Testing Mobile Device Applications With .Net Compact Framework 2.0

Steve Love

steve.love@arventech.com

Testing Applications With .Net Compact Framework 2.0

Steve Love

steve.love@arventech.com

Testing Applications With .Net

Steve Love

steve.love@arventech.com

Testing Applications

Steve Love

steve.love@arventech.com

Applications

Steve Love

steve.love@arventech.com

Software

Steve Love

steve.love@arventech.com

Testing Mobile Device Applications With .Net Compact Framework 2.0

Steve Love

steve.love@arventech.com

The Mobile Challenge

- Mobile Devices Present Unusual Challenges
- Lack of Development Tools
  - Debuggers
  - Automated Test Harnesses
  - Stack Tracing
- Available Tools Less Reliable
- Variety of devices

Who's Tough?

- Windows Programmers
  - `LPCTSTR lpszString;`
- Unix Shell
  - `sed 's/^\(.*\)l3?t[ \t]*h[4a]x0r5\([ \t]*.*\)$/$1nubbin$2/g' < 'cat foo | tail'” > /dev/null`
- COM Programmers
- VB Programmers
- Embedded Systems Developers
- Game Programmers

So Why Are We Here?

Testing isn't easy

Development for mobile devices doesn't make it easier

So....

The Grass IS *Greener*

- Don't Test On A Mobile Device
  - Tools exists but are unsatisfactory
    - NUnit Compact Framework Bridge
    - Microsoft Team System
- Don't Code On A Mobile Device
  - Who does?
  - Anyone get Visual Studio 2005 to run on PocketPC yet?
- Portable Code is Testable Code
- Contain Non-portable Parts

Whats And What Nots

.Net Compact Framework is...

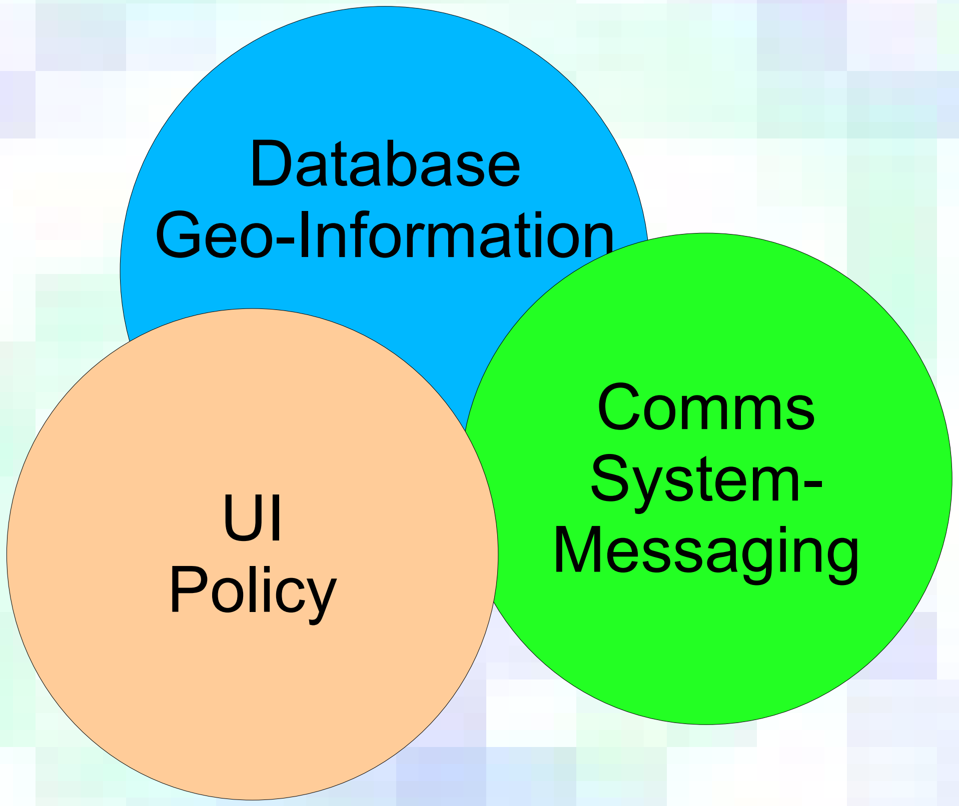
- Almost indistinguishable from full .Net, but ...
- a bit smaller, and...
- just as loveable!

OK on to the secrets...

Top Secret

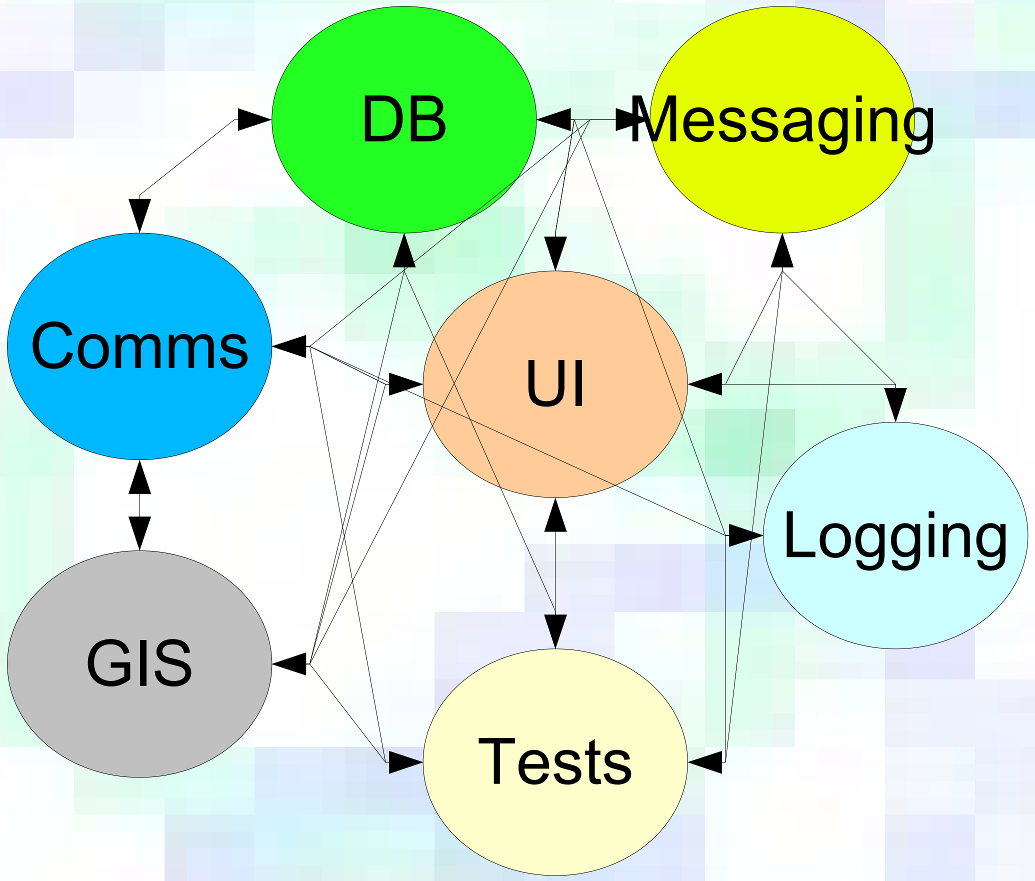
Well not really...

The Usual Suspects



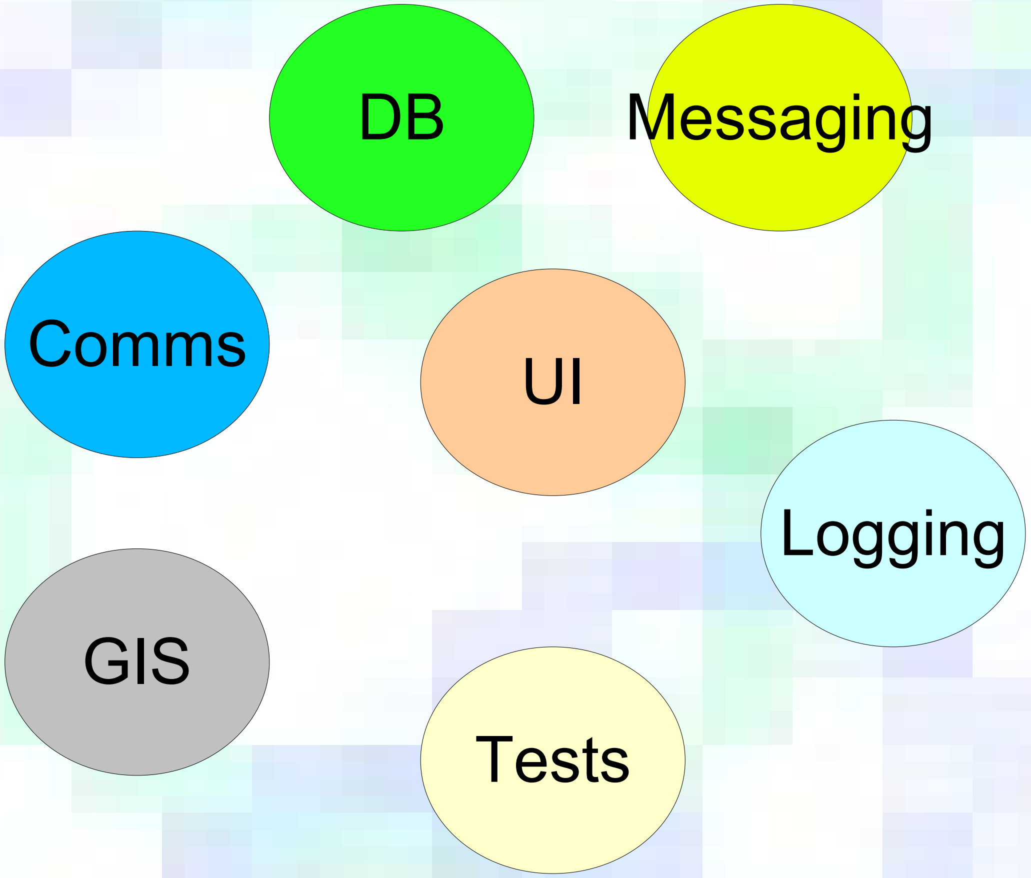
Uncohesive systems mix responsibilities.
AIM HIGH!

The Usual Suspects



Highly coupled systems
lead to impossible
dependencies.
AIM LOW!

The Usual Suspects



But not too low!

