overload 117

OCTOBER 2013 £3

Code as a Crime Scene
Investigating how to predict code crime scenes

Has the Singleton Not Suffered Enough
Why is this common design pattern so maligned?

C++11 Range and Elevation
Elevating C++ iterators to a high level range library

Automatic Navigation Mesh Generation
Adding navigation meshes to walk 3D environments

Lies, Damn Lies, and Estimates
It's long been one of the dark arts of the programming pursuit. We review why timescale estimation is so tricky.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 117

October 2013

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 118 should be submitted by 1st November 2013 and those for Overload 119 by 1st January 2014.

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

4 Code as a Crime Scene
Adam Tornhill demonstrates how to predict code crimes.

9 Lies, Damn Lies and Estimates
Seb Rose reviews difficulties in providing estimates.

12 YAGNI-C as a Practical Application of YAGNI
Sergey Ignatchenko offers a constrained version of YAGNI.

15 Has the Singleton Not Suffered Enough
Omar Bashir considers why singletons are much maligned.

22 Automatic Navigation Mesh Generation in Configuration Space
Stuart Golodetz presents navigation meshes to walk 3D environments.

28 C++ Range and Elevation
Steve Love elevates C++ iterators to a high level range library.

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.

Copyrights and Trade Marks

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

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

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

Decisions, Decisions

Providing precise definitions can be difficult. For example, what is a program?

Previously we considered how to learn a, possibly non-existent, programming language. This clearly pre-supposes we understand what a programming language is. Upon realising I only had a couple of weeks to get an editorial together, and as ever, with no opinions to share or muse on, I decided to spends hours wondering what a programming language actually is, and furthermore, what constitutes a computer program.

Opinions seem to vary. Some people do not regard VB, VBA or Excel as 'proper' programming, while others get upset by this viewpoint. What about mathematics or statistics packages, such as MatLab or R? They can include data structures and ways of controlling the flow of execution. They allow one to issue instructions to a computer. This seems 'programish'. Of two colleagues at work, one does not regard R as programming since it is very high-level compared to his roots, programming in assembler. Another does regard R as a proper language, though paused for a while when I asked what the difference between using a program and writing a program was. It seems there may be a difference between a tool for getting a job done, and the building blocks for making such a tool. Ignoring the obvious emotive nature of such discussions, we can broaden this out from maths and stats packages. Is Markup a language? It seems strange to regard XML as a programming language, but XSLT almost certainly is, since it is Turing complete [Kepser]. In other words, it can 'compute every Turing computable function'. [Turing_completeness] That was helpful, wasn't it? Equivalently, it can simulate a universal Turing machine. Some may argue that nothing can simulate a universal Turing machine, since such a machine has infinite memory and is indestructible, but let us leave aside such practical details.

Returning to our original question, 'What counts as a programming language?', it might ironically be possible to regard an attempt to create a mathematics package as building the foundations of programming. Whether this means mathematics packages count as programming languages is a matter for further discussion, but for now let us consider a little history. Quite early in his career, David Hilbert published a book establishing a foundation for geometry, demonstrating that its axioms, based on Euclid's, were consistent, in other words it does not contain or give rise to a contradiction. Why does this matter? If you start with a contradiction, you can prove anything. 'Ex falso, quodlibet.' From falsity, whatever you like. For example, assume A and !A, for some statement A. Formally, using & for 'and', ! for 'not', | for 'or' and => for 'implies',

A & !A => A (1)

and similarly

A & !A => !A (2)

by definition of &.

From (1), for any B, such as 'Unicorns exist' since A holds, A or B holds:

A | B (3)

by definition of |.

From (2), !A, (3) => B, by definition of | (if A or B is true and A is false, B must be true)

From our original assumption, A & !A, we have deduced B, formally

A & !A => B

that is unicorns exist, or whatever we chose for B.

Hilbert would not be impressed. Later, he gave a major address to the Second International Congress of Mathematics in August 1900. In his speech he mentioned 10 unsolved mathematics problems, though the published version went on to list 23. He said,

> This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call. There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus. [Hilbert]

His second problem called for proof of the consistency of the real numbers, which he had rather assumed in his earlier geometry book. The tenth problem called for someone to 'Devise a process according to which it can be determined by a finite number of operations whether the [Diophantine] equation is solvable in rational integers' [Hilbert]. The full set of twenty three problems can be readily found, for example see Wikipedia [Hilbert's_problem]. A curious point to note is a mathematician asking for a process or algorithm, rather than an answer. We here see the beginnings of what is now known as Hilbert's Programme; an attempt to formalise mathematics in axiomatic form together with a proof of its consistency and completeness, including no superfluous axioms, and with every well-formed formula being 'decidable' that is provable or falsifiable. This would allow all of mathematics to be deduced from the axioms. In 1928, Hilbert generalises these ideas and questions to the Entscheidungsproblem – the decision problem. Can you devise an algorithm which concludes whether a correctly formed statement is deducible from a set of axioms? Over time, people engaged with the problem, and some dream of devising a procedure that could be carried out by a machine. The quest to put mathematics on firm logical foundations had started a quest for the ultimate mathematics package.

In 1936 Alonzo Church and Alan Turing both published separate papers showing there can be no general solution to the decision problem. Church used his λ-calculus [Church] and Turing invented his Turing machine [Turing]. Turing's paper is only 36 pages long but ground-breaking. He introduces the idea of an automatic machine, which can read a symbol at a time from a tape, alter or erase the symbol, move the tape (or read head)

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

and write further symbols. In conjunction with a state register, which stores one of finitely many states, and a table of instructions mapping states to symbols, we have a Turing machine. Some of the instructions for a given state and input symbol involve moving the tape which reads almost exactly like a `goto`. Taking the automatic machine a step further, Turing moves on to describe a universal machine, U, which is supplied with a tape containing the instructions and initial state of an automatic machine, M. U will then compute the same sequence as M. Furthermore, it can be given the instructions for any automatic machine M. As noted, this gives it infinite memory and assumes the machine and tape never break, nonetheless imagining a machine that can be instructed, or programmed, rather than building a special purpose machine for each and every problem is a revolutionary idea.

> Turing's paper ... contains, in essence, the invention of the modern computer and some of the programming techniques that accompanied it. [Minksy]

Turing's paper is quite difficult to read, so on my second attempt I opted for Petzold's *The Annotated Turing* [Petzold] which weighs in at 372 pages, making it just over ten times the length of the original paper and Petzold's shortest book. He tells us:

> I have sometimes tried to write 300-page books. I would very much like to write a 300-page book. But I have always failed. [Petzold2]

At least he has made an effort.

Unfortunately, Church and Turing had both scuppered Hilbert's programme (there's that word 'program' again). They had both been influenced by the earlier work of Gödel, from 1931, which burrowed down into the requirements for consistency and completeness. Many people refer to Gödel's incompleteness theorem, but it should be noted there are two. The first shows that for any consistent axiomatic system capable of describing the natural numbers, there are true statements which the system cannot prove. At a high level, this is demonstrated by encoding 'This statement is not provable' within the system. If the system can prove that statement it means it cannot prove that statement. If it cannot prove that statement, this proves it can prove the statement. Either way it is inconsistent. A consistent system cannot therefore be complete. The second theorem shows that any such consistent system cannot prove its own consistency. The technical difficulty here comes from describing the consistency of the system within the system, but when you manage that the proof follows almost immediately.

What have we learnt about the nature of a computer program? First, you do not need a computer in order to write a program. Next programming languages have a syntax. Hilbert desired the decidability of a 'well-formed formula' in his program. How do you decide if a formula is well-formed? Strict syntax rules pin that down very easily. Gödel's incompleteness theorems do not apply here, since the syntax checker is external to the program itself. Finally, a program does something. Given its current state and an input, using a list of instructions, it acts: Turing's machine wrote a 1 or 0, move the tape or erased a symbol on the tape, not thrashing the program but rather keeping its rough working up to date, leaving the program in the next state. It is of course, easier to program if you have a computer, simpler if another program checks the syntax of your program for you, and very satisfying if the program does what was required.

Where does this leave a programmer today? Firstly, with a computer, which must makes things easier than they were for Turing. Secondly, armed with considerably more programming languages than existed half a century ago, confronted by more than a fistful of idioms and a plethora of patterns, a programmer can control MRI scanners, create games, numerically solve hard mathematical problems, but can still never be sure if the program will halt, run out of memory, or behave correctly. Clearly, the decision problem suggests we are, in general, unable to decide if any given program will halt, or contain a bug. Perhaps we must acknowledge 'ignorabimus' and uncertainty. I know I certainly couldn't decide on a suitable editorial topic, and hope you accept my apologies and excuses yet again.

References

[Church] 'An unsolvable problem of elementary number theory', *American Journal of Mathematics*, 58 (1936), pp 345–363, and 'A note on the Entscheidungsproblem', *Journal of Symbolic Logic*, 1 (1936), pp 40–41.

[Hilbert] Quoted by Charles Petzold in *The Annotated Turing* Wiley, 2008 (see below)

[Hilbert's_problem] http://en.wikipedia.org/wiki/Hilbert%27s_problems

[Kepser] Stephan Kepser 'A Simple Proof for the Turing-Completeness of XSLT and XQuery', *Extreme Markup Languages* 2004

[Minksy] *Computation: Finite and Infinite Machines*, Prentice-Hall, Inc., N.J., 1967.

[Petzold] *The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and The Turing Machine'* Wiley, 2008

[Petzold2] 'The 300 page ideal' http://www.charlespetzold.com/blog/2008/05/The-300-Page-Ideal.html

[Turing] Alan Turing, 'On computable numbers, with an application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society*, Series 2, 42 (1936–7), pp 230–265

[Turing_completeness] http://en.wikipedia.org/wiki/Turing_completeness

Code as a Crime Scene

Forensic techniques can predict possible future crimes. Adam Tornhill shows how they can be applied to code.

As projects grow, social and organizational aspects interact with technical challenges to inflate the complexity in the solution domain. To address that multi-dimensional complexity we need to look at the historic evolution of our systems. Inspired by modern offender profiling methods from forensic psychology, we'll develop techniques to utilize historic information from our version-control systems to identify weak spots in our code base. Just like we want to hunt down offenders in the real world, we need to find and correct offending code in our own designs.

The maintenance puzzle

Maintenance is a challenge to every software project. It's the most expensive phase in any product's life cycle, consuming approximately 60 percent of the cost. Maintenance is also different in its very essence when compared to greenfield development. Of all that maintenance money, a whole 60 percent is spent on modifications to existing programs [Glass02].

The immediate conclusion is that if we want to optimize any aspect of software development, maintenance is the most important part to focus on. We need to make it as cheap and predictable as possible to modify existing programs. The current trend towards agile project disciplines makes it an even more urgent matter. In an agile environment we basically enter maintenance mode immediately after the first iteration and we want to make sure that the time we spend in the most expensive life cycle phase is well-invested.

So is it actually maintenance that is the root of all evil? Not really. Maintenance is a good sign – it means someone's using your program. And that someone cares enough to request modifications and new features. Without that force, we'd all be unemployed. I'd rather view maintenance as part of a valuable feedback loop that allows us to continuously learn and improve, both as programmers and as an organization. We just have to be able to act on the feedback.

The typical answer to that maintenance puzzle is refactoring [Fowler99]. We strive to keep the code simple and refactor as needed. Done by the book, refactoring is a disciplined and low-risk investment in the code base. Yet, many modifications require design changes on a higher level. Fundamental assumptions will change and complex designs will inevitably have to be rethought. In limited and specific situations refactoring can take us there in a controlled series of smaller steps. But that's not always the case. Even when it is, we still need a sense of overall direction.

Intuition doesn't scale

Since the 70s we have tried to identify complexity and quality problems by using synthetic complexity measurements (e.g. Halstead [Halstead77] or McCabe [McCabe76] complexity measures). It's an approach that has gained limited success. The main reason is that traditional metrics just cannot capture the complex nature of software.

If complexity metrics don't work, at least not to their full promise, what's left for us? Robert Glass, one of my favorite writers in the software field, suggests intuition [Glass06].

Human intuition is often praised as a gift with mystical, almost magical qualities. Intuition sure is powerful. So why not just have our experts glance at our code and pass an immediate verdict on its qualities and virtues? As we're about to discover, intuition has its place. But that place is more limited than my previous reference to Robert Glass suggests.

Intuition is largely an automatic psychological process. This is both a strength and a weakness. Automatic processes are unconscious cognitive processes characterized by their high efficiency and ability to make fast and complex decisions based on vast amounts of information. But efficiency comes at a price. Automatic processes are prone to social and cognitive biases.

Even if we somehow manage to avoid those biases, a task that is virtually impossible for us, we would still have a problem if we rely on intuition. Intuition doesn't scale. No matter how good a programmer you are, there's no way your expertise is going to scale across hundreds of thousands or even million lines of code. Worse, with parallel development activities, performed by different teams, our code base gets another dimension too. And that's a dimension that isn't visible in the physical structure of the code.

A temporal dimension of code

Over time a code base matures. Different parts stabilize at different rates. As some parts stabilize others become more fragile and volatile which necessitates a shift in focus over time. The consequence is that parts of the code base may well contain excess complexity. But that doesn't mean we should go after it immediately. If we aren't working on that part, and haven't been for quite some time, it's basically a cold spot. Instead other parts of the code may require our immediate attention.

The idea is to prioritize our technical debt based on the amount of recent development activity. A key to this prioritization is to consider the evolution of our system over time. That is, we need to introduce a temporal axis. Considering the evolution of our code along a temporal axis allows us to take the temperature on its different parts. The identified hot spots, the parts with high development activity, will be our priorities.

This strategy provides us with a guide to our code. It's a guide that shows where to focus our cognitive cycles by answering questions like:

- What's the most complex spot in our software that we're likely to change next?
- What are the typical consequences of that change?
- Is is likely to be a local change or will other parts have to change too? Can we predict the impact on related and seemingly unrelated modules?

Adam Tornhill Combining degrees in engineering and psychology, Adam tries to unite these two worlds by making his technical solutions fit the human element. While he gets paid to code in C++, C#, Java and Python, he's more likely to hack Lisp or Erlang in his spare time. Other interests include modern history, music and martial arts.

The profiling techniques are tools used to focus scarce manual efforts and expertise on where they're needed the most

In their essence, such open problems are similar to the ones forensic psychologists face. Consider a series of crimes spread out over a vast geographical area:

- Where can we expect the next crime to occur?
- What area is most likely to serve as home base of the offender?
- Are there any patterns in the series of crimes? Can we predict where the offender will strike again?

Modern forensic psychologists and crime investigators attack these open, large-scale problems with methods useful to us software developers too. Follow along, and we'll see how. Welcome to the world of forensic psychology!

Geographical profiling of crimes: a 2 minutes introduction

Modern geographical profiling bears little or no resemblance to the Hollywood cliché of 'profiling' as seen in movies. There the personality traits of an anonymous offender are read like an open book. Needless to say, there's little to no research backing that approach. Instead geographical profiling has a firm scientific basis in statistics and environmental psychology. It's a complex subject with its fair share of controversies and divided opinions (in other words: just like our field of programming). But the basic principles are simple enough to grasp in a few minutes.

The basic premise is that the geographical location of crimes contain valuable information. For a crime to occur, there must be an overlap in space and time between the offender and a victim. The locations of crimes are never random. Most of the time criminals behave just like ordinary, law abiding citizens. They go to work, visit restaurants and shops, maintain social contacts. During these activities, each individual builds a mental map of the geographical areas he visits and lives in. It's a mental map that will come to shape the decision on where to commit a crime.

The characteristic of a crime is a personal trade-off between potential opportunities and risk of detection. Most offenders commit their offenses close to home (with some variation depending on the type of offense and the environment). The reason is that the area close to home is also the area where the known crime opportunities have been spotted. As the distance to the home increases, there's a decline in crimes. This spatial behavior is known as distance decay. At the same time, the offender wants to avoid detection. Since he may be well-known in the immediate area around the home base, there's typically a small area where no crimes are committed.

Once a crime has been committed, the offender realizes there's a risk associated with an immediate return to that area. The typical response is to select the next potential crime target in the opposite direction. Over time, the geographical distribution of the crimes become the shape of a circle. So while the deeds of an offender may be bizarre, the processes behind them are rationale with a strong logic to them [Canter08]. It's this underlying dark logic that forms the patterns and allows us to profile a crime series. By mapping out the locations on a map and applying these two principles we're able to get an informed idea on where the offender has his home base.

Finally, a small disclaimer. Real-world geographical profiling is more sophisticated. Since psychologically all distances aren't equal the crime locations have to be weighted. One approach is to consider each crime location a center of gravity and mathematically weight them together. That weighted result will point us to the geographical area most likely to contain the home base of the offender, our hot spot. But the underlying basic principles are the same. Simplicity does scale, even in the real world.

The geography of code

Geographical profiling does not point us to an exact location. The technique is about significantly narrowing the search area. It's all about highlighting hot spots in a larger geographical area. The profiling techniques are tools used to focus scarce manual efforts and expertise on where they're needed the most.

It's an attractive idea to apply similar techniques to software systems. Instead of trying to speculate about potential technical debt amongst thousands or perhaps million lines of code, geographical profiling would give us to a prioritized lists of modules, the hot spots, our top offenders. That leaves us with the challenge of identifying both a geography of code and a spatial movement within our digital creations. Let's start with the former.

Over the years there have been several interesting attempts to visualize large-scale software systems. My personal favorite is Code City [CodeCity] where software systems are visualized as cities. Each package becomes a city block, each class a building with the number of methods defining the height and the number of attributes defining the base of the building. Not only does it match the profiling metaphor; it's also visually appealing and makes large, monolithic classes stand out. Figure 1 is a geography of code as visualized by Code City. Each building represents a class with the number of methods giving the height and the attributes the base.

Visualizations may give us a geography, but the picture is no more complete than the metrics behind it. If you've been following along in this chapter, you'll see that we need another dimension – it's the overlap between code characteristics and the spatial movement of the

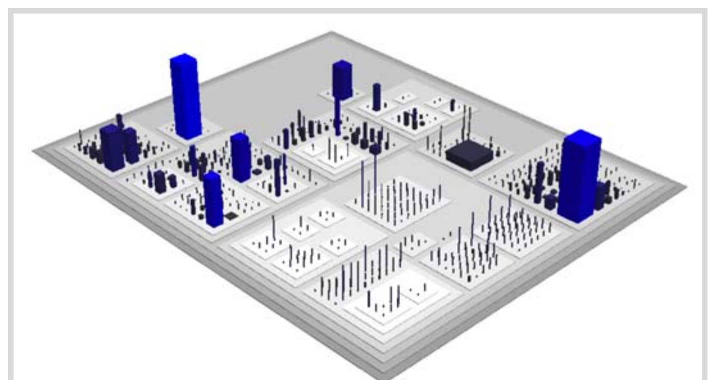


Figure 1

VCS data allows us to **trace changes** over series of commits in order to **spot patterns in the modifications**

programmers within the code that is interesting. That overlap between complex code and high activity are the combined factors to guide our refactoring efforts. Complexity is only a problem when we need to deal with it. In case no one needs to read or modify a particular part of the code, does it really matter if it's complex? Sure, it's a potential time bomb waiting to go off. We may choose to address it, but prefer to correct more immediate concerns first. Our profiling techniques allows us to get our priorities right.

Interaction patterns identify hot spots

Just like each crime provides the geographical profiler with valuable information, so does each code change in our system. Each change we make, each commit, contain information on how we as developers interact with the evolving system. The information is available in our version-control systems (VCS). The statistics from our VCS is an informational gold mine. Mining and analyzing that information provides us with quite a different view of the system (see Table 1).

The VCS data is our equivalent to spatial movement in geographical profiling since it records the steps of each developer. The subset I chose to mine is derived from organizational metrics known to serve as good predictors of defects and quality issues (see for example Nagappan et al. [Nagappan08]):

1. **Number of developers**: The more developers working on a module, the larger the communication challenges. Note that this metric is a compound. As discussed below, the metric can be split into former employed developers and current developers to weight in potential knowledge drain.
2. **Number of revisions**: Code changes for a reason. Perhaps because a certain module has too many responsibilities or because the feature area is poorly understood. This metric is based on the idea that code that has changed in the past is likely to change again.

