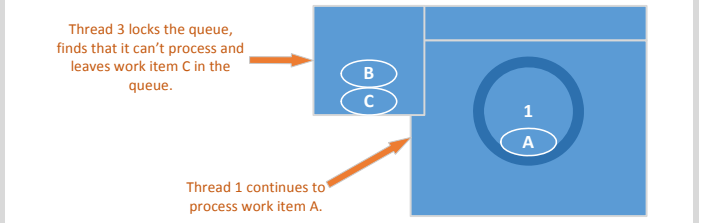


Figure 5

another operation occurs whilst thread 1 is checking the queue then the thread with the operation, in this case thread 4, will block for a time. See Figure 6.

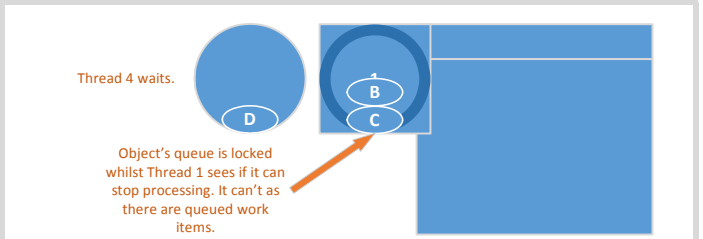


Figure 6

Thread 4 can then enter the queue, check to see if it can process and if it cannot then enqueue its operation for processing by the processing thread. See Figure 7.

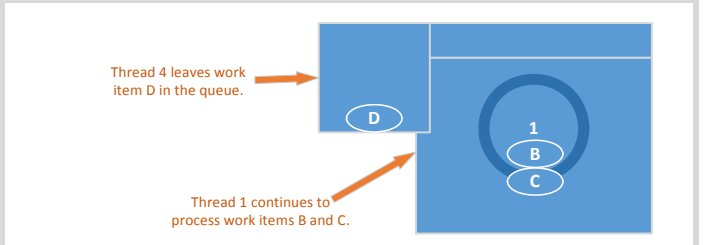


Figure 7

Eventually the processing thread will reach a point where no further operations are queued for the object and it can surrender 'processing mode' and return to do other work. See Figure 8.

The key features of the design used above are as follows:

- Operations can be enqueued quickly and efficiently so that the lock around the object's operation queue is held for the least amount of time possible.
- Only one thread can be 'processing' at any given time.

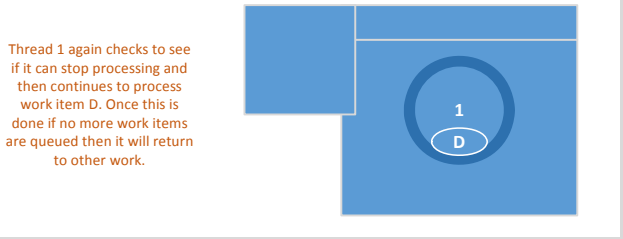


Figure 8

- The processing thread removes all queued operations when it checks the queue.
- Locks are not held whilst processing occurs.

When I was designing this code for integration into my networking system several other requirements emerged during integration:

- The processing thread must be able to enqueue a new operation.
- Some operations require that they are processed synchronously, before the thread that is to enqueue them is released to do other work as the other work may require that the operation on the object has already been processed.

The final requirement for synchronous operations introduces some complication as these operations should block each other but not affect threads with asynchronous operations. This requires a second lock and an event for the blocking threads to wait on. If a processing thread discovers other threads waiting to process synchronous operations it relinquishes processing and allows one of the waiting threads in to process. That thread processes its operation and then continues to process any queued operations in the usual way, unless, of course, there's another thread waiting to process.

A synchronous operation works exactly the same as all operations would on the alternative thread-safe object but doesn't affect any of the asynchronous operations. Obviously a thread executing a synchronous operation on an Activatable Object will block until the asynchronous operations that are currently being processed are complete, but this is the same situation as would be encountered with a thread-safe object.

Whilst my current implementation of an Activatable Object includes support for synchronous operations it's possibly a better design to preclude these and rely solely on commands which can be completed asynchronously.