Abstract Thinking

Interfaces are (by definition) Abstract

Deciding on *what* the abstraction should be is hard

Think in terms of
what you want to do
not
how it gets done

The Bug Factory

THERE'S NO BUGS IN NO CODE

THE EARLY BIRD GETS THE BUG

Design for Test

The evil

The bad

The ugly

```
public class Untestable
{
    public
    Untestable()
    {
        Error.Log.Instance.Write(
            "Starting" );

        store = new DataStore();
    }
    private
    bool IsValidCode( string code )
    {
        return code.StartsWith( "C" )
            && code.Length < 5;
    }
    private DataStore store;
}
```

Design for Test

```
public class MoreTestable
{
    public
    MoreTestable(
        Ilogger log )
    {
        this.log = log;
        log.Write( "Starting" );
    }
// ...
    private Ilogger log;
}
```

Q. Why isn't this still a Singleton?

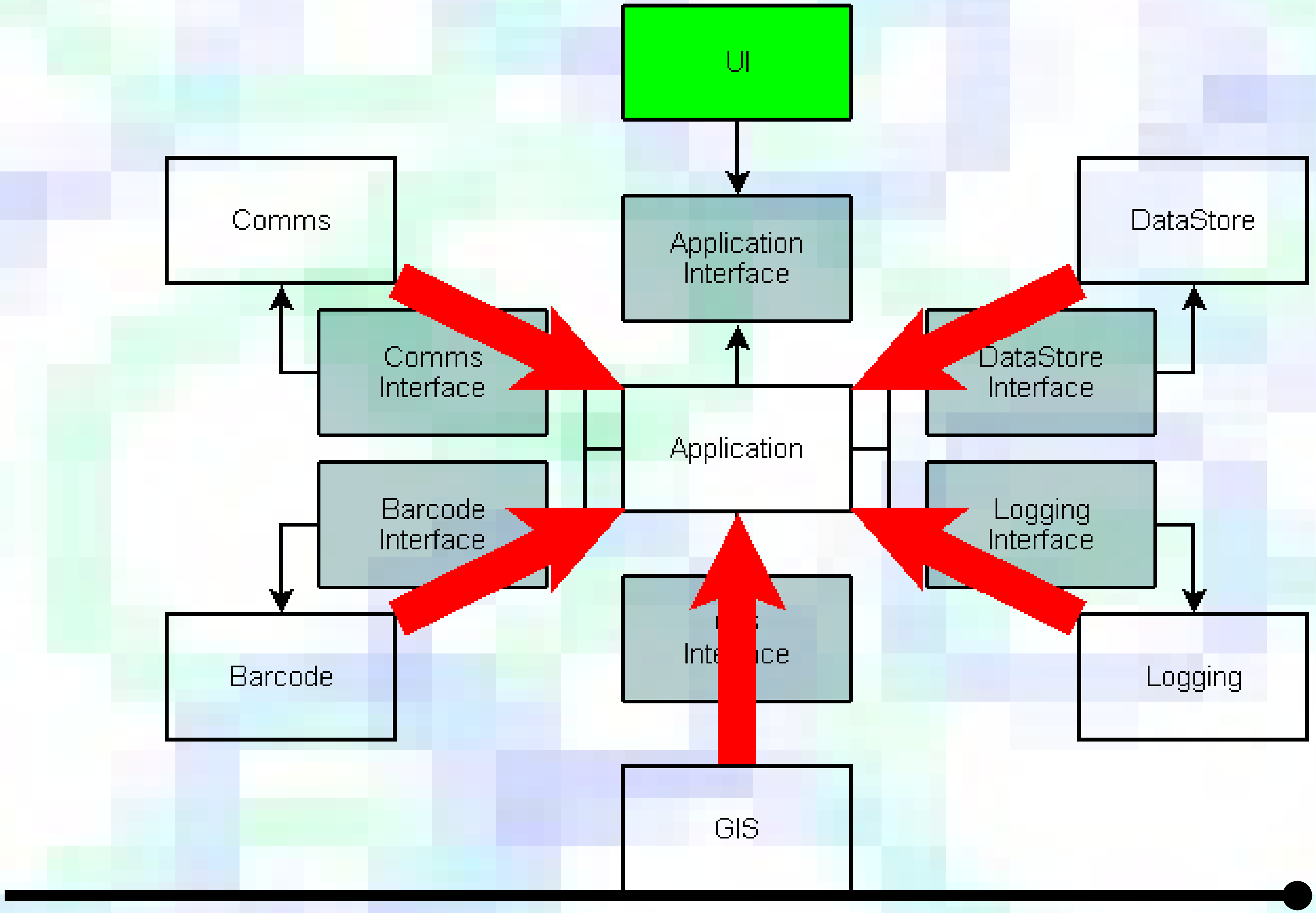
A. It is – in spirit.

Design for Test

- Interface Based Programming
- Dependency Injection/Inversion
- Parameterise From Above/Without

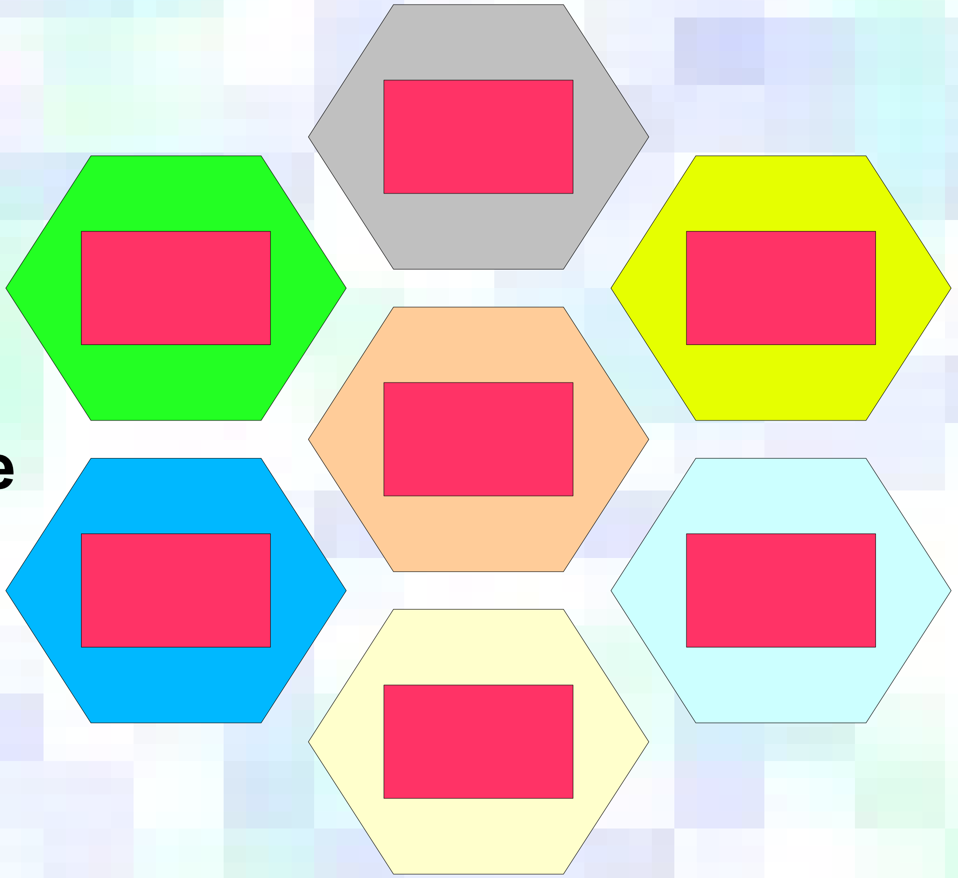
```
public class Testable
{
    public
    Testable(
        ILogger log,
        IDataStore store )
    {
        this.log = log;
        this.store = store;
        log.Write( "Starting" );
    }
// ...
    private ILogger log;
    private IDataStore store;
}
```

From Without



Ports and Adapters

The Hexagonal Architecture



Alistair Cockburn
http://alistair.cockburn.us/index.php/Hexagonal_architecture

Dependency Inversion

“High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions”

Robert Martin
http://www.objectmentor.com/resources/articles/dip.pdf

Ports and Adapters

- Ports are the methods of an API
  - Active
  - Passive
- Adapters plug in to ports
  - The UI is just an adapter
  - Active adapters may be test fixtures
  - Passive adapters may be Mock Objects

And So To Testability

- The Alternative To A Fake:
  - Multiple Development and Release Environments can be hard to maintain
  - The “Real Thing” is often too slow for meaningful testing
  - The “Real Thing” might not even be available!

Mocks Aren't Just For Testing

Truth In Design

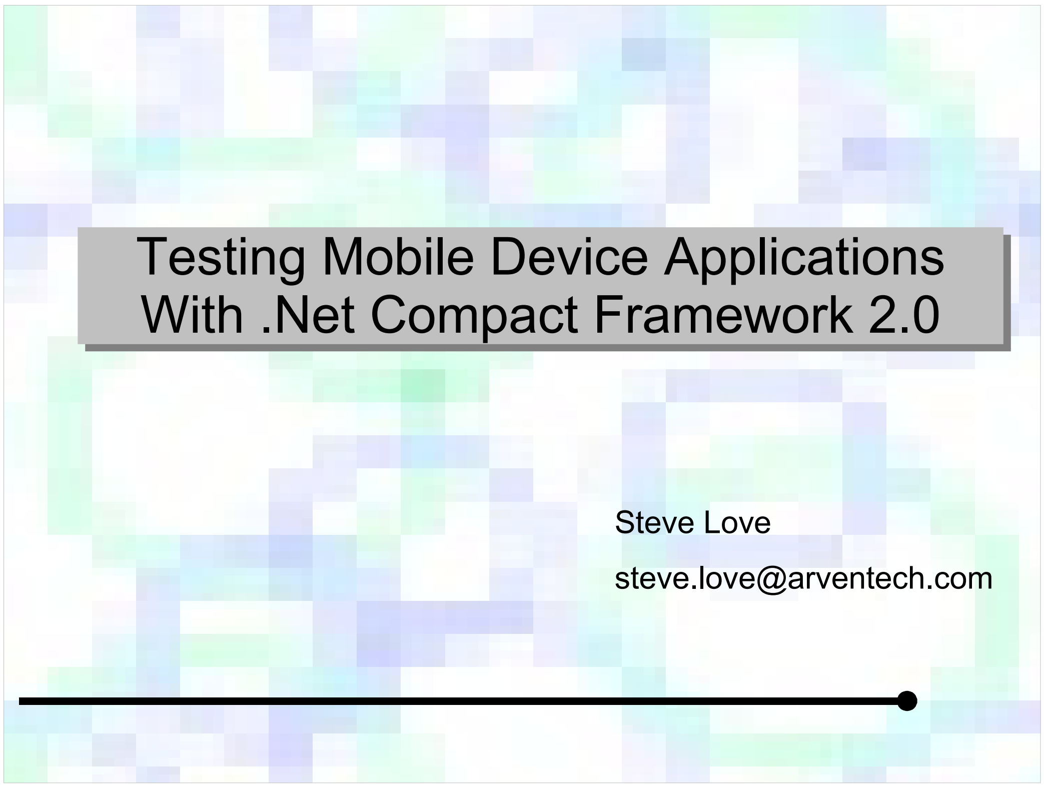
- Testable code leads to bugs being caught early, thence better software
- Well-designed code leads to more robust components, thence better software
- Designing for Test leads to better designs
- Designing for Test leads to testable code
- Repeat from step 1

Testing Mobile Device Applications With .Net Compact Framework 2.0

Thanks for listening

Steve Love

steve.love@arventech.com



Testing Mobile Device Applications With .Net Compact Framework 2.0

Steve Love

steve.love@arventech.com

This is a talk about “testing”. That these principles are applicable to testing code intended to be run on a mobile device is really a bit of a coincidence. The point is not to test code on a mobile device, but to test it on your development PC.