In contrast to traditional metrics, organizational metrics carry social information. Depending on the available information, additional metrics may be added. For example, it's rare to see a stable software team work together for extended periods of time. Each person that leaves drains the accumulated knowledge. When this type of information is available, it's recommended to weight it into our analysis.

Example on data mined from a version-control system. The data serves as the basis to identify hot spots in our code base.

Module	Number of developers	Number of revisions
Protocol.java	7	134
ClientConnection.java	8	76
Message.java	4	74
DispatchTable.java	12	53

Table 1

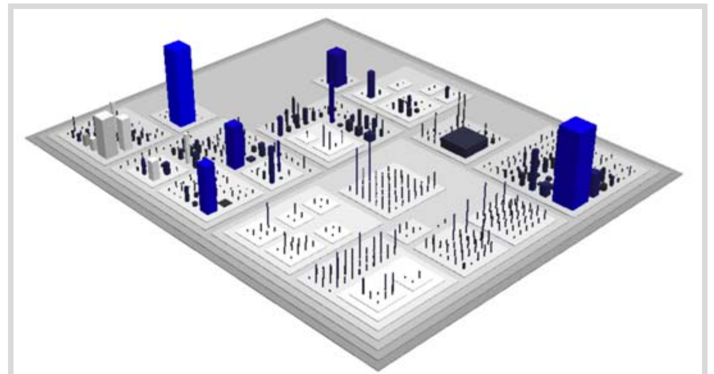


Figure 2

Together those metrics let us identify the areas of the system that are subject to particularly heavy development, the areas with lot of parallel development activity by multiple developers or the parts with the highest change frequency. In geographical profiling we combined the principles of distance decay and the circle hypothesis to predict the home base of an offender. In the same spirit we can visualize the overlap between traditional complexity metrics and our new organizational metrics. Figure 2 shows custom color mark-up in Code City to highlight areas of intense parallel development by multiple programmers; the more intense the colors, the more activity.

The outcome of such hot spot analysis in code has often surprised me. Sometimes, the most complex areas are not necessarily where we spend our efforts. Instead, I often find several spread-out areas of intense development activity. With multiple developers crowding in those same areas of a code base, future quality issues and design problems will arise.

Emergent design driven by software evolution

Our crash-course in geographical profiling of crimes taught us how linking crimes and considering them as a related network allows us to make predictions and take possible counter steps. Similarly VCS data allows us to trace changes over series of commits in order to spot patterns in the modifications. The resulting analysis will allow us to detect subtle problems that go beyond what traditional metrics are able to show. It's an analysis that suggests directions for our refactorings based on the evolution of the code itself. The basis is a concept called *logical coupling*.

Logical coupling refers to modules that tend to change together. The concept differs from traditional coupling in that there isn't necessarily any physical software dependency between the coupled modules. Modules that are logically coupled have a hidden, implicit dependency between them such that a change to one of them leads to a predictable change in the coupled module. Logical coupling shows-up in our VCS data as shared commits between the coupled modules.

With multiple developers crowding in those same areas of a code base, future quality issues and design problems will arise

Logical coupling analysis of the module NodeConnection.cs

Coupled module	Coupling (%)	Shared revisions (total)	Mean revisions (avg)
NodeSender.cs	80	12	15
NodeReceiver.cs	67	20	30
node_connection_test.py	60	9	15
UserStatistics.cs	48	11	23
...	...	...	...
Message.cs	7	1	15

Table 2

Analyzing logical coupling

At the time of writing, there's limited tool support for calculating logical coupling. There are capable academic research tools available [D'Ambros06], but as far as I know nothing in the open-source space. For the purpose of this article, I've started to work on Code Maat, a suite of open-source Clojure programs, to fill that gap [CodeMaat].

Code Maat calculates the logical coupling of all modules (past and present) in a code base. Depending on the longevity of the system, that may be a lot of information to process. Thus, I usually define a pretty high coupling threshold to start with and focus on sorting out the top offenders first. I also limit the temporal window to the recent period of interest; over time many design issues do get fixed and we don't want old data to interfere with our current analysis of the code.

The actual thresholds and temporal period depend on product-specific context. Even when I have full insight into a project, I usually have to tweak and experiment with different parameters to get a sensible set of data. Typically, I ignore logical coupling below 30 percent and I strip out all modules with too few revisions to avoid data skew. With much developer activity a temporal window of two or three months is a good heuristic.

Once we've decided on the initial parameters we can put Code Maat to work. Table 2 gives an example, limited to a single module, of the data I derive from a logical coupling analysis.

Visualizing logical coupling

In its simplest form logical coupling can be visualized by frequency diagrams. But just like the analysis of the organization metrics above, it's the overlap between traditional complexity and our more subtle measure of logical coupling that is the main point of interest. Where the two meet, there's likely to be a future refactoring feast. It's a data point we want to stand-out in our visualizations. Figure 3 shows a combined visualization of logical coupling and module complexity using tree maps.

A temporal period for logical coupling

I've been deliberately vague in my definition of logical coupling. What do I really mean with "modules that end to change together"? To analyze our VCS data we need to define a temporal period, a window of coupling. The precise quantification of that period depends on context.

In its most basic form, I consider modules logically coupled when they change in the same commit. Often, such a definiton takes us far enough to find interesting and unexpected relationships in our system. But in larger software organizations, that definition is probably too narrow. When multiple teams are responsible for different parts of the system, the temporal period of interest is probably extended to days or even weeks.

To visualize that multi-dimensional space I use tree maps where each tile represents a module. The size of each tile is proportional to its module's degree of logical coupling. The complexity of the coupled module (e.g. lines of code or Cyclomatic Complexity) is visualized using color mark-up; the darker the color, the more complex the module. Figure 4 shows a hierarchical view of the logical coupling partitioned on a per-layer basis.

Tree maps are an excellent choice in cases where the differing sizes or sheer amount of individual components would render a pie chart unreadable. Tree maps are also well-suited to illustrate hierarchical structure. One possible hierarchical visualization is to aggregate the logical coupling over packages and present a multi-layered top-down view of the total coupling in the system as in Figure 4.

Whatever we chose, once the logical coupling data is mined, our next step is to find out why specific modules keep changing together.

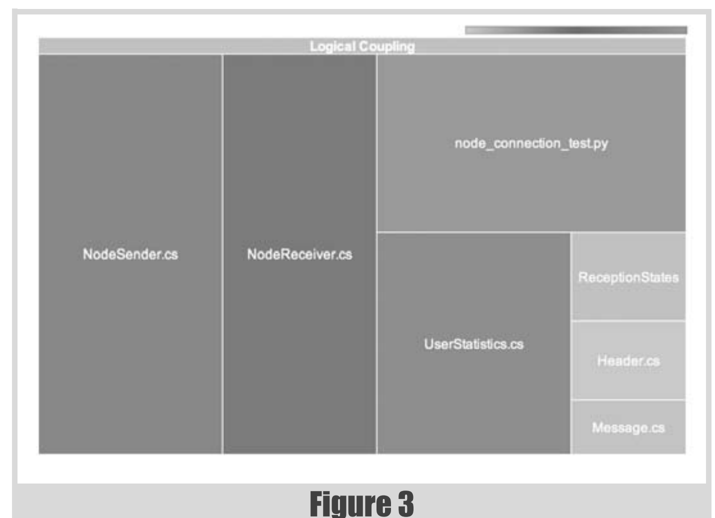


Figure 3

A couple for a reason

Logical coupling arises for a reason. The most common case is copy-paste code. This one is straightforward to address; extract and encapsulate the common functionality. But logical coupling often has more subtle roots. Perhaps the coupled modules reflect different roles, like a producer and consumer of specific information. In the example above, `NodeSender.cs` and `NodeReceiver.cs` seem to reflect those responsibilities. In such case it's not obvious what to do. Perhaps it's not even desirable to change the structure. It's situations like these that require our human expertise to pass an informed judgment within the given context.

The third reason for logical coupling is related to our timeless principles of encapsulation and cohesion. As the data above illustrates, the `UserStatistics.cs` module changed together with our `NodeConnection.cs` 48 percent of the time. To me there's no obvious reason it should. To find out why they're coupled we need to take the analysis a step further. Using our VCS data we can dig deeper and start to compare changed lines of code between the coupled files over the common commits. Once we see the pattern of change, it usually suggests there's some smaller module looking to get out. Refactoring towards that goal breaks the logical coupling.

Cases like these teaches us about the design of our system and points-out the direction for improvements. There's much to learn from a logical coupling analysis. Better yet, it's a language neutral technology.

A holistic view by language neutral analysis

VCS data is language neutral. Since our analysis allows us to cross language boundaries, we get a holistic picture of the complete system. In our increasingly polyglot programming world this is a major advantage over traditional software metrics.

Software shops often relay on multiple implementation technologies. One example is using a popular language such as Java or C# for the application development while writing automated tests in a more dynamic language like Python or Ruby. In Table 2 there's an example on this case where `node_connection_test.py` changes together with the module under test 60 percent of the time. Such a degree of coupling may be expected for a unit test. But in case the Python script serves as an end-to-end test it's probably exposed to way too much implementation detail. Web development is yet another example where a language neutral analysis is beneficial. A VCS-based analysis allows us to spot logical coupling between the document structure (HTML), the dynamic content (JavaScript) and the server software delivering the artifacts (Java, Clojure, Smalltalk, etc).

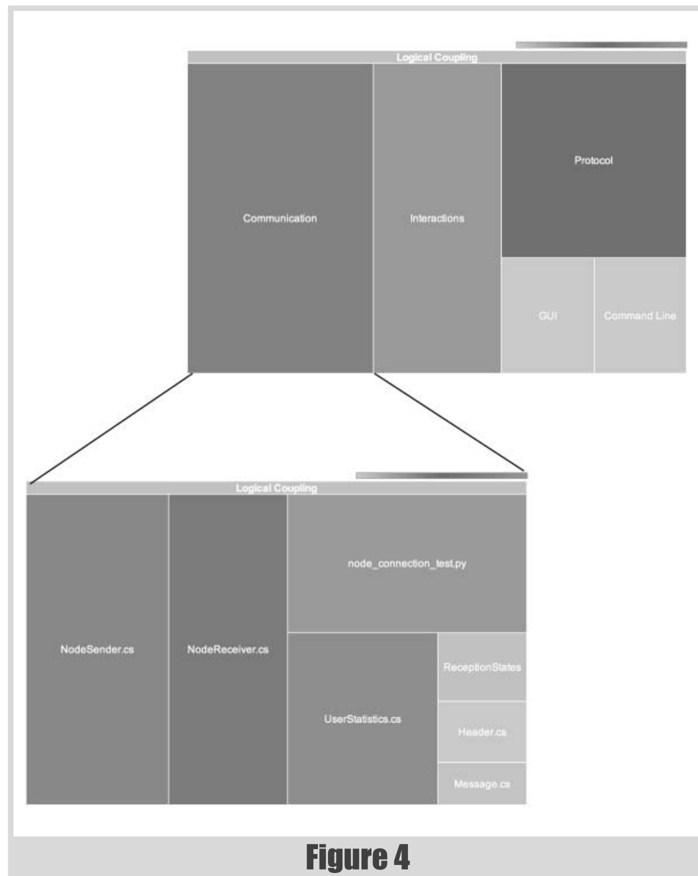


Figure 4

The road ahead

To understand large-scale software systems we need to look at their evolution. The history of our system provides us with data we cannot derive from a single snapshot of the source code. Instead VCS data blends technical, social and organizational information along a temporal axis that let us map out our interaction patterns in the code. Analyzing these patterns gives us early warnings on potential design issues and development bottlenecks, as well as suggesting new modularities based on actual interactions with the code. Addressing these issues saves costs, simplifies maintenance and let us evolve our systems in the direction of how we actually work with the code.

The road ahead points to a wider application of the techniques. While this article focused on analyzing the design aspect of software, reading code is a harder problem to solve. Integrating analysis of VCS data in the daily workflow of the programmer would allow such a system to provide reading recommendations. For example, 'programmers that read the code for the Communication module also checked-out the UserStatistics module' is a likely future recommendation to be seen in your favorite IDE.

Integrating VCS data into our daily workflow would allow future analysis methods to be more fine-grained. There's much improvement to be made if we could consider the time-scale within a single commit. As such, VCS data serves as both feedback and a helpful guide.

'Code as a crime scene' is an adapted chapter from my upcoming book with the same name. Please check it out for more writings on software design and its psychological aspects [Tornhill13]. ■

References

[Canter08] Canter, D. & Youngs, D. (2008). *Principles of Geographical Offender Profiling*

[CodeCity] Code City: http://www.inf.usi.ch/phd/wettel/codecity.html

[CodeMaat] Code Maat: http://www.adampetersen.se/code/codemaat.htm

[D'Ambros06] D'Ambros, M., Lanza, M. & Lungu, M. (2006). *The Evolution Radar*

[Fowler99] Fowler, M. (1999). *Refactoring*

[Glass02] Glass, R. L. (2002). *Facts and Fallacies of Software Engineering*

[Glass06] Glass, R.L. (2006). *Software Creativity 2.0*

[Halstead77] Halstead, M.H. (1977). *Elements of software science*

[McCabe76] McCabe, T.J. (1976). *A Complexity Measure*

[Nagappan08] Nagappan, N., Murphy, B. & Basiliet, V. (2008). *The influence of organizational structure on software quality*

[Tornhill13] Tornhill Petersen, A. (2013). Code as a Crime Scene: https://leanpub.com/crimescene

Lies, Damn Lies and Estimates

Predicting how long something will take is hard. Seb Rose takes us on a brief tour through the swamp that is estimation.

What are estimates for?

Estimates seem to be an essential part of the software development process. People want to know how much things will cost before work starts, and want regular updates on how we're performing as the project progresses. This seems entirely reasonable until you start looking under the covers at why we're being asked for estimates and what we're saying when we give them.

Estimates can be broadly categorised into two types: forecasting and tracking. Forecasts are used before a project starts to decide if it is worth implementing, while tracking estimates are used during project development to manage resources and act as a project health indicator.

Humans aren't good at estimating in general [Bowle]. We're over optimistic, as described by Pullitzer prize winner Douglas Hofstadter:

> It always takes longer than you expect, even when you take into account Hofstadter's Law.

Steve McConnell wrote a whole book about trying to 'demystify the black art'of software estimation [McConnell], which in the end is far too theoretical and (to my mind) largely impractical. DeMarco and Lister take a less formal approach [DeMarco], advising us to emphasise the imprecision of our estimates. Despite these, and other contributions to the field, I've seen no evidence to suggest that estimates have got any better over the past 30 years.

Investment decisions

Some years ago, I was working for a retail bank when I was asked to estimate how long it would take to implement a new feature. It took several days to analyse the requirements and come up with a 'ballpark' estimate of 3 months. The client reacted with horror, "That's too long!" but I stuck by the estimate and the feature was shelved.

Why was it shelved? In a rational world it might have been shelved because the value of the feature wasn't worth the investment. I never saw any prediction of the feature's value (I don't believe they had one), but I don't think this was the reason. I think that they had identified a 'resource' surplus and were trying to see if a low priority feature would 'fit'. When it didn't, they simply left it.

Five months later they came back with the same feature request and asked me to estimate it again. My estimate remained the same and the request was shelved again. They came back again a couple of months later with the same request. Same outcome.

Why did they keep asking the same question? Partly because they had forgotten that it had already been estimated and partly because they hoped that this time the estimate would 'fit' their plan. I've spent months of my life being asked to re-estimate features to try and get the numbers smaller. I've also had managers cut my estimates before passing them to clients. We don't want to disappoint people (especially our bosses) so we tell them what they want to hear – recriminations will happen in the future and, anyway, we might just get lucky on this project.

Waltzing bears

Demarco and Lister said, "If a project has no risk, then don't do it" [DeMarco]. This isn't intended to encourage us to do dangerous things, it's simply an observation that when we deliver value we are going to do something new, which is inherently risky. They talk at length about why estimates are often interpreted as commitments, and why we should provide estimates as a range with our level of confidence of being able to deliver within that range. So, instead of providing a 'point' estimate that a project will take 6 months, restate it as "I am 95% confident that we can deliver this within a 3 month to 24 month period." This is a probability distribution around the original 'point' estimate of 6 months. That seems like a huge range, and it is. However, they observe that a 400% overrun (based on 'point' initial estimates) is not uncommon in our industry.

Steve McConnell has documented the Cone of Uncertainty [McConnell], which describes how our estimates become more accurate as a project progresses. I believe this assumes that we address the most risky parts of the project early, which itself assumes that we 'know' what the most risky parts are. As Donald Rumsfeld observed [Rumsfeld], however, there's always the possibility that there are 'Unknown unknowns' lurking, that could come to light at any time to de-rail the project.

The other observation is that The Cone of Uncertainty is symmetric – implying that projects are just as likely to come in below estimate as over estimate. Laurent Bossavit has looked into the research that underpins this [Bossavit] and has found that it does not support this assumption. Depressingly, it seems that empirical evidence shows that projects rarely come in quicker than our 'point' estimates, so the estimate quoted above becomes 'I am 95% confident that we can deliver this within a 6 month to 24 month period.'

Over confident

Even with such a wide estimate, how can we be 90% confident? Maybe this is based on relevant historical data from your organisation, but every project is different. Different problems, different teams, different context. And we are very bad at estimating – including estimating confidence.

The Brier Score [Brier] is a 'proper score function that measures the accuracy of probabilistic predictions'. Try this yourself at home (thanks to Laurent Bossavit):

> Instructions: for each of the statements below, please give an answer between 0% (you are totally certain it is false) and 100% (you are totally certain it is true) – an answer of 50% means you are unable to say one way or the other. No cheating by looking things up!
>
> 1. The language JavaScript was released to the public after 31/12/1994. Certainty: ____

Seb Rose is an independent software developer, trainer and consultant based in the UK. He specialises in working with teams adopting and refining their agile practices, with a particular focus on delivering software through the use of examples. He can be contacted at seb@claysnow.co.uk

Once the project has kicked off, teams are often asked to break the coarse functionality down into fine grained tasks and estimate these individually

2. As of January 2013, LinkedIn reports more registered users than Brazil has citizens. Certainty: ____
3. UML (Unified Modeling Language) is a registered trademark of IBM Rational. Certainty: ____
4. The average salary for an engineer in test (SDET) at Google is > 85K$ (55K£) yearly. Certainty: ____
5. More than 3000 people registered for the Agile2012 conference in the US. Certainty: ____

Now turn to page 11 and score yourself.

How small is your Brier score? The good news is that we can train ourselves to be less over-confident [Web1] [Web2] [Web3]. The bad news is that it is very hard to become more precise.

The ROI fallacy

The forecasting charade has two sides: estimation and value. From this, we are told, those who know best can determine the return on investment (ROI). The magic number that predicts whether our work will deliver value to the business or not.

In my experience their value predictions are even less robust than our estimates. They surface in the project proposal documents and, once the project has kicked off, are never seen again. I have never seen an organisation track the value delivered and compare this to the value promised. Partly, this is because success criteria are not rigorously defined in advance and partly this is because tracking value delivered is itself highly subjective.

For example, a team I worked beside at a large online retailer was responsible for the algorithms that delivered targeted advertising on their checkout page. This team is the jewel in the crown of that development centre because, based on a few hours of outage several years ago, they have calculated a massive amount of extra up-sales based on their algorithms. Whether these sales are truly attributable to the algorithms is unknown, and the number is so large that no one wants to turn them off for a longer period to do a more significant investigation.

Anyone for poker?

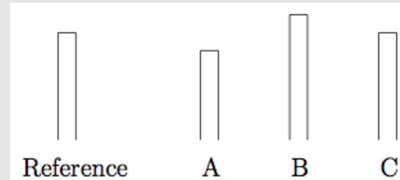
Once the project has kicked off, teams are often asked to break the coarse functionality down into fine grained tasks and estimate these individually. Within some agile methods this is done to decide how much work the team can commit to completing in the iteration. The *de facto* method in use today is Planning Poker [Poker].

