overload 171

OCTOBER 2022 £4.50

The Power of Ref-qualifiers

Andreas Fertig reminds us why these are so useful even though they are frequently overlooked.

User Stories and BDD – Part 1

Seb Rose explores the origins and evolution of the user story.

The ACCU Conference

Frances Buontempo extols the benefits of attending – or presenting at – the annual conference.

The Model Student

A reprint of an article from Richard Harris, exploring some interesting mathematics for those modelling using computers.

Afterwood

Chris Oldwood explores the categorisation and storage of computer books.

A magazine of ACCU

ISSN: 1354-3172

October 2022

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in *Overload* 172 should be submitted by 1st November 2022 and those for *Overload* 173 by 1st January 2023.

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

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.

Sustainability: An Impossible Dream?

Sustainable development is currently receiving a lot of attention. Frances Buontempo questions what might make this possible.

The UK had a heatwave for four days in August, which happened to coincide with a festival we attended. Lugging kit and sleeping in a tent in the heat was hard work. We did manage to see a few bands, but came back sunburnt and tired. This means I haven't written an editorial, as ever. We might try to hire a campervan next time. That way, we can keep going, catch up with friends and see some bands, but without exhausting ourselves. In order to keep doing what you love, you sometimes need to find new approaches, otherwise everything becomes unsustainable.

Though the UK does have heat waves from time to time, this one seemed unprecedented. Climate change may well be to blame. Regardless of the cause, when things change, a new approach is usually called for, hence our camper van decision. Sometimes nothing has changed, but you start noticing things have gotten out of hand. It's all very well saying "DevOps means eat your own dog-food," but if you get woken up to handle a prod issue every few hours, finding and fixing the root-cause is important. I suspect most programmers will put up with annoyances to a point, and often for far longer than many others would endure, including tedious data manipulation, or sifting through gigantic log files to find errors. Eventually, the tedium causes action. "We aren't going to take this anymore," cry the devs. While putting up with the tedium, you can spot repetitive actions that can possibly be automated. This allows you to change the process in order to keep everything the same, at least when looked at from the outside. Keeping a process running usually involves measuring or tinkering in order to keep things ticking over. From the outside, things may appear smooth, unchanging and reliable, but under the calm surface the metaphorical duck may be paddling frantically. Whether this is sustainable or not depends on how much, and how often, tinkering is needed.

Sustainability does not mean keeping everything the same. The UK is having a fuel crisis, with the prices spiraling out of control. Some people are suggesting fracking in order to generate our own gas, the thinking being we would then have a cheaper supply and could carry on as we are. That may not be the best solution for a variety of reasons. Some people are crying out for new ways of heating homes or better insulation instead. Sometimes, we solve the wrong problems. It's tempting to try to tinker in order to keep things as they are, but sometimes you need a radical change. The same happens with a code base. Sometimes a rewrite is called for. Some style guides include ways to ensure, or at least attempt to ensure, code is maintainable. Adding new parameters to a function, starting with a leading comma springs to mind. It's easy to forget to add a comma on the line before, and this can cause hard to track down compile errors. Adding `, int yet_another_parameter` to the end of the existing inputs ensures this doesn't happen. However, adding more and more parameters might be causing another problem. Large and potentially confusing function signatures add to cognitive load. Though such guidelines attempt to help maintain the codebase, they might allow or even encourage the rot to set in. Treating yourself to a whole new function might be better. Anything to make life easier. Well, maybe not anything. Sometimes the open-closed principle is offered as a way to future proof your code, by making it open for extension but closed for modification. Chris Oldwood did a deep dive into the SOLID principles [Oldwood14]. Perhaps making code easy to change is more important than lots of wiring to avoid future changes to existing code.

'Net zero' and 'sustainable' can be something of a bandwagon companies jump on for marketing purposes, often described as 'greenwashing'. I recently noticed 'news' about a company that had created reusable printer paper. The idea seemed to be that you buy special paper, and possibly ink, along with their printer, which 'sucks' the ink off a page and uses it again, for up to about five times. Perhaps this is a good idea, but it will depend on the total energy and resources needed compared to producing paper every time. Finding new ways to do old things is all very well, but that can miss chances for improvement. Whether electric cars will solve the climate emergency, or whether we need to rethink how much transportation and travel is unnecessary, is worth pondering. We can see the same thing happening when we code. It might feel quicker to shoe-horn a small hack in place to get something working, but sometimes that takes far longer than expected, or worse, seems to work but causes hard to diagnose problems later. Do you ever find yourself looking at a function and thinking, "Oh no, not this again." It could be a badly named variable that confuses you silly, or a very long `switch` statement. That could make it time for a radical rethink and some rather intrusive refactoring. If net zero means some positive steps are combined with some negative steps, leading to no overall steps that do not seem to be much progress. I have committed code before and been relieved to notice I have deleted as many lines as I have added. The huge spaghetti monster is no worse. Net zero. Perhaps I should challenge myself to make more negative code commits. Let's aim for better than net zero. Using the climate emergency analogy, a sum of zero is not a sustainable approach, since it fails to make anything better.

We haven't defined sustainable yet. The word has a sense of keeping going and not dying out, which C++ is managing, despite a period onslaught from other new languages claiming to do so much better. The rival languages do all offer improvements and new perspectives, which C++ sometimes takes onboard. As C++ progresses forwards with each new version, the committee tries to keep backwards compatibility. Bjarne Stroustrup notes:

> C++'s C compatibility was a key language design decision rather than a marketing gimmick. Compatibility has been difficult to achieve and maintain, but real benefits to real programmers resulted, and still result today.

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 since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

He also says, "C++20 is backwards compatible with C++11, that is almost completely backwards compatible with C++98" [Stroustrup21]. The attempt to keep versions compatible includes the application binary interface (ABI), which can make implementing new features surprisingly difficult. Johnathan Wakely talked about this at an ACCU conference, saying "There are lots more things that get done implicitly by the language and so look simple in the code, but…" The rest of the slides explain the 'but'. [Wakely15]. Compatibility is a challenge. Compatibility is not the same as sustainability, of course; however, it is part of the jigsaw. Trying to keep up to date with C++ changes can be a challenge. Hopefully, some of the *Overload* articles help and I usually find various talks at the ACCU conference give excellent overviews of new language features. Check out the YouTube channel if you've never been to the conference [ACCUConf]. Sustainability means something can continue, whether this means planting new trees for each that gets chopped down, or not using up all your resources, so your program crashes. Sustainability is about avoiding irreparable damage, but measuring anything's total impact can be difficult, and drawing a line beyond which something becomes irreparable is hard.

Talking about a code base or manufacturing process as sustainable is one thing, but until AI takes over, programs will be created by people, and people get broken too. If you do the same thing for a very long time, you might suffer from burnout. The World Health Organisation (WHO) recognized burnout as an occupational phenomenon, rather than a medical condition, in 2019 [WHO19]. The WHO give this definition:

> Burn-out is a syndrome conceptualized as resulting from chronic workplace stress that has not been successfully managed. It is characterized by three dimensions:
>
> - feelings of energy depletion or exhaustion;
> - increased mental distance from one's job, or feelings of negativism or cynicism related to one's job;
> - reduced professional efficacy.
>
> Burn-out refers specifically to phenomena in the occupational context and should not be applied to describe experiences in other areas of life.

After my PhD, I couldn't face reading any more. I managed to nurse myself round by reading *The Sandman* comics by Neil Gaiman [Gaiman]. If you find you can no longer face doing something you used to love, take a step back. You may not be suffering from burnout, but some 'me time' and self-care once in a while is good for you.

If doing the same thing over and over without a break might cause burn-out, doing something familiar can bring joy. I frequently get tasked with writing unit tests for other people after they have written code. I have Opinions, with a capital O, about writing tests after you've written code. Test first, people. However, I am happy to add tests to a code base. It can be a good way of finding out how the code works, and it's impossible to break something vital in production. You might even find out why something goes wrong in prod. I feel comfortable adding tests, because I know how to do this. Some people are surprised that I am willing to do something they think of as tedious, but using a skill you have built up to do something productive is joyful. Bertrand Russell explores happiness in *The Conquest of Happiness* [Schmitz16]. He says:

> Two chief elements make work interesting: first, the exercise of skill, and second, construction. … All skilled work can be pleasurable, provided the skill required is either variable or capable of indefinite improvement. Even more important as a source of happiness is the element of constructiveness.

A sense of purpose and constructiveness might just avoid burnout. He also says:

> The most satisfactory purposes are those that lead on indefinitely from one success to another without ever coming to a dead end.

