Tonight we're going to talk about how to be a better developer.
I'm going to share a bit of my personal journey, some of the practices that I've found to be useful, and some tips that you may be able to use.

Image credit: Flickr user jichikawa

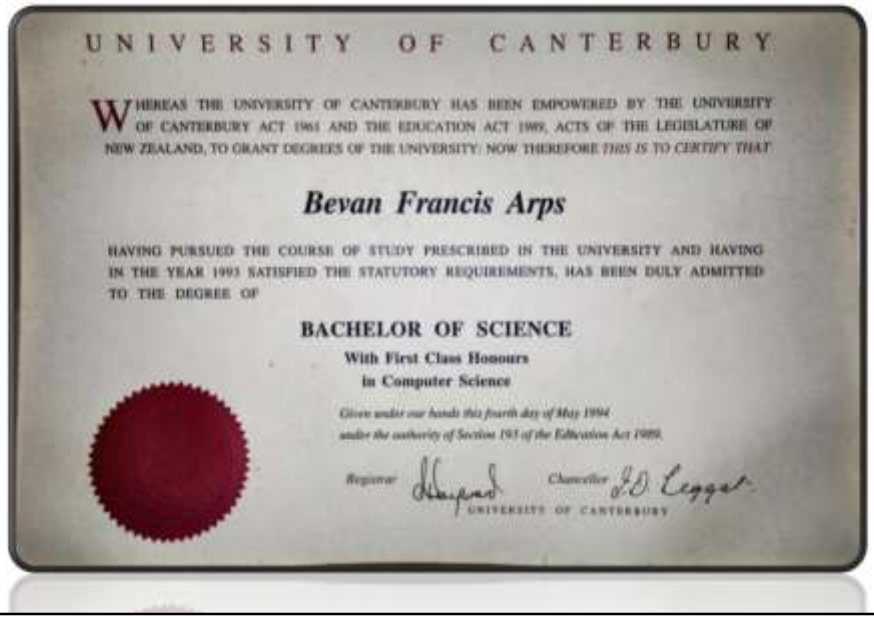
I started programming in 1982 when my parents purchased my first computer.
It was a ZX81.
The standard model (as shown) came with 1KiB of memory …
I had an expansion pack with a whole 16KiB.

But my parents were mean. They wouldn't buy me any games to play.

Instead, they bought me the Sinclair Learning Lab,
a course in learning to program Sinclair Basic
So I figured I'd make my own games.

As an aside – the ZX81 used audio cassette tapes for storage.

At High School, many hundreds of hours working with Apple //e machines.
Applesoft Basic, 6502 assembly language, USCD Pascal
and lots and lots of games.



UNIVERSITY OF CANTERBURY

WHEREAS THE UNIVERSITY OF CANTERBURY HAS BEEN EMPOWERED BY THE UNIVERSITY OF CANTERBURY ACT 1961 AND THE EDUCATION ACT 1989, ACTS OF THE LEGISLATURE OF NEW ZEALAND, TO GRANT DEGREES OF THE UNIVERSITY: NOW THEREFORE *THIS IS TO CERTIFY THAT*

Bevan Francis Arps

HAVING PURSUED THE COURSE OF STUDY PRESCRIBED IN THE UNIVERSITY AND HAVING IN THE YEAR 1993 SATISFIED THE STATUTORY REQUIREMENTS, HAS BEEN DULY ADMITTED TO THE DEGREE OF

BACHELOR OF SCIENCE

With First Class Honours
in Computer Science

Given under our hands this fourth day of May 1994
under the authority of Section 193 of the Education Act 1989.

Registrar Harpard *Chancellor* J.D. Legge

UNIVERSITY OF CANTERBURY

I went to Canterbury University and with a degree in Computer Science.

I’ve worked for Apple as a hardware service technician.

In 1997 I started my career as a full time professional software developer, working with Borland Delphi .

Then in 2004 I moved into the world of C# and .NET


So
What?

I’ve been programming in some form for 33 years.
I’ve been doing it professionally for 18 years.

Shouldn’t I be good at it by now?

Shouldn’t I at least *think* that I’m good at it?

Fact is, just a few years ago, I felt like a muppet.

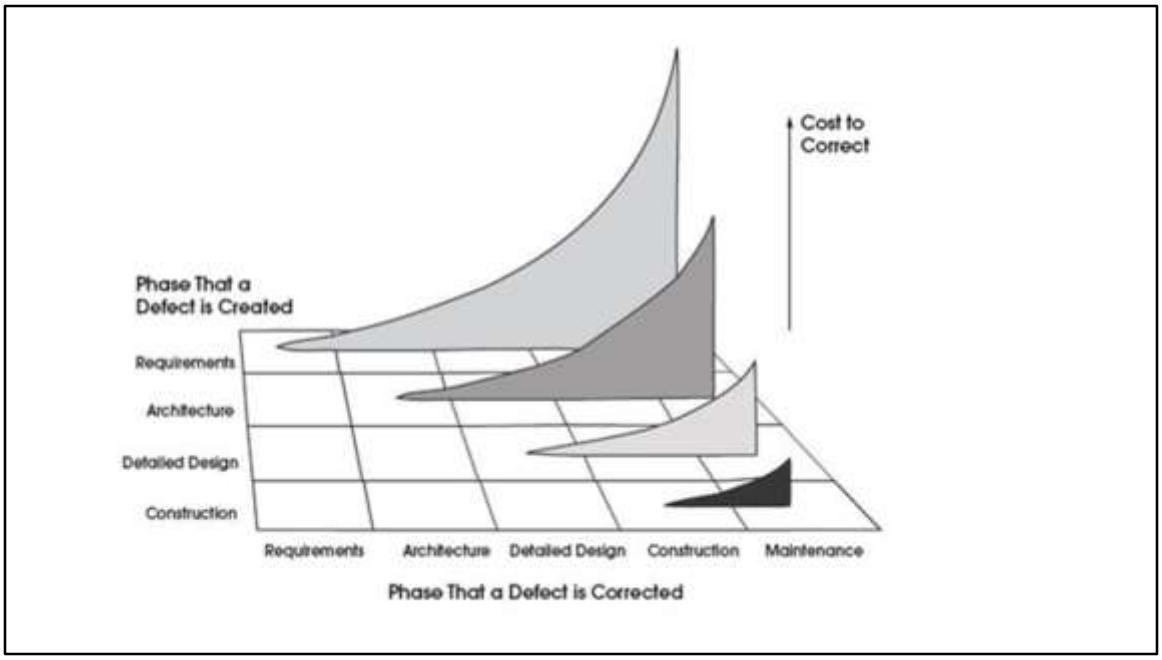
I believed my code substandard,
With too many defects
and that I'd soon end up obsolete and unemployable.

Animal image credit Disney

Worse, writing code wasn't fun any more.

As you can imagine, I was in a dark place.

Image Credit: Flickr user jseliger2



Here's one of the reasons I thought I had cause to worry.

This graph is from Steve McConnell's famous book "Code Complete"
showing that the earlier any defect is identified, the cheaper it is to remedy.

In fact, the cost of fixing a defect after go-live can be as high as 100,000 times the cost of fixing it earlier.

My believed code had too many defects, too many "issues".
And so I thought I was too expensive a developer.

So that's where I was.

But where did I want to be?

I've always loved puzzles.
I wanted to recapture the fun of being a puzzle solver.
I wanted to get back to the puzzle solving delight that led me into being a professional software developer in the first place.

I wanted to regain the feeling of mastery – and to feel like I was contributing value and solving other peoples problems.