Planning Poker has some theoretical basis. By asking the whole team to make independent estimates we make use of diverse opinions [Berra]. Diverse opinions (aka the wisdom of crowds) harnesses the 'Diversity Prediction Theorem', which basically says that a "diverse crowd always predicts more accurately than the average of the individuals". Of course this depends on the the group being diverse. How diverse is your team?

The team will discuss the task, but are supposed not to talk about task size. Each member estimates the size 'in secret' and then the whole team reveal their estimation at the same time. This is intended to avoid the phenomenon of 'anchoring', which is when some members of a group are influenced by the opinions of others – which would negate their independence. In case of disagreement, however, planning poker then requires further rounds of discussion and estimation, during which anchoring becomes very evident.

Anchoring

In an experiment conducted in 1956, a group was asked to compare the lengths of a reference line with three other lines marked A, B, and C.



Each group member was asked, in turn, to compare A, B and C to the reference line and decide whether it was longer, shorter or the same length. In groups that had members, planted by the researcher, answer first with purposefully wrong answers, it was found that the rest of the group then went on to give incorrect answers about a third of the time. [Asch56]

Relativity

As has already been stated, humans have been found to be bad at estimating. However, we are somewhat less bad at comparing equivalent tasks. Estimating using story points harnesses this by asking the team to compare the current task to tasks already completed and scoring it accordingly. In this way we make our estimates relative.

This benefit is clearly hard to realise at the beginning of a project when there is little historical data to go on. It's also difficult to make comparisons when the task being estimated is different to anything the team has done before. And if the stories being estimated are large, then it is hard to to make realistic comparisons.

For most teams I have worked with, the biggest single change that can make their task estimation more accurate is to break every story down into several, much smaller stories. This is something that most teams find incredibly hard, not helped by the belief that agile methods require each story to deliver a complete piece of end-user functionality. The team's 'definition of done' should allow teams to use low-fidelity [Scotland] stories to incrementally deliver valuable functionality, but is being generally misapplied to keep stories large.

One of the best examples I've heard about is from when Matt Wynne worked at Songkick [Wynne]. They systematically decomposed their stories till they reached a size that could usually be fully implemented in 1 day. They were then able to skip the estimation phase entirely and simply predict how many stories could be delivered in an iteration. How good does that sound?

No estimates

Over the past year or so there has been a #NoEstimates thread running on Twitter. Two of the major proponents are Woody Zuill [Woody] and Neil

Killick [Killick] who argue that since estimates are little more than guesses they deliver no value. Work that delivers no value is waste and so should be avoided.

The debate was interesting and useful, but, as Ron Jeffries pointed out [Jeffries] this is a reminder of an idea that was discussed earlier this century by Arlo Belshee and Joshua Kerievsky. Ron also pointed out some issues that most teams will have getting their organisation to agree to working without estimates. I'm not going to repeat the whole discussion here – go read the blog posts :)

Instead, I'll relate a story about a team I was working with earlier this year. They were very keen to work in a more responsive and responsible way, with all team members collaborating during the development process. The company, like so many, has serious cost constraints which led the executives to want more certainty around project costs, for which they needed ever more accurate estimates.

I tried to persuade the team that this extra estimation work would mean they had *less* time to deliver value to the business. I explained the tension between detailed, up-front estimation of a project and more lightweight just-in-time, last-responsible-moment techniques. The team accepted all my arguments, but would respond with "Yes, but in this financial situation they need to know how much it will cost." They didn't feel it was a battle worth fighting, and I believe that this is a common situation.

What a Bohr

Niels Bohr is credited with having said that "Prediction is very difficult, especially about the future." There's no evidence that Bohr ever said this [Bohr], so I feel justified in modifying the statement to "prediction is very difficult, even about the past".

We don't know how long a project is going to take until we do it and even once we've done it we a) don't really know how long it took us and b) won't know how long it would take us to do again. And yet we are regularly asked for estimates and regularly give them. In a bid to reframe the debate, Allan Kelly wrote a 'Dear Customer' letter [Kelly] explaining how estimates are being used as tools to try and shift the risk one way or the other. Any claim that they are scientifically derived is questionable at best.

Can you quote me for that?

There is another famous quotation, dubiously attributed to Disraeli which describes 'the persuasive power of numbers, particularly the use of statistics to bolster weak arguments'. [Disraeli] Irrespective of its actual provenance, I think it is equally applicable to the realm of estimates: 'There are lies, damn lies and estimates'.

Estimates produced before a project starts are lies about how much something will cost, usually tailored depending on whether the source of the estimate wants the project to go ahead or not. Estimates produced once a project has started are lies that compensate for the inaccuracies of earlier estimates. Both contribute towards an illusion of control that is no more real in software than it is in civil engineering (see the Edinburgh Tram project, for example [Edinburgh]).

Until customer and development team operate from a basis of trust, estimates will remain the weapon of choice. They will continue to be misinterpreted as commitments, and the next death march will always be just around the corner. But as Ron Jeffries says, the "old fogies know your estimates will be bogus, they know you won't get them right, they know you won't hit the deadline with full scope" [Jeffries2]. So, stay calm, make your best guess and have that estimate on my desk on Monday morning. ■

References

[Asch56] Asch, S. E. (1956). 'Studies of independence and conformity: A minority of one against a unanimous majority' *Psychological Monographs*, 70: 416

[Berra] http://vserver1.cscs.lsa.umich.edu/~spage/teaching_files/modeling_lectures/MODEL5/M18predictnotes.pdf

[Bohr] http://en.wikiquote.org/wiki/Niels_Bohr

[Bossavit] https://leanpub.com/leprechauns

[Bowle] http://blog.robbowley.net/2011/09/21/estimation-is-at-the-root-of-most-software-project-failures/

[Brier] http://en.wikipedia.org/wiki/Brier_score

[DeMarco] http://www.amazon.co.uk/Waltzing-Bears-Managing-Software-Projects/dp/0932633609

[Disraeli] http://en.wikipedia.org/wiki/Lies,_damned_lies,_and_statistics

[Edinburgh] http://en.wikipedia.org/wiki/Edinburgh_Trams

[Jeffries] http://xprogramming.com/articles/the-noestimates-movement/

[Jeffries2] http://xprogramming.com/articles/artifacts-are-not-the-problem/

[Kelly] http://agile.techwell.com/articles/original/dear-customer-truth-about-it-projects

[Killick] http://neilkillick.com

[McConnell] http://www.amazon.co.uk/Software-Estimation-Demystifying-Black-Art/dp/0735605351

[Poker] http://en.wikipedia.org/wiki/Planning_poker

[Rumsfeld] http://en.wikipedia.org/wiki/There_are_known_knowns

[Scotland] http://availagility.co.uk/2012/09/14/feature-injection-fidelity-and-story-mapping/

[Web1] http://predictionbook.com/

[Web2] https://www.goodjudgmentproject.com/

[Web3] http://calibratedprobabilityassessment.org/

[Woody] http://zuill.us/WoodyZuill/

[Wynne] Personal conversation

Calculating Your Brier Score

Confidence Quiz Answers

1) True; 2) True; 3) False; 4) True; 5) False

Total score: ________

Scoring - use the table below for statements that were in fact **true**:

Certainty:	Score:
0%	200
10%	162
20%	128
30%	98
40%	72
50%	50
60%	32
70%	18
80%	8
90%	2
100%	0

Scoring - use the table below for statements that were in fact **false**:

Certainty:	Score:
100%	200
90%	162
80%	128
70%	98
60%	72
50%	50
40%	32
30%	18
20%	8
10%	2
0%	0

YAGNI-C as a Practical Application of YAGNI

YAGNI can seem vague. Sergey Ignatchenko offers a more precise definition.

ALGOL 68 was the first (and possibly one of the last) major language for which a full formal definition was made before it was implemented.
~ C.H.A. Koster

... as a tool for the reliable creation of sophisticated programs, the language [ALGOL 68] was a failure ...
~ C.A.R. Hoare

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

The YAGNI ('You aren't gonna need it') principle is well-known in the agile world, going back to XP (as in 'eXtreme Programming', not 'Windows XP') in the end of 1990s. Unfortunately, this concept is too open to interpretation, which causes lots of confusion and heated debates both in industry [Fowler04] [Devijver08] [Litzenberger11] and in academia [Boehm02].

This article describes a practical approach to YAGNI, which has been tried in practical agile projects (one of which has had releases to millions of customers every 2–4 weeks). For the purposes of this article, we'll name it YAGNI-C (as 'YAGNI-Clarified'). While not being universal, we hope that YAGNI-C might be useful in quite a wide range of projects. Oh, and if somebody is about to say "Hey, we've been doing exactly the same things for years" – of course, YAGNI-C is not something really new; the problem is that such practices (which we think are best practices) are not often described, and therefore cannot be widely used.

The very beginning

The story starts when we're about to start our new agile project to make our super-duper app. The first question is – are we going to have some architecture? In general, it depends, but let's consider projects where architecture is essential, so the answer is 'yes'. The second question is – within the architecture chosen, are we going to have our own set of libraries (let's name them 'infrastructure libraries') which needs to be common for a significant part of project? Again, in general, it depends, but let's assume that in our project (for example, due to the project size/complexity) we've decided to have such a set of infrastructure libraries (it may be just a glue, or something more substantial – it doesn't matter too much, the key thing is that these libraries are supporting a big part of the whole project). Now, let's assume that APIs to these libraries are designed and supported by one or more people, let's name them 'library API maintainers' (with a hope that there is at least one of the architects involved in this group). Now let's try to define some principles and procedures of how 'library API maintainers' should approach YAGNI within our YAGNI-C model.

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

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

Thinking, not implementing

The First Principle in YAGNI-C is that 'thinking ahead is good, implementing ahead is bad'. While the second part of the First Principle is actually YAGNI in its pure form, the first part of the First Principle may need a bit more of explanation. There are numerous complaints out there (see, for example, [Fowler04]) that YAGNI is (or at least can be) misused to the point where any thinking about architecture is prohibited, and the project becomes a mess of ad hoc tactical decisions. The other side of the spectrum (library which does everything in sight) is also well-known to have led to disasters (ALGOL 68, DCE RPC, and especially X.500 are good examples of the over-designed systems which were so complicated that nobody was able to implement them properly). The First Principle above aims to strike the balance between these two undesirable extremes, and in practice it seems to work reasonably well (while in some cases, preliminary proof-of-concept prototypes may be needed before First Principle can be applied, starting from post-prototype development seems to work pretty well).

One other way to look at the First Principle is to rephrase it as 'as long as you can think about design without starting to implement it – you're fine, anything beyond that is over-design'. Essentially it restricts the amount of 'thinking ahead' to the amount of information which fits into the heads of the 'library API maintainers', which is subject to the cognitive limitations of the human brain, so it is fairly limited. In fact, the First Principle is much closer to the 'very lean' end of spectrum, while still allowing a certain amount of thinking ahead.

Specific cases

The Second Principle of YAGNI-C is 'if nobody in the team can describe a very specific use case for a problem – the problem doesn't exist' (and whatever doesn't exist doesn't need to be solved). This Second Principle is of extreme importance for the whole process to function. What it allows is the transfer of discussion from the space of "Hey, why are we not using XYZ?" and "Why we don't support paradigm ABC?" (which are subjects which can easily take months to deliberate on) to the space of very specific use cases, applicable to the current project, and while decisions might be not so obvious, at least it can be reasoned about not from the Swiftian big- and little-endians[1] point of view, but from the point of view where at least some logic can be applied.

Prohibit misuses

The Third Principle of YAGNI-C is 'If in doubt how it should behave – prohibit it'. If, as a library implementer, you don't have specification on a certain behaviour (for example, answering "what will happen if `x.g()`

1. Not to be confused with Intel/DEC little- and big-endians

if nobody in the team can describe a very specific use case for a problem – the problem doesn't exist

will be called before `x.f()`" is not specified, and you yourself have doubts about what will happen in this case) – you should prohibit such behaviour (for example, inserting an assertion, but other means are also possible).

This Third Principle is essentially a manifestation of agile principle known as 'deferring commitment'. In practice, whenever a library is released, people start using it in all kinds of ways, including those ways which were never intended. Prohibiting unintended uses (effectively deferring commitment of library writers regarding these uses) is good for at least two reasons: first, it makes code more reliable (making sure that caller really understands what is going on), and second, it allows implementation details to be hidden as deep as possible, reducing chances that library modifications which don't change the specification can break the client code.

How it works

Within YAGNI-C, the process of infrastructure API design is as follows.

1. First, library API maintainers design a very minimalistic API.
2. Then, people from the rest of the project start to come and say, "Hey, your library doesn't support this call, please add it." According to the second principle, each such request MUST be accompanied with a specific use case – "WHY this call is necessary?"
3. This is a point where library API maintainers perform analysis, deciding if a new call (class/...) should be added to the library API. As practice shows for good library design, about 30% of requests from step 2 above are turned down as "You're solving the wrong problem, what you really need for your use case is...", another 50% are turned down as "This can be done using existing API as follows:...", and remaining 20% lead to extending the APIs. And as 'library API maintainers' did think about potential requests (see the First Principle), implementing additional calls is normally not that a big deal.
4. Rinse and repeat from step 2.

It should be noted that YAGNI-C is substantially iterative, and therefore can't possibly work in strictly-waterfall development environments.

Example

Let's take a look and see on a specific example, how YAGNI-C might work in practice. Of course, this example is inherently extremely limited and oversimplified, but it still provides a good illustration of the concepts involved.

Let's assume that Alice is a library API maintainer, and one of the classes she develops, is a class `File` (Listing 1).

This class, as written, has a problem: it has an implicit copy constructor, which, combined with the nature of `~File()`, will cause all kinds of problems. Now, Alice faces a question: what to do about it? One thought quickly crosses her mind: to add something like Listing 2 but she quickly realizes that as there is no requirement to copy class `File`, this is a feature which would violate our First Principle. Now, to comply with both the First Principle and the Third Principle, she writes Listing 3.

```
class File {
  FILE* f;
  public:
  File( const char* filename ) {
    f = fopen( filename, ... ); }
  size_t read( char* buf, size_t bufsize ) {
    /* ... */ }
  void write( const char* buf, size_t bufsize ) {
    /* ... */ }
  ~File() { fclose( f ); }};
```

Listing 1

```
File( const File& ff )
{
  f = fdopen( dup( fileno( ff.f ) ) );
}
File& operator =( const File& ff )
{
  fclose(f);
  f = fdopen( dup( fileno( ff.f ) ) );
}
```

Listing 2

```
private: //copying/assigning of File
         //objects is prohibited
File( const File& );
File& operator=( const File& );
//in C++11, File( const File& ) = delete;
// and File& operator =( const File& ) = delete;
// can be used instead
```

Listing 3

This construct prevents other classes from calling the `File` copy constructor/assignment operator. This implementation is consistent with all the principles stated above.

Some time later, Bob comes to Alice and complains, "Hey, why don't you support a copy constructor for `File`?". Given such a request, it is impossible to judge if it has merits or not, as it is not clear exactly what problem Bob faces; formally, this request violates the Second Principle and is therefore denied. As a next step, Bob elaborates:

> The following piece of code doesn't compile:
> `void f( File ff ) { /* ... */ }`

Now request is specific enough, but it immediately becomes obvious (at least to Alice) that Bob has just forgot to put `&` in the function declaration, so his problem can be solved without introducing a copy constructor for the class `File`.

YAGNI is (or at least can be) misused to the point where any thinking about architecture is prohibited, and the project becomes a mess of ad hoc tactical decisions

At some point, Charlie comes to Alice and complains:

> Why is the assignment operator prohibited for class **File**? I want to copy a file and am trying to write:

```
File f1( filename1 );
File f2( filename2 );
f1 = f2;//compiler error"
```

Once again, when a specific use case is provided, it is obvious that adding an assignment operator to **File** would be quite the wrong thing to do to solve Charlie's problem, so Alice explains to Charlie how he can implement file copying for **File** (or she may decide to add static **File::copy()** for this purpose).

Summary

YAGNI-C is an attempt to clarify YAGNI so it becomes less vague and easier to follow. While YAGNI-C (though not under this name) has been successfully used for agile projects, it is certainly not a silver bullet, so you'll still think to think if it is beneficial for your project. It should not be considered as a new approach to development (we know many teams which follow these or very similar approaches), but as a new way to describe existing best practices. ■

References

[Boehm02] Boehm, B. , 'Get ready for agile methods, with care' *Computer*, vol 35

[Devijver08] Steven Devijver, 1The YAGNI Argument (You Ain't Gonna Need It)' http://architects.dzone.com/news/yagni-argument-you-aint-gonna-

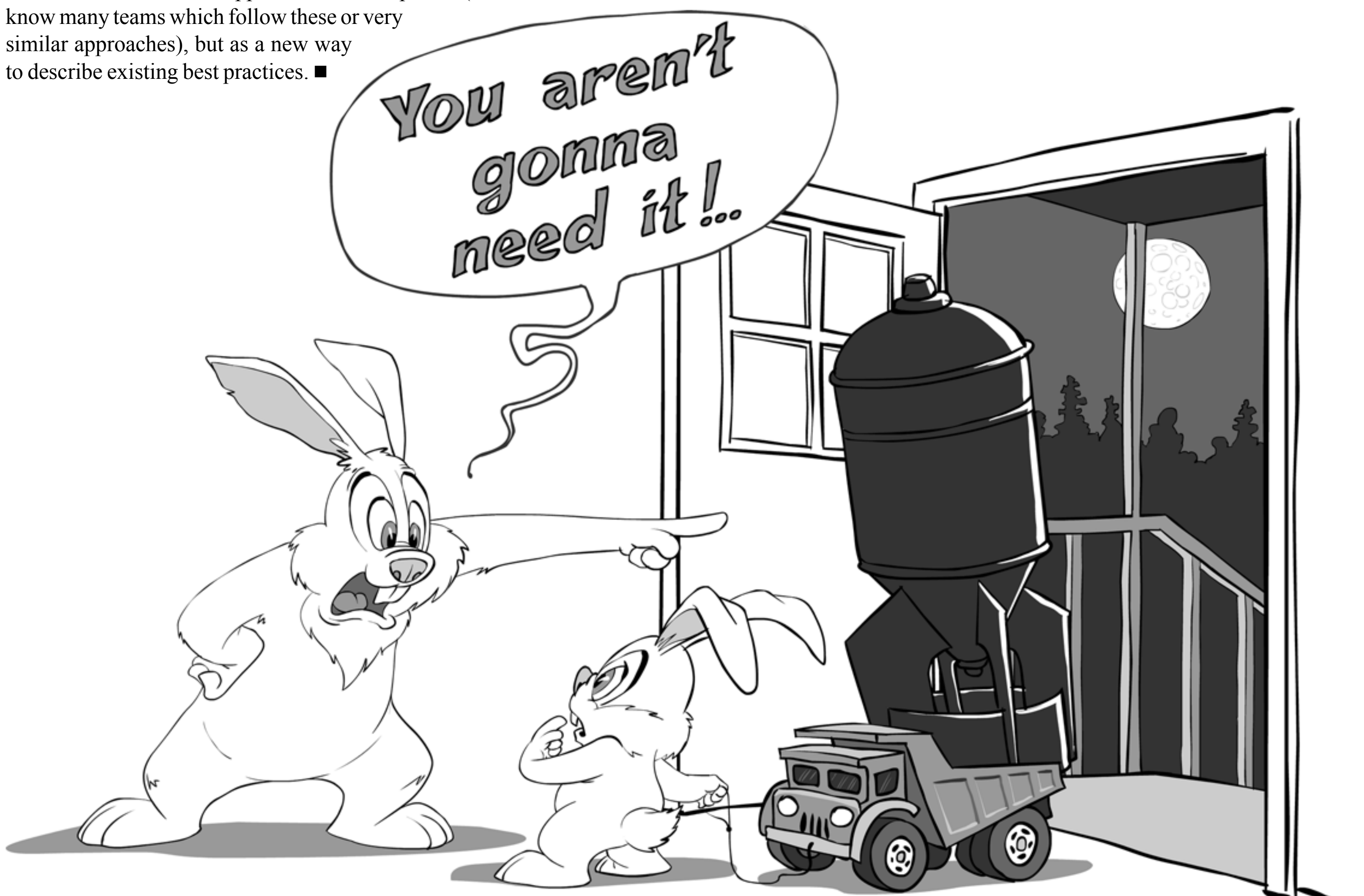
[Fowler04] 'Is Design Dead?' Martin Fowler, http://martinfowler.com/articles/designDead.html

[Litzenberger11] Dwayne C. Litzenberger, 'The Price of YAGNI' https://www.dlitz.net/blog/2011/02/the-price-of-yagni/

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

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.