Testing Applications With .Net Compact Framework 2.0

Steve Love

steve.love@arventech.com

Testing Applications With .Net

Steve Love
steve.love@arventech.com

Testing Applications

Steve Love

steve.love@arventech.com

Applications

Steve Love

steve.love@arventech.com

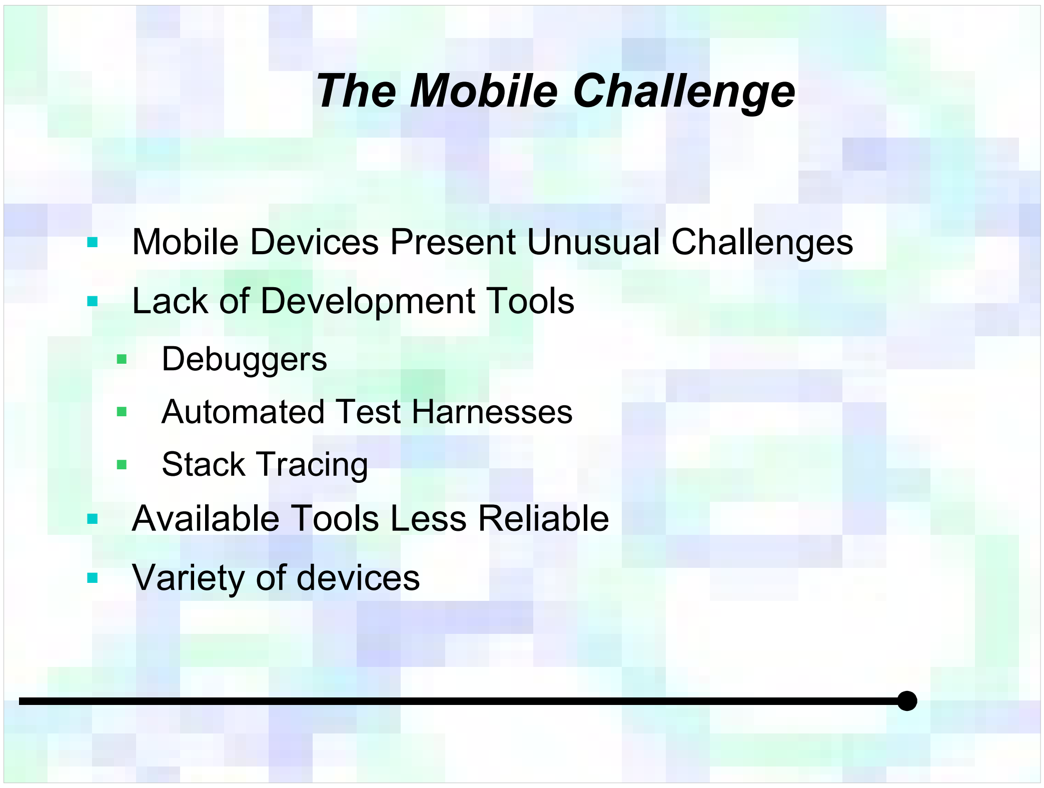
Software

Steve Love

steve.love@arventech.com

Testing Mobile Device Applications With .Net Compact Framework 2.0

Steve Love
steve.love@arventech.com

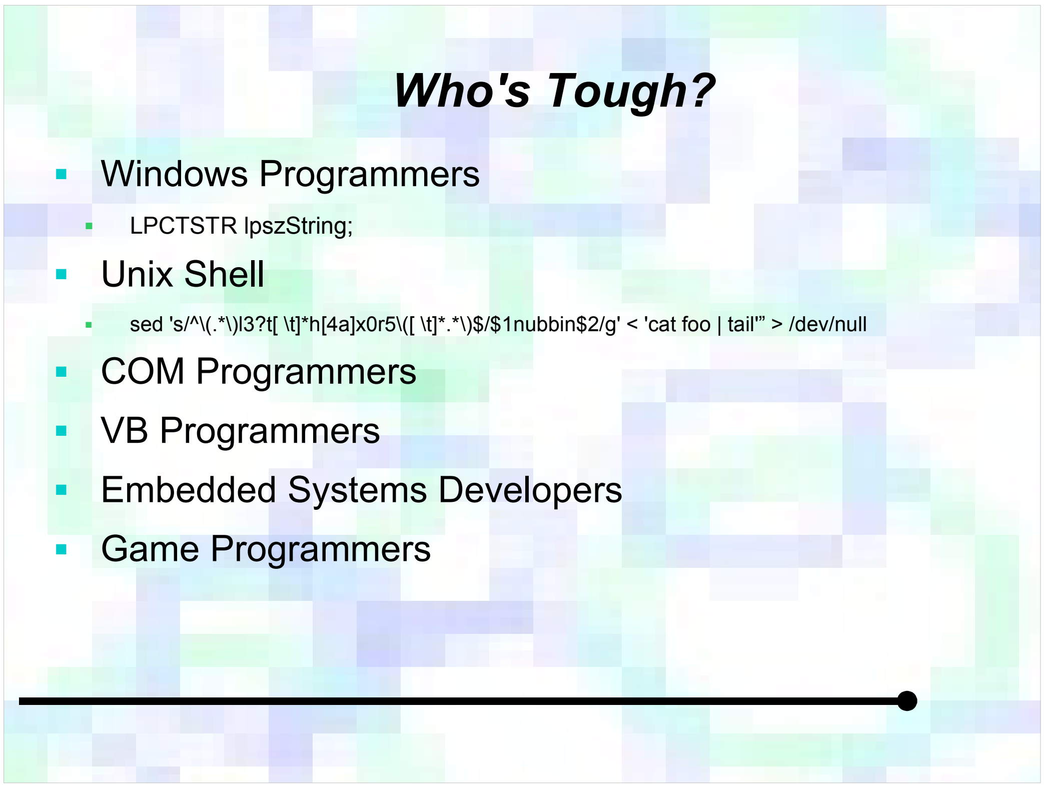


The Mobile Challenge

- Mobile Devices Present Unusual Challenges
- Lack of Development Tools
  - Debuggers
  - Automated Test Harnesses
  - Stack Tracing
- Available Tools Less Reliable
- Variety of devices

The whole point of the talk boils down to this: my experience (admittedly not *wide* as such) with developers writing software for mobile devices is that testing that software is harder than it is for desktop applications, mainly because a mobile device ain't a desktop PC.

Certainly, writing software for mobile devices presents some challenges not really experienced by Windows Desktop Application Developers: there are far far fewer development support applications that run on mobile devices, things like debuggers, automated testing tools, and so on, and those that are available tend to be somewhat more idiosyncratic.

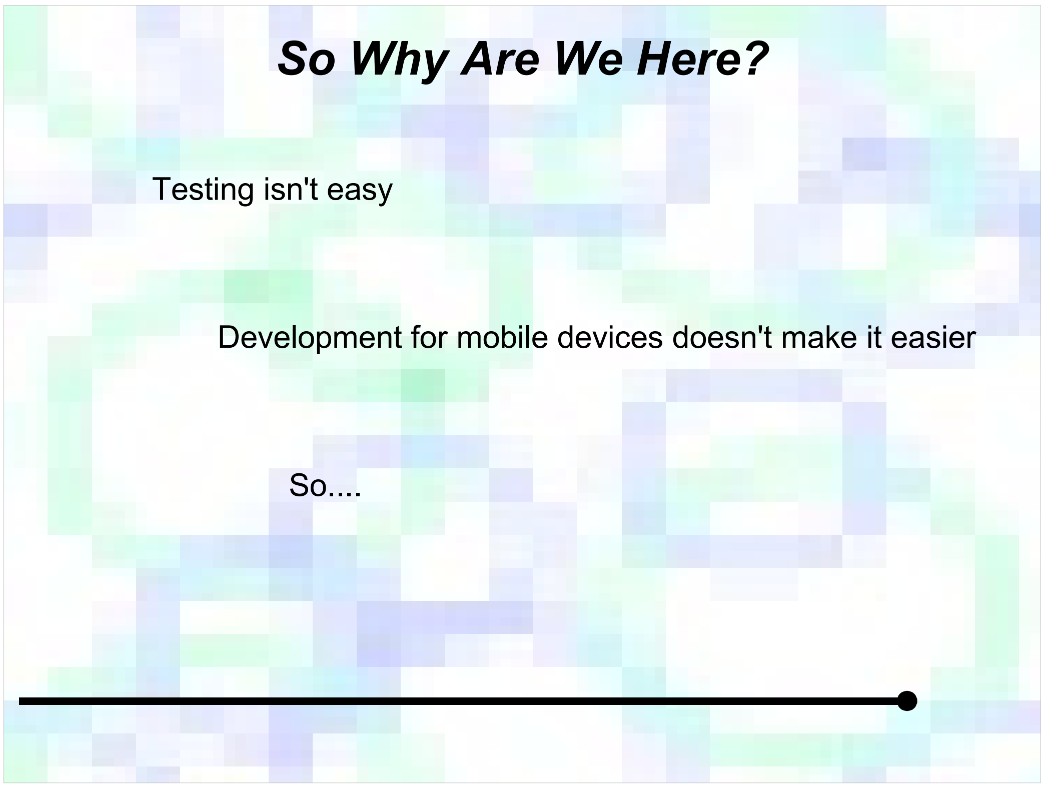


Who's Tough?

- Windows Programmers
  - `LPCTSTR lpszString;`
- Unix Shell
  - `sed 's/^\(.*\)l3?t[ \t]*h[4a]x0r5\([ \t]*.*\)$/$1nubbin$2/g' < 'cat foo | tail'" > /dev/null`
- COM Programmers
- VB Programmers
- Embedded Systems Developers
- Game Programmers

Developers of all kinds think that they've got it tougher than everyone else. And for varying reasons. The truth is that all developers face different challenges., but some challenges are faced by developers of all kinds.

Additionally, all developers can follow some of the guidelines this talk discussess to make better, and more easily testable software.



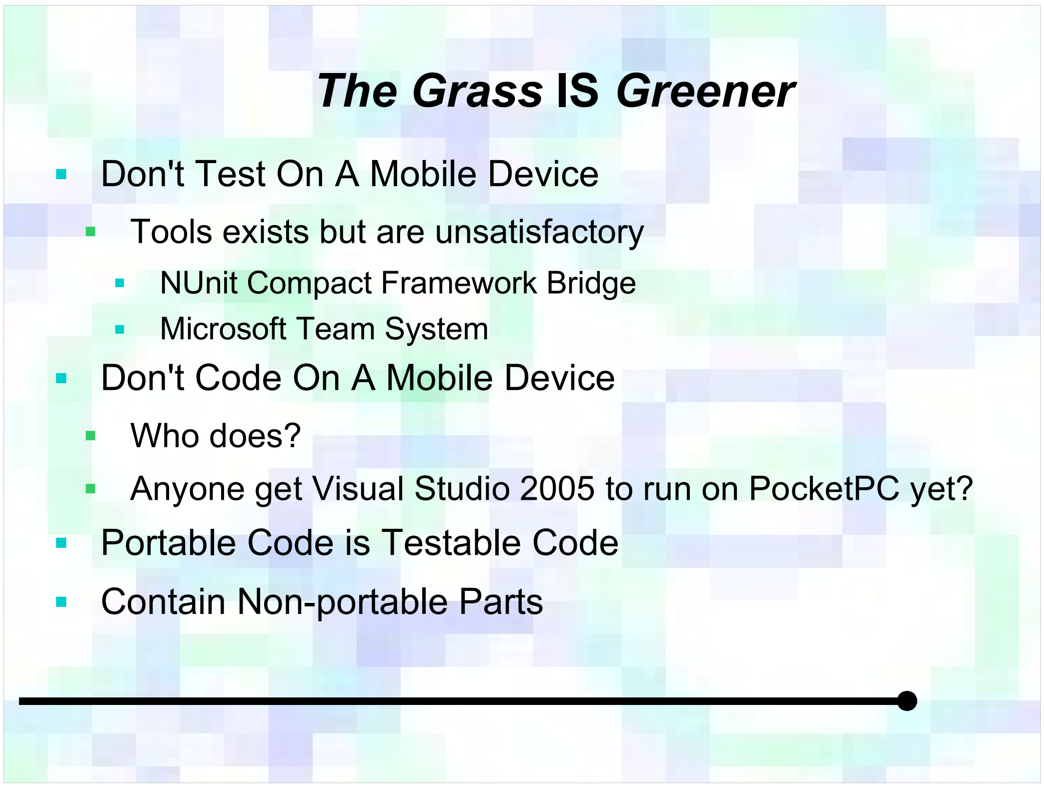
So Why Are We Here?