I wish I could tell you the journey was a simple one,
That the path I took was straightforward.

Image: Winding road by Flickr user kriechstrom

But, it was more like this.

I knew **where** I wanted to be – I knew **who** I wanted to be.

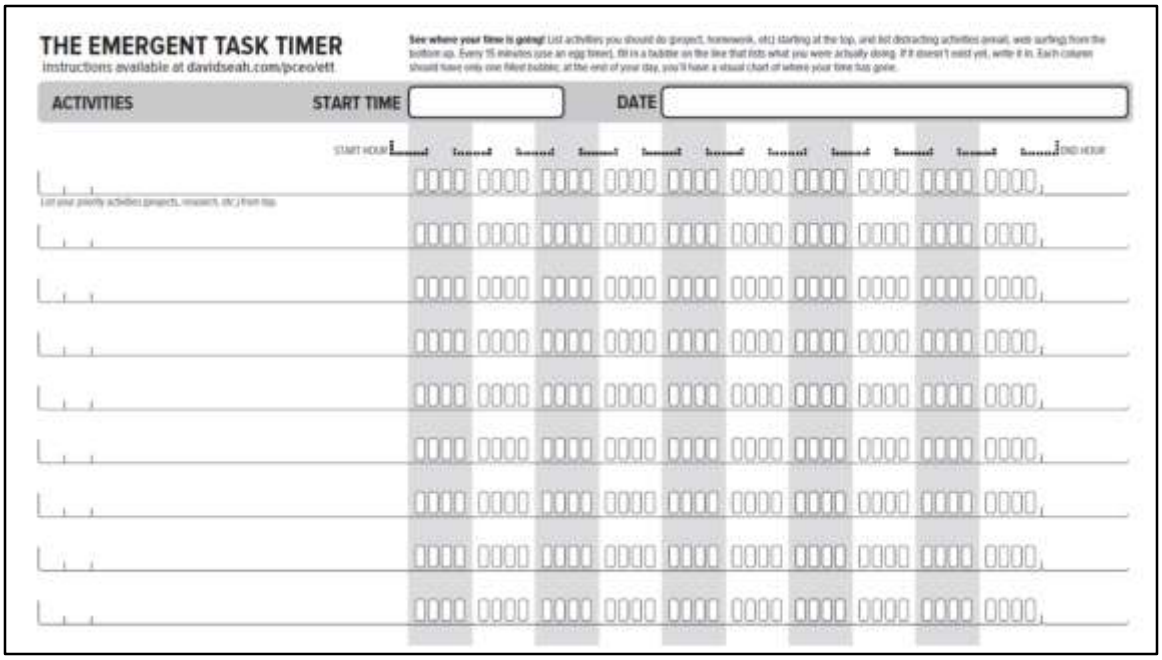
But how did I get there from here?

Longleat Maze – Flickr user joncandy

The first thing I needed was data.

I went to work every day and I was busy.
I was rushed off my feet.
But where did my time go?
I really had no idea.

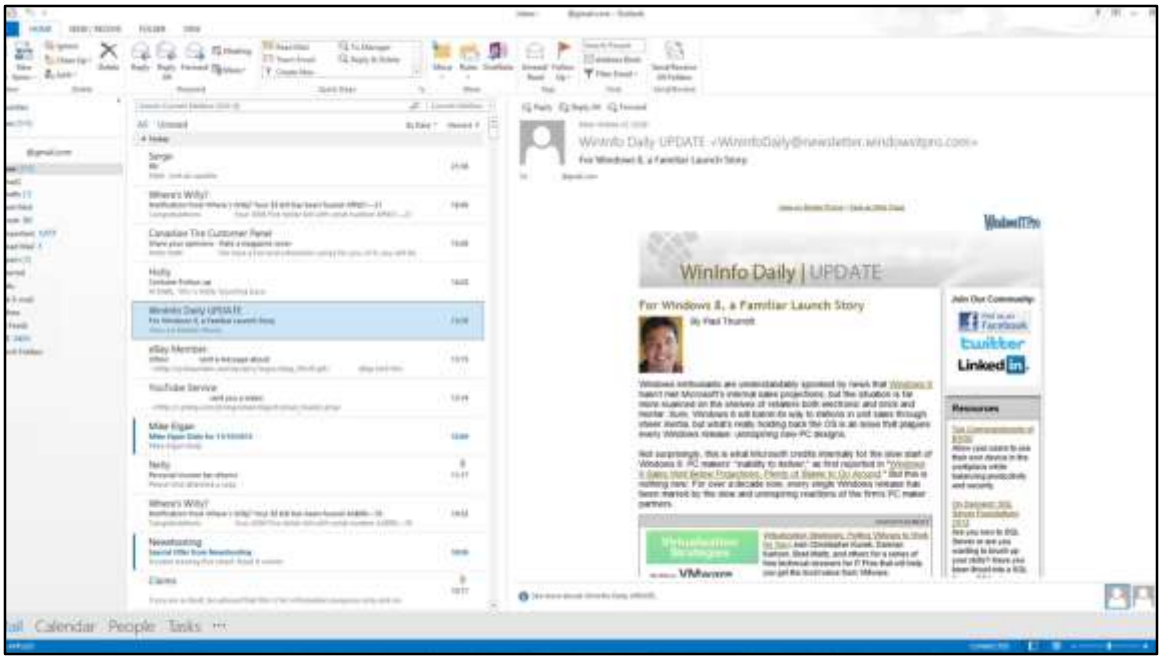
Image Credit: Flickr user Patrick Slattery



I started using the Emergent Task Timer
from the Printable CEO,
a collection of resources published by David Seah at davidseah.com.
Each bubble represents 15 minutes of time.
By tracking all of my time, every day, I could see what I actually worked on.

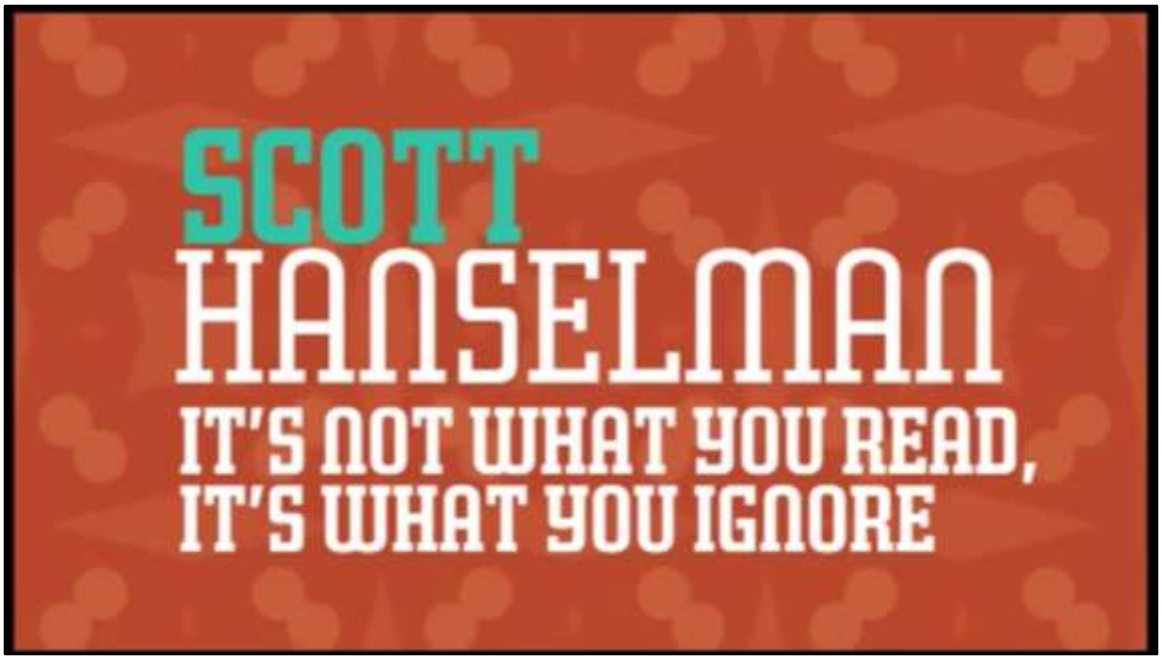
Anyone hazard a guess?

http://davidseah.com/node/the-emergent-task-timer/



Most of my time was spent in Outook.
Literally hours of every day.
Great communication, but I wasn't getting anything done.
There are times that I need to provide a quick response to an email.
But did I need to be responding to every email within minutes?
No, I didn't.