Has the Singleton Not Suffered Enough

Singletons are much maligned. Omar Bashir considers why.

SINGLETON is an object creational design pattern that ensures only a single instance of a class can ever exist in an application and it provides a global point of access to that instance [GoF94]. The classical SINGLETON structure is the simplest among design patterns. It involves a single class with a private or protected constructor and a class method which returns the same instance of that class whenever it is invoked. Its simplicity leads to a temptation to use it instead of global objects and in situations that require single instances of implementing classes. For these reasons, it is a very widely used pattern.

Hahsler's quantitative study on adoption of design pattern also indicates that nearly 50% of Singleton implementations analysed were removed in subsequent maintenance [Hahsler04]. While Hahsler suggests overuse of the pattern due to its simplicity, these removals may also be due to the evolution in applications and their requirements as well as the context in which they are used resulting in the need for more than one instance of the previously SINGLETON classes. This may also be due to inadequacies of the classical structure of this pattern which focuses only on the creation of the SINGLETON instance but ignores most other aspects such as concurrency, substitutability, extensibility, lifetime management etc. and introduces coupling.

It is interesting to consider Ignatchenko's example of a flight simulator for a single one-engined aircraft that is constructed using singletons for the engine and the aircraft [Ignatchenko12]. After its resounding success, the company is asked to deliver a simulator for a twin engined aircraft. The developers have to go through a significant refactoring exercise to remove singletons from their code. It is important to note the underlying implications of the initial decision to use singletons and then the potential consequences of their removal.

A major implication of using SINGLETON is instance control. Once the instantiation of a class is controlled within a context, global accessibility of that instance within that context is natural. The classical description of the SINGLETON pattern intends the context to be the executing program. Therefore, singletons are globally accessible within applications that use them.

If a limited number of instances of a class are required within a context, the ability to construct objects without any restriction adds vulnerability to the application. Developers may mistakenly or unknowingly instantiate additional objects, which may result in inconsistent operation of the program at best. Therefore, instance control needs to be enforced for such classes. SINGLETON is a form of this enforcement. Furthermore, instance control may be considered separately from accessibility even within a context. If accessibility of singletons is restricted, they need to be passed to their dependants via interfaces which makes associations more obvious.

This discussion argues that the intent of the SINGLETON to limit instances of a class is the actual pattern and not the prescribed structure. Many of the criticisms of SINGLETON's classical structure may be addressed with alternatives, some of which are discussed here. Furthermore, the structure can be adapted to suit the requirements of the problem, the technology being used and the desired qualities of the system being implemented. Finally, dependency instantiation in the context of the dependants will be discussed. In this text, SINGLETON refers to the design pattern and singleton refers to only one instance of a class of which multiple instances cannot be created.

Criticisms

SINGLETON is largely considered similar to a global variable as it penetrates a scope via mechanisms less obvious than the public interface of the dependant. Therefore, it adds coupling to its dependants which reduces their flexibility and extensibility [Radford03]. This coupling is further exacerbated because a classical singleton usually requires the dependant to depend on a concrete class rather than an abstract interface. Therefore, it may be difficult to provide a mock or a stub of a singleton for unit testing its dependants.

The SINGLETON pattern only focuses on object creation and not its lifetime management. Even when a singleton is needed for the entire lifetime of the dependant application, issues may arise in the order in which a singleton is destroyed at program termination. This is because applications may run into issues if a singleton is destroyed while its dependants still hold a reference to it. This, therefore, requires mechanisms to manage the lifetime of singletons based on the context in which they are being used [Alexandrescu01], [Levine00]. However, this may not be a concern in managed environments where objects are only garbage collected when their references are no longer held.

Critics argue that the decision to create dependencies, the number of their instances and their lifetime management should rest with the domain and not the class itself [Radford03]. They prefer PFA (PARAMETERISE FROM ABOVE) where the application instantiates the required objects and passes them as dependencies via a public interface. This provides greater opportunities to decouple dependencies and dependants via abstract interfaces [Radford03], [Love06]. However, there are arguments that PFA violates information hiding as the application may require explicit knowledge of coupling between the dependencies and dependants [Radford03]. Furthermore, instantiating objects at the top level and passing them through several layers to the layer where these objects are required clutters the application layer with objects not required there and may also result in elaborate interfaces between layers to pass these as parameters.

Dependencies may be encapsulated into an object to simplify the interfaces of the dependants as described in the ENCAPSULATE CONTEXT pattern [Kelly04]. It may be argued that ENCAPSULATE CONTEXT achieves this through obscurity by collecting (at times unrelated) parameters into a single object. Balancing openness (by using finer grained context objects) with interface cluttering and coupling can be challenging. As discussed

Omar Bashir A programmable calculator with 0.5 kB of user memory and a very basic version of BASIC inspired Omar into computing in 1986. He has had development experience in defence, telecoms, logistics and finance. His current interests include distributed systems and interesting applications of design patterns. He can be contacted at obashir@yahoo.com

dependants should have **explicit information** regarding the **initialisation of their singleton dependencies**

later, instantiation of all dependencies at the application level may neither be appropriate nor feasible hence making ENCAPSULATE CONTEXT and its variants inappropriate solutions for such relationships.

SINGLETON initialisation is also not obvious in the pattern. If a singleton is initialised at construction then initialisation parameters may need to be passed via the global access point. This means that dependants should have explicit information regarding the initialisation of their singleton dependencies. This may be alleviated via environment variables but at the cost of further obscurity. Also, the behaviour to be expected if a dependant passes different configuration parameters than the one passed earlier is unclear. Should the new configuration parameters be ignored? Should the existing instance be deleted and a new instance created? Or should the existing instance be re-initialised? One option is to separate initialisation from construction and make initialisation an instance operation. Once a dependant obtains the dependency, it should be able to determine if the dependency is already initialised. If not, the dependant should be able to initialise it. If the initialisation parameters change, it may be possible to reinitialise the dependency. However, issues may arise if the dependency is shared between a number of dependants.

SINGLETON creation and initialisation become more challenging in multi-threaded environments. Synchronising the global access point for a singleton instance is the safest but the least optimal option. Furthermore, construction or initialisation may be time consuming operations that may have performance implications for the entire application. For construction, optimisation is suggested using DCLP (DOUBLE CHECKED LOCKING PATTERN) [Schmidt97]. However, it has its limitations too when porting to multiprocessing architectures and using optimising compilers [Meyers04]. Their remedies can, in turn, add considerable complexity to the creation and initialisation of a singleton.

Pattern is in the intent

The intent of SINGLETON is to provide a *mechanism* that allows only a single instance of a class and global access to that instance. Prescribing further than that starts making the pattern more implementation and technology specific. In fact, a strict prescription of any pattern structure has been criticised as practitioners tend to consider the structure as the pattern and not the intent and the context in which the pattern is to be applied [Petersen13]. Hence, even with slight changes in the implementation, application or the context, the classical SINGLETON structure may result in issues discussed above. Furthermore, some recent languages provide SINGLETON as a language feature making the SINGLETON structure somewhat redundant for these languages. A typical example is the object construct in Scala. It is, therefore, unfortunate that the pattern receives significant criticism where many critics are actually criticising the prescribed structure and its weaknesses.

The key characteristic of the pattern is instance restriction, i.e., limiting the number of instances of a class to only one within an application. Thus, the mechanism providing a singleton should only ever return the same instance whenever an instance is requested. This makes global access to this instance inherent to the pattern. This is the strongest distinction between a global and a singleton. Normally global instances have public constructors allowing any number of global and local instances of their classes to be created [GoF94]. SINGLETON aims to restrict public accessibility of the constructor so that it may only be called in a controlled manner allowing the developer to control the number of instances that can be created.

There have been arguments that dependency injection (DI) containers have the ability to provide singletons. However, DI mechanisms largely depend on public constructors of classes for which they provide instances. As discussed above, public constructors on a class allow a programmer to create an instance without using or bypassing DI thereby having multiple instances of the same class. So, these DI containers may claim to provide singletons only if these containers are used for instance creation and management. Some DI containers like Spring may provide support for the classical SINGLETON structure by allowing invocation of static methods on classes that return an instance of the respective classes. Furthermore, it can be argued that DI can also allow penetration of a scope in ways other than public interfaces, just as SINGLETON does.

Gang of Four (GoF) discuss variation of SINGLETON also referred to as Multiton or Limiton [Stencel08], which provides multiple controlled instances of a class. They also discuss subclassing in SINGLETON which can be useful for substitutability, lack of which in the classical structure hinders flexibility and testability of systems using singletons. However, their detailed description on providing these features focuses on adapting the classical structure which adds significant complexity to just one class and requires additional support for creating an instance from a type hierarchy.

Therefore, most technical issues related to SINGLETON can be attributed to the structure prescribed by GoF rather than the pattern itself. While the classical structure satisfies the intent of the pattern, it violates key object oriented design principles like separation of concerns and substitutability, leading to higher coupling, low cohesion and significant rigidity in its implementation and use.

Separation of concerns

The classical SINGLETON structure exhibits low cohesion as it requires a single class to perform instance creation, instance management and provide the required functionality. This inhibits the extension or variation of these orthogonal aspects of the SINGLETON structure. Separating these non-overlapping and unrelated aspects into different classes allows independence between instance control and functionality of the singleton which enables each to be varied independently and increases cohesion in the respective classes.

`SingletonHolder` is a well documented SINGLETON implementation in C++ that focuses on the intent of the pattern and uses separate classes to perform singleton object creation, instance management and thread safety [Alexandrescu01]. **`SingletonHolder`** itself is not a singleton and does not contain any application or domain functionality. All construction methods of the application specific class whose instance is to be managed by **`SingletonHolder`** are private to avoid unintended construction. This

the mechanism providing a singleton should only ever return the same instance whenever an instance is requested

```
public class SingletonFactory {
  private static
     Map<Class<?>,Object> instanceMap;
  static{
    instanceMap = new HashMap<Class<?>,Object>();
  }

  public <T> T getInstance(Class<T> type)
                throws InvocationTargetException,
                       IllegalAccessException,
                       InstantiationException,
                       NoSuchMethodException{
    if (!instanceMap.containsKey(type)){
      Constructor<T> constructor =
         type.getDeclaredConstructor();
      constructor.setAccessible(true);
      instanceMap.put(type,
                     constructor.newInstance());
    }
    return (T) instanceMap.get(type);
  }
}
```

Listing 1

requires a friendship to be declared between the application specific class and the class implementing the creation policy. `SingletonHolder` is an example of policy-based design. Type parameterisation allows customisation of instance creation, thread safety and singleton lifetime management independently of the functionality of the singleton object a `SingletonHolder` manages.

Absence of the friend relationship in Java and the difference between Java Generics and C++ templates will not allow a similar implementation in Java. Classes whose constructors have restricted access may not be instantiated except through reflection. Furthermore, it is not possible to declare a static variable of generic type in Java to hold the reference to the singleton instance. A minimal Java-based `SingletonFactory` that only instantiates a singleton and holds its reference is shown in Listing 1.

The instance created by an instance of `SingletonFactory` is inserted into a map (`instanceMap`) with the key being the class of the instance created. A `SingletonFactory` instance creates an instance of a specified class only if no other instance of that class exists in the `instanceMap`. The `getInstance()` method always returns a class's instance from the `instanceMap`. The following snippet shows the usage of `SingletonFactory` where the constructor of `Singleton1` class is private.

```
...
SingletonFactory factory =
   new SingletonFactory();
Singleton1 o1 =
   factory.getInstance(Singleton1.class);
...
```

While using reflection to access private members of a class may raise eyebrows, the `SingletonFactory` implementation removes the instantiation logic from the application/domain class allowing `SingletonFactory` to be implemented only once and adapted independently of the domain/application logic contained in the singleton classes.

Extensibility and substitutability

GoF suggest extensibility in the classical SINGLETON through subclassing, which also allows substitutability. They mention selection of an instance of a specific subclass as a key challenge in a SINGLETON type hierarchy. Some of the solutions they suggest include using a registry of instances of SINGLETON classes in a hierarchy, statically linking a specific SINGLETON subclass and using a single method in the base class for the entire hierarchy to determine the actual subclass and return its instance. Some dependency injection containers like Spring support classical SINGLETON by allowing invocation of a static method on a specified type to obtain an instance. This can support SINGLETON type hierarchies in an extensible manner. Therefore, implementation of a SINGLETON type hierarchy and its usage remains a technology specific issue.

Furthermore, if a single class variable is used in a classical SINGLETON implementation to hold the reference of the instance and it is exposed to subclasses via protected accessibility, then only one instance of that entire type hierarchy exists in the application. This is again a decision that should not rest with the implementation rather it should be ascertained from the domain. If the domain allows each class within a hierarchy of singletons to have an instance each (and no more), the implementation should support it. Adaptations to the classical SINGLETON implementation may allow this, for example, using a private instance variable for each class in the hierarchy or using an associative container in the base class mapping the concrete type to the instance. The `SingletonHolder` implementation [Alexandrescu01] and the `SingletonFactory` discussed above provide this facility by default.

Substitutability is very strongly desired in singletons as coupled with dependency injection it enhances the testability of the dependant components. Substitutability has been implicitly discussed in the SINGLETON pattern via subclassing only. If singletons (classical or otherwise) implement abstract interfaces and there is support for dependency injection of these singletons, extensibility and testability of the dependant application can be enhanced considerably.

Injecting singletons

Dependency injection (DI) attempts to decouple dependants and dependencies by moving instantiation and initialisation of dependencies from the dependants to a DI framework component normally referred to as injector or container. Dependants identify their dependencies using identifiers which map to the specification of dependencies in the injector's configuration. Thus, dependencies can simply be changed by altering the injector's configuration. Therefore, DI along with interface-based

it is possible to create the same package within a different project thereby circumventing these access restrictions

development has enhanced the configurability, extensibility and testability of object oriented software.

Some DI frameworks like Spring allow specification of a static method on a class, the execution of which returns an instance of that class. Thus Spring provides direct support for injecting classical SINGLETON objects into their dependants. If these SINGLETON classes implement an abstract interface, instantiating them via Spring provides all of the above abilities. Furthermore, restriction on instantiation within the classes ensures that even if a Spring injector is not used in some part of the application and a dependant attempts to obtain instances directly by calling static methods of these classes, the same instances of respective classes are returned.

Using a decoupled singleton instantiation mechanism like the **`SingletonFactory`** (Listing 1) with dependency injection may require obtaining an instance of the factory and then using the factory instance in the dependant to obtain the required singleton instance. As discussed earlier, this provides opportunities to extend **`SingletonFactory`** for specialised construction of individual singletons independently of their classes.

Singletons exist in context

In most cases, limiting the number of instances of a class within an application depends on the requirements of the application or a particular context in which the instance is required. For example, only one instance of a particular screen (e.g., an application configuration editor) is allowed within an application whereas many instances of other screens can be created. In other cases, infrastructure objects like network and database connections and logging utilities are singletons or multitons (e.g., the `Logger` class in log4j).

Thus, there may always be a requirement for limiting the number of instances of a class based on the requirements of the application. While such instances can be created at the application layer and passed to the corresponding modules or layers using PfA, the public accessibility of constructors of their classes poses a risk that a developer may instantiate a new object elsewhere in the system. Undetected, this can result in inconsistent operation of the system at best.

Thus, the requirement to control instantiation of specific classes is orthogonal to parameterisation from above. SINGLETON instances may still be created at the top layer and passed around via interfaces if a greater visibility of associations between dependants and dependencies is required. Restrictions on constructor invocation ensures that no unintended instances exist within the applications.

Therefore, if the instantiation of classes needs to be controlled, accessibility of constructors of these classes needs to be limited. Furthermore, the mechanism to return instances of such classes depends on the context in which these classes are being instantiated. The context should define the number of instances that can be created, the specific instance to be returned and the concrete type of the instance being returned. The rigid Singleton structure defined by the GoF does not provide such flexibility in object instantiation.

Revisiting Ignatchenko's example flight simulator [Ignatchenko12], not implementing the aircraft and engine as singletons allows the extension of this simulator to a multi-engine multi-aircraft application. However, some form of instance control is still needed in this example so that engines, for example, are instantiated only in the context of aircraft. This would allow only one engine for a single engined aircraft and two engines for a twin engined aircraft.

Contextualising instantiation

The context for a dependency is its dependants. A dependant should be able to specify or manage the instances of its dependencies. Considering the example of aircraft and engines mentioned above, if engines are dependencies of aircraft, the type and number of engines for a particular aircraft exist in the context of that aircraft. Thus, a single engine jet aircraft will have one jet engine but a twin piston engine aeroplane will have two piston engines.

In contrast, both the SINGLETON pattern as well as PFA consider dependency instantiation outside the context of the dependant. SINGLETON is especially suited if the context is the entire application rather than its finer components. PFA may be used to instantiate the required number and types of dependencies and populate the dependant objects only if the higher level layers have the contextual knowledge about the dependencies and the relationships between them and their dependants. This may, however, violate the principle of information hiding and introduce unintended coupling. GoF also describe a variant of SINGLETON that can return multiple instances of a given type in a controlled manner [GoF94]. But that may also require the SINGLETON implementation to know about the relationships between the dependencies and dependants and may also make the instantiated dependencies accessible to objects other than the dependants. Thus it is not possible to ensure dependants to have exclusive ownership of singleton dependencies

This leads to a fundamental requirement of making the dependency construction mechanism accessible only to the corresponding dependants. Accessibility at this level of granularity is currently not available in most object oriented languages. In C++ this may be achieved via a friend relationship between dependencies and their dependants as in the **`SingletonHolder`** [Alexandrescu01] implementation. Java does provide additional levels of access restrictions where protected and package-private members can only be accessed by classes within a package. However, it is possible to create the same package within a different project thereby circumventing these access restrictions and accessing such members of the same package in a different project. Scala can scope access restriction on a class to other packages enclosing the package containing that class. However, it suffers from the same accessibility loophole as Java. C# allows internal members to be accessible only by members from the same assembly or an assembly explicitly specified as the former's friend. Relying only on these measures for controlling the access to dependency instantiation would require the dependants to be defined in specified packages for Java and Scala. In case of C#, reuse of dependencies may be severely restricted.

The context determines the type and the number of dependencies it needs and the factory creates those instances

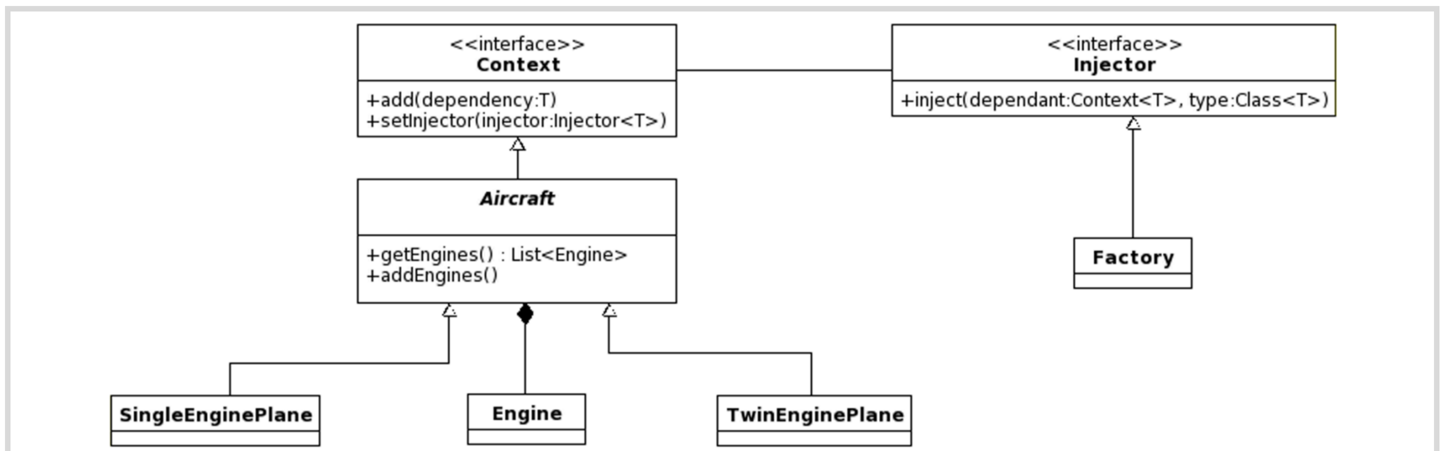


Figure 1

In managed languages such as Java and C#, private members can also be accessed via reflection. As using reflection to access private members is not considered good practice, it should not be used arbitrarily. Instead, dependency instantiation may be delegated to a factory as in the case of `SingletonFactory` above. The dependant (i.e., the context) and the factory hold references to each other. The context determines the type and the number of dependencies it needs and the factory creates those instances and injects them into the context. The context now holds references to its dependencies. It may make them available to other objects and manage their lifetimes as required. Finally, there may no longer be a need for static members to hold references to dependencies.