Constant dead ends make tasks unsustainable. Trying to write an editorial, or article, or book may seem beset with dead ends and wrong turns; however, sometimes you find a direction to head in, and the words almost write themselves. Something similar can happen with code; you might find the right data structure and then you're on a roll. Or a small refactor that opens up infinite new possibilities. In order to find that joy and momentum, you need some agency in the process. Allow me one more quote from Bertrand Russell:

> One of the causes of unhappiness among intellectuals in the present day is that they find no opportunity for the independent exercise of their talents, but have to hire themselves out to rich corporations directed by Philistines, who insist upon their producing what they themselves regard as pernicious nonsense.

We mentioned SOLID earlier. We have names for patterns we use when we code. We also have names for things that go wrong. Off by one errors, undefined behaviour, the list goes on. This happens outside of the programming world too. I watched a television programme a while ago called *Why buildings collapse*. Clearly something of a clickbait title; however, it was very interesting. One phrase stuck in my mind: 'punching shear failure'. [RISA] This phrase could well be applied to some code I have written. Having a name for your pain helps you discuss it and think it through.

In order to be sustainable, we need to step back and check everything is going ok from time to time. A recent podcast from Stackoverflow [May22] briefly discussed why some organisations are moving AI back from the cloud to on-prem hardware. For a while, it seemed as though everything was going to end up in the cloud; however, people are often seeing huge bills, having got some settings wrong. They are also finding it generally cheaper, even with the right settings, and more performant, than sharing resources in the cloud. Some decisions need revisiting. Have you ever marked a unit test with an ignore attribute, just temporarily, while you fix something? And then forgot to remove the attribute. This may not be the worst form of greenwashing, but let's avoid 'pernicious nonsense' and see if we can make our coding sustainable.

References

[ACCUConf] ACCU Conference YouTube channel. https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw

[Gaiman] Neil Gaiman, *The Sandman*, published by DC Comics. See https://sandman.fandom.com/wiki/The_Sandman

[May22] Eira May 'Why AI is having an on-prem moment', published 23 August 2022 at https://stackoverflow.blog/2022/08/23/why-ai-is-having-an-on-prem-moment-ep-476/

[Oldwood14] Chris Oldwood, 'KISSing SOLID Goodbye', *Overload* 122 pages 14–17, available at https://accu.org/journals/overload/22/122/overload122.pdf#page=15, published August 2014.

[RISA] Punching Shear – Design, available at https://risa.com/risahelp/risafoundation/Content/Common_Design/Punching%20Shear%20-%20Design.htm

[Schmitz16] Nele Schmitz 'What we can learn about sustainable development from B. Russell's *The Conquest of Happiness*', posted March 2016 and available from https://www.researchgate.net/publication/299407539_What_we_can_learn_about_sustainable_development_from_B_Russell%27s_The_conquest_of_Happiness

[Stroustrup21] Bjarne Stroustrup 'What is the difference between C++98 and C++11 and C++14?' on https://www.stroustrup.com/bs_faq.html, last updated 23 July 2021.

[Wakely15] Jonathan Wakely 'What is an ABI and why is it so complicated?' presented at the ACCU conference 2015 and available at: https://accu.org/conf-docs/PDFs_2015/JonathanWakely-What%20Is%20An%20ABI%20And%20Why%20Is%20It%20So%20Complicated.pdf

[WHO19] World Health Organization, 'Burn-out an "occupational phenomenon": Internation Classification of Diseases, 28 May 2019, at: https://www.who.int/news/item/28-05-2019-burn-out-an-occupational-phenomenon-international-classification-of-diseases

User Stories and BDD – Part 1

Where did it all begin? Seb Rose explains the origins and evolution of the user story.

This article is taken from the first in a series of blogs that take a look at user stories, what they're used for, and how they interact with a BDD approach to software development.You could say that this is a story about user stories. And like every other story, it's important to choose where to begin – because, contrary to the advice given in the *Sound of Music*, it's not always a good idea to "start at the very beginning".

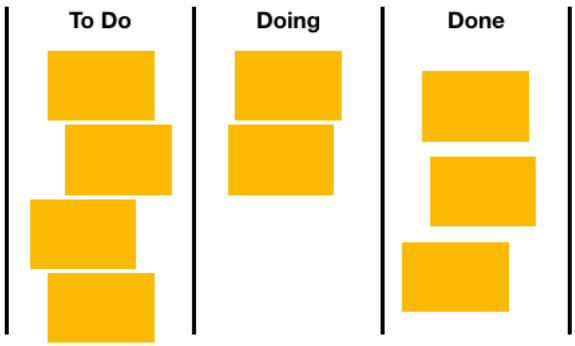

Figure 1

What's the problem?

Several years ago, I gave a talk at CukenFest called 'Let Me Tell You A Story' [Rose14]. It was inspired by a number of feature files that I'd seen online that started with text that looked remarkably like a user story (Figure 2).

```
Feature: Account Holder withdraws cash

As an Account Holder
I want to withdraw cash from an ATM
So that I can get money when the bank is closed

  Scenario Outline: Account has sufficient funds
    Given the account balance is "$100"
```

Figure 2

This seemed very strange to me, because **most features require the delivery of several user stories**. So how could you choose just one one to put at the top of your feature file?

My colleague, Matt Wynne, believes the practice was encouraged by the original feature file snippet [Cucumber] that shipped with TextMate [TextMate] many years ago when Cucumber was first gaining popularity.

The feature file snippet that ships with Sublime

Figure 3

Similar shortcuts also ship with Sublime Text and Visual Studio Code – and probably many other editors and IDEs (Figure 3).

The cargo cult effect [Wikipedia-1] has meant the majority of new Cucumber users who pick up these tools wrongly assume that putting their user story at the top of the feature file is the right thing to do.

The relationship between user stories and feature files is not well understood. This article is taken from a series of blogs aiming to put that right.

Software development before agile

Let's take a journey back in time, to the 20th century. How did people develop software back in the dim, distant past, before the agile manifesto? [Agile] The usual answer that people give is: "they used to use waterfall".

In the waterfall metaphor, software development is seen as a linear progression through various project stages: requirements analysis, architectural design, coding, testing, operations. The assumption is that work flows smoothly downhill – just like water. (See Figure 4.)

Waterfall

The term waterfall was first used to describe this way of working in a 1976 paper by Bell & Thayer [Bell76]. The authors credited a 1970 paper by Winston Royce [Royce70] that never actually uses the word, waterfall. Not only that, but Royce's paper emphasises that this way of working "*is risky and invites failure*".

Even sadder, in 1985 – 15 years after Royce had warned about the risks – the US Department of Defence "*captured this approach in DOD-STD-2167A, their standards for working with software development contractors.*" [Wikipedia-2]

It may surprise you to learn that even before Royce wrote his paper there were people who were working in a completely different way – using Iterative and Incremental Development (IDD). A paper written by Larman & Basili [Larman03] found traces of IID, that were "*indistinguishable from XP*", from as far back as 1958!

Seb Rose Seb has been a consultant, coach, designer, analyst and developer for over 40 years. Co-author of the BDD Books series Discovery and Formulation (Leanpub), lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

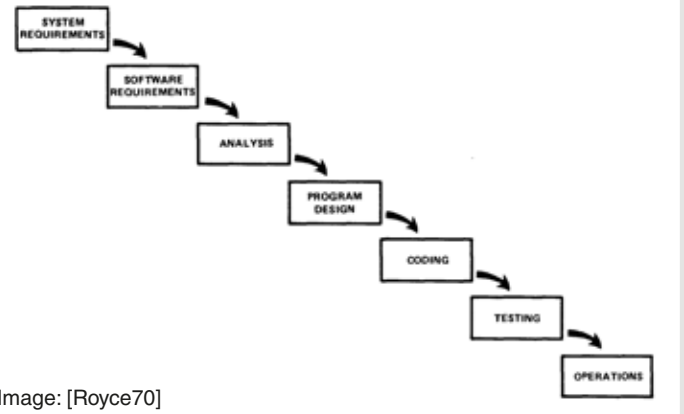

Image: [Royce70]

Figure 4

The problem with the waterfall approach is that at each stage people would spend weeks (or months) preparing detailed documents before passing them on to the team responsible for the next stage. Building software the same way you'd build a bridge or a skyscraper. It could be years before the first line of code was written, by which time requirements and priorities had often changed.

Most of the requirements that were discussed in detail were not delivered in the way that they had originally been envisaged. In fact, many never got delivered at all.