This is Scott Hanselman.
Blogger, Technologist, Teacher, Evangelist.



Webstock 2012 presentation “It’s not what you read, it’s what you ignore”

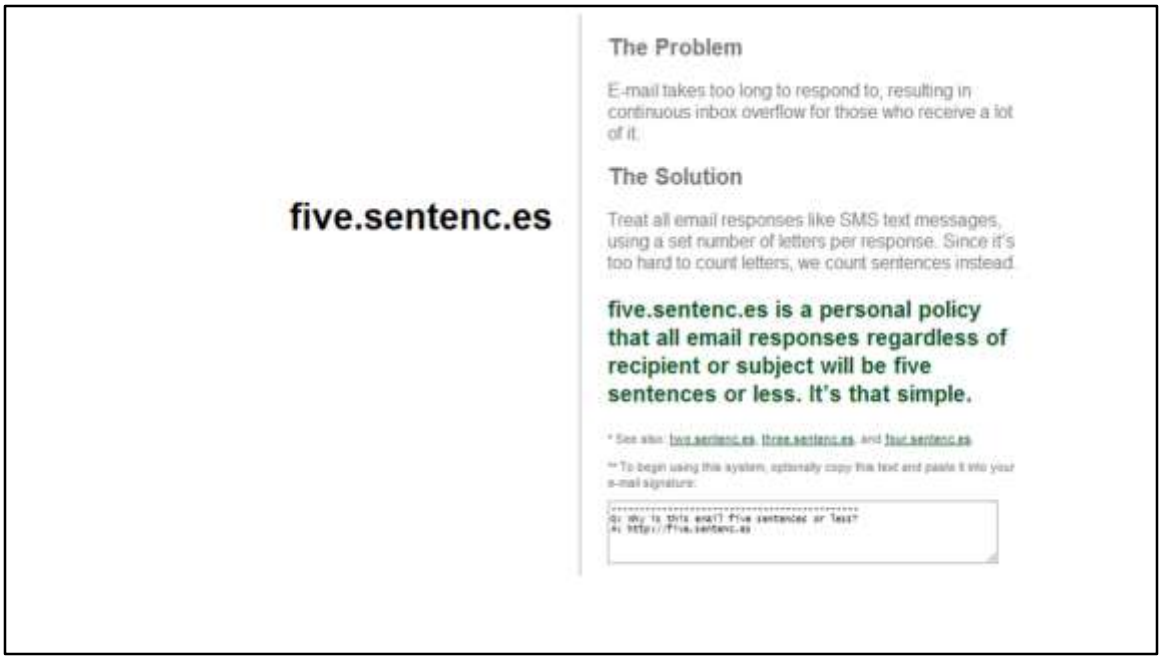
https://vimeo.com/39020426

Scott's advice:
Deal with your email at specific times of the day **only**.
Fence off the rest of the time as a chance to concentrate.
Book it as a meeting of one if you have to.

So that was my next change: to stop living in Outlook all the time

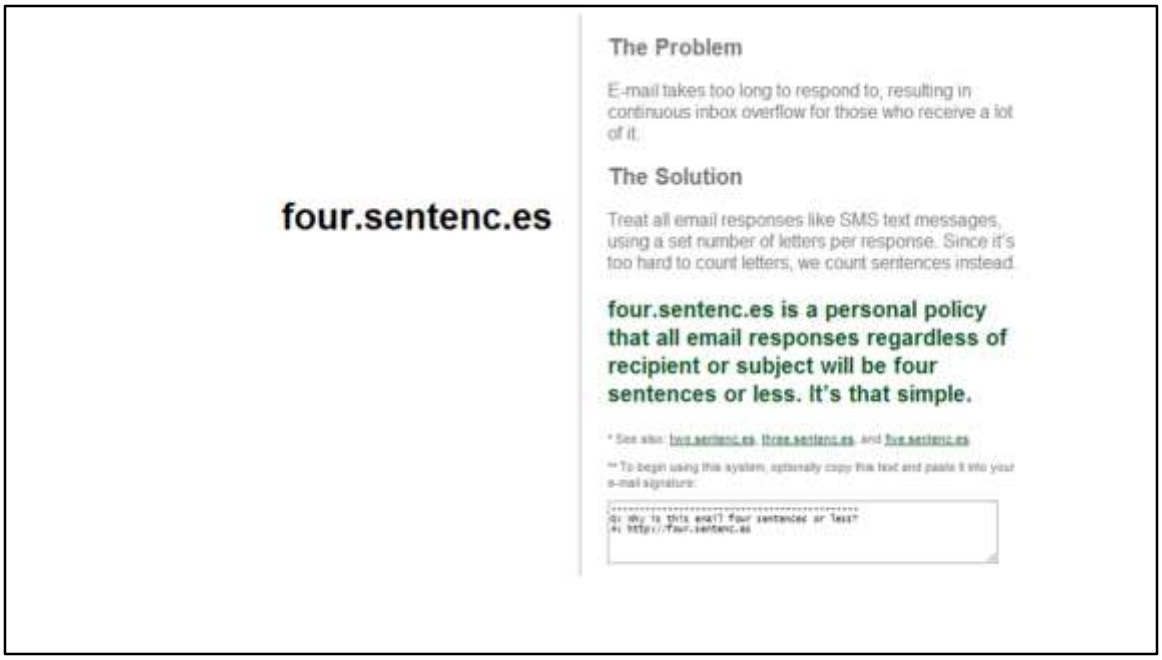
When you do deal with your email, Scott suggests there are only four possible actions to take on a message in your Inbox.

I've found that writing long emails is a waste of time.
People don't read them.
Some people will come back asking questions
Already answered in the third paragraph.



So don't send long emails.
Limit your responses to a few sentences (or less).
If an email requires more, go have a conversation or schedule a meeting.
Then email a summary of what's agreed.

Think that five sentences is too liberal?



