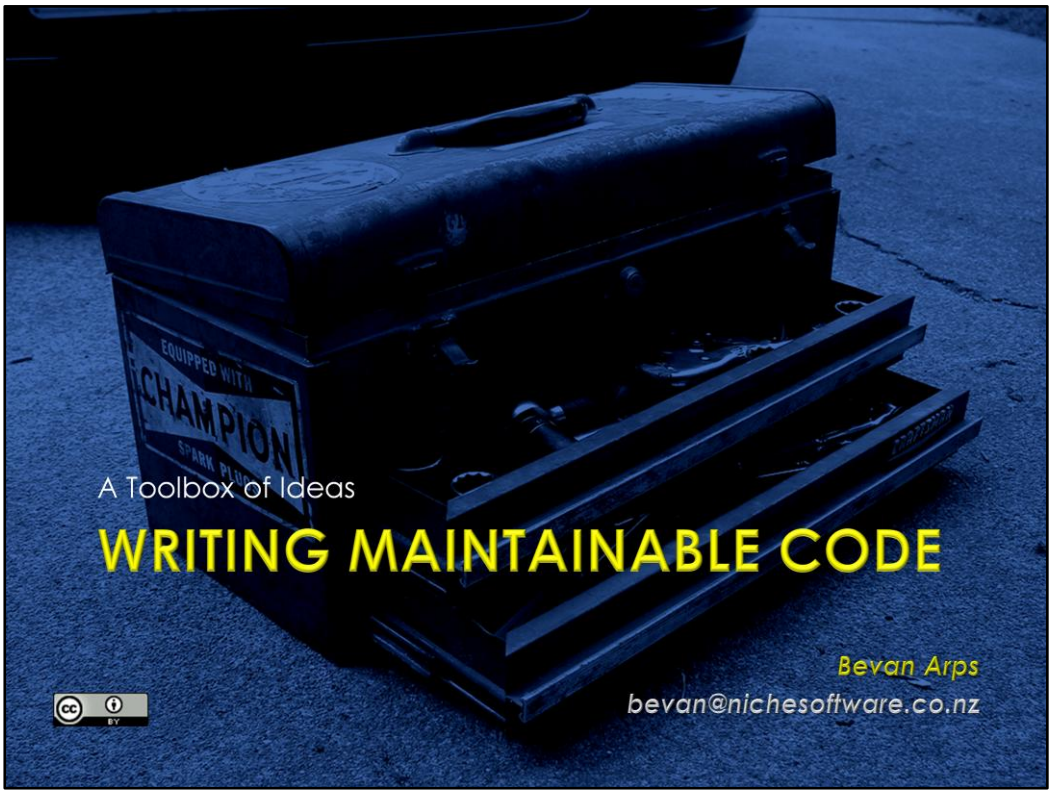
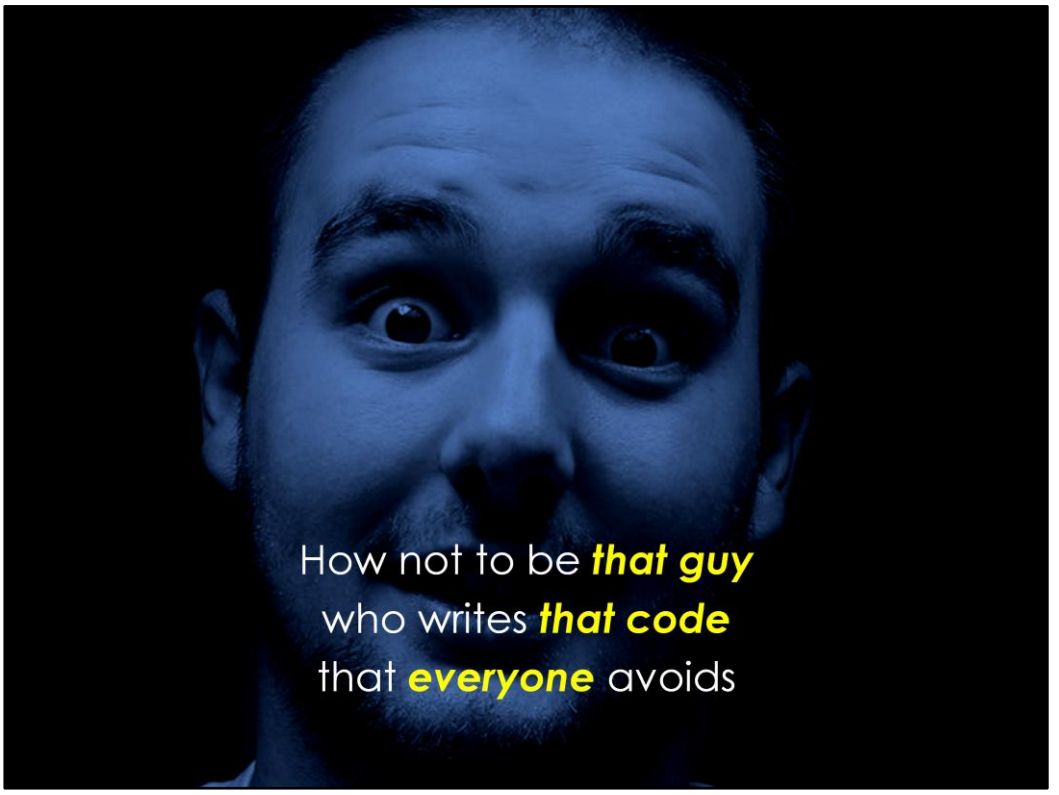