This was hugely wasteful, both in terms of the cost of delay [Wikipedia-3], and the time spent analysing and specifying features that would never see the light of day.

The risks associated with waterfall development have been known about for a long time. Iterative and incremental approaches – that mitigate those risks – are not a recent development, though it's only recently that they've become mainstream.

What are stories for?

Fast forward a few decades, and we find a handful of experienced programmers determined to tackle the fact that the waterfall approach led to significant waste.

These consultants were breaking the mould, coming up with their own practices and methodologies aimed at minimising the waste inherent in spending time doing detailed design work prematurely. Instead, they suggested working in an incremental and iterative fashion, deferring detailed design work until it was clear that the functionality was about to be delivered. Collectively, these became known as lightweight methodologies [Wikipedia-4].

One of these methodologies was called eXtreme Programming (XP) [XP].

The XP folk created the concept of a story as a "placeholder for a conversation" [Jeffries01]. This allowed them to do high-level decomposition of the work that needed to be done (stories), while accepting that more detailed analysis (conversations) would be needed before implementation started.

Another popular methodology at the time was Scrum (Figure 5). The Scrum Guide [Scrum] doesn't mention stories, but includes a concept that embraces them, called the Product Backlog Item (PBI). Each PBI represents some functionality that can be prioritised by the Product Owner. Scrum is not prescriptive about what format a PBI should take, although they are often represented as stories.

A story is a way of remembering that a piece of work might need to be done, without committing to actually doing it, or diving into the details too soon.

Putting the 'user' into user story

Lightweight methodologies were a big hit, but there were so many of them that it was difficult for anyone to decide which one to use. This led

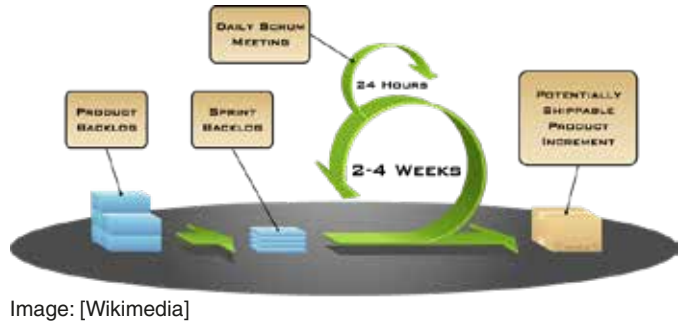

Image: [Wikimedia]

Figure 5

in 2001 to a gathering of 17 men at a ski resort, which in turn led to the agile manifesto [Agile]. It's been a very influential document, but many of the things that people associate with "agile" aren't mentioned in it at all. On the subject of stories, the manifesto is entirely silent. However, one of the twelve principles speaks to the problem stories aim to solve:

> Simplicity – the art of maximizing the amount of work **not** done – is essential.

The original description of the Planning Game in *Extreme Programming Explained* [Beck04] does mention stories – but the book doesn't use the term *user stories* anywhere. So, where did the inclusion of the word *user* come from?

I asked on Twitter [Rose19], and it seems that no one really knows – at least no one who's telling. Martijn Meijering dug into the internet archive and discovered that in a snapshot from 2002, *story* and *user story* were used interchangeably on the C2 wiki [C2] (which is where the XP community discussed things).

However it happened, calling them *user stories* emphasised that the goal is to "maximise the value of software produced by the team" [Beck04]. As stories began to be used more widely, teams needed a strong steer to ensure they prioritised work that would deliver value to the *user*, and the term *user story* did just that.

However, the use of the word *user* has led to many arguments. Should every story be from the perspective of a user? What users does the system have? Is the system administrator a user? Is the support team? Are the developers? What about another system consuming your API?

It's called a user story to help people focus on maximising the value of what's produced – but we must remember that value comes in many shapes and sizes.

A template is born

The challenges teams had writing user stories led to the creation of the *Connextra template* by Rachel Davies in the early 2000s. She found that it wasn't enough to just write the story from the user's perspective. Her teams seemed to have better conversations when the story contained three specific pieces of information:

1. The feature that needs to be discussed
2. The role that will get the benefit from the feature
3. The benefit that is expected

Rachel's template is now almost universal, in part due to its inclusion by Mike Cohn in his seminal books *Agile Estimating and Planning* [Cohn05] and *User Stories Applied* [Cohn04]:

> As a <*type of user*>
>
> I want <*capability*>
>
> so that <*business value*>

The template is guidance to help teams capture enough information in the story, so that the conversation they have is valuable.

Figure 6

Like any one-size-fits all system, while the template can be very useful for beginners, blind adherence to it can become problematic. Rachel herself describes some of the problems that treating this template as catechism can cause [Davies06]:

> ... this format can lead people to focus more on end users interests rather than the perspective of the person putting the business case forward. Also when given a template people can start treating story cards written this way as mini-requirements specs focussing on the written words rather than using stories as tools for driving a conversation. Even worse, stories that don't fit this form will be battered into shape until they do.

Stories help you ask the right questions about the context and reason for the request. The important part is not about the words on the card but the shared understanding developed in the team.

Back to the future

Fast forward another couple of decades, and we find the user story as an almost universally recognised term in our industry's vocabulary. However, in practice, we see the understanding of the term varies widely from team to team, from organisation to organisation.

Even though stories were intended to defer detailed analysis until the functionality was about to be developed, this is often not how they are used. There are many teams that use electronic tracking software (e.g. Jira) to build backlogs that contain hundreds of stories, each concealing detailed requirements. And despite the original intent of a story as a placeholder for a conversation, these details have never been discussed with the team.

So, while the term user story is widely used, it can mean very different things depending on who you talk to. This series of posts aims to fix that.

More ...

In this article, I've given an overview of the evolution of the user story as it is commonly used today. My blog digs into the BDD practice of Discovery, and the uncomfortable metamorphosis that stories undergo in the process. ■

References and further resources

[Agile] 'Manifesto for Agile Software Development', available at: https://agilemanifesto.org/

[Beck04] Kent Beck (2004) *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professinal, ISBN: 978-0321278654

[Bell76] T. E. Bell and T. A. Thayer, 'Software requirements: are they really a problem?' published by TRW Defense and Space Systems Group, available at https://static.aminer.org/pdf/PDF/000/361/405/software_requirements_are_they_really_a_problem.pdf

[C2] C2 wiki (2002): https://web.archive.org/web/20021105182544/http://c2.com/cgi/wiki?PlanningGame

[Cohn04] Mike Cohn (2004) *User Stories Applied for Agile Software Development*, Addison-Wesley Professional, ISBN 978-0321205681

[Cohn05] Mike Cohn (2005) *Agile Estimating and Planning*, Pearson, ISBN: 978-0131479418

[Cucumber] Cucumber Plain Text Feature Completions.tmPreferences, available on github at: https://github.com/bmabey/cucumber-tmbundle/blob/bb89925f54372282e6f7500cc53b746e44dbc31a/Preferences/Cucumber%20Plain%20Text%20Feature%20Completions.tmPreferences

[Davies06] Rachel Davies 'As a coach I want a story template so that people ask questions' posted on 1 December 2006, available at https://agilecoach.typepad.com/agile-coaching/2006/12/as-a-coach-i-want-a-story-template-so-that-people-ask-questions.html

[Davies19] Rachel Davies conversation on Twitter: https://twitter.com/rachelcdavies/status/1186313143611469826

[Jeffries01] Ron Jeffries 'Essential XP: Card, Conversation, Confirmation' posted 30 August 2001, available at http://www.extremeprogramming.org/

[Larman03] Craig Larman and Victor Basili, 'Iterative and Incremental Development: A Brief History', published in Computer June 2003, available at: https://www.craiglarman.com/wiki/downloads/misc/history-of-iterative-larman-and-basili-ieee-computer.pdf

[Rose14] Seb Rose, 'Let me tell you a story', a presentation delivered on 3 April 2014 as part of *CukeUp! 2014*, available from https://skillsmatter.com/skillscasts/5101-let-me-tell-you-a-story

[Rose19] Seb Rose, initiated conversation on Twitter: https://twitter.com/gojkoadzic/status/1186316368121159680

[Royce70] Winston Royce 'Managing the Development of Large Software Systems', *Proceedings of IEEE WESCON* August 1970, available at http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf

[Scrum] The Scrum Guide published by Scrum.org: https://www.scrum.org/resources/scrum-guide

[TextMate] 'TextMate: The missing editor for OX X', published 6 August (year unknown) at https://dhh.dk/arc/000270.html

[Wikimedia] Srum diagram (labelled), developed by Mountain Goat Software and available from https://commons.wikimedia.org/wiki/File:Scrum_diagram_(labelled).png under a Creative Commons Attribution 2.5 Generic Licence.