Figure 1 shows the class structure of a solution to the problem of instantiating engines for aircraft. The `Factory` class implements the `Injector` interface. The inject method takes an instance of an implementation of the `Context` interface as the dependant and the type of the dependency to be created. The `Factory` instance creates this dependency instance and injects it into the dependant by calling the add method implementation. Listing 2 shows the code for `Context` and `Injector` interfaces and the `Factory` class.

The `Aircraft` abstract class implements the `Context` interface and specifies methods relevant to aeroplanes in this example. It contains a map that holds the objects of the `Engine` class for a specific instance of the `Aircraft` class. The `getEngines()` method returns a list of `Engine` objects for the corresponding `Aircraft` instance. Two classes, `SingleEnginePlane` and `TwinEnginePlane` extend the `Aircraft` abstract class and implement the `addEngines()` method. The implementation in the `SingleEnginePlane` allows the instantiation of one object of the `Engine` class if none exists. The implementation in the `TwinEnginePlane` allows the instantiation of two objects of the `Engine` class if they do not exist already. These instantiations are performed using an instance of an `Injector` implementation, i.e., the `Factory` class in this case. Figures 2 and 3 show these interactions and the code for these classes is listed in Listing 3.

```
public interface Context<T> {
  void add(T dependency);
  void setInjector(Injector<T> injector);
}
public interface Injector<T> {
  void inject(Context<T> dependant,Class<T> type)
         throws InvocationTargetException,
                IllegalAccessException,
                InstantiationException,
                NoSuchMethodException;
}
public class Factory<T> implements Injector<T>{
  private T getInstance(Class<T> type) throws
              InvocationTargetException,
              IllegalAccessException,
              InstantiationException,
              NoSuchMethodException{
    Constructor<T> constructor =
       type.getDeclaredConstructor();
    constructor.setAccessible(true);
    return (T) constructor.newInstance();
  }
  @Override
  public void
     inject(Context<T> dependant,Class<T> type)
        throws InvocationTargetException,
               IllegalAccessException,
               InstantiationException,
               NoSuchMethodException{
    dependant.add(getInstance(type));
  }
}
```

Listing 2

implementing instance control or restriction is non-trivial at best and SINGLETON in this regard is only of limited use

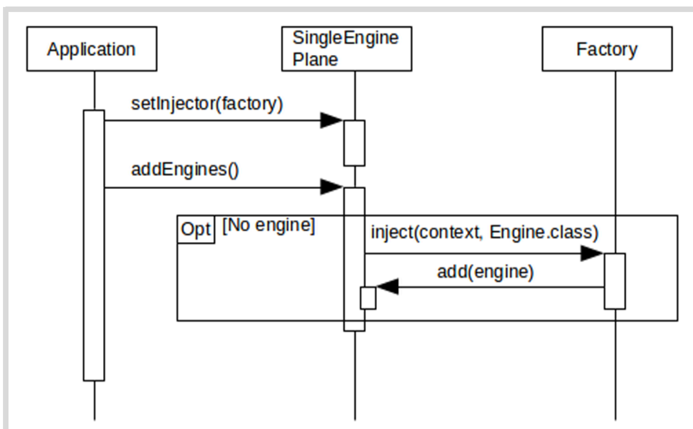


Figure 2

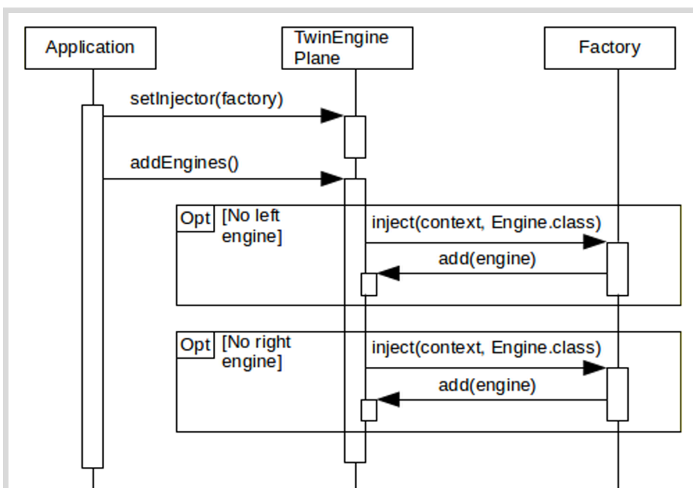


Figure 3

The following code snippet shows the usage of these classes,

```
SingleEnginePlane plane1 =
   new SingleEnginePlane();
TwinEnginePlane plane2 = new TwinEnginePlane();
Factory<Engine> factory = new Factory<>();
try{
  plane1.setInjector(factory);
  plane2.setInjector(factory);
  plane1.addEngines();
  plane2.addEngines();
  ...
} catch (Exception exp){
  ...
}
```

Hence, dependencies are only instantiated in the context implemented in the dependants. While **`Factory`** uses reflection to invoke private constructors of the specified classes, it only injects them into their dependants. Therefore, dependencies with restricted constructors cannot be instantiated using a **`Factory`** instance without a context. None of the participants here have any static variables nor are there any classical singletons within this structure.

Conclusions

The discussion above suggests that issues with the SINGLETON pattern arise from its structure which violates most key principles of effective object oriented design. The intent of this pattern, which is primarily to restrict the instantiation of a class to have only one instance, is largely unsupported in most programming languages. Scala offers object as a SINGLETON implementation. Some of the issues of the pattern can be resolved by altering its structure so that various orthogonal requirements of instance creation, lifetime management and concurrency are delegated to additional classes. If these additional classes support substitutability, requirement for different operating environments and applications may be supported. This leaves the class of which a singleton is to be instantiated to focus only on domain related functionality. Without the need for static variables in that class, its extensibility and substitutability may be considerably enhanced.

SINGLETON needs to be viewed as a broader issue of controlling instantiation of specified classes in the contexts in which their objects may be used. Contexts for such classes exist in their dependants and hence their dependants must be able to control the instantiation of these dependencies. Additionally, instance control can only be achieved by controlling the accessibility of constructors of respective classes. However, such fine grained accessibility that allows classes to specify other classes that may access their private members is not commonly available. In C++, this may be achieved via friend relationship between dependants and dependencies where as in managed languages like Java and C#, reflection may be exploited.

A structure in Java is discussed that uses a factory employing reflection to instantiate objects of classes that have private constructors. Dependants contain the context in which dependencies are to be instantiated and invoke a factory accordingly to obtain those instances. The factory instance uses a reference to the context (i.e., dependant) and injects the dependencies it instantiates directly into the associated dependant. This structure, while more complex than the classical Singleton structure, is flexible, extensible, testable and has better support for concurrency.

This reaffirms that implementing instance control or restriction is non-trivial at best and SINGLETON in this regard is only of limited use, i.e., enforcing a single instance of a class within an application. Instance control at finer granularities without language support requires structures more elaborate than the classical SINGLETON structure. ■

```
public abstract class Aircraft
   implements Context<Engine> {
  protected Map<String, Engine> engines;

  public Aircraft(){
    this.engines = new TreeMap<>();
  }
  public abstract void addEngines()
     throws Exception;
  public List<Engine> getEngines(){
    List<Engine> engineList = new LinkedList<>();
    for(Map.Entry<String, Engine> entry :
       this.engines.entrySet()){
    engineList.add(entry.getValue());
  }
  return engineList;
  }
}

public class SingleEnginePlane extends Aircraft {
  private Injector<Engine> injector;
  private final String id = "Main";
  @Override
  public void add(Engine dependency) {
    dependency.setId(id);
    this.engines.put(id, dependency);
  }
  @Override
  public void
     setInjector(Injector<Engine> injector) {
    this.injector = injector;
  }
  @Override
  public void addEngines() throws Exception{
    if (!this.engines.containsKey(this.id)){
      this.injector.inject(this, Engine.class);
    }
  }
}
```

Listing 3

```
public class TwinEnginePlane extends Aircraft {
  private final String leftEngineId =
     "Left Engine";
  private final String rightEngineId =
     "Right Engine";
  private Injector<Engine> injector;
  @Override
  public void add(Engine dependency) {
    if (!this.engines.containsKey
       (this.leftEngineId)){
      dependency.setId(this.leftEngineId);
      this.engines.put(this.leftEngineId,
                       dependency);
    } else if (!this.engines.containsKey
       (this.rightEngineId)){
      dependency.setId(this.rightEngineId);
      this.engines.put(this.rightEngineId,
                       dependency);
    }
  }
  @Override
  public void
     setInjector(Injector<Engine> injector) {
    this.injector = injector;
  }
  @Override
  public void addEngines() throws Exception {
    if (!this.engines.containsKey
       (this.leftEngineId)){
      this.injector.inject(this, Engine.class);
    }
    if (!this.engines.containsKey
       (this.rightEngineId)) {
      this.injector.inject(this, Engine.class);
    }
  }
}
public class Engine {
  ...
}
```

Listing 3 (cont'd)

References

[Alexandrescu01] Andrei Alexandrescu, Scott Meyers and John Vlissides, *Modern C++ Design: Applied Generic and Design Patterns (C++ in Depth)*, Addison Wesley, 2001.

[Levine00] David L. Levine, Christopher D. Gill, Douglas C. Schmidt, *Object Lifetime Manager, in Design Patterns in Communications* edited by Linda Rising, Cambridge University Press, 2000, pp

[GoF94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design patterns : Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[Hahsler04] Michael Hahsler, 'A Quantitative Study of the Adoption of Design Patterns By Open Source Software Developers' in *Free/Open Source Software Development* by Stefan Koch, IGI Publishing, pp103-124.

[Ignatchenko12] Sergey Ignatchenko, 'Keep it Simple, Singleton!' ACCU (Association of C and C++ Users) *Overload* #111, October 2012, pp 19–20.

[Kelly04] Allen Kelly, 'The Encapsulate Context Pattern' ACCU (Association of C and C++ Users) *Overload* #63, October 2004.

[Love06] Steve Love, 'The Rise and Fall of Singleton Threaded' ACCU (Association of C and C++ Users) *Overload* #73, June 2006, pp 18–21.

[Meyers04] Scott Meyers, Andrei Alexandrescu, 'C++ and The Perils of Double-Checked Locking: Part I' *Dr Dobbs Journal*, July 2004.

[Petersen13] Adam Petersen, 'The Signs of Trouble: On Patterns, Humbleness and Lisp' ACCU (Association of C and C++ Users) *Overload* #113, Feb 2013, pp 12-13.

[Radford03] Mark Radford, 'SINGLETON – The Anti-Pattern' ACCU (Association of C and C++ Users) *Overload* #57, Oct 2003, pp 20-22.

[Schmidt97] Douglas Schmidt, Tim Harrison, *Double-Checked Locking-An Optimisation Pattern for Efficiently Initialising and Accessing Thread-safe Objects, Pattern Languages for Program Design 3*, edited by Robert Martin, Frank Buschmann and Dirke Riehle, Addison Wesley, 1997.

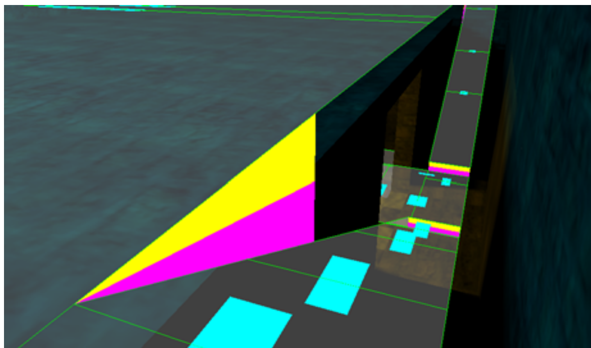
[Stencel08] Krzysztof Stencel and Patrycja Wegrzynowicz, 'Implementation Variants of the Singleton Design Pattern' *OTM 2008 Workshops* LNCS 5333, pp 396–406, Springer-Verlag, 2008

Automatic Navigation Mesh Generation in Configuration Space

Walkable 3D environments can be automatically navigated. Stuart Golodetz demonstrates how navigation meshes achieve this.

The representation of the walkable area of a 3D environment in such a way as to facilitate successful navigation by intelligent agents is an important problem in the computer games and artificial intelligence fields, and it has been extensively studied. As surveyed by Tozour [Tozour04], there are a variety of common ways to represent such an environment, including:

- *Regular Grids*. These support random-access lookup but do not translate easily into a 3D context. They also use a lot of memory and can yield aesthetically unpleasing paths for navigating agents.
- *Waypoint Graphs*. These connect large numbers of nodes (often manually placed) using edges that imply walkability in the game world. They were previously popular in games but are costly to build and tend to constrain agents to walking 'on rails' between connected waypoints.
- *Navigation Meshes*. These represent the walkable surface of a world explicitly using a polygonal mesh. Polygons within a navigation mesh are connected using links that imply the ability of the agent to walk/step/jump/etc. between them (see Figure 1).



An example navigation mesh and its links: cyan = walk link; magenta = step up link; yellow = step down link. (Note that in greyscale, the walk links are light grey, the step up links are dark grey and the step down links are white.)

Figure 1

Since their introduction by Greg Snook [Snook00], navigation meshes have proved to be a particularly successful approach due to their ability to represent the free space available around paths through the world (this is extremely useful because it provides the pathfinder with the information it needs to successfully avoid local obstacles). As a result, they have seen widespread use in both games themselves, and popular games engines such as Source and Unreal, and many games authors have contributed to their theoretical development (most notably in the *Game Programming Gems* and *AI Game Programming Wisdom* book series). There has also been significant interest from researchers in academia (e.g. see [Hale09, Kallmann10, Pettré05, VanToll11]).

One facet of using navigation meshes is how to build them in the first place, and numerous methods have been described in the literature. An early approach due to Tozour [Tozour02] works by first determining the walkable polygons in a 3D environment by comparing their normals[2] with the up vector, and then iteratively merging together as many polygons as possible using the Hertel-Mehlhorn algorithm [Hertel83, ORourke94] and a 3 to 2 merging technique. Hamm [Hamm08] generates a navigation mesh using an empirical method that involves sampling the environment to create a grid of points, identifying a subset of points both on the boundary of and within the environment, and connecting these points to form a mesh. Ratcliff [Ratcliff08] creates a navigation mesh by tessellating all walkable surfaces in the world, merging the results together to form suitable nodes and then computing links between neighbouring nodes. Van Toll *et al.* [VanToll11] build a navigation mesh for a multi-layer environment by constructing a mesh based on the medial axis for each layer and then connecting the medial axes by 'opening' the connections between the layers. The same authors also demonstrate how such a mesh can be dynamically modified [VanToll12]. Mononen's open-source Recast library [Mononen09] first voxelizes[3] the 3D environment before running a watershed transform [Beucher90, Gonzalez02] on the walkable voxels and creating a mesh from the resulting partition of the walkable surface.

In this article, I describe the implementation of navigation mesh construction in my homemade *hesperus* engine [hesperus], based heavily on the techniques of van Waveren in [VanWaveren01]. The goal is to provide a helpful, implementation-focused introduction for those with no prior experience in the area. At a high-level, the method is as follows:

- Firstly, given a 3D environment made up of brushes (simple convex polyhedra, each consisting of a set of polygonal faces), and a set of axis-aligned bounding boxes (AABBs) used to represent the possible sizes of the agents that will navigate the environment, expand the brushes by appropriate amounts (see the 'Configuration Space' section) to create a set of expanded brushes for each AABB.

Stuart Golodetz obtained his DPhil in Computer Science in 2011, working on 3D image segmentation and feature identification. He has since spent two interesting years in industry, working in the areas of credit risk management, logic programming and software analytics. His areas of interest include medical image analysis, computer games development and the intricacies of different programming languages, especially C++.

2. A normal to a polygon is a vector that is perpendicular to the plane containing the polygon. In practice, since polygons are often oriented (i.e. they are considered to have a front face and a back face), it is common for implementations to calculate normals that point out of the front faces of polygons. It is also sometimes convenient for implementations to normalise the calculated normals (that is, to make them unit length), in which case normals for polygons are uniquely defined and we can talk about the normal rather than a normal to a given polygon.
3. A voxel (or *volume element*) is the 3D equivalent of a pixel, so voxelizing a 3D environment means converting from e.g. a polygonal representation of the environment to a representation consisting of many small cubes.

The goal is to provide a **helpful, implementation-focused introduction** for those with **no prior experience** in the area

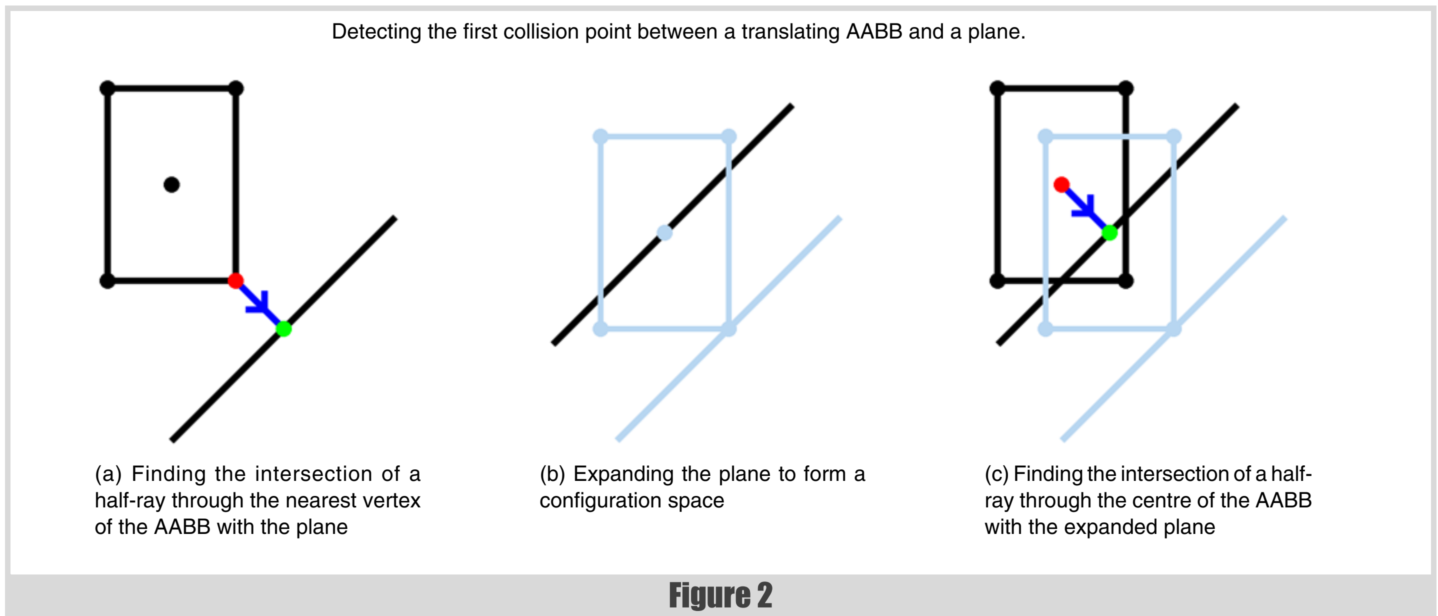


Figure 2

- Next, using constructive solid geometry (CSG) techniques, union the expanded brushes for each AABB together to form a polygonal environment. Filter the polygons of the environment to find those that are *walkable* (judged by comparing their face normals with the up vector). This gives us the polygons of a navigation mesh for each AABB, but without any links to indicate how agents should navigate between them. See the 'Basic Mesh Generation' section.
- Finally, generate walk and step links between the polygons of each navigation mesh (see the 'Walk and Step Links' section) – these indicate, respectively, that an agent can walk from one polygon of the mesh to another, or step up/down from one polygon to another. These links can be used to generate a graph for the purposes of path planning.

