overload 127

JUNE 2015 £3

Meet the Social Side of Your Codebase

Programming requires collaboration. We see how to uncover communication paths in your organisation.

Mocks are Bad, Layers are Bad

Is there a better unit testing technique than using mock objects?

Terse Exception Messages

On the importance of clear error messages

Get Debugging Better

Some simple tricks to save debugging time and pain

Make and Forward Consultables and Delegates

A technique to avoid boilerplate when forwarding control to a contained object

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 127

June 2015

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony.ajw@gmail.com

Matthew Wilson
stlsoft@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 128 should be submitted by 1st July 2015 and those for Overload 129 by 1st September 2015.

The ACCU

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

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

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

A little more conversation, a little less action

A tension exists between talking and doing. Frances Buontempo wonders which matters more.

For almost three years we have had much talk of editorials and reasons for not getting round to actually writing one. This gives the lasting impression that much verbosity and prevarication can stand in the way of actions. The time has probably come to knuckle down and just do it, to coin a phrase. Actions, we are told, speak louder than words. I notice we are told this in words, and not through actions. Why do people say such things? "This calls for immediate discussion," to borrow another quote [Monty Python].

I was recently called practical – a woman of actions, not words. I was left wondering if that was a veiled insult or not. Or perhaps a way of continuing the discussion thereby avoiding getting stuff done. It is possible it was just an observation and I should be less suspicious. Whether it is appropriate to call someone practical based on what they are saying, rather than seeing them in action is another question. In all fairness, the opposite would presumably be 'impractical' and I must say I'm not sure how you could fairly accuse anyone of such a thing. A person in themselves is not impractical as such, but their ideas could be. They may even be 'inconceivable', yet as most of us are often reminded, "That word you keep using. I do not think it means what you think it means," [Inigo Montoya]. The precise details of the accusation of practicality evade me now, but it certainly stemmed from a tension between talking things through to build concord in the group and just starting to get something in place and revisiting it when we all had a bit more knowledge and experience. Some people would prefer to blaze a trail, getting things up and running, fine tuning them afterwards, while others would rather discuss the possible approaches first and wait until everyone is in agreement on the best course of action. A few others will take a dictatorial standpoint and just tell people how they should do things, even going as far as setting up FXCop or other style rules to enforce their point of view, or perhaps inventing a new programming language to enforce code layout [BDFL]. These tensions find parallels in many areas, for example performance versus readability, or even premature optimisation versus deliberate pessimisation [Rule9], where the former finds us trying to make something faster, often via complicated tricks which may not be needed and the latter,

> "Is when you write code that is slower than it needs to be, usually by asking for unnecessary extra work, when equivalently complex code would be faster and should just naturally flow out of your fingers." [GOTW]

I'm not sure I've ever seen code flow out of my fingers. Feel free to write in if you have. Of course, we have all seen people argue over the relative merits of either big upfront design or a more incremental approach. How does systems thinking fit into all of this?

If you are starting on a new software project, or even adding features to a legacy code-base, would you get a continuous integration (CI) server up and running straight away, possibly even adding a unit test project, which might well have 0 tests, upfront? Or would you wait until you had explored the code or requirements a little, and added a few tests to see what was possible or appropriate? Or would you allow the team to discuss whether Jenkins or Team City was the best? Or toss a coin to solve that altercation, and then spend weeks arguing about which C++ unit testing framework is the best, eventually hand-rolling your own, without JUnit-style xml output, making it hard to grok the results on your shiny new CI box? Perhaps before any of this you need to spend a few weeks defining 'Unit test'. Google currently proudly tells me it can find 3,660,000 results (in less than half a second). Good luck with that one. I try to get quick running tests to run on every commit, and might just run a full suite of regression, smoke, hell-fire and brimstone tests in the nightly build. Trying to decide when night is on a distributed team is yet another matter. I wonder if this is what caused me to be called practical? I'll never know. It would be foolhardy to allow each developer to choose their favoured testing framework, and CI setup and expect them to work together on a project. You could have a couple of rival setups and see which wins, but you many find this time-consuming and there may be no clear winner in the long-run. Let's face facts – programmers are frequently opinionated and won't always agree. Even with the previous statement.

Instead of rushing in 'where angels fear to tread' [Pope], it might be wise to talk things through as a team first, though be aware that in the previous line Pope does say, "They'll talk you dead." Programming is a creative process, needing space and time to flourish. This often comes from bouncing ideas off one another, or sharing experiences, be they war-stories, or tales of a job well done. Be aware that some people do not manage to speak up in a loud group of people for various reasons; strive to include the quieter ones. They may be shy, or perhaps they think you are all wrong and will listen in silence then stay late and do something completely different, unsupervised. Others may be encountering a new approach for the first time, and would rather go away and read around the subject before offering an opinion. Some might want to think things through by trying out a few ideas, maybe via a blog inviting comments from the world and her husband, or by trying a spike project. Others may want to see what a brief internet search turns up, and may even edit the Wikipedia page you were trying to use as evidence while you are arguing. Equally importantly, not everyone means the same things when they say the same words. This project may inspire 'awe'. It may be 'interesting'. And not everyone on your team may speak the language you are using as their first language, be that human or programming. Be careful with colloquialisms. Even be aware of whether nodding one's head means 'Yes' or 'No'. I am told this is not consistent across all cultures. Burkhard

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 works at Bloomberg, has been a programmer since the 90s, 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.

Kloss talked about 'Polyglot programming' at the conference [Kloss]. In the preview to the ACCU London chapter, he talked about differing human languages and reminded us of the Bible story of the Tower of Babel, emphasising that using different languages can stand in the way of working well together. He then considered how to get various programming languages communicating. The words you use matter and various protocols can help. Don't speak over people. Say what you mean and mean what you say. Try to find out what others think you mean when they listen to what you say.

It is important to listen as well. We are told, "In space, no-one can hear you scream." We may also have been asked, "If a tree falls in an empty forest, does it make a sound?" Bishop Berkeley asked a more fundamental question. "If a tree, growing in a quadrangle at a university is not being observed, does it continue to exist?" Esse est percipi – to be, is to be perceived [Berkeley]. As I said, don't ignore the quieter people on your team. Berkeley himself needs to be understood in the context of Descartes's dualism and Hobbes' insistence that only material things exist, perhaps meaning there can be no ideas. Clearly this is completely off-topic, even for an editorial avoidance exercise. Nonetheless, some of Berkeley's ideas may be pertinent. When studying his writing at University, I was left wondering what people heard and understood when others spoke. When we describe colour, or taste, are we discussing the same experience or do we all have our own world view? If my husband tells me the blue light is flashing on the coffee machine, I may observe it looks orange to me. If the guy beside me at work states that, "All coffee tastes disgusting" while others claim mushrooms are disgusting, does this mean mushrooms are coffee flavoured? Or none of these people have any taste? Even when using the same human language, our experiences and our physiology affect our perceptions. What is blue anyway? Perhaps everyone else is just seeing in shades of grey. Children appear to learn words by repeating what those around them say until they manage to reformulate the sounds into phrases of their own, eventually appearing to be understood. All words are a convention, which can change over time or end up as parallel forks, such as differences between American and British English. Even if you appear to be speaking the same dialect of the same language, communication often seems like a best guess. You need to assume 'blue' means the same thing to someone else. Sometimes you need to add extra clarification, as in "That would be a black coffee WITHOUT milk," or "No, the other left." Sometimes you seem to be 'in sync' with the people you are trying to commune with – you finish each other's sentences. You grab the keyboard and sort the typos out or refactor the code to something that makes you both nod; hopefully in agreement. This presupposes you are pair programming – some may complain that this is a form of action, not communication. It does suggest they may be more apposite than opposite. It is possible to communicate without words – a band can jam together seemingly spontaneously, without talking it through for hours first. It is possible to express something through mime. Or just pointing and shouting. Or banging your head on the desk.

Looking at the etymology of 'action' and 'conversation' is revealing. For those who claim they want a little less of the yacky-yack, chatter, natter or to put in more succinctly, in fewer works, talk, and a little more action, don't forget that an action can also be a lawsuit – a written communication. It also relates to the word 'deed', involving legal documents too. With a beautiful symmetry, I notice an etymology website suggests conversation stems from 'the act of living with' [Etymonline]. Conversation is something you do, not something you talk about. Acting is putting on a performance, perhaps by reading aloud words someone has written. If we ask for more action and less conversation, or vice versa, the lines are blurred. Sometimes ideas are communicated more effectively by diagrams or precise symbols, say mathematics, or by working on a small code sample together than by vocalising thoughts. Things are not always clear cut. In the end, acting is not enough. There needs to be interaction. This might need to be through some form of conversation. As geeks we might not always do this face to face. An email is ok. Some demo code, or notes on a wiki count. Writing an article can be good. Once in a while, do consider talking to those around you though. Try to listen to others, if you expect them to listen to you. Make an effort to understand their perspective, and try to learn to communicate clearly. If someone really won't listen, try putting some tests on your CI box and making that email the offender for you when the tests fail. I know I did, and it worked. And always recall the sage advice

> "Omit needless words." [Style]

But don't take it too far.

> "Omit ~~needless~~ words." [anon]

References

[BDFL] 'Benevolent dictator for life' originally used to refer to Guido van Russom, inventor of Python: http://en.wikipedia.org/wiki/Benevolent_dictator_for_life

[Berkeley] http://plato.stanford.edu/entries/berkeley/

[Etymonline] http://www.etymonline.com/index.php?term=conversation

[GOTW] http://herbsutter.com/2013/05/13/gotw-2-solution-temporary-objects/

[Inigo Montoya] *The Princess Bride* http://www.imdb.com/title/tt0093779/quotes

[Kloss] 'Thriving in a polyglot world.' ACCU2015 http://accu.org/index.php/conferences/accu_conference_2015/accu2015_sessions#thriving_in_a_polyglot_world

[Monty Python] *The Life of Brian* http://montypython.50webs.com/scripts/Life_of_Brian/23.htm

[Pope] Alexander Pope *An essay on criticism*, 1709.

[Rule9] *C++ Coding Standards* Sutter and Alexandrescu. 2004.

[Style] *The Elements of Style* Strunk. 1918.

Meet the Social Side of Your Codebase

Programming requires collaboration. Adam Tornhill suggests some ways to uncover communication paths in your organisation.

Let's face it – programming is hard. You could spend an entire career isolated in a single programming language and still be left with more to learn about it. And as if technology alone weren't challenging enough, software development is also a social activity. That means software development is prone to the same social biases that you meet in real life. We face the challenges of collaboration, communication, and team work.

If you ever struggled with these issues, this article is for you. If you haven't, this article is even more relevant: The organizational problems that we'll discuss are often misinterpreted as technical issues. So follow along, learn to spot them, react, and improve.

Know your true bottlenecks

Some years ago I did some work for a large organization. We were close to 200 programmers working on the same system. On my first day, I got assigned to a number of tasks. Perfect! Motivated and eager to get things done, I jumped right in on the code.

I soon noticed that the first task required a change to an API. It was a simple, tiny change. The problem was just that this API was owned by a different team. Well, I filed a change request and walked over to their team lead. Easy, he said. This is a simple change. I'll do it right away. So I went back to my desk and started on the next task. And let me tell you: that was good because it took one whole week to get that 'simple' change done!

I didn't think much about it. But it turned out that we had to modify that shared API a lot. Every time we did, the change took at least one week. Finally I just had to find out what was going on — how could a simple change take a week? At the next opportunity, I asked the lead on the other team. As it turned out, in order to do the proposed change, he had to modify another API that was owned by a different team. And that team, in turn, had to go to yet another team which, in turn, had the unfortunate position of trying to convince the database administrators to push a change.

No matter how agile we wanted to be, this was the very opposite end of that spectrum. A simple change rippled through the whole organization and took ages to complete. If a simple change like that is expensive, what will happen to larger and more intricate design changes? That's right — they'll wreak havoc on the product budget and probably our codebase and sanity too. You don't want that, so let's understand the root causes and see how you can prevent them.

Understand the intersection between people and code

In the story I just told, the problem wasn't the design of the software, which was quite sound. The problem was an organization that didn't fit the way the system was designed (see Figure 1).

Adam Tornhill Adam is a programmer who combines degrees in engineering and psychology. He's the author of *Your Code as a Crime Scene*, has written the popular 'Lisp for the Web' tutorial and self-published a book on *Patterns in C*. Adam also writes open-source software in a variety of programming languages. His other interests include modern history, music and martial arts.