No returns

One thing that you may have noticed is that none of the operations that are performed on the Activatable Object can return a result. All are fully asynchronous and are initiated in a 'fire and forget' style. If results need to be delivered somewhere then this must be done asynchronously, usually via some form of callback interface or by adding the result to another Activatable Object's input queue.

Implementation details

One of the key pieces of the C++ design that I ended up with was the simplicity of the command queue. In my implementation this is just a simple memory buffer which can be expanded when it fills up. Each command is represented by a one byte value and a variable number of parameter bytes. The user of the queue is responsible for adding their commands and deblocking the commands once a thread is processing. To enable a swift switch from inserting data into the queue to processing the queued commands I have two queues; the active queue and the processing queue. When a thread makes the transition to processing mode it swaps the active queue for the processing queue and processes all of the commands in the processing queue. New commands are always queued to the active queue. There's a lock around the active queue which must be acquired before you can manipulate the active queue and a separate lock to enter processing mode.

When attempting to leave processing mode a thread must first check the active queue for commands. If the active queue contains commands then it must swap the active queue for the, now empty, processing queue and continue processing. A thread can only leave processing mode when both the active and processing queues are empty. This in turn means that when there are lot of operations to be performed on a given object in a short period of time it's usually a single thread that will end up doing all of the work.

It's possible that an 'over-active' Activatable Object could never relinquish control of a thread. If a constant stream of work items comes in at a rate that means there's always work to do then the processing thread will just continue processing. In many designs this isn't an issue, but it could be a problem if you're using a fixed sized pool of threads and expecting to share the threads between a larger number of objects with a lot of work to do. In this situation I've found that the Activatable Object needs to know a little more about how work is fed to the threads that service it. It's then often possible to have the object decide that it has done enough for now and have the processing thread stop processing even if there are items left in the object's work queue. Care must be taken to ensure that the object will eventually have these outstanding work items processed.

An important consequence of the Activatable Object design is that the processing thread need not hold any lock whilst processing. Locks are only held when manipulating the command queue or entering and leaving processing mode. This simplifies command processing code as the processing thread can safely call into user-defined callbacks without risk of creating lock inversions. ■

Acknowledgements

Thanks to Chris Oldwood for encouraging me to adjust my original blog posting and write it up for *Overload*. He also helpfully pointed out the Active Object similarity which was obvious once he'd mentioned it but didn't occur to me initially. Thanks also to Fran, the *Overload* editor, for making the whole process of submitting an article so painless.

References

[DrDobbs] http://www.drdobbs.com/cpp/avoid-calling-unknown-code-while-inside/202802983

[Kegel] 'Dan Kegel's Web Hostel', http://www.kegel.com/c10k.html#threaded

[ServerFramework] http://www.serverframework.com/asynchronousevents/2011/06/tcp-flow-control-and-asynchronous-writes.html

[Wikipedia-1] http://en.wikipedia.org/wiki/Active_object

[Wikipedia-2] http://en.wikipedia.org/wiki/Locality_of_reference

Previously published

This article was previously published at:
http://www.lenholgate.com/blog/2014/07/efficient-multi-threading.html

KISSing SOLID Goodbye

Can you remember what SOLID stands for? Chris Oldwood distills it down to two easy to remember principles.

The SOLID acronym and the design principles behind it have been the guiding light for achieving good object-orientated (OO) designs for as long as I can remember. It also features in nearly every job interview I've attended, and yet I have to revise it because I can never remember exactly what each of the letters stands for!

Over the last few years the reason for my inability to remember these 5 core design principles has become apparent – they are just not dissimilar enough to make them memorable. You would think that any design review meeting where OO was being used as the fundamental paradigm would feature one or more of these terms at almost every turn. But no, there are two other terms which I find always seem better suited and universally covers all the SOLID principles, but with much less confusion – Separation of Concerns (SoC) and Program to an Interface (PtaI).

This article attempts to highlight the seeming redundancy in the SOLID principles and shows where either of the two terms above could be used instead. This does not mean the SOLID principles themselves have no intrinsic value, individually they may well hold a much deeper meaning, but that at a higher level, which is where you might initially look for guidance, it is more beneficial to keep it simple (KISS).

Already on shaky ground

In recent years I have seen a number of talks where this theme has come up. Kevlin Henney, a man who seems to know a thing or two about object-orientated design (amongst many other things), has given a talk where he has deconstructed SOLID [Henney11] and even provided (in collaboration with Anders Norås) his own alternative – FLUID [Henney12].