Much of the content is generic and could apply to any platform

Presenting examples in a .NET environment, and where a language is necessary, C#

Focus on Code, not on other things; not that other things are irrelevant, but need to focus somewhere.

Also not touching on Governance stuff – selecting platform, vendor etc

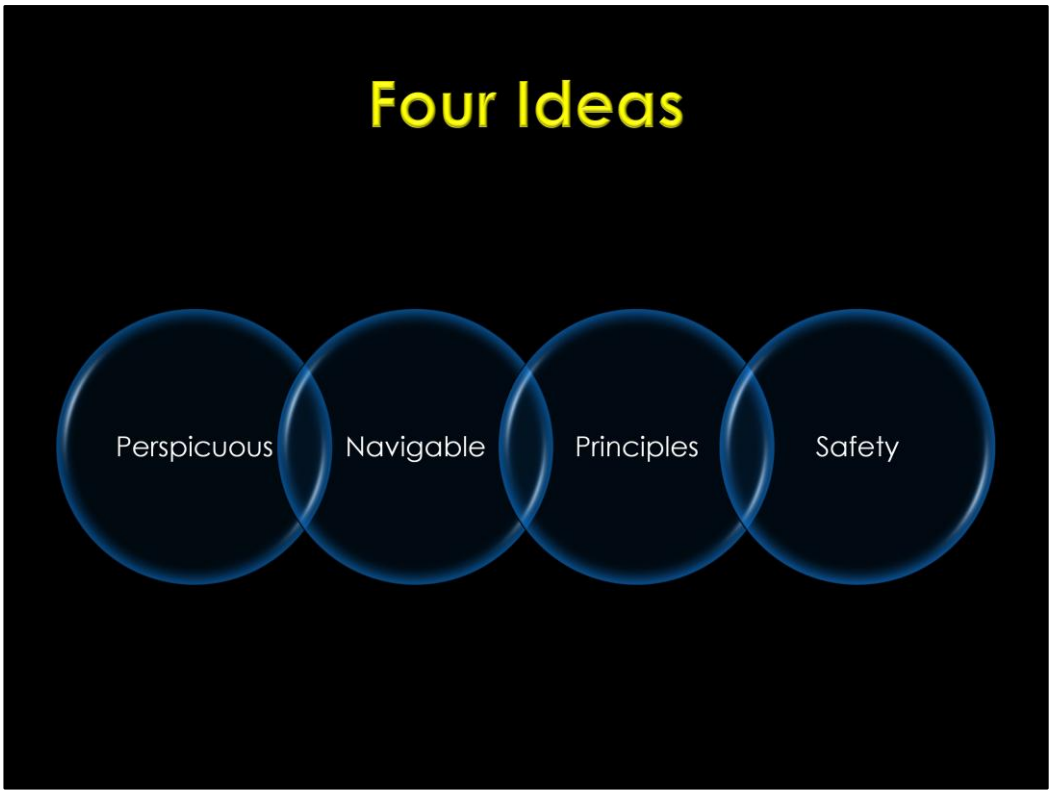


Ever worked on a large code base
where there was an area marked
“here be dragons”?

Or, worse, an area that should be signposted ...
... but isn’t.

It seems that many large codebases have these areas,
swamps that promise to trap the unwary developer