When we have a software system whose different components depend upon each other and those components are developed by different programmers, well, we have a dependency between people too. That alone is tricky. The moment you add teams to the equation, such dependencies turn into true productivity bottlenecks accompanied by the sounds of frustration and miscommunication.

Such misalignments between organization and architecture are common. Worse, we often fail to recognize those problems for what they are. When things go wrong in that space, we usually attribute it to technical issues when, in reality, it's a discrepancy between the way we're organized versus the kind of work our codebase supports. These kind of problems are more severe since they impact multiple teams and it's rare that someone has a holistic picture. As such, the root cause often goes undetected. That means we need to approach these issues differently. We need to look beyond code.

Revisit Conway's Law

This common problem of an organization that's misaligned with its software architecture isn't new. It has haunted the software industry for decades. In fact, it takes us all the way back to the 60's to a famous observation about software: Conway's Law. Conway's Law basically claims that the way we're organized will be mirrored in the software we design; Our communication structure will be reflected in the code we write.

Conway's Law has received a lot of attention over the past years, and there are just as many interpretations of it as there are research papers about it. To me, the most interesting interpretation is Conway's Law in reverse. Here we start with the system we're building: given a proposed software architecture, what's the optimal organization to develop it efficiently?

When interpreted in reverse like that, Conway's Law becomes a useful organizational tool. But, most of the time we aren't designing new architectures. We have existing systems that we keep maintaining, improving, and adding new features to. We're constrained by our existing architecture. How can we use Conway's Law on existing code?

Let Conway's Law guide you on legacy systems

To apply Conway's Law to legacy code, your first step is to understand the current state of your system. You need to know how well your codebase

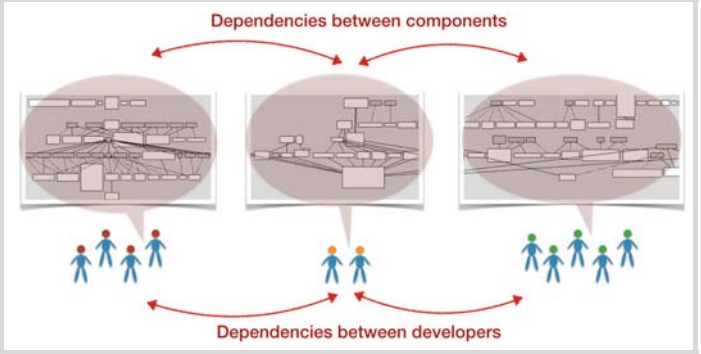

Figure 1

No matter how agile we wanted to be, this was the very opposite end of that spectrum. A simple change rippled through the whole organization and took ages to complete.

supports the way you work with it today. It's a tricky problem — we probably don't know what our optimal organization should look like. The good news is that your code knows. Or, more precisely, its history knows.

Yes, I'm referring to your version-control data. Your version-control system keeps a detailed log of all your interactions with the codebase. That history knows which developers contributed, where they crossed paths, and how close to each other in time they came by. It's all social information — we're just not used to thinking about version-control data in that way. That means we can mine our source code repositories to uncover hidden communication structures. Let's see how that looks.

Uncover hidden communication paths

According to Conway, a design effort should be organized according to the need for communication. This gives us a good starting point when reasoning about legacy systems. Conway's observation implies that any developers who work in the same parts of the code need to have good communication paths. That is, if they work with the same parts of the system, the developers should also be close from an organizational point of view.

As you see in Figure 2, we follow a simple recipe. We scan the source code repository and identify developers who worked on the same modules:

1. Every time two programmers have contributed to the same module, these developers get a communication link between them.
2. If another programmer contributes to the same code, she gets a communication link as well.
3. The more two programmers work in the same parts of the code, the stronger their link.

Once you've scanned the source code repository, you'll have a complete graph over the ideal communication paths in your organization. Note the emphasis on ideal here; These communication paths just show what should have been. There's no guarantee that these communication paths exist in the real world. That's where you come in.

Now that you have a picture over the ideal communication paths, you want to compare that information to your real, formal organization. Any discrepancies are a signal that you may have a problem. So let's have a look at what you might find.

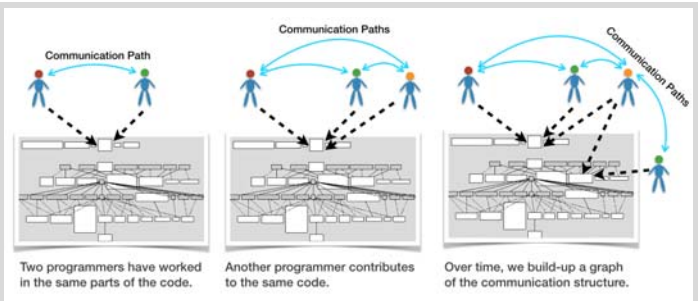

Figure 2

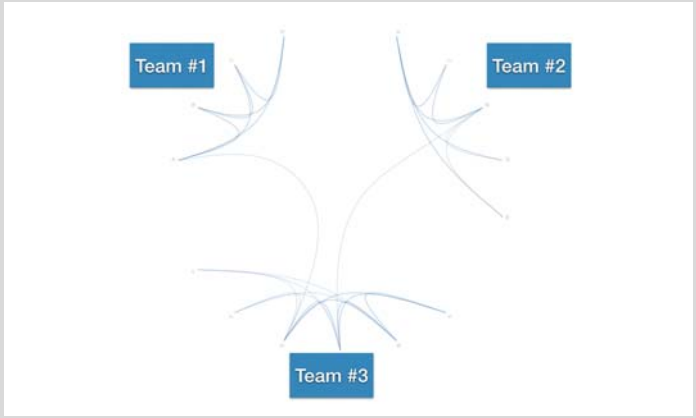

Figure 3

Know the communication paths your code wants

In the best of all worlds, you'll be close to what Conway recommended. Have a look at Figure 3. It shows one example of an ideal communication diagram.

The example in Figure 3 illustrates three teams. You see that most of the communication paths go between members of the same teams. That's a good sign. Let's discuss why.

Remember that a communication diagram is built from the evolution of your codebase. When most paths are between members on the same team, that means the team members work on the same parts of the code. Everyone on such a team has a shared context, which makes communication easier.

As you see in the picture above, there's the occasional developer who contributes to code that another team works on (note the paths that go between teams). There may be different reasons for those paths. Perhaps they're hinting at some component that's shared between teams. Or perhaps it's a sign of knowledge spread: while cohesive teams are important, it may be useful to rotate team members every now and them. Cross-pollination is good for software teams too. It often breeds knowledge.

The figure above paints a wonderful world. A world of shared context with cohesive teams where we can code away on our tasks without getting in each other's way. It's the kind of communication structure you want.

However, if you haven't paid careful attention to your architecture and its social structures you won't get there. So let's look at the opposite side of the spectrum. Let's look at a disaster so that you know what to avoid.

A man-month is still mythical

About the same time as I started to develop the communication diagrams, I got in contact with an organization in trouble. I was allowed to share the story with you, so read on — what follows is an experience born in organizational pain.

Now, how do you take something you know takes a year and compress it down to just three months? Easy – just throw four times as many developers at it.

The trouble I met at that organization was a bit surprising since that company had started out in a good position. The company had set out to build a new product. And they were in a good position because they had done something very similar in the past. So they knew that the work would take approximately one year.

A software project that's predictable? I know – crazy, but it almost happened. Then someone realized that there was this cool trade-show coming up in just three months. Of course, they wanted the product ready by then. Now, how do you take something you know takes a year and compress it down to just three months? Easy – just throw four times as many developers at it.

So they did.

The project was fast-paced. The initial architecture was already set. And in shorter time than it would take to read The Mythical Man-Month, 25 developers were recruited to the project. The company chose to organize the developers in four different teams.

What do you think the communication paths looked like on this project? Well, they are in Figure 4.

It's a cool-looking figure, we have to agree on that. But let me assure you: there's nothing cool about it in practice. What you see is chaos. Complete chaos. The picture above doesn't really show four teams. What you see is that in practice there was one giant team of 29 developers with artificial organizational boundaries between them. This is a system where every developer works in every part of the codebase – communication paths cross and there's no shared context within any team. The scene was set for a disaster.

Learn from the post-mortem analysis

I didn't work on the project myself, but I got to analyze the source code repository and talk to some of the developers. Remember that I told you that we tend to miss organizational problems and blame technologies instead? That's what happened here too.

The developers reported that the code had severe quality problems. In addition, the code was hard to understand. Even if you wrote a piece of code yourself, two days from now it looked completely different since five other developers had worked on it in the meantime.

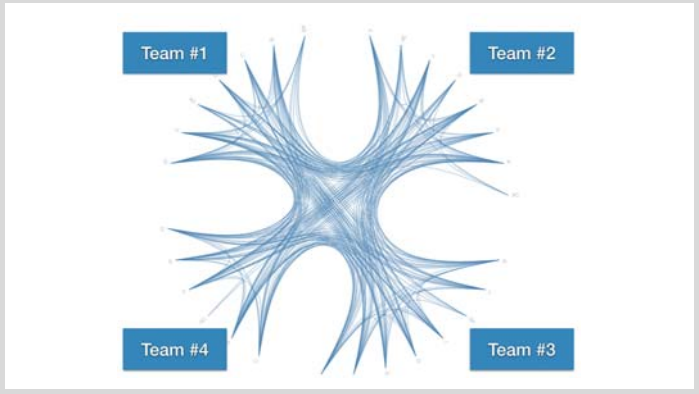

Figure 4

Finally, the project reported a lot of issues with merge conflicts. Every time a feature branch had to be merged, the developers spent days just trying to make sense of the resulting conflicts, bugs, and overwritten code. If you look at the communication diagram above you see the explanation. This project didn't have a merge problem – they had a problem that their architecture just couldn't support their way of working with it.

Of course, that trade show that had been the goal for the development project came and went by without any product to exhibit. Worse, the project wasn't even done within the originally realistic time frame of one year. The project took more than two years to complete and suffered a long trail of quality problems in the process.

Simplify communication by knowledge maps

When you find signs of the same troubles as the company we just discussed, there are really just two things you can do:

1. Change your architecture to support your way of working with it.
2. Adapt your organization to fit the way your architecture looks.

Before you go down either path you need to drill deeper though. You need to understand the challenges of the current system. To do that efficiently, you need a knowledge map.

Build a knowledge map of your system

In *Your Code as a Crime Scene*, we develop several techniques that help us communicate more efficiently on software projects. My personal favorite is a technique I call Knowledge Maps.

A knowledge map shows the distribution of knowledge by developer in a given codebase. The information is, once again, mined from our source code repositories. Figure 5 is an example of how it looks.

In Figure 5, each developer is assigned a color. We then measure the contributions of each developer. The one who has contributed most of the code to each module becomes its knowledge owner. You see, each colored circle in the figure above represents a design element (a module, class, or file).

You use a knowledge map as a guide. For example, the map in Figure 5 shows the knowledge distribution in the Scala compiler. Say you join that project and want to find out about the Backend component in the upper right corner. Your map immediately guides you to the light blue developer (the light blue developer owns most of the circles that represent modules in the backend part). And if she doesn't know, it's a fairly good guess the green developer knows.

Knowledge maps are based on heuristics that work surprisingly well in practice. Remember the story I told you where a simple change took a week since the affected code was shared between different teams? In that case there were probably at least 10 different developers involved. Knowledge maps solve the problem by pointing you to the right person to talk to. Remember – one of the hardest problems with communication is to know who to communicate with.