More recently I attended a talk at the 2014 ACCU conference by James Grenning [Grenning14] about applying the SOLID principles to a non-native OO language – C. During this talk he raised a similar question about which are the two core principles of SOLID, and provided the Single Responsibility Principle (SRP) and the Dependency Inversion Principle (DIP) as his selection.

Whilst the recognition that there is some duplication within the SOLID principles is useful, I wanted to avoid the confusion that would result from trying to always pick two of the five and then use them consistently in any future discussion. What I realised is that I've naturally always side-stepped this issue by using the terms Separation of Concerns (SoC) and Program to an Interface (PtaI) instead.

The remainder of this article looks at each of the SOLID principles and frames them within the context of the two more generalised ones. Rather than discuss them in acronym order I've therefore grouped them in two so they are discussed collectively under the same banner.

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

```
public ProductNameEnricher : Enricher
{
  public void Enrich(Order order)
  {
    // code to make HTTP request to retrieve
    // product details

    // code to transform the object model
  }
}
```

Listing 1

Separation of concerns

The basis of the Separation of Concerns (SoC) principle is not specifically within software design; it transcends that and applies to many other pursuits in life too, such as writing. Consequently it is far less prescriptive in nature, instead preferring only to state that 'stuff' should be thought of in a different light to other 'stuff'. The exact nature of that stuff, how big it is, what it does, who owns it, etc. is entirely context dependent. In terms of a software system we could be talking about the client and server, or marshalling layer and business logic, or production code and test code. It's also entirely possible that the same two areas of discussion be considered separate in one context and yet be united in another, e.g. subsystems versus components.

Single Responsibility Principle

It should be fairly obvious that the Single Responsibility Principle (SRP) is essentially just a more rigid form of SoC. The fact that an actual value has been used to define how many responsibilities can be tolerated has lead some to taking this principle literally, which is why I've found a personal preference for the more general term.

The subtext of SRP is that any class should only ever have one reason to change when that class has a single, well-defined role within the system. For example, I once worked on a codebase where a class was used to enrich an object model with display names for presentation. The class that did the enriching looked similar to Listing 1.

When it came to testing the class a virtual method was introduced to allow the HTTP request to be mocked out; however virtual methods in C# are often a design smell [Oldwood13]. Although it might seem that the class has only one responsibility – enrich the object model with data – in this instance it really has two:

- Make the remote HTTP call to retrieve the enrichment data
- Transform the object model to include the presentation data

When it comes to further responsibilities later, such as error handling of the remote call, caching of data, additional data attributes, etc. the entire responsibility of the class grows and often for more than one reason – HTTP request invocation or object model transformation.

The exact nature of that stuff, how big it is, what it does, who owns it, etc. is entirely context dependent

```
public ProductService : ProductFinder
{
  public Product[] FindProducts(...)
  {
    // code to make HTTP request to retrieve
    // product details
  }
}

public ProductNameEnricher : Enricher
{
  public void Enrich(Order order)
  {
    var products = _proxy.FindProducts(...);

    Enrich(order, products);
  }

  private static void Enrich(order, products)
  {
    // code to transform the object model
  }

  private readonly ProductFinder _proxy;
}
```

Listing 2

Hence the two concerns can be split out into separate classes that can then be independently tested (therefore removing the need for the virtual method), e.g. see Listing 2.

Interface Segregation Principle

The clue that the Interface Segregation Principle (ISP) is just a specialisation of SoC is the use of the word 'segregation'. Replace it with 'separation', jiggle the words around a little and soon you have Separation of Interfaces.