Testing isn't easy

Development for mobile devices doesn't make it easier

So....

Largely, the topic of this talk is about how to design code to *avoid* problems associated with code being hard to test. Of course, these difficulties can't be ignored altogether – that would mean ignoring the mobile devices altogether, which wouldn't make anybody very happy – but they can be avoided to a large degree.


The Grass IS *Greener*

- Don't Test On A Mobile Device
  - Tools exists but are unsatisfactory
    - NUnit Compact Framework Bridge
    - Microsoft Team System
- Don't Code On A Mobile Device
  - Who does?
  - Anyone get Visual Studio 2005 to run on PocketPC yet?
- Portable Code is Testable Code
- Contain Non-portable Parts

If testing code on a mobile device is hard because the tools aren't available, then:
Don't test on the device

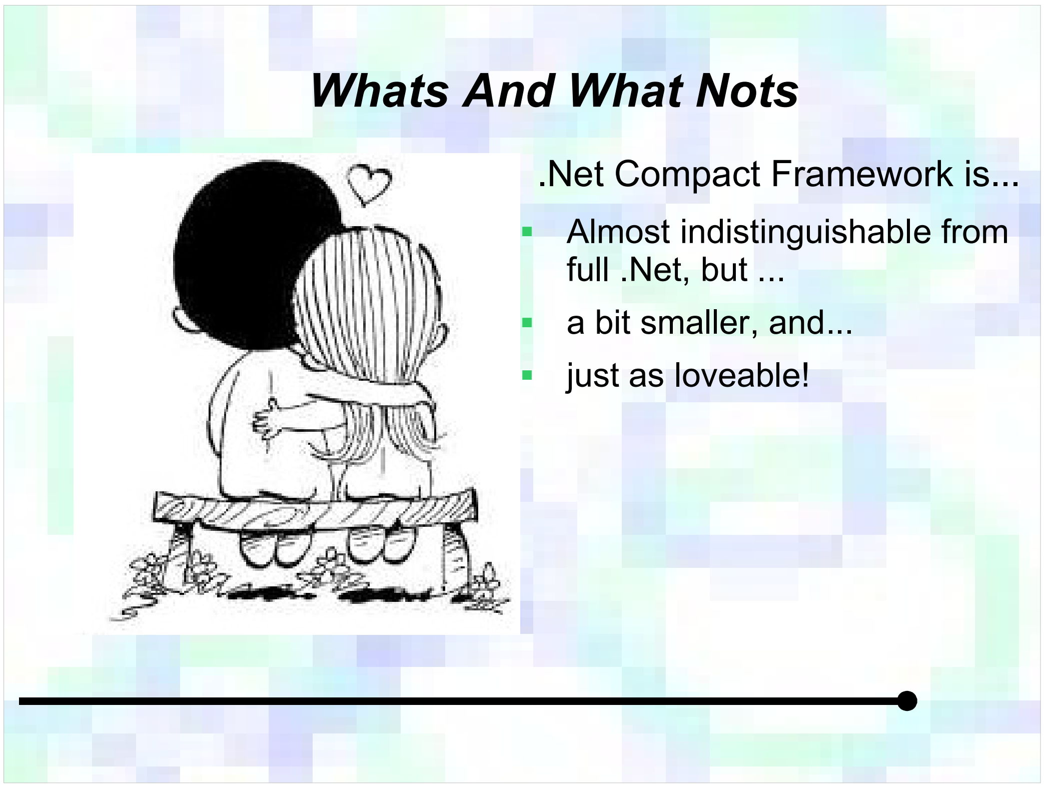
What this really means is, write code so it doesn't require the device to run on – test on your development machine. Taken another step further...
Don't write code on the device

Which is what all mobile developers do anyway. We all use a development environment that runs on our PC, and then transfer the assemblies to a mobile device or device emulator to run (test) it. If code can be *written* on the desktop, it can be *tested* there too.

.Net is a platform abstraction. Write code to the abstraction not the platform.

Device Specific Code isn't unique to mobile devices. Contain it!

Portable code is easier to test. Aim for portability.



Whats And What Nots

.Net Compact Framework is...

- Almost indistinguishable from full .Net, but ...
- a bit smaller, and...
- just as loveable!

The .Net Framework Library is pretty large – containing detailed support for technologies such as Windows programming (Windows Forms), XML, data access (ADO.Net), files, threads, etc., etc. Mobile devices, however, tend to be limited in terms of both storage space and available memory, and so the .Net Compact Framework is a cut-down version of the .Net library. Support for windows, XML, data access and all the other things are there, but they are *smaller*. More importantly, they are provided as a sub set of the full framework facilities. This is a crucial point for this talk – because **all code written for the .Net Compact Framework is valid for the full framework also**.

Mobile devices are different to PC desktop environments in many ways. Already mentioned are restricted storage – both available RAM and disk storage; the latter is commonly provided as FLASH memory.

Mobile Devices use ARM processors, a RISC based CPU instead of the CISC x86 architecture. The final compiled code is therefore targeting a different CPU – but this is one of the differences abstracted away by using .Net/.Net CF; ultimately all code is compiled to .Net.

The UI for a mobile device is restricted in size and functionality, and generally use a touch-screen UI paradigm in replacement of a mouse. Apps written for such Uis are very different in look-and-feel to Desktop Windows Apps.

Apart from those devices already mentioned, mobile applications make use of some other very different devices to PC applications, such as GPRS, bar code scanners, bluetooth connectivity, and so on and so on.

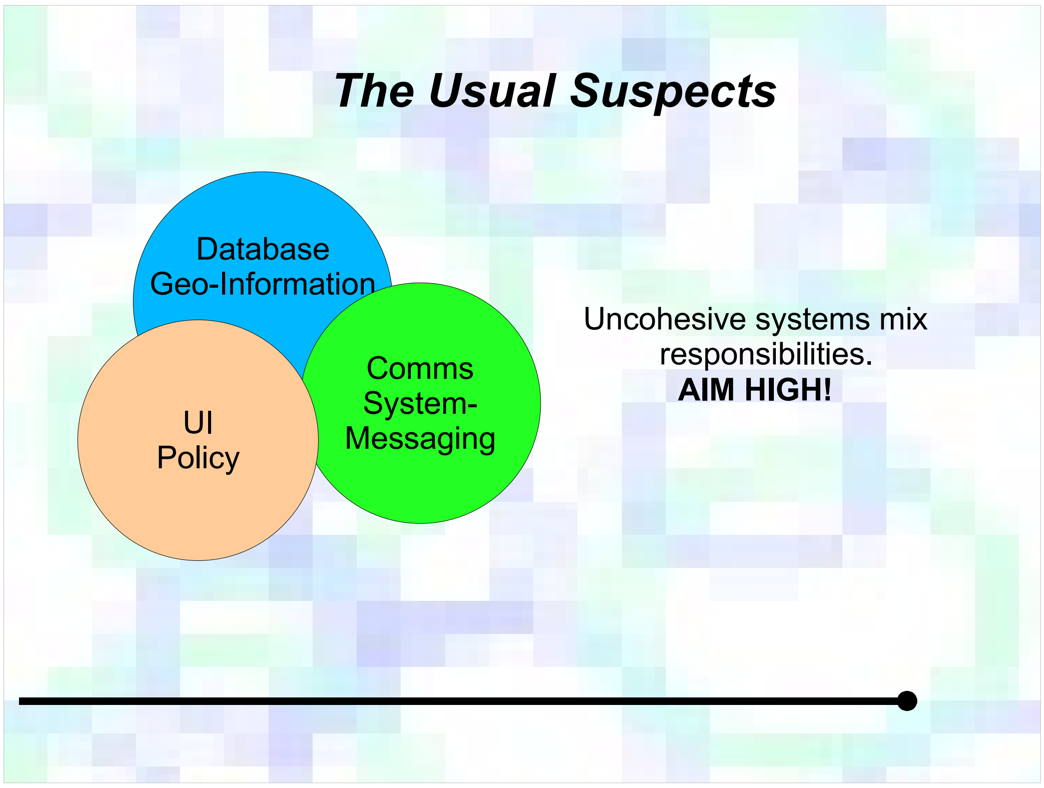
Essentially what's really different is the same as what's different for other types of software: UI, device-specifics, connectivity. Other differences can also be applied: 3rd party libraries, data storage, graphics, controls, ad-infinitum.

The bottom line is it's not really that different at all!

OK on to the secrets...

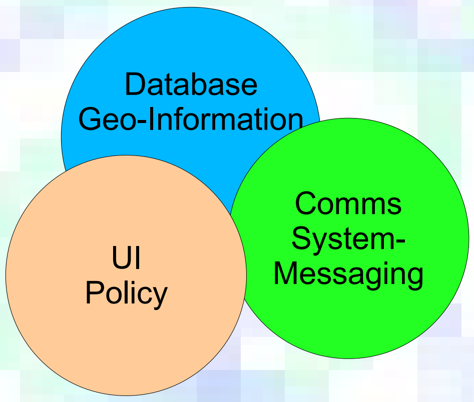
Top Secret

Well not really...