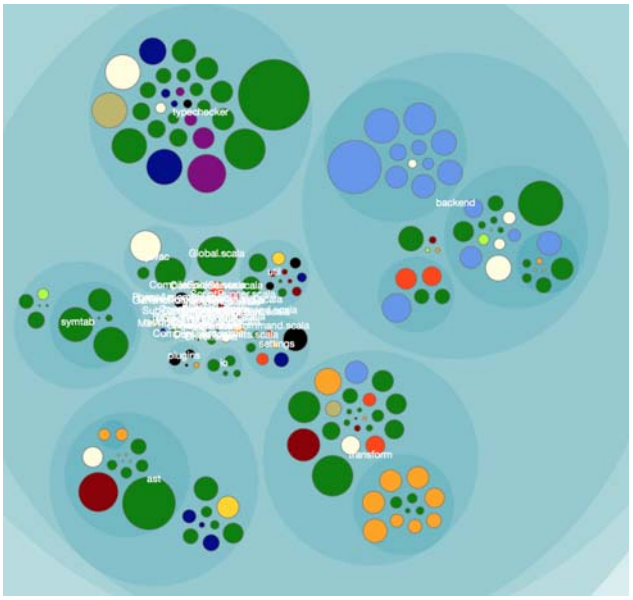

Figure 5

Scale the knowledge map to teams

Now that we have a way of identifying the individual knowledge owners, let's scale it to a team level. By aggregating individual contributions into teams, you're able to view the knowledge distribution on an organizational level. As a bonus, we get the data we need to evaluate a system with respect to Conway's Law. How cool is that?

From the perspective of Conway, the map in Figure 6 looks pretty good. We have three different teams working on the codebase. As you see, the Red team have their own sub-system. The same goes for the Pink team (yes, I do as the mob boss Joe in *Reservoir Dogs* and just assign the colors). Both of these teams show an alignment with the architecture of the system.

But have a look at the component in the lower right corner. You see a fairly large sub-system with contributions from all three teams. When you find something like that you need to investigate the reasons. Perhaps your organization lacks a team to take on the responsibility of that sub-system? More likely, you'll find that code changes for a reason: If three different teams have to work on the same part, well, that means the code probably has three different reasons to change. Separating it into three distinct components may just be the right thing to do as it allows you to decouple the teams.

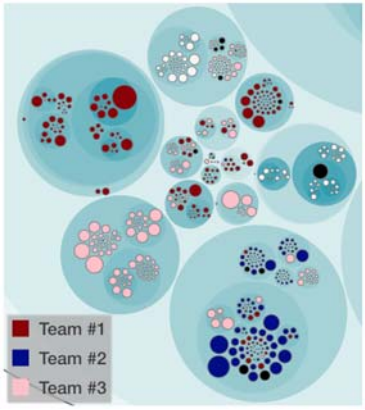

Figure 6

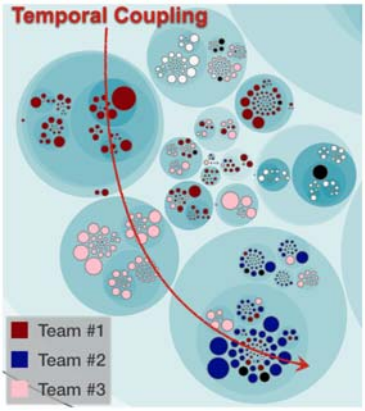

Figure 7

Identify expensive change patterns

Mapping out the knowledge distribution in your codebase is one of the most valuable analyses in the social arsenal. But there's more to it. What if you could use that information to highlight expensive change patterns? That is, change patterns that ripple through parts of the code owned by different teams. Figure 7 show how it looks.

The picture in Figure 7 highlights a modification trend that impacts all three teams. Where does the data come from? Well, again we turn to our digital oracle: version-control data.

There's an important reason why I recommend the history of your code rather than the code itself. The reason that dependencies between multiple teams go unnoticed is because those dependencies aren't visible in the code itself. This will just become even more prevalent as our industry moves toward distributed and loosely coupled software architectures as evidenced by the current micro-services trend.

The measure I propose instead is temporal coupling. Temporal coupling identifies components that change at (approximately) the same time. Since we measure from actual modifications and not from the code itself, temporal coupling gives you a radically different view of the system. As such, the analysis is able to identify components that change together both with and without physical dependencies between them.

Overlaying the results of a temporal coupling analysis with the knowledge map lets you find team productivity bottlenecks. Remember the organization I told you about, the one where a small change took ages? Using this very analysis technique lets you identify cases like that and react on time.

Explore the evolution

We're almost through this whirlwind tour of software evolutionary techniques now. Along the way, you've seen how to uncover the ideal communication paths in your organization, how to evaluate your architecture from a social perspective, and how you can visualize the knowledge distribution in your codebase.

We also met the concept of temporal coupling. Temporal coupling points to parts of your code that tend to change together. It's a powerful technique that lets you find true productivity bottlenecks.

As we look beyond the code we see our codebase in a new light. These software evolutionary techniques are here to stay: the information we get from our version-control systems is just too valuable to ignore. ■

Everything in the article is explained in detail with worked examples from real projects in Adam's new book, *Your Code as a Crime Scene* (https://pragprog.com/book/atcrime/your-code-as-a-crime-scene)

The techniques are available as a command line tool at https://github.com/adamtornhill/code-maat.

Mocks are Bad, Layers are Bad

Many people use mocks in their unit tests. Andy Balaam asks if there's a better way.

It's time we admitted something: use of complex mocks is a code smell, and must be eliminated from a healthy code base.

Sometimes mocks are necessary, but I will argue that we need to structure our code to minimise their use, and to make them simple when they are needed. Let's start with a feeling.

Some days the tests just feel bad.

We've all written the unit tests I'm talking about – they are painful to write because you have to construct layers of mocks to satisfy your module's dependencies, and when you've written them you notice a strange sense of unease: you don't get that safe feeling tests usually give you.

It's almost like writing those tests was a waste of time.

Because let's face it, all you did was check that your code calls the methods you think it should call. If you ever change the details of the implementation, the tests will need to change to match.

At this moment, your colleague (you know, the one who 'doesn't see the point of tests') leans over your shoulder and whispers: "You're just writing the code twice."

That sense of unease indicates a smell: we must expunge it.

How can we do things differently?

How do we get rid of mocks?

We want to avoid complex mocks, especially those that embody implementation details of the code under test.

Of course, we could just stop writing unit tests: then we can have that sense of unease all the time. Alternatively, we could adopt 'Classical TDD' [Fowler-1], where tests cover several layers of functionality at once, rather than being restricted to a single unit. This makes our tests quite effective, since they cover the interactions between layers, but can make it much harder to debug when something fails.

What I want to argue is that we should do something different: avoid layers.

Reject layering

I hope to convince you that thinking of our software as a series of layers is damaging.

I'm going to start with an example. Imagine you are asked to implement a very simple markup rendering engine that accepts a subset of HTML (involving only text in paragraphs) and renders the result as an image. A layered approach might lead us to write an HTML parsing layer, consumed by a font rendering layer, consumed by a flow layout layer.

Andy Balaam Andy 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

[If this design seems crazy to you, you're way ahead of me. Consider: is the complexity of our day-to-day work hiding decisions that are actually as crazy as this one?]

Let's write down some classes and interfaces. We'll make our code look a bit like Java, since the Java community can be quite keen on layers (see Listing 1).

Each layer of this code works at its own layer of abstraction, and does a single well-focussed job. Each layer consumes objects of the layer below. We know that there will be many different fonts to choose from, so we ensure the details of font rendering are abstracted behind an interface.

When we come to consider tests for this code, we will find we need to introduce some more seams [Feathers], where we can insert mocks: now `ComicSansFontCalculator` will take an `IHtmlParser` in its constructor, so we can test it without needing to instantiate the real parser.

With this in place, we have a layered architecture similar to what many of us work with day-to-day. Notice that each time we want to test any layer, we need to write mocks for the layer below. Each layer of the system is dependent on the details of the layer below it. When a system becomes much more complex than our example, even if each layer is well-defined

```
class ParsedHtml {
  List<Paragraph> paragraphs()
}

class HtmlParser {
  // Argument naming in honour of [Hilton]
  HtmlParser( String data )
  ParsedHtml parse()
}

interface IFontCalculator {
  void setFontSize()
  void setFontFamily()
  List<Bitmap> render()
}

class ComicSansFontCalculator {
  ComicSansFontCalculator( HtmlParser htmlParser )
  void setFontSize()
  void setFontFamily()
  List<Bitmap> render()
}

class LayoutManager {
  LayoutManager( IFontCalculator fontCalculator,
                 int pageWidth )
  Bitmap layOut()
}
```

Listing 1

Enterprise forwarding aside, it is generally agreed that a particular piece of code should be written at a single layer of abstraction

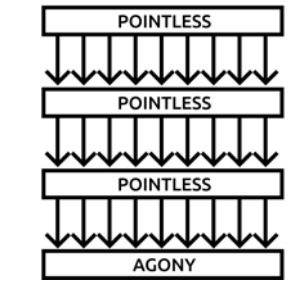

Figure 1

and kept to its own level of abstraction, the coupling between layers can become extremely complex and wide-ranging.

Coupling between parts of our code that ought to be separate is bad. When we define 'layers' we have good motivations: we aim to simplify our code, making each layer deal with a single set of concepts (or 'layer of abstraction'). However, often when we define layers we are really specifying a complex coupling between two separate areas of code: although the internals of a layer may be simple, the interfaces *between* layers are wide and complex, with many moving parts. Layers, like sheets of paper, are wide and flat. When sheets of paper are stacked on top of each other, adjacent sheets touch each other in lots of places.

Enterprise forwarding

We will start with the easy part, by arguing against a common layering technique: what I will call 'enterprise forwarding'. In our example, it might look like Listing 2.

And so on and so on for each object in the system. By repeating ourselves many times in different languages, we eventually achieve that pinnacle of layering: a multi-tiered system (see Figure 1).

But let's move on: everyone agrees that layer upon layer of identical method-declaration code alternating between Java and XML is horribly, horribly wrong. The only thing that could make things worse would be if the code did absolutely nothing at all until it was hooked up by some opaque, undebuggable blob of magic XML. But we would never do that.

```
class HtmlParser {
  HtmlParser( String data )
  ParsedHtml parse()
}

<layerDefinition>
  <object class="HtmlParser">
    <constructor args="String" argNames="data"/>
    <method name="parse" valueType="ParsedHtml"
            args="" argNames=""/>
  </object>
<layerDefinition>
```

Listing 2

Onion skins

Enterprise forwarding aside, it is generally agreed that a particular piece of code should be written at a single layer of abstraction [Henney-1], and we see that certain areas of our code operate at the same level as each other, so we may find ourselves defining layers like onion skins, each building on the one beneath. If we test the outer layers together with the inner ones we may be Classical TDDers (Figure 2) and if we write mocks to go under each layer we may be Mockist TDDers [Fowler-1] (Figure 3).

Whether classical or mockist in style, the onion skin approach leads us towards having wide and complex interfaces between parts ('layers') of our program. In our example, `IFontCalculator` classes depend on `HtmlParser`, and `LayoutManager` depends on `IFontCalculator`.

If we choose to isolate each layer during unit testing, we must write the kinds of complex mocks described in the introduction. Finding ways to keep our mocks simple enough that we can be confident they are not simply a second copy of the code under test becomes increasingly difficult.

Instead of onion skins, we should strive to write small, genuinely self-contained units of code, that interact with other parts via simple, narrow interfaces. We will see some techniques and examples to help us with this later.

Some other things that are bad

As a side note, it's worth saying that we are touching on some explanations for why many people are beginning to view anything described as a 'framework' with caution.

A framework is itself a layer (or series of layers) that surrounds your code, requiring you to plug your code into predefined slots. Often this means your code can't be used outside the framework (possibly even in tests), and can't work in a straightforward way, as it would if it were written as an independent module.