Unfortunately, these always seem to be critical to the system



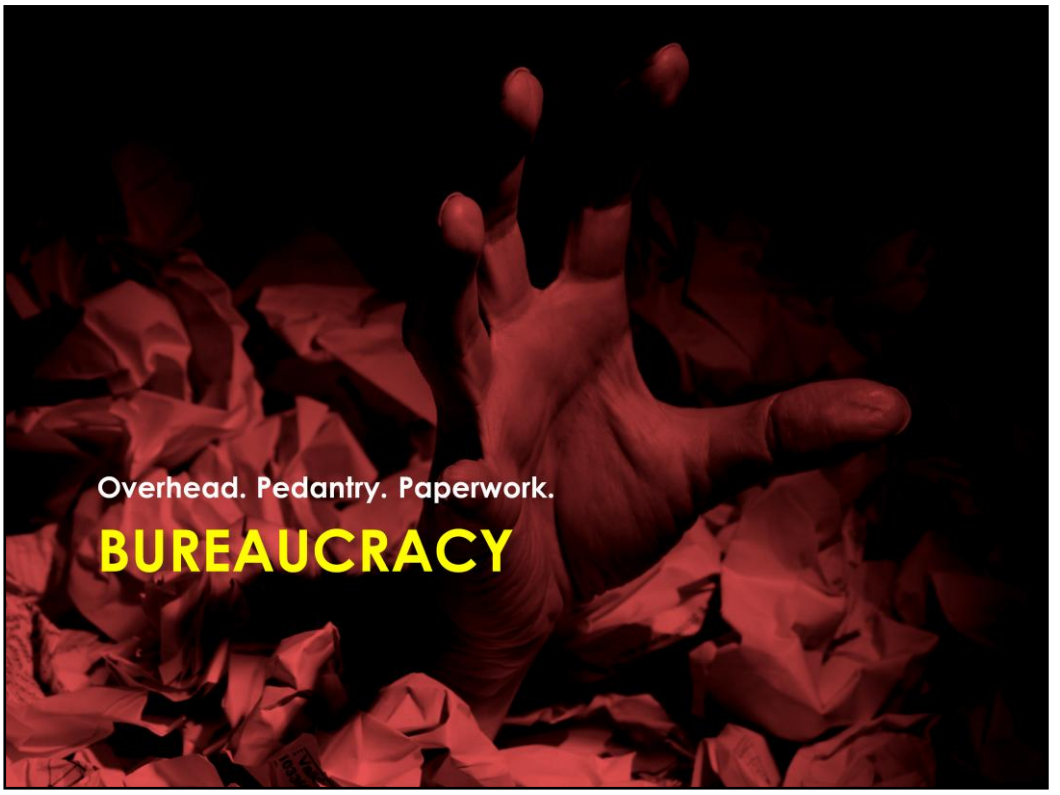
Four Key Ideas

Perspicuous
Clarity of communication

Navigable
Easy to work out where to go, and easy to get there

Principled
Work with known principles

Safe
No boobytraps, sinkholes or mazes



Focus is on Value
not on Obstruction

Keep your eye on the goal:
Stable, Reliable, Maintainable, Valuable systems

If a practice or convention is delivering value,
embrace it
If not, change it

The ideas I'm talking about here are
not about red tape or getting in the way
It's about value.