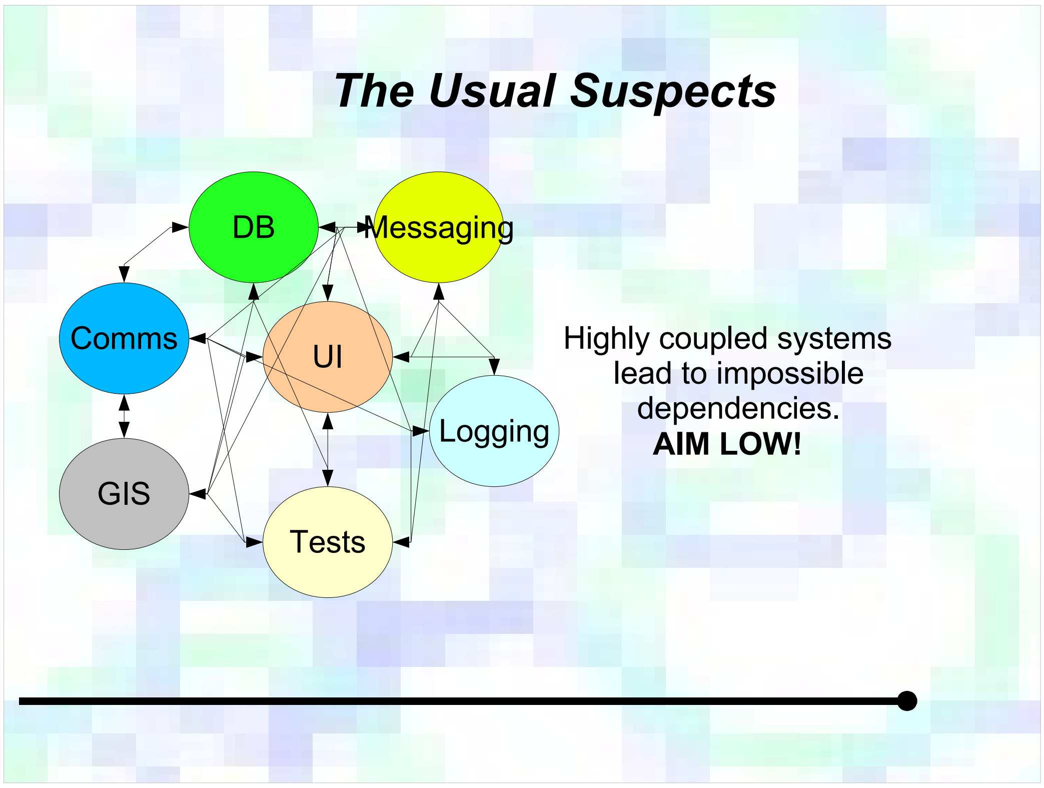


The Usual Suspects

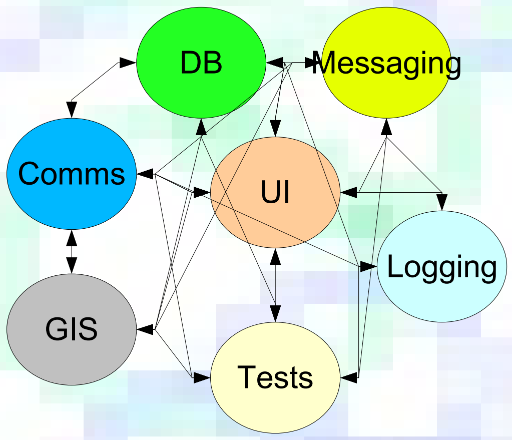
Uncohesive systems mix responsibilities.
AIM HIGH!



Our aim should be to ensure that if we have a need for a UI, the code that provides the UI does so *and nothing else*. Similarly, access to local data, network communications, information from Geographic Information Systems. And so on and so on and so on.
Ultimately, following this route we end up with a bunch of services like 'DB', 'UI', 'GIS', that don't actually talk to each other.

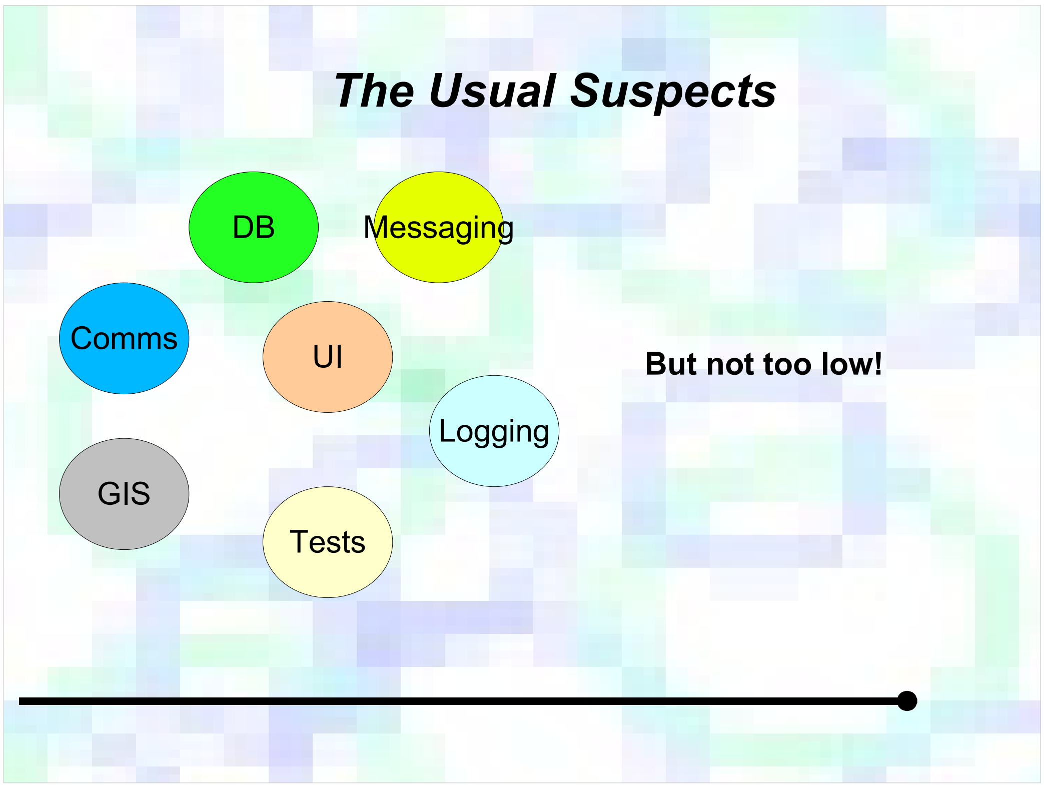


The Usual Suspects



Highly coupled systems lead to impossible dependencies.
AIM LOW!

We want to minimise the coupling between the different parts of a system, so that each can be tested independently of the others. This is important for things other than testing as well, and relates strongly to dependency management, maintenance and reliability in the face of change. However, a system with *no* coupling isn't a system, it's a group of unrelated components, and of no use whatsoever to anybody at all.

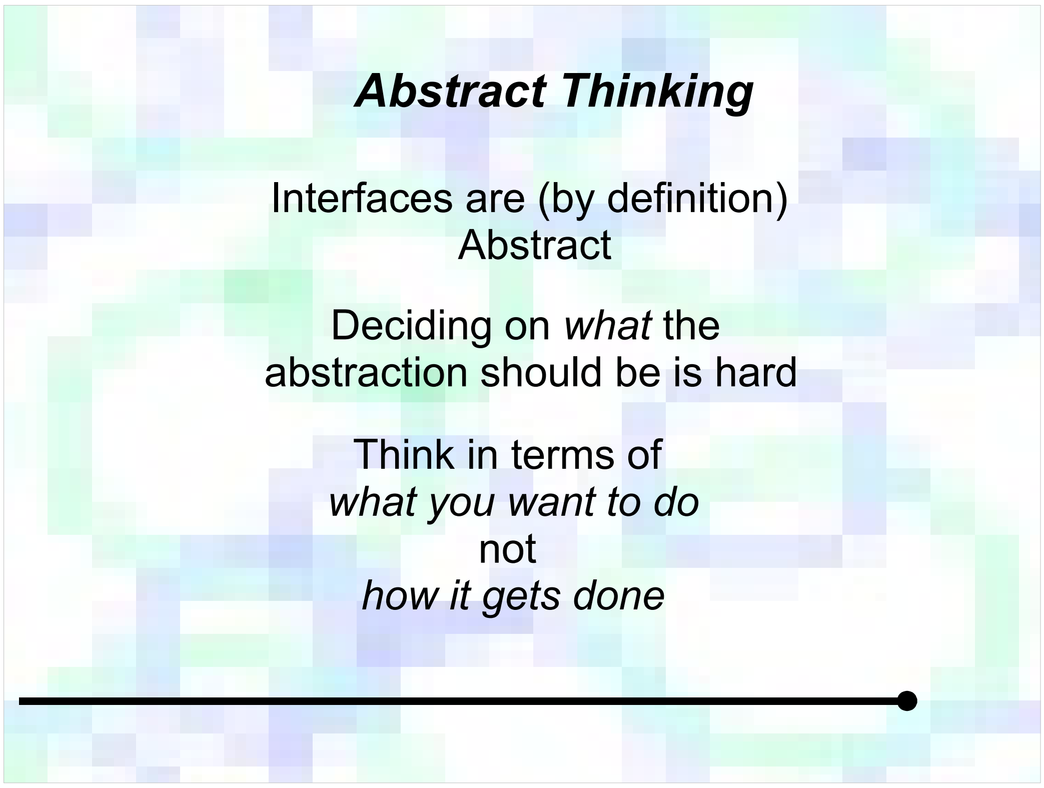


The Usual Suspects

DB
Messaging
Comms
UI
Logging
GIS
Tests

But not too low!

Rather than have any component/service/sub-system in a system be able to chat directly with any other (high-coupling), the trick is to minimise the surface-contact-area between the parts. This is achieved through use of well-defined interfaces and communication channels. The point of programming to an interface is that the implementation behind that interface can change (we'll look at this in more detail soon). Using *only* that interface as the line of communication between components reduces the dependency between them – and thus the coupling, which is made as low as possible (but no lower). But there's more to it than that (isn't there always?)
It's common that some components need to talk to several other components at the same time, to build up a 'big picture'. UI's are the most common culprit, needing simultaneous access to databases, networks, etc. all the time. The main thing to notice here is that the UI shouldn't need to know that the answers to its questions are routed over a network, or come from a mixture of local and remote data storage, or some GI system – it needs only to know that they *are* the answers to the questions.



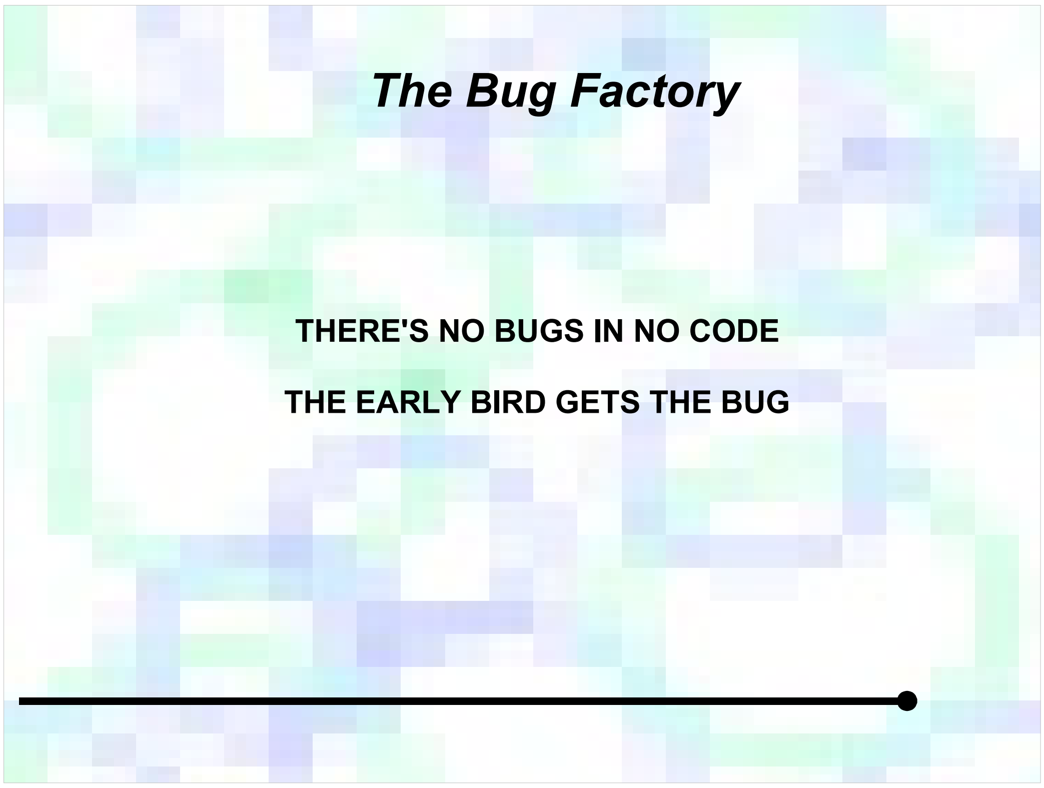
Abstract Thinking

Interfaces are (by definition) Abstract

Deciding on *what* the abstraction should be is hard

Think in terms of
what you want to do
not
how it gets done

NetCF is a *platform abstraction*, insulating you, the developer, from the vagaries of hardware, OS and system functionality. What we want to achieve is abstraction within the application, so for example, if a form on the screen allows the user to enter a surname and have a list of matches presented, the UI code needs to ask a question: “Give me a list of names matching this entered string”. Instead of composing a SQL select statement, calling directly to the DB layer and/or the comms layer and parsing out the resulting dataset, the UI wants to ask a simple question, and have an answer returned in a format simple for it to understand and display.
This needs some kind of application layer, a manager component which routes requests/responses between components in the system. Commonly this layer will also encompass business rules and policies, too – the ubiquitous 'Business Logic' component.