The idea behind ISP is that although a class may support a rich interface, its clients are quite often only interested in a (common) subset of those at any one time. Consequently it would be better to split the interface into just the surface areas that are commonly of interest. A typical example of this is a class for manipulating files or in-memory streams:

```
public class MemoryStream
{
  void Read(byte[] buffer, int count);
  void Write(byte[] buffer, int count);
}
```

It is more prevalent for a client to only be concerned with reading or writing to/from a stream at any one time. Therefore we can use the Extract Interface refactoring [Fowler] and split those responsibilities into two separate interfaces (see Listing 3).

```
public interface Reader
{
  void Read(byte[] buffer, int count);
}

public interface Writer
{
  void Write(byte[] buffer, int count);
}

public class MemoryStream : Reader, Writer
{
  void Read(byte[] buffer, int count);
  void Write(byte[] buffer, int count);
}
```

Listing 3

Program to an interface

I first came across the expression 'program to an interface, not an implementation' when reading the seminal work on *Design Patterns* by the Gang of Four [GoF]. The idea is that you should not (need to) care about how a class is implemented, only that its interface provides you with the semantics you require. Unlike languages such as Java and C#, C++ (amongst others) does not support interfaces as a first-class concept which means the notion of 'interface' is somewhat subtler than just the methods declared within an independently defined interface-style type.

Where the two SOLID principles that fall into the SoC category above are fairly easily to guess from their names alone, the latter three are probably a little more obscure. And this, I feel, is in stark contrast to the name of this encompassing principle.

Liskov Substitution Principle

The Liskov Substitution Principle (LSP) is usually 'explained' by quoting directly from it. What it boils down to is the proposition that two types are related if one can be used in place of the other and the code still works exactly as before. One less intuitive example of this would be switching containers in C++ (see Listing 4).

```
int sum(const std::vector<int>& values)
{
  int sum = 0;
  for (auto it = values.begin();
       it != values.end(); ++it)
    sum += *it;
  return sum;
}
```

Listing 4

If the client only depends on abstractions, then the opportunity to substitute different implementations becomes possible

```
public interface Writer
{
  void Write(string message);
}

public class FileWriter : Writer
{
  void Write(string message);
}

public class DatabaseWriter : Writer
{
  void Write(string message);
}
```

Listing 5

In this example the container could be replaced with `std::dequeue` or `std::list` and the program would still compile, execute and give the exact same result. And yet none of the C++ containers derive from a common 'container interface'; they merely implement the same set of methods to provide the same semantics and can therefore be considered directly substitutable.

Interestingly Kevlin Henney, in the first talk I saw about SOLID [Henney11], made the observation that the Liskov Substitution Principle states that a condition of the relationship is based on there being "no change in the program's behaviour". The example above adheres precisely to that constraint. However many interpret the principle as allowing substitution if it conforms to the same abstract interface, despite the fact that the *implementation* of that interface could produce a program that does something completely different (Listing 5).

Whichever way you interpret it, the ability to substitute one implementation for another is predicated on the need of the two implementations to confirm to the same interface, and for the consumer of that interface to only invoke it in such a way that is compatible with the common semantics of all potential implementations.

Dependency Inversion Principle

Whilst LSP considers what it means for the client code to be able to consume similar types based on semantics, the Dependency Inversion Principle (DIP) tackles how the implementation can be physically partitioned without the client needing direct access to it. If the client only depends on abstractions, then the opportunity to substitute different implementations becomes possible. And because the details of the abstraction is all that is required by the client, then the implementation can live in a different part of the codebase.

The 'inversion' aspect comes from the idea of a layered architecture where the consuming code that only relies on the abstraction can live in the lower layers whilst the actual implementation can live in the higher ones. In an onion-like architecture the abstractions can live in the core whilst the implementations live in the outer service layers.

```
switch (shape->type)
{
  . . .
  case Shape::Square:
    DrawSquare((const Square*)shape);
    break;
  case Shape::Circle:
    DrawCircle((const Circle*)shape);
    break;
  . . .
}
```

Listing 6

The word 'abstraction' could be considered as synonymous with 'interface' if considering it in the context of a programming language that supports it as a first-class concept. In a duck typing environment the subtler meaning is harder to translate as there is nothing concrete to use as an aid in ensuring you're only relying on the abstract behaviour. Either way Program to an Interface is really another way of saying Program to an Abstraction.

Open/Closed Principle

The Open/Closed Principle (OCP) has taken quite a beating in recent times with the likes of Jon Skeet [Skeet13] putting it under the microscope. It appears that some have taken the 'closed for modification' aspect quite literally meaning that the code cannot be modified under any circumstances, even to fix a bug! Others have interpreted it a little more liberally and employed polymorphism as a means to shield both the caller and 3rd party (i.e. uncontrolled) implementations from change.