Choose processes that work for you,
for your team, in your business.

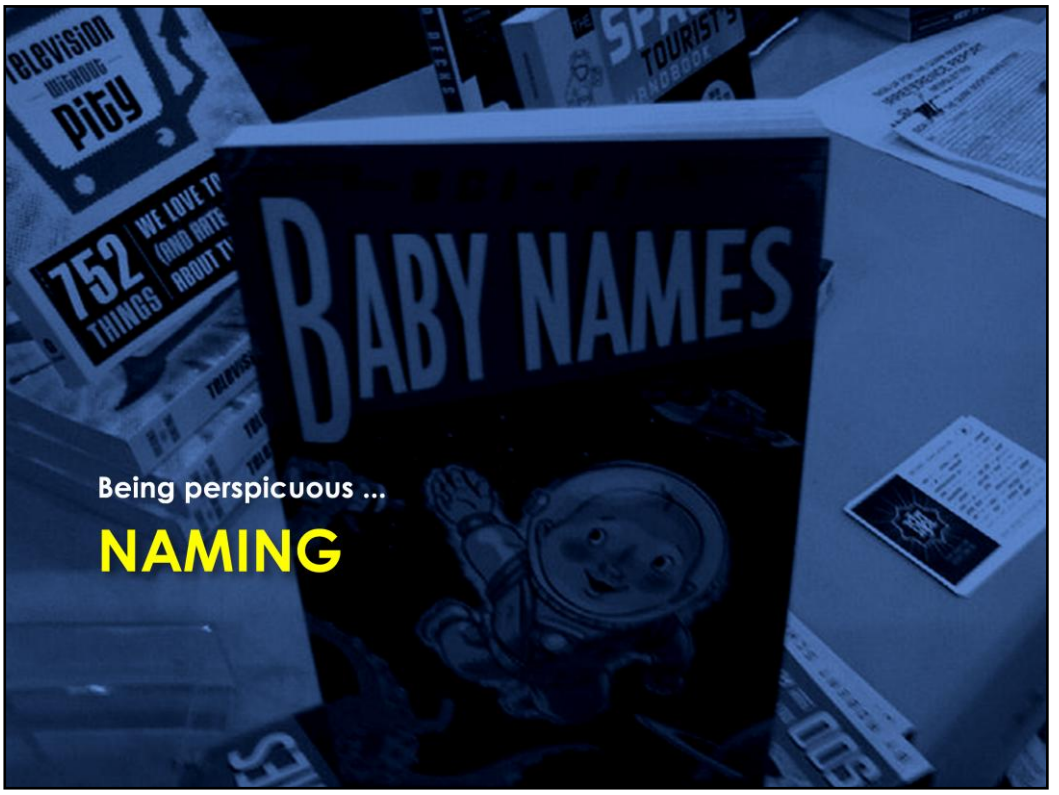
Change one thing at a time
Keep What works
Change what doesn't

Clarity
Intelligibility

Useful to have Standards & Conventions

All about clear communication with other humans
Including people who aren't developers
System Administrators, DBAs, Help Desk, Trainers, and yes, Users too

Concise
Don't want to overload people with
too much
or repetitive



Naming of types, of methods, of members, of locals.

Naming is **important**.
Clarity, Accuracy, Intelligibility and Reliability

Names are the first aspect encountered by other developers,
may be the most persistent and long lived aspect,
will be present even if everything else is lost

Names need to be clear and accurate
(singular vs plural).

Trustworthy
Appropriate to Culture – eg "C" prefix in MFC

Demo: Naming.cs

Photo Credit: http://www.flickr.com/photos/20976676@N00/2561333250/

Commentary: We call talk about commenting our code,
but often the comments are rubbish.