The Bug Factory

THERE'S NO BUGS IN NO CODE

THE EARLY BIRD GETS THE BUG

At the end of the day, what we really need is a system that works. What 'works' means is a bit vague, but often there is some functional specification which must be adhered to, and the app (or whatever) must run 'in the field' without *too many* problems. As programmers, when we talk about 'works' we generally mean that it compiles cleanly and runs without crashing. Um. Too often. There may be subtle bugs that we know of, and some we don't (yet), but it basically 'works', we just need a bit more time to tidy things up a bit.

Software development is often likened to a factory environment. Generally inaccurately, but there it is. The real truth is that software development is a *bug* factory – a harsh reality!

The act of writing code is the act of introducing bugs. It's inevitable, unalienable and intransigent. And here we reach the heart of it, the real point.

Testing is all about lessening the effect of writing code.

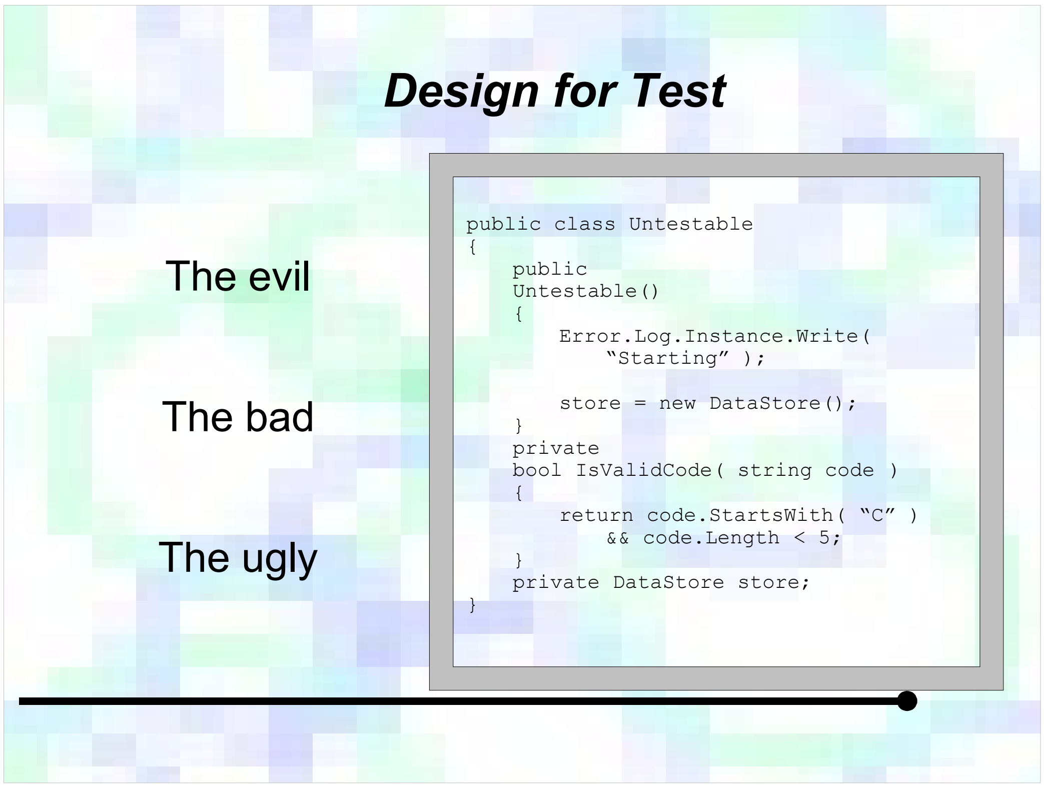
We all know the mantra – the cost of a bug increases exponentially along the lifetime of a project.

The early bird gets the bug

Obviously enough. More specifically, tests performed earlier find the earliest bugs. Test *code* also performs well as regression test, so once a bug is identified and fixed, it has a test to ensure it doesn't poke its little head up again. Or at least, if it does there's always a bird early enough to snap it up before it becomes a menace.

The difficulty comes when code can't be tested anywhere except when it's running on some tiny device with no debugger, probably no keyboard, a tiny UI, no console output and your best friend is either an emulator or ActiveSync. There's a limit to the patience of any developer under these circumstances.

So let's say it again – test your code as if it were a regular app written for the desktop. This can only be achieved if the code is designed to be tested.



Design for Test

The evil

The bad

The ugly

```
public class Untestable
{
    public
    Untestable()
    {
        Error.Log.Instance.Write(
            "Starting" );

        store = new DataStore();
    }
    private
    bool IsValidCode( string code )
    {
        return code.StartsWith( "C" )
            && code.Length < 5;
    }
    private DataStore store;
}
```

This is only a slightly contrived example. It's taken from real commercial code, and in principle and spirit it's the same code. In fact the original was a bit worse, for a number of reasons, but this small snippet serves as an example of several 'bad' practices.

SINGLETON

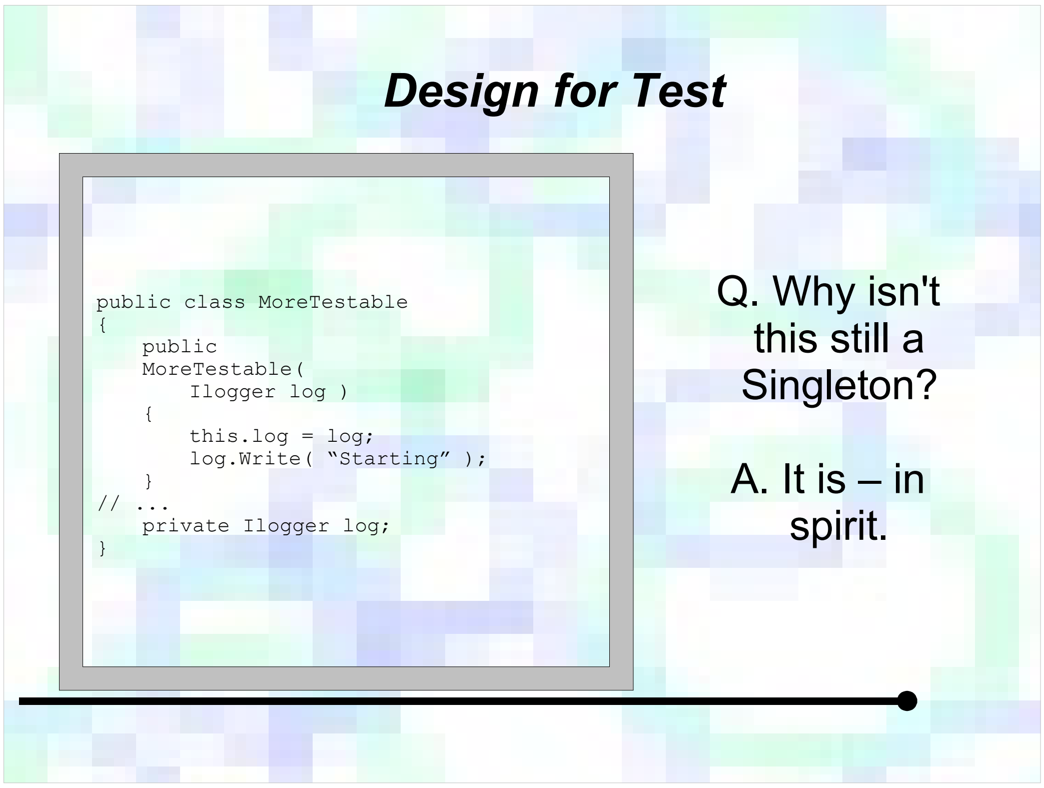
The writing of the log is tied directly to the class, meaning it can only ever use the "ONE" true log. This is bad for testing because as a rule, we don't want huge debug logs when tests get run every few minutes, but it's bad for design and coupling as well.It makes it hard to find and hard to change. Note also that the dependency is 2 levels deep : "Error" and "Log" are *both* needed to even compile this class.

Inside-Out Dependency

For some reason this class needs access to a 'DataStore', presumably some kind of database. The problem is that dependency is hidden in much the same way as the Singleton Instance, but more subtly. The responsibility for creating the database should surely be someplace higher up the foodchain – especially if other parts of the app need access to it; creating more than one datastore isn't often a good sign :-)

Hidden Policy Rules

The rules about what is a val 'code' are encoded within this class, which appears to go against its prime reason for being, and besides, this sort of code is a prime target for tests – which would be impossible here. All these things make it hard to test: In order to compile the code, the test code needs to ensure access to the logger singleton is available, and that the correct references for DataStore are available. The IsValidCode method can only be tested directly by making it public.