For example, in languages where polymorphism is not supported as a first-class concept, such as C, switching on a field holding an object's underlying 'type' is a common way of varying behaviour (Listing 6).

With direct support for polymorphism the client code reduces to a single line and is 'open for extension' without requiring modification of the caller by virtue of the fact that any new subclasses will be automatically supported:

```
shape->Draw();
```

Similarly an implementation can be made 'open for extension' by breaking the functionality down into small, focused methods that can be overridden piecemeal by a derived class, e.g. via the Template Method design pattern [Wikipedia]. See Listing 7.

In both these cases the mechanism that decouples the client from the implementation and allows the implementation to be composed at an abstract-level is the interface. Whilst inheritance is a common choice for implementation reuse, composition is more desirable due to its lower coupling. The interface provides the means of describing the extensible behaviour.

Whilst inheritance is a common choice for implementation reuse, composition is more desirable

```
class DocumentPrinter
{
public:
  virtual void Print(Document doc)
  {
    PrintHeader(doc);
    PrintBody(doc);
    PrintFooter(doc);
  }
protected:
  virtual void PrintHeader();
  virtual void PrintBody();
  virtual void PrintFooter();
};
```

Listing 7

If taken in a literal sense the notion of programming to an interface encourages the design of abstractions rather than concrete types which in turn promotes looser coupling between caller and callee. Consequently it also relaxes the need to rely on inheritance as the sole method of extension and instead paves the way for composition to be used as an alternative.

Further simplification

In a private conversation about this idea Kevlin Henney mused that you could even drop Program to an Interface as that is in itself just another variation of the Separation of Concerns principle:

> "If you think about it, Programming to an Interface is an articulation of separating concerns: separate interface from implementation, have dependent code work in terms of interface, i.e., separate from implementation dependency."

Whilst in theory this is an interesting observation, I not sure what, if any, practical value it has. Einstein once said "Everything should be made as simple as possible, but no simpler". The reduction of SOLID down to just the Separation of Concerns and Program to an Interface principles feels to me to be a valuable simplification, whilst the above would be a step too far.

Summary

The goal of this article was to reduce the five core SOLID principles to a more manageable and memorable number for the purposes of everyday use. Instead of getting distracted with the finer details of SOLID I've shown that the two more generalised principles of Program to an Interface and Separation of Concerns are able to provide just as much gravitas to shape a software design. ■

Acknowledgements

This article is merely a refinement of similar ideas already put forward. As such I'm just standing on the shoulders of the giants that are Kevlin Henney and James Grenning. The *Overload* peer reviewers have also helped me stand on my tippy-toes to see that little bit further afield.

References

[Fowler] 'Refactoring: Improving the Design of Existing Code' by Martin Fowler – http://refactoring.com/catalog/extractInterface.html

[GoF] *Design Patterns: Elements of Reusable Object-Oriented Software* by Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma

[Grenning14] 'Designing SOLID C' by James Grenning (ACCU Conference 2014) – http://www.slideshare.net/JamesGrenning/solid-c-accu2014key

[Henney11] 'Will the Real OO Please Stand Up?' by Kevlin Henney (ACCU 2011 conference) – http://accu.org/content/conf2011/Kevlin-Henney-Will-the-Real-OO-Please-Stand-Up.pdf and http://www.slideshare.net/Kevlin/solid-deconstruction

[Henney12] 'SOLID deconstruction' by Kevlin Henney (ACCU 2012 conference) – http://accu.org/content/conf2012/Kevlin_SOLID_Deconstruction.pdf and http://www.slideshare.net/Kevlin/introducing-the-fluid-principles

[Oldwood13] 'Virtual Methods in C# Are a Design Smell' by Chris Oldwood – http://chrisoldwood.blogspot.co.uk/2013/09/virtual-methods-in-c-are-design-smell.html

[Skeet13] 'The Open-Closed Principle', in review by Jon Skeet – http://msmvps.com/blogs/jon_skeet/archive/2013/03/15/the-open-closed-principle-in-review.aspx

[Wikipedia] 'Template Method from Design Patterns' – http://en.wikipedia.org/wiki/Template_method_pattern

TDD Discussions and Disagreements

Recently people have been arguing about the benefits and pitfalls if TDD. Seb Rose summarises the differing standpoints and presents his own.