Anyone seen comments that were just plain wrong?
Comments should be intentional, not declarative.

Don't Repeat Yourself
comments shouldn't indicate what is going on, but why we need to do it.

Don't just write things that can be worked out by reading the code:
give more information, things that otherwise would be guessed at.

Include references – to documents, websites –
so that others can learn what you know.

If something might need to be improved,
leave reminders to later self.

If an approach doesn't work,
leave notes so the next person doesn't repeat the mistake

Not everything needs commenting – good naming helps a lot.

Demo: Comments.cs

Image Credit: http://www.flickr.com/photos/wfiupublicradio/4398924160/

Make it save to Explore

Don’t let users get lost in your code

Provide maps of your code

Make it easy to find out where to go
And easy to get there

Image Credit: http://www.flickr.com/photos/julianbleecker/3077913602/

Everything has a place
and everything in its place

This helps both writers and maintainers

When adding functionality to the system: you know where it should go

When you're working on a fix: You know where to look for the problem

Design Patterns
Many of the popular design patterns – MVC MVP MVVM
– are ways of being structured

Structure is the best way to handle a large code base
If you have 600k lines of code (or more)
You don't have time to read the whole thing
Structure allows you to concentrate your efforts

Take a shortcut
and put code in the wrong place
You're sabotaging future maintenance
incurring technical debt

Demo: Layers.png

Image Credit: http://www.flickr.com/photos/takashi/18862634/

How do you know where to put things?

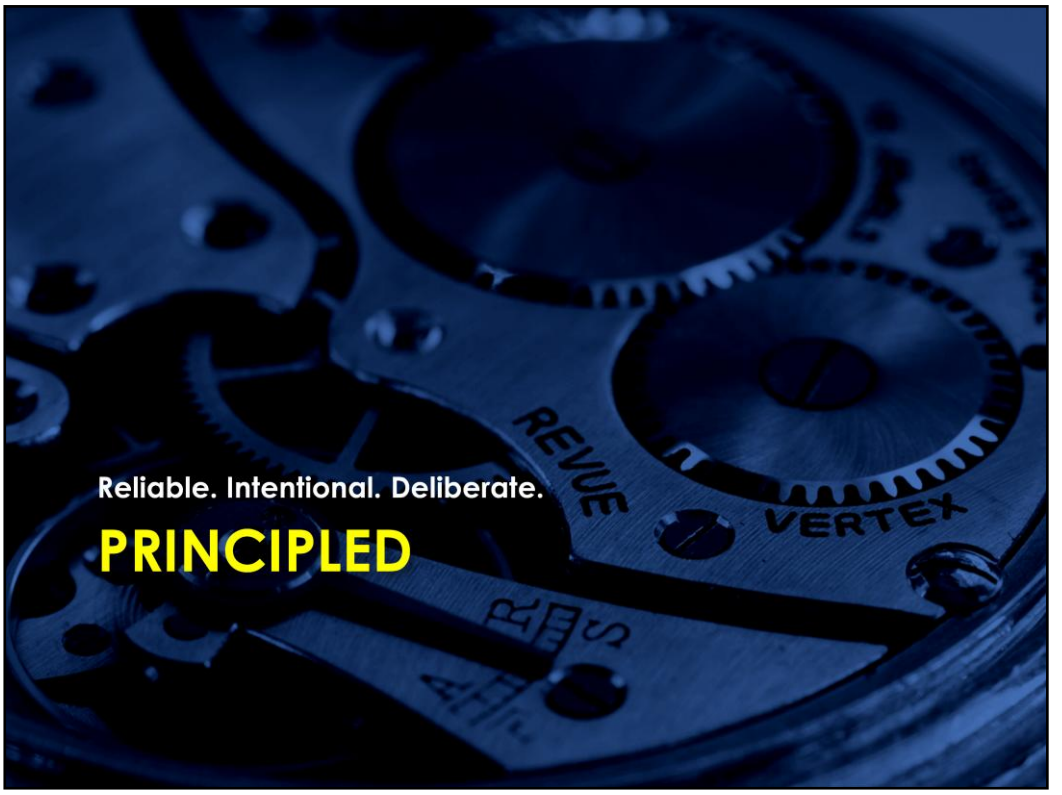
You need maps to tell you where to go
and how to get there