The following sections look at each of these steps in more detail.

Configuration space

When planning the movement of intelligent agents (e.g. robots), a *configuration space* is the space of possible configurations in which an agent can validly exist. As a first example of what this concept means and when it can be useful, consider detecting the first point of collision between a plane and an AABB that is moving by translation only – this might normally involve determining which vertex of the AABB is nearest to the plane and finding the point at which a half-ray oriented in the AABB's direction of movement and starting at that vertex would intersect the plane (see Figure 2(a)). The configuration space alternative is to initially expand the plane in the direction of its normal by a fixed amount so that the centre of the AABB touches the expanded plane precisely when the nearest vertex of the AABB touches the non-expanded plane (see Figure 2(b)). The first point of intersection can then be calculated by finding the point at which a half-ray starting at the centre of the AABB would touch the expanded plane (see Figure 2(c)) – there is no longer a need to first determine which vertex is nearest to the plane. Put another way: by expanding the plane, we have created the space of possible configurations for the centre of the AABB, and thereby restricted our testing to making sure that the AABB's centre is always in a valid location.

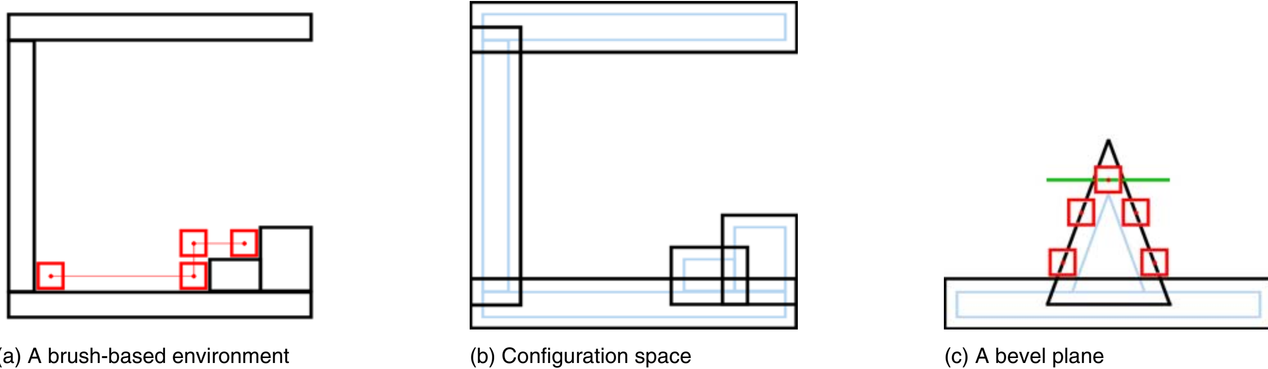
As explained in [VanWaveren01], the concept of configuration space can be extended to an entire 3D environment, allowing us to test an agent represented as an AABB against such an environment using only point-based (rather than AABB-based) intersection tests: this was the approach taken in the popular *Quake III Arena* game. Starting with a *brush-based* 3D environment (i.e. one that is built up by combining instances of simple convex polyhedra such as cuboids, cylinders and cones – a common approach in 3D world editors), a configuration space for agents with a specific AABB can be constructed by expanding each brush by an appropriate amount (see Figures 3(a) and 3(b)). Note that expanding each brush correctly can require the introduction of additional *bevel* planes as described in [VanWaveren01] (see Figure 3(c)).

Basic mesh generation

Brush unioning

Having expanded the brushes of a brush-based environment to construct a configuration space for an AABB in the manner described, the next step is to union the expanded brushes together to generate a set of polygons that represent the expanded environment as a whole. These polygons can then be processed further to construct a navigation mesh.

A configuration space can be generated for the entirety of a brush-based environment by expanding all of the brushes by an appropriate amount: (a) shows a brush-based environment, together with the range of movement of a simple agent; (b) shows the configuration space that would be generated for the centre of that agent; (c) shows that expanding non-axis-aligned brushes may require bevel planes (shown in green) in order to correctly determine the range of movement.



(a) A brush-based environment (b) Configuration space (c) A bevel plane

Figure 3

In conceptual terms, the process of brush unioning is relatively simple: given an input set of brushes, each of which consists of a set of (outward-facing) polygonal faces, it suffices to clip each brush face to all the other brushes within range of its own brush in the environment. From an implementation perspective, a convenient way to do this is to build a binary space partitioning (BSP) tree for each brush and clip each face to the trees of the other brushes. The high-level implementation of this process can be found in Listing 1 and full source code is available online [hesperus]. The end result is a set of polygons that represent the expanded environment.

Finding walkable polygons

Given the set of polygons generated in the previous section, finding those polygons that are *walkable* is straightforward: it suffices to compare the angle between each polygon's normal, $\hat{\mathbf{n}}$, and the up vector ($\hat{\mathbf{u}} = (0,0,1)^T$) to some predefined threshold. The angle can be computed using the dot product:

```
function union_all
: (brushes: Vector<Brush>) -> List<Polygon>

var result: List<Polygon>;

// Build a tree for each brush.
var trees: Vector<BSPTree> := map(build_tree,
                                  brushes);

// Determine which brushes can interact.
var brushesInteract: Vector<Vector<bool>>;
for each bi, bj in brushes
  if j = i then
    brushesInteract(i, j) := false;
  else
    brushesInteract(i, j) := in_range(bi, bj);

// Clip each polygon to the tree of each brush
// within range of its own brush.
for each bi in brushes
  for each f in faces(bi)
    var fs: List<Polygon> := [f];
    for each bj in brushes
      if brushesInteract(i, j) then
        fs := clip_polygons(fs, trees(j), i < j);
    result.splice(result.end(), fs);

return result;
```

Listing 1

$$\theta = \cos^{-1}(\hat{\mathbf{n}} \cdot \hat{\mathbf{u}})$$

We then keep precisely those polygons whose angle is less than or equal to the threshold. In the *hesperus* engine, a suitable threshold for human characters was found to be $\pi/4$ (i.e. 45 degrees to the horizontal).

Walk and step links

To generate simple links between walkable polygons in a navigation mesh, the general strategy is as follows:

- We first create an *edge plane table* that maps each vertical plane through one or more walkable polygon edges to two sets of edges that lie in the plane (edges in one set are oriented in the same direction as a 'canonical' plane; edges in the other have the opposite orientation).
- For each plane in the table and for each ordered pair of opposing edges for that plane, we check to see whether any links need to be created. This is done by transforming the opposing edges into a 2D coordinate system in the plane and calculating the intervals (if any) in which the various types of link need to be created.

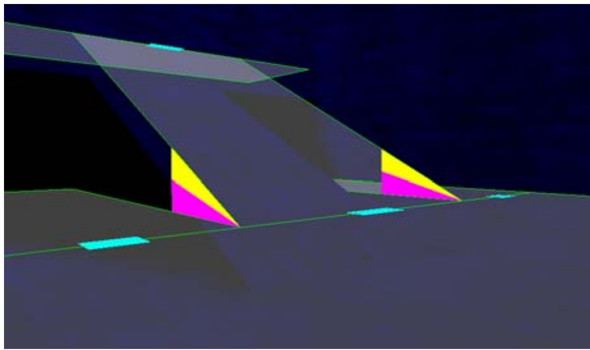
Edge plane table construction

The edge plane table is a map of type Plane to ({Edge},{Edge}). To construct it, we proceed as shown in Listing 2. For each edge of a walkable

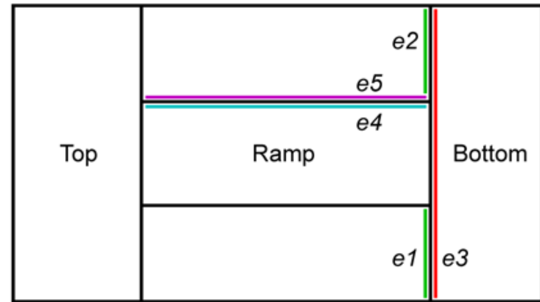
```
function build_edge_plane_table
:(walkablePolygons: List<Polygon>) ->
  Map<Plane,EdgeRefsPair,PlaneOrdering>

var ept: Map<Plane,EdgeRefsPair,PlaneOrdering>;

for each poly in walkablePolygons
  for each p1 in vertices(poly)
    var p2: Vec3 := next_vertex(poly,p1);
    var plane: Plane :=
      make_vertical_plane(p1,p2);
    var canon: Plane := plane.make_canonical();
    var sameFacing: bool :=
      plane.normal().dot(canon.normal()) > 0;
    if sameFacing then
     ept(canon).sameFacing.add(EdgeRef(poly,p1));
    else
      ept(canon).oppFacing.add(EdgeRef(poly,p1));

return ept;
```

Listing 2

An illustration of (part of) the edge plane table for the hesperus test level called Ramp: the ordered pairs of opposing edges are (e1,e3), (e2,e3), (e3,e1), (e3,e2), (e4,e5) and (e5,e4). The first four pairs will cause walk links to be created; the last two pairs will cause step links to be created.



(a) The navigation mesh for Ramp



(b) A top-down view, with some edges highlighted

Plane	Same-Facing Edges	Opposite-Facing Edges
1	{e1, e2}	{e3}
2	{e4}	{e5}

(c) The part of the edge plane table corresponding to the highlighted edges

Figure 4

polygon, we first build the vertical plane passing through it and then add it to the table based on its facing with regard to the 'canonical' vertical plane through the edge. This has the effect of separating the edges of walkable polygons into two sets on each vertical plane. These can then be checked against each other in a pairwise manner to create navigation links – see Figure 4 for an example. A few details are needed to make this work:

- *Vertical Plane Construction*. Each edge is necessarily non-vertical (because it belongs to a walkable polygon), so the normal vector of a vertical plane through it can be calculated straightforwardly using the cross-product. If the endpoints of the edge are $\mathbf{p_1}$ and $\mathbf{p_2}$, and $\hat{\mathbf{u}}$ is once again the up vector, then the desired normal can be calculated as:

  $$\hat{\mathbf{n}} = (\mathbf{p_2} - \mathbf{p_1}) \times \hat{\mathbf{u}}$$

  Normalising this to give $\hat{\mathbf{n}} = \mathbf{n} / |\mathbf{n}|$, the equation of the desired plane is then:

  $$\hat{\mathbf{n}} \cdot \mathbf{x} = \hat{\mathbf{n}} \cdot \mathbf{p_1}$$

- *Canonical Plane Determination*. Given a plane with equation $ax + by + cz - d = 0$, we define the 'canonical' form of this plane to be the one where the first of *a*, *b* or *c* to be non-zero is positive. Thus, the canonical form of both $0x + 1y - 1z - 23 = 0$ and $0x - 1y + 1z + 23 = 0$ would be $0x + 1y - 1z - 23 = 0$. Note that a plane is either already in canonical form, or can be canonicalised straightforwardly by negating all of its coefficients.
- *Plane Ordering*. In order to use planes as the key for the edge plane table, we need to define a suitable way of ordering them. This can be done using a variant of the approach described in [Salesin92]. In practice, the ordering was found to be easier to debug (although somewhat less efficient) if implemented as shown in Listing 3.

Link creation

To generate walk and step links, we transform each ordered pair[4] of opposing edges that lie in the same (canonical) plane into a 2D (orthonormal) coordinate system in the plane, and then determine the intervals (if any) in which links need to be created.

A suitable 2D coordinate system for a plane $\hat{\mathbf{n}} \cdot \mathbf{x} - d = 0$ can be generated as follows. To determine a suitable origin $\mathbf{o}'$ for our coordinate system, we find the point on the plane that lies nearest to the world origin $\mathbf{0}$: this is simply $d\hat{\mathbf{n}}$. Given that the plane is vertical, suitable axis vectors for our coordinate system can be defined as:

$$\hat{\mathbf{i}}' = \frac{\hat{\mathbf{n}} \times \hat{\mathbf{u}}}{|\hat{\mathbf{n}} \times \hat{\mathbf{u}}|}$$

$$\hat{\mathbf{j}}' = \hat{\mathbf{u}}$$

```
function less: (lhs: Plane; rhs: Plane) -> bool

var nL, nR: Vec3 := lhs.normal(), rhs.normal();
var dL, dR: double := lhs.dist(), rhs.dist();

// If these planes are nearly the same (in terms
// of normal direction and distance value), then
// neither plane is "less" than the other.
var dotProd: double := nL.dot(nR);

// acos(x) is only defined for x <= 1, so clamp
// dotProd to avoid floating-point problems.
if dotProd > 1.0 then dotProd := 1.0;

var angle: double := acos(dotProd);
var dist: double := dL - dR;
if |angle| < ε_angle and |dist| < ε_dist then
  return false;

var aL, bL, cL: double := nL.x, nL.y, nL.z;
var aR, bR, cR: double := nR.x, nR.y, nR.z;

// Otherwise, compare the two planes
// "lexicographically".
return (aL<aR) or
       (aL=aR and bL<bR) or
       (aL=aR and bL=bR and cL<cR) or
       (aL=aR and bL=bR and cL=cR and dL<dR);
```

Listing 3

4. Note that because the pairs are ordered, we consider each unordered pair of edges twice when creating links, once in each direction.

Creating links between edges: in (a), the gradients differ and a step down link is created from e to f in the region labelled sdl, and a step up link is created in the region labelled sul; in (b), the gradients are the same and a step up link is created from e to f in the region labelled xOverlap; in (c), a step down link is created from e to f in the region labelled xOverlap. The remaining case, of parallel edges leading to a walk link, is not shown.

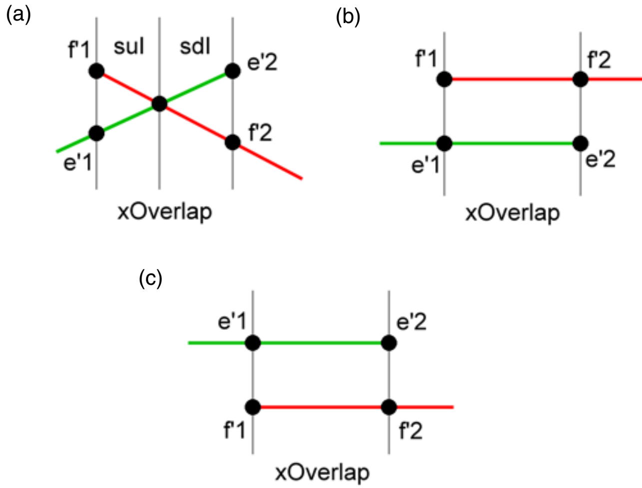


Figure 5

This gives us a coordinate system in which $\hat{\mathbf{i}}'$ is horizontal (in terms of the surrounding world) and $\hat{\mathbf{j}}'$ is vertical. To transform an edge e on the plane into this new coordinate system, we transform each of its endpoints $\mathbf{e_1}$ and $\mathbf{e_2}$ as follows:

$$\mathbf{e_n} \mapsto ((\mathbf{e_n} - \mathbf{o}') \cdot \hat{\mathbf{i}}', (\mathbf{e_n} - \mathbf{o}') \cdot \hat{\mathbf{j}}') = \mathbf{e}'_{\mathbf{n}}$$

Having transformed a pair of opposing edges e and f into the plane's coordinate system, we next calculate the horizontal interval in which each edge lies; e.g. for e this would be:

$$\left[\min\{e'_{1x}, e'_{2x}\}, \max\{e'_{1x}, e'_{2x}\}\right]$$

If the horizontal intervals for the opposing edges do not overlap, then there can be no links between them. Otherwise, we compute the gradients m and cut points c of the lines $y = mx + c$ through the two edges (still in the plane's coordinate system) using basic mathematics; e.g. for e these would be:

$$m_e = \frac{e'_{2y} - e'_{1y}}{e'_{2x} - e'_{1x}}$$

$$c_e = e'_{1y} - m_e * e'_{1x}$$

Based on a comparison of the gradients, we then create the link segments in the plane as shown in Listing 4 and Figure 5. The endpoints l_1 and l_2 of each link segment can be straightforwardly transformed back into world space to create the actual links as follows:

$$l_{\mathbf{n}} \mapsto \mathbf{o}' + l_{nx} * \hat{\mathbf{i}}' + l_{ny} * \hat{\mathbf{j}}'$$

```
function calculate_link_segments
:   (e'1: Vec2; e'2: Vec2; f'1: Vec2; f'2: Vec2;
     xOverlap: Interval) -> LinkSegments

xO = xOverlap;
var result: LinkSegments;
var m_e: double := (e'_2y - e'_1y) / (e'_2x - e'_1x);
var m_f: double := (f'_2y - f'_1y) / (f'_2x - f'_1x);
var c_e: double := e'_1y - m_e * e'_1x;
var c_f: double := f'_1y - m_f * f'_1x;
var Δm, Δc: double := m_f - m_e, c_f - c_e;
if |Δm| > ε then
  // If the gradients of the source and
  // destination edges are different, then we get
  // a combination of step up/step down links.
  // We want to find:
  // (a) The point walkX where y_f = y_e
  // (b) The point stepUpX where y_f - y_e = STEPTOL
  // (c) The point stepDownX where
  //      y_e - y_f = STEPTOL
  var walkX: double := -Δc / Δm;
  var stepUpX: double := (STEPTOL - Δc) / Δm;
  var stepDownX: double := (-STEPTOL - Δc) / Δm;

 // Construct the step down and step up intervals
 // and clip them to the known x overlap interval.
  var sdI, suI: Interval :=
    [min{walkX,stepDownX}, max{walkX,stepDownX}],
    [min{walkX,stepUpX}, max{walkX,stepUpX}];
  sdI  := intersect(sdI, xO);
  suI  := intersect(suI, xO);

  // Construct the link segments.
  if not sdI.empty then
    result.downToF :=
      [(sdI.low, m_e * sdI.low + c_e),
       (sdI.high, m_e * sdI.high + c_e)];
    result.upToE :=
      [(sdI.low, m_f * sdI.low + c_f),
       (sdI.high, m_f * sdI.high + c_f)];
  if not suI.empty then <analogously>
else if |Δc| < STEPTOL then
  // If the gradients of the source and destination
  // edges are the same (i.e. parallel edges), then
  // we either get a step up/step down combination,
  // or a walk link in either direction.
  var s1: Vec2 := (xO.low, m_e * xO.low + c_e);
  var s2: Vec2 := (xO.high, m_e * xO.high + c_e);
  var d1: Vec2 := (xO.low, m_f * xO.low + c_f);
  var d2: Vec2 := (xO.high, m_f * xO.high + c_f);

  if Δc > ε then
    // The destination is higher than the source.
    result.upToF := [s1,s2];
    result.downToE := [d1,d2];
  else if Δc < -ε then
    // The destination is lower than the source.
    result.downToF := [s1,s2];
    result.upToE := [d1,d2];
  else
    // The destination and source are level.
    result.walk := [s1,s2];
```

Listing 4

Potential extensions

At present, only walk and step links are implemented in *hesperus*, but there are various additional links that it would be helpful to add.

Crouch links

One obvious extension is to add *crouch* links – these are links that tell agents when they need to crouch in order to traverse low areas (e.g. a low archway or a pipe). As the example in Figure 6 illustrates, these are inter-mesh links; they should be created so as to link the standing and crouching meshes for an agent at the boundary of areas that can be traversed on the crouching mesh but not on the standing one. Assuming that the AABBs for the two meshes differ only in their heights (as is the case in the example), one way of automatically detecting crouch links would be to match edges on the standing mesh that do not cause any walk or step links to be created with edges in the same plane on the crouching mesh that do.