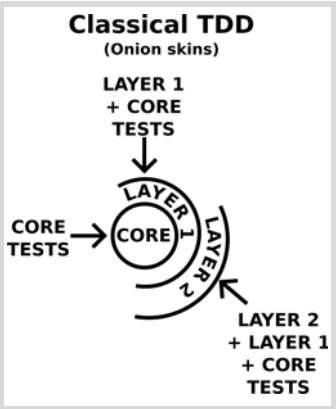

Figure 2

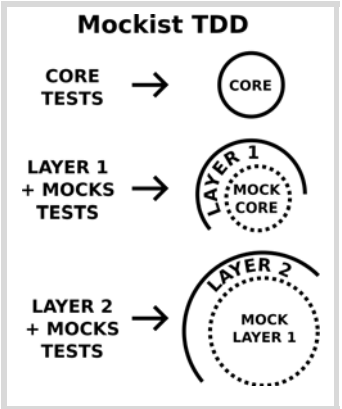

Figure 3

If we are able to **stick to types provided** by the programming language we are working in then we completely **eliminate dependencies** between different areas of our program

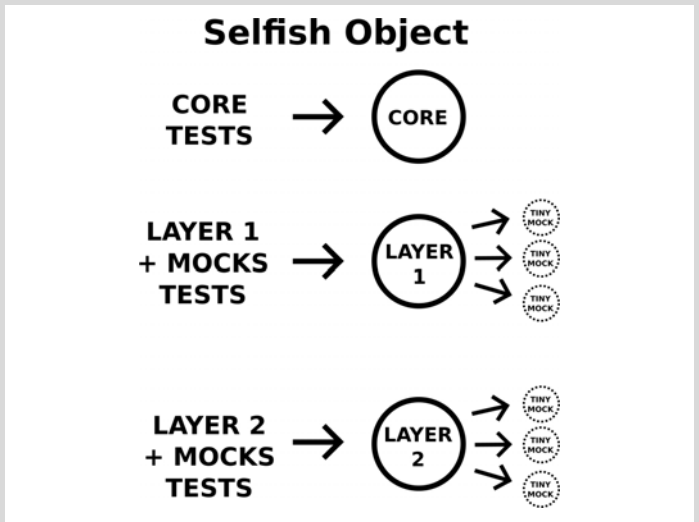

Figure 4

Similarly, a complex inheritance hierarchy is precisely an example of the kind of layering that can cause problems: the well-motivated desire to keep coherent units of code together has accidentally pushed us towards complex and subtle interactions between these units, so that while they look simple (because each source code file is small), they are actually highly coupled with many other units higher and lower in the inheritance chain.

Techniques for removing layering, or 'some things that are good'

If we agree that layers should be avoided, we must find techniques and structures that allow us to escape them, without sacrificing testability or coherence of our code.

The SELFISH OBJECT [Henney-2] pattern is a powerful tool in our quest. Whereas layers lead us to wrap each implementation class in an interface that exactly reflects it, SELFISH OBJECT encourages us to build interfaces from the point of view of the classes using them, making them tightly focussed on the job the object is being used for in that context, rather than the full functionality of the underlying class (see Figure 4).

SELFISH OBJECT encourages us to look 'selfishly' from the point of view of the object using the interface, ignoring parts that are irrelevant to it. This can lead us to make smaller interfaces that reflect a single aspect of an object's role, and lead us away from copying an object's full set of methods into a single interface it implements. It may well mean classes implement multiple interfaces if they are used in multiple ways. In some cases this may lead us further, towards breaking a class into smaller pieces, since we may realise the different ways it is used are actually different responsibilities.

For example, our `LayoutManager` class may have no interest in changing font sizes, so it may be able to use a reduced

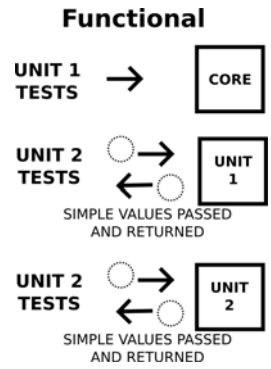

Figure 5

`IFontCalculator` interface. While we're at it, maybe we could rename it to `BlockRenderer` since the `LayoutManager` has no interest in whether the `Bitmap`s being dealt with originated from letters or anything else:

```
interface BlockRenderer {
  List<Bitmap> render()
}
```

If other classes deal with font calculators, they may well have other needs. In those cases, separate interfaces could be provided, also implemented by e.g. `ComicSansFontCalculator`.

A powerful technique for avoiding complex mocks and layering is REFACTOR TO FUNCTIONAL (see e.g. [Balaam]), which involves restructuring code so that the core logic exists in free functions with no state. These functions operate on simple, easily-constructed value-typed objects, meaning that unit tests no longer have complex set-up costs (see Figure 5).

For example, our `HtmlParser` class may not require any internal state, meaning we can refactor it to look like this:

```
class HtmlParser {
  static ParsedHtml parse( String data )
}
```

Some Java developers may look at you oddly if you suggest a static method, having been burned in the past by global static state in their enterprise code. If this happens, it is important to emphasise the difference between mutable static state (which is another name for a global variable), and a stateless static method (which is another name for a 'pure' function). The former is to be avoided at all costs, and the latter is one of the simplest, most predictable and most testable structures in programming [Oldwood].

Taking these ideas further, we can simplify the interactions between our classes, and reduce the need for complex mocks by ensuring we pass only simple types as parameters and return values of methods. If we are able to

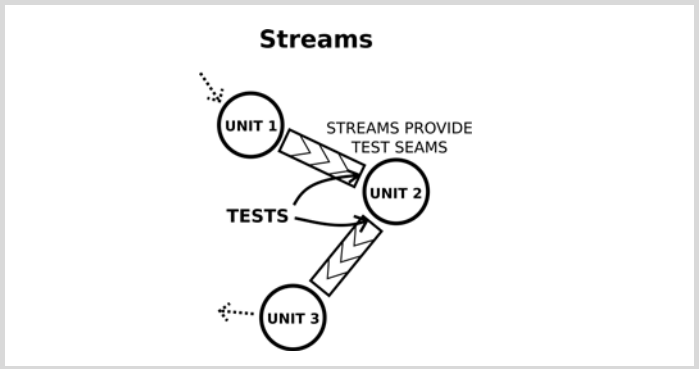

Figure 6

stick to types provided by the programming language we are working in (such as strings, numbers, lists, structs or tuples) then we completely eliminate dependencies between different areas of our program.

Obviously, this technique, which we will call TALK IN FUNDAMENTALS, can be taken too far. Classes that are conceptually close, and within the same level of abstraction, should pass classes and interfaces between them that allow rich communication, and do not require us to deconstruct and reconstruct objects that could simply have been passed untouched. Particularly, when code at one level is being used to build a 'language' that is then 'spoken' by code at a higher level (see e.g. [SICP]), TALK IN FUNDAMENTALS is certainly not appropriate, since the building block classes actually are the fundamental types being used by the higher level.

However, many interactions between classes can be expressed using fundamental types without any loss of expressiveness. For example, `LayoutManager` need not consume a `BlockRenderer` (or `IFontCalculator`), but simply a list of blocks to be rendered. If we also use REFACTOR TO FUNCTIONAL as well, we might have something like this:

```
class LayoutManager {
  static Bitmap layOut( List<Bitmap> blocks )
}
```

leading to the counter-intuitive conclusion that we can reduce coupling between two classes by *removing* an interface. `BlockRenderer` (or `IFontCalculator`) is not needed any more.

The further apart conceptually two communicating classes are, the more compelling is the case for using only simple types in their communication.

These techniques break our code into smaller independent units. We build libraries instead of frameworks, and functions instead of classes. We choose composition instead of inheritance, and we simplify the means of communication between distant pieces of code so that there is no dependency at all between them.

The Unix philosophy (of course)

If all of this sounds familiar, that's because it is mostly a re-expression of the UNIX PHILOSOPHY pattern [Unix] (we'll follow it by the word 'pattern' either to annoy the hackers, or to make it sound official to the enterprise programmers). In Unix, code is decomposed into small, stateless functions called 'programs' which interact through very simple fundamental types called 'streams'.

Modern programming languages allow very flexible and type-safe streaming approaches using iterators, meaning that we can write our code to be agnostic not just to what other parts of the code are doing but also when they are doing it. We can consume our input as it arrives, and stream it to other consumers in the same way, potentially enabling different parts of the system to work concurrently (see Figure 6).

For more on coding in a streaming style, see Section 3.5 of [SICP] and [Fowler-2].

If we apply everything we've learnt so far, we can refactor our example again, (leaving out class names since they are now noise):

```
static Iterator<Paragraph> parseHtml(
  InputStream text )

static Iterator<Bitmap> comicSans(
  Iterator<Paragraph> html )

static Bitmap layOut(
  Iterator<Bitmap> blocks, int pageWidth )
```

Some aspects of what we've ended up with may be distasteful (for example, representing parsed HTML as `Iterable<Paragraph>`makes me feel a little uneasy), but there is no doubt that we have ended up with three wholly independent units of code that are easily tested and re-used, have no interdependencies, and may in principle execute in parallel.

Once we start taking the Unix philosophy seriously, the key consideration is how to make the units of code we write composable. To do this we need to find a shape for our primitives that can be composed via a common operation. In Unix the primitives are programs and composition is via text streams. We have some other examples of composable structures (see everything ever written in Lisp, and e.g. [Freeman]), but it seems we still have much to learn about how to achieve composability in mainstream programming languages.

Conclusion

We can avoid mocks by avoiding layers and building independent, composable units in our programs, as in the Unix Philosophy.

While the Unix philosophy is always appealing, it is sometimes hard to see how to apply it outside of Unix. The techniques discussed above may help to break our code into independent blocks, rather than interdependent layers.

So remember: if unit testing is becoming painful, don't mock – decompose. ■

References

[Balaam] Andy Balaam 'Avoid Mocks by Refactoring to Functional' (http://www.artificialworlds.net/blog/2014/04/11/avoid-mocks-by-refactoring-to-functional/)

[Feathers] Michael Feathers 'Working Effectively with Legacy Code' (http://www.informit.com/store/working-effectively-with-legacy-code-9780131177055)

[Fowler-1] Martin Fowler 'Mocks Aren't Stubs' (http://martinfowler.com/articles/mocksArentStubs.html)

[Fowler-2] Martin Fowler 'Collection Pipeline' (http://martinfowler.com/articles/collection-pipeline/)

[Freeman] Steve Freeman and Nat Pryce 'Building SOLID Foundations' (http://www.infoq.com/presentations/design-principles-code-structures)

[Henney-1] Kevlin Henney 'How to Write a Method' (https://vimeo.com/74316116)

[Henney-2] Kevlin Henney, 'The Selfish Object' (http://accu.org/content/conf2008/Henney-The%20Selfish%20Object.pdf)

[Hilton] Peter Hilton 'How to name things' (http://hilton.org.uk/presentations/naming)

[Oldwood] Chris Oldwood (2014) 'Static – A Force for Good and Evil' in F. Buontempo, editor, *Overload* 120 (http://accu.org/index.php/journals/1900)

[SICP] Abelson, Sussman and Sussman 'Structure and Interpretation of Computer Programs' (http://mitpress.mit.edu/sicp/)

[Unix] Wikipedia 'The Unix Philosophy' (https://en.wikipedia.org/wiki/Unix_philosophy)

Non-Superfluous People: Architects

No developer is an island. Sergey Ignatchenko continues his series of non-superfluous people by looking at the role of architects.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please 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.

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

This article continues a mini-series on the people who're often seen as 'superfluous' either by management or by developers (and often by both); this includes, but is not limited to, such people as testers, UX (User eXperience) specialists, and BA (Business Analysts). However, in practice, these people are very useful – that is, if you can find a good person for the job (which admittedly can be difficult). The first article in the mini-series was about testers; the second article was about User eXperience specialists. This third article takes an unexpected twist – and talks about architects.

WTF – Is there ANYBODY out there thinking that architects are superfluous?

Honestly, I didn't plan to argue about architects in the Non-Superfluous People mini-series. For me, it goes without saying that you do need an architect, and I assumed that everybody else shares this understanding. After all, I am an architect myself ☺ . However, Mother Nature has once again demonstrated that assumption is indeed the mother of all mess-ups.

A few weeks ago, I wrote a blog post presenting my understanding of the qualities which are necessary to become a software architect [NoBugs15]. Actually, it was more about psychological obstacles which senior developers face when they need to become one, but that's not the point. The post was discussed on Reddit [Reddit]. To my surprise, the very first comment was about Software Architects being 'superfluous'; moreover, the comment was upvoted. This was the point when I realized that there are indeed people out there who think that architects are superfluous, and decided to include Architects to the series on Non-Superfluous People.

Software architects in successful projects

Actually, if you take a look at successful projects, you'll notice that each and every of them has an architect. This might be an official title, or it might be a de-facto architect, or it can even be a collective architect (i.e. several people performing this function) – but it exists. If we take a look at widely-known open source projects, we'll see that each and every of them has an architect. In some cases (like with Linux or Python) the architect is very obvious, in some other cases (such as the httpd Apache [daemon] project, where the architect is a de-facto collective architect) it is a bit more complicated, but in any case there is somebody or some body (pun intended) making decisions about project architecture, and the rest of the people in the project comply with these decisions. There are two common reasons why non-architects comply with architects' decisions. The first one is because architects manage to convince non-architects that their decisions are right; this is an ideal situation (and from my experience, it is possible to achieve it for 90–95% of decisions). While it takes time on the architect's side, it both avoids confusion due to miscommunication, and keeps the team spirit. The second reason to comply is because of a formal 'chain of command'; while it is significantly worse than complying due to the first scenario above, the difference between these two scenarios is really minor compared to the difference between complying and not complying. The range of scenarios can be represented as a table:

Approach	Results
Complying with Architects' decisions because of being convinced	The best
Complying with Architects' decisions because of formal 'Chain of Command'	A bit worse, but still workable
Not Complying with Architects' decisions	Disastrous
Not having an Architect	Very unlikely (see below), but disastrous

Can you really avoid having a de-facto architect?