[Wikipedia-1] 'Cargo cult': https://en.wikipedia.org/wiki/Cargo_cult

[Wikipedia-2] 'Waterfall model': https://en.wikipedia.org/wiki/Waterfall_model

[Wikipedia-3] 'Cost of delay': https://en.wikipedia.org/wiki/Cost_of_delay

[Wikipedia-4] 'Lightweight methodology': https://en.wikipedia.org/wiki/Lightweight_methodology

[XP] 'Extreme Programming: A gentle introduction', available at: http://www.extremeprogramming.org/

Resources

Rebecca J. Wirfs-Brock 'Responsibility Driven Design', reprinted from *The Smalltalk Report* (date unknown) and available at https://www.wirfs-brock.com/PDFs/Responsibility-Driven.pdf

A Laboratory for Teaching Object Oriented Thinking - Kent Beck, Ward Cunningham

This article was published on Seb's blog on 31 October 2019: https://cucumber.io/blog/bdd/user-stories-are-not-the-same-as-features/

The Power of Ref-qualifiers

Ref-qualifiers are frequently overlooked. Andreas Fertig reminds us why they are useful.

In this article, I discuss an often unknown feature: C++11's ref-qualifiers. My book, *Programming with C++20* [Fertig21], contains the example in Listing 1.

```
class Keeper {
  std::vector<int> data{2, 3, 4};

public:
  ~Keeper() { std::cout << "dtor\n"; }
  // Returns by reference
  auto& items() { return data; }
};

// Returns by value
Keeper GetKeeper()
{
  return {};
}

void Use()
{
  // ❶ Use the result of GetKeeper and return
  // over items
  for(auto& item : GetKeeper().items()) {
    std::cout << item << '\n';
  }
}
```

Listing 1

What I have illustrated is that there is an issue with range-based **for**-loops. In ❶, we call **GetKeeper().items()** in the head of the range-based **for**-loop. By doing this, we create a dangling reference. The chain here is that **GetKeeper** returns a temporary object, **Keeper**. On that temporary object, we then call **items**. The issue now is that the value returned by **items** does not get lifetime-extended. As **items** returns a reference to something stored inside **Keeper**, once the **Keeper** object goes out of scope, the thing **items** references does as well.

The issue here is that as a user of **Keeper**, spotting this error is hard. Nicolai Josuttis [Josuttis21] has tried to fix this issue for some time (see [P2012R2]). Sadly, a fix isn't that easy if we consider other parts of the language with similar issues as well.

Okay, a long bit of text totally without any reference to ref-qualifiers, right? Well, the fix in my book is to use C++20's range-based **for**-loop with an initializer. However, we have more options.

An obvious one is to let **items** return by value. That way, the state of the **Keeper** object doesn't matter. While this approach works, for other scenarios, it becomes suboptimal. We now get copies constantly, plus we lose the ability to modify items inside **Keeper**.

ref-qualifiers to the rescue

Now, this brings us to ref-qualifiers. They are often associated with **move** semantics, but we can use them without **move**. However, we will soon see why ref-qualifiers make the most sense with **move** semantics.

```
class Keeper {
  std::vector<int> data{2, 3, 4};

public:
  ~Keeper() { std::cout << "dtor\n"; }

  // ❷ For lvalues
  auto& items() & { return data; }

  // ❸ For rvalues, by value
  auto items() && { return data; }
};
```

Listing 2

A version of **Keeper** with ref-qualifiers looks like Listing 2.

In ❷, you can see the ref-qualifiers: the **&** and **&&** after the function declaration of **items**. The notation is that one ampersand implies lvalue-reference and two mean rvalue-reference. That is the same as for parameters or variables.

We have expressed now that in ❷, **items** look like before, except for the **&**. But we have an overload in ❸, which returns by value. That overload uses **&&**, meaning it is invoked on a temporary object. In our case, the ref-qualifiers help us make using **items** on a temporary object safe.

Considering performance

From a performance point of view, you might see an unnecessary copy in ❸. The compiler isn't able to implicitly move the return value here. It needs a little help from us.

```
class Keeper {
  std::vector<int> data{2, 3, 4};

public:
  ~Keeper() { std::cout << "dtor\n"; }

  auto& items() & { return data; }

  // ❹ For rvalues, by value with move
  auto items() && { return std::move(data); }
};
```

Listing 3

In Listing 3, in ❹, you can see **std::move**. Yes, I have told you in the past only rarely to use **move** [Fertig22], but this is one of the few cases where moving actually helps, assuming that data is movable and that you need the performance.

Andreas Fertig is a trainer and lecturer on C++11 to C++20, who presents at international conferences. Involved in the C++ standardization committee, he has published articles (for example, in iX) and several textbooks, most recently Programming with C++20. His tool – C++ Insights (https://cppinsights.io) – enables people to look behind the scenes of C++, and better understand constructs. He can be reached at contact@andreasfertig.com

Ref-qualifiers give us finer control over functions, especially in cases where the object contains moveable data

Another option is to provide only the lvalue version of the function, removing the second `items` function. Without this function, all calls from a temporary object to the remaining lvalue `items` function result in a compile error. You have a design choice here.

Summary

Ref-qualifiers give us finer control over functions. Especially in cases like above, where the object contains moveable data, providing the l- and rvalue overloads can lead to better performance – no need to pay twice for a memory allocation.

We are using a functional programming style in C++ more and more. Consider applying ref-qualifiers to functions returning references to make them safe for this programming style. ■

This article was published on Andreas Fertig's blog on 5 July 2022 and is available at: https://andreasfertig.blog/2022/07/the-power-of-ref-qualifiers/

References

[Fertig21] Andreas Fertig (2021) Programming with C++20: Concepts, Coroutines, Ranges, and more, Fertig Publications, ISBN 978-3949323010

[Fertig22] Andreas Fertig (2022) 'Why you should use std::move only rarely', posted 1 February 2022 at: https://andreasfertig.blog/2022/02/why-you-should-use-stdmove-only-rarely/

[Josuttis21] Nicolai Josuttis, on Twitter: https://twitter.com/NicoJosuttis/status/1443267749854208000

[PR2012R2] Nicolai Josuttis, Victor Zverovich, Filipe Mulonde and Arthur O'Dwyer 'Fix the range-based for loop R2' available at https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2012r2.pdf

The ACCU Conference

Frances Buontempo extols the virtues of attending (or presenting).

ACCU is a group of programmers who care about their craft, sharing knowledge, continually learning, and generally seeking *Professionalism in Programming*. We welcome new members (https://accu.org/menu-overviews/membership/). In addition to our journals, discussion lists and local groups, we hold an annual Spring conference. Members get a conference discount, which more than covers the cost of joining.

The conference has a C++ focus, but covers all aspects of development and often includes other programming languages. The ACCU 2023 conference will be held in Bristol, UK, from 2023-04-19 to 2023-04-22; it will be run as a hybrid event, so you can either take part online or in person. There will also be two days of full-day tutorials on 2023-04-17 and 2023-04-18.

The next call for papers will start mid-October, so watch the ACCU website or the accu-general mailing list (https://accu.org/members/mailing-lists). You could propose a full day tutorial, a 90-minute session or a shorter session. Get your thinking hats on! If you have an idea, but haven't submitted a proposal before, plenty of ACCU members will be willing to read through an initial draft and give you feedback.

If you're not brave enough to give a 90-minute full talk, you could always consider giving a shorter talk or even give a lightning talk instead. These are five minute slots at the end of the day and volunteers step up during the conference. They almost always have everyone laughing and provide a great way to round off the day.

As mentioned, ACCU members get a discount, as do speakers. Also, keep your eyes open for early bird rates. If you personally can't afford the conference, it is worth asking if work will pay for you under a training budget. That was how I first managed to attend. I had to write up why the conference would be useful, and again, getting someone to read a business case through give early feedback is worth considering. Failing that, consider seeing if IncludeCpp (https://www.includecpp.org/) can help you find a scholarship, or check the ACCU website to see if the conference needs volunteers.

Previous conference schedules and slides are available on the ACCU website (https://accu.org/conf-previous/) and we now have a YouTube channel (https://www.youtube.com/c/ACCUConf), so you can listen to talks from 2016 onwards.