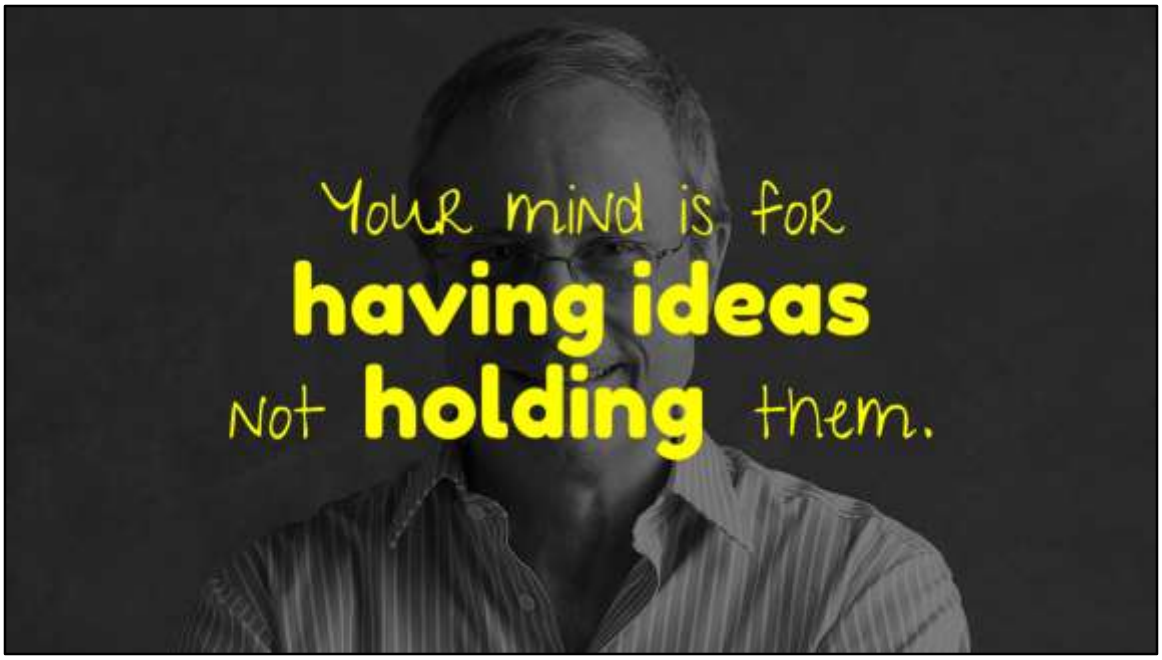
Perhaps you need to limit yourself to four. Or less.

Me, I tend to run off at the mouth a bit – sticking to five sentences is a challenge.

Having put email into it's place was a start – but only a start.

This is David Allen, inventor of “Getting Things Done”,
a philosophy and system of personal organisation.

He’s built an entire business around his ideas,
But Dave has a simple philosophy.



Let me illustrate

Think of what's going on when your
brain is full of everything you have to do:

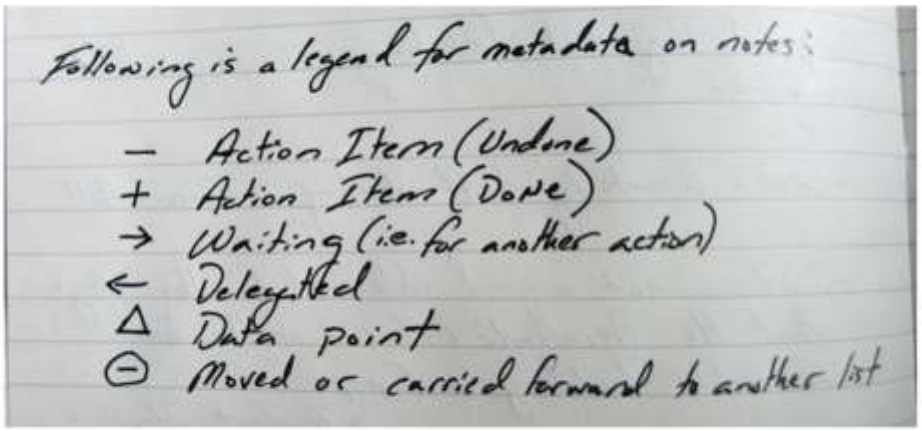
- Today's production bug fix
- The half completed feature
- Next weeks change freeze
- Groceries to buy on the way home
- Taking your daughter to Scouts tonight

There's very little "head space" free to actually get anything done.

Use some other system – that your brain trusts – and you can clear your head of all of the distractions and have much more capacity to do what needs to be done.
You might be five times smarter than you were.

So I started keeping a notebook that lists all of my “actions” – things that need doing, whether work related or not.
It’s the next action to take for *everything*.

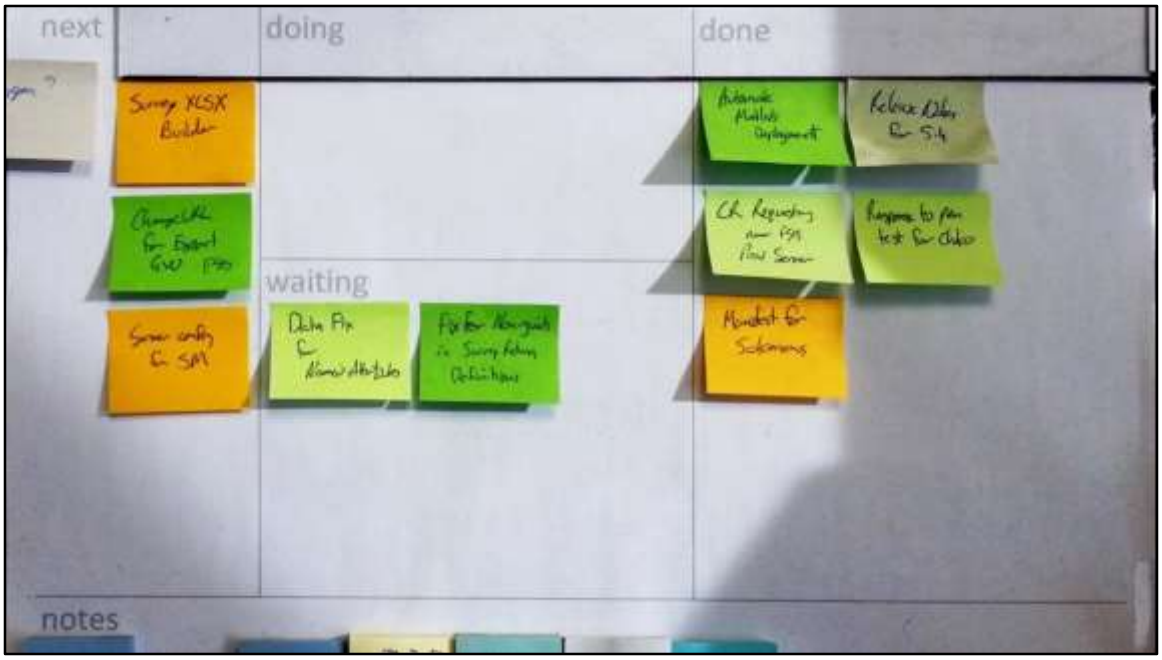
Key: For a To-Do list to work, it can only be “next action” items;
no place here for “plan a holiday” or anything fuzzy.



I've been using the dash-plus system as it's simple and explicit.
Every mark starts with a dash.