Let's take a bit closer look at the project mechanics. If your project already has a formal 'Chain of Command', most likely you do have a formal architect (whether she's good or bad is a different story). If your project is an Open Source project which was started by one developer and the others joined when version 1.0 was already out, you have at least a de-facto architect. But what happens if you're in an Open Source project with 5+ people working together who started more or less at the same time? Almost universally, it is the same pattern – at some point (for the project's sake, it should happen sooner rather than later) the most influential developers come together and try to come to an agreement about the most important (whatever that means) issues for the project. If they aren't able to reach agreement, the project is pretty much dead. At the very best, you have two fork projects wasting resources and causing confusion, which is usually disastrous at these early stages of a project. If they are able to agree – Bingo! At this point you've got a collective de-facto architect.

In fact, in my whole life I've never seen a project without some kind of architect. While I won't claim that this observation is 100% universal, 99% is also quite a good number for our purposes.

'No Bugs' Hare 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 currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

the reasons behind rebellion are not as important as the strategy which the 'rebel' pursues

If you are bound to have an architect, then why I am arguing for having one?

Those experienced with formal logic, may say:

- "You've just argued that every software project is pretty much bound to have some kind of the architect"
- "Then, what's the point of arguing that we should have one? As you have already said – we'll get one pretty much regardless of what we're doing"

This logic is perfectly correct. However, I'm not arguing that you should have an architect (as noted above, it will happen regardless). My real point is a bit different – while you will have an architect pretty much no matter what, you SHOULD comply with her decisions.

Whatever you say, Dude!

Therefore, the real question is not whether you have an architect (you will), but whether you comply with the architect's decisions. In most cases, this is not a problem. However, the chances are that you have at least one member of your team (let's call him a 'rebel') who doesn't think that discipline is a good thing, or just doesn't have any respect for the architect guy. The reasons for rebellious behavior vary, and are not necessarily selfish; in particular, two common reasons for rebellion are 'I would do it much better' (and it doesn't matter if it is really the case) and 'What you have said just makes my life more difficult'.

However, the reasons behind rebellion are not as important as the strategy which the 'rebel' pursues. The first category of 'rebels' which I've seen is 'open rebels'. They're usually very loud and outspoken. The impact of such an open rebellion is usually not too bad, and necessary solutions (assuming that both architect and 'rebel' are guided by interests of the project, and are ready to listen to the arguments) can usually be found without ruining the project.

Rebels from the second category, 'underground rebels', are much more dangerous. If the 'rebel' doesn't openly challenge the architect's decisions, but instead doesn't comply with them – then you may have a real problem.

Coming from theorizing back to Earth, consider just one rather typical example. There is a decision that all the code should be cross-platform. It is understood that writing cross-platform code takes additional time, but it is considered to be worth the trouble. However, one of the team members strongly disagrees with it. From now on, he may either raise an open rebellion, questioning "Who was the Smart Guy who decided to spend time on this nonsense?", or may start pushing non-cross-platform code quietly. In the case of open rebellion, at least the team knows that there is a disagreement, and has the option to discuss it (hopefully coming to some resolution).

In the case of 'underground rebellion' which is not noticed soon enough, the things tend to be much worse. First, you as a team may end up with code which is not cross-platform (while you were 100% sure it is). Second, this tendency may be worsened by the fact that other (non-rebel and non-architect) team members may start using the existing code as a reference of 'how to do things' (or just copy-paste), which tends to proliferate non-cross-platform code across the whole code base. While it is newer team members who're the most susceptible to this copy-paste effect, even architects, when their mind is on the other things, were observed to copy-paste code which is against their own decisions/guidelines.

What to specify?

Ok, we've established that it is important to have a consistent vision for the project, and to comply with it. But what should be included in this vision? In other words – what must be common across the whole project, and what can be left to case-by-case decisions by individual developers? While there is no one single answer on what to include in guidelines, there are some observations:

- all deviations from common-practices-out-there MUST be explicit
- all prohibitions on which-parts-of-the-language-we-dont-use MUST be explicit. For example, if there is a decision not to use C++ `iostream` in the project (IMHO a wise decision for a pretty much any project out there, but this is outside the scope now), it MUST be explicitly written somewhere
- all not-so-common requirements (i.e. requirements which might be present, or might not be present in other projects of similar scope) MUST be explicitly stated. For example, if the code is intended to be cross-platform, it MUST be explicitly stated
- things such as naming conventions are usually useful (i.e. time spent on following them is worth the trouble)
- conventions 'how exactly to place curly brackets' (for Python an equivalent is 'how many spaces to use') are usually not

On job titles

Ok, ok, but what about job titles? Is it important whether an architect is called an architect or not? From my experience, the answer is 'it doesn't matter'. The whole thing is not about titles, it is about getting the job done.

On non-coding architects

Pretty often it happens that an architect starts as one of the coders, but then code reviews, processing pull requests, writing specs, discussions of issues with rebels, answering 'how we should do this?' questions etc. etc. start to eat too much time, and the architect starts to write less and less code. Whether it is normal or is a Bad Thing is an open question. From my personal experience, it is more or less bound to happen, at least if you're a sole architect; however, it is always a good idea from time to time to take a limited-size-but-internally-very-complicated project and code it yourself (both because nobody else will be able to do it and to show the others that you still can code at least on par with the very best of them).

On enterprise architects

It should be noted that for the purposes of this article, we didn't make any distinction between Project Architects and Enterprise Architects. However, while we've discussed things mostly from the point of view of

the Project Architect, the very same logic will apply if we consider the whole Enterprise as a larger project (which is not strictly correct in general, but won't affect our reasoning above). In short: you need both Project Architects and Enterprise Architects – all for the reasons described above.

Conclusions

1. You will have an architect (formal, informal, collective...) pretty much no matter what you're doing.
2. What is important is to comply with architect's decisions.
3. If you disagree with architect's decisions, challenge them openly!
4. If you're an architect, it is a part of your job to make sure that your decisions are complied with. BTW, recent project management tendencies (such as moving towards github with its 'pull requests' compared to giving direct access to the repository) seem to provide better support for ensuring compliance.
5. What to include into decisions/vision/guidelines is subjective, and there is a balance between specifying too much and specifying too little; some observations in this regard are listed above.
6. Issues such as 'what job title should an architect have' and 'whether the architect still codes' are pretty much immaterial. ■

References

[daemon] http://httpd.apache.org/docs/2.2/programs/httpd.html

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

[NoBugs15] http://ithare.com/7-prerequisites-to-become-a-software-architect/

[Reddit] http://www.reddit.com/r/programming/comments/32jeka/prerequisites_to_become_a_software_architect/

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

Terse Exception Messages

Log files often contain 'error' information. Chris Oldwood suggests they rarely contain anything that could be considered helpful.

Users are not the only people who have to deal with cryptic error messages; sadly support staff and developers can have an equally bad time too. Users have every right to complain as they are rarely able to do anything about it, whereas we programmers have it within our power to write better diagnostic messages to aid our colleagues, and in turn better serve our users by helping resolve their issues more quickly.

This article looks at some of the techniques we can apply when writing code that is going to throw an exception in response to some unexpected condition. Whilst exceptions are clearly not the only style of error reporting available, I have found that due to the 'distance' that often exists between the **throw** and **catch** sites it is all the more important that thought be put into the message generated. I also have my suspicions that their very name 'exceptions' lures the unwary programmer into believing they will only rarely occur and so will demand far less attention than other diagnostic messages.

A rude awakening

One 'very early' Saturday morning, during my week on the support rota, I got a phone call from the 1st line support team to say that one of the weekend tasks had failed. They had kindly sent me the relevant portion of the service log file and the error said something along these lines:

```
ERROR: Failed to calibrate currency pair!
```

The task was a fairly lengthy one (taking over an hour) and there were no obvious clues as to which particular currency pair (of the non-trivial number being processed) that might be the source of the problem. I decided my only recourse was to set a breakpoint on the site of the thrown error and then wait for it to fail. When I located the line of code where the error was generated I was more than slightly annoyed to see it was written something like this:

```
void calibrate(string ccy1, string ccy2, . . .)
{
  . . .
  if (. . .)
    throw Error("Failed to calibrate currency
    pair!");
}
```

That's right, the throw site knew what the exact combination of currency pairs were, but had neglected to include them in the error message. Needless to say once I got back in the office on Monday morning, the first thing I did was to fix the error message to save someone else wasting time unnecessarily in the future, and to allow the support team to do a bit more diagnosis themselves.

Clearer exception messages

The examples that provided the impetus for this article are equally terse:

```
throw new Exception(String.Empty);
throw new NotFoundException("Invalid ID");
```

One hopes that the first example is simply an unfortunate mistake where the programmer truly intended to come back and write something useful, but clearly forgot. The second is better, but I would contend that it's only marginally better because there is so much more than could be said by the message.

If we imagine the message in the context of a busy log file, rather than surrounded by its related code, it will likely appear with only the usual thread context information to back it up, e.g.

```
[date & time, thread, etc] ERROR: Invalid ID
```

If you also have a stack trace to work from that might at least give you a clue as to what type the 'ID' refers to. You may also be able to hunt around and find an earlier diagnostic message from the same thread that could give you the value too, but you're probably going to have to work to find it.

Consequently at a bare minimum I would also include the 'logical' type and, if possible (and it usually is), then the offending value too. As such what I would have thrown is something more like this:

```
throw new NotFoundException
  ("Invalid customer ID '{0}'", id);
```

This message now contains the two key pieces of information that I know are already available at the throw site and so would generate a more support-friendly message such as this:

```
[...] ERROR: Invalid customer ID '1234-5678'
```

Discerning empty values

You'll notice that the value, which in this case was a string value, is surrounded by quotes. This is important in a message where it might not be obvious that the value is empty. For example if I had not added the quotes and the customer ID was blank (a good indication of what the problem might be) then it would have looked thus:

```
ERROR: Invalid customer ID
```

Unless you know intimately what all your log messages look like, you could easily be mistaken for believing that the error message contains little more than our first example, when in fact the value is in fact there too, albeit invisibly. In the improved case you would see this:

```
ERROR: Invalid customer ID ''
```

For strings there is another possible source of problems that can be hard to spot when there are no delimiters around the value, and that is leading and trailing whitespace. Even when using a fixed width font for viewing log files it can be tricky to spot a single erroneous extra space – adding the quotes makes that a little easier to see:

```
ERROR: Invalid customer ID '1234-5678 '
```

As an aside my personal preference for using single quotes around the value probably stems from many years working with SQL. I've lifted values out of log files only to paste them into a variety of SQL queries

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

Whilst it's perhaps understandable why the more generic methods are suitably terse, sadly the same also goes for the more focused string parsing methods too

countless times and so automatically including the right sort of quotes seemed a natural thing to do.

Discerning incorrect values

It's probably obvious but another reason to include the value is to discern the difference between a value that is badly formed, and could therefore never work (i.e. a logic error), and one that is only temporarily unusable (i.e. a runtime error). For example, if I saw the following error I would initially assume that the client has passed the wrong value in the wrong argument or that there is an internal logic bug which has caused the value to become 'corrupted':

```
ERROR: Invalid customer ID '10p off baked beans'
```

If I wanted to get a little more nit-picky about this error message I would stay clear of the word 'invalid' as it is a somewhat overloaded term, much like 'error'. Unless you have no validation at all for a value, there are usually two kinds of errors you could generate – one during parsing and another during range validation. This is most commonly seen with datetimes where you might only accept ISO-8601 formatted values which, once parsed, could be range-checked, for example, to ensure that an end date does not precede a start date.