I have attended several conferences now, and spoken at a few. The journey from attendee to speaker has been valuable. I could never have imagined getting up and speaking years ago, but ACCU members and the conference itself are very supportive. I have learnt so much from the talks and made friends over coffee, lunch breaks and a few pints of beer. ACCU always had a reputation of being quite hard-core, with the audience potentially digging deep with knowledgeable questions. That is true, but the combination of difficult questions and a supportive environment make this my go-to conference.

As our conference chair, Felix, explains below, times are changing. The move to make the conference hybrid came about because of the pandemic. This has allowed people to take part who otherwise couldn't attend, which is a good thing. The next conference will be run by a different organizing committee, as Felix explains. We are bound to still see some usual suspects and new faces, ask probing questions and make new friends. Times change. C++ changes. New programming languages come and go. No matter what, it is always wonderful to meet with like-minded people, and go away inspired.

I look forward to potentially meeting you at 2023's conference, and perhaps listening to you talk. I should stop and hand over to Felix. I have a proposal to write! ■

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 since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

Thank you to Julie Archer and Archer Yates Ltd.

Julie Archer from Archer Yates Ltd, the organising company of the ACCU Conference, told me during the conference in April that she wants to step down and that she will hand over the conference to her very close friend Sarah Byrne who runs her own event organising company mosaicevents.co.uk.

This was a great shock for me, because Julie and her great team were inseparable from the ACCU conference, in my mind. She explained to me that because of comprehensible reasons she cannot continue to organise the conference.

When I attended the ACCU Conference for the very first time in 2015, Julie and her team were the first faces I saw when I arrived. They welcomed me very warmly and I felt immediately among friends. And this feeling continued throughout the whole event. I had attended several different events before but during this time I fell in love with the ACCU conference and started to play an active role. Julie's support from my very first day on the programme committee and later as chair was amazing! Whenever an attendee, a speaker, a sponsor or one from the committee had a problem or needed something unusual, the team from Archer Yates made it possible. Beside the 'normal' organisation they took special care that the conference dinner became an unforgettable event every year!

It was a stroke of luck that ACCU met Archer Yates Ltd. Their knowledge of planning and organising an event, combined with the association's know-how of a subject that is evolving so fast became a very successful joint venture. The ACCU Conference has evolved from a group of sessions after a WG21 (https://isocpp.org/std/the-committee) meeting to one of the first class software engineering conferences. This would never have been possible without a partner that had worked so absolutely brilliantly in the background. Julie and her team did everything possible that the conferences went so smoothly. The interaction between speakers and attendees lays in the centre of the event.

I will miss Julie and her team, but I wish them the best in the future!

Felix Petriconi

ACCU Conference Chair

The Model Student: The Regular Travelling Salesman – Part 1

Richard Harris begins a series of articles exploring some of the mathematics of interest to those modelling problems with computers.

The travelling salesman problem, or TSP, must be one of the most popular problems amongst computer science students. It is extremely simple to state; what is the shortest route by which one can tour n cities and return to one's starting point? Figure 1 shows random and optimal tours of a 9-city TSP.

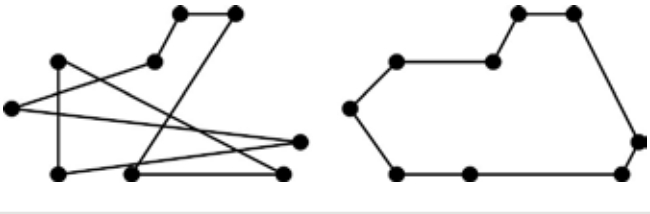

Figure 1

On first inspection, it seems to be fairly simple. The 9-city tour in Figure 1 can be solved by eye in just a few seconds.

However, the general case is fiendishly difficult. So much so that finding a fast algorithm to generate the optimal tour, or even proving that no such algorithm exists, will net you $1,000,000 from the Clay Mathematics Institute [Clay].

This is because it is an example of an NP-complete (nondeterministic polynomial complexity) problem. This is the class of problems for which the answer can be checked in polynomial time, but for which finding it has unknown complexity. The question of whether an NP-complete problem can be solved in polynomial time is succinctly expressed as 'is P equal to NP?' and answering it is one of the Millennium Prize Problems, hence the substantial cash reward.

The answer to this question is so valuable because it has been proven that if you can solve one NP-complete problem in polynomial time, you can solve them all in polynomial time. And NP-complete problems turn up all over the place.

For example, secure communication on the internet relies upon P≠NP. The cryptographic algorithms used to make communication secure depend on functions that are easy to compute, but hard to invert. If P=NP then no such functions exist and secure communication on an insecure medium is impossible. Since every financial institution relies upon such communication for transferring funds, I suspect that you could raise far more than $1,000,000 if you were able to prove that P=NP. Fortunately for the integrity of our bank accounts the evidence seems to indicate that if the question is ever answered it will be in the negative.

Richard Harris When he wrote this article, Richard had been a professional programmer since 1996. He had a background in Artificial Intelligence and numerical computing and was employed writing software for financial regulation.

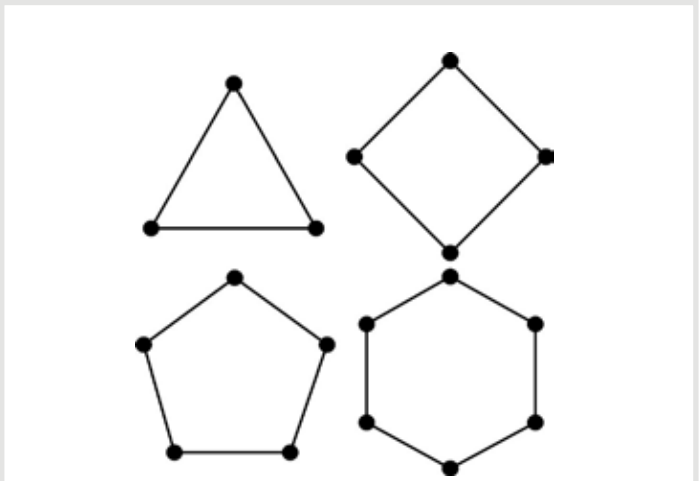

Figure 2

So, given that some of the keenest minds on the planet have failed to solve this problem, what possible insights could an amateur modeller provide?

Not many, I'm afraid. Well, not for this problem exactly.

I'd like to introduce a variant of the TSP that I'll call the regular travelling salesman problem. This is a TSP in which the cities are located at the vertices of a regular polygon. Figure 2 shows the first four regular TSPs.

The question of which is the shortest tour is rather uninteresting for the regular TSP as it's simply the circumference of the polygon. Assuming that the cities are located at unit distance from the centre of the polygons (i.e. the polygons have unit radius), the length of the optimal tour can be found with a little trigonometry.

Figure 3 shows the length of a side.

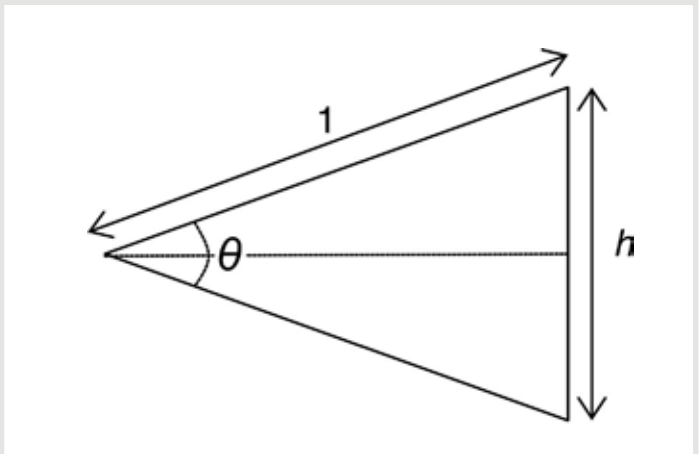

Figure 3

So are there any remotely interesting questions we can ask about the regular TSP?

For a tour of n cities, the length, l, of the optimal tour is given by:

$$\theta = \frac{2\pi}{n}$$

$$\sin\frac{\theta}{2} = \frac{h}{2}$$

$$l = n \times h = 2n\sin\frac{\pi}{n}$$

As n gets large, so θ gets small, and for small θ, $\sin\theta$ is well approximated by θ itself. We can conclude, therefore, that for large n, the length of the optimal tour is approximately equal to 2π. This shouldn't come as much of a surprise since for large n a polygon is a good approximation for a circle. In fact, it was this observation that Archimedes [Archimedes] used to prove that

> The ratio of the circumference of any circle to its diameter is less than $3^1/_7$ but greater than $3^{10}/_{71}$.

It's also fairly easy to find the length of the most sub-optimal tour. The key is to note that for odd n the furthest two cities from any given city are those connecting the opposite side of the polygon. Figure 4 shows the longest single steps and tour in a 5-city regular TSP.