In recent weeks, a TDD debate has been raging (again) in the blog-o-sphere and on Twitter. A lot of big names have been making bold statements and setting out arguments, of both the carefully constructed and the rhetorically inflammatory variety. I'm not going to revisit those arguments – go read the relevant posts, which I have collected in a handy timeline at the end of this post.

Everyone is right

Instead of joining in the argument, I want to consider a conciliatory post by Cory House entitled 'The TDD Divide: Everyone is right'. He proposes an explanation for these diametrically opposed views, based upon where you are in the software development eco-system:

> Software 'coaches' like Uncle Bob believe strongly in TDD and software craftsmanship because that's their business. Software salespeople like Joel Spolsky, Jeff Atwood, and DHH believe in pragmatism and 'good enough' because their goal isn't perfection. It's profit.

This is a helpful observation to make. We work in different contexts and these affect our behaviour and colour our perceptions. But I don't believe this is the root cause of the disagreement. So what is?

How skilled are you?

In Japanese martial arts they follow an age old tradition known as Shu Ha Ri, which is a concept that describes the stages of learning to mastery. This roughly translates as "first learn, then detach, and finally transcend". (I don't want to overload you with Japanese philosophy, but if you are interested, please take a look at Endo Shihan's short explanation.)

This approach has been confirmed, and expanded on, in modern times by research conducted by Stuart and Hubert Dreyfus, which led to a paper published in 1980. There's a lot of detail in their paper, but Figure 1 shows the main thrust of their findings.

For me, the important point is that novices follow rules and 'don't know what they don't know'. They need to be given unambiguous instructions while they learn what's important.

At the other end of the spectrum, experts use their intuition and metacognition (an awareness and understanding of one's own thought processes). They often can't accurately describe how or why they arrived at a particular decision.

Seb Rose is an independent software developer, trainer and consultant based in the UK. He specialises in working with teams adopting and refining their agile practices, with a particular focus on automated testing. He first worked as a programmer in 1980 writing applications in compiled BASIC on an Apple II. He has worked with many mainstream technologies since then, and is a regular conference speaker, contributor to O'Reilly's *97 Things Every Programmer Should Know* and co-author of the Pragmatic Programmer's *The Cucumber-JVM Book*.

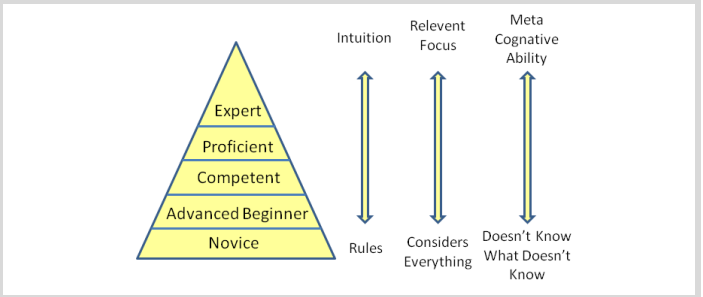


Figure 1

Who are they talking to?

When presenting information it's important to understand who your audience is. Material that's suitable for novices is often not useful for experts. Material for experts can be dangerous in the hands of novices.

Some of the contributors to the TDD debate do try to indicate who they are writing for, but most don't. And even when you do say something like "for experts only" you're leaving it to the reader to decide if they are an expert or not, which is a decision that we are poorly equipped to make. In fact, the Dunning-Kruger effect shows that people who are in the early stages of acquiring a skill frequently overestimate their competence, while those who are very skilled tend to underestimate it.

The internet is an open resource. No matter what sort of health warning you put on your blog posts, they will inevitably be read by people of differing skill levels. And therein lies the problem. The bloggers have an implicit idea of who their audience is – and it might not be you.

The people who warn against TDD usually come from environments where the team is working at competent level or above and that's who they are talking to. If you respond to these posts by thinking "Oh cool. This person is saying that I don't have to do this difficult thing" then you're probably a novice. Carry on practising until it becomes easy and then you'll know **why** they didn't find it useful.

Those who promote TDD often work with teams that are less skilled and they have seen the benefits that derive from acquiring these skills. If TDD is something you've already become comfortable with, then you'll know **why** you've chosen not to use it any more and it won't be "because it was difficult".