To me the kind of error below would still be a terse message, better than some of our earlier ones, but could be even better:

```
ERROR: Invalid datetime '01/01/2001 10:12 am'
```

My preferred term for values that fail 'structural validation' was adopted from XML parsing – malformed. A value that parses correctly is considered 'well formed' whereas the converse would be one that is 'malformed'. Hence I would have thrown:

```
ERROR: Malformed datetime '01/01/2001 10:12 am'
```

For types like a datetime where there are many conventions I would also try and include the common name for the format or perhaps a template if the format is dynamically configurable so the caller doesn't have to root around in the specs to find out what it should have been:

```
ERROR: Malformed datetime '01/01/2001 10:12 am'
  (Only ISO-8601 supported)
ERROR: Malformed datetime '01/01/2001 10:12 am'
  (Must conform to 'YYYY-MM-DD HH:MM:SS')
```

Once a value has passed structural validation, if it then fails 'semantic validation' I would report that using an appropriate term, such as 'out of range' or better yet include the failed comparison. With semantic validation you often know more about the role the value is playing (e.g. it's a 'start' or an 'end' date) and so you can be more explicit in your message about what the problem is:

```
ERROR: The end date '2010-01-01' precedes the
start date '2014-02-03'
```

Terse framework messages

Framework exception messages are notoriously terse. The reason is probably that, like all general purpose library functions, there is a balance between performance, security, etc. and so including the value in the exception may well be highly undesirable in some types of application.

One of the best examples of brevity in the .Net framework is probably the one below which is thrown from the LINQ extensions, such as `Single()`:

```
Sequence contains no matching element
```

The prevalent use of LINQ in modern C# code is to be applauded over traditional `for-next` loops, but as their use grows, so does the possibility of this kind of message popping up when things go awry. Whilst it's a little extra code to write for the caller, one obvious choice is to use the non-throwing version `SingleOrDefault()`, so that you can detect the failure manually and throw a more suitable exception, e.g.

```
var customer =
  customers.SingleOrDefault(c =>c.Id == id);
if (customer == null)
{
  throw new NotFoundException("Failed to find
    customer with ID '{0}'", id);
}
```

As always there is a trade-off here. On the one hand I could write less code by directly using the LINQ methods provided, but at the cost of poorer diagnostics. However, as we shall see next, in C# there is nothing to stop me wrapping this extra code in my own extension method to keep the call site more readable.

Custom parsing methods

Whilst it's perhaps understandable why the more generic methods are suitably terse, sadly the same also goes for the more focused string parsing methods too, such as `int.Parse()`. On the kinds of systems I work with, I'd happily trade-off much better diagnostics for whatever little extra garbage this technique might incur, especially given that it's only (hopefully) in exceptional code paths.

My solution to this problem is the proverbial 'extra level of indirection', hence I wrap the underlying non-exception-throwing `TryParse()` style framework methods with something that produces a more satisfying diagnostic. As suggested a moment ago, extension methods in C# are a wonderful mechanism for doing that.

For example I tend to wrap the standard configuration mechanism with little methods that read a string value, try to parse it, and if that fails then include the setting name and value in the message. Consequently instead of the usual terse `FormatException` affair you get from the .Net framework when parsing an integer value with code like this:

```
int.Parse(settings["TimeoutInMs"]);
```

You invoke a helper method like this instead:

```
settings.ReadInt("TimeoutInMs");
```

And if this fails you'll get a much more useful error message:

```
ERROR: The 'TimeoutInMs' configuration setting
value '3.1415' was not a well formed integer value
```

This handles the structural validation side of things. For the semantic validation it can either be done with traditional conditional statements or by chaining on a validation method, a là fluent style:

```
settings.ReadInt("TimeoutInMs")
  .EnsuringValueIsPositive();
```

For common 'types' of settings this pair can itself be wrapped up further to once again simplify the call site:

```
settings.ReadTimeout("SendRequestTimeoutInMs");
```

Formatting exceptions in C#

Whenever I create a custom exception type in C#, I generally add a variety of overloads so you can create one directly with a message format and selection of arguments without forcing the caller to manually use `String.Format()`. This is how the example near the beginning worked:

```
throw new NotFoundException("Invalid customer ID
 '{0}'", id);
```

All the class needs for this kind of use is a signature akin to `String.Format`'s:

```
public NotFoundException(string format,
  params object[] args)
  : base(String.Format(format, args))
{ }
```

However this is not enough by itself – it's dangerous. If we pass a raw message that happens to contain formatting instructions but no arguments (e.g. `"Invalid list {0, 1, 2}"`) it will throw during the internal call to `String.Format()`, so we need to add a simple string overload as well:

```
public NotFoundException(string message)
  : base(message)
{ }
```

As an aside, I never add a public default constructor by default because I'm not convinced you should be allowed to throw an exception without providing some further information.

Due to the framework exception classes not having a variable arguments (var-args) style constructor you might already be happy putting formatting calls into the callee, either directly with `String.Format()` or via a simple extension method like `FormatWith()` [Newton-King], e.g.

```
throw new NotFoundException(String.Format
  ("Invalid customer ID '{0}'", id));
throw new NotFoundException
  ("Invalid customer ID '{0}'".FormatWith(id));
```

One scenario where the var-args style constructor falls down in C# is when you start adding overloads to capture any inner exception. You can't just add it on the end of the signature due to the final params-based parameter, so you have to add it to the beginning instead:

```
public NotFoundException(Exception inner,
  string format, params object[] args)
  : base(String.Format(format, args), inner)
{ }
```

Sadly, as you can see from the `base()` constructor call, this is the exact opposite of what the framework does; it expects them on the end because the preceding argument is always a simple string.

In my experience little effort is put in by developers to simulate faults and validate that exception handling is behaving correctly, i.e. verifying 'what' is reported and 'how'. Consequently this makes the var-args overload potentially dangerous as adding an exception argument on the end (by following the .Net framework convention) would cause it to be silently swallowed as no string format placeholder would reference it.

Whilst this has never been a problem for me in practice because I virtually always throw custom exception types and nearly always end up formatting a non-trivial diagnostic message, that's easy to say when you're the one who discovers and instigates a pattern – others are more likely to follow the one they're used to, which is almost certainly the way the .Net framework does it.

Testing exception messages

At the tail end of the previous section I touched briefly on what I think is at the heart of why (server-side) error messages can often be poor – a lack of testing to make sure that they satisfy the main 'requirement', which is to be helpful in diagnosing a problem. When unit tests are written for error scenarios it's all too common to see something simplistic like this:

```
Assert.That(() => o.DoIt(...), Throws.Exception);
```

This documents and verifies very little about what is expected to happen. In fact if the `o` variable is a null reference, the test will still pass and not even invoke the behaviour we want to exercise! Our expectations should be higher; but of course at the same time we don't want to over-specify the behaviour and make the test brittle instead.

If a caller is going to attempt recovery of the failure then they at least need to know what the type of the exception is, lest they be forced to resort to 'message scraping' to infer its type. Consequentially it's beneficial to verify that the exception is of at least a certain type (i.e. it may also be a derived type). Where an obvious value, such as an input argument, is a factor in the error we can include that as a real property if we believe it may be useful for recovery. If the exception type is likely to remain vague we can still loosely test that the value is contained somewhere within the message instead:

```
const string id = "malformed-value";
Assert.That(() => o.DoIt(id),
  Throws.InstanceOf<ValidationException>()
  .And.Message.Contains(id));
```

If we're writing tests for a method where there are many arguments to be validated, it's more important that we try and match on the message too (or some richer-typed property) to discern which of the many arguments caused the exception to be thrown. It's all too easy when making a change to a method that you end up finding one or more of your tests are passing because they are detecting a different error to the one they were intended to, as the null reference example above highlights.

Whilst that takes care of our code-based contractual obligations we also have an obligation to those who will be consuming these messages to make sure they form a useful narrative within any diagnostic output. This is an article in its own right [Oldwood], but using a text editor that performs spell checking of string literals certainly goes a little way towards helping avoid silly mistakes.

Summary

If you're going to rely on exception messages as a major source of your diagnostic output, such as through the use of Big Outer Try Blocks [Longshaw] and classic log files, then it pays to make sure that what you're going to end up writing contains the information you really need. This article has provided a number of suggestions about how the content of these messages can be improved, along with some ideas about how you can help ensure the code doesn't get swamped with these tangential concerns so that the underlying logic becomes obscured. The final section looked at what we can do to leverage automated testing as a means of helping to formalise the errors that we generate as an aid to throwing 'better quality' exceptions. ■

Acknowledgements

Thanks as always to the *Overload* review team for sparing my blushes.

References

[Longshaw] 'The Generation, Management and Handling of Errors', Andy Longshaw, *Overload* #93, http://accu.org/index.php/journals/1586

[Newton-King] 'FormatWith Extension Method', James Newton-King, http://james.newtonking.com/archive/2008/03/27/formatwith-string-format-extension-method

[Oldwood] 'Diagnostic & Support User Interfaces', Chris Oldwood, http://chrisoldwood.blogspot.co.uk/2011/12/diagnostic-support-user-interfaces.html

Get Debugging Better!

The GNU debugger has several useful features you may not know. Jonathan Wakely shows us how to save time and pain with some simple tricks.

The GNU Debugger (GDB) is a powerful tool, but if you're used to working in an IDE then using a command-line debugger can be daunting and may seem to be lacking features you take for granted in an IDE. This article has some simple tips that might help you have a more pleasant debugging experience, and might inspire you to read the documentation [GDB] to see what other tricks are waiting to be discovered.

The first tip, and maybe the most important, is to make sure you're using a recent version. Support for debugging C++ code got much better with GDB 7.0 and has continued to improve since then. If you're using anything older than GDB 7.0 you should upgrade right away, and it's probably worth upgrading anything older than a couple of releases (GDB 7.8.2 and 7.9 were both released in early 2015). If you can't get a pre-built version for your OS then compiling GDB from source is very easy, just download the tarball, unpack it, run configure (setting your preferred install directory with **`--prefix=dir`**) and run **make**.

One of the simplest GDB features, but one I always miss when using the venerable 'dbx' debugger on 'proper' UNIX machines, is the ability to use abbreviations for commands. Any unambiguous prefix for a command will run the full command, so instead of typing **`print foo`** you only need **`p foo`** and instead of **`break source.cc:325`** just **`br source.cc:325`** (and while not strictly a prefix of the full command, **`bt`** will print a stack trace just like **`backtrace`**).

You can also very easily create your own commands by defining them in your personal `~/.gdbinit` file, which gets run by GDB on startup. I use the following to quit without being asked to confirm that I want to kill the process being debugged:

```
define qquit
  set confirm off
  quit
end
document qquit
Quit without asking for confirmation.
end
```

This allows me to use **`qquit`** (or just **`qq`**) to exit quickly. The **`document`** section provides the documentation that will be printed if you type **`help qq`** at the gdb prompt.

Sometimes stepping through C++ code can be very tedious if the program keeps stepping into tiny inline functions that don't do anything interesting. This is very obvious in C++11 code that makes heavy use of **`std::move`** and **`std::forward`**, both functions that do nothing except cast a variable to the right kind of reference. The solution is to tell gdb not to bother stepping into uninteresting functions (or ones that get called a lot but which you know are not the source of the problem you're debugging). Running the **`skip`** command with no arguments will cause the current function to be skipped over next time it is reached, instead of stepping into it. You can also use **`skip FUNCTION`** to cause a named function to be skipped instead of the current one, or **`skip file FILENAME`** to skip whole files (as with most commands, run **`help skip`** at the gdb prompt for more information). Unfortunately the **`skip`** command treats every specialization of a function template as a separate function and there's no way to skip over all specializations of say, **`std::move`**, but if you skip each one as you step into it at least you know you won't step into that particular specialization again. I define the following command in my `.gdbinit` to make this even easier:

```
define skipfin
  dont-repeat
  skip
  finish
end
document skipfin
Return from the current function and skip over
all future calls to it.
end
```

This lets me use **`skipfin`** to mark the current function to be skipped in future and to finish running it and return to the caller. The **`dont-repeat`** line tells gdb that (unlike most built-in commands), hitting enter again after running **`skipfin`** should not run **`skipfin`** again, so that I don't accidentally finish running the caller and mark that to be skipped as well!

Another useful entry in my `.gdbinit` is **`set history save`**, which causes gdb to save your command history when exiting, so you can use the cursor keys to scroll through your history and easily create the same breakpoints or watchpoints as you used in an earlier debugging session.