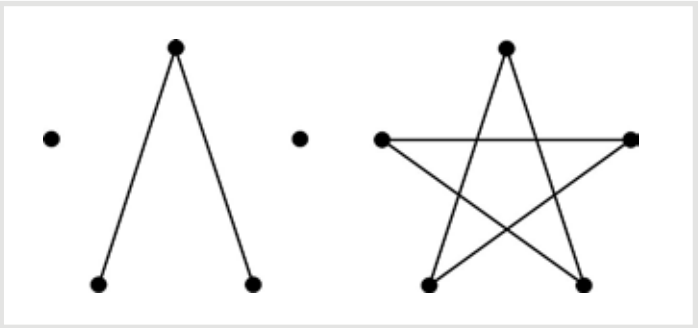

Figure 4

For odd regular TSPs, we can take a step of this length for every city, giving us a star shaped tour. We can calculate the length of this tour in a similar way to that we used to calculate the shortest tour length. Figure 5 shows the length of the longest single step.

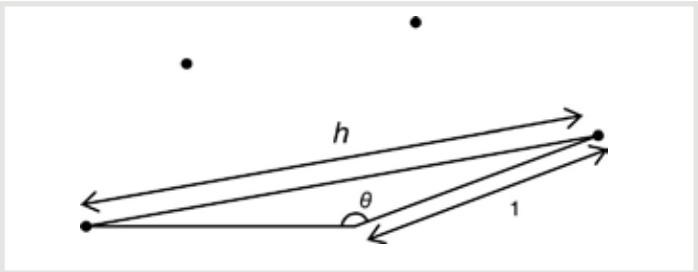

Figure 5

So for a tour of odd n cities, the length, l', of the worst tour is given by:

$$\theta = \frac{2\pi}{n} \times \frac{n-1}{2} = \pi - \frac{\pi}{n}$$

$$\sin\frac{\theta}{2} = \frac{h}{2}$$

$$l' = n \times h = 2n\sin\left(\frac{\pi}{2} - \frac{\pi}{2n}\right) = 2n\cos\frac{\pi}{2n}$$

This time for sufficiently large n, θ is small enough that $\cos\theta$ is well approximated by 1. For large odd n, therefore, the length of the worst tour is approximately equal to $2n$. Once again, we could have equally well concluded this from the fact that for large n the polygon is a good approximation for a circle for which the largest step is across the diameter.

For an even number of cities the worst single step is to the city on the opposite side of the polygon with a distance of 2. Unfortunately, each time we take such a step we rule it out for the city we visit, which will have to take a shorter step. So we can have $½n$ steps of length 2 and $½n$ steps of length strictly less than 2, giving a total length strictly less than $2n$.

This doesn't show that for an even number of cities the limit is $2n$, just that it cannot exceed $2n$. However, we can follow the longest step with the second longest to one of the first city's neighbours. We can repeat this for all but the last pair of cities for which we can take the longest step followed by the shortest. Figure 6 shows a $2n-2$ limit tour for 6 cities.

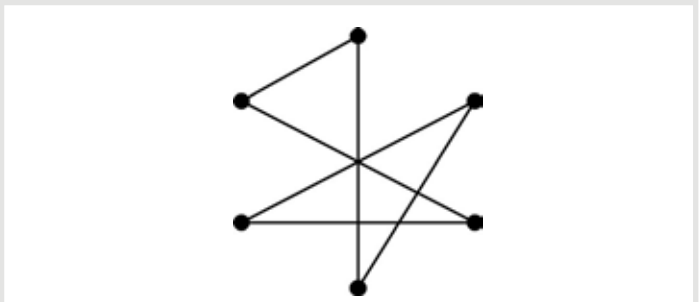

Figure 6

Whilst I haven't shown that this is the worst strategy, it does have a limit close to $2n$ for large n. It takes $½n$ steps with length 2, $½n-1$ steps with length approximately equal to 2, and one step with length approximately equal to 0 giving a total of $2n-2$.

So, are there any remotely interesting questions we can ask about the regular TSP?

How about what the average length of a tour is? Or, more generally, how are the lengths of random regular TSP tours distributed?

This is where the maths gets a little bit tricky, so we'll need to write a program to enumerate the tours directly. The simplest way to do this is

We can exploit the fact that our cities are located at the vertices of regular polygons

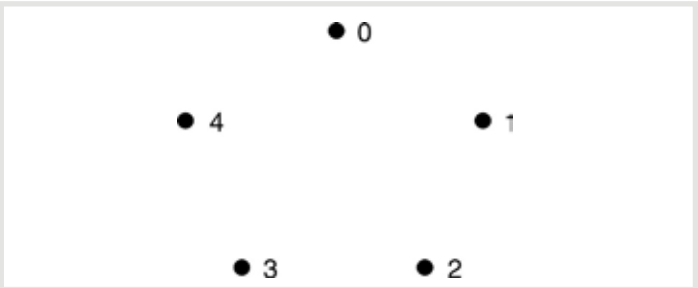

Figure 7

to assign each city a number from 0 to $n-1$ so we can represent a tour as a sequence of integers. Figure 7 shows labels for a 5-city regular TSP.

A tour can be defined as:

```
#include <vector>
  namespace tsp
  {
    typedef std::vector<size_t> tour;
  }
```

We'll need some code to calculate the distance between the cities. We can save ourselves some work if we calculate the distances in advance rather than on the fly. We can exploit the fact that our cities are located at the vertices of regular polygons by noting that due to rotational symmetry the distance between two cities depends only on how may steps round the circumference separate them. Listing 1 shows a class to calculate distances between cities.

The constructor does most of the work, calculating the distances between cities 0 to $n-1$ steps apart. Listing 2 calculates the distances between cities.

Of course, we could have also exploited the reflectional symmetry that means the distance between cities separated by i and $n-i$ steps are also the same, but I'm not keen to make the code more complex for the relatively small improvement that results. The code to retrieve the distance between two cities is relatively simple and is shown in Listing 3.

```
namespace tsp
{
  class distances
  {
  public:
    distances(size_t n);

    size_t size() const;
    double operator()(size_t step) const;
    double operator()(size_t i, size_t j) const;

  private:
    typedef std::vector<double> vector;

    size_t n_;
    vector distances_;
  };
}
```

Listing 1

```
tsp::distances::distances(size_t n) : n_(n),
   distances_(n)
{
  if(n_<3) throw std::invalid_argument("");

  static const double pi = acos(0.0) * 2.0;
  double theta = 2.0*pi / double(n_);
  double alpha = 0.0;

  vector::iterator first = distances_.begin();
  vector::iterator last = distances_.end();

  while(first!=last)
  {
    *first = 2.0 * sin(alpha/2.0);
    ++first;
    alpha += theta;
  }
}
```

Listing 2

```
double
tsp::distances::operator()(size_t step) const
{
  if(step>=distances_.size())
     throw std::invalid_argument("");
  return distances_[step];
}
double
tsp::distances::operator()(size_t i,
                           size_t j) const
{
  if(i>=n_ || j>=n_)
     throw std::invalid_argument("");
  return (*this)((i>j) ? i-j : j-i);
}
```

Listing 3

To calculate the length of a tour we need only iterate over it and sum the distances of each step (Listing 4).

```
double
tsp::tour_length(const tour &t,
                 const distances &d)
  {
  if(t.size()==0)
     throw std::invalid_argument("");

  tour::const_iterator first = t.begin();
  tour::const_iterator next = first+1;
  tour::const_iterator last = t.end();

  double length = 0.0;
  while(next!=last) length += d(*first++,
                                *next++);
  length += d(*first, t.front());

  return length;
}
```

Listing 4

unless we specify otherwise, we'll use twice as many buckets as we have vertices

```
namespace tsp
{
  class tour_histogram
  {
  public:
    struct value_type
    {
      double length;
      size_t count;
      value_type();
      value_type(double len, size_t cnt);
    };

    typedef std::vector<value_type>
       histogram_type;
    typedef histogram_type::
       size_type size_type;
    typedef const value_type &
       const_reference;
    typedef histogram_type::
       const_iterator const_iterator;

    tour_histogram();
    explicit tour_histogram(size_t vertices);
    tour_histogram(size_type vertices,
                   size_type buckets);

    bool      empty() const;
    size_type size() const;
    size_type vertices() const;

    const_iterator begin() const;
    const_iterator end() const;
    const_reference operator[](
        size_type i) const;
    const_reference at(size_type i) const;
    void add(double len, size_t count = 1);

  private:
    void init();
    size_type      vertices_;
    histogram_type histogram_;
  };
}
```

Listing 5

The final thing we'll need before we start generating tours is some code to keep track of the distribution of tour lengths. Listing 5 shows a class to maintain a histogram of tour lengths.