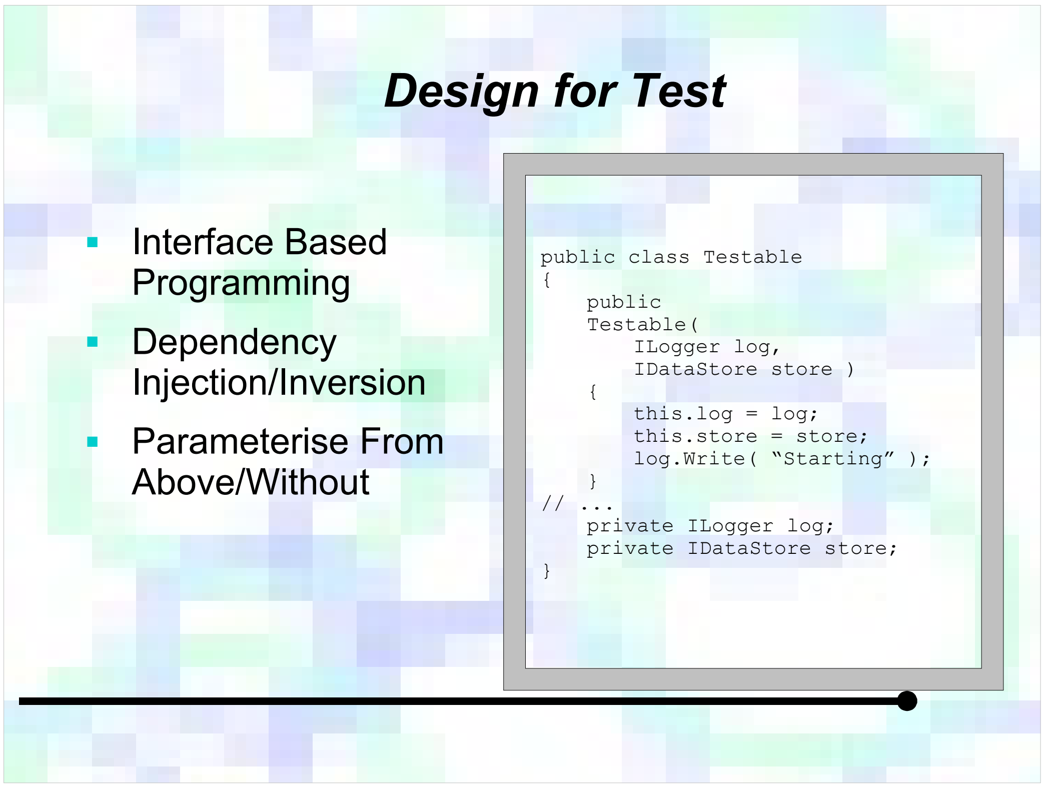
Design for Test

```
public class MoreTestable
{
    public
    MoreTestable(
        Ilogger log )
    {
        this.log = log;
        log.Write( “Starting” );
    }
// ...
    private Ilogger log;
}
```

Q. Why isn't this still a Singleton?

A. It is – in spirit.

If we want to test the code without the logger doing anything, we need a way to swap the real logger for a faked one. One simple way might be to inherit from the existing logger and override all the stuff that actually writes to a log output. That's OK if (a) the class isn't sealed and (b) those methods are virtual. It also doesn't get rid of the dependency: the test code still needs to reference the logger class. By having a real logger and a fake one implement a common interface, our class can make its dependency only on the interface. To make this work, the class needs to have an instance of the logger object passed to it; a constructor seems a likely place:
This has two main effects. The responsibility for creating the log is obviously somewhere else: since Ilogger is an interface, and the class depends only on that interface, it can't create one of its own. AS far as its concerned, there is one and only one instance of “log”. It's always accessible from a well known point, and is guaranteed (by the program) to be instantiated. So it *IS* a Singleton. Right?
The other effect is that any implementation of the Ilogger interface can be passed in.

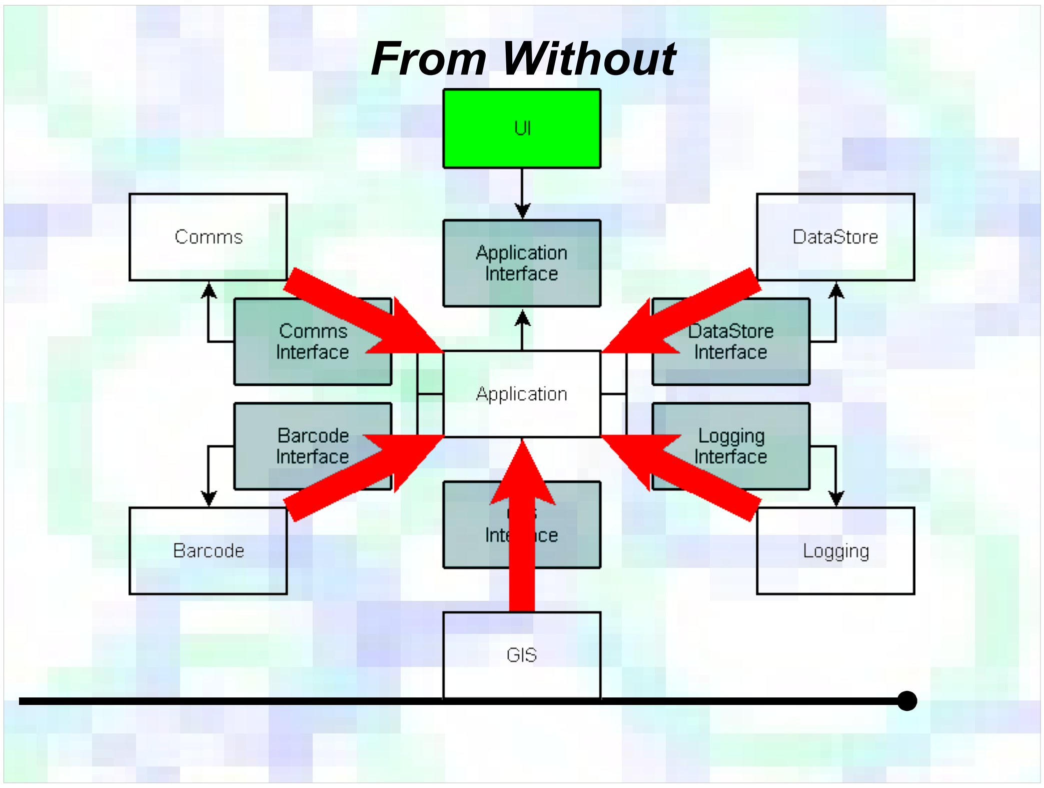


Design for Test

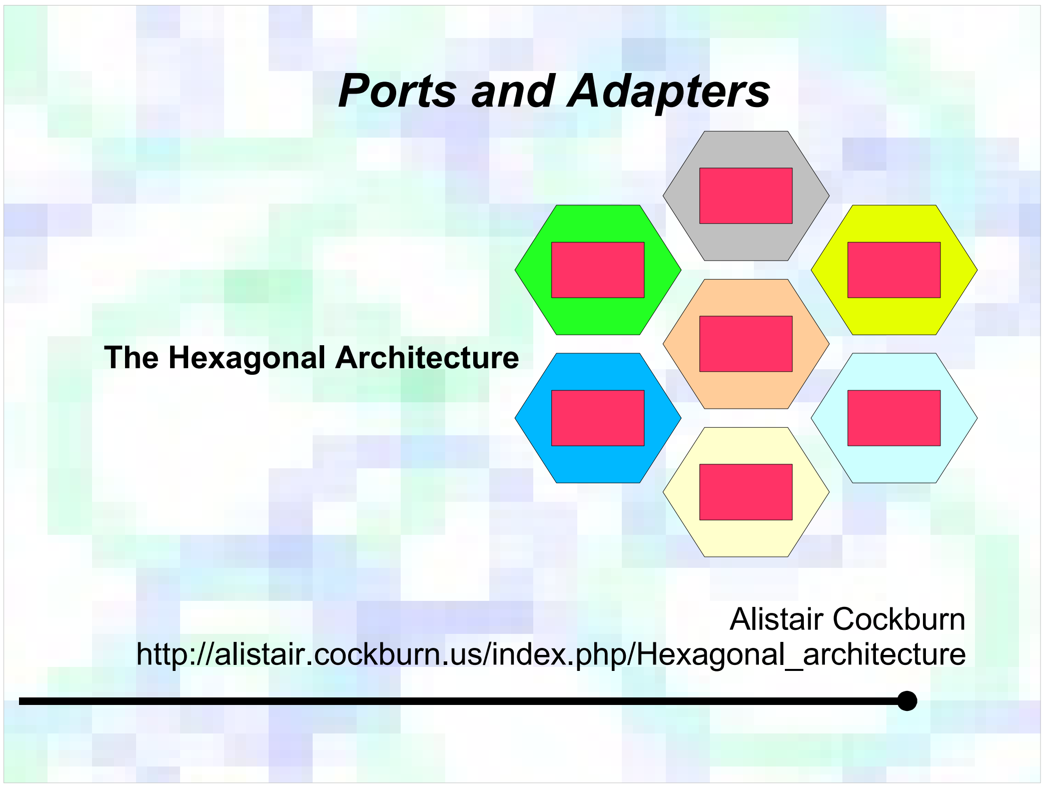
- Interface Based Programming
- Dependency Injection/Inversion
- Parameterise From Above/Without

```
public class Testable
{
    public
    Testable(
        ILogger log,
        IDataStore store )
    {
        this.log = log;
        this.store = store;
        log.Write( "Starting" );
    }
// ...
    private ILogger log;
    private IDataStore store;
}
```

What we want to lose now is that encoded business logic. It's harder to decide exactly where this kind of logic should go: perhaps a method call on some other object, perhaps an interface in its own right. Perhaps it makes perfect sense exactly where it is.
The point here is to identify this kind of code as application policy – it's a rule of some kind, not strictly functional. Having such rules dotted all around the code may be unmanageable, but gathering them all together into one place may unhygienic. Decide how they need to be tested, and design for that accordingly. We;ll arbitrarily decide that this rule belongs in its own class, perhaps as a static method. At least this reduces the responsibilities exhibited by this class.
So now we come to the database. Obviously our class has some responsibility which requires access to a data store of some kind. The main problem we identified just now was that the class shouldn't have the knowledge built in to it of just exactly which data store to create.
Once again, we need to pass in a data store interface, for *exactly* the same reasons as for the logger. We mentioned already that both the original creation/use of a data store and the singleton instance access represented Inside-Out dependencies. Here it's easy to see the dependencies have been inverted, with the *outside* code having responsibility to create the objects and *pass them in*. In other words, our class has been Parameterised From Above. Well, actually, Parameterised From Without.



A traditional layered or tiered architecture would have the datastore at the bottom, the UI at the top with the application in the middle. But where would the other parts go? As part of the Application? Probably they'd all be clumped together in the middle layer.
Here we see a different shape to the system, more of a star really. What's interesting is that it identifies “services”on which the application depends, but doesn't really differentiate them from database (bottom) or UI (top) components – they're just other services.
Let's not confuse this with SOA – that's a technology. This is just a diagam...
The other interesting feature is the highlighting of interface components to manage the dependencies. Without them, we might introduce circular references. Imagine that the Comms component uses the Logging component to record failures, and that occasionally the logging component needs to send messages to a remote host (the whole log for example). If they depend directly upon each other, they have to be A) the same component or B) depend on some other common component.
By using interface projects, we achieve B without contorting the design.
The whole point here is that implementation projects depend only only interface projects, and interface projects depend only on other interface projects where necessary (to introduce types usually).
The application “layer” for want of a better description depends on most if not all other aspects of the system, and so it has references *passed in* to it.