The GDB feature that I most wish I'd known about sooner is the 'TUI' mode, which is activated by the **`-tui`** command-line option, or can be turned on and off in an existing debugging session with Ctrl-X Ctrl-A [TUI]. This splits the terminal window horizontally, with the bottom pane showing the usual prompt where you type commands and get output, and the top pane showing the source code for the function being debugged, just like your IDE would. This gives you a much more immediate view of the code than using **`list`** to print out chunks of it. One thing to be aware of is that the TUI mode changes the behaviour of the cursor keys, so they scroll up and down in the source code rather than through your command history. If you're familiar with them from the terminal, you can still use readline key bindings (Ctrl-P, Ctrl-N etc.) to scroll through the command history.

After **`-tui`**, the command-line option I most often use when starting gdb is **`--args`**. This can be used to start debugging a program with a specific set of arguments, so instead of running **`gdb ./a.out`** and then setting arguments for it with **`set args a b c`** then running it with **`run`**, you can start gdb as **`gdb --args ./a.out a b c`** and then just **`run`**. This is very useful when the program needs a long and complicated set of arguments, as you don't need to find them and copy & paste them into gdb, just add **`gdb --args`** before the usual command to run the program.

Jonathan Wakely Jonathan's interest in C++ and free software began at university and led to working in the tools team at Red Hat, via the market research and financial sectors. He works on GCC's C++ Standard Library and participates in the C++ standards committee. He can be reached at accu@kayari.org

One of the most useful features of modern version of GDB is the embedded Python interpreter. This allows you to write pretty printers for your own types

One of the most useful features of modern version of GDB is the embedded Python interpreter. This allows you to write pretty printers for your own types (or use the ones that come with GCC for printing standard library types such as containers). Defining pretty printers for the types in your system can be very useful, and although it's not too complicated there isn't room to explain here, however the embedded Python interpreter is also very useful for running simple one-liners without leaving gdb. For example if you have a `time_t` variable containing a unix timestamp you can easily print it using Python's **datetime** module:

```
(gdb) python import datetime
(gdb) python print
  datetime.datetime.fromtimestamp(1425690208)
2015-03-07 01:03:28
```

If what you want to do isn't suitable for a one-liner you can create multiple-line blocks of Python by entering just `python` on a line on its own, and then end the block with `end`. Many of gdb's features are exposed via a python API ('import gdb') [Python] that lets you inspect variables, so you can examine their value, type, members etc.

None of these tips are groundbreaking, but I hope they give an idea of ways you can customise your debugging experience and define shortcuts to simplify the most repetitive tasks. ■

References

[GDB] https://sourceware.org/gdb/onlinedocs/gdb/index.html#Top

[Python] https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html#Python-API

[TUI] https://sourceware.org/gdb/onlinedocs/gdb/TUI.html

Make and Forward Consultables and Delegates

Sometimes forwarding to a contained object requires lots of boilerplate code. Nicolas Bouillot introduces consultables and delegates to automate this.

Separation of functionalities, code reuse, code maintainability: all these concerns seem critical. When writing a C++ class, *selective* delegation and inheritance are tools enabling both separation of functionality implementation and code reuse. Before introducing the making and forwarding of Consultable and Delegate, let us review some points that make selective delegation and inheritance two very different tools in the hand of the programmer. In particular, it will be explained why they might be seen as poorly scaling tools with increase of code base, and why the new approach *Consultables* and *Delegates* presented here scales better.

With public inheritance, public methods from the base class are made available directly through the invocation of the inherited methods. While this makes available an additional functionality without extra code, inheriting from multiple functionalities need cleverly written base classes in order to avoid members and methods collisions. As a result the more base classes are inherited from, the more collisions are probable, including the probability of inheriting multiple times from the same base class (the diamond problem) and related issues.

With traditional C++ delegation, called here *selective delegation*, the added functionality is a private class member (the delegate), which can be accessed by the user thanks to wrappers in allowing to access the member's methods. This scales well with the number of added functionalities. One can even have two functionalities of the same type used as delegate, which cannot be achieved with inheritance. However, manually writing wrappers does not scale well with the increased use of delegation: once the signature of a method is changed in the delegate, the wrappers need to be manually updated.

Public inheritance and selective delegation allows access to delegates, but what about read/write access? With inheritance, all public methods, regardless of their constness are made available to the user. There is no option to give access to `const` only methods, and accordingly reserve the write access to the member for internal use while giving read access to the user. With selective delegation, the access rights to the functionality are given through wrapper, giving full control to the programmer who can do read/write access manual if desired. With the *Consultables* and *Delegates* presented here, templated wrappers are automatically generated according to the `const`ness, enabling or not the methods with the desired `const`ness.

Let us take an example. A class (delegator) is having a `std::vector` (delegate) for internal use (write access). However, it is wanted to give a read access to the user (delegator owner). In this case, the delegator inheriting from `std::vector` is not an option, and selective delegation requires the programmer wrap all `const` methods and operators for the user, or a subset (more probable) of methods. When wrapping manually, is it considered necessary to include access to `cbegin` and `cend`? Probably not if the user does not use them when writing the wrapper. With templated wrapper generation, no manual selection is required and wrappers are generated only if invoked by a user. As a result, no wrappers with errors, no wrapper forgetting and no need to rewrite wrappers if the contained type changes (moving from a `string` vector to an `int` vector).

Let us go further: what about forwarding delegation? The class (delegator) owning the delegate is used by an other class (delegator owner). With selective delegation, the wrapper needs to be rewritten in order to make initial delegate available to the user of the delegator owner. Once more, manual wrapper should probably be avoided.

The following is presenting the use and implementation[1] of *Consultable* and *Delegate*. It is built with C++11 template programming. As briefly said earlier, *Consultable* makes available all `public const` methods for safe access, and *Delegate* that is making all public delegate methods available. The benefits of this approach is 1) no specific wrappers, so less code to write and maintain, 2) a change in the delegate code becomes immediately available at the end caller, producing a more easily maintainable code base and 3) non-const method access of a delegate can be blocked in any forwarding class by forwarding the delegate as a Consultable.

Selective delegation vs. Consultables & Delegates

Brief history: the Gang of Four [GoF95] describes delegation as consisting of having an requested object that delegates operations to a delegate object it is owning. This method is comparable to subclassing where inheritance allows for a subclass to automatically re-uses code (no wrapper) in a base class. While delegation requires the implementation of wrappers, subclassing has been seen as providing poor scaling and tends to lead to code that is hard to read, and therefore hard to correct and maintain [Reenskaug07]. As a rule of thumb, Scott Meyers proposes to use inheritance only when objects have an 'is-a' relationship [Meyers05]. It is, however, tempting for programmers to choose inheritance instead of delegation since writing wrappers is a tedious task and can still be error-prone.

Delegation through automatic generation of wrappers for *selected* methods has been extensively proposed and implemented[2]. While this gives a full control of which method are available, this also requires specification of each delegated methods along the possible classes that delegate,

Nicolas Bouillot Nicolas is a research associate at the Society for Arts and Technology (SAT, Montreal, Canada). He likes C++ programming, team-based working, writing research papers, networks and distributed systems, data streaming, distributed music performances, teaching, audio signal processing and many other unrelated things.

1. Source code and examples are available at https://github.com/nicobou/cpp_make_consultable
2. (Accessed March 2015)
http://www.codeproject.com/Articles/11015/The-Impossibly-Fast-C-Delegates
http://www.codeproject.com/Articles/7150/Member-Function-Pointers-and-the-Fastest-Possible
http://www.codeproject.com/Articles/18886/A-new-way-to-implement-Delegate-in-C
http://www.codeproject.com/Articles/384572/Implementation-of-Delegates-in-Cplusplus11
http://www.codeproject.com/Articles/412968/ReflectionHelper

The goal of the approach proposed here is to avoid specifying the delegate methods

subdelegate, sub-subdelegate, etc. This may result in laborious specification in multiple files, leading to duplication in the code base. In turn, safety remains in the hands of the developer who decides a non-`const` method can be exposed.

The goal of the approach proposed here is to avoid specifying the delegate methods, but instead makes available all or `const` only methods. This is achieved by making a class member `Consultable` or `Delegate` that is accessed through a programmer specified delegator method. The use of a Consultable – invoking a method of the delegate – by the requester is achieved by invoking the consult method, whose arguments are the delegate method followed by the arguments.

Consultable

Usage

Listing 1 shows how consultables are used. Two `Widget`s will be owned by a `WidgetOwner`. These two members are made consultable in `WidgetOwner` (lines 14 and 15) and will be accessed respectively with the `consult_first` and `consult_second` methods. `Make_consultable`, as explained in 'Implementation' below, is a macro that internally declares templated consult methods. Its arguments are the type of the delegate, a reference to the delegate member and the `consult` method.

In the main function, the WidgetOwner object `wo` is instantiated and the Widget const method `get_name` is invoked on both delegates (lines 24 and 25). Note the comment in line 28 showing an example of a use of `consult_first` with the non const `set_name` method, which does not compile.

Forwarding Consultable

As seen before, an object is made consultable and accessible through a consult method. Accordingly, a class composed of delegate owner(s) can access the delegate methods. However, this class does not have access to the reference of the original delegate, making impossible to apply `Make_consultable`. In this case, `Forward_consultable` allows selection of the consult method and *forwards* it to the user. A forwarded consultable can be re-forwarded, etc.

Listing 2 shows how it is used: `WidgetOwner` is making `first_` and `second_` consultables. Box is owning a `WidgetOwner` and forward access to the consultables using `Forward_consultable` (line 20 and 21). Then, a Box Owner can access the consultable through `fwd_first`, a `Box` method installed by the forward (line 28).

Overloads

The use of consultable requires the user to pass a pointer to the delegate method as argument. Accordingly, overloaded delegate members need more specification for use with consult methods. As shown in Listing 3, this is achieved giving return and argument(s) types as template parameters (line 14 and 15). Overload selection can also be achieved using static casting the member pointer (line 21). This is however more verbose and might not be appropriate for use.

Enabling setters

Although a *Consultable* makes available all `public const` methods, it may need to have a setter that can be safely used. For instance, *consultable*

Consultable declaration and use. This illustrates how `get_name` is accessed by a `WidgetOwner` without explicit declaration of it in the `WidgetOwner` class. At line 14, the macro `Make_consultable` is actually declaring several methods named `consult_first` that will make consultation available to the user.

```
 1 class Widget {
 2  public:
 3   Widget(const string &name): name_(name){}
 4   string get_name() const { return name_; }
 5   string hello(const string str) const {
       return "hello " + str;};
 6   void set_name(const string &name) {
       name_ = name; }
 7
 8  private:
 9    string name_{};
10 };
11
12 class WidgetOwner {
13  public:
14   Make_consultable(Widget, &first_,
       consult_first);
15   Make_consultable(Widget, &second_,
       consult_second);
16
17  private:
18   Widget first_{"first"};
19   Widget second_{"second"};
20 };
21
22 int main() {
23   WidgetOwner wo; // print:
24   cout << wo.consult_first(&Widget::get_name)
     // first
25        << wo.consult_second(&Widget::get_name)
          // second
26        << wo.consult_second(&Widget::hello,
          "you")  // hello you
27        << endl;
28   // compile time error (Widget::set_name
     is not const):
29   // wo.consult_first(&Widget::set_name,
     "third");
30 }
```

Listing 1

Forwarding consultable example.

```
 1 class Widget {
 2  public:
 3   Widget(const string &str): name_(str){}
 4   string get_name() const {return name_;}
 5  private:
 6   string name_;
 7 };
 8
 9 class WidgetOwner {
10  public:
11   Make_consultable(Widget, &first_,
     consult_first);
12   Make_consultable(Widget, &second_,
     consult_second);
13  private:
14   Widget first_{"First"};
15   Widget second_{"Second"};
16 };
17
18 class Box {
19  public:
20   Forward_consultable(WidgetOwner, &wo_,
     consult_first, fwd_first);
21   Forward_consultable(WidgetOwner, &wo_,
     consult_second, fwd_second);
22  private:
23   WidgetOwner wo_;
24 };
25
26 int main() {
27   Box b{};
28   cout << b.fwd_first(&Widget::get_name)
     // prints First
29        << b.fwd_second(&Widget::get_name)
        // prints Second
30        << endl;
31 }
```

Listing 2

could provide a setter that configure the refresh rate of an internal measure in a thread, or a setter for registering a callback function. Since **`Make_consultable`** does not feature selective inclusion of a non-const methods, a possible way for setter inclusion is to declare it const concerned member **`mutable`**. This is illustrated with Listing 4, where the enabling of a setter is obtained from inside the delegate class.