Use Class Diagrams in Visual Studio
(Much underused and valuable)

Draw diagrams
in Visio
or Enterprise Architect
or Rational Rose
or with Pencil and Paper

Don't leave it to *oral tradition* to pass key knowledge

Demos: Class diagrams in /doc/

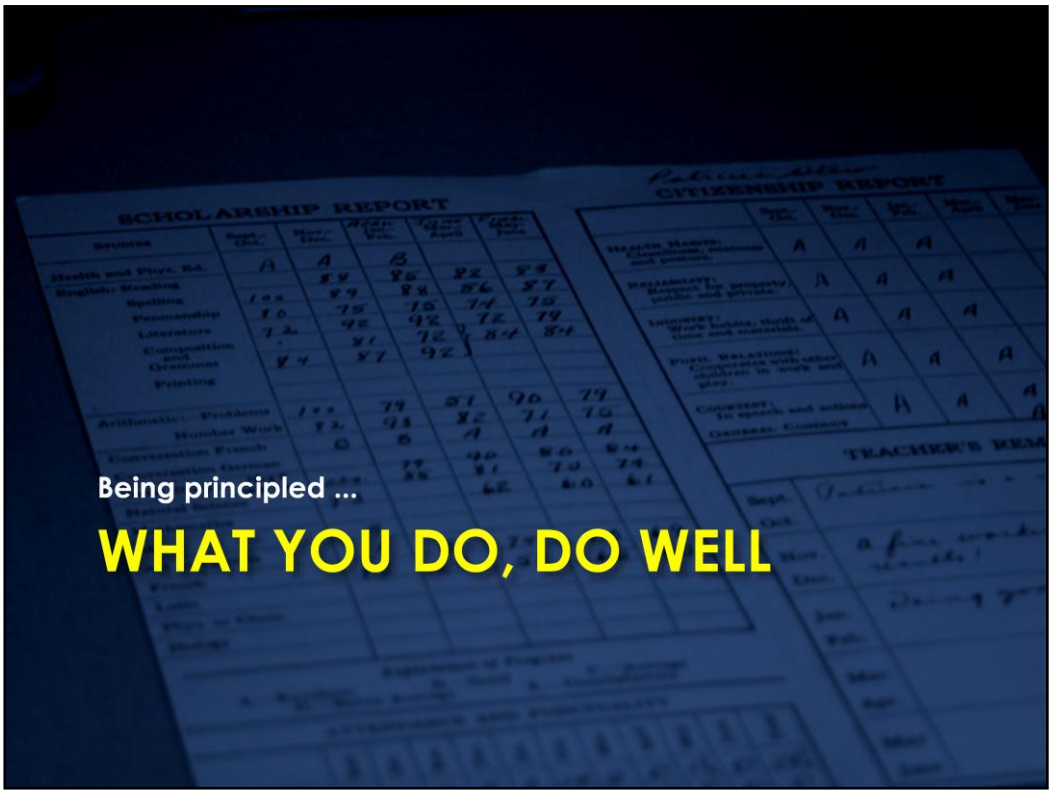


Principled Development

Know what you are doing, and why you are doing it
Share your principles with others

Even if they don't agree, they'll be able to predict how you work

Always be willing to give a defence of why
you are doing things a particular way.



What you do, do well

We've all heard about "Best Practice"
Sometimes difficult to define "best" – religious arguments

Instead, focus on doing what you do well
"Well" changes slowly – not following the latest hype or fashion
– a commitment to quality

Commit to doing things as well as you know how –
if you are going to cut corners, know which corners you're cutting and why.
Know what you might need to clean up later (write it down!)