Ports and Adapters

The Hexagonal Architecture

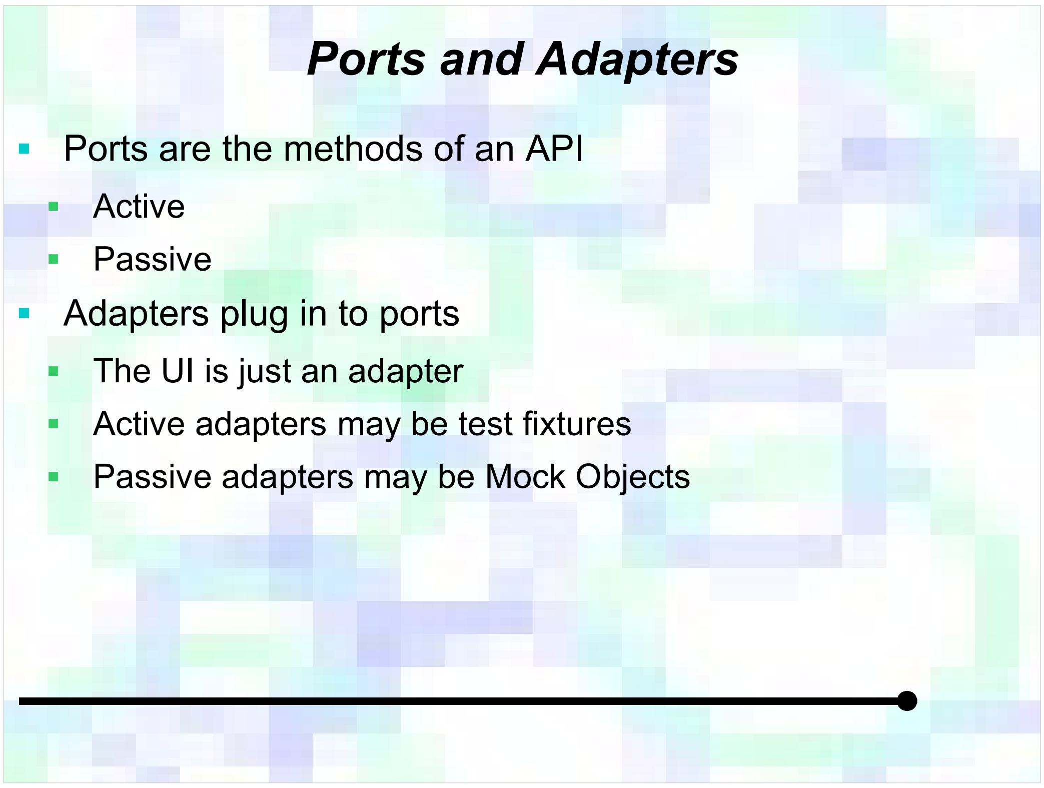
Alistair Cockburn
http://alistair.cockburn.us/index.php/Hexagonal_architecture

This architecture, proposed by Alistair Cockburn, talks about how an application has “ports”. In this context, the Application is some entity which represents the Business Logic (here we go again), and it provides an API or several APIs over which “Adapters” can communicate with it. Adapters, in this context, might include a database, a UI, some communication mechanism. Etc etc etc.
I've found it useful to think of apps in this way, because test projects/harnesses are just another adapter, as are mocked objects and so on.
The Ports and Adapters Pattern is a detailed discussion of various aspects of the architecture and its uses and potential, but the headline is basically this:
Hexagon edges are interfaces.
All communication is done through an interface.

Dependency Inversion

“High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions”

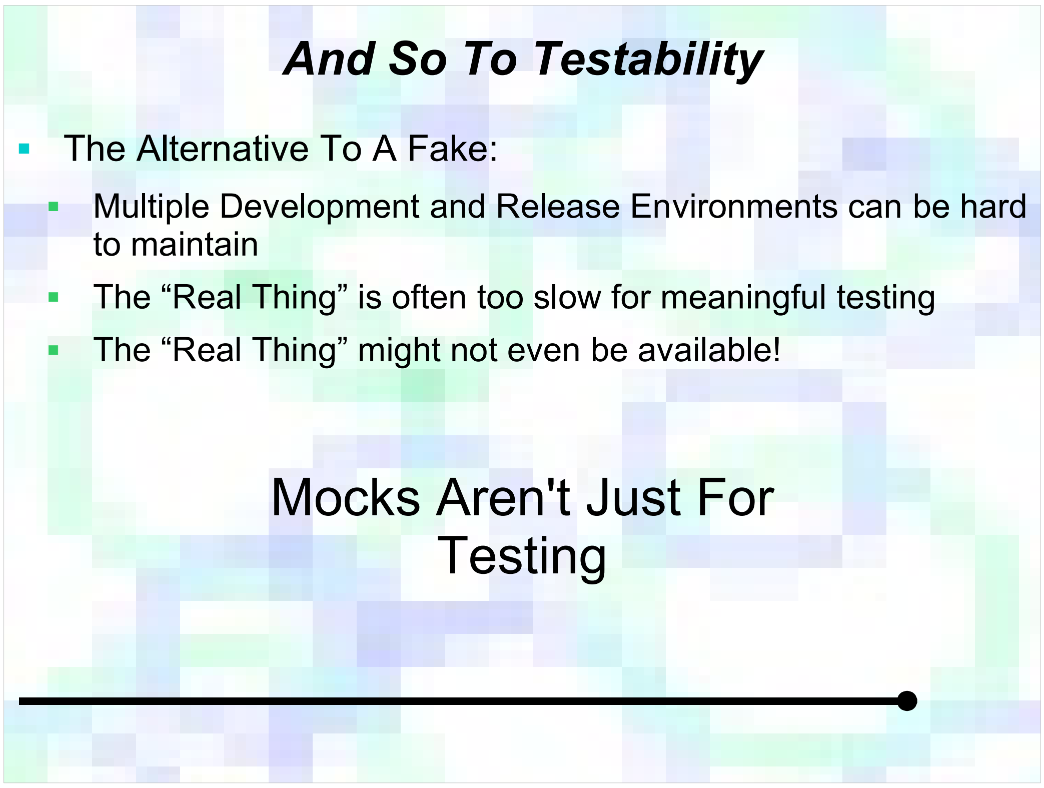
Robert Martin
http://www.objectmentor.com/resources/articles/dip.pdf



Ports and Adapters

- Ports are the methods of an API
  - Active
  - Passive
- Adapters plug in to ports
  - The UI is just an adapter
  - Active adapters may be test fixtures
  - Passive adapters may be Mock Objects

The Hexagonal architecture follows the Dependency Inversion Principle,
The other point of this architecture is the idea of active and passive adapters. Alistair uses the terms Driving and Driven which accurately captures the distinction between adapters which are mocked out for testing (passive/driven), such as databases, hardware abstractions and the like, and adapters which are replaced by unit tests (active/driving) such as the UI itself, or some input component.
Because a port is effectively an interface, it is a plug-in point for any adapter which has or uses the correct protocol (API). An active adapter uses the API of a corresponding passive port; a passive adapter exposes the correct API for its port, upon which active adapters depend.



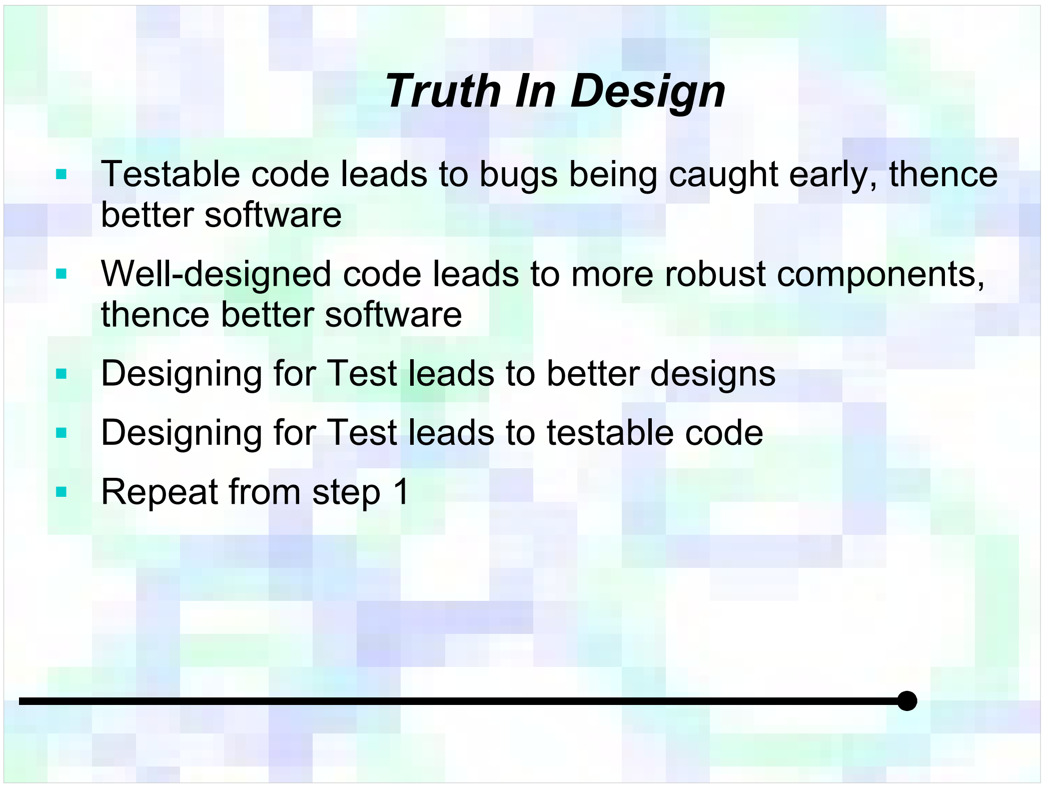
And So To Testability

- The Alternative To A Fake:
  - Multiple Development and Release Environments can be hard to maintain
  - The “Real Thing” is often too slow for meaningful testing
  - The “Real Thing” might not even be available!

Mocks Aren't Just For Testing

This idea of “passing in” instances of these interfaces means that we can use *any* object which implements that interface. If, instead of interfaces, we used real concrete objects, it would be much more difficult to “mock” the behaviour (so to speak), and so testing this code would mean having access to a real database, but we'd probably not want to work on live data, so that would have to be handled by some config somewhere, which we might forget to change when releasing the software, with the obvious consequences.
An often overlooked fact is that unit testing at this level needs to be quick, so having the code under test access real data would make the test much less likely to be run often enough to be useful.
Another often overlooked benefit of fake or mocked objects in place of the real thing is their general usefulness.

Suppose the code depends on a remote comms component which is provided by a 3rd party supplier, and they don't deliver on time, or something. (Unheard of!). If your own code depends only on the interface to their code (if they've been thoughtful enough to provide one) or you've wrapped it inside your own interfaces, you can happily develop your code using the fake, and hopefully just plug in the real thing when it arrives.



Truth In Design

- Testable code leads to bugs being caught early, thence better software
- Well-designed code leads to more robust components, thence better software
- Designing for Test leads to better designs
- Designing for Test leads to testable code
- Repeat from step 1

In the large, this architecture is what you'd be trying to achieve, with test harness code filling the active adapters (UI, messaging) , and mock objects filling the passive ones (DB, Comms).
What it's really all about is separating concerns, and decoupling the system to a large extent. Of course what this has to do with testing should be obvious: any module can be tested in splendid isolation, without worrying about the effects if its dependencies.
So what has it to do with mobile devices?
Well, everything and nothing.
We said early on that the difference between .Net and .Net CF are minimal; the same holds true for the practices you as developers employ to write apps.
Because in our small example we separated out the concerns, and inverted dependencies by using interfaces, we managed a loosely coupled and more cohesive class in the process. Additionally, we achieved a more abstract view of the whole (well, more of it) system by “hiding” the datastore and logging facilities behind abstractions.
The fact that we did those things to make the code more easily tested is just the icing on the cake.
What has resulted is a *better design* because we wanted to achieve testability.

Testing Mobile Device Applications With .Net Compact Framework 2.0

Thanks for listening

Steve Love

steve.love@arventech.com