To TDD or not to TDD. That is not the question.

TDD is a technique that has costs and benefits. Whether it is right for you and your colleagues depends on your team's context – the domain, the skill level, the schedule, the risk. Like any technique, it's no use if you do it badly. Like any technique, it can neither help nor hinder if you don't apply it. If you want to get good at TDD you have got to practice it.

The internet is an open resource. No matter what sort of health warning you put on your blog posts, they will inevitably be read by people of differing skill levels

You will need to get good at writing robust unit tests that exercise the behaviour not the implementation. You will find yourself getting better at creating designs that are made up of smaller, cohesive, decoupled components. You will have an automated suite of tests that give you fast feedback, the confidence to refactor and protection from regressions. You will have executable documentation that doesn't rot over time.

If your development process doesn't deliver these benefits, you'll have inevitably felt the pain caused by long debugging sessions, unexpected regressions, poorly structured code and stale (or missing) documentation. There will be parts of the codebase where you fear to tread.

You can acquire unit testing and design skills without practicing TDD, but many people find that the structure of TDD really helps keep focus. Once you get good, you'll be competent (or better) in all these skills and well placed to decide whether the cost of TDD outweighs the benefits or not. The skills you've picked up on the way will be invaluable whichever choice you make.

But if you've never tried TDD, or you've never practised enough to be comfortable with it, then you're still a novice. My advice is to keep on practising – it'll be worth it.

Teaching TDD (TTDD)

Another flurry of discussion about TDD was sparked off by a recent post from Justin Searls [Searls]. In it he lists a number of failures that range from "Encouraging costly Extract refactors" to "Making a mess with mocks" all of which distract attention from the concept that "TDD's primary benefit is to improve the *design* of our code". He concludes by suggesting that once you have written a failing test, rather than get-to-green (see Figure 2) in the simplest way possible you should "intentionally defer writing any implementation logic! Instead, break down the problem by dreaming up all of the objects you wish you had at your disposal". In essence, design the elements of the solution while the first test is still red.

It's an interesting post that raises a number of issues, but for me its value lies chiefly in opening the subject up for debate. The introduction is particularly pertinent – just setting a class a bundle of katas to do does not, of itself, encourage learning. The pains experienced while doing the exercise need to be teased out, discussed and have alternative approaches described. If you don't hear the penny drop, then it hasn't dropped.

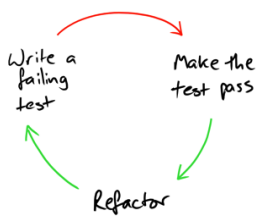


Figure 2

Pitching in with characteristic vigour and brimstone came Uncle Bob with a robust rebuttal containing both heat and light (though some have been put off by the heat and never got to the light). Bob makes some good points regarding the fallacy of writing tests around extracted classes, the tool support for extract refactoring and the central place of refactoring in the Red-Green-Refactor cycle.

By the conclusion, however, Bob has switched tack. He states that while refactorings are cheap within architectural boundaries, they are expensive across them. Whether he's right or wrong on this point[1] is of no concern because now he's addressing the wrong question. The question at hand is "how is it best to teach TDD?" and it was taken up in a short Twitter exchange between me, Kevlin Henney and Mike Long.

Mike, being hard-core, says that he teaches TDD by "start[ing] with writing the test framework. Start from assert and up." There's much more to discover about this approach, and it's certainly reminiscent of how Kent Beck learns a new language (by reimplementing xUnit).

Kevlin, in an e-mail, is more discursive. He works with a mixture of prepared material, discussions, instructor-led demonstrations and learning exercises. There are katas in the material, but there's lots of interaction to draw out the key points and really get them to stick.

In my TDD training, I start by focusing on fundamental unit testing practices. The key word here is 'fundamental'. TDD is unit testing++, so you need to have a firm grasp of what a good unit test looks like. We work through a series of simple bank account examples (in pairs, using Cyber-Dojo, of course [CyberDojo]) that bring out the 6 essential properties of unit testing [Rose12]. I then use an example (based on an idea of Rob Chatley [Chatley]) to introduce test doubles before moving into one of my own legacy code exercises. In-between I use a couple of the katas that ship with Cyber-Dojo – usually LcdDigits and PrintDiamond – to get a more varied domain experience.