Learn from other peoples examples, and always work on improving yourself.

Tension:
need to balance between fossilisation on one extreme
and chaotic inconsistency on the other

Make changes, but on purpose, not by accident
Change things slowly and deliberately

Peer reviews useful

Image Credit: http://www.flickr.com/photos/jessflickr/183287090/

Don't Work Too Hard

Don't reinvent the wheel

Learn the framework
take the time to learn and explore

Use the power of the framework
don't work harder than you need to.

Not only do you waste your own time, but you end up
writing more code that someone else needs to understand.

Demo: Working.cs

Image Credit: http://www.flickr.com/photos/slworking/651075071/

Perhaps the major difference between an amateur and a professional developer
is not in how well the code runs
but how well code *fails*

Will your code generate incorrect results on dodgy data?

Or will it flag the data as incorrect
so that your users don't make wrong decisions?

When something goes wrong, your code should tell you how!
(But without revealing any secrets to your users)

Diagnostics should give more than the end error
Include the context as well

Log files
Event Logs
Notifications – by email

Demo: FailWell.cs

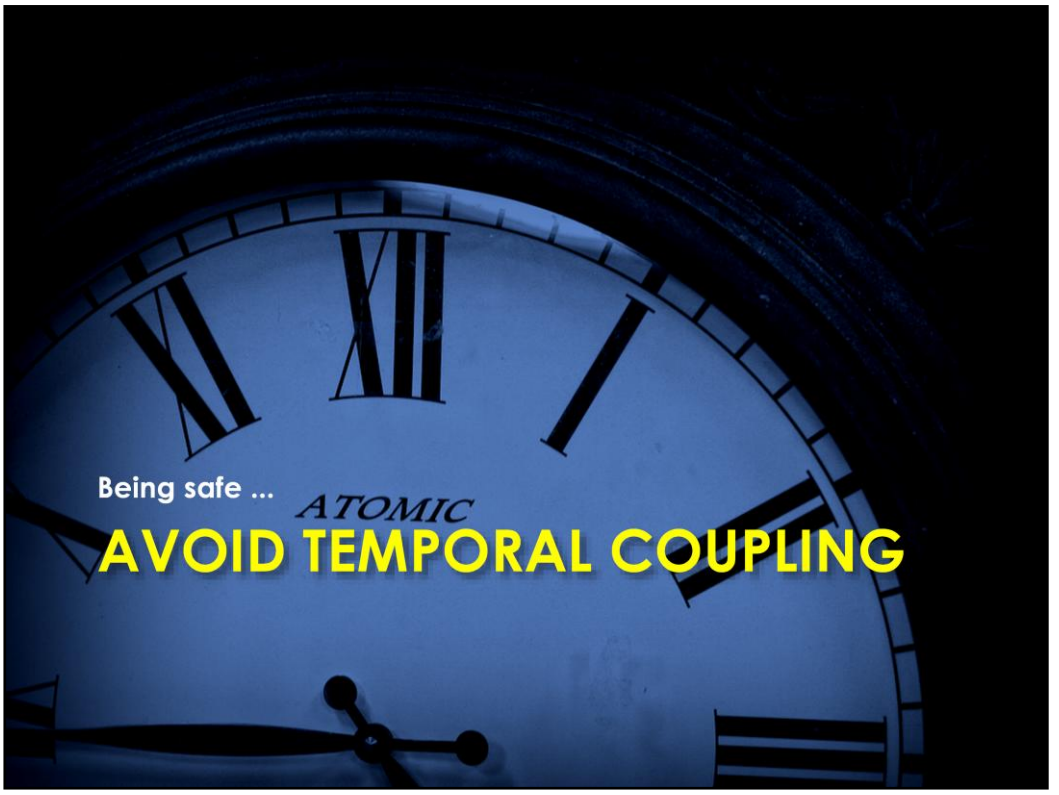
Image Credit: http://www.flickr.com/photos/cjdaniel/3312922051/

Make it safe for someone to work on your code.