Ladder links

The addition of ladder links (indicating that an agent can travel from one floor to another using a ladder) is not conceptually very difficult, but it requires tool support. In *hesperus*, the map editor would need to be augmented to handle ladders and other static entities; when placing a ladder, it would then be a simple matter to create a link at either end of the ladder to represent travel in each direction. It should be noted that *traversing* ladder links is significantly more complicated than traversing walk or step links, because it takes time to climb or descend a ladder and someone may be coming the other way, but traversal is beyond the scope of this article.

Jump links

A third type of extremely useful link would be jump links – these are used to indicate places at which an agent can jump to reach another part of the navigation mesh. Calculating jump links can be somewhat costly because it involves simulating the agent making jumps to determine whether or not they are possible. In our case, the situation is made slightly easier because we are working in configuration space and can avoid worrying about clearance, but general-purpose jump links are still non-trivial to generate automatically. One easy type of jump link that could be generated immediately would be vertical jumps – these can be generated in the same way as step up links, but using a larger height threshold.

Conclusions

In this article, I have illustrated how to generate navigation meshes at an implementation level using an approach based on the work of van Waveren [VanWaveren01]. Whilst there are many alternative techniques for navigation mesh construction, as surveyed in the introduction, this configuration space approach is useful because it allows us to avoid the difficulties regarding clearance height that have to be dealt with by other approaches; it also means that each agent occupies a single point on the mesh, completely avoiding the problems caused by an agent straddling multiple mesh polygons.

Navigation mesh *generation*, however, is only part of the picture – in a future article, I hope to write more about using navigation meshes for localisation, movement and path planning. ■

Acknowledgements

As always, I would like to thank the Overload team for reviewing this article and suggesting ways in which to improve it.

Creating crouch links between navigation meshes can allow tall characters to pass through low areas. Here, crouch links should be created between the standing (green) and crouching (red) meshes to allow agents to traverse this low archway.

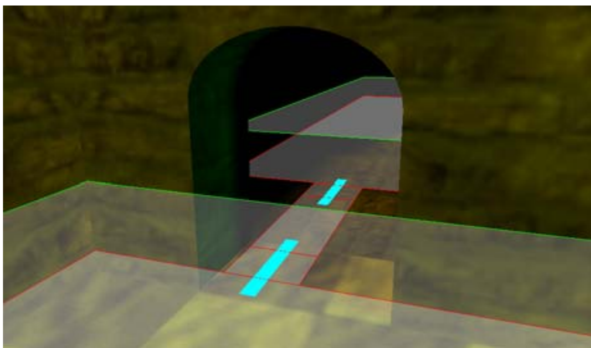


Figure 6

References

[Beucher90] *Segmentation d'Images et Morphologie Mathématique* (Image Segmentation and Mathematical Morphology). Serge Beucher. PhD thesis, E.N.S. des Mines de Paris, 1990.

[Gonzalez02] *Digital Image Processing*. Rafael C Gonzalez and Richard E Woods. Pearson Education, 2nd edition, 2002.

[Hale09] Full 3D 'Spatial Decomposition for the Generation of Navigation Meshes' D Hunter Hale and G Michael Youngblood. In *Proceedings of the Fifth Artificial Intelligence for Interactive Digital Entertainment Conference*, pages 142–147, 2009.

[Hamm08] 'Navigation Mesh Generation: An Empirical Approach' David Hamm. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 113–123. Charles River Media, 2008.

[Hertel83] 'Fast triangulation of simple polygons' S Hertel and K Mehlhorn. In *Proceedings of the 4th International Conference on the Foundations of Computation Theory*, volume 158 of Lecture Notes in Computer Science, pages 207–218. Springer Verlag Berlin, 1983.

[hesperus] The hesperus 3D game engine. Stuart Golodetz. Source code available online at: https://github.com/sgolodetz/hesperus2.

[Kallmann10] 'Navigation Queries from Triangular Meshes' Marcelo Kallmann. In *Proceedings of the Third International Conference on Motion in Games (MIG)*, pages 230–241. Springer-Verlag Berlin, 2010.

[Mononen09] *Navigation Mesh Generation via Voxelization and Watershed Partitioning*. Mikko Mononen. AiGameDev.com, March 2009. Slides available online (as of 30th July 2013) at https://sites.google.com/site/recastnavigation/MikkoMononen_RecastSlides.pdf.

[ORourke94] *Computational Geometry in C*, pages 60–61. Joseph O'Rourke. Cambridge University Press, 2nd edition, 1994.

[Pettré05] 'A navigation graph for real-time crowd animation on multilayered and uneven terrain' Julien Pettré, Jean-Paul Laumond and Daniel Thalmann. In *Proceedings of the 1st International Workshop on Crowd Simulation*, Lausanne, Switzerland, 2005.

[Ratcliff08] 'Automatic Path Node Generation for Arbitrary 3D Environments' John W Ratcliff. In Steve Rabin, editor, *AI Game Programming Wisdom 4,* pages 159–172. Charles River Media, 2008.

[Salesin92] Grouping Nearly Coplanar Polygons into Coplanar Sets. David Salesin and Filippo Tampieri. In David Kirk, editor, *Graphics Gems III*, pages 225–230. Morgan Kaufmann, 1992.

[Snook00] 'Simplified 3D Movement and Pathfinding Using Navigation Meshes' Greg Snook. In Mark DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[Tozour02] 'Building a Near-Optimal Navigation Mesh' Paul Tozour. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[Tozour04] 'Search Space Representations' Paul Tozour. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 85–102. Charles River Media, 2004.

[VanToll11] 'Navigation Meshes for Realistic Multi-Layered Environments' Wouter G van Toll, Atlas F Cook IV and Roland Geraerts. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3526–3532, San Francisco, California, USA, 2011.

[VanToll12] 'A navigation mesh for dynamic environments' Wouter G van Toll, Atlas F Cook IV and Roland Geraerts. *Computer Animation and Virtual Worlds*, 23:535–546, 2012.

[VanWaveren01] *The Quake III Arena Bot*. Jean Paul van Waveren. Master's thesis, Delft University of Technology, 2001.

C++ Range and Elevation

C++ provides many features for higher-level programming, but lacks some common ones present in other languages.

Back in 2009, Andrei Alexandrescu gave a presentation at the ACCU Conference about Ranges. The short version is that although you can *represent* a range in C++98 using a pair of iterators, the *usage* is cumbersome for most simple tasks. Even for a simple loop, using iterators can be a trial.

```
for( vector< int >::iterator pos = data.begin();
  pos != data.end(); ++pos )
{
  .... do interesting things with ints
}
```

The C++ Standard Library provides a simple enough algorithm which improves on this a bit, but you need to provide the action to be performed as a functor argument, thus losing the locality of reference for that action.

```
for_each( data.begin(), data.end(), action );
```

And all that's just using the iterators the C++ Standard Library gives you; defining your own iterator is notoriously complex. So, Andrei introduced a much simpler – and more powerful – abstraction: the Range [Alexandrescu09].

There have been a few attempts to implement Andrei's ideas in C++ (he implemented them for D, and they form the basis of the D Standard Library) but for one reason or another, it just hasn't caught on. Part of the reason for that is that in order to take advantage of this new abstraction as Andrei envisioned it, you need a rewrite of the Standard Library that uses Ranges instead of Iterators. For some reason, there seems to be little appetite for this. Some implementations have arisen that are interoperable with C++ Standard Library algorithms [Boost], [Github1], but even they appear to not have had as much traction in the C++ community at large as might have been hoped. Similarly, there have been proposals to the C++ Standards effort [Standards], but still, not much apparent interest in something that is little more than a thin wrapper around the existing C++ container types which are, after all, really just iterator factories.

Part of the reason Andrei himself implemented his idea of Ranges in D, rather than C++, was that C++ at the time didn't provide good enough language support to make it straightforward. In particular, there was a C++ proposal at the time for `auto` variable declarations, which would have been crucial, but had experimental support in only one widely used compiler. Now, C++ officially has `auto`, and it's very widely supported. But not a widely-used – let alone standard – range type.

All of this is highly relevant, of course, but misses an important point: what do these range types actually achieve? What problem are they attempting to solve?

Time for a quick segue into a different world....

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He can be contacted at steve@arventech.com

The cross-pollination conundrum

It's common knowledge among experienced programmers that an intimate understanding of a few different languages (and a possibly less intimate knowledge of many) is a good thing. Techniques from one language can inform and inspire neat solutions to problems in other languages. It's also common knowledge among experienced programmers that idiomatic features of one language are not necessarily transferable to other languages, and that doing so can result in code that is truly incomprehensible to its readers. With both of those things in mind, I want to explore a little modern C# idiom: `IEnumerable`, the C# Iterator.

This interface is what permits the `foreach` loop in C#:

```
foreach( var item in container )
{
    .... do interesting things with items
}
```

C# has had `IEnumerable` from the very beginning, although it's undergone a few revisions over the years.

`IEnumerable` forms the basis of a much higher-level abstraction than merely accessing the contents of containers, however. It underpins all the functionality of LINQ[1], introduced in Visual Studio 2005 with .Net 3.5, and builds on one key feature of .Net 2.0 – the `yield` keyword, which creates an **`IEnumerable` on demand** (actually, it creates an implementation of `IEnumerator<T>`, which is the *real* iterator type). This facility means that iterating over an `IEnumerable` is lazy – access to an element isn't performed until it's asked for. In C#, this is referred to as Deferred Execution.

```
var results =
  container.Where( item => item.Id == expected )
  .Select( filtered => filtered.Count.ToString() )
  .Take( 2 );
```

The reason lazy access is important is that no matter how many elements `container` has, the clauses for `Where` and `Select` will be called a maximum of 2 times. Obviously, this is significant if `container` has 20 million items in it.

So what has all this to do with C++? In the first case, the reason that `IEnumerable` works *as an interface* in C# is that it is implemented by all the standard containers, and is in fact very simple to implement for your own container types. C++ has no such interface, and in fact, there is no actual relationship – inheritance or otherwise – between the standard containers, or their iterator types. In the second case, does this entirely idiomatic C# translate at all into C++? Or does it make for an incomprehensible mess? That is the nature of the cross-pollination conundrum.

The missing lin[qk]?

It should be clear that `IEnumerable` in C# has much more in common with Andrei Alexandrescu's vision of a Range than it does with C++

1. Specifically, LINQ for Objects, which operate on containers rather than DataSources.

IEnumerable in C# has much more in common with Andrei Alexandrescu's vision of a Range than it does with C++ iterators

iterators – or even iterator pairs. C++11 introduces many language features which allow a neat syntax for it, such as lambda and type deduction facilities. Suppose there were a range type in C++; would it on its own enable the implementation of something like `Select` or `Where` from C#? What might that look like?

```
auto result =
    container.where( [&]( const thing & t )
    { return t.name == expected; } )
    .select( []( const thing & t )
    { return to_string( t.count ); } )
          .take( 2 );
```

It's not inconceivable.

But.

What type is container? We could implement a single type that has `where`, `select`, `take` and all the other things needed as member functions. What really makes C#'s `IEnumerable` work is the existence of extension methods. The power of that mechanism really shines through when you need to write your own function that fits in with other LINQ facilities, and C++ has no analogue for that.

A much neater idea would be closer to the Alexandrescu range concept whereby `select` returns a specific kind of range, and `where` returns a different kind of range. The chaining of those operations together as shown above would still require some common base interface, and would still be hard to extend.

Lastly there is also the question of lazy evaluation (or Deferred Execution) and efficiency. It's hard to see how to make lazy evaluation safe in the above scenario, without resorting to passing functions around under the hood and capturing state. As for efficiency, this idea of a single base class is raising the spectre of allocating stuff on the heap (**gasp!**), and too many C++ programmers have got used to the flexibility and efficiency of templates and type deduction of iterators to give it up that easily.

So, there are questions.

Perhaps we should wind back our expectations a little, start with something very, very simple, and see if it can be implemented. Then we can look to see if we can build on small beginnings to do something more elaborate.

So. Where do we start?

Write a test

For the purposes of testing examples, assumptions and results, I'm using Catch [Github2] because it's easy to read, clear to write and not too verbose.

It should be obvious by now that the first step is to define a very simple and lightweight range type, upon which we can somehow 'hang' all of the operations we need. This range should be trivially initialisable from some standard container. I'm going to make a conceptual leap here, because it's clear the range type needs a way to access the 'current' element, and to move forwards one position. With these operations, it's possible to make a simple check that the 'contents' of the range match the original data.

```
TEST_CASE( "Range is constructable from standard
  collection and is iterable",
  "[range][init][iterable][stl]" )
{
  std::vector< int > data { 1, 2, 3 };
  auto range = make_range( data );
  REQUIRE( *range++ == data[ 0 ] );
  REQUIRE( *range++ == data[ 1 ] );
  REQUIRE( *range++ == data[ 2 ] );
}
```

Listing 1

Andrei Alexandrescu asserted that the pointer-like interface for iterators is a Bad Thing™, but I think it makes for a neat syntax, so I will stick with it. (See Listing 1.)

What is required to make this test compile? The most obvious thing is `make_range` – is that a function or a class? In order to make it as general as possible, it should (obviously) be a template, and to take full advantage of type-deduction it should be a function returning....what? Some type which exhibits the right interface. With just the information in the test, we can sketch it out. Note that, with the use of `auto`, the actual type is never named. This isn't a crucial observation, but does give us a lot of leeway on the choice of name. I have been unimaginitive, however. (See Listing 2.)

There is nothing particularly startling about this. The `iterable_range` class just squirrels away a pair of iterators (the observant will already have noticed that `end` isn't used, but its purpose should be obvious!), and the `make_range` function is really just a convenience for creating an instance of the class. The `iterable_range` is a kind of 'proto-range'; it's not terribly useful on its own, but it does form the basis for other things.

True to type

It's already time to do something a bit harder. Listing 3 is another test.

I already mentioned the idea of having different range types, e.g. `select` provides a 'selecting range', and `where` provides a 'filtering range'. The line `auto result = select( ...` now needs some function `select`, and something to return – a transforming range type. It's going to need the same basic operations as the `iterable_range`, and it needs to operate on an underlying range. The operations on the new range defer to that underlying range type – which in this case is an `iterable_range`.

```
template< typename range_type,
  typename transformation >class transforming_range
{
  public:
    transforming_range( range_type r,
      transformation fn ) : r{ r }, fn{ fn } { }
  private:
    range_type r;
    transformation fn;
};
```

C++11 provides an army of tools for determining the types of things at compile time

```
template< typename iterator_type >
class iterable_range
{
  public:
    iterable_range( iterator_type begin,
                    iterator_type end )
      : pos{ begin }, end_pos{ end } { }
    auto operator*() const ->
      typename iterator_type::value_type
    {
      return *pos;
    }
    auto operator++( int ) -> iterable_range
    {
      iterable_range tmp{ *this };
      ++pos;
      return tmp;
    }
  private:
    iterator_type pos, end_pos;
};
template< typename container_type >
auto make_range( const container_type & ctr ) ->
   iterable_range
   < typename container_type::const_iterator >
{
  return iterable_range
    < typename container_type::const_iterator >
  { begin( ctr ), end( ctr ) };
}
```

Listing 2

```
TEST_CASE( "Transformation of elements results in
   new range leaving originals intact",
   "[range][transform]" )
{
  std::vector< int > data { 1, 2, 3 };
  auto range = make_range( data );
  auto result = select( range, []( int i ) {
    return std::to_string( i ); } );
  std::string expected[] = { "1", "2", "3" };
  REQUIRE( *result++ == expected[ 0 ] );
  REQUIRE( *result++ == expected[ 1 ] );
  REQUIRE( *result++ == expected[ 2 ] );
  REQUIRE( *range++ == data[ 0 ] );
  REQUIRE( *range++ == data[ 1 ] );
  REQUIRE( *range++ == data[ 2 ] );
}
```

Listing 3

The incrementing operator should be straightforward enough – it needs to increment the underlying range object and return a copy of its previous self.

```
auto operator++( int ) -> transforming_range
{
  transforming_range tmp{ *this };
  r++;
  return tmp;
}
```

What about the dereference operator? It needs to call the transformation function `fn` with the current element, and return its result.

```
auto operator*() const -> ???
{
  return fn( *r );
}
```

C++11 provides an army of tools for determining the types of things at compile time for situations like this. The one with the most visibility, `decltype`, provides the declared type of an expression – including the result of calling a function.

```
auto fn( int ) -> bool { return true; }
auto fn( double ) -> int { return 10; }
auto type = decltype( fn( 10 ) );
// type is bool - type returned if fn were
// called with an int
```

This makes determining the result of `operator*()` very simple. The only restriction on this use is that both `fn` and `r` need to have already been ‘seen’, which leads to the need to declare them before they are used in the `decltype` expression.

```
private:
  range_type r;
  transformation fn;

public:
  auto operator*() const -> decltype( fn( *r ) )
  {
    return fn( *r );
  }
```

I normally much prefer to declare classes with the `public` section at the top, where it’s most obvious and visible. However, it seems a fair trade in this case to allow the use of `decltype` in such a simple fashion. The alternative is **much** worse![1]

With this addition, the `transforming_range` class should pass all the tests, after we add the now-trivial `select` function. (See Listing 4.)

1. Alright, you’d need something like `auto operator*() const -> typename std::result_of< transformation( decltype( *std::declval< range_type >() ) ) >::type` I hope you agree that declaring `private` at the top is a small price to pay!

We need some sort of **check that the range object is valid** – that it hasn't run out of elements

```
template< typename range_type,
          typename transformation >
auto select( range_type r, transformation fn ) ->
transforming_range< range_type, transformation >
{
  return transforming_range< range_type,
         transformation >{ r, fn };
}
```

Listing 4

```
TEST_CASE( "Filtering elements contains just the
matches", "[range][filter]" )
{
  std::vector< int > data { 1, 2, 3 };
  auto range = make_range( data );

  auto result = where( range, []( int i ) {
    return i % 2 != 0; } );

  REQUIRE( *result++ == data[ 0 ] );
  REQUIRE( *result++ == data[ 2 ] );
}
```

Listing 5

Just the good ones

Time for another test (Listing 5).

With all the funky stuff learned implementing the **transforming_range**, implementing a **filtering_range** to be returned by a **where** function should be pretty straightforward (Listing 6).

The cleverness is all in **operator++()**, which keeps incrementing until the predicate is false. The **where** function is simplicity itself – almost identical to **select**, just returning a different range type.

```
template< typename range_type, typename filter >
auto where( range_type r, filter fn ) ->
  filtering_range< range_type, filter >
{
  return filtering_range< range_type, filter >{
    r, fn };
}
```

Run the tests....and they all pass. Time for the next one....

Wait a moment!

Once again, the observant among you will have already spotted the completely fatal flaws in that code. Just to labour the point, Listing 7 is a test that exposes one of them.

The problem here is that the filtering range is fine if the first element in the range matches the predicate. In any other case, it fails the test.

```
template< typename range_type,
          typename unary_predicate >
class filtering_range
{
  private:
    range_type r;
    unary_predicate fn;
  public:
    filtering_range( range_type r,
       unary_predicate fn ) : r{ r },
       fn{ fn } { }
    auto operator*() const -> decltype( *r )
    {
      return *r;
    }
    auto operator++( int ) -> filtering_range
    {
      filtering_range tmp{ *this };
      r++;
      while( !fn( *r ) ) r++;
      return tmp;
    }
};
```

Listing 6

Hopefully, this starts a chain of thought leading to questions like 'what if no elements match?' and then 'what if the initial range is empty?'. **Now** we know why the **iterable_range** has an **end** iterator! We need some sort of check that the range object is valid – that it hasn't run out of elements. Fortunately, it is trivial to implement a safe conversion to **bool** on all three of our range types. For **iterable_range**, it returns **false** if **pos==end_pos** and **true** otherwise. For **transforming_range** and **filtering_range**, it simply returns the value for the underlying range. C++11 provides an **explicit** conversion operator – which means the new function won't take part in any arithmetic or otherwise unsafe operations.

```
TEST_CASE( "Filtering range returns correct
matches when first element is a mismatch",
"[range][filter][empty]" )
  {
    std::vector< int > data { 2, 3, 4 };
    auto range = make_range( data );

    auto result = where( range, []( int i ) {
return i % 2 != 0; } );

    REQUIRE( *result++ == data[ 1 ] );
  }
```

Listing 7

we could add a common base class declaring all the high-level functionality like select, where and so on...the trouble with this approach is that it's inflexible

```
explicit iterable_range::operator bool() const
  { return pos != end_pos; }
explicit transforming_range::operator bool()
  const { return !!r; }
```

The slightly odd `!!r` says what it does: 'not not', invoking the underlying range's `bool` conversion. Because the conversion is `explicit`, just `return r;` won't work, of course![1]

With the addition of this rather important facility, we can add a helper function to `filtering_range` called `find_next_match`.

```
void find_next_match()
{
  while( r && !fn( *r ) )
    ++r;
}
```

This function is invoked by `operator*()` (thus making `filtering_range` lazy-evaluated), so finds the first matching element. The increment operator also needs to invoke it to ensure a range with no further matches becomes invalidated.