What I found was that my list of work stuff was intruding into the rest of my week.

http://patrickrhone.com/2013/04/22/the-dash-plus-system/



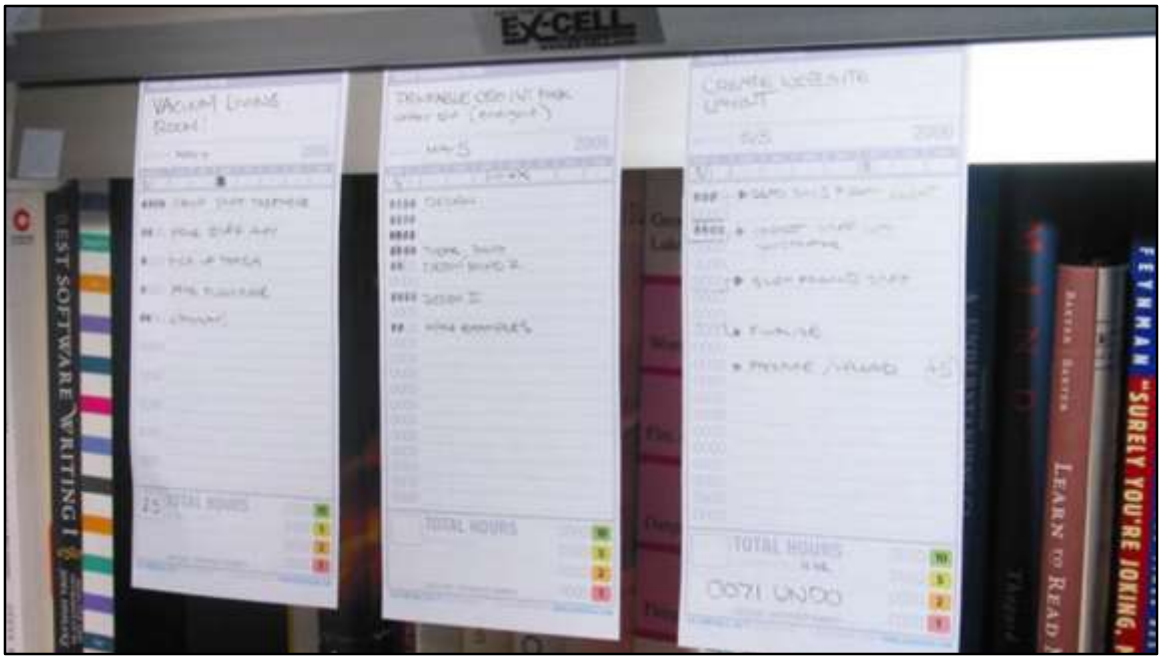
So I created a Kanban board to list work tasks.

Advantage: other people can see what you are doing without having to bother you.

I once caught my project manager checking my Kanban board for an update!

When I need to plan for an event, I end up with future tasks - ones that I can't start yet.

To keep them out of the core TODO list,
I use these "task order-up" forms, also from David Seah.



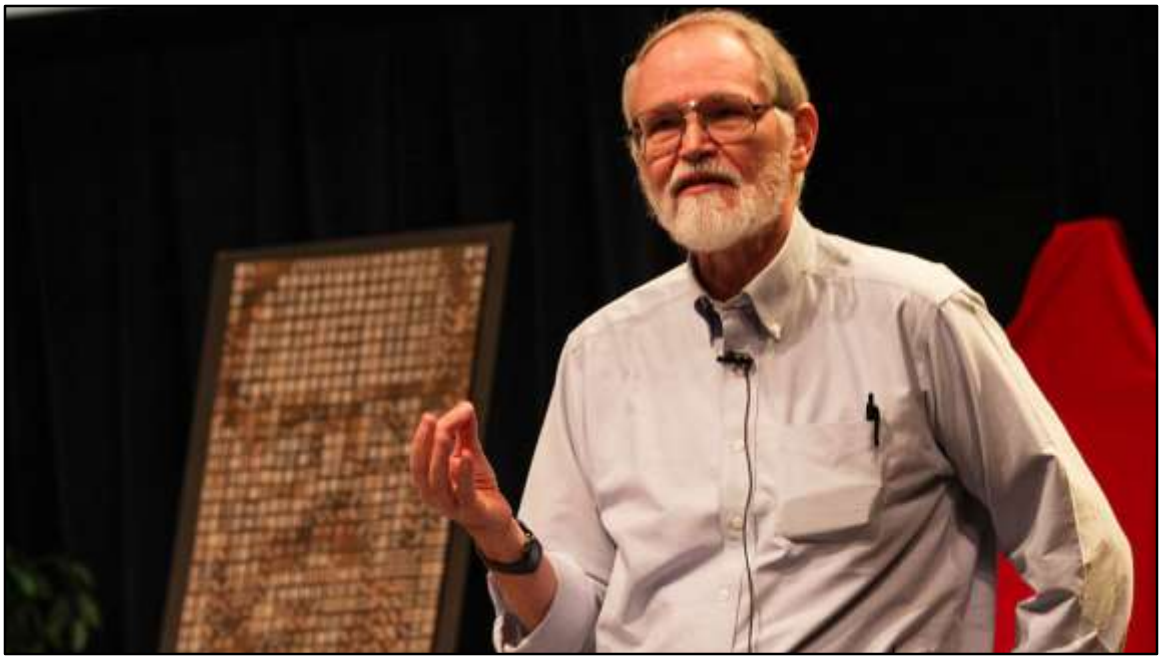
They're designed to be used on a rail, so you can see all the tasks in priority order
Like a short order cook
But I don't have a rail (yet).

What I've discovered: My brain leaks. A lot.

Keeping a list of things to do
– a list that was written down outside my head
– has meant that I'm no longer losing track of things.

This has helped a lot.

With my time and brain freed up, I turned my attention to a more technical level, beginning with debugging – something I felt I was spending to much time on.



This is Brian Kernighan. Co-author of the “C Programming language”, Co-developer of the original Unix.

Photograph by Ben Lowe (Creative Commons via Wikipedia)

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Photograph by Ben Lowe (Creative Commons via Wikipedia)

Since my original concern was around the rate of bugs in my code, I started keeping track.

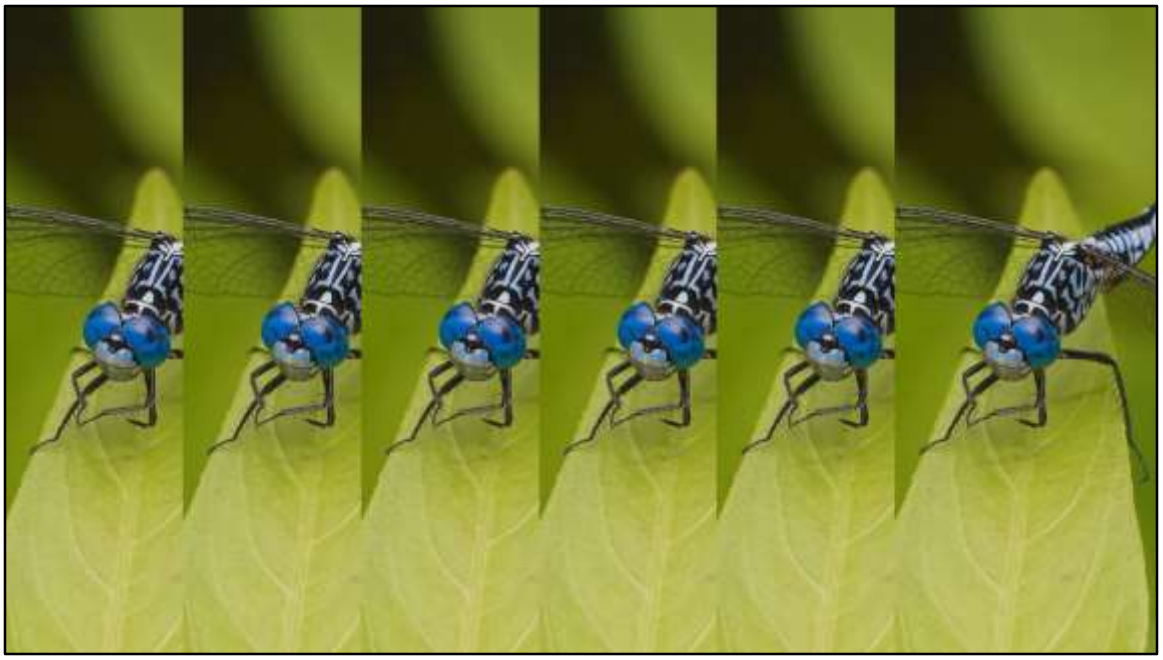
Now, I'm not talking about logging every bug in JIRA – that starts once the code is released into testing. But just to track – with just a few words in my notebook – the bugs that I found while coding.