Don’t let them get hurt.

Don’t write boobytrap code

Image Credit: http://www.flickr.com/photos/pong/288491653/



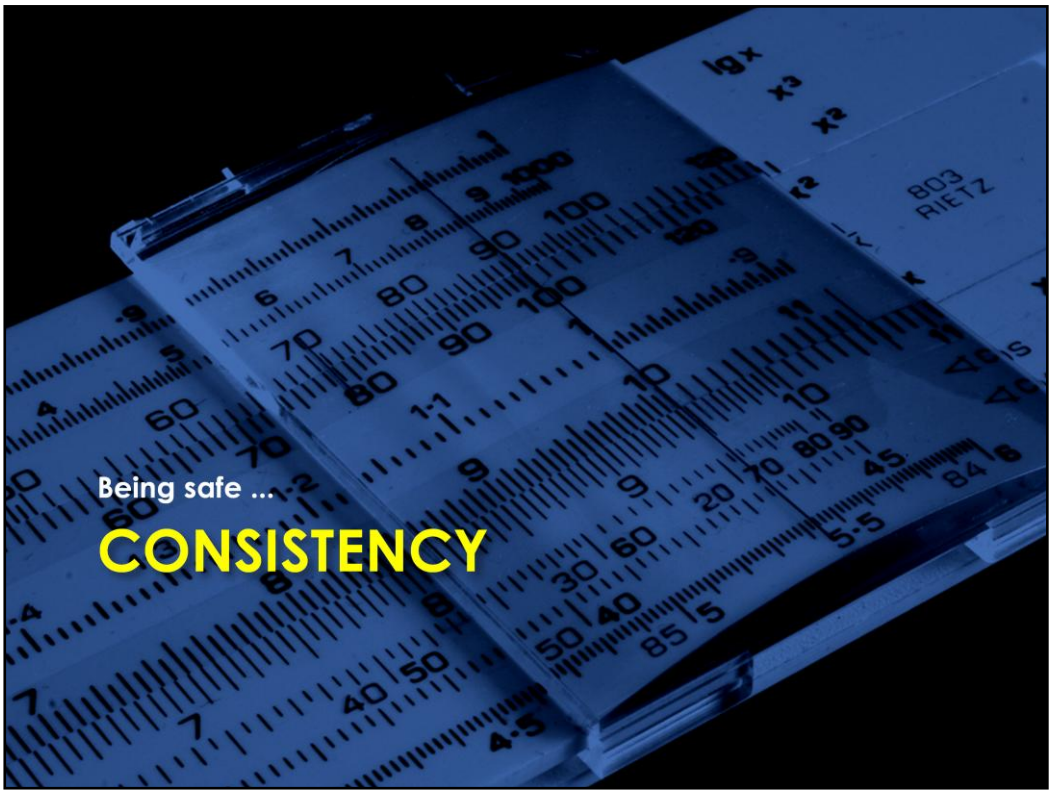
Avoid Temporal Coupling

Elements of Responsibility
Who is responsible for ensuring that things are performed in the right order
The object should take responsibility for itself, not be reliant on others

Single Responsibility Principle
Object Oriented vs Procedural Development
Code Smell: Object Envy

Demo: TemporalCoupling.cs

Image Credit: http://www.flickr.com/photos/nooccar/4667918541/



Keep things Consistent
Consistency builds trust; be careful not to violate that trust

Compiler Warnings
Use the compiler to your advantage
Target zero warnings – use Warnings as Errors
Use other tools to identify and report on potential errors

Coding conventions
Doing things the best we know how

Image Credit: http://www.flickr.com/photos/dominicspics/3915904999/


ANY QUESTIONS?


bevan@nichesoftware.co.nz
http://www.nichesoftware.co.nz
THANKS