This implies that client code must first check that the range is valid before invoking `operator*()`[2], and so `operator bool()` must *also* invoke it in order to detect a range with no matching elements.

Time for some more tests! (Listing 8)

Whilst we're at it, we'll add a prefix `operator++()` to all the range types, too (did I manage to slip that one past you in the implementation of `find_next_match`?), since efficiency is one of our design principles.

For `filtering_range`, that operator is used by the postfix version, and looks like the following:

```
auto operator++() -> filtering_range &
{
  ++r;
  find_next_match();
  return *this;
}
```

Now we're really cooking. It must be time to turn the world upside down yet again.

Joined up

It should be fairly clear how to implement `take` using the techniques already discussed here. What's missing is the ability to chain expressions together. In fact, our existing API allows a limited form of composing expressions.

```
auto result = select( where( []( int i ) {
  return i % 2 != 0; } ),
[]( int i ) { return i * 2; } );
```

```
TEST_CASE( "Filtering range is immediately
  invalid when no elements match",
  "[range][filter][nomatch]" )
{
  std::vector< int > data { 2, 3, 4 };
  auto range = make_range( data );
  auto result = where( range, []( int )
  { return false; } );

  REQUIRE( ! result );
}

TEST_CASE( "Filtering range returns correct
   matches when first element is a mismatch",
   "[range][filter][empty]" )
{
  std::vector< int > data { 2, 3, 4 };
  auto range = make_range( data );
  auto result = where( range, []( int i )
  { return i % 2 != 0; } );

  REQUIRE( !!result );
  REQUIRE( *result++ == data[ 1 ] );
}
```

Listing 8

This is somewhat unwieldy, however, especially when composing more than a small number of expressions.

As already noted, we could add a common base class declaring all the high-level functionality like `select`, `where` and so on. The trouble with this approach is that it's inflexible; it would be difficult to extend with new operations.

It's not possible to overload `operator.()` in C++, so directly mimicking the syntax of extension methods is out, but there are other operators we could use. There is something appealing about hijacking the 'pipe' operation common in filesystem operations. Time for another test. (Listing 9.)

The main thing to note here is that the signatures of the range functions `select` and `where` have changed – they no longer take a range object – almost as if the range is being passed in through some 'standard input'. This suggests some global operator, perhaps like this:

```
template< typename left_range_type,
          typename right_range_type >
auto operator|( left_range_type left,
                right_range_type right ) -> ???
{
  ???
}
```

1. More verbose but perhaps more descriptive might be `return static_cast< bool >( r );`
2. The 'functional' approach to this problem is `any`, which returns `false` if a range contains no elements.

One of the motivations for the ease of chaining operations was to make the API easy to extend

```
TEST_CASE( "Range results can be composed using
   simple syntax", "[range][composition]" )
{
  std::vector< int > data { 1, 2, 3 };
  auto range = make_range( data );

  auto result = range
  | where( []( int i )
    { return i % 2 != 0; } )
  | select( []( int i )
    { return i * 10; } );

  REQUIRE( *result++ == data[ 0 ] * 10 );
  REQUIRE( *result++ == data[ 2 ] * 10 );
  REQUIRE( ! result );
}
```

Listing 9

The question is – what should it return? Perhaps it would be simpler to see the solution to this using actual named funtions instead of overloaded operators.

```
auto result =
range.apply( where( []( int i )
           { return i % 2 != 0; } ) )
     .apply( select( []( int i )
           { return i * 10; } ) );
```

This is similar enough to actual chaining, there appears to be some merit to following it to see where it leads.

The first obstacle is this redefinition of **`select`** and **`where`**. To take **`where`** as an example, it cannot construct an instance of **`filtering_range`**, because that type requires an underlying range object on which to operate. The range object isn't supplied until **`apply`** is called – whatever that is. It looks from here very much like a member function common to all range types. Given that **`where`** can't return a **`filtering_range`**, but must capture the filtering predicate somehow, another intermediate type is indicated. This intermediate is what gets passed to **`apply`**, and is a factory for the real range type. The **`apply`** function then invokes that factory to create a real range type.

One approach to this might be to have a separate intermediate factory for every range type; this would certainly make the implementation simple: the **`filtering_range`** would have a **`filtering_range_factory`**, and the implementation of that would know how to construct a **`filtering_range`** given an underlying range object to construct it with.

However, for the cases in this example at least, there is a more general solution. Once again we look to Andrei Alexandrescu, and make use of a simple policy template. The **`where`** function 'knows' it requires a **`filtering_range`**, but has insufficient data or template parameters to actually create one. If the **`filtering_range_factory`** makes use of a template-template parameter, it can create a **`filtering_range`** when all the parameters required are available.

```
template< template< typename, typename >
  class range_type, typename expression_type >
  class range_factory
{
  public:
    range_factory( expression_type action ) :
      action{ action } { }
    template< typename range_of >
    auto operator()( range_of r ) const
      -> range_type< range_of, expression_type >
    {
      return range_type< range_of,
        expression_type >{ r, action };
    }
  private:
    expression_type action;
};
```

Listing 10

This can be generalised out to a factory that can create any range type that takes two template parameters. (See Listing 10.)

Now, the **`where`** function instantiates the factory with the required range *type* (**`filtering_range`**), and the predicate function to be captured.

```
template< typename unary_predicate >
auto where( unary_predicate fn )
  -> range_factory< filtering_range,
     unary_predicate >
{
  return range_factory< filtering_range,
         unary_predicate >{ fn };
}
```

The **`select`** function can do the analogous operation using **`transforming_range`**.

With this information, we can now implement the **`apply`** function. Our original vision was to replace **`apply`** with an overload of **`operator|()`**. Whilst **`apply`** needed to be a member function to allow chaining calls together, the overloaded operator does **not** need to be a member. We can sidestep **`apply`** altogether, and jump straight to the **`operator|()`** implementation, to make our original test for this pass. All that's needed is to call the function-call operator on the provided factory with the provided underlying range object. (Listing 11.)

Extensions

One of the motivations for the ease of chaining operations was to make the API easy to extend. Let's see how well we've achieved that, and write **`take`**. The idea is that **`take`** produces a range of up to a given number of elements.

```
template< typename range_type,
   typename range_factory_type >
  auto operator|( range_type range,
    range_factory_type factory ) ->
    decltype( factory( range ) )
{
  return factory( range );
}

TEST_CASE( "Range results can be composed using
   simple syntax", "[range][composition]" )
{
  std::vector< int > data { 1, 2, 3 };
  auto range = make_range( data );

  auto result = range | where( []( int i )
    { return i % 2 != 0; } )
    | select( []( int i ) { return i * 10; } );

  REQUIRE( *result++ == data[ 0 ] * 10 );
  REQUIRE( *result++ == data[ 2 ] * 10 );
  REQUIRE( ! result );
}
```

Listing 11

```
TEST_CASE( "Range can be limited to a number of
   elements with take", "[range][take]" )
{
  std::vector< int > data { 1, 2, 3, 4, 5 };
  auto range = make_range( data );

  auto result = range | take( 2 );

  REQUIRE( *result++ == data[ 0 ] );
  REQUIRE( *result++ == data[ 1 ] );
  REQUIRE( !result );
}
```

Listing 12

If the original range has fewer than the requested number, then all are returned. (See Listing 12.)

Implementing the range used – let's call it `limited_range` – looks straightforward; `operator bool()` just needs to return `false` after a certain number of iterations of the underlying range. The problem with this one is the implementation of the helper function: `take` itself. Up to this point, the `range_factory` has had a simple task of creating any one of a number of different range types that all had in common their template parameters and construction; each one with two template parameters, and a constructor that took a range-type and functor-type.

Let's take a quick look at the basic construction of a `limited_range`:

```
template< typename range_type >
class limited_range
{
  public:
    limited_range( range_type r, int count )
```

Only one template parameter, and the count, which can't be made into a template parameter because it might not be a constant. The `range_factory` now needs to do something different, i.e. be able to create objects having either one or two template parameters.

The existing `range_factory` exists because the type of the underlying range isn't known until the range operation (e.g. `select`) is invoked via the `operator|()` mechanism. Introducing a new range type with only one template parameter means the original 'action expression', which for `take` is just a number, still needs to be captured before the range object is created, but the range type is still not known.

```
template< template< typename, typename... >
   class range_type, typename expression_type >
  struct range_factory { };

  template< template< typename, typename,
     typename... > class range_type,
     typename expression_type >
  struct range_factory< range_factory
    < range_type, expression_type > > { };
```

Listing 13

This means the basic mechanism is the same, but the implementation of `operator()()` needs to vary according to the number of template arguments for the target range type. The ideal solution to this would be to create a partial specialization of `range_factory` using variadic templates to represent the differences. Something like Listing 13.

This would rely on factory types with two or more template parameters matching the specialisation, and those with only one template parameter matching the primary. However, the rules of partial specialization don't allow this; the primary template's implicit types (`range_type` and `expression_type`) aren't distinguishable from the specialization, so the second `struct` is ambiguous.

It doesn't prevent a new factory type which understands how to create types having only one template parameter, and having the `take` function use it directly.

```
template< template< typename > class range_type,
   typename expression_type >
class range_factory_1
{
```

It is otherwise idential to `range_factory`, except for `operator()()`

```
template< typename range_of >
auto operator()( range_of r ) const
   -> range_type< range_of >
{
  return range_type< range_of >{ r, action };
}
```

The only difference here is the number of template arguments provided to the `range_type` in the factory function.

This solution works, but it does require any other extensions to know which ...`factory` class to invoke. It's not a catastrophe, but it could be made easier. Instead of variadic templates and specialization, we turn to ordinary function overloading to come to the rescue. Instead of creating the correct factory type directly, `take` can use a call to a function which is overloaded based on the same template-template upon which we wished we could specialize `range_factory`. (Listing 14.)

```
template< template< typename, typename >
  class range_type, typename expression_type >
auto make_range_factory( expression_type expr )
  -> range_factory< range_type, expression_type >
{
  return range_factory< range_type,
               expression_type >{ expr };
}

template< template< typename > class range_type,
   typename expression_type >
  auto make_range_factory( expression_type expr )
     -> range_factory_1< range_type,
                         expression_type >
{
  return range_factory_1< range_type,
                         expression_type >{ expr };
}
```

Listing 14

```
auto take( size_t count ) ->
  decltype( make_range_factory< limited_range >
  ( count ) )
{
  return make_range_factory< limited_range >
    ( count );
}

template< typename unary_predicate >
auto where( unary_predicate fn ) ->
  decltype( make_range_factory
    < filtering_range >( fn ) )
{
  return make_range_factory< filtering_range >
  { fn };
}
```

Listing 15

This pair of function overloads selects the correct factory class to construct based on the number of template parameters required by the **range_type** template-template. Any function that now needs a **range_factory** or **range_factory_1** can just use the overloaded function and provide the correct type for that factory to create, and overloading will do the rest (Listing 15).

Here is the **take** function implemented to use the new factory factory function (**!**), and using the same function return type deduction as used previously, with **decltype**. I've also re-implemented **where** to show the usage is identical; these two functions make use of *different* range factory types, but that is merely 'implementation detail'.

It's not unreasonable to imagine range implementation classes needing more template arguments. In such cases, the corresponding **range_factory**_N and **make_range_factory** overload pair would be needed, but in practice, one and two template parameter range types cover most of the most useful things.

All done

This article set out with the stated aim of implementing something not dissimilar to the simple C# LINQ expression which does simple filtering, transformation and range-limiting. With the implementation so far, it's **very** similar (but not exactly the same, for some good reasons). See Listing 16.

There was a mention in passing, however, of being able to interoperate with the existing C++ Standard Library implementations, without losing efficiency. The high-level API achieved here is certainly not just a thin wrapper around C++ iterators; it provides a very rich and type-safe platform which is extended fairly easily (at least, no more difficult than extending C#'s **IEnumerable** facilities). How hard is it to go the extra step, and allow this new range type to play nicely with C++ algorithms?

With some restrictions, it's very simple. Those restrictions again are inspired by C#'s LINQ: **IEnumerable** is an *immutable* interface. You cannot modify elements of it, nor modify the range represented by it, without going via some concrete container type, which in C# means calling **ToList()** or **ToArray()** on the range.

```
var result = container.Where
    ( item => item.Id == expected )
 .Select( filtered => filtered.Count.ToString() )
 .Take( 2 );

auto result = range
  | where ( [&]( const thing & t )
    { return t.name == expected; } )
  | select( []( const thing & t )
    { return to_string( t.count ); } )
  | take( 2 );
```

Listing 16

```
TEST_CASE( "Range can be used to populate a
   standard lib container",
   "[range][export][stl]" )
{
  std::vector< int > data { 1, 2, 3 };
  auto range = make_range( data ) |
               select( []( int x )
  { return std::to_string( x ); } );

  std::vector< std::string > result
  { begin( range ), end( range ) };

  REQUIRE( result.size() == data.size() );
  REQUIRE( std::to_string( data[ 0 ] ) ==
     result[ 0 ] );
  REQUIRE( std::to_string( data[ 1 ] ) ==
     result[ 1 ] );
  REQUIRE( std::to_string( data[ 2 ] ) ==
     result[ 2 ] );
}
```

Listing 17

I believe immutability isn't an unreasonable restriction on this kind of programming. With increased focus on concurrency and multi-processing to achieve better performance, and with both of those techniques benefiting greatly from the use of immutable data, making the ranges that this API operates on immutable isn't a restriction, it's a design principle. Turning an immutable range into mutable data which works with mutating algorithms requires the C++ equivalent of **ToList()**.

Let's see if we can express what is required for that in another test (Listing 17).

This highlights the fact that in C++, containers and algorithms work with *pairs* of iterators – the **[begin, end)** 'range', whereas the range types described in this article encapsulate the pair of iterators into a single item.

Implementing **begin** for the range type looks straightforward – perhaps the necessary operations (only an **InputIterator**'s operations are required) could be added to the range types, so equivalence operators **!=** and **==**.

The possible difficulty might be in implementing **end**.

There is also precedence for writing **end** when an end position isn't known: **std::ostream_iterator** uses a sentry type (effectively an invalid position created by the default constructor), so a similar technique could be employed here. The only way to find the end of a Range is to consume it, which is obviously undesirable.

There's more to a C++ Iterator than **++**, *****, **!=** and **==** however, in practice. There are some embedded typedefs to consider as well, which is usually captured by inheriting from **std::iterator**. So, instead of making changes to all of the range types, or some common base class, it seems to be a better separation of concerns to implement the necessary iterator type separately. As already noted, only the operations and types associated with **InputIterator**s are needed if range types are immutable. Most of the required operations are easy enough to write (see Listing 18).

The sticking point is to decide how to implement **operator==()**. How to compare two ranges to see if their current positions are the same, in the same range of iterators? Time to step back and consider: how would the **std::vector** constructor taking two iterators be implemented? Something very like:

```
vector( iterator b, iterator e )
{
  while( b != e )
    push_back( *b++ );
}
```

The point here is that the most important consideration is the comparison to the 'end' position, which our ranges *already do* to determine **operator bool()**, needed for other things internally. If we make the assumption that

```
template< typename range_type,
  typename value_type >
  class range_output_iterator : public
     std::iterator< std::input_iterator_tag,
                    value_type >
{
  public:
    range_output_iterator( const range_type & r )
      : r{ r }
    {
    }
    auto operator==
       ( const range_output_iterator & )
       const -> bool
    {
      ???
    }
    auto operator!=
       ( const range_output_iterator & r )
       const -> bool
    {
      return !operator==( r );
    }
    auto operator++() -> range_output_iterator &
    {
      ++r;
      return *this;
    }
    auto operator*() const -> value_type
    {
      return *r;
      }
  private:
    range_type r;
};
```

Listing 18

such comparisons are only ever between a valid range and the end of the same range, then `operator==()` can use `operator bool()` – two ranges always compare equal if the first is valid.

A side effect of this is that the right-hand-side of an expression `range_l == range_r` is never used, so it doesn't matter what `end` returns – it doesn't even need to be a default-constructed sentry value (which would be difficult, given the lack of default constructors of the range types).

```
auto operator==
  ( const range_output_iterator & ) const -> bool
{
  return !r;
}
```

The implementations of `begin` and `end` therefore become as shown in Listing 19

That's not a printing error: they are identical in implementation. `end` merely has to return something of the correct *type* – it is never used, not even to compare with anything.

A side effect of this is that interoperability with non-mutating C++ algorithms is achieved for free, for example

```
{std::copy( begin( r ), end( r ),
            std::back_inserter( my_list ) ); }.
```

To conclude

There have been a few range libraries developed for C++ over recent years, but none seem to have had the same take-up as ranges have in D, for example. I think it's partly because C++ iterator pairs have become such a central part of writing C++ programs that use the Standard Library, partly because such range libraries that there are are either clunky to use, or have disappointing behaviour regarding the C++ Standard Library, and partly because there are actually no really compelling use cases for them that can't be achieved using other techniques.

In this article, I set out to demonstrate some uses for a very lightweight range type that's very easy to use, and provides facilities that are very widely used in a different language – C# – which in turn took the ideas of functional languages and coupled them with what some people describe as the ultimate declarative language, SQL.

C++ has many functional facilities, and many of the standard algorithms mirror functional constructs. `std::transform` is essentially a list comprehension, but lacks the brevity and simplicity of transformations in truly functional languages, at least in part because it uses two iterator-pairs to represent separate ranges.

The C# `Select` expression to transform one `IEnumerable` into another captures the simple case that is most common – turn every element into a new range of some new type – and C++'s `transform` doesn't really even compete. A simple range type for C++ overcomes the problems with having that simplicity, whilst still allowing simple and efficient interoperability with existing C++ Standard Library facilities.

All the code in this article compiles with GCC 4.6.3 (with -std=c++0x) on Ubuntu, and GCC 4.8.1 (with -std=c++11) and Microsoft Visual Studio 2013 CP on Windows. It's available from https://github.com/essennell/narl. ■

Acknowledgements

Many thanks to Andy Sawyer and Roger Orr for fixing some of my misunderstandings of `decltype` and trailing return types, Jon Wakely for helping me understand **why** partial specialisation of types based just on the number of template-template parameters doesn't work, and to Frances Buontempo, Roger Orr and Chris Oldwood for reading early revisions of the article and spotting my inevitable errors.

References

[Alexandrescu09] http://accu.org/content/conf2009/AndreiAlexandrescu_iterators-must-go.pdf

[Boost] http://www.boost.org/doc/libs/1_54_0/libs/range/doc/html/range/reference.html

[Github1] https://github.com/dietmarkuehl/kuhllib/wiki/STL-2.0#andrei-alexandrescus-ranges

[Github2] https://github.com/philsquared/Catch

[Standards] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3350.html

```
template< typename range_type >
auto begin( range_type r ) ->
range_output_iterator< range_type,
                       decltype( *r ) >
{
  return range_output_iterator< range_type,
                       decltype( *r ) >{ r };
}
template< typename range_type >
auto end( range_type r ) ->
  range_output_iterator< range_type,
                         decltype( *r ) >
{
  return range_output_iterator< range_type,
                         decltype( *r ) >{ r };
}
```

Listing 19