There are lots of ways to teach TDD and Justin Searls has certainly identified one way of doing it sub-optimally. I have to disagree with his conclusion that the fix for this is to defer implementation till after the solution design is fully sketched out. At the opposite end of the spectrum is Keith Braithwaite's "TDD as if you meant it" [Braithwaite09] which is an exercise you can (and should) try at home.

In my opinion, the success of any training is dependent on the trainer – the material is of secondary importance. So if you decide that some TDD training is for you, remember to think about **who** is training you, not just how long the course is and how much it costs. ■

1. Grady Booch once said: "All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change." Steve Tooke pointed me to an old post by Nat Pryce, which hints at a different trade off between change and cost. After all, when was the last time you were perfectly happy with an 'architectural' decision that was made more than a week ago by somebody else?

In my opinion, the success of any training is dependent on the trainer – the material is of secondary importance

References

[Braithwaite09] Keith Braithwaite, (2009) 'Thought-provoking TDD exercise at the Software Craftmanship conference', February 2009. http://gojko.net/2009/02/27/thought-provoking-tdd-exercise-at-the-software-craftsmanship-conference/

[Chatley] Robert Chatley http://chatley.com/

[CyberDojo] Cyber-dojo website: cyber-dojo.org

[Rose12] 'Bad Test, Good Test', Seb Rose, available on slideshare, http://www.slideshare.net/sebrose/bad-test-good-test

[Searls] Justin Searl's blog: http://blog.testdouble.com/

Selected posts and tweets

September 30, 2008 – Kent Beck
http://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565

Unknown date, 2009 – J.B. Rainsberger
http://www.jbrains.ca/permalink/how-test-driven-development-works-and-more

October 6, 2011 – Ian Cooper
http://codebetter.com/iancooper/2011/10/06/avoid-testing-implementation-details-test-behaviours/

May 1, 2013 – Steve Fenton
http://www.stevefenton.co.uk/Content/Blog/Date/201305/Blog/My-Unit-Testing-Epiphany/

May 13, 2013 – Steve Fenton
http://www.stevefenton.co.uk/Content/Blog/Date/201305/Blog/My-Unit-Testing-Epiphany-Continued/

July 15, 2013 – Philip Ledgerwood
http://thecuttingledge.com/?cat=5

January 13, 2012 – Dan North
https://twitter.com/tastapod/status/157633913009864704

June 12, 2013 – Ian Cooper
http://vimeo.com/68375232

January 25, 2014 – Justin Searls
http://blog.testdouble.com/posts/2014-01-25-the-failures-of-intro-to-tdd.html

January 27, 2014 – Uncle Bob
http://blog.8thlight.com/uncle-bob/2014/01/27/TheChickenOrTheRoad.html

February 25, 2014 – Santiago Basulto
https://medium.com/tech-talk/e810d9b4fb02

April 23, 2014 – David Heinemeier Hansson
http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

April 25, 2014 – Uncle Bob
http://blog.8thlight.com/uncle-bob/2014/04/25/MonogamousTDD.html

April 29, 2014 – Kent Beck
https://www.facebook.com/notes/kent-beck/rip-tdd/750840194948847

April 29, 2014 – David Heinemeier Hansson
http://david.heinemeierhansson.com/2014/test-induced-design-damage.html

April 30, 2014 – Gary Bernhardt
https://www.destroyallsoftware.com/blog/2014/tdd-straw-men-and-rhetoric

April 30, 2014 – Uncle Bob
http://blog.8thlight.com/uncle-bob/2014/04/30/When-tdd-does-not-work.html

April 30, 2014 – Tom Stuart
http://codon.com/how-testability-can-help

May 1, 2014 – Uncle Bob
http://blog.8thlight.com/uncle-bob/2014/05/01/Design-Damage.html

May 1, 2014 – Cory House
http://www.bitnative.com/2014/05/01/the-tdd-divide/

Acknowledgements

The diagram in Figure 2 was provided by Nat Pryce.

Previously published

This article was previously published at: http://claysnow.co.uk/to-tdd-or-not-to-tdd/ and http://claysnow.co.uk/teaching-tdd-ttdd/