This approach seems to be more appropriate than introducing selective delegation from the delegator. First, the selection of a method from the delegator would break safety: it will be potentially forwarded and then accessed by multiple class, allowing internal state modification from several classes. In this case, the setter may need internal mechanism for being safe to be accessed (mutex, ...), which should probably be ensured from the delegate itself. Second, the use of **`mutable`** and **`const`** for the member setter is making explicit the intention of giving a safe access and is accordingly indicating to the user the method is available for consultation directly from the delegate header.

Implementation

An extract of implementation of both **`Make_consultable`** and **`Forward_consultable`** are presented here. Although not presented here, the available source code hosted on github provides additional overloads for methods returning void.

Listing 5 presents the **`Make_consultable`** implementation. The use of a macro allows for declaring the consultation methods. Accordingly, a consult method (**`_consult_method`**), which name is given by a macro argument, can be specified as public or protected. The declaration consists of two overloads that are distinguished by their constness. More

Overloaded methods in the delegate class. Types from overloaded method are given as template parameters in order to select the wanted implementation.

```
 1 class Widget {
 2  public:
 3  ...
 4   string hello() const { return "hello"; }
 5   string hello(const std::string &str) const {
       return "hello " + str; }
 6 };
 7
 8 ...
 9
10 int main() {
11   WidgetOwner wo{};
12   // In case of overloads, signature types give
        as template parameter
13   // allows to distinguishing which overload to
        select
14   cout << wo.consult_first<string>
        (&Widget::hello)  // hello
15        << wo.consult_first<string,
          const string &>(
16             &Widget::hello, std::string("ho"))
               // hello ho
17        << endl;
18
19        // static_cast allows for more verbosely
             selecting the wanted
20        cout << wo.consult_first(
21             static_cast<string(Widget::*)
22                         (const string &)
                            const>(&Widget::hello),
23                          "you")  // hello you
24             << endl;
25 }
```

Listing 3

Enabling setters in the delegate class. This is achieved by making the setter a const method (line 5), and accordingly making the member mutable (line 9). The setter is used in order to pass a lambda (lines 23–25) function that will be stored by the Widget (line 6). As a result, the setter is explicitly enabled when Widget is made consultable. This is also made explicit to the programmer who is reading the Widget header file.

```
 1 class Widget {
 2  public:
 3   // make a setter consultable from inside
     the delegate
 4   // set_callback needs to be const and cb_
     needs to be mutable
 5   void set_callback(std::function<void ()> cb)
     const {
 6     cb_ = cb;
 7   }
 8  private:
 9   mutable std::function<void()> cb_{nullptr};
10 };
11
12 class WidgetOwner {
13  public:
14   Make_consultable(Widget, &first_,
     consult_first);
15
16   private:
17   Widget first_{};
18 };
```

Listing 4

```
19
20 int main() {
21   WidgetOwner wo{};
22   // accessing set_name (made const from the
     Widget)
23   wo.consult_first(&Widget::set_callback,
     [](){
24       std::cout << "callback" << std::endl;
25     });
26 }
```

Listing 4 (cont'd)

particularly, the non-const overload fails to compile with a systematic **static_assert** when this overload is selected by type deduction, disabling accordingly the use of delegate non-const methods. The consult method (line 13) uses a variadic template for wrapping any const methods from the delegate (**fun**), taking the method pointer and the arguments to pass at invocation. Notice the types of fun arguments (**ATs**) and the types of the consult method arguments (**BTs**) are specified independently, allowing a user to pass arguments which does not have the exact same type, allowing them to pass a **char *** for an argument specified as a **const string &**.

The delegate type is saved for later reuse when forwarding (lines 6 and 7). This is using macro concatenation **##** in order generate a type name a forwarder can find, i.e. the consult method name concatenated with the string **Consult_t**.

Implementation of **Make_consultable**. Overload resolution for **_consult_method** select delegate member according to their **const**ness. If not **const**, compilation is aborted with **static_assert**.

```
 1 #define Make_consultable(_member_type,          \
 2                          _member_rawptr,        \
 3                          _consult_method)       \
 4                                                 \
 5   /*saving consultable type for the             \
     forwarder(s)*/                                \
 6   using _consult_method##Consult_t = typename   \
 7       std::remove_pointer<std::decay
           <_member_type>::type>::type;            \
 8                                                 \
 9   /* exposing T const methods accessible by T
        instance owner*/                           \
10   template<typename R,                          \
11            typename ...ATs,                     \
12            typename ...BTs>                     \
13   inline R _consult_method(R
     (_member_type::*fun)(ATs...) const,           \
14                           BTs ...args) const {\
15     return ((_member_rawptr)->*fun)
       (std::forward<BTs>(args)...);               \
16   }                                             \
17                                                 \
18   /* disable invocation of non const*/          \
19   template<typename R,                          \
20            typename ...ATs,                     \
21            typename ...BTs>                     \
22   R _consult_method(R
        (_member_type::*function)(ATs...),        \
23                     BTs ...) const {            \
24     static_assert(std::is_const<decltype
       (function)>::value,                         \
25       "consultation is available for const
        methods only");                            \
26     return R();  /* for syntax only since
       assert should always fail */                \
27   }                                             \
```

Listing 5

This brings us to the implementation of **Forward_consultable** (Listing 6). As with **Make_consultable**, it is actually an inlined wrapper generator using variadic template. However, it has no reference to the delegate, but only to the consult method it is invoking at line 18. The delegate type in obtained from the previously saved member type (line 7 to 9) and used for invoking the consult method (line 19). Again, the delegate type is saved for possible forwarders of this forward (line 7).

Make & Forward Delegate

Usage

The Delegate presented here is similar to Consultable, except that non-const methods are also enabled. They are also forwardable as Consultable, blocking the access to non-const methods of the owner of the forwarded. Listing 7 illustrates how access to non-const method can be managed and blocked when desired: **hello** is a non-const method in Widget. **widgetOwner** is Making two Delegates **first_** and **second_** (lines 13 & 14). hello is accordingly available from the owner of a **WidgetOwner** (lines 32 & 33). However, **Box** is forwarding access to **first_** as consultable (line 23) and access to **second_** as delegate (line 24). As a result, the owner of a **Box** have access to non-const method of **second_** (line 39), but access to const only methods of **first_** (line 37).

Implementation

The core implementation of **Make_Delegate** is actually the same as Consultable. In the source code, consultable and delegate is chosen with a flag that enables or disables access to non-const methods[3]. **Make_consultable** and **Make_delegate** are actually macros that expand to the same macro with the appropriate flag.

Listing 8 is presenting how this flag is implemented in the **Make_access** macro. This template is selected when a non-const pointer to the delegate is given (**fun** is not a const member). The flag is a non-type template parameter **flag** (line 15) which value is determined by the concatenation of macro parameters **_consult_method** and **_access_flag**. This value is tested (**static_assert** line 18) against the pre-defined enum values (line 6-9), enabling access to non-const methods or not at compile

Implementation of **Forward_consultable**.

```
 1 #define Forward_consultable(_member_type,      \
 2                             _member_rawptr,    \
 3                             _consult_method,   \
 4                             _fw_method)        \
 5                                                \
 6   /*forwarding consultable type for other
       forwarder(s)*/                            \
 7   using _fw_method##Consult_t = typename       \
 8       std::decay<_member_type>::type::         \
 9       _consult_method##Consult_t;              \
10                                                \
11   template<typename R,                         \
12            typename ...ATs,                    \
13            typename ...BTs>                    \
14   inline R _fw_method(                         \
15       R( _fw_method##Consult_t
             ::*function)(ATs...) const,          \
16       BTs ...args) const {                     \
17     return (_member_rawptr)->                  \
18         _consult_method<R, ATs...>(            \
19             std::forward<R(
               _fw_method##Consult_t ::*)
               (ATs...) const>(                  \
20                 function),                    \
21             std::forward<BTs>(args)...);       \
22   }                                            \
```

Listing 6

3. For expected improvement of clarity, Listing 5 is a simplification of actual code, hiding the flag mechanism.

time. Accordingly, **_access_flag** must take one of the two following string: **non_const** or **const_only**.

The implementation of **Forward_delegate** and **Forward_consultable** is also using the same flag mechanism for disabling or not the use of non-const methods.

Summary and discussion

Consultable and *Delegate* has been presented, including how they can be used with overloaded methods inside the delegate class, how specific setters can be enabled with for consultable. Forwarding is also presented, allowing to make available delegate methods to the user through any number of classes with access rights (public const methods only or all public methods).

Their implementation is based on C++11 template programming and macros inserted inside the class declaration, specifying the member to delegate and the name of the access method to generate to the user. Examples and source code are available on github and have been compiled and tested successfully using with gcc 4.8.2-19ubuntu1, clang 3.4-1ubuntu3 & Apple LLVM version 6.0.

Consultable is currently used in production code where it was found very useful for refactoring delegate interface for adaptation or experimentation. This is also reducing size of code in classes that use delegation/consultation extensively, and makes clearer from the code what can be done. Without *Consultables*, the wrappers from the various delegates makes the code very noisy and hard to read. Additionally, this project makes use of **Forward_consultable_from_map**, which has not been presented here since very specific. A more generic way of Forwarding from containers should be designed and developed in order to let the user manage container searching, along with error handling. ■

Acknowledgement

This work has been done at the Société des Arts Technologiques and funded by the Ministère de l'Économie, de l'Innovation et des Exportations (Québec, Canada).

References

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Meyers05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd Edition). Addison-Wesley Professional, 2005.

[Reenskaug07] Trygve Reenskaug. *Computer Software Engineering Research*, chapter 'The Case for Readable Code'. Nova Science Publishers, Inc., 2007.

Make and forward delegate example.

```
 1 class Widget {
 2  public:
 3   std::string hello(const std::string &str) {
 4     last_hello_ = str;
 5     return "hello " + str;
 6   }
 7  private:
 8   std::string last_hello_{};
 9 };
10
11 class WidgetOwner {
12  public:
13   Make_delegate(Widget, &first_, use_first);
14   Make_delegate(Widget, &second_, use_second);
15
16  private:
17   Widget first_{};
18   Widget second_{};
19 };
20
21 class Box {
22  public:
23   Forward_consultable(WidgetOwner, &wo_,
     use_first, fwd_first);
24   Forward_delegate(WidgetOwner, &wo_,
     use_second, fwd_second);
25  private:
26   WidgetOwner wo_;
27 };
28
29 int main() {
30   WidgetOwner wo{};
31   // both invocation are allowed since first_
     and second are delegated
32   cout << wo.use_first(&Widget::hello, "you")
     << endl;   // hello you
33   cout << wo.use_second(&Widget::hello, "you")
     << endl;   // hello you
34
35   Box b{};
36   // compile error, first_ is now a consultable:
37   // cout << b.fwd_first(&Widget::hello, "you")
     << endl;
38   //  OK, second_ is a delegate:
39   cout << b.fwd_second(&Widget::hello, "you")
     << endl;   // hello you
40 }
```

Listing 7

Extract of the core implementation of both **Make_Delegate** and **Make_consultable**. **_access_flag** is used in order to enable/disable the use of non const methods.

```
 1 #define Make_access(_member_type,              \
 2                     _member_rawptr,             \
 3                     _consult_method,            \
 4                     _access_flag)               \
 5                                                 \
 6   enum _consult_method##NonConst_t {            \
 7     _consult_method##non_const,                 \
 8         _consult_method##const_only             \
 9         };                                      \
10                                                 \
11   /* disable invocation of non-const
     if the flag is set*/                          \
12   template<typename R,                          \
13            typename ...ATs,                     \
14            typename ...BTs,                     \
15            int flag=
              _consult_method##_access_flag>       \
16   inline R _consult_method
     (R(_member_type::*fun)(ATs...),               \
17      BTs ...args) {                             \
18     static_assert(flag ==
       _consult_method##NonConst_t::               \
19     _consult_method##non_const,                 \
20     "consultation is available for const
       methods only "                              \
21     "and delegation is disabled");              \
22     return ((_member_rawptr)->*fun)
       (std::forward<BTs>(args)...);               \
23   }                                             \
```

Listing 8