Most of the member functions of our histogram class are pretty trivial, so we'll just look at the interesting ones. Firstly, the constructors. Listing 6 shows how the tour histograms are constructed.

As you can see, unless we specify otherwise, we'll use twice as many buckets as we have vertices. This is because we've already proven that the maximum tour length is bounded above by $2n$, so it makes sense to restrict our histogram to values between 0 and $2n$ and dividing this into unit length ranges is a natural choice.

```
tsp::tour_histogram::tour_histogram(
   size_t vertices) :

vertices_(vertices),

histogram_(2*vertices)

  {
    init();
  }

tsp::tour_histogram::tour_histogram(
   size_type vertices,
   size_type buckets) :

vertices_(vertices),

histogram_(buckets)

{
  init();
}

void
tsp::tour_histogram::init()
{
  if(empty()) throw std::invalid_argument("");

  double step =
     double(2*vertices_) / double(size());
  double length = 0.0;

  histogram_type::iterator first
     = histogram_.begin();
  histogram_type::iterator last
     = histogram_.end();
  --last;

  while(first!=last)
  {
    length += step;
    *first++ = value_type(length, 0);
  }

  *first = value_type(double(2*vertices_), 0);
}
```

Listing 6

The `init` member function simply initialises the ranges for each of the buckets and sets their counts to 0. Note that, for our default histogram, the bucket identified with length l records tours of length greater than or equal to l-1 and less than l. Listing 6 shows initialising the tour histogram.

We can exploit the fact that our histogram buckets are distributed evenly over the range 0 to $2n$ when recording tour lengths. To identify the correct bucket we need only take the integer part of the tour length multiplied by the number of buckets and divided by $2n$.

The problem is that the number of tours grows extremely rapidly with the number of cities.

```
void
tsp::tour_histogram::add(double len)
{
  size_type offset((double(size())*len)/
     histogram_.back().length);
  if(offset>=size())
     throw std::invalid_argument("");
  histogram_[offset].count += 1;
}
```

Listing 7

Listing 7 shows adding a tour to the histogram.

Now we have all of the scaffolding we need to start measuring the properties of random tours of the regular TSP. Before we start, however, we should be mindful of the enormity of the task we have set ourselves.

The problem is that the number of tours grows extremely rapidly with the number of cities. For a TSP with n cities, we have a total of $n!$ tours which are going to take a lot of time to enumerate.

Table 1 shows the growth of $n!$ with n.

n	n!
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Table 1

We can improve matters slightly by considering symmetries again.

First, we have a rotational symmetry, in that we can start at any of the cities in a given tour and generate a new tour. By fixing the first city, we improve matters by a factor of n.

Secondly, we have a reflectional symmetry in that we can follow any given tour backwards and get a new tour. By fixing which direction we take around the polygon, we reduce the complexity of the problem by a further factor of 2.

Whilst exploiting the rotational symmetry is relatively straightforward, the reflectional symmetry once again requires quite a bit of house-keeping. Hence I shall only attempt to exploit the former for the time being.

The first thing we're going to need is a way to generate the initial tour.

```
void
tsp::generate_tour(tour::iterator first,
                   tour::iterator last)
{
  size_t i = 0;
  while(first!=last) *first++ = i++;
}
```

Once we can do that it is a simple matter of iterating through each of the remaining tours and adding their lengths to our histogram. Fortunately there's a standard function we can use to iterate through them for us; `std::next_permutation`. This takes a pair of iterators and transforms the values to the lexicographically next largest permutation, returning false if there are no more permutations. Using this function to calculate the histogram of tour lengths is relatively straightforward, as shown in Listing 8.

```
void
  tsp::full_tour(tour_histogram &histogram)
  {
    distances dists(histogram.vertices());
    tour t(histogram.vertices());
    generate_tour(t.begin(), t.end());

    do histogram.add(tour_length(t, dists));
    while(std::next_permutation(t.begin()+1,
                                t.end()));
  }
}
```

Listing 8

Note that exploiting the rotational symmetry of the starting city is achieved by simply leaving out the first city in our call to `std::next_permutation`.

Now we are ready to start looking at the results for some tours, albeit only those for which the computational burden is not too great.

Figure 8 (overleaf) shows the tour histograms for 8, 10, 12 and 14 city regular TSPs.

We can also use the histograms to calculate an approximate value for the average length of the tours. We do this by assuming that every tour that is added to a bucket has length equal to the mid-point of the range for that bucket. For our default number of buckets, this introduces an error of at most 0.5, which for large n shouldn't be significant. If you're not comfortable with this error, it would not be a particularly difficult task to adjust the add member function to also record the sum of the tour lengths with which you could more accurately calculate the average length. I'm not going to bother, though.

The approximate average lengths of the above tours, as both absolute length and in proportion to the number of cities, are given in Table 2.

The distributions shown by the histograms and the average tour lengths both hint at a common limit for large n , but unless we can analyse longer tours we have no way of confirming this. Unfortunately, the computational expense is getting a little burdensome as Table 3 illustrates.

So can we reduce the computational expense of generating the tour histograms? Well that, I'm afraid, is a question that shall have to wait until next time. ■

n	μ	μ/n
8	10.99	1.37
10	13.51	1.35
12	16.07	1.34
14	18.62	1.33

Table 2

n	time (seconds)
8	0.002
10	0.180
12	22.140
14	4024.410

Table 3

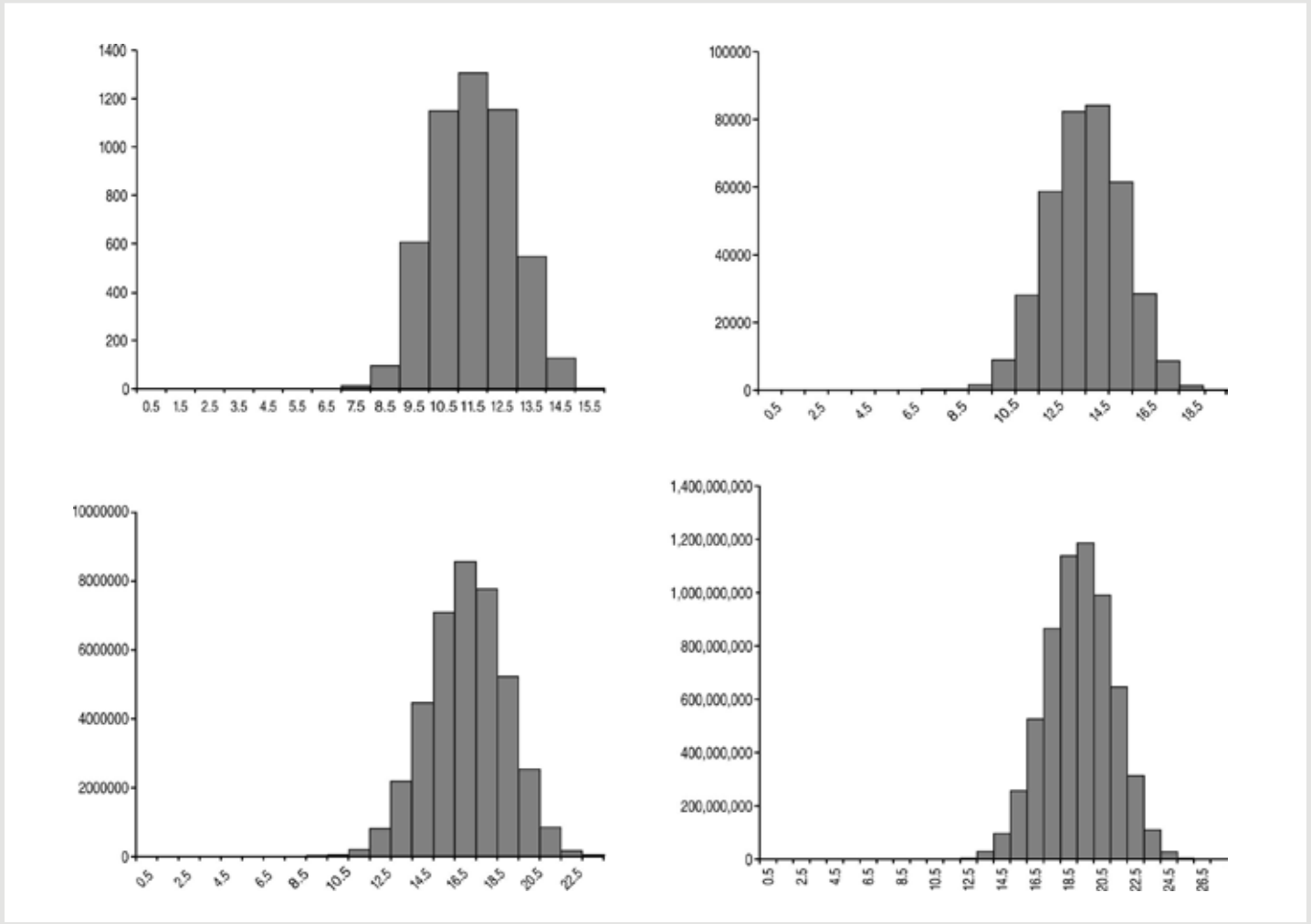

Figure 8

Acknowledgements

With thanks to Larisa Khodarinova for a lively discussion on group theory that lead to the correct count of distinct tours and to Astrid Osborn and John Paul Barjaktarevic for proof reading this article.