I've found three useful questions to ask about every bug

What is the bug, exactly. Be precise.

Why was the code written that way?

How should the code have been written to avoid the issue?



What I found was that some of my defects were recurring.
- I made some of the same mistakes over and over again.
Paying particular attention to eliminating those had a big payoff.

For me, one of these recurring mistakes is this: paying attention to the “happy path” of the code, and not considering all the different ways it can – and will - go wrong.

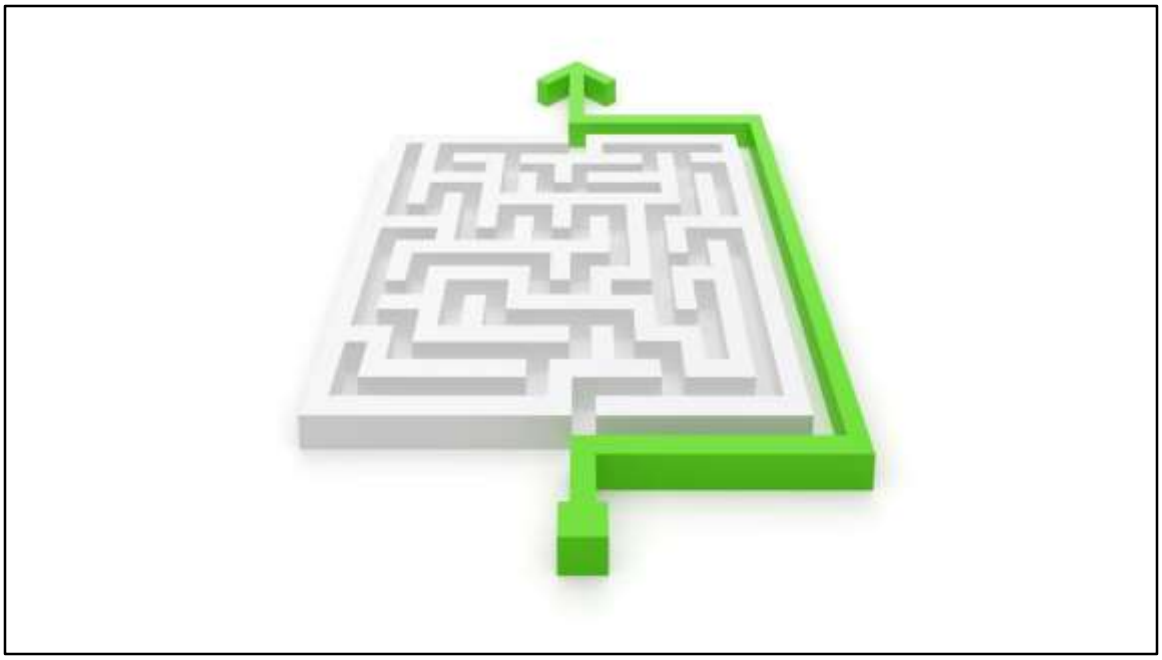
This is Linus Torvalds. Leader of the Linux project, inventor of GIT for source control.

Photograph by Ben Lowe (Creative Commons via Wikipedia)

“If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.”

As someone who deals with a very large and actively maintained code base, I figure his opinion might be worth listening to.

Photograph via winpoin.com



The advice here is to keep things simple – as simple as possible.

So I started actively working on it.

- Writing simple code;
- Simplifying the code I was maintaining.

I spotted a pattern – many times I've been able to both fix a bug and retain the desired behaviour with simpler code that was easier to maintain.

The right tooling can be very useful

Caution: Satisfying the tools doesn’t mean you have great code

If the compiler is complaining, do something about it.

Some messages might be trivial –
but good luck spotting the important message in all the noise.

Aim for zero warnings. That way you know it's fixed.

Image credit: Wikimedia

I've found a useful ally in tooling.

If you have a good rules engine available – like Visual Studio's Code Analysis – make use if it.

It's like having a co-pilot who will warn you if anything you do is obviously stupid.

Key is to customize the rule set you use;
put any overrides in the code with a justification message.
Helps to avoid a specific class of bugs.

Image credit: Alireza Shekarian at Flickr

Code Metrics can help you identify areas of your code that need attention.
Keeping metrics “healthy” as you work can guide you towards good practices.

For .NET Users, built into Visual Studio Premium or above, available since VS2008. A command line runner is available from Microsoft Research.

Image credit: Stevie Spiers Photograph at Flickr

Now lets talk about techniques for writing better code

Don’t repeat yourself.

Avoid repeating key facts – although you can’t avoid it 100%.

Know – document – which is the master

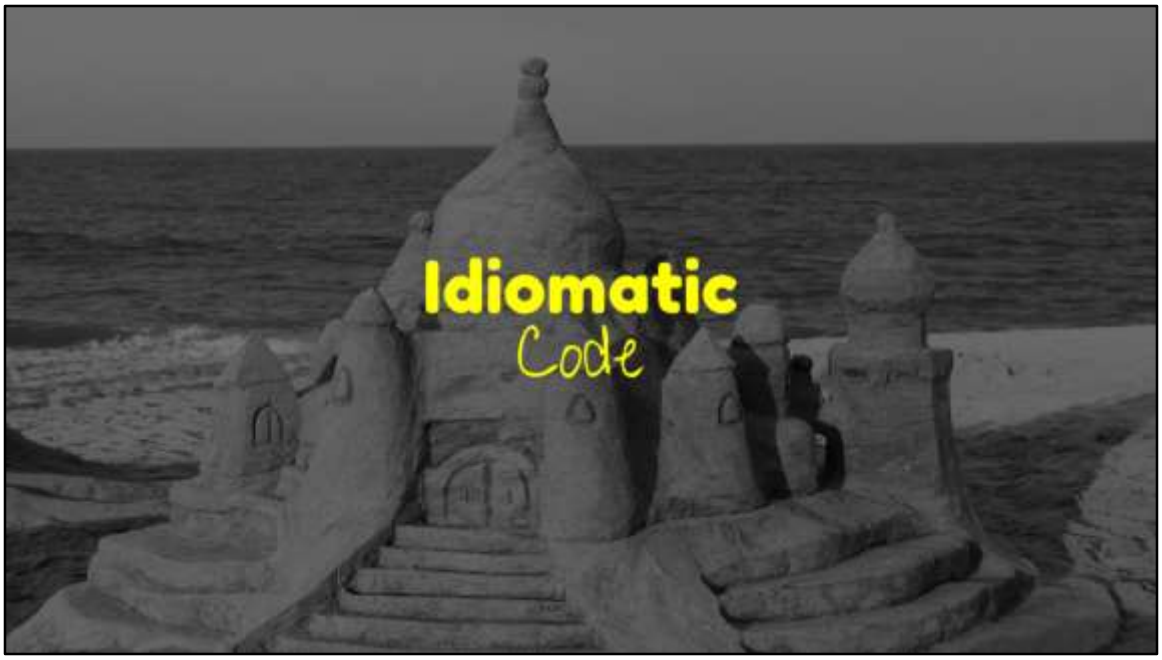
Unit tests to verify consistency

Image credit: aigle_dore @ flickr

Embrace the SOLID principles

Single Responsibility
Open-Closed
Liskov Substitution
Interface segregation
Dependency inversion

Image credit: zeldman @ flickr



Write code in the way that's expected for the language you're using

Image credit – vpickering@ Flickr

```
for (int index = 0; index < PropertyCount; index++)
{
   var p = Properties[index];
   // ...
}
```

This might be C# code, but it's been written with a Delphi mindset - a separate property that gives the count of items, and a **for** loop to iterate over them.

An experienced C# developer will look at this and either
(a) Assume you're incompetent, or
(b) Assume you're compentn and try to find the reason why the usual approach wasn't used

```
foreach (var p in Properties)
{
   // ...
}
```

This is idiomatic C# code, iterating over the collection with a **foreach**.

Use specific types for the information being passed around
E.g. FileInfo and DirectoryInfo instead of using strings

Image credit: scttw @ Flickr

```
public void SaveAsHtml(string destination)
{
    // ...
}
```

By accepting a string parameter,
the method can be passed any string at all,
not just ones that represent file paths.

```
public void SaveAsHtml(FileInfo destination)
{
   // ...
}
```

By instead using a FileInfo, only the right kind of thing can be passed.

Not limited to built in types – create your own

E.g. “SeriesId” used to identify a series

If you have a pair or triple (or worse) of values that are consistently carried together, you probably have a buried object that should be extracted

```
public bool FindIssue(
    int issueId,
    out string summary,
    out string details,
    out IssueSeverity severity)
{
    // ...
}
```

Imagine you found this in your application

```
public Issue FindIssue(int issueId)
{
   // ...
}
```

By instead returning an issue object, the code is clearer
Number of parameters is down
You can add a new field without having to change the code very much

Keeping track of related information in parallel lists is a code smell.

Image credit: paloetic @ Flickr

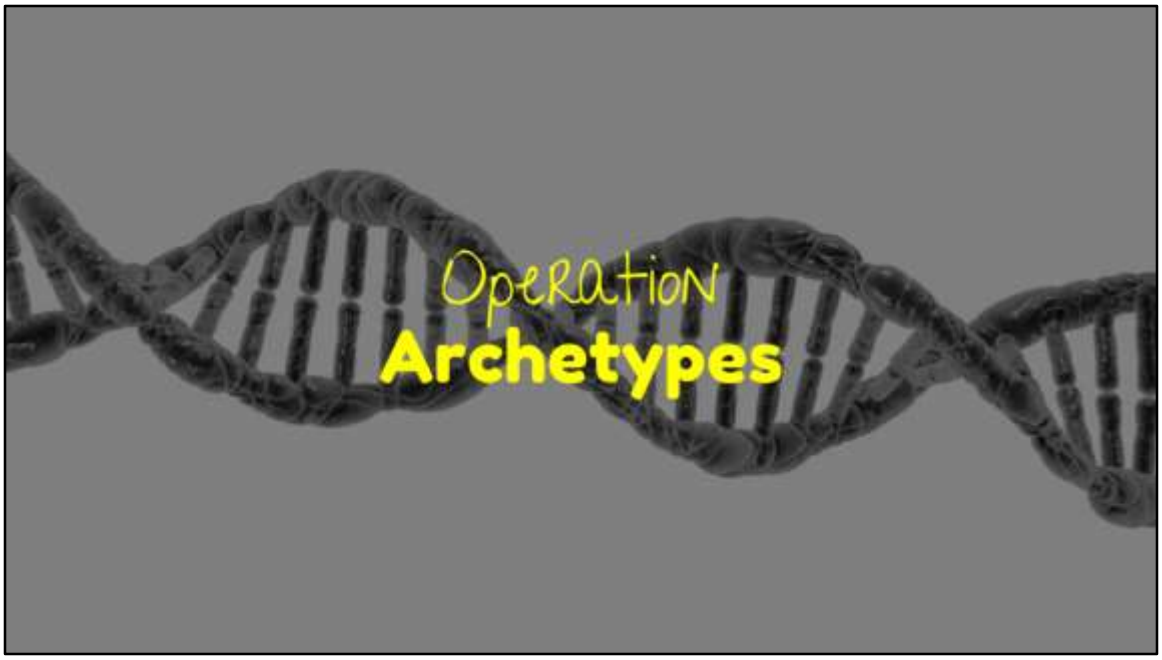

Person
Gift
Mum
DVD
Dad
Biography
Suzie
XBox Game
Bob
Movie Tickets
Fido
Chew Toy
Nana
Toiletries
Granddad
Joke book
Uncle Jim
Jim Beam
Uncle Tony
Sopranos
Aunt Mabel
Baking

Immutable types can’t be changed after they’ve been constructed.

Performance can be better than mutable types – due to shared content and caching.

Reasoning about your application can be much simpler

Image credit: digitalmindphotography @ Flickr



Keep each method simple and predictable
Makes your code easier to reason about

Image credit: Pixabay

Keep the purpose of each method simple.

Queries – obtain information; don't change state; repeatable, cachable.

Commands – make changes; often void, but sometimes return value(s).

Orchestrations – coordinate the functions of the two.

Don't make a method that is a mix of two or more types.

```
if (FileExists(inputFile))
{
   // ...
}
```

Imagine you read this code – what do you expect?

```
public bool FileExists(FileInfo file)
{
   if (file.Length == 0)
   {
      file.Delete();
      return false;
   }

   return file.Exists();
}
```

Here's an implementation – recreated from real life – that does something unexpected.

Now lets talk about techniques for writing better code

Get your code reviewed by a peer

Goal is to identify ways to improve the code, so don’t wait until finished

Good reviewing is a learned skill

Image credit: vfsdigitaldesign @ flickr

Here's a trick I've stolen from Adam Codgan - before you mark a feature as complete or a bug as fixed, record a screencast demonstration.

This serves to demonstrate completion, and will usually flush out anything you've forgotten.

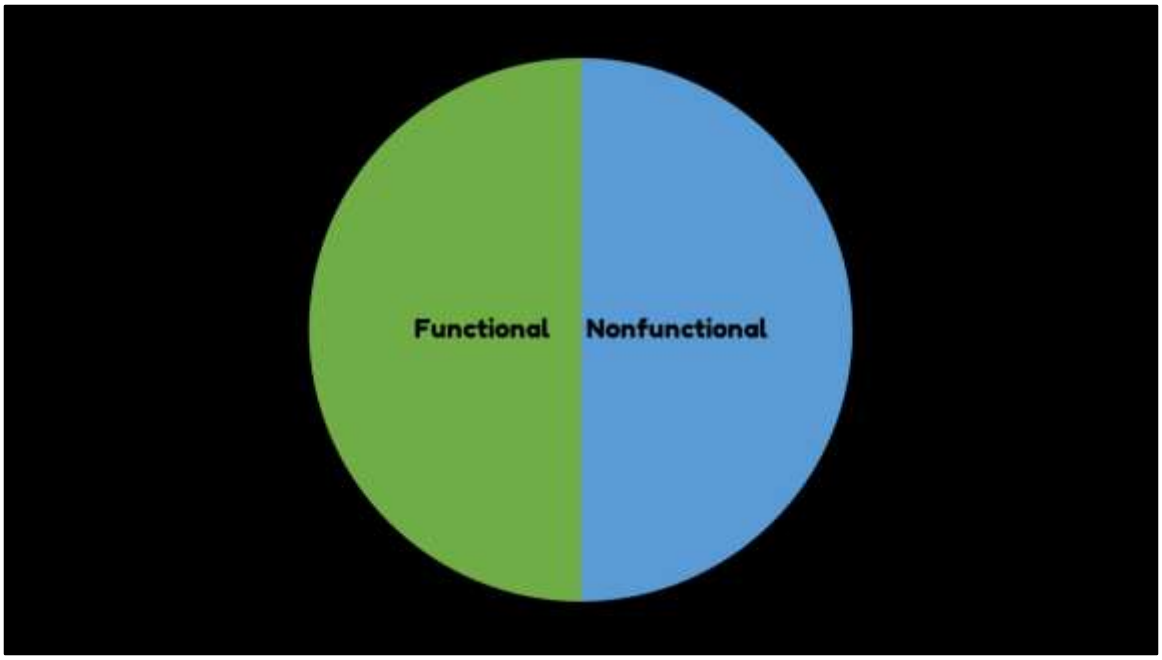
One last suggestion.

Image Credit: AV Hire London @ Flickr

We're writing code to create an asset – a piece of software that provides business value.

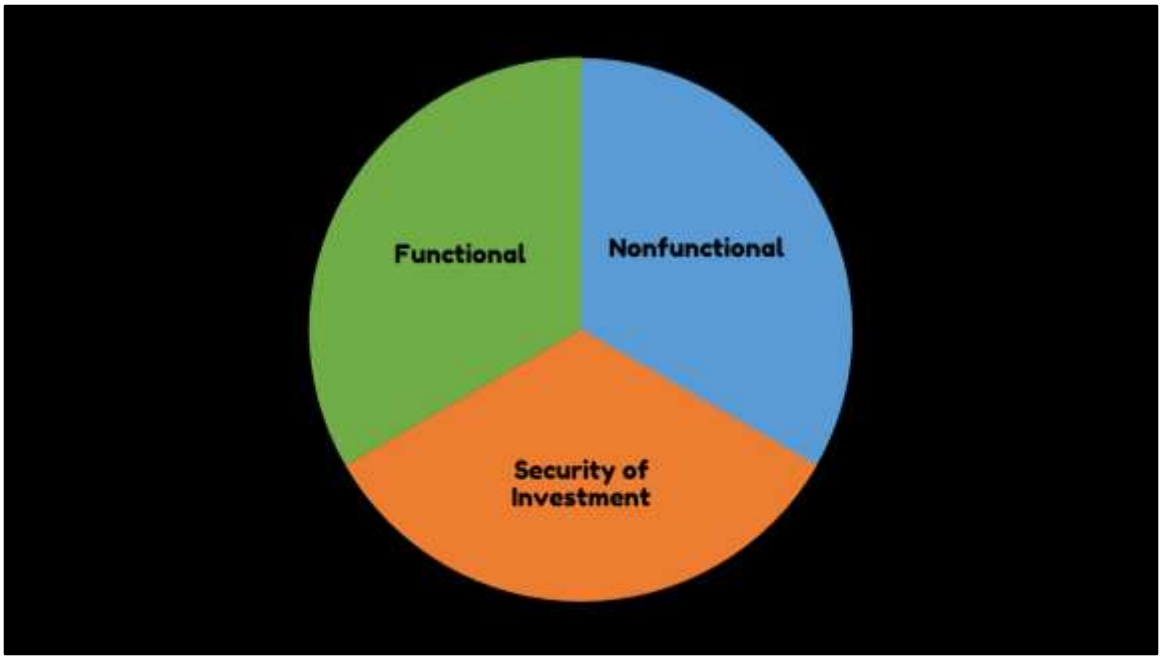
The value of that asset won't disappear at the end of the project
And the value of that asset should increase over time, not decrease
---
Here's one way to think about it.

Image credit: barnyz @ Flickr



So many times we divide the requirements of our system into two groups

Functional requirements – what the software should do
Nonfunctional requirements – Other issues, like performance and scale
---
But I'm convinced this is incomplete, that we end up focusing entirely on the needs of the project, and not on the long term.



We also need to take into account “Security of Investment”

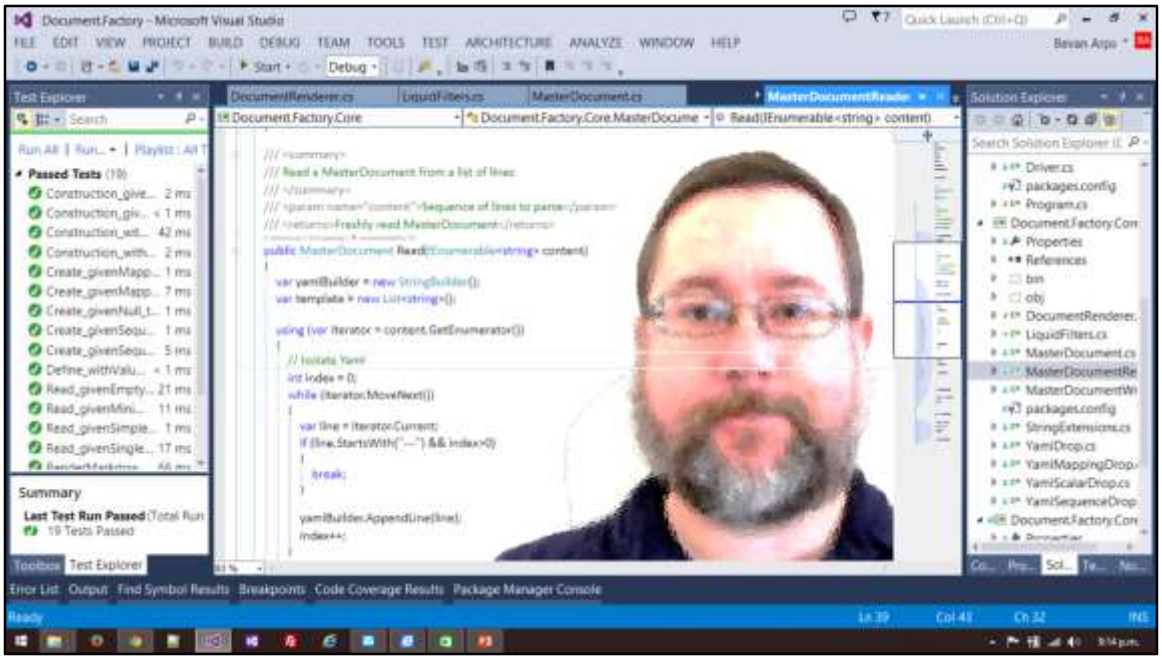
This extends our thinking past the end of the project, forcing us to consider things like

- Maintainability – how easy is it to make changes
- Extensibility – how easy is it to extend
- Understandability – how easy is it for someone else to understand it
- Stability – how well will it run if left uninterrupted

Everything we've looked at to this point is self determined.
But, remember that our self view can be distorted.
A mentor – or simply a good friend – can provide useful perspective.

This is Phil Haack.
He works at Github.
His job title is Windows Badass.
He spends his day doing whatever he can to make it more awesome, especially for windows users.

Phil Haack loves to code.
He really does.
He gave a keynote at the very first Code Mania conference in 2012 with the title “I fucking love to code”.



And you know what?
I love to code as well.

Questions?

Image: Flickr user omcoc

Thanks for your time, I hope you got something useful.