References and further reading

[Agnihothri98] Agnihothri, 'A Mean Value Analysis of the Travelling Repairman Problem', *IEE Transactions*, vol. 20, pp. 223-229, 1998.

[Archimedes] Archimedes, *On the Measurement of the Circle*, c. 250-212BC.

[Basel01] Basel and Willemain, 'Random Tours in the Travelling Salesman Problem: Analysis and Application', *Computational Optimization and Applications*, vol. 20, pp. 211-217, 2001.

[Beardwood59] Beardwood, Halton and Hammersley, 'The Shortest Path Through Many Points', *Proceedings of the Cambridge Philosophical Society*, vol. 55, pp. 299-327, 1959.

[Clay] Clay Mathematics Institute Millennium Problems: http://www.claymath.org/millennium

[Hoffman96] Hoffman and Padberg, 'Travelling Salesman Problem', Encyclopedia of Operations Research and Management Science, Gass and Harris (Eds.), Kluwer Academic, Norwell, MA, 1996.

[Jaillet93] Jaillet, 'Analysis of Probabalistic Combinatorial Optimization Problems' in *Euclidean Spaces, Mathematics of Operations Research*, vol. 18, pp. 51-71, 1993.

Dr Richard Harris died over the summer. His unexpected death was a shock and those who knew him miss him so much. Richard had been a member of ACCU for a very long time. He wrote many articles and spoke at the conference on many occasions. Our member's magazine, *CVu*, September edition included people's memories of Richard. They only hint at who he was.

He wrote a website, https://www.thusspakeak.com/, which will eventually disappear. It has many puzzles and explanations of maths and stats. It contains a maths library written in JavaScript, just to prove what's possible. His usual language of choice was C++, but he had fun writing various numeric types and functions in JavaScript. He claimed he wrote it all by hand, including the style-sheets, in Notepad. If you knew him, this isn't entirely surprising. Some ACCU members are attempting to archive the website so it doesn't disappear forever. Watch this space.

We will reprint the follow-up travelling salesman article in the next *Overload*, but feel free to look back through the other articles he wrote for ACCU (listed at https://accu.org/journals/nonmembers/overload_author_members/#[IH]).

Afterwood

War and Peace is a famously long novel that mixes fiction with history and philosophy. Chris Oldwood muses on the categorisation of computing books – and where to put them.

After serving many years as the playroom (nay, 'dumping ground') the kids have finally grown-up to the point at which it can serve as something a little more refined. The need to redecorate has created the impetus required to pull the books off the shelves and have a sort out before putting them back in what I hope is a more 'logical' order. The room now takes on the grandiose title of 'library', not by virtue of the number of books, it has less than before, but simply because the ratio of books to other 'crap' is such that they now dominate significantly, and you stand a chance of being able to use the sofa and chairs for actually sitting down.

Having my programming books share the same space as the family's other books is a fairly recent event. Prior to this they were stacked on shelves in the downstairs toilet until gravity conspired with Jeffrey Richter *et al* to cause the shelves to sag and bow to such extent that they made the occupants using the facilities as nature intended to become deeply uncomfortable. Richter wasn't the only culprit, although he seems to have a penchant for mighty tomes; other notable authors that stress-tested the woodwork included the likes of Brockschmidt (OLE), Hohpe (Patterns) and Meyer (OO), with Robert Binders' *Testing Object-Oriented Systems* the heavyweight champion.

Book obesity is not a new topic to this journal with its editor raising similar concerns some 10 years ago [Buontempo12]. In what became yet another failed attempt at writing an editorial, Fran distracted herself by weighing the classic K&R book on the C programming language (375g it turns out) while lamenting the 'cost' (time-wise) of modern technical books. I've definitely got books where even the index would weigh more than 375g!

For the sake of balance I should point out there is an upside to having 1000+ page books – they make excellent weights when pressing leaves or flowers, or fixing things with glue that require continuous pressure while setting. Such weight comes with great responsibility, though, and one needs to be careful not to place the book too precariously, lest it slip off the table and land on one's toes. Fran may have used K&R for her book scale, but misfortune has suggested to me that the Richter scale can equally be applied to the pain level of bruised toes as much as it does earthquakes.

Putting the 'technical library' in the downstairs toilet was partly borne out of practicality. The desk performed admirably as a level 1 cache but the latency of accessing the level 2 cache in the attic was dreadful, especially in the evening when my kids were asleep as they played host to the hatch and loft ladder. The shelves in the toilet drastically improved the (amortized) L2 access time without disrupting the facilities due to a clustered configuration (aka secondary toilet in the upstairs bathroom).

There was also a slightly humorous element to using such an unusual location as a library – my wife would suggest to visitors that it was to keep them 'hidden away' and avoid tainting the collections of Penguin Classics and Dr Seuss. In retrospect, I'm not buying this argument as when I was finally granted safe haven with the rest of the family's books, they were still relegated to the topmost shelf as if they were some kind of illicit material. The extra reinforcement for the new top shelf may also have had a bearing on this decision along with me, the tallest member of the family, being the only one who needed to reach that high up.

Unpacking the boxes of books once the fresh lick of paint had dried provided a much needed opportunity to revaluate the existing organization which was analogous to the complexity guarantee for a `std::vector` – inserting anywhere except near the end was time-consuming. Naturally, I paid the price and ended up with a retrieval complexity approaching O(N). (Sorting the books was very much left as an exercise for the reader, which they never seemed to get around to…)

I can't say that organising technical books primarily by author, title, or year has ever felt natural to me. Luckily (based on my limited data set) authors tend to stick to the same topics and, as technologies tend to come and go, grouping around them ends up leading to a clustering based on author and age as a side-effect. My programming career to date has largely involved writing applications and services on Windows in C++ and C#, so that provides three obvious initial collections with only *Windows++* and *Windows via C++* causing any immediate hesitation. But what about the rest?

What I really need is a book on category theory. I'm aware that Bartosz Milewski has written one specifically for programmers, but the trouble then is where do I file that? I already have Milewski's *C++ in Action* from the early 2000s so do I create a new category just for him? Looking at the remaining stack of books they are largely singletons, although design patterns also now stands out as a notable genre too.

Naturally C++ and C# got amalgamated into a larger section on programming languages, which I did alphabetise. It all went swimmingly until I came to the x86 and Z80 handbooks – do they go at the beginning under A for assembly language, or the end under their CPU names? I did briefly consider making the dilemma go away by redesigning the bookshelf as a circular linked list so that A and Z would adjoin, but only very briefly.

What's left after that little exercise are a whole bunch of books – old and new – about a variety of software development topics covering process, reviewing, quality, history, etc. with no obvious home. Hence, with no self-contained sub-categories leaping out I'm inclined to borrow from J.B. Rainsberger [Rainsberger12]:

> Junior programmer's bookshelf: 90% APIs and programming languages; Senior programmer's bookshelf: 80% applied psychology.

…and file them all under a single section: applied psychology. Sorted! ■

References

[Buontempo12] Frances Buontempo, 'Editorial: Too Much Information', *Overload* 111, published October 2012, available at: https://members.accu.org/index.php/journals/1885

[Rainsberger15] R. B. Rainsberger, on Twitter, tweeted 1 July 2015, https://twitter.com/jbrains/status/616228270841962496

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the comfort of his breakfast bar. He has resumed commentating on the Godmanchester duck race but continues to be easily distracted by messages to gort@cix.co.uk or @chrisoldwood

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe