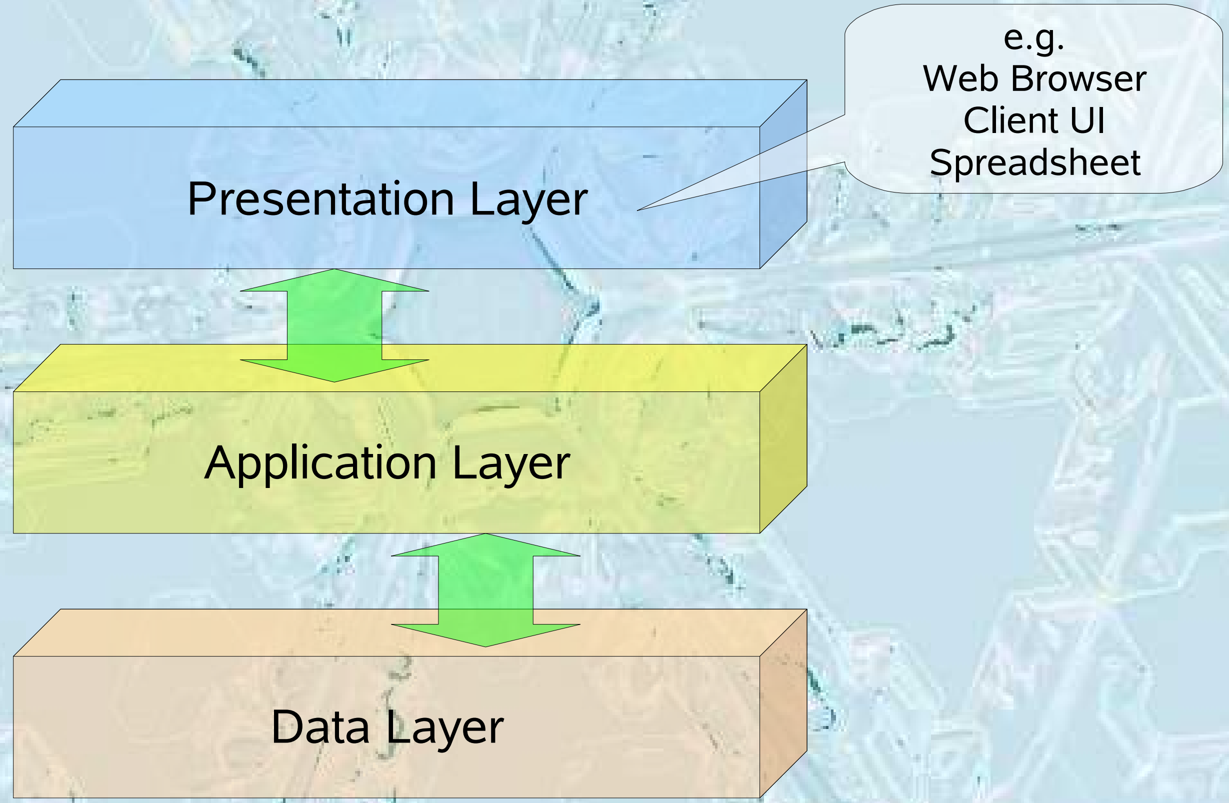
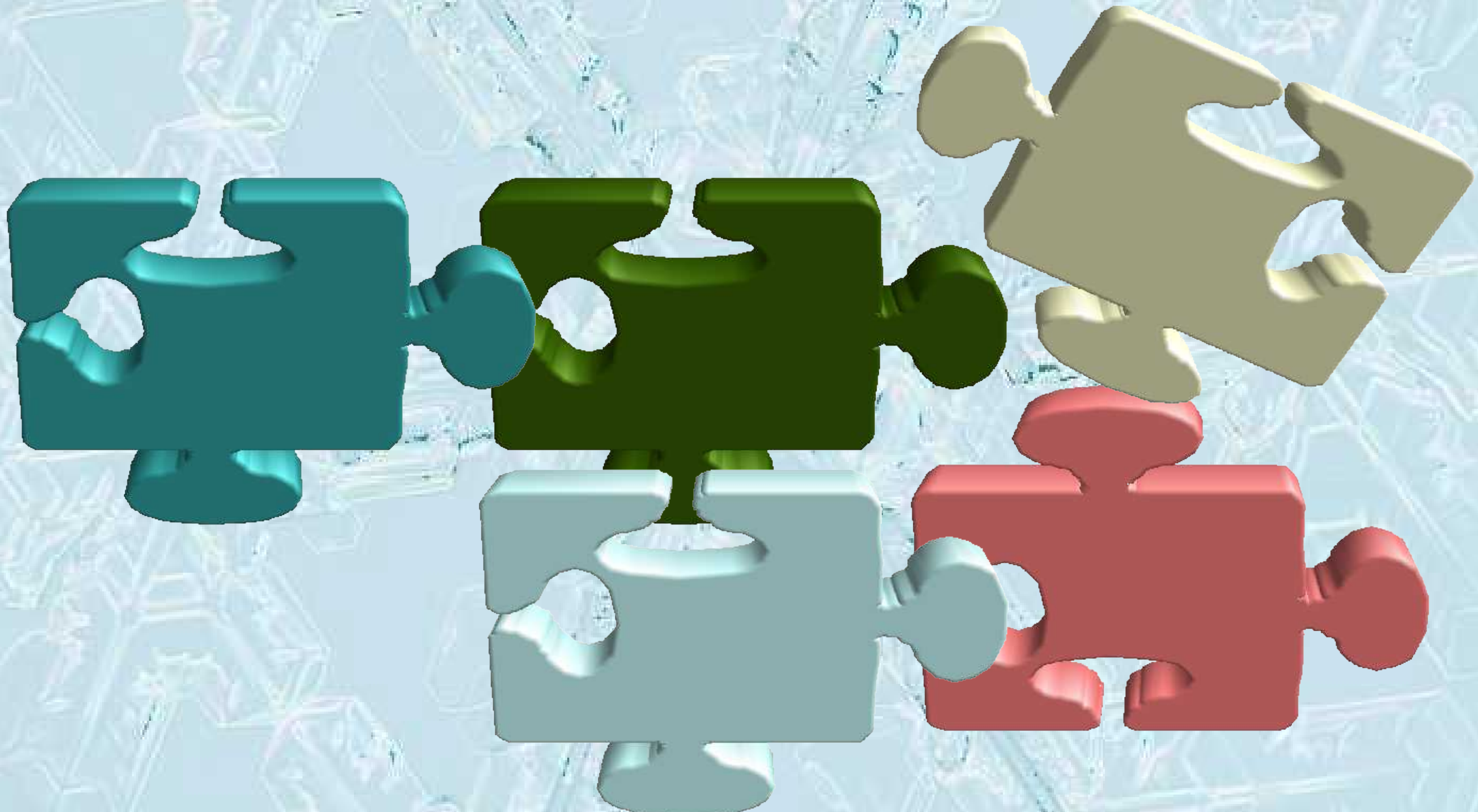
Snowflakes and Architecture

Steve Love
steve@arventech.com

Layer Cake


e.g.
Web Browser
Client UI
Spreadsheet
Presentation Layer
Application Layer
Data Layer

Do The Parts Fit?



Layered Architecture

"...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface."

(Grady Booch, "Object Solution", 1996).

Simple Systems

What's in an application?

Comms	Protocol Validation	GIS
Messaging	Processing	Storage
Audit Logging	UI	Policy/ Security

The Good, The Bad, The Ugly

- Rigid
- Fragile
- Immobile

- Cohesive
- Decoupled
- Layered

Is it really that simple?

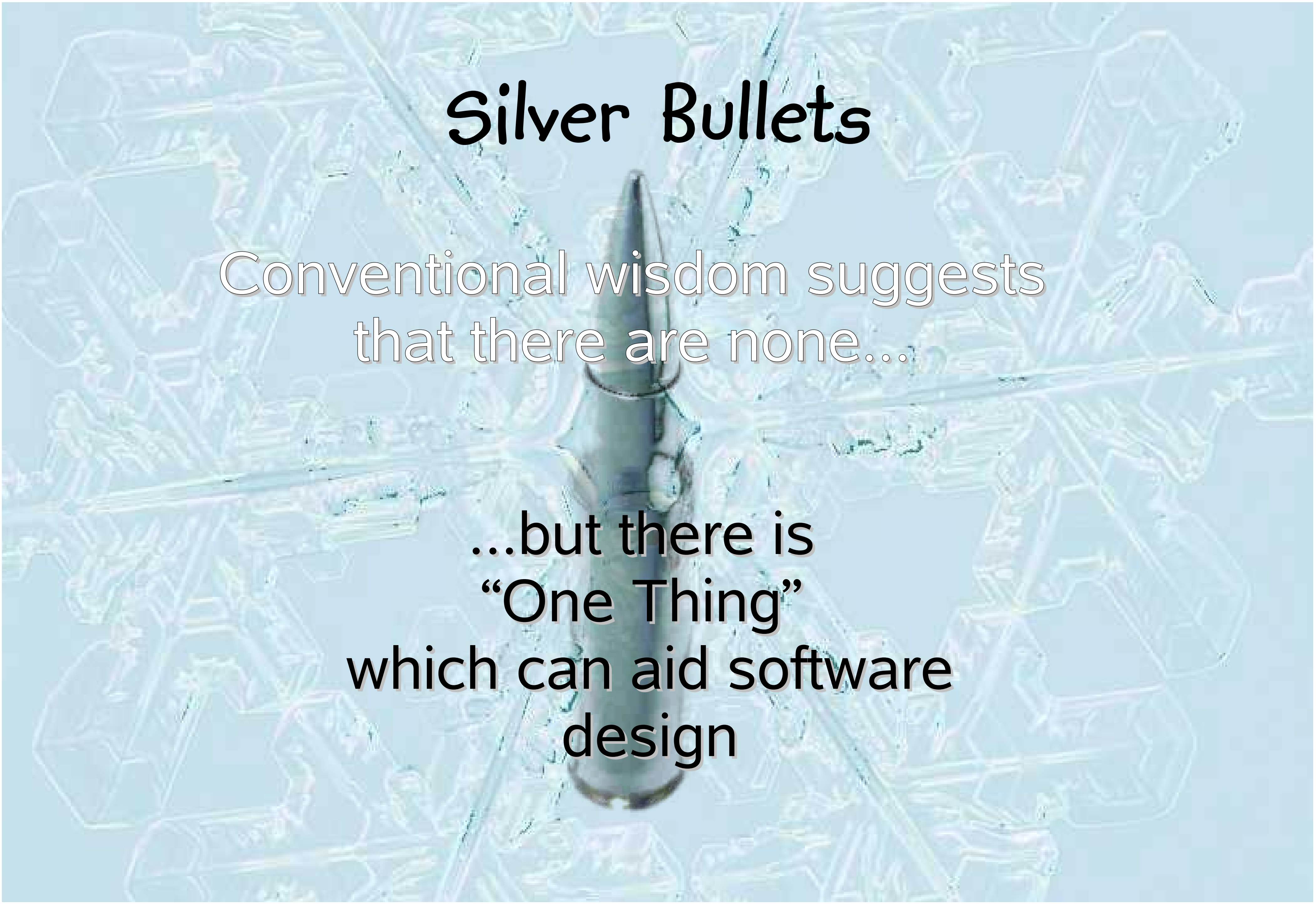
*ities

- Quality
- Maintainability
- Flexibility
- Adaptability
- Generality
- Comprehensibility
- Extensibility

- Utility
- Stability
- Testability
- Substitutability
- Clarity
- Brevity
- (Re) Usability

SIMPLICITY

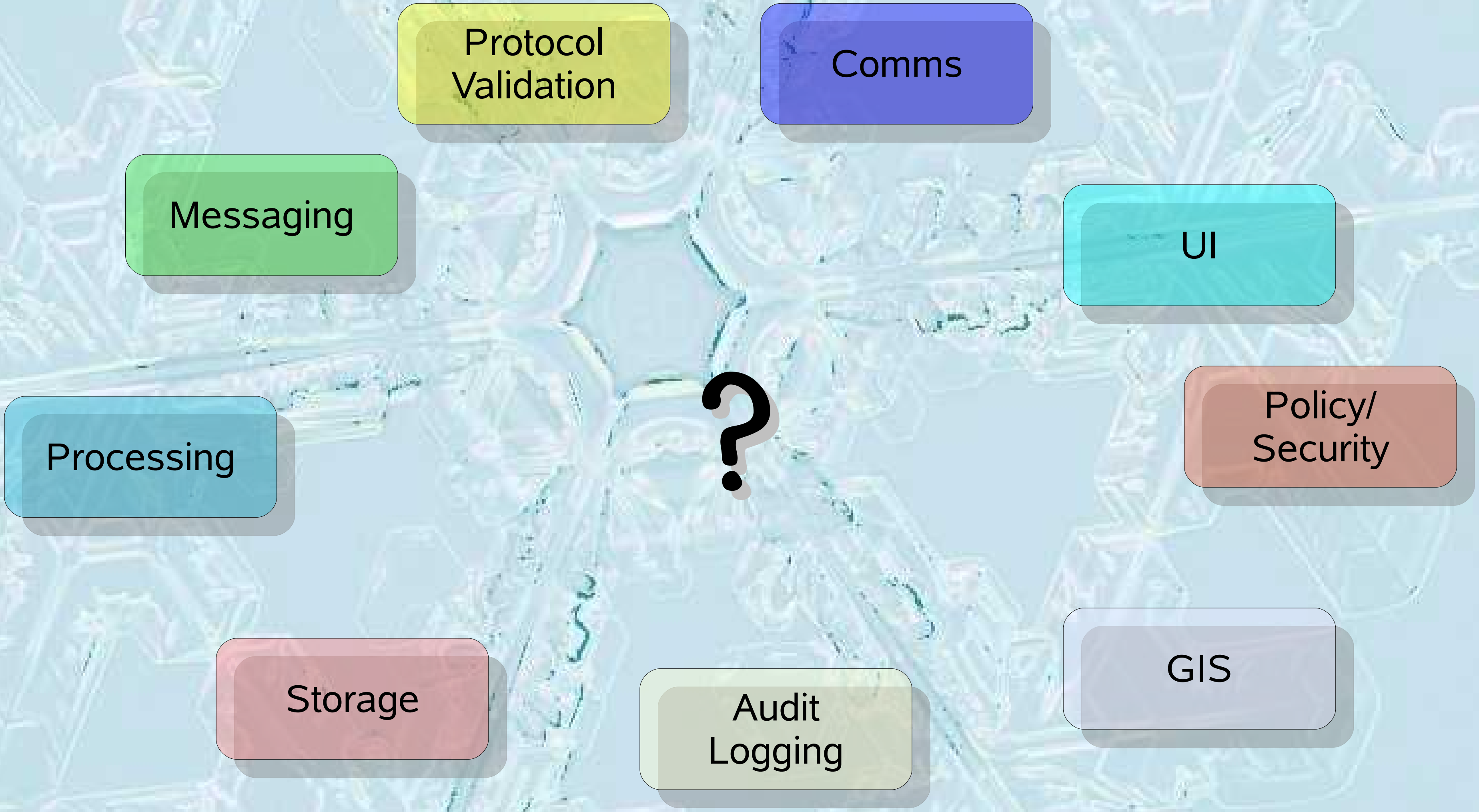
Silver Bullets



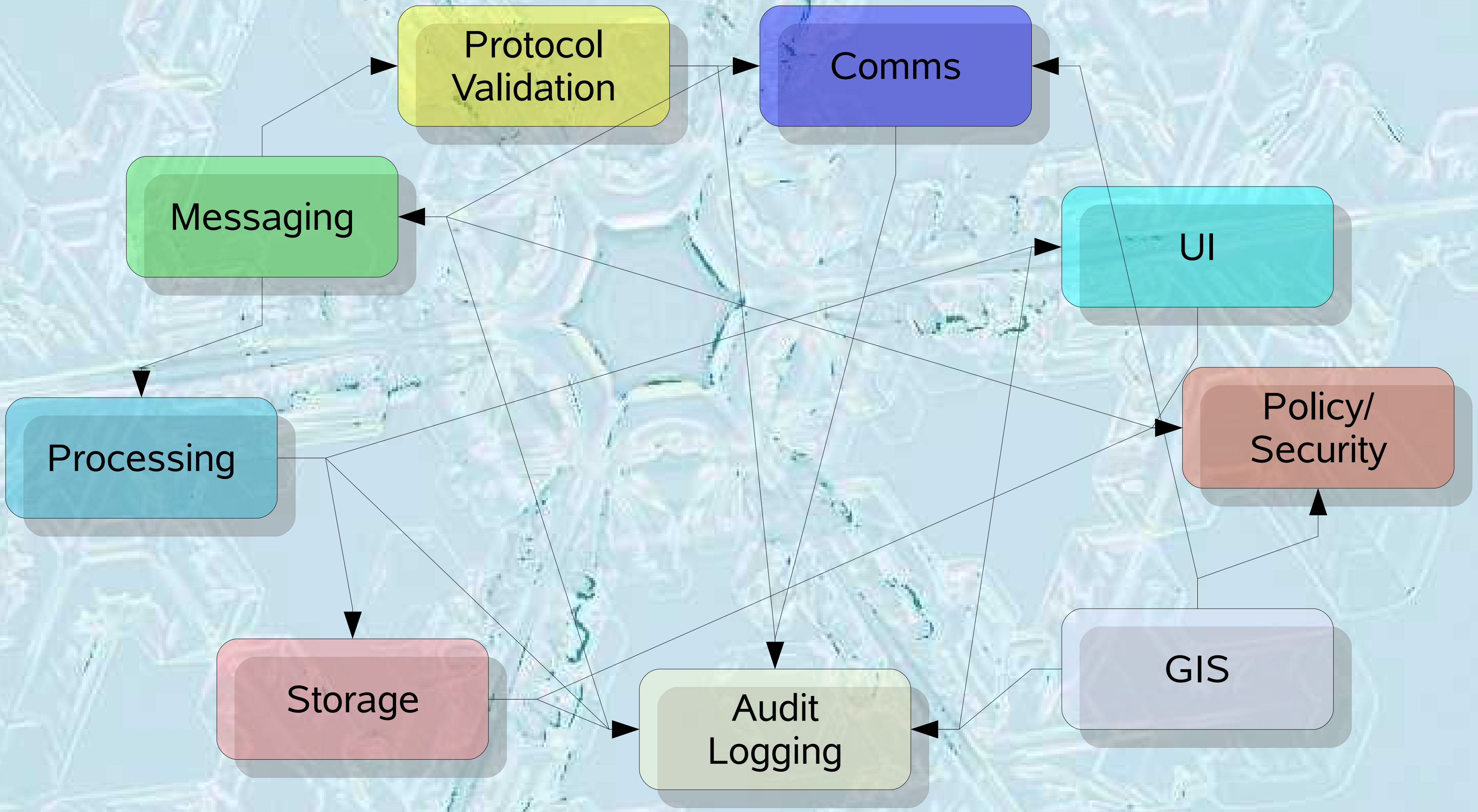
Conventional wisdom suggests
that there are none…

…but there is
“One Thing”
which can aid software
design

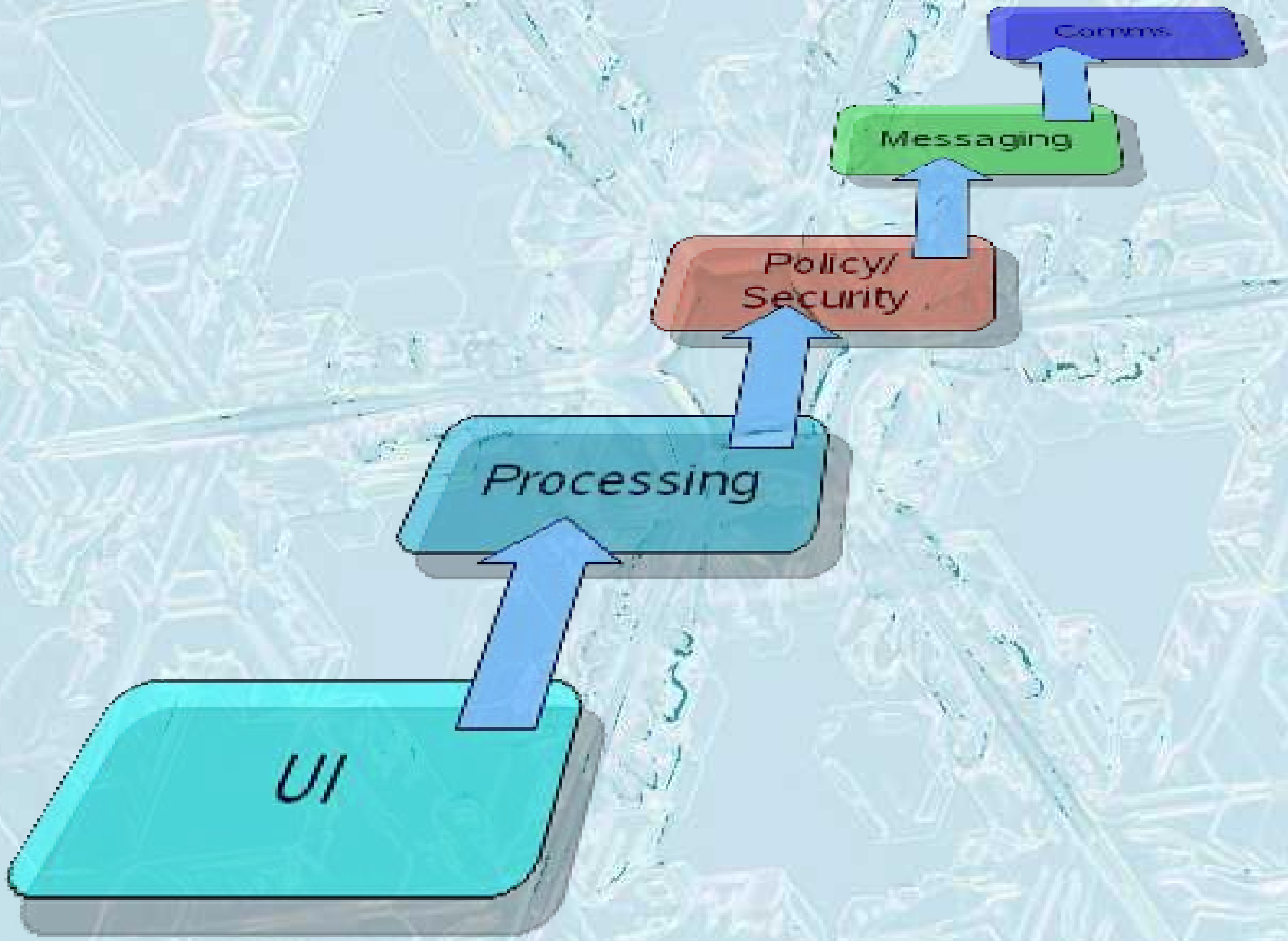
Frontier Arrangements



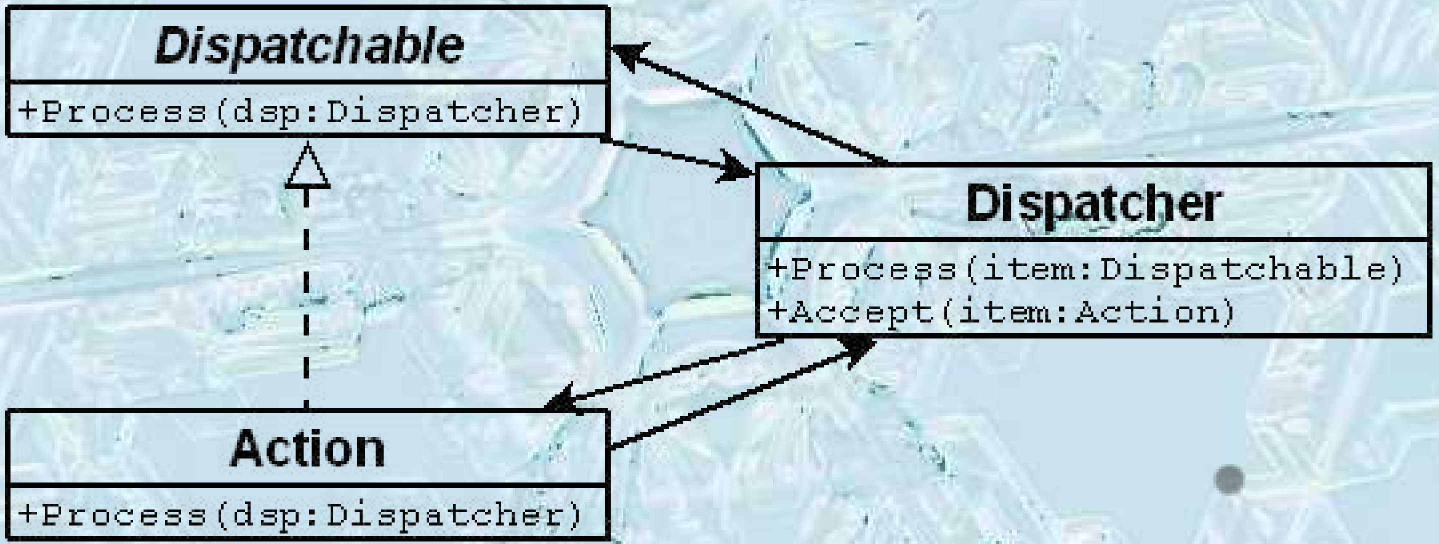
Frontier Arrangements



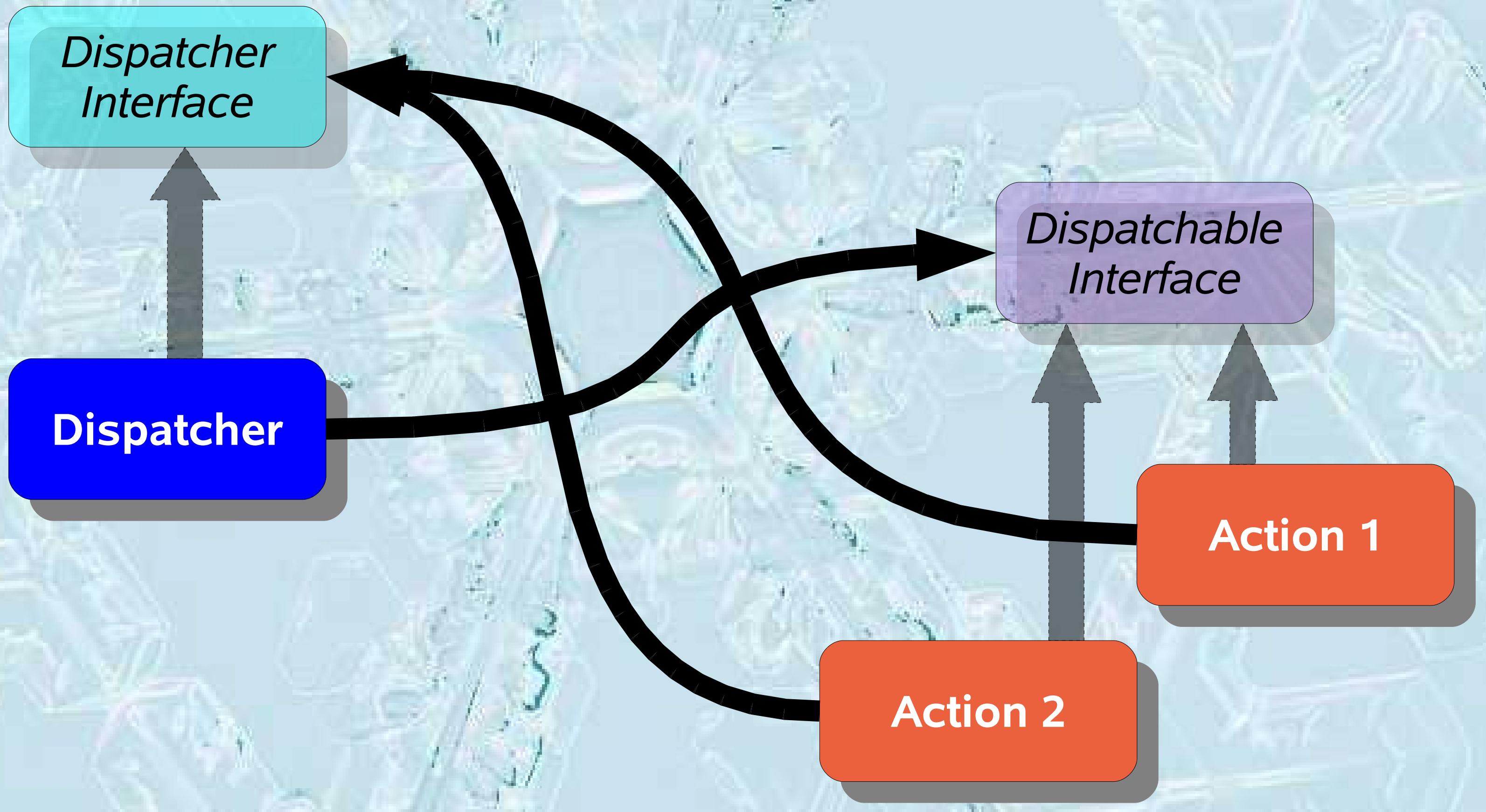
Dependency Horizon



Circular Dependencies



Interfaces


Dispatcher Interface
Dispatchable Interface
Dispatcher
Action 1
Action 2

The Dependency Inversion Principle

- High level modules should not depend upon low level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

Robert C. Martin

The Counter Example: Singleton

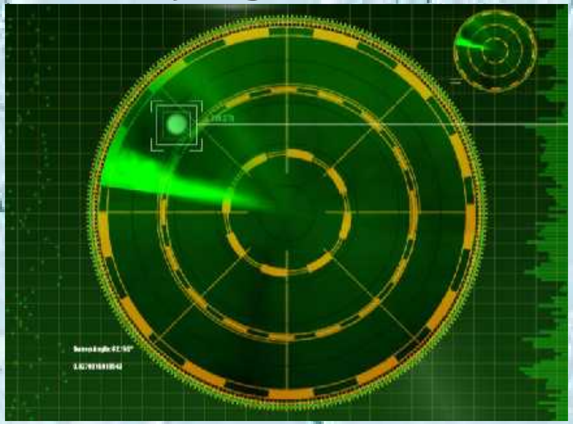
Upside Down Inside Out Back to Front

It's the antithesis of Detail Depending on Abstraction

Upside Down, Inside Out

- Hard-wired Dependency
- Dependency from within
- Testing is difficult
- Rigid, Fragile & Immobile
- Unpredictable Ownership
- Unpredictable Lifetime
- Multiple threads???

Back to Front



Destruction-Managed Singleton:

INTENT Ensure the destruction of interdependent singletons in the correct order, and guarantee that there are no attempts to use previously deallocated objects.

Evgeniy Gabrilovich, C++ Report 24/2/2000
http://www.adtmag.com/joop/carticle.aspx?ID=325

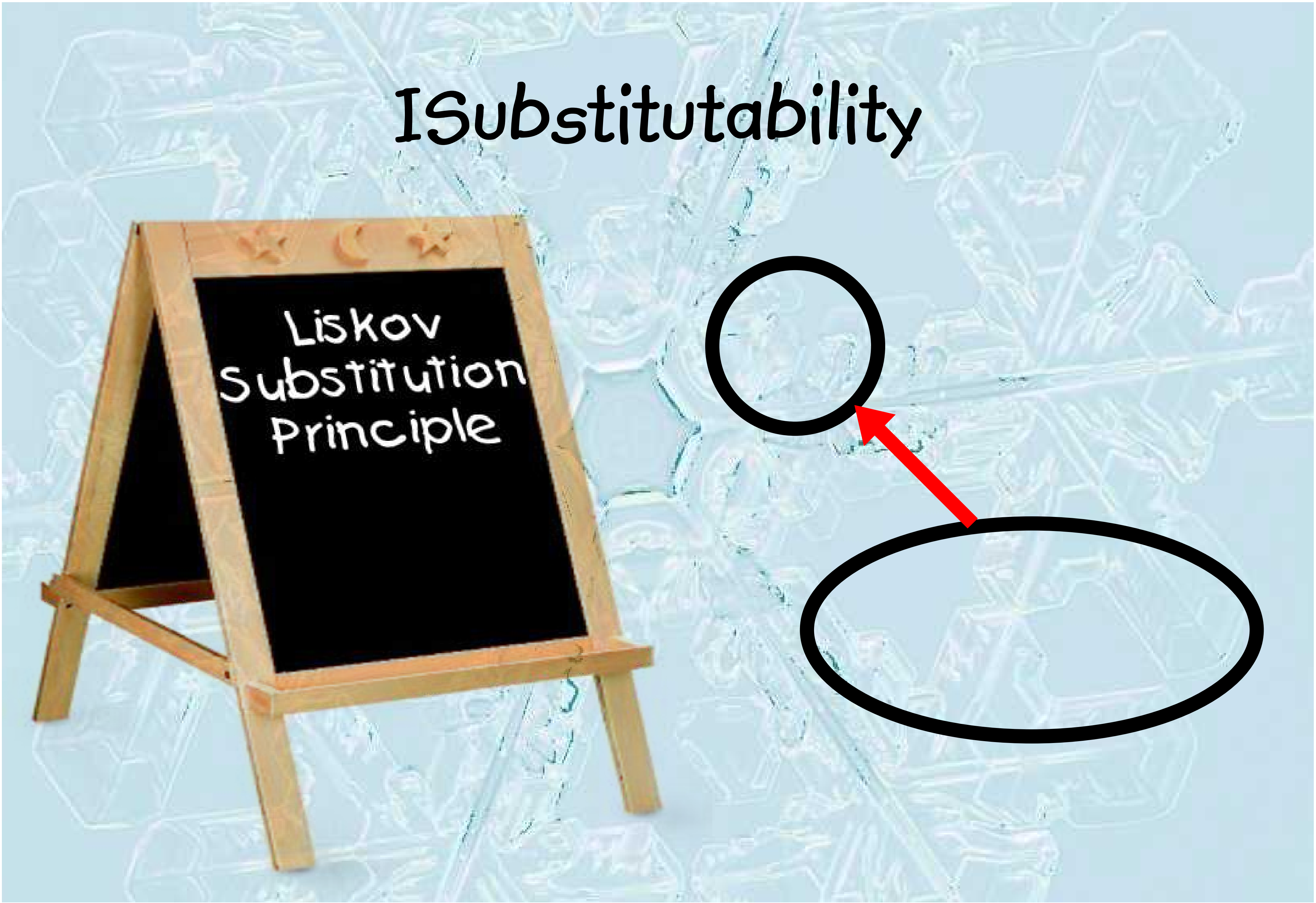
Unsingleton

- Give objects what they want, don't let them take it for themselves.
- What client code doesn't know about the implementation of a service can't hurt it.
- Don't sweat the small stuff – the answer to long argument lists is NOT Singleton!

Design to an Interface

- "interface" keyword
- COM/CORBA – IDL
- C++ Pure Virtual Base Class
- Duck-typing : C++ templates, Ruby, Python
- C# and Java Generics

ISubstitutability



IInheritance

Polymorphism by Dynamic Dispatch (the conventional model)

Virtual Functions

IGenericity

Polymorphism by Generics (Duck typing)

Containers

Iterators and algorithms

Traits and Policy classes

IOverload

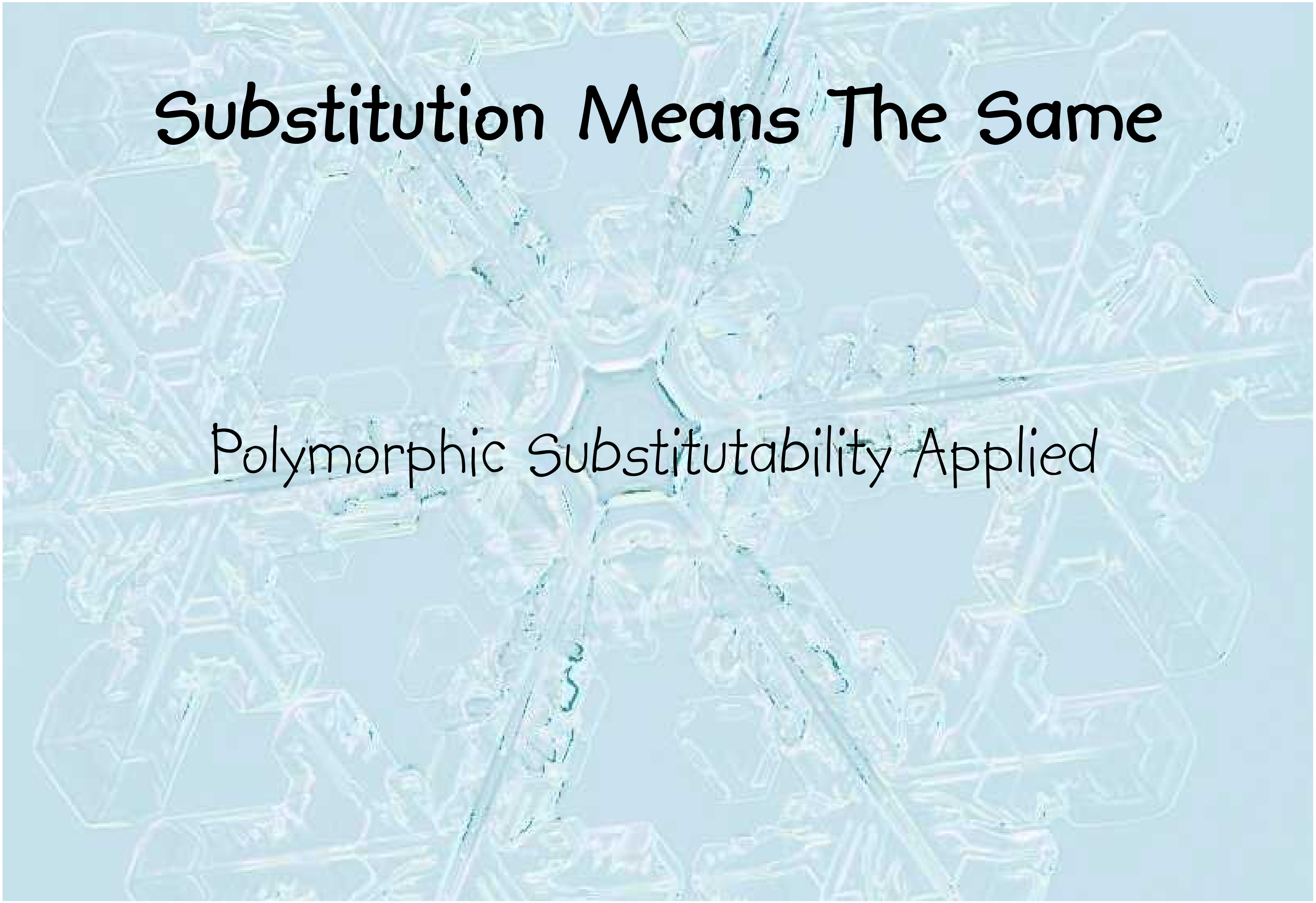
Polymorphism by Overloaded Functions

Member Function Overloading
Global functions and operators

ICoercion

Polymorphism by Conversion

Implicit Casting
Constness



Substitution Means The Same

Polymorphic Substitutability Applied

Testability

Test Independent Parts Independently
Don't get hung up on the small stuff
Interfaces == Substitutability
Substitutability underpins Mockery
Design to an Interface

Parallel Development

Define an interface for the component(s)

All teams/developers work to the interface

Continuous Integration

Testing (again)

3rd Party Parallel Dev

- Define the interface you need
  - You can always "Adapt" it later
- Create a Mock for the unavailable component
  - Just like testing!

Adaptability

- New requirements arise
- The app and other components depend only on interfaces
- Plugging-in a new component *just like* an existing one is trivial

Flexibility, Generality and Reuse

- The false idols of OO?
- Interfaces provide the means of re-use
- Component architecture provides the means of flexibility…
  - Make it talk to some wire-feed instead of the UI
- and Generality
  - Use it in a different application or context

Segregation & Selfishness

Clients should not be forced to depend on interfaces or components they don’t use.

Focus design on what an object wants, not what it can use.

“Don’t Call Us, We’ll Call You”

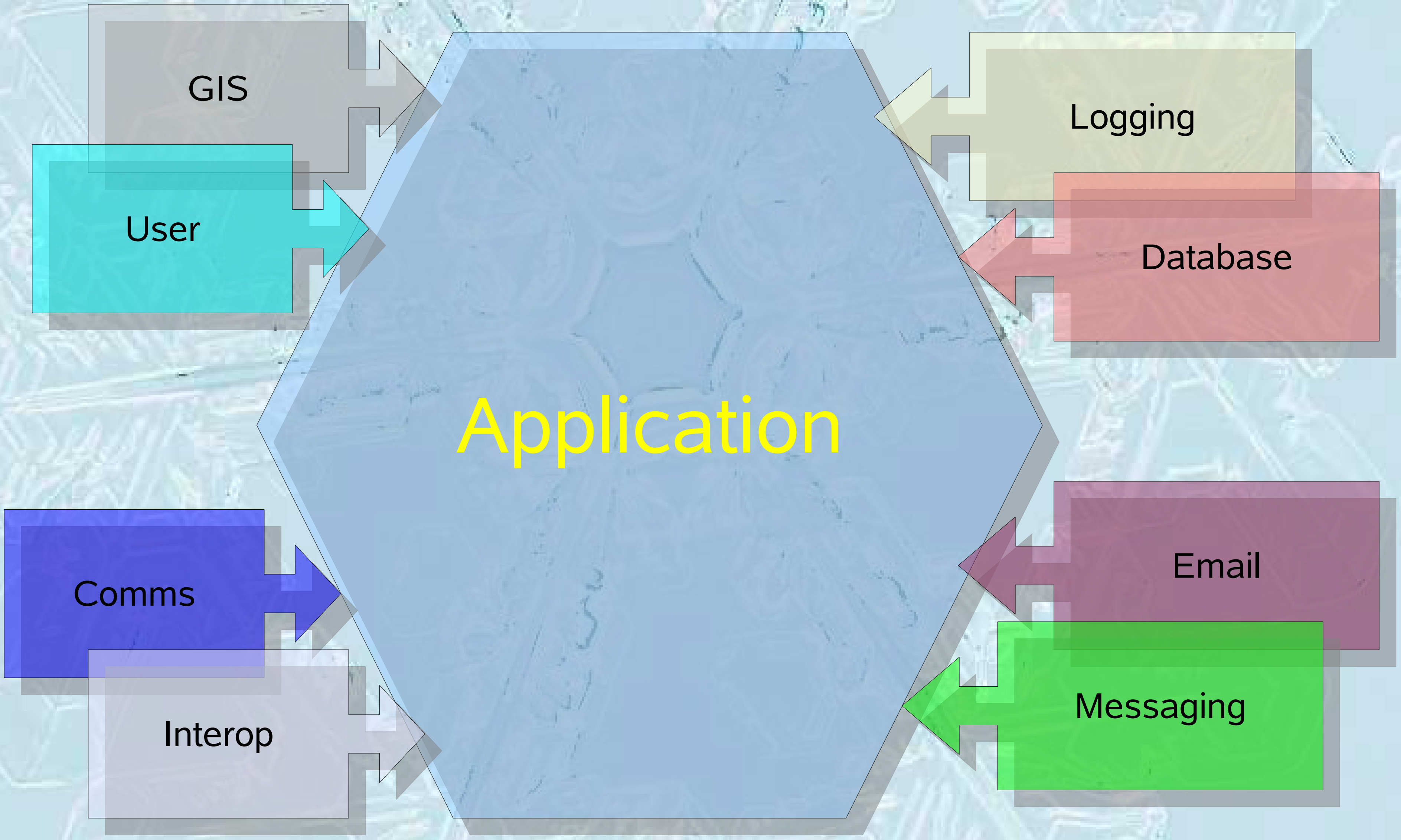
On Singleton

- Adapt the interface
  - Make an interface to publish
  - The implementation instantiates the Singleton
  - The rest of the app uses the interface

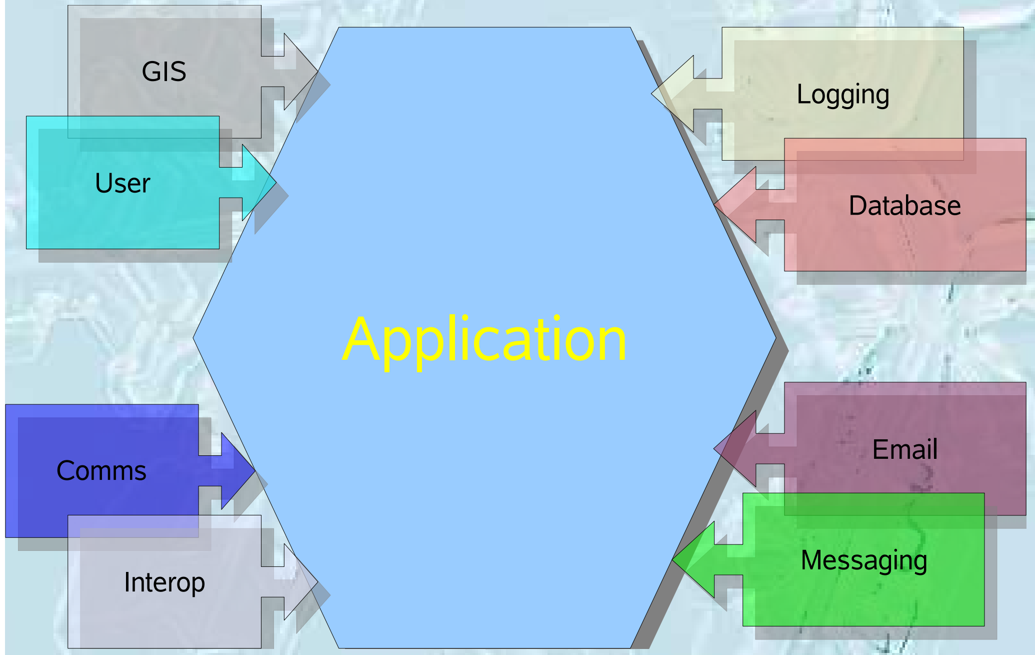
Dependency Injection

- Spring and its fellows
  - Sometimes a good idea
  - More often than not, sledgehammer for a walnut
- Service Oriented Architecture
- PfA is the root-solution

Hexagonal Architecture



Ports and Adapters



- A Port is an API
  - Exposed by the app
  - 1 or more interfaces
- An adapter is a component
  - Plugs into a port exposed by the application.

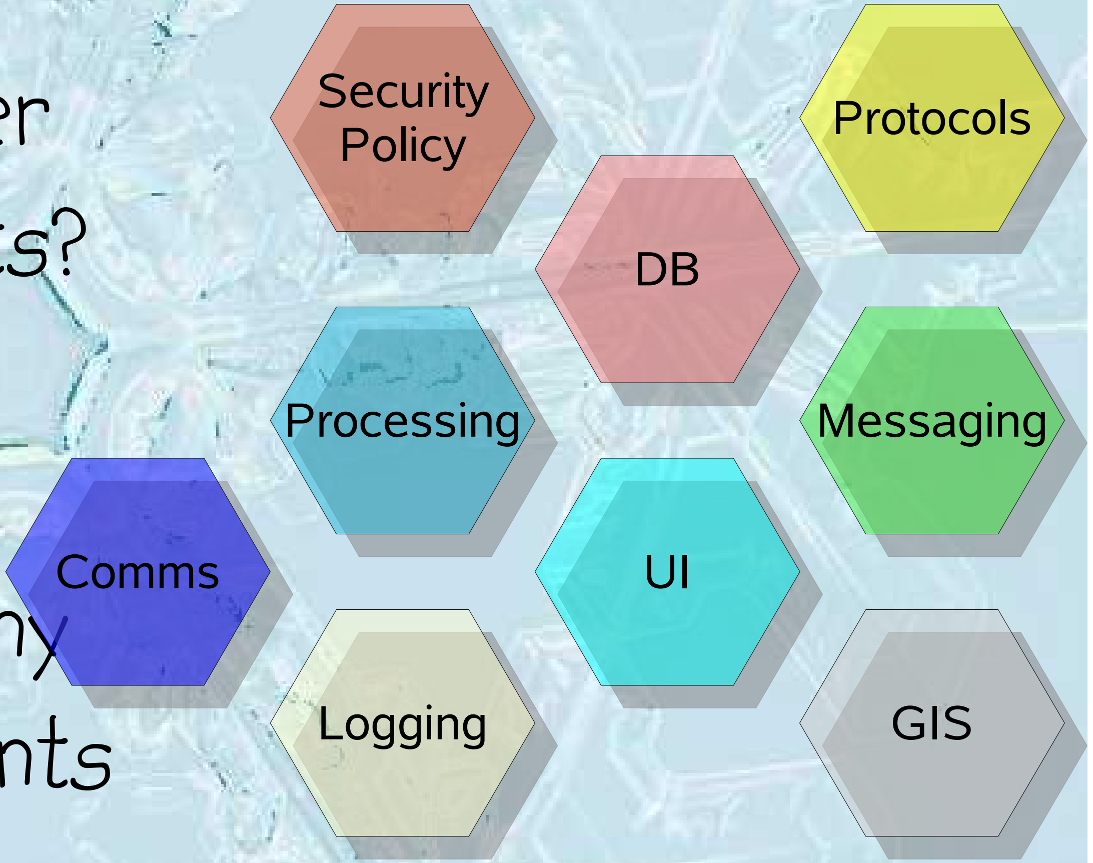
Adapters

- Are *substitutable* for adapters fitting the same ports
  - A mock database instead of a real one
  - A batch script instead of a UI
- May well be plug'n'play
  - But that depends on many things – most importantly, is it *sensible*?

A Step Further

What if each adapter exposes its own ports?

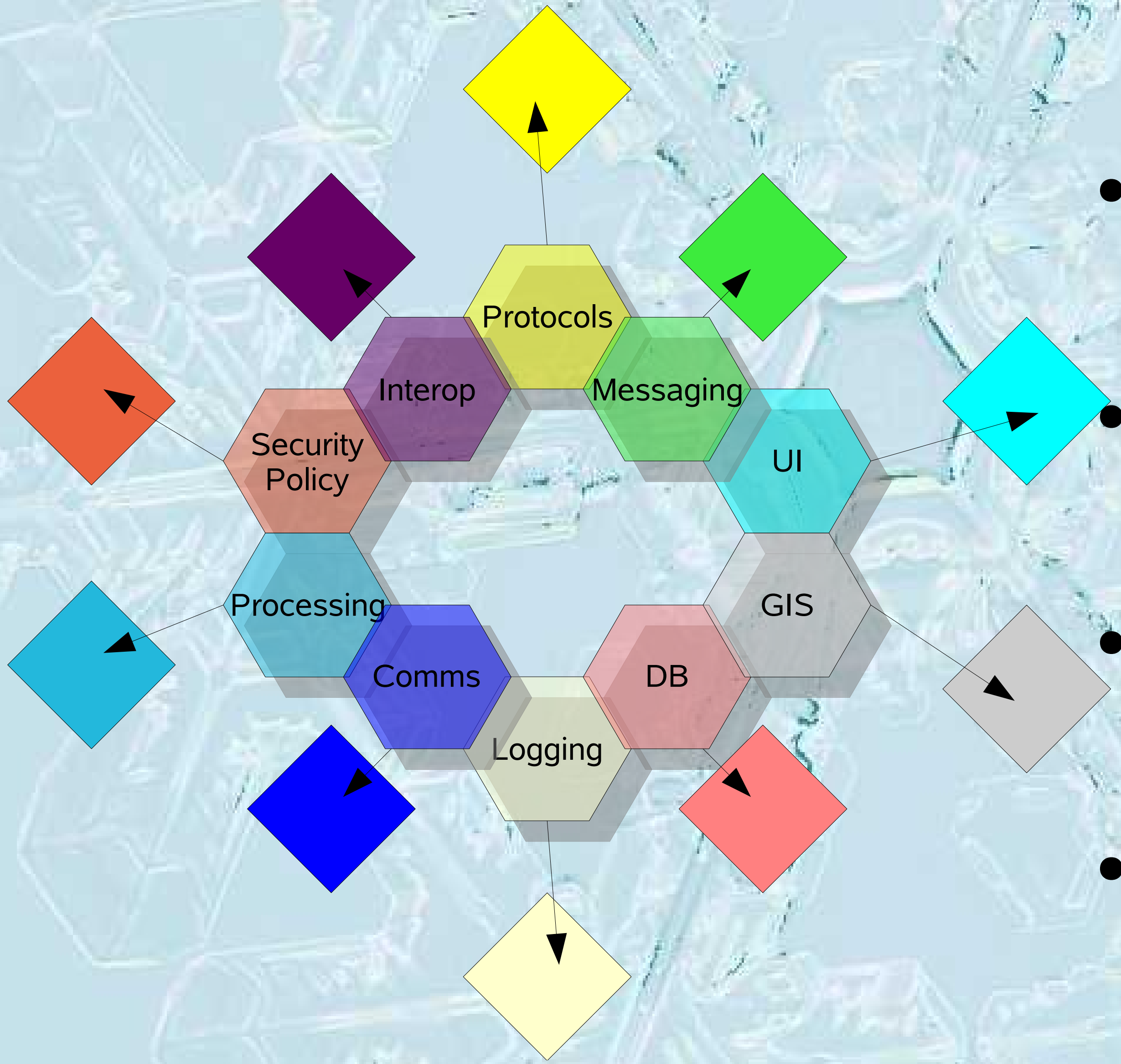
Any component can communicate with any other – the components *become* the API



Back to Circular Dependencies?

Interfaces form the separation points.
Circular dependencies mean the design is wrong (still)!

Snowflake



- Solid diamonds are implementations
- Hexagons inside are interfaces
- Diamonds depend ***only*** on hexagons.
- Hexagons depend ***only*** on other hexagons

The Return of the App

Somewhere there needs to be a part of the program which creates the concrete instances of the adapters and manage their lifetimes with respect to each other.

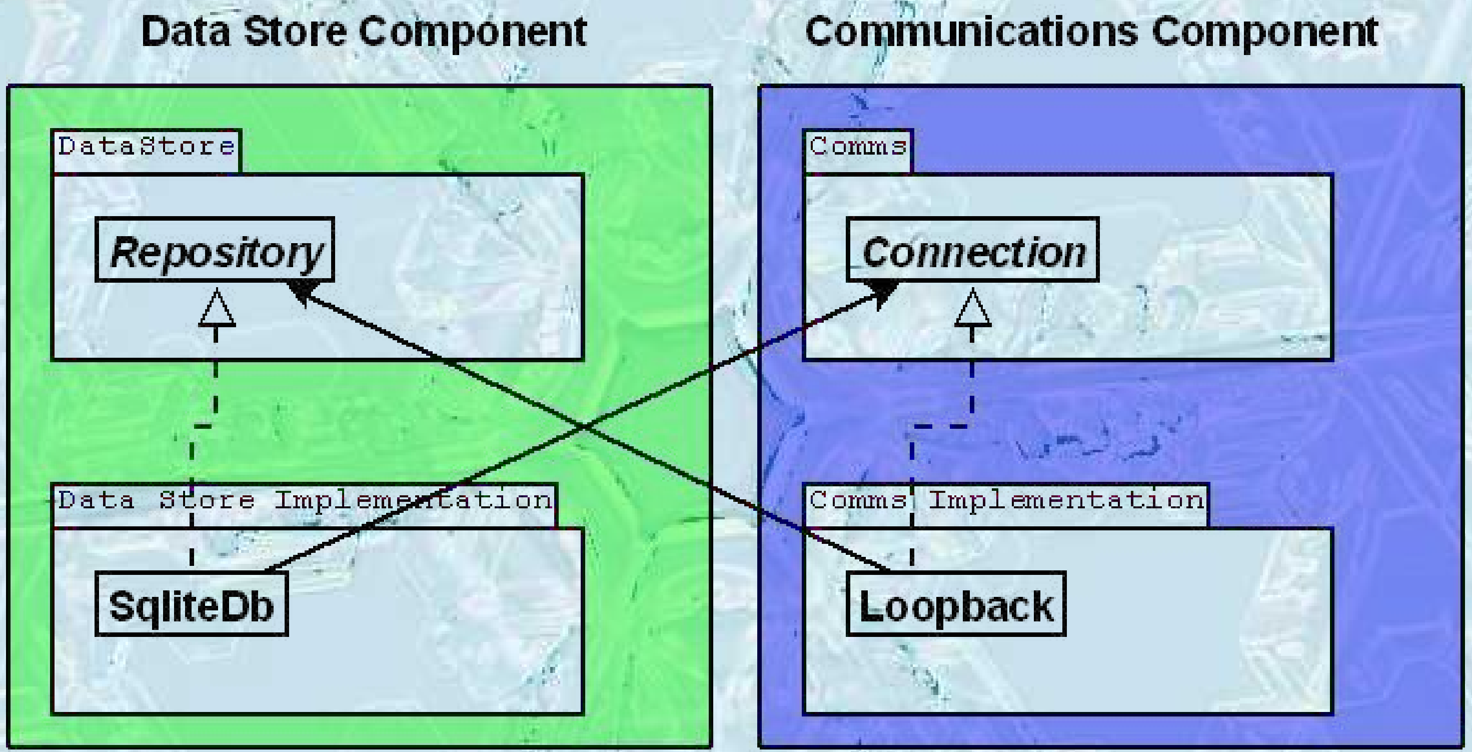
Substitutability Redux

App can choose which implementations to instantiate – tests, real, alternate

The Main Attraction

"Main" is the application.
Manages object instantiation and lifetime.
Maybe another level of indirection (e.g. Spring)

Project Organisation



Each component consists of interface and impl.
Separate project for each (static lib/assembly)

In C++

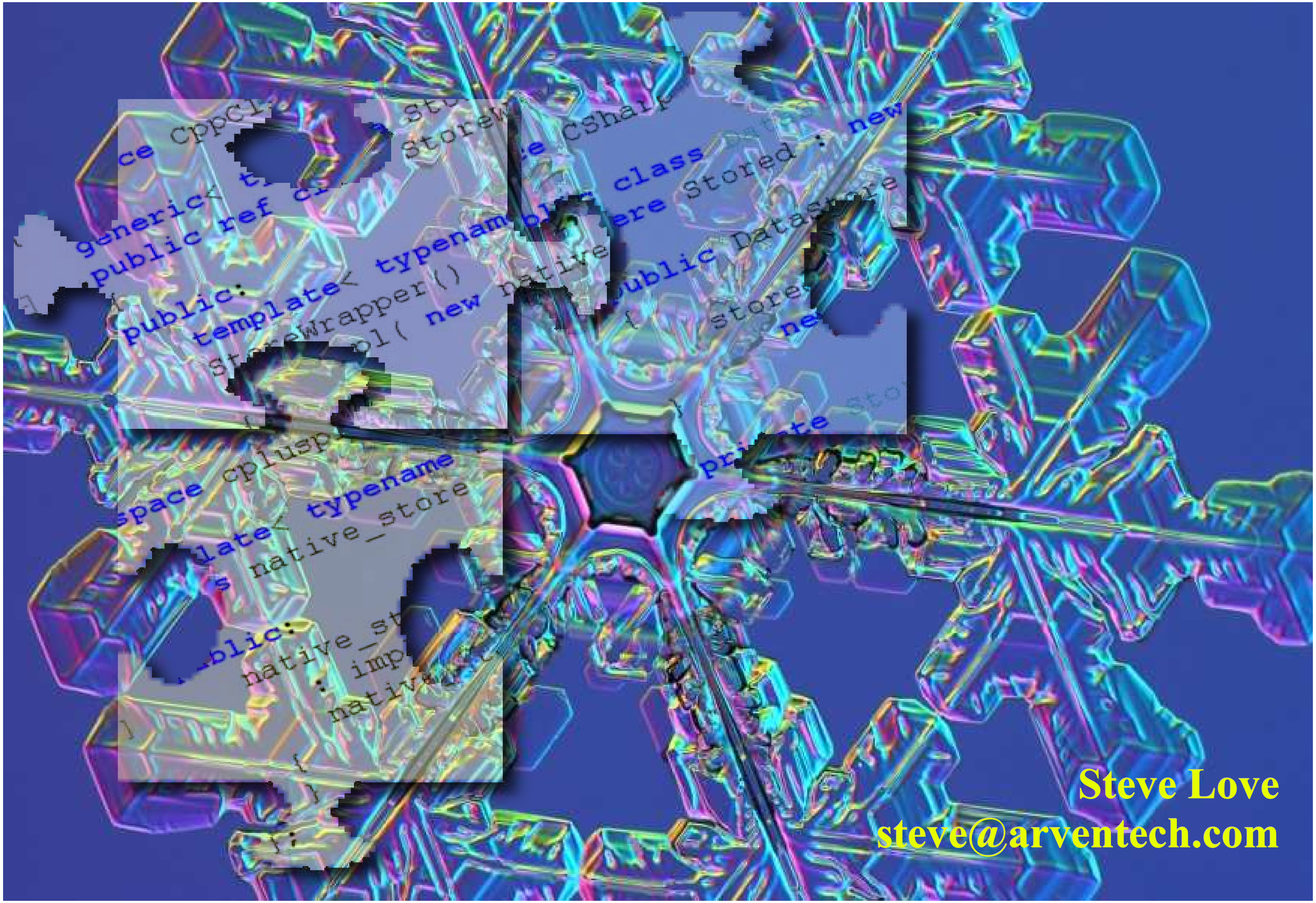
```
int main()
{
    auto_ptr< logging > log ( new file_log );
    auto_ptr< comms > c ( new net_comms );
    auto_ptr< messaging > msg (
         new buffered_messaging( log.get() ) );

    msg->send( c.get(), message( "Starting" ) );

    return 0;
}
```

In C#

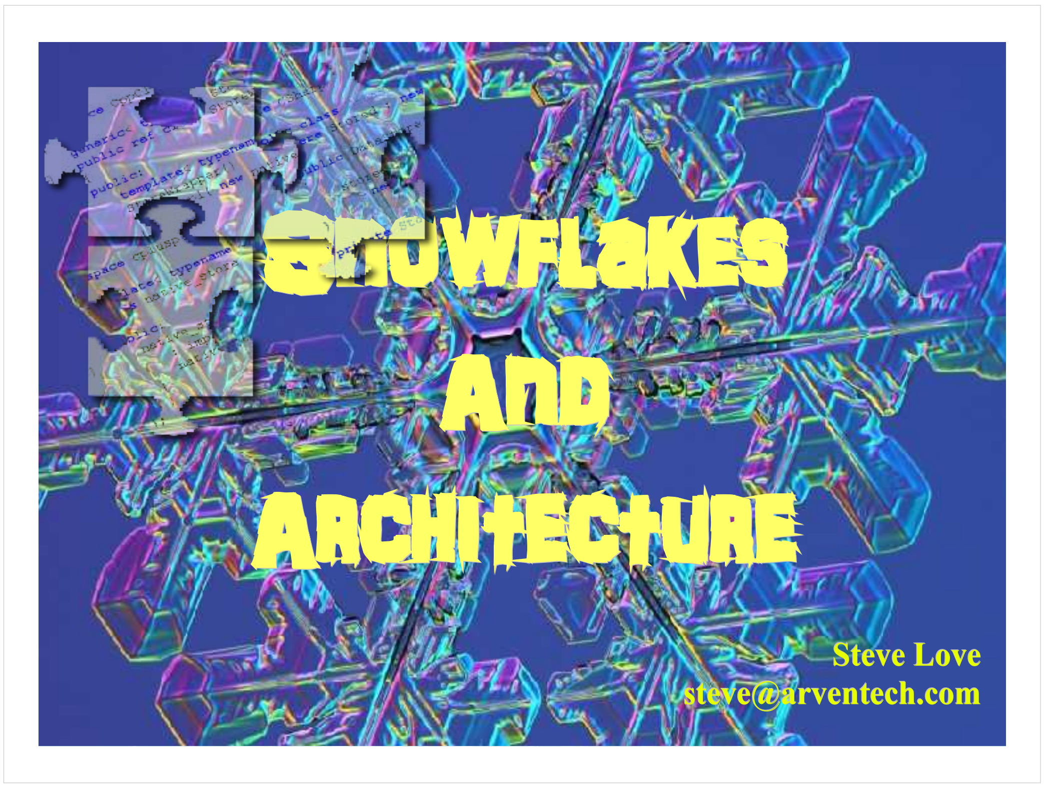
```
int Main()
{
    using( Logging log = new FileLog() )
    {    using( Comms comms = new NetComms() )
         {    using( Messaging msg =
                        new BufferedMessaging( log ) )
              {
                        msg.Send( comms, "Starting" ) );
              }
         }
    }
    return 0;
}
```

Paramaterise From Without

Define abstractions
Program to interfaces
Interface Segregation
Selfish Objects



Steve Love
steve@arventech.com



Snowflakes and Architecture

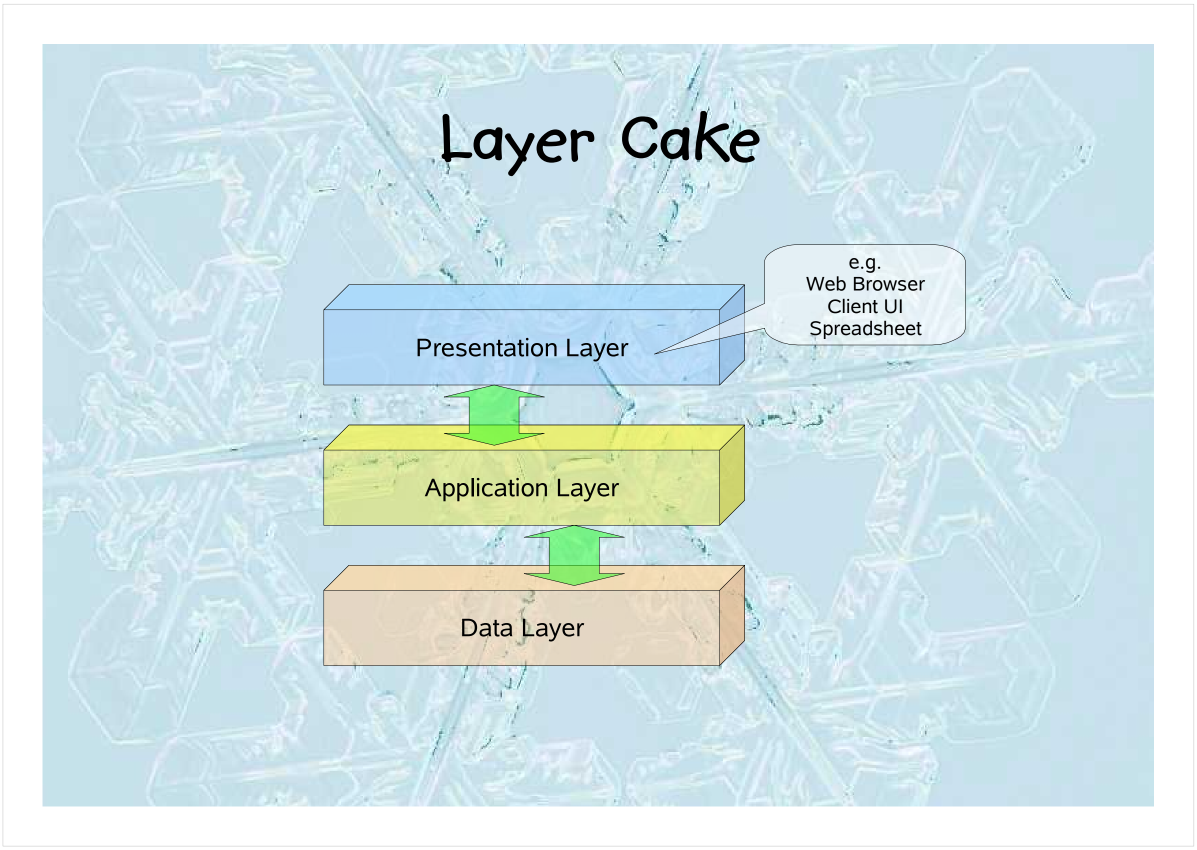
Steve Love
steve@arventech.com

This talk is about architecture and design in software.

Topics covered include the roles of architecture and design, application layering, what exactly is meant by “application”, and how the granularity of a design model influences its implementation.

We will look at how best to capture some of the abstractions of an application, and how they can be easily represented by abstract classes or interfaces. We'll also see how software designed this way is more decoupled, which helps testing, and in turn how designing for testability helps design.

Finally, all will be revealed about why the talk is about snowflakes!

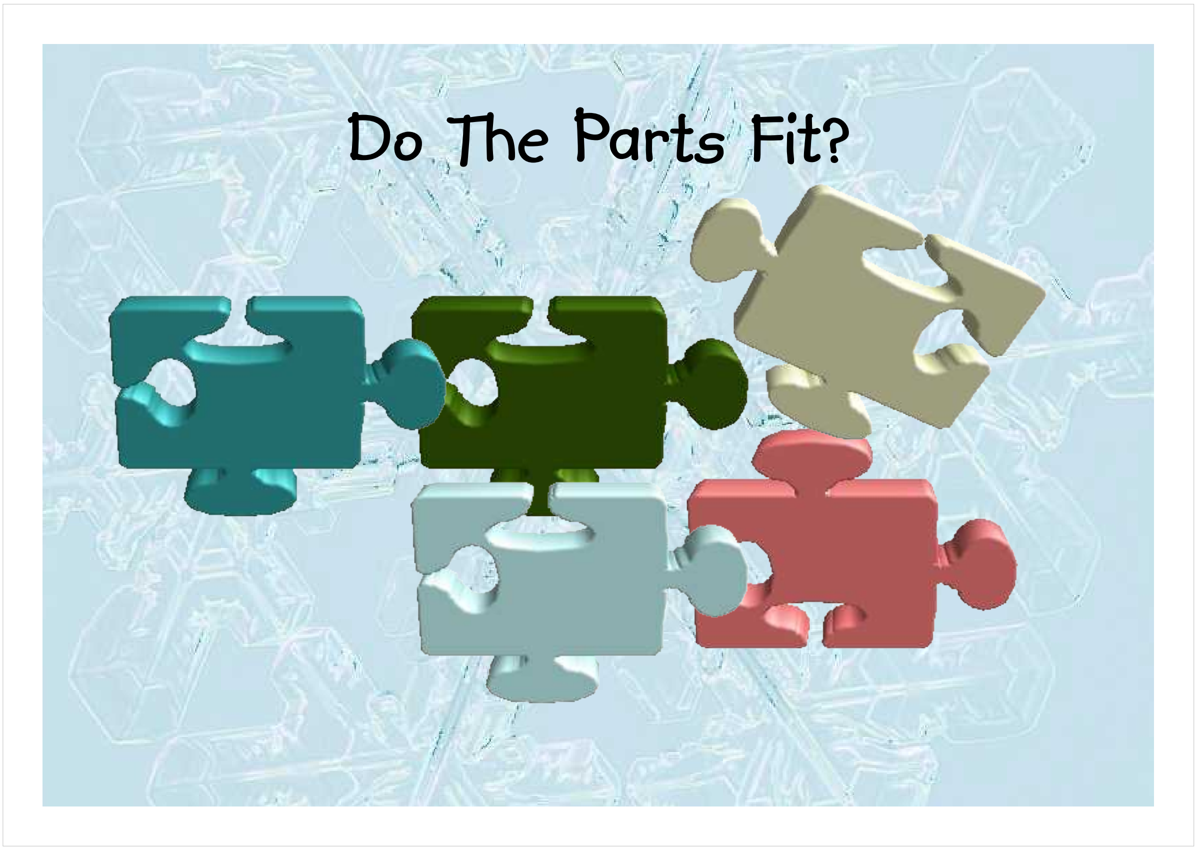


It's probable that most programmers know this diagram, or one very much like it. It represents separation of concerns between the major components of a system, and is not necessarily a UI, a DB and a “Business Logic” layer.

This model is very course grained, and very often software architecture only goes as far as a diagram like this. It does not capture the different aspects of what an application is, which is much more important, and this is what leads to spaghetti code and the “Big Ball of Mud”.

In addition, it's long been an industrial-strength problem that business logic leaks into UI and Database code layers, which leads to untestable, unchangeable software which *must* be attached to the database and can *only* be driven from its UI.

So why should the UI and database get special treatment?


Do The Parts Fit?

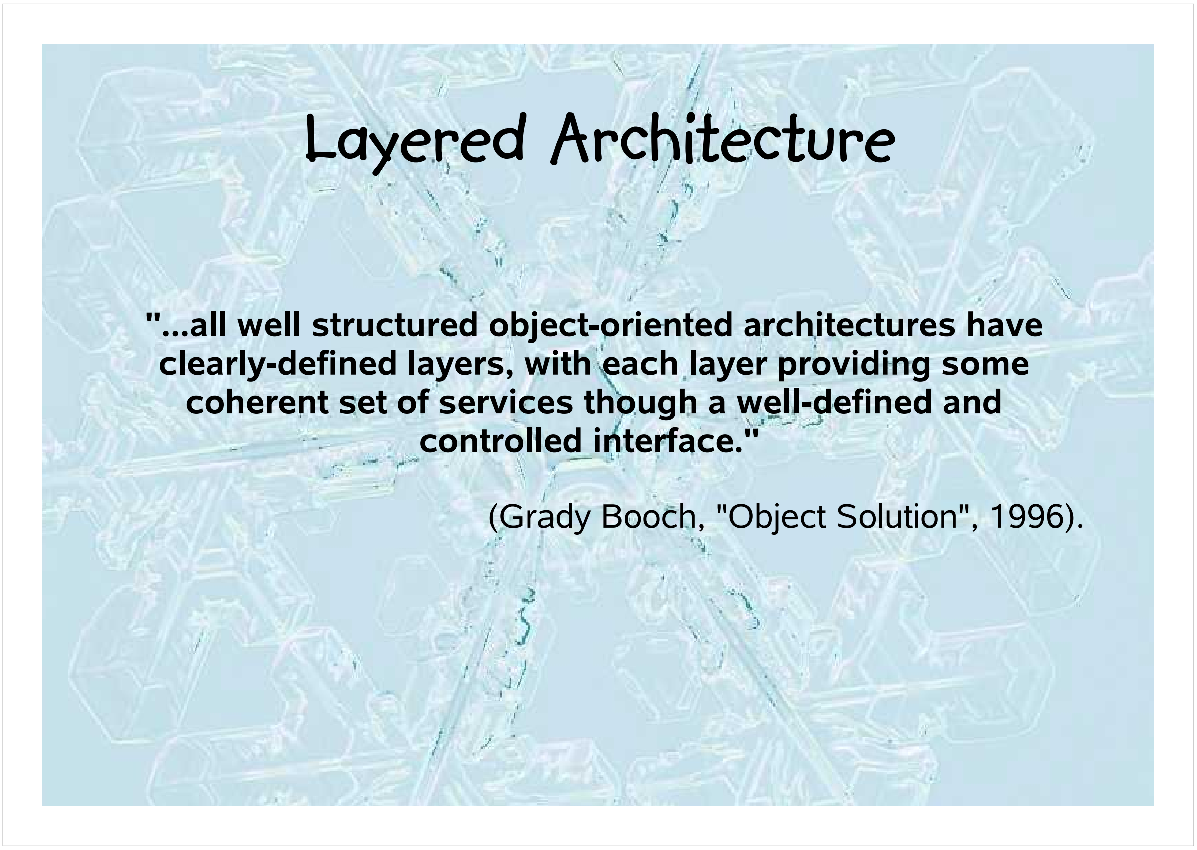
The layer cake design also presupposes that the presentation layer must always communicate with the data layer via the application – that somehow the application is the right abstraction for this. What if it makes perfect sense for some direct communication? The application can end up being no more than an artificial device for the UI to get data from a database, or a screen from an instrumentation monitor.

The 3-tier architecture may not be appropriate for your application. Ultimately, software architecture is about much more than a drawing – but all too often, that is what is referred to as “the architecture”.

By having a model that represents high-level abstract modules (the UI) looking down through progressively more detailed layers, we presuppose that the application really *is* decomposable that way. What results is unnecessary complexity, and in extreme cases, code that subverts the intended design – just to get the data.

In this talk we'll look at a different approach to capturing and describing the architecture in a much more fine-grained way, resulting in a much more open and extensible system – by design.

In the end, there are more than 3 ways to skin cats, cook eggs, and design software.

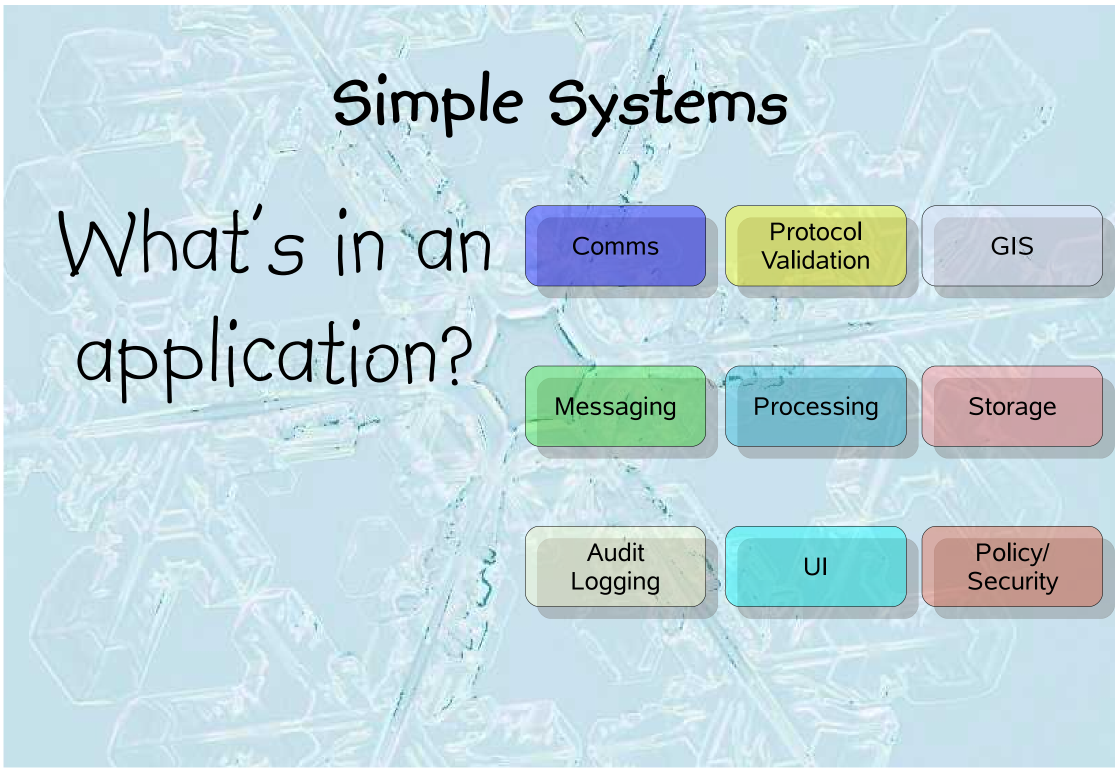


Layered Architecture

"...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface."

(Grady Booch, "Object Solution", 1996).

The separation of concerns into layers is not entirely redundant. It is the granularity at which this separation – the layering – is done that is commonly the problem. If you end up with 3 balls of spaghetti, that is hardly better than one big one!



Simple Systems

What's in an application?

Comms
Protocol Validation
GIS
Messaging
Processing
Storage
Audit Logging
UI
Policy/ Security

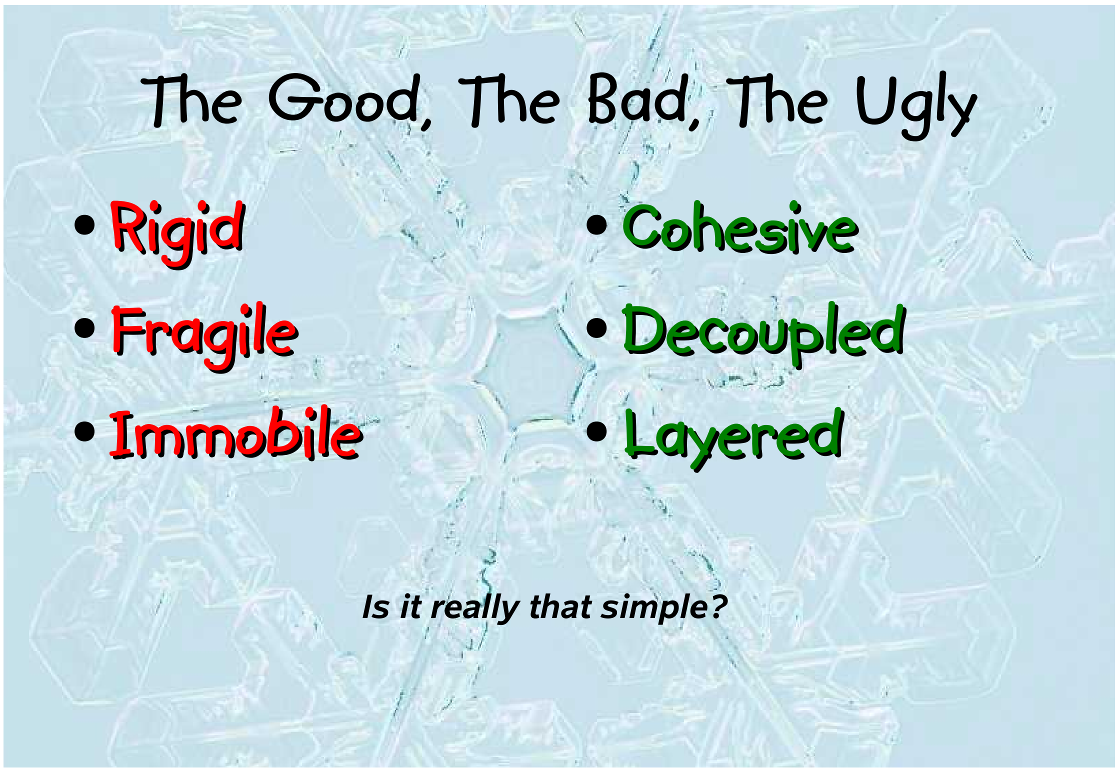
The 3 tier architecture has this thing called the “Business Logic”, “Application Layer”, or whatever, but what does that really mean?

Applications generally are made up of several – for want of a better word – services, communicating with each other, and quite separate from the UI and the Database, yet still part of the whole.

Instead of layering the architecture, it makes more sense to separate the services as their own abstractions, to capture the significance of various aspects of the application's needs, for example message handling, network communications, protocol handling, outputs *other* than a window on a UI.

Even simple systems have some inherent complexity, to a lesser or greater extent, and actually capturing that (necessary) complexity is helpful in characterising the services needed.

Of course a separate question is needed about whether the correct abstractions are captured. Saying an application needs TCP/IP communications over a wireless network is probably too specific – that communications method should have a higher level of abstraction so that the app can interface with it sensibly at the API level.



The Good, The Bad, The Ugly

- Rigid
- Fragile
- Immobile

- Cohesive
- Decoupled
- Layered

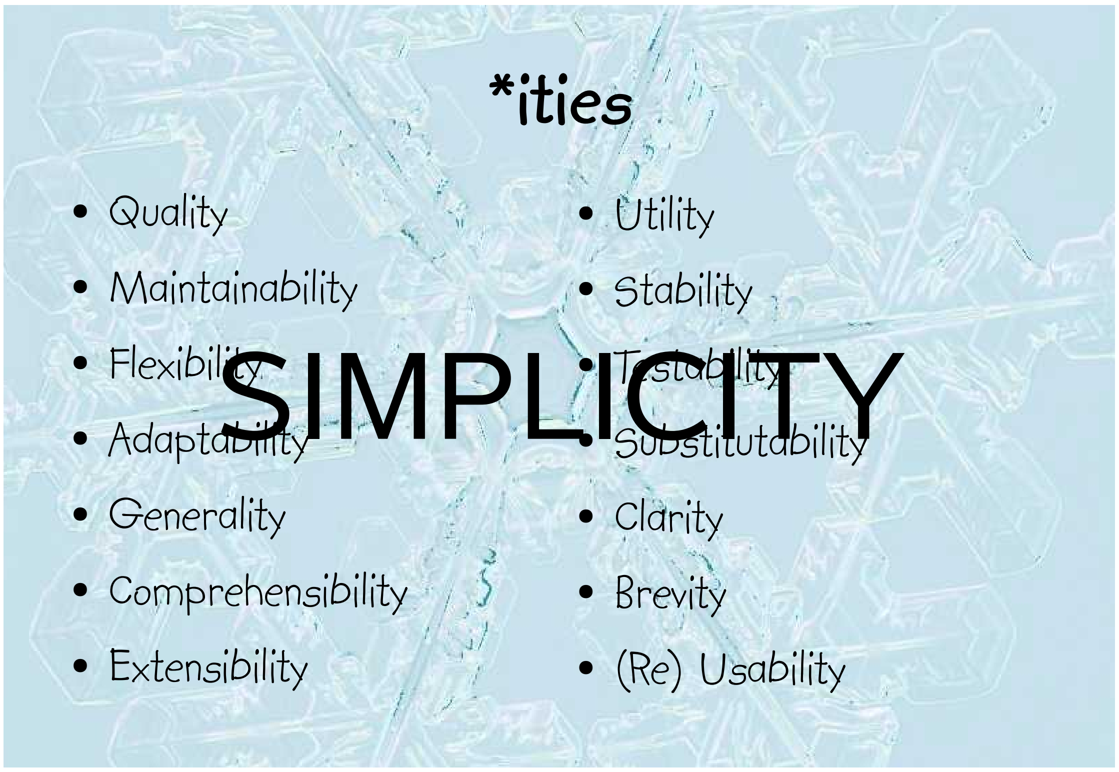
Is it really that simple?

So what makes a design good or bad? Robert Martin, in his paper "The Dependency Inversion Principle", describes *bad* designs as Rigid, Fragile and Immobile. Rigid in that the design is hard to change (and ultimately the realisation in code is hard to change, too) because there is a network of dependencies forming a rigid structure; fragile in that changes in one part of the system causes unexpected problems in separate parts of the system; and immobile because it's difficult to disentangle parts of the system for (re) use elsewhere.

By contrast, then, we look for cohesion, decoupling and layering in good designs. That's all very well, but how can we measure them?

Identifying good and appropriate abstractions leads to cohesive components. Decoupling comes from ensuring that components have a clearly defined purpose (cohesion) and stick to it. Layering is about how an application's deployment is realised, for example client-server or highly distributed applications have boundaries beyond the 3-tier model we've seen.

But can it really be that simple? The idea of cohesive, decoupled components is certainly not new, and yet it still appears difficult to achieve – spaghetti code and big balls of mud are recurring themes in the software development world.



*ities

- Quality
- Maintainability
- Flexibility
- Adaptability
- Generality
- Comprehensibility
- Extensibility
- Utility
- Stability
- Testability
- Substitutability
- Clarity
- Brevity
- (Re) Usability

SIMPLICITY

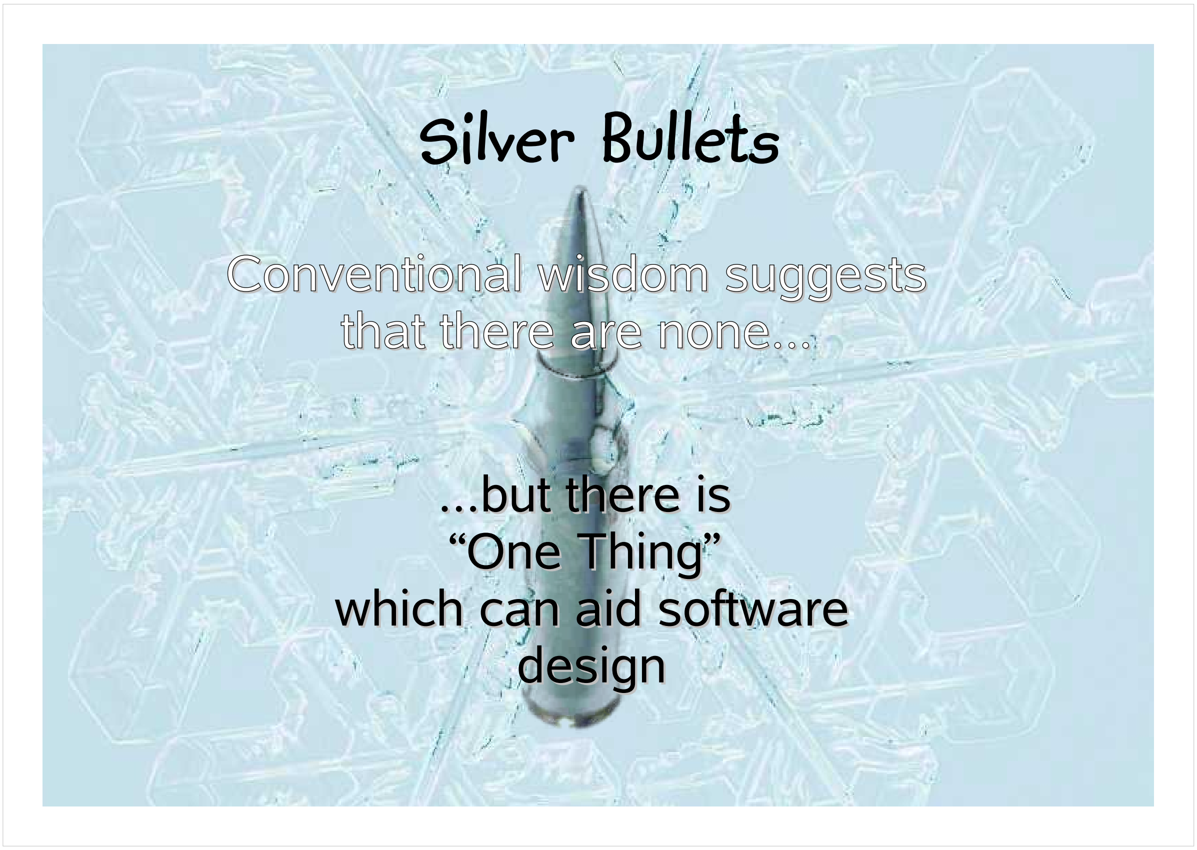
It's commonly thought that if your software design conforms to lots of these ities, the design is good. Certainly many of these aspects go into making for good design – testability, clarity and adaptability all spring immediately to mind as **very** important ideals. How are they measured? Quality is a very subjective notion, and re-usability is held up as the primary false idol of OO.

Generality is also not necessarily a good thing. The principles of agile design – and XP in particular – suggest making designs *less* general: you aren't gonna need it. Functionailty gets added only when it's required. Overengineering is a common mistake in many endeavours, not just software development.

In the end, brevity leads to clarity, substitutability to adaptability and testability to maintainability.

Is there one ity missing here? I'm certain there is – one which captures many of the others already shown.

Simplicity.



Silver Bullets

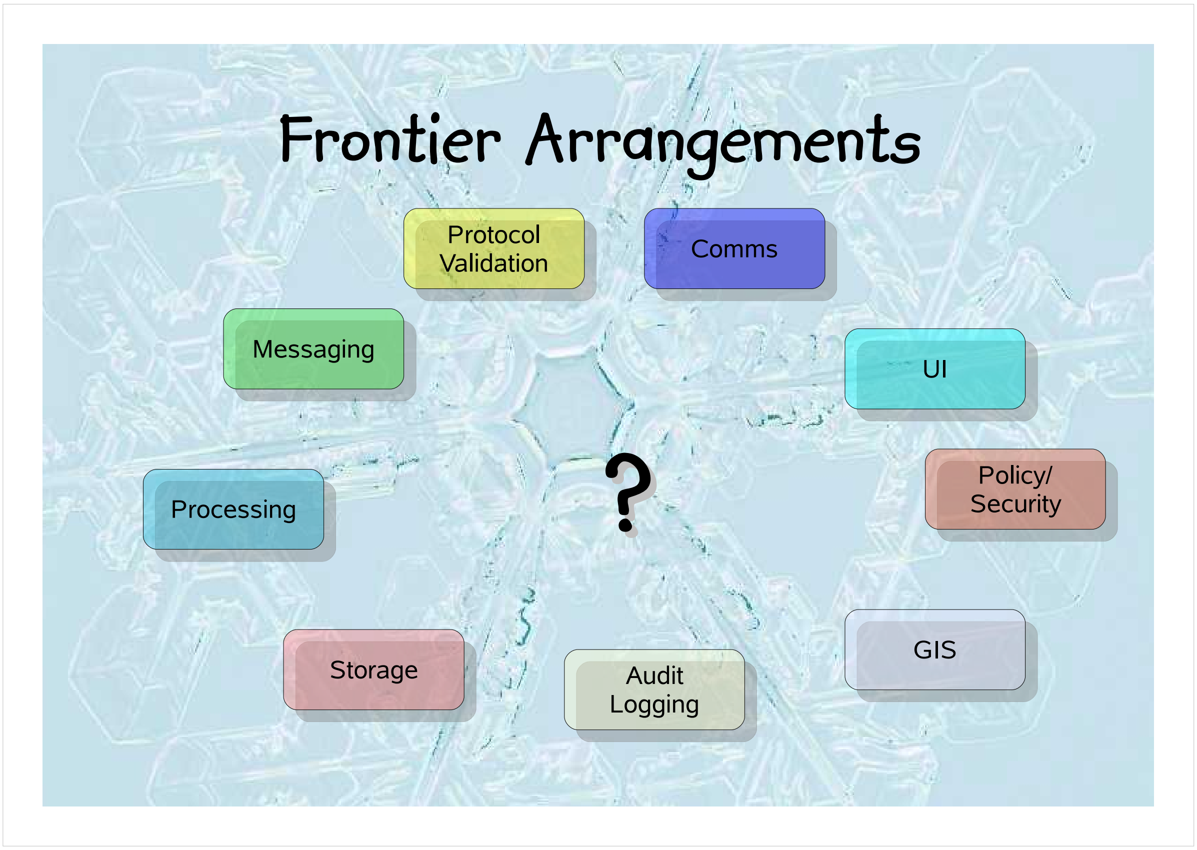
Conventional wisdom suggests that there are none...

...but there is "One Thing" which can aid software design

So is there one thing we can attack the problems of software design with? We all know there **are** no silver bullets – software design is not a werewolf to be slain, but a much more complex and detailed problem for which there is no one single answer...

If we think about getting the architecture right, then good design follows. With a good design, and a bit of due diligence and discipline we can implement the software according to the design...and voila! Magic software. Well, not quite. But it's a start.

Good architecture is indeed the first step in producing better software. In order to think about architecture, we need to take a step *inward* and start to see some of the details – the abstractions needed. There seems little point in creating a fabulous multi-tier, distributed (or distributable) **architecture** for a washing machine control program. By identifying the abstractions we need, this information feeds back into what *sort* of architecture we need, and so design influences architecture. There's more to it than that, of course – no silver bullets, remember? - but at the design level, if we identify the right abstractions at the right granularity we have gone a long way towards producing a system.

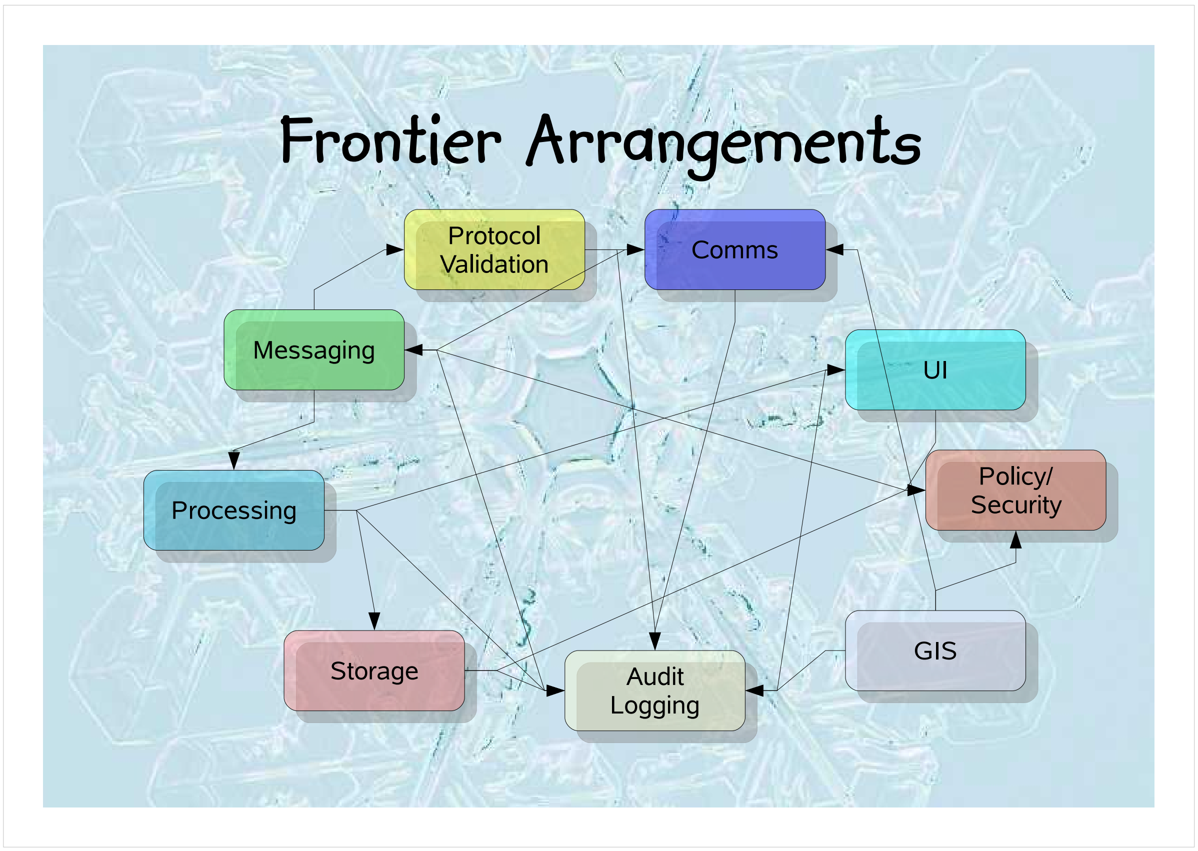


Frontier Arrangements

The right level of abstraction is a hard thing to achieve. We've already talked a little about what components might exist, and here are some examples. What we need to identify is how they communicate together to form an application.

Having identified our abstractions -we'll assume that the examples shown here are appropriate – the next step is to define the interfaces between them. A design may be highly cohesive and well decoupled, but if the components don't communicate, they aren't a system!

There must be some restraint, however...



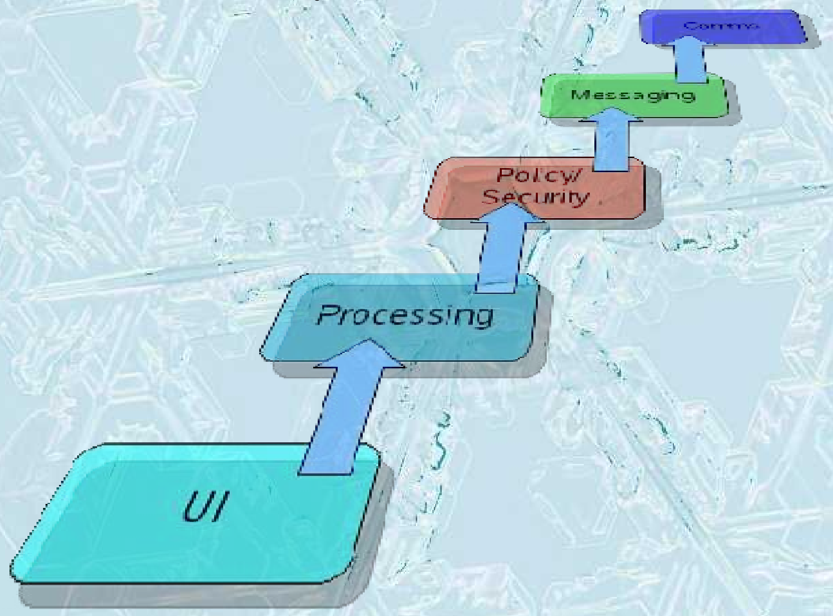
Frontier Arrangements

Maybe it makes perfect sense for each of the communication channels shown – and more! - to exist. In some cases, there may be a need for bi-directional information flows between 2 or more components, and this leads once again to a highly coupled design.

Here we have the exact opposite of a layered approach, which is intended to show a directional relationship between parts of a system – the presentation layer communicates with the application, which in turn communicates with the data layer, percolating data back *upward*.

The main issue that needs addressing in both the layered model *and* this component model is that of dependencies.

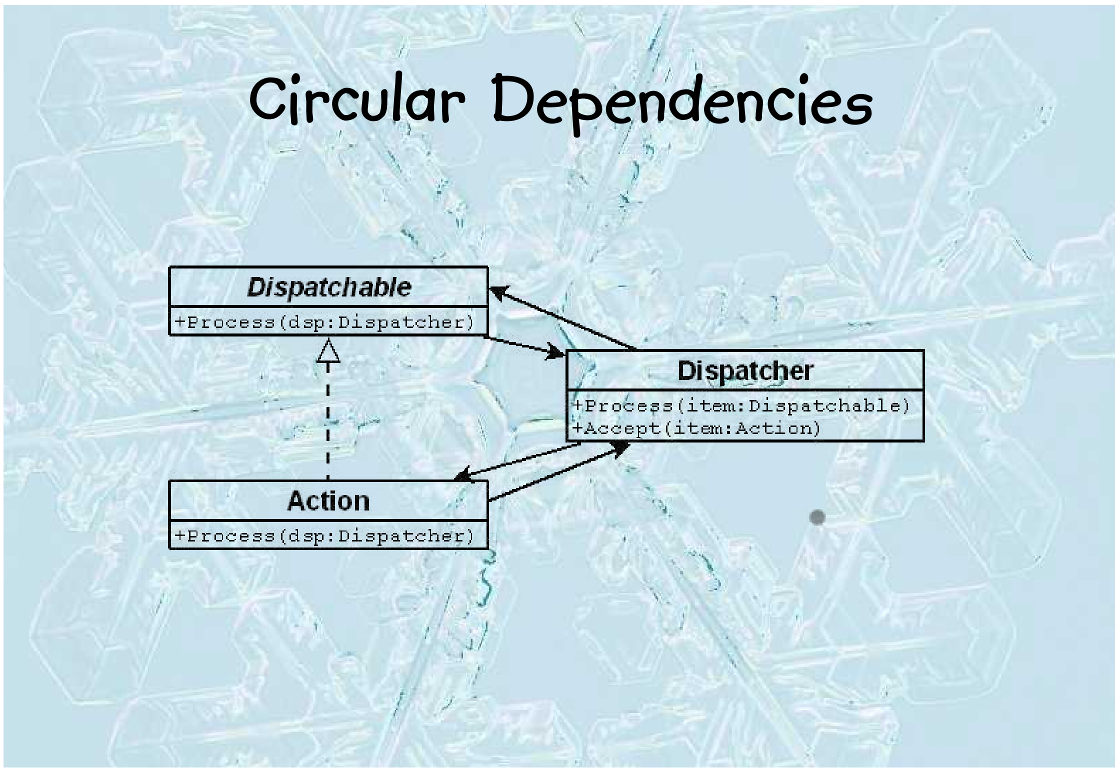
Dependency Horizon



Dependencies are transitive, which is to say that where component B depends on component C, anything which is dependent on B also depends upon C. This is called the dependency horizon, and clearly, the further away it is, the more tightly coupled a system is.

A depenency horizon which is very far away becomes extremely difficult to manage, and runs the risk of introducing circular dependencies which may be very difficult to resolve.

C++ with its separately compiled modules has facilities to mitigate the difficulty in breaking circular dependencies – forward declarations, handle-body idiom, etc., but in languages such as C# and Java, circular dependencies between units are forbidden, triggering compile-time errors.



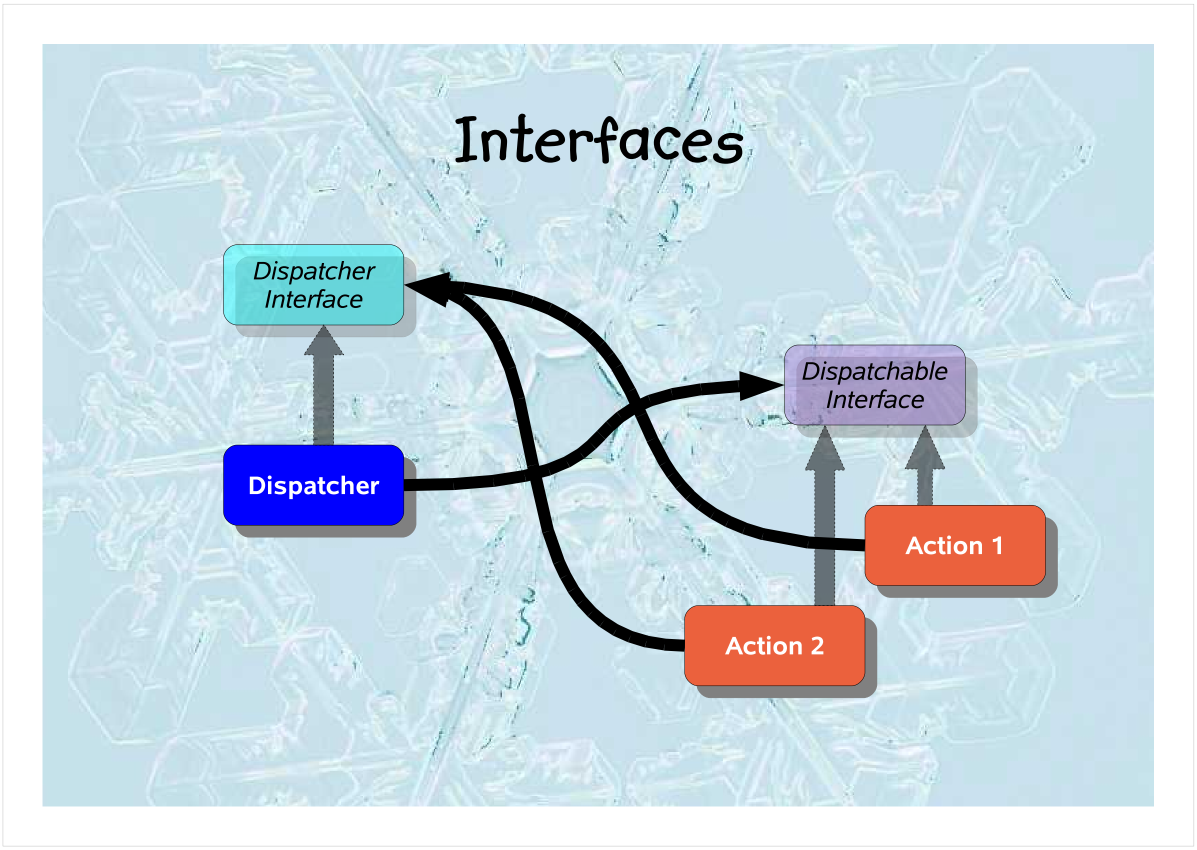
Circular Dependencies

The double-dispatch pattern is a prime example of a circular dependency. The main dispatcher object accepts other objects by reference to their base class, and asks the object to call it back on a method overloaded for the concrete instance of each type it knows about.

In this way, a dispatchable object has a dependency on the dispatcher itself, each concrete dispatchable object also has that dependency, along with the inheritance dependency on the base class, and the dispatcher object needs to know about (depends on) each concrete type as well as the base class.

Resolving this dependency circle can be difficult in a naive way.

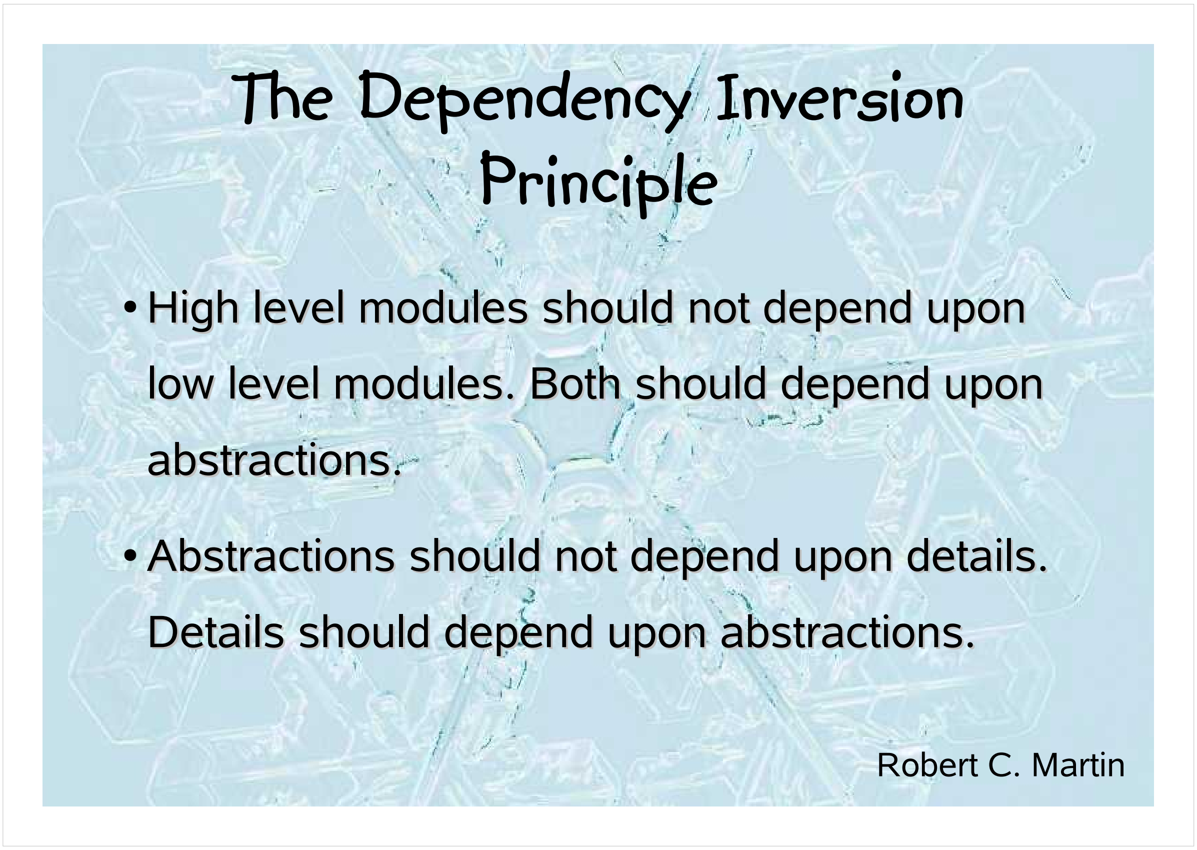
The answer to the problems here is to see the required abstractions, and to see where to add the proverbial extra level of indirection.



Interfaces

By introducing an interface for the dispatcher, the dispatchable base class (interface) requires a dependency *only* on that interface type. Each concrete action also needs to know about only the interface.

This is the key principle in breaking circular dependencies, and as a concept can be expanded to bring the dependency horizon much closer to an interested object.



The Dependency Inversion Principle

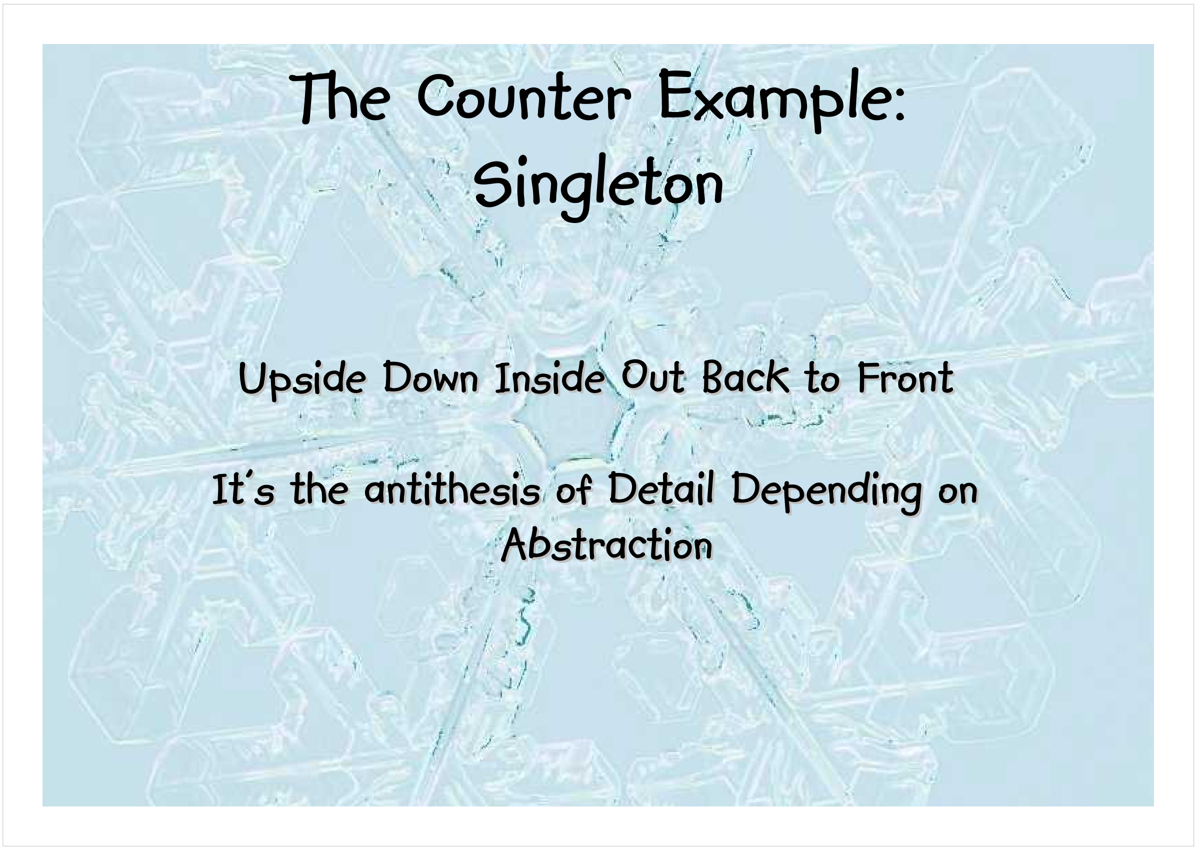
- High level modules should not depend upon low level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

Robert C. Martin

In his “Dependency Inversion Principle” Robert Martin describes the basic principles of managing the dependency horizon.

It describes an essentially layered solution, but not at the course-grained level we have previously looked at, more at the service level – the facilities required by a simple component which would form part of a whole application.

It maintains the idea that high-level modules should “look down” through successively more detailed interfaces, that details should filter upward through clean and cohesive APIs. The APIs themselves are presented as true interfaces, allowing the dependency horizon to be no further than a single dependency away.



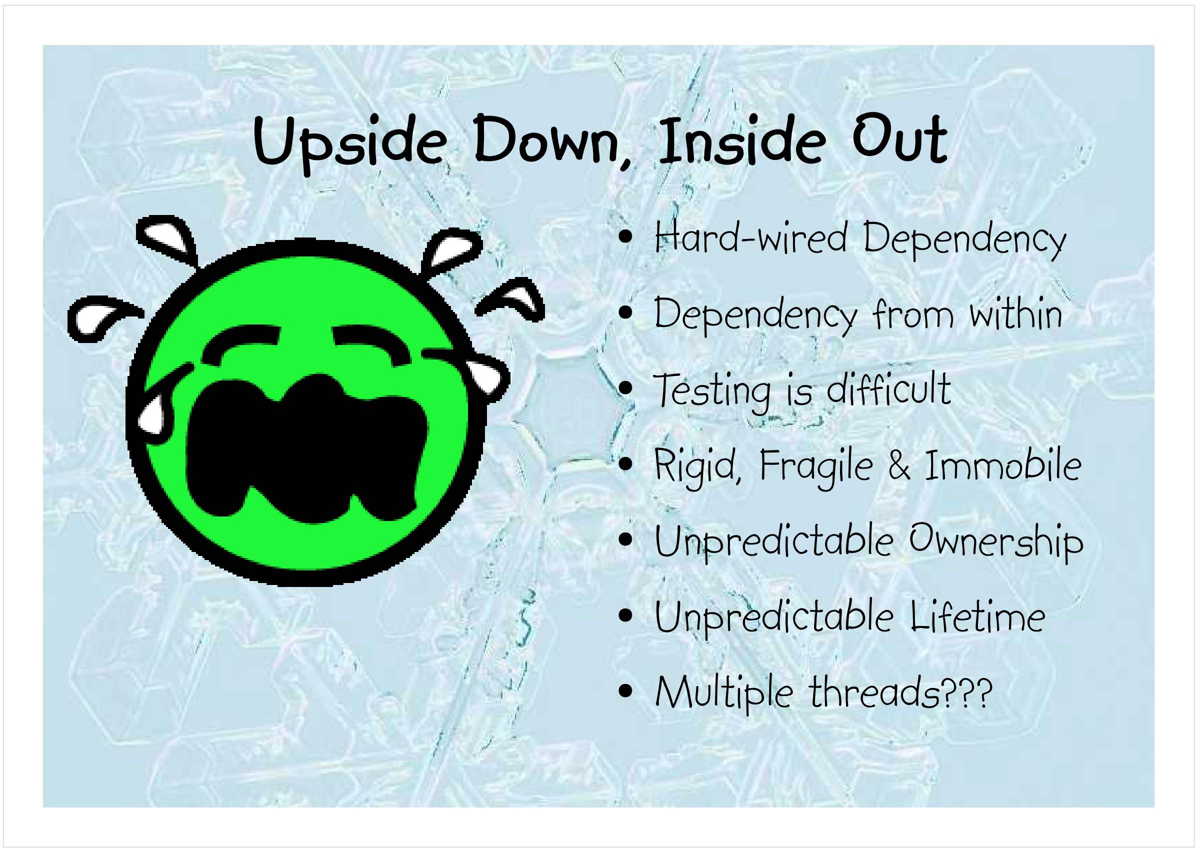
The Counter Example: Singleton

Upside Down Inside Out Back to Front

It's the antithesis of Detail Depending on Abstraction

By contrast, Singleton *is* a dependency representing detail – and it is a direct dependency from within implementation code. It is most certainly *not* abstract.

The main problem – if there is a main one – with Singleton is the reversal of dependency it introduces. Not only is client code dependent on the Singleton itself, the client code uses the Singleton directly, without any facility for abstracting the service(s) provided by the Singleton. (OK so there are ways of providing an abstract interface to Singleton objects, which we'll look at in a while, but that really is solving the wrong problem).



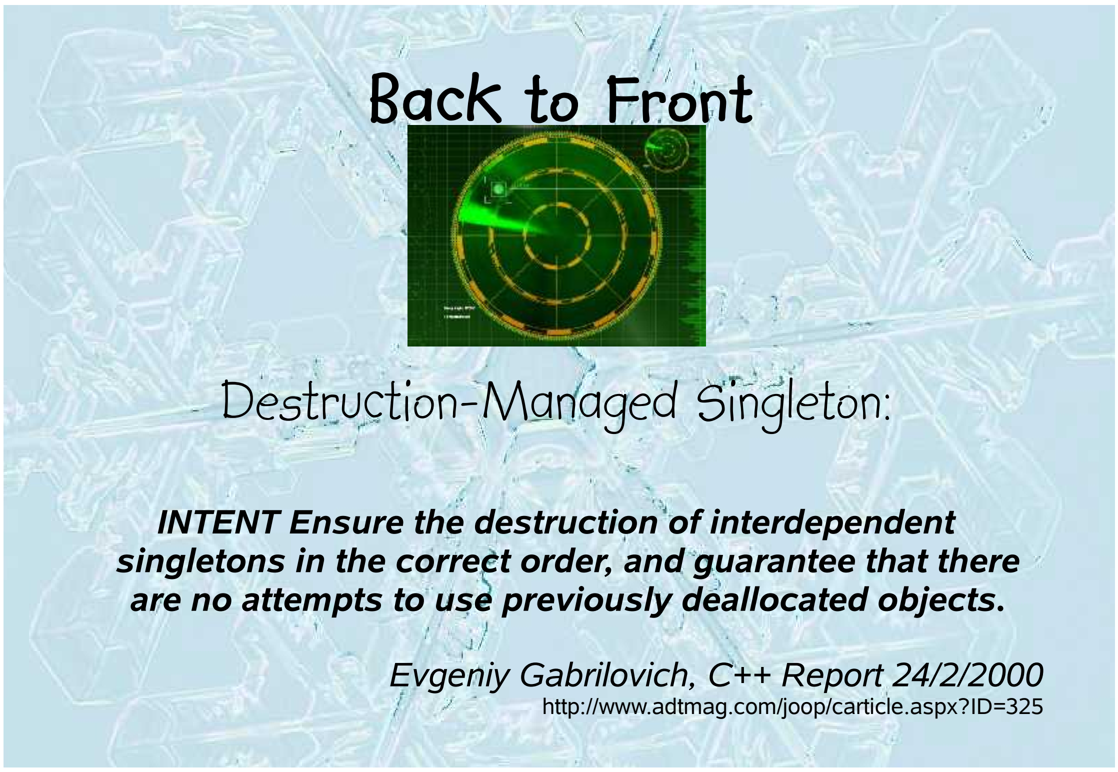
Upside Down, Inside Out

- Hard-wired Dependency
- Dependency from within
- Testing is difficult
- Rigid, Fragile & Immobile
- Unpredictable Ownership
- Unpredictable Lifetime
- Multiple threads???

The dependency introduced by Singleton is pernicious because it hides inside client code with very few clues to expose it. Client code that *uses* a Singleton object has the dependency hard-wired in to it, leaving few opportunities to break the dependency and have that code use something different.

This can make testing such code difficult, because it requires access to the Singleton, which will often represent some "big" aspect of a system – database, logging facility, UI, network – of which "there can be only one". Testing independent parts independently of all other parts of the system becomes *very* challenging.

The Singleton fails all three of Robert Martin's tests for bad design, and worse, introduces some issues of its own.



Back to Front

Destruction-Managed Singleton:

INTENT Ensure the destruction of interdependent singletons in the correct order, and guarantee that there are no attempts to use previously deallocated objects.

Evgeniy Gabrilovich, C++ Report 24/2/2000
http://www.adtmag.com/joop/carticle.aspx?ID=325

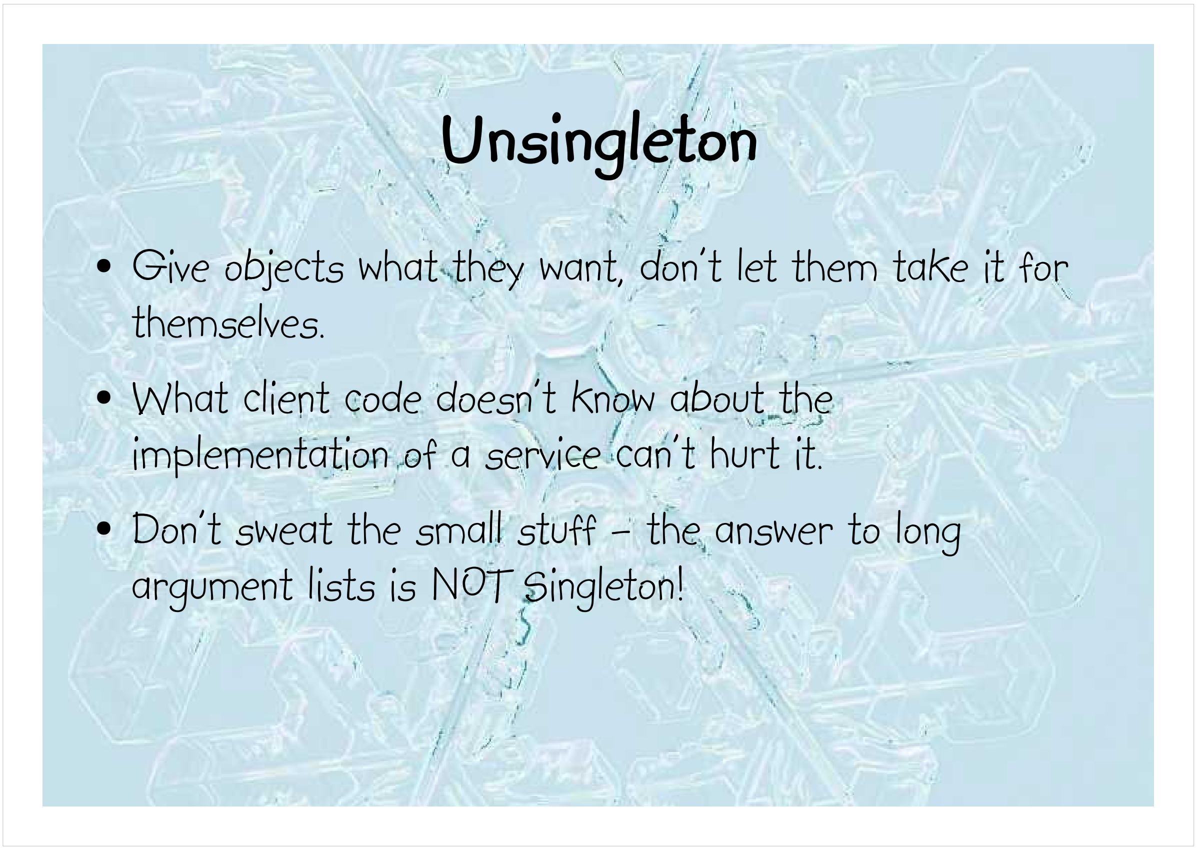
Interdependent Singletons?

Ensure their correct destruction?

Not using recycled objects?

Maybe this is a solution looking for a problem. The next step, of course – and one which is advocated and explored in the paper - is to make the DMS (Destruction Managed Singleton) a ... yep! - Singleton.

So who then manages the destruction of the destruction managed Singleton?



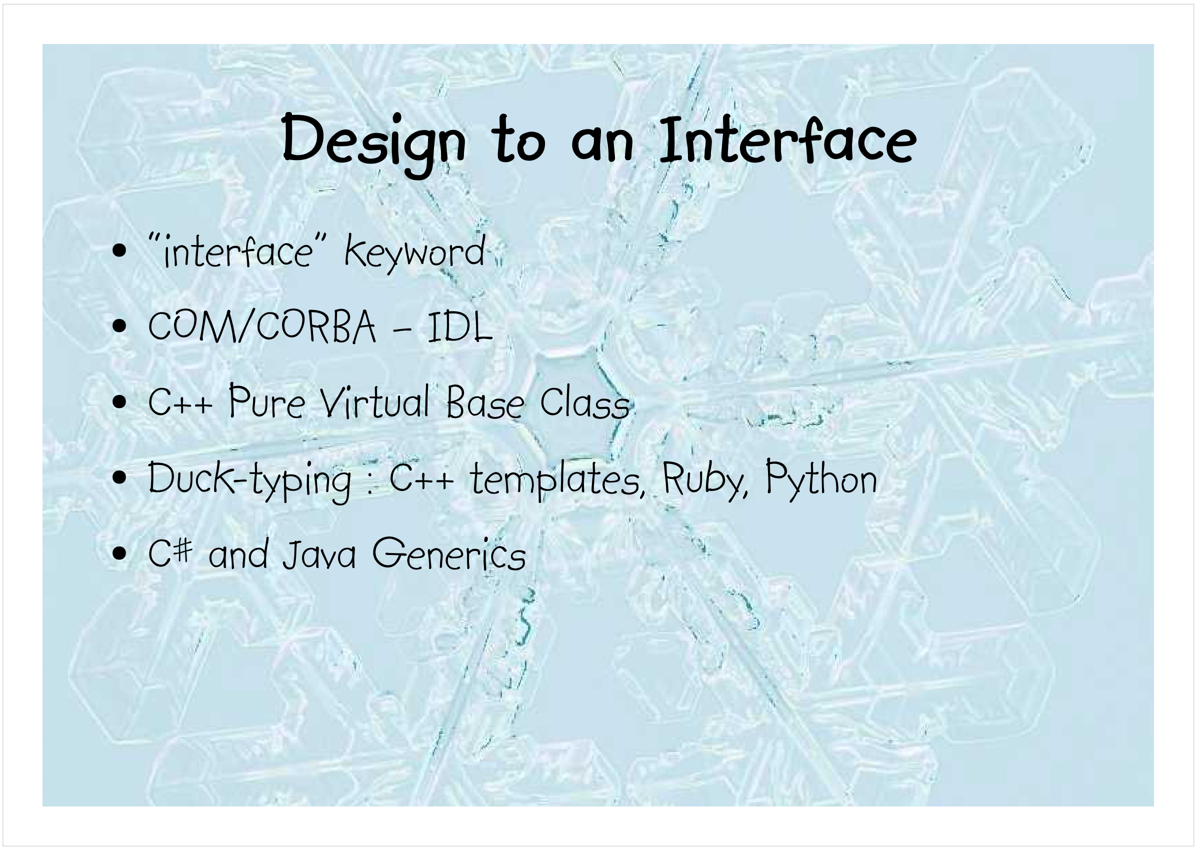
Unsingleton

- Give objects what they want, don't let them take it for themselves.
- What client code doesn't know about the implementation of a service can't hurt it.
- Don't sweat the small stuff – the answer to long argument lists is NOT Singleton!

Code should work using what it has been given, not what it can find or use elsewhere. This is the main problem with Singleton because it introduces this inside-out dependency that can be hard to break. Pass In don't Reach Out.

Interfaces are separation points, hiding implementation detail. Code should not depend on what it *knows* goes on behind the scenes, it should stick to the script and get on with the job.

If all of this ends up with big arguments lists, well, so what? It may indicate a problem with the design (this object needs too much stuff), or it may make perfect sense, in which case Singleton is *Not* the answer!


Design to an Interface

- "interface" Keyword
- COM/CORBA - IDL
- C++ Pure Virtual Base Class
- Duck-typing : C++ templates, Ruby, Python
- C# and Java Generics

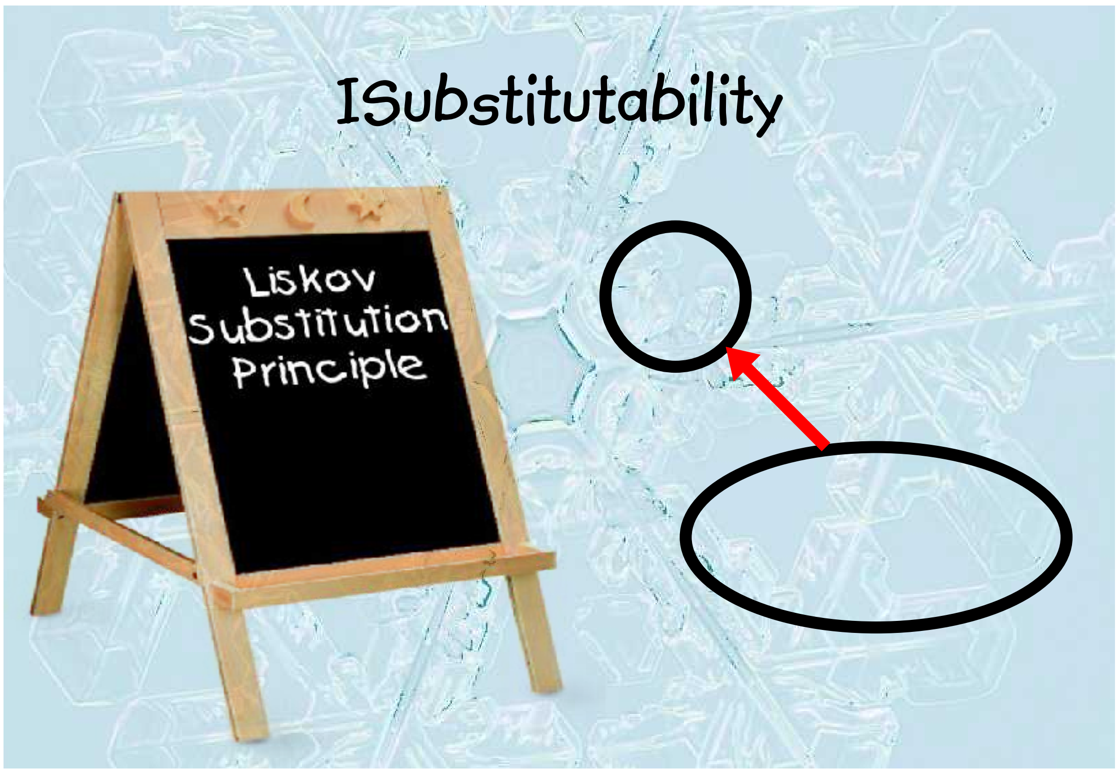
Designing to an interface is an extension of the idea of separating implementation from presentation. An interface is a protocol and a contract, specifying precisely what client code can do, and what it can expect in return.

Importantly, an interface deliberately hides details of the implementation from its clients, and in turn, interfaces expect client code to program *only* to the interface, and not depend on any details.

The obvious manifestation of interfaces have the "interface" keyword, such as is found in C# and Java. There are other kinds of interface however.

What this really boils down to is the idea of polymorphism, where an interface can "stand in for" an instance of any concrete type which implements it. Polymorphism is not just a class-related, or even late-binding thing, however.

An interface is all about **Type** and **Subtype**, which is not necessarily what programmers – or even compilers! - mean by those words. A type defines what operations are supported for a thing (its behaviour), and subtyping is the relationship between types according to that behaviour.



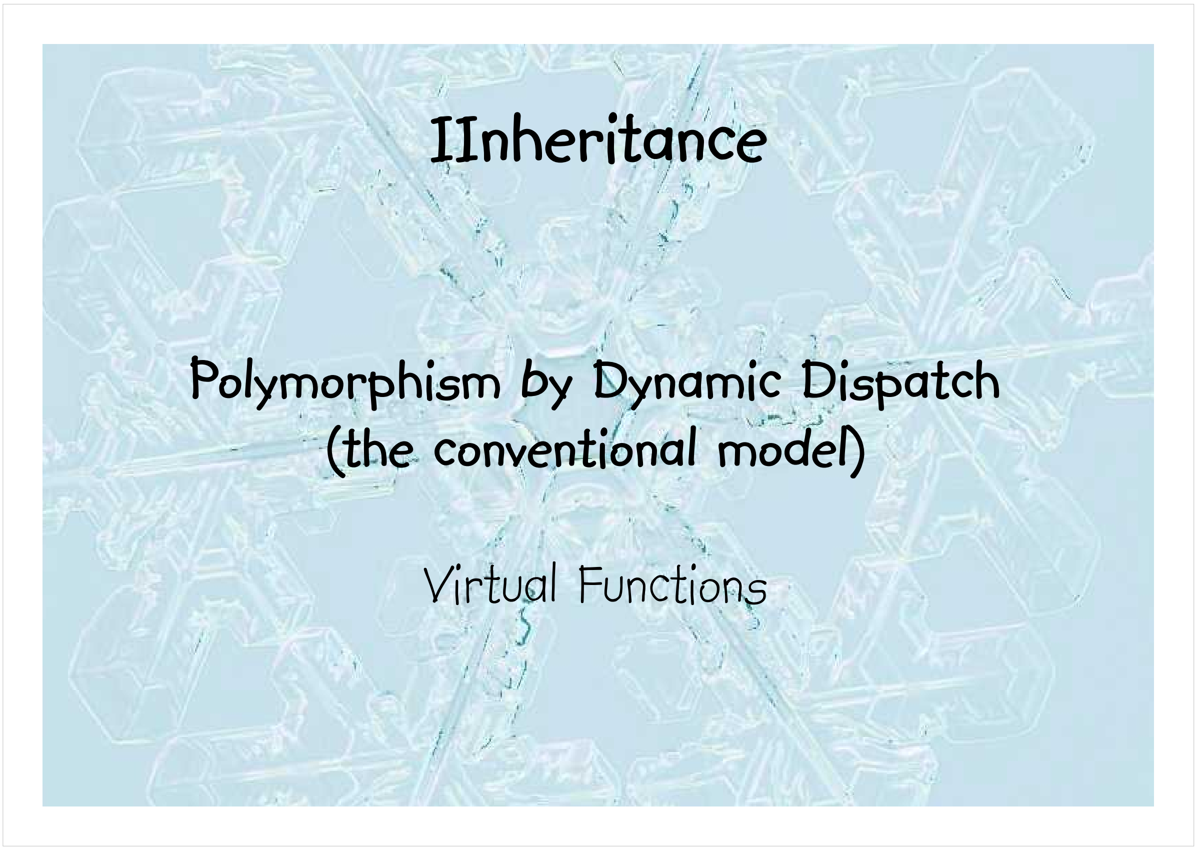
The key principle defining an interface is the Liskov Substitutability Principle – commonly referred to as the "IS-A" relationship.

Common questions relating to this are:

Is an ellipse a circle? Is a circle an ellipse? Is a cat a dog?

Clearly, the answer must be "No" in all three cases. Specifically, code that expects a circle and receives an ellipse will have some suprises in store! On the other hand, code that expects a circle may be depending *too much* on implementation detail. What is required is an interface common to both circles and ellipses, which allows code which is agnostic of the actual "shape" to operate on just the "shape", suggesting a "shape" interface which both circle and ellipse conform to in the LSP "IS-A" sense.

The "shape" is a type, and exposes the set of operations common to *all* shapes – so circle and ellipse are related by type according to behaviour that is common to both, and in common with other shapes. We begin to see how a thing may have *many* types, exposing different sets of behaviour. Cats and dogs are animals, as well as (perhaps) being pets (or not), but there may be few behavioural aspects in common between a "pet" type and an "animal" type.



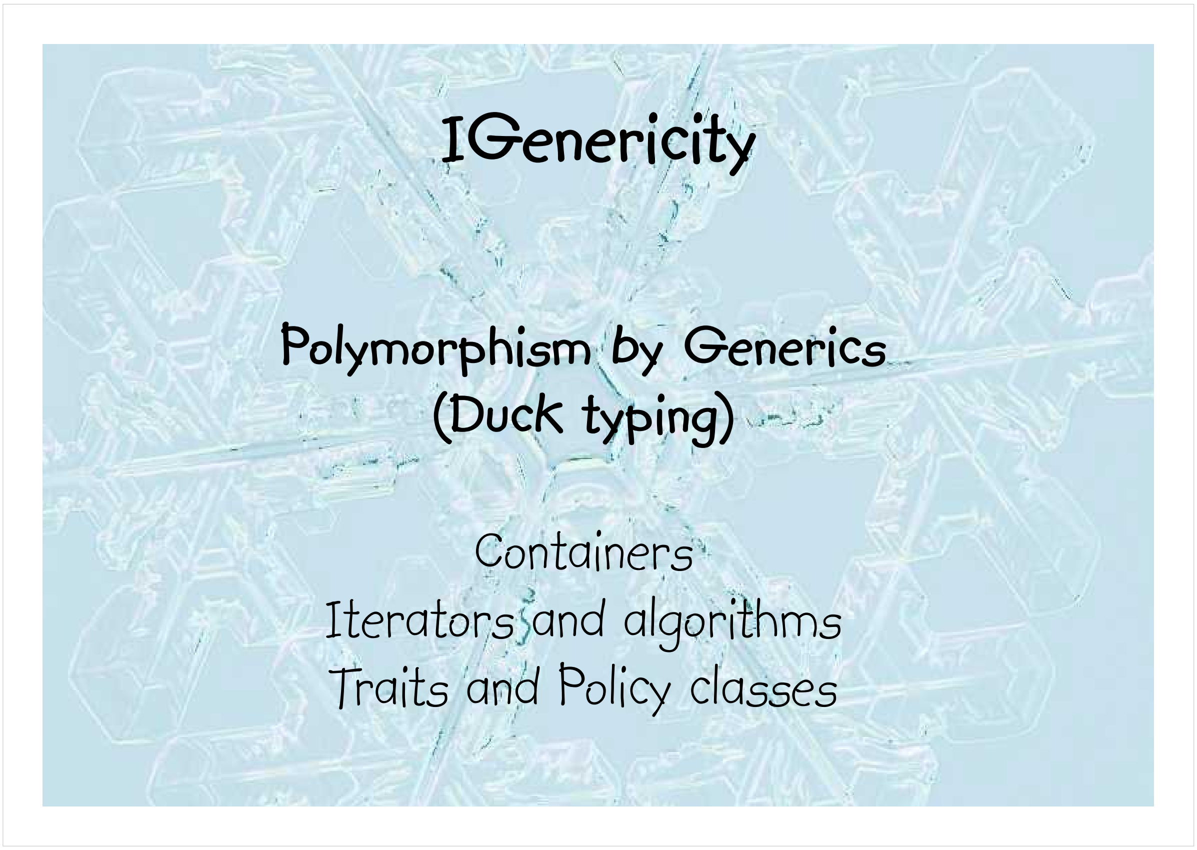
IInheritance

Polymorphism by Dynamic Dispatch
(the conventional model)

Virtual Functions

This is the type of polymorphism commonly meant when the term is used. It is characterised in most OO languages by the “virtual” keyword for methods which can be overridden in derived classes.

It is interesting to note that this version of polymorphism – called the inclusion model – forms the tightest possible coupling between classes, that of inheritance.



IGenericity

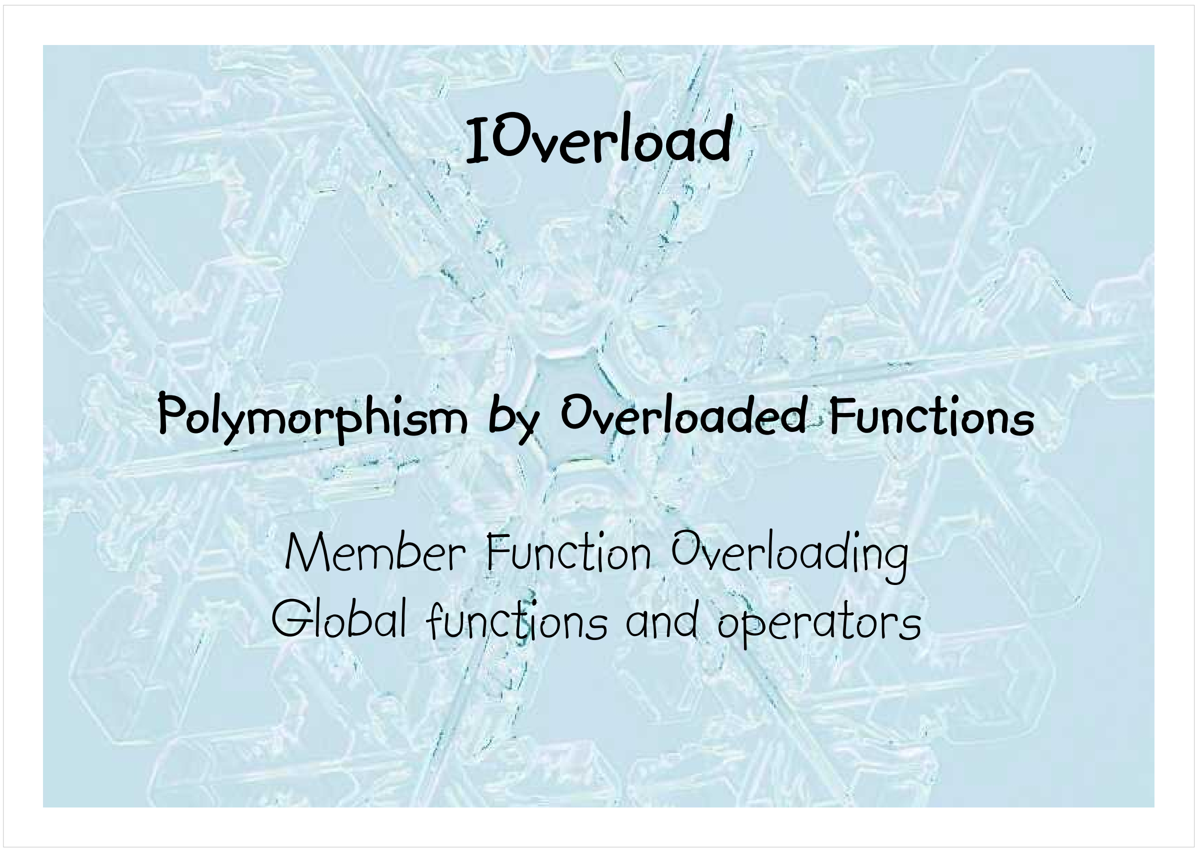
Polymorphism by Generics (Duck typing)

Containers
Iterators and algorithms
Traits and Policy classes

Generic, or parametric, polymorphism is used when some piece of code can operate on a generic parameter (“Template”) without knowing or caring about its actual type. Sometimes referred to as Duck Typing (presumably because the whole issue of Type is Ducked!), this method relies on the actual type provided to a generic piece of code being able to behave in the way required by the using code.

This feature doesn't really require templates or generics as seen in strongly-typed languages such as C++, Java and C#. Weakly- and Un-typed languages (e.g. Python) make use of parametric polymorphism natively; if the “thing” given to a piece of code doesn't do the right thing, then the code fails. Where strongly typed languages really differ is that a compiler can usually spot such errors **prior** to running the code, although sometimes this requires extra complexity in both generic code and the concrete types used as generic arguments.

Traits and Policy Classes are widely used in the C++ Standard Library and provide a way for generic code to find stuff out about the generic types it has been given, and perhaps behave differently based on that information.



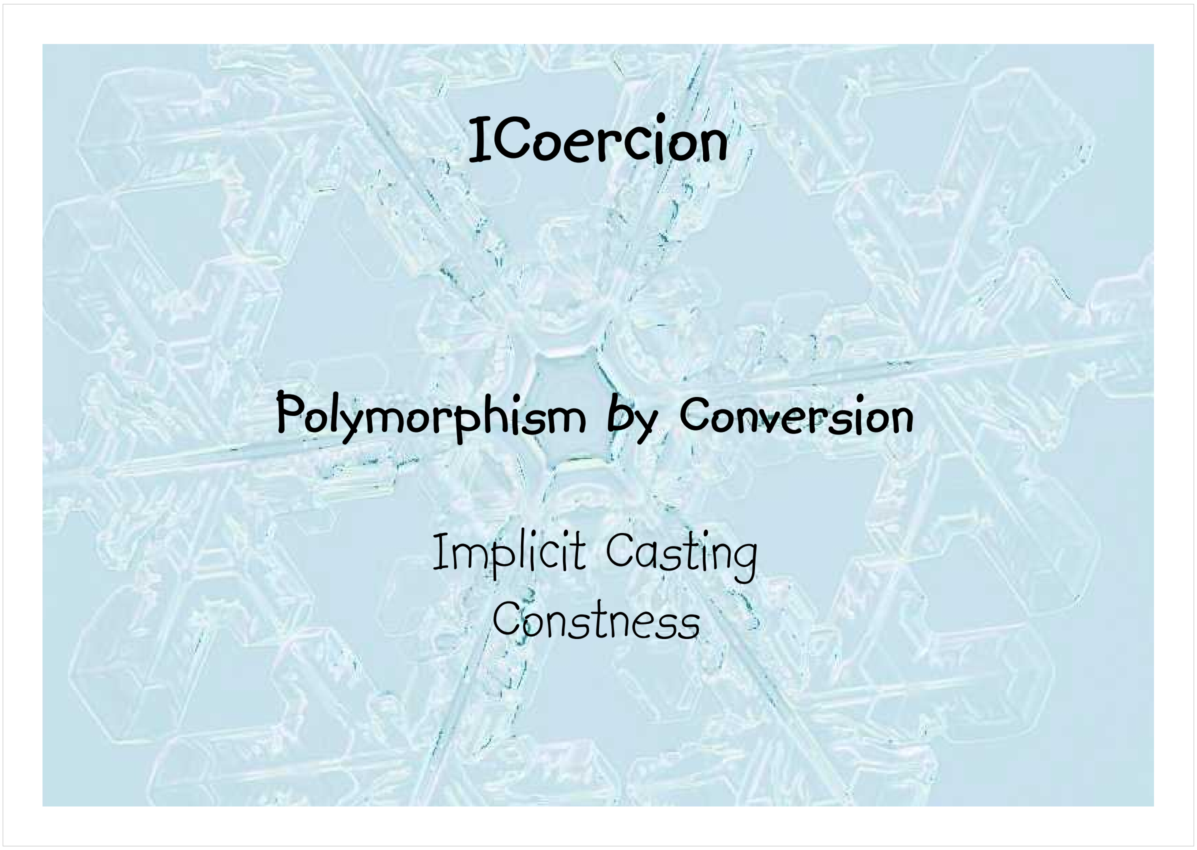
IOverload

Polymorphism by Overloaded Functions

Member Function Overloading
Global functions and operators

Function overloading allows a single function/procedure/method name to mean many things depending on the type and/or number of the arguments provided to it. A simple example might be a a function called “Add” which accepts 2 integer parameters and sums them together, which is overloaded by another function called “Add” which accepts two strings and concatenates them.

This method, in conjunction with Traits and Policy Classes mentioned in the generic polymorphism, can provide an extremely powerful way of making code truly polymorphic, at the expense perhaps of some clarity and directness.



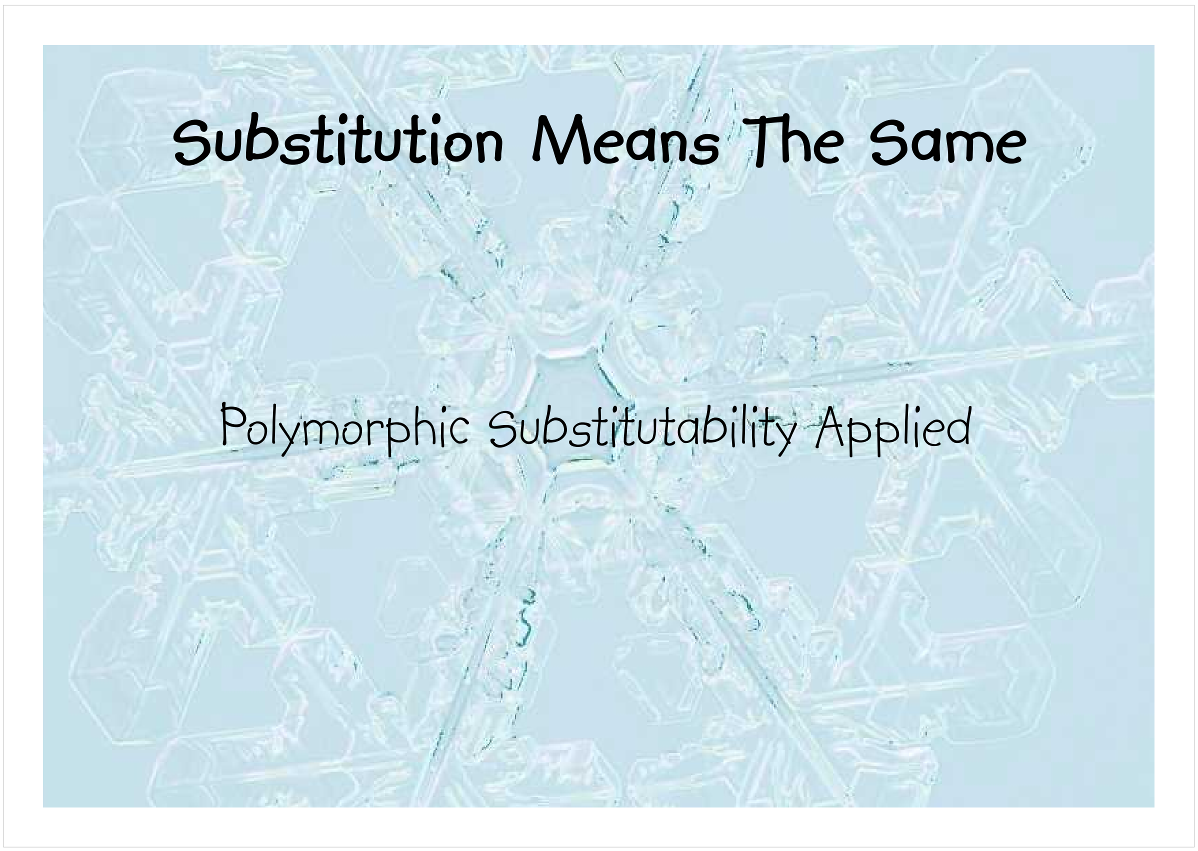
ICoercion

Polymorphism by Conversion

Implicit Casting
Constness

In strongly typed languages, a value can be many things just by the power of what the compiler will allow it to take part in, for example in C ++, an int may become a double at the drop of a hat – or more controversially, a bool may become an int at the drop of the same hat!

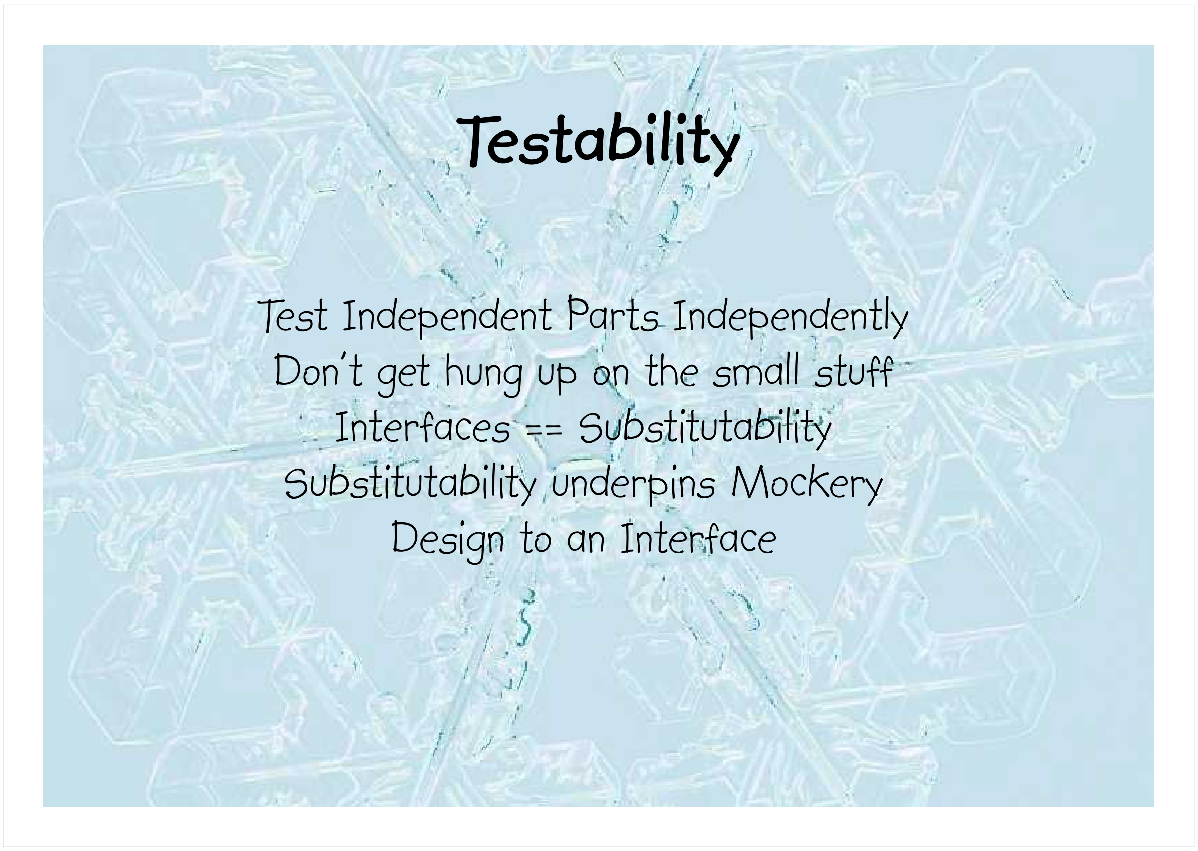
In common with parametric polymorphism, this model relies on what a type can **do**. Novice C++ programmers are often confused and dismayed that the standard string class has no implicit conversion to int (or indeed to the native C string as an array of characters). If they use a string type which *does* have those conversions, however, the dismay is usually increased as their strings take part in operations for which they were never designed – with occasional disastrous consequences.



Substitution Means The Same

Polymorphic Substitutability Applied

So all this talk of substitutability and polymorphism, but what does it all mean. In the next few slides we'll take a look at ***why*** this substitutability idea is so important for coherent and extensible design.



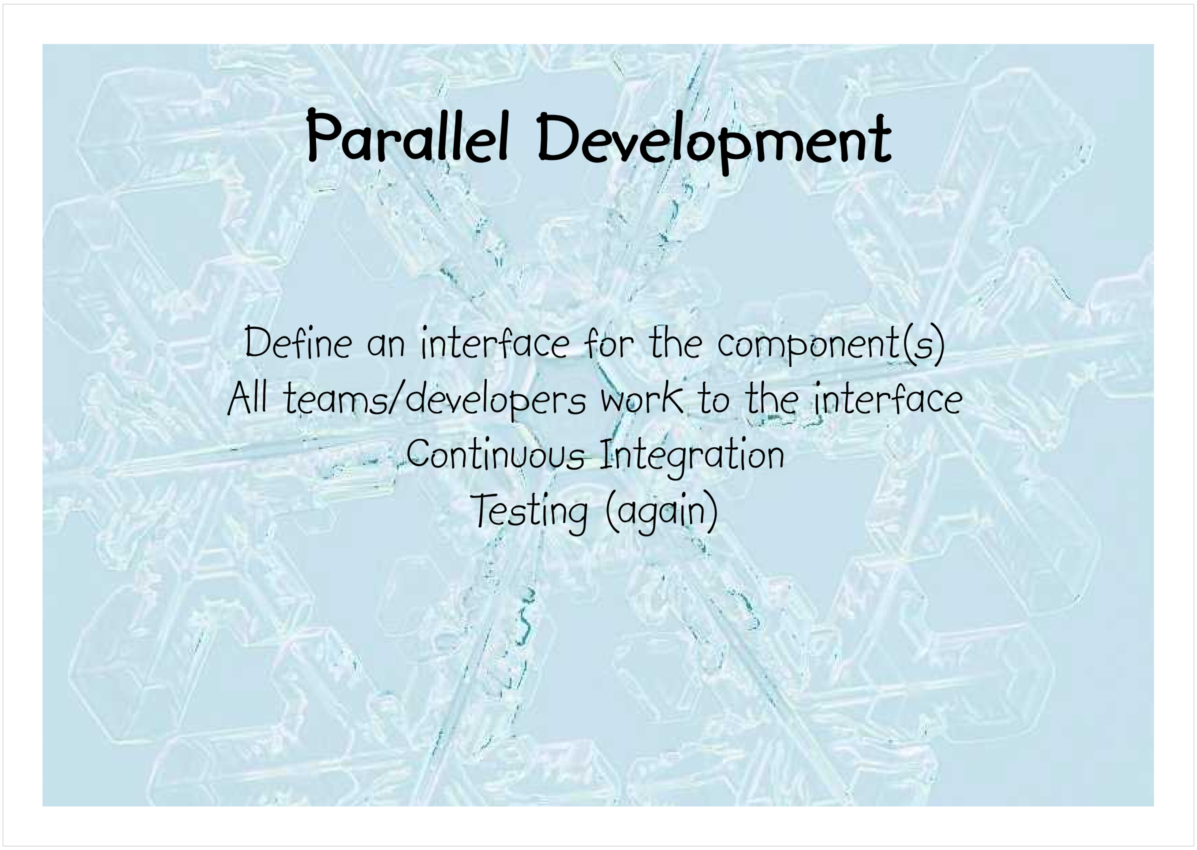
Testability

Test Independent Parts Independently
Don't get hung up on the small stuff
Interfaces == Substitutability
Substitutability underpins Mockery
Design to an Interface

Unit / Programmer tests are an important aspect to the project lifecycle, providing a confidence net for changes you make to your code. But here we're talking about software *design*, and in particular, substitutability. These things don't apply so much to the tests themselves as the code being tested.

If the code under test depends only on the interfaces for the services it requires – e.g. Database, comms, logging, whatever – then those services can be faked so that the code still does the job it's supposed to do (and can be tested doing it). Extending this a bit, having multiple implementations of the services, one of them fake, perhaps a different one being a true "Mock" which might *instrument* the code under test, checking timings, ordering, threading issues, whatever. The key principle is that the code under test *doesn't know a thing*!

By depending and knowing about only the interface and the API it exposes, the tested code is entirely indifferent to the actual implementation.



Parallel Development

Define an interface for the component(s)
All teams/developers work to the interface
Continuous Integration
Testing (again)

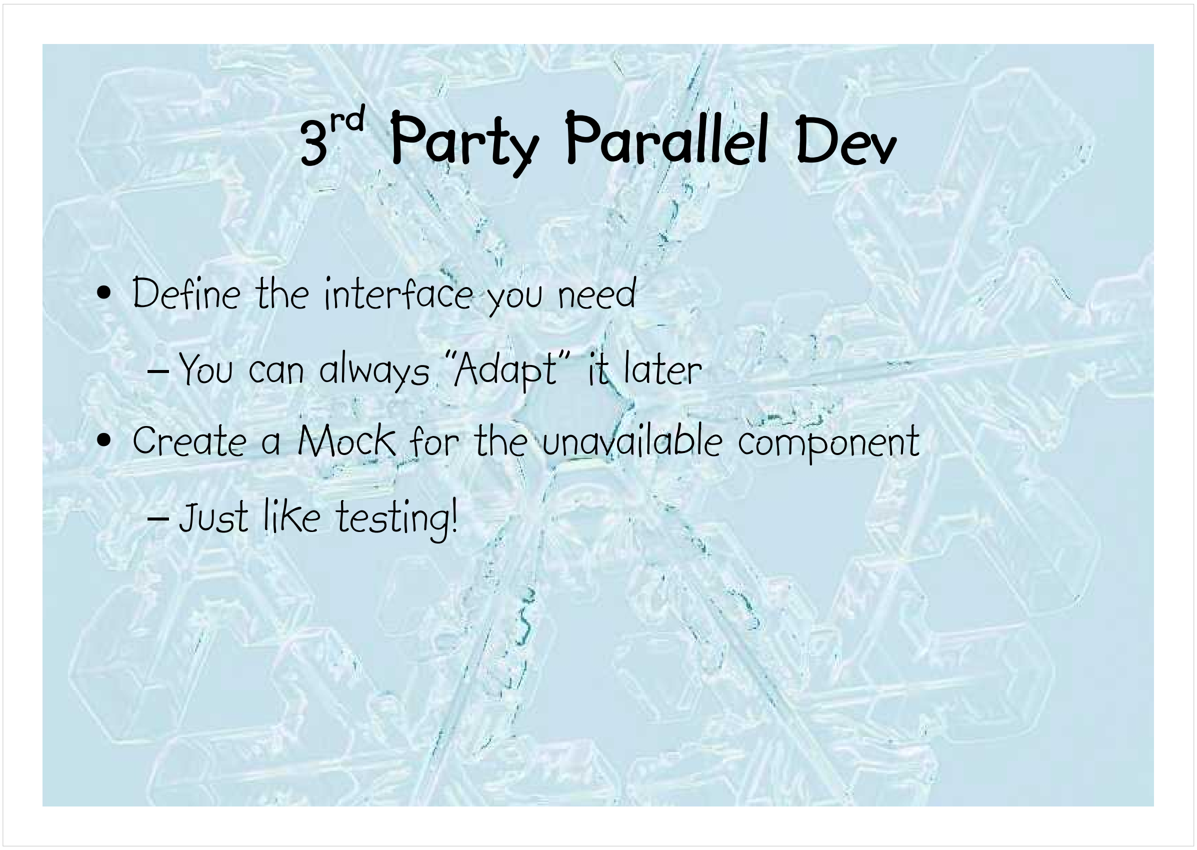
Extending the idea of *testable* code a bit, components can be developed in parallel, as long as the interfaces can (broadly) be well-defined in advance.

This means that if you are working on some component which depends on e.g. A database component, you can develop against a FAKE ONE until the real one is ready. It may even make sense to allow other sub-teams working on different components access to fake implementations, or perhaps the development of another component is scheduled for a later development cycle.

The importance is in the substitutability provided by developing against ***only*** the required interface.

By continuously integrating all components, a solid and broad test coverage can be achieved; in turn, this *ensures* that components individually depend only on interfaces to work.

This concept in turn leads to...



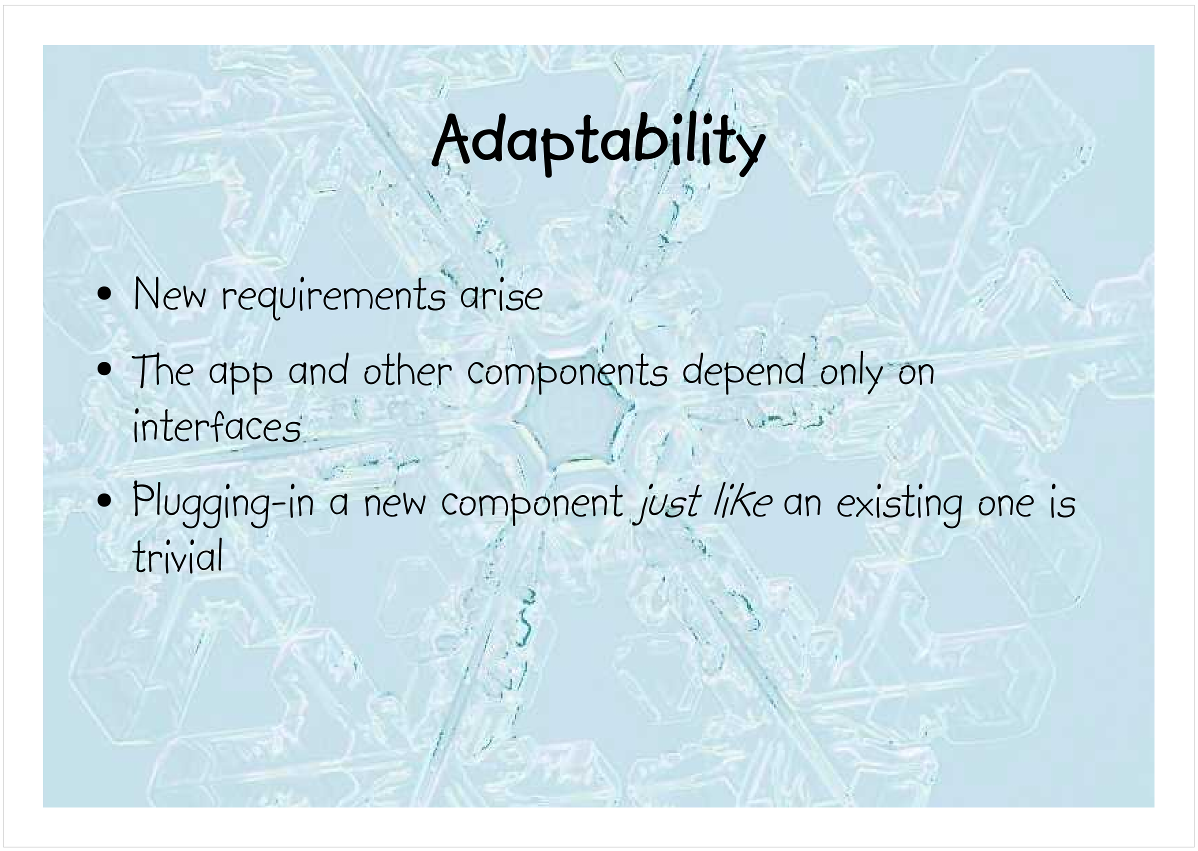
3rd Party Parallel Dev

- Define the interface you need
  - You can always "Adapt" it later
- Create a Mock for the unavailable component
  - Just like testing!

A component which is being developed by a third party is not so different from one being developed in another part of your own team, except you may have less (or perhaps more!) influence on its timescales.

It becomes ever more important to define the interface required up front, however. Even so, it is not a complete disaster should the interface change a bit – if you can persuade your 3rd party supplier to send you updates frequently, or even better, check-in directly (perhaps with some controls) to your version control system and take part in continuous integration – the actual component, when it arrives, can be adapted to the interface you've used all along anyway.

Once again, it's just like testing!



Adaptability

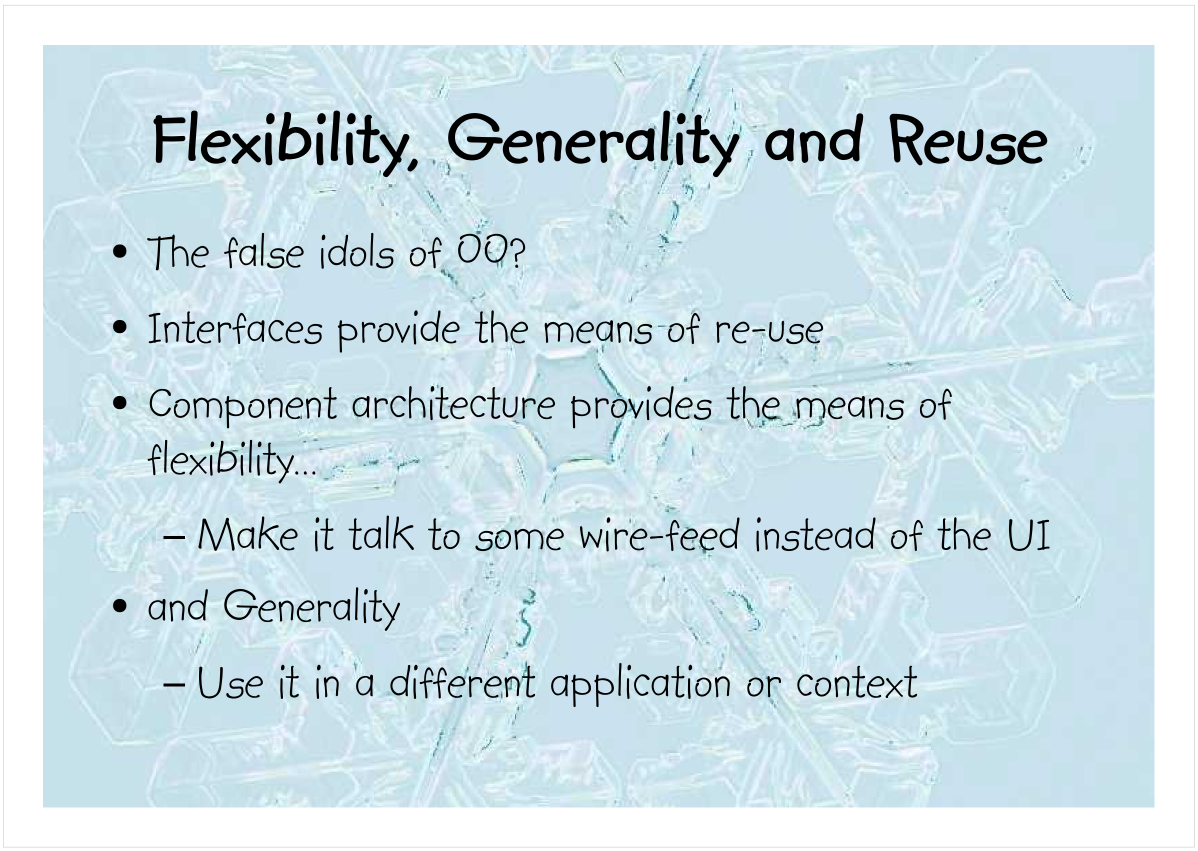
- New requirements arise
- The app and other components depend only on interfaces
- Plugging-in a new component *just like* an existing one is trivial

Finally, it is a truism that requirements change, and new requirements arise, and that the software we design is infrequently the same as what gets delivered.

At the indivudual component API level, changes can *only* be sensibly managed by *continuously integrating* new requirements, and refactoring code and interfaces together.

At the broad implementation level however, if a decision is made to use a WiFi network interface instead of a serial line one, or develop an in-house database instead of using a huge RDBMS, as long as the code using those components uses only the well-defined interface, and has no dependence on their realisation, they will not be affected *at all* by the new requirement.

Perhaps the new functionality needs to be made available according to some configuration – and this can be made truly trivial, provided the client code doesn't need to know.



Flexibility, Generality and Reuse

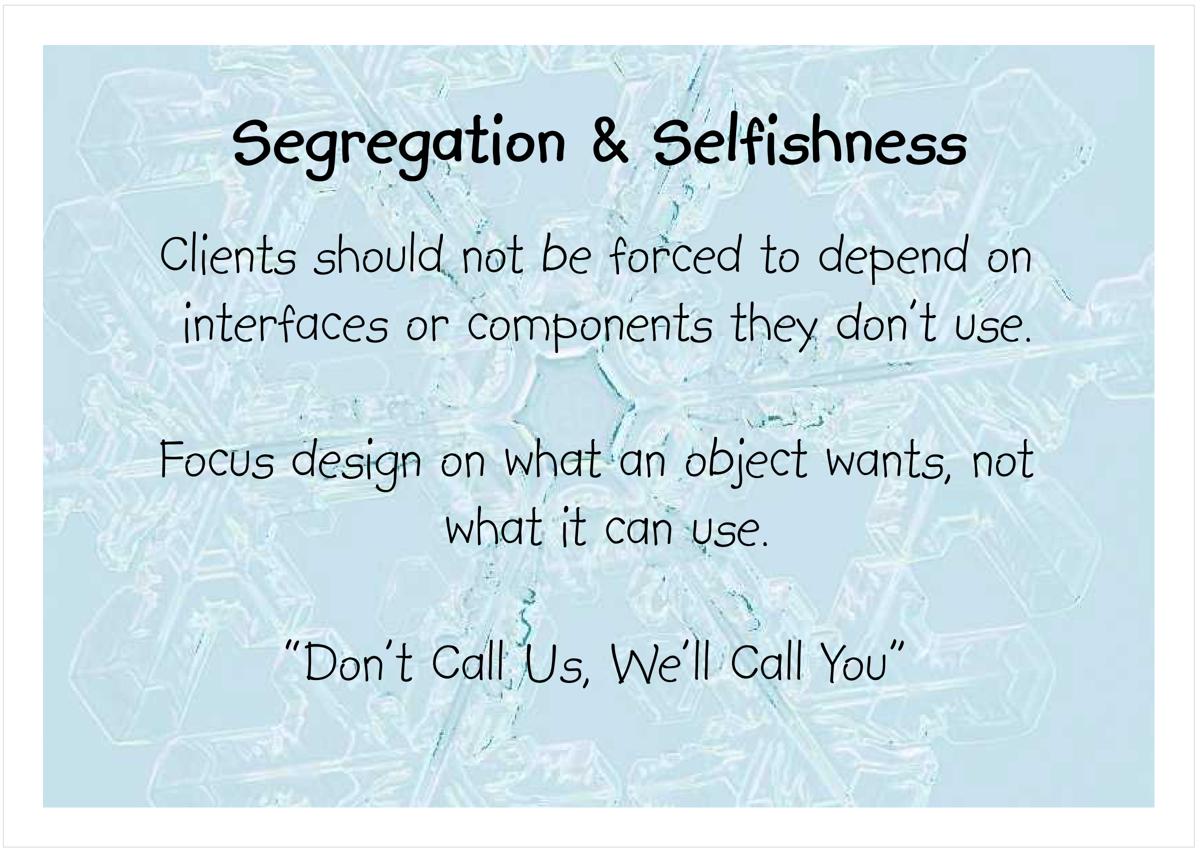
- The false idols of OO?
- Interfaces provide the means of re-use
- Component architecture provides the means of flexibility…
  - Make it talk to some wire-feed instead of the UI
- and Generality
  - Use it in a different application or context

Making a design flexible and general enough to promote reuse is difficult. Reuse in Object Orientation has traditionally been a matter of taking a class written by someone else and refine it by using it as a base, usually with implementation inheritance, or sub-classing.

This led to over-general and over-flexible interfaces in base-class libraries, to the extent they become unusable, never mind un re-usable.

Generality and flexibility are better realised through simplicity; leaner, more direct interfaces that appear less general because they are more specific in fact *promote* reuse because they bring with them less baggage – fewer dependencies and easier understanding.

Interfaces are the basic building block for reuse, not general purpose classes providing default behaviour.



Segregation & Selfishness

Clients should not be forced to depend on interfaces or components they don't use.

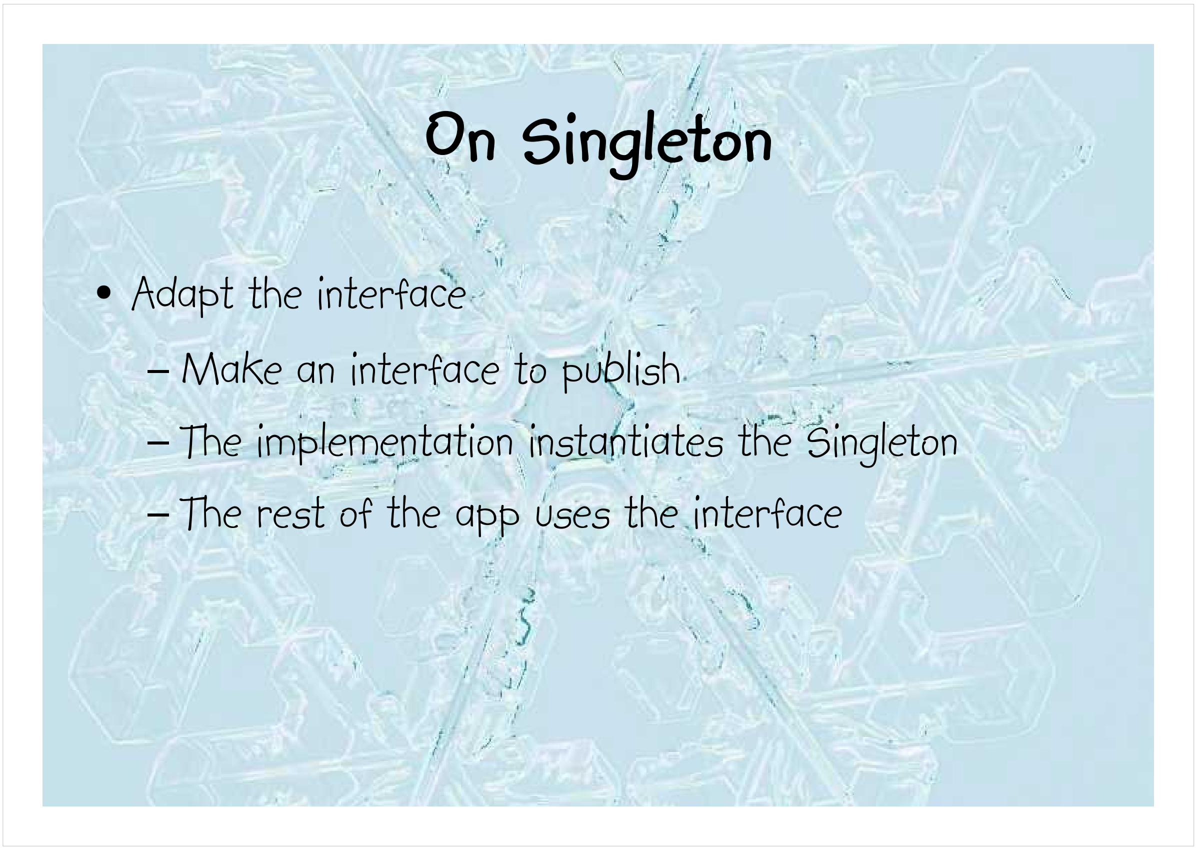
Focus design on what an object wants, not what it can use.

"Don't Call Us, We'll Call You"

Class interfaces get fat for a variety of reasons, but perhaps the most common is to make it more flexible, that is, to make it more generally useful.

This has the effect of making the class *less* generally useful, however, because client code then depends on all sorts of things it doesn't need – especially if the fat-interface class has other transitive dependencies sprawling off in all directions as a result of the richness of its interface.

Interfaces need to be designed according to what the client code *wants* to be able to do. Some services have naturally "fat" interfaces because they perform a variety of tasks; locality of reference brings everything into one place. If the ***interface*** is split up into several different interfaces each exposing different functionality, the result is that class becomes more generally useful.

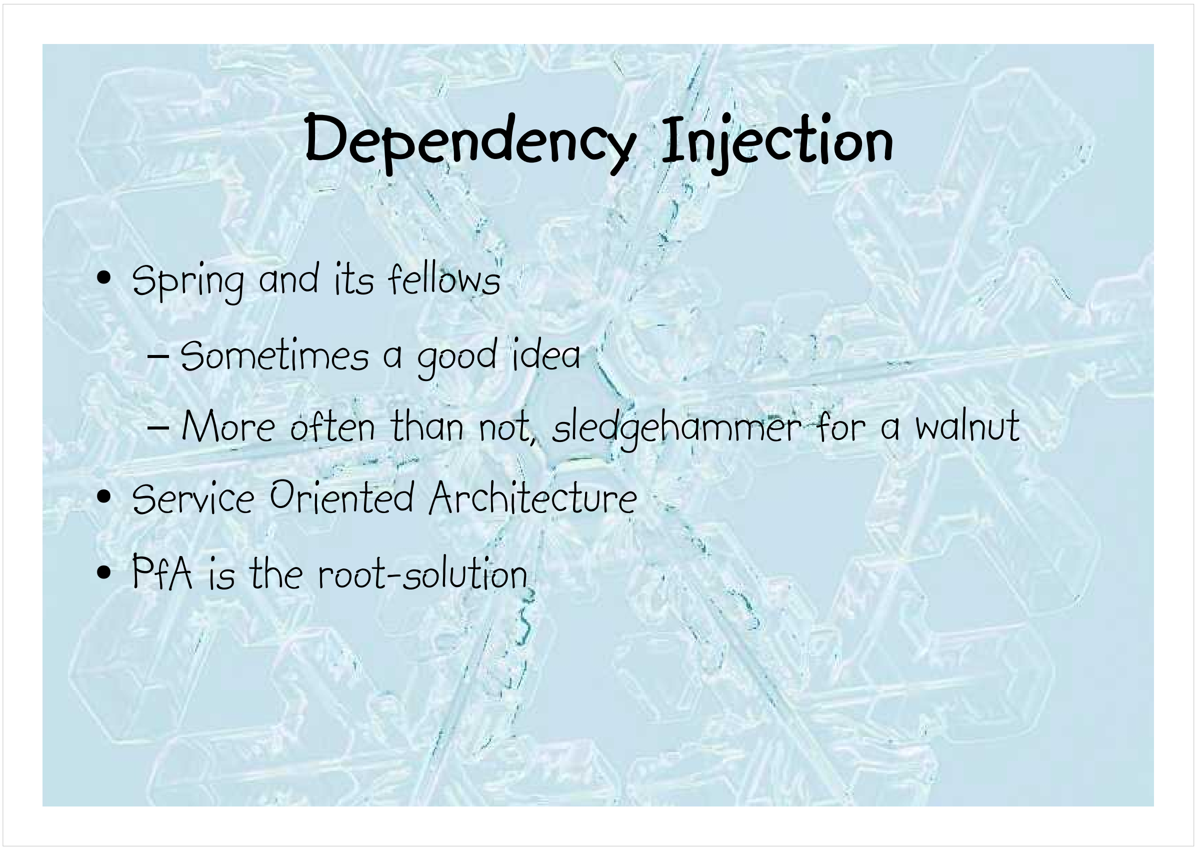


On Singleton

- Adapt the interface
  - Make an interface to publish
  - The implementation instantiates the Singleton
  - The rest of the app uses the interface

Removal of Singleton objects shouldn't be the goal – solving the problem that suggested use of a Singleton in the first place is where we should be looking. Sometimes it is just too much effort to remove Singleton instances in code, so there are other ways to remediate the problems associated with Singleton.

I'm sure you've guessed it by now – because it is the same solution used throughout this talk: introduce an abstract concept to represent the detail of implementation.



Dependency Injection

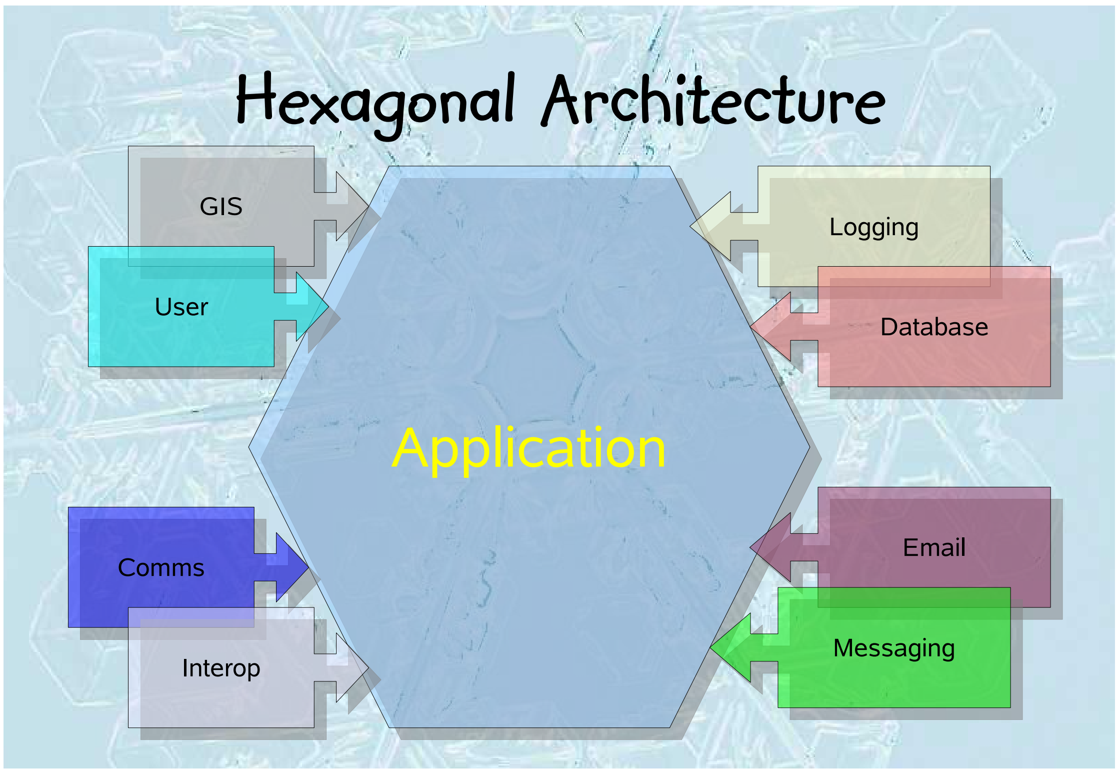
- Spring and its fellows
  - Sometimes a good idea
  - More often than not, sledgehammer for a walnut
- Service Oriented Architecture
- PfA is the root-solution

There are a number of frameworks and libraries available which promise the idea of true decoupling, whereby the interfaces for various *services* are defined, and concrete instances are instantiated according to a configuration file.

This all sounds fine and dandy – and it works very well – but it's often a step too far. Small self-contained applications have no real need of the extreme separation provided by the likes of Spring and Spring.net.

Similarly, the promises of the Service Oriented Architecture go yet another step, and define ways for applications written in different languages, operating on disparate platforms to communicate together. Each publishes *services* (that word again!) which can be used in other applications. SOA specifies *contracts* for data and operations that must be met by implementors so that clients can depend on them.

But ultimately the ideas of separation of concerns, contracts (interfaces) and service provision lie with the individual programmer, at the application level. SOA, or even DI tools like Spring, are not always necessary, and themselves can be a hindrance to clarity and simplicity, and so may lead to *bad* instead of *good* design. It can be tempting to suppose that these tools – shiny new buzzwords and all – are the silver bullet which has so far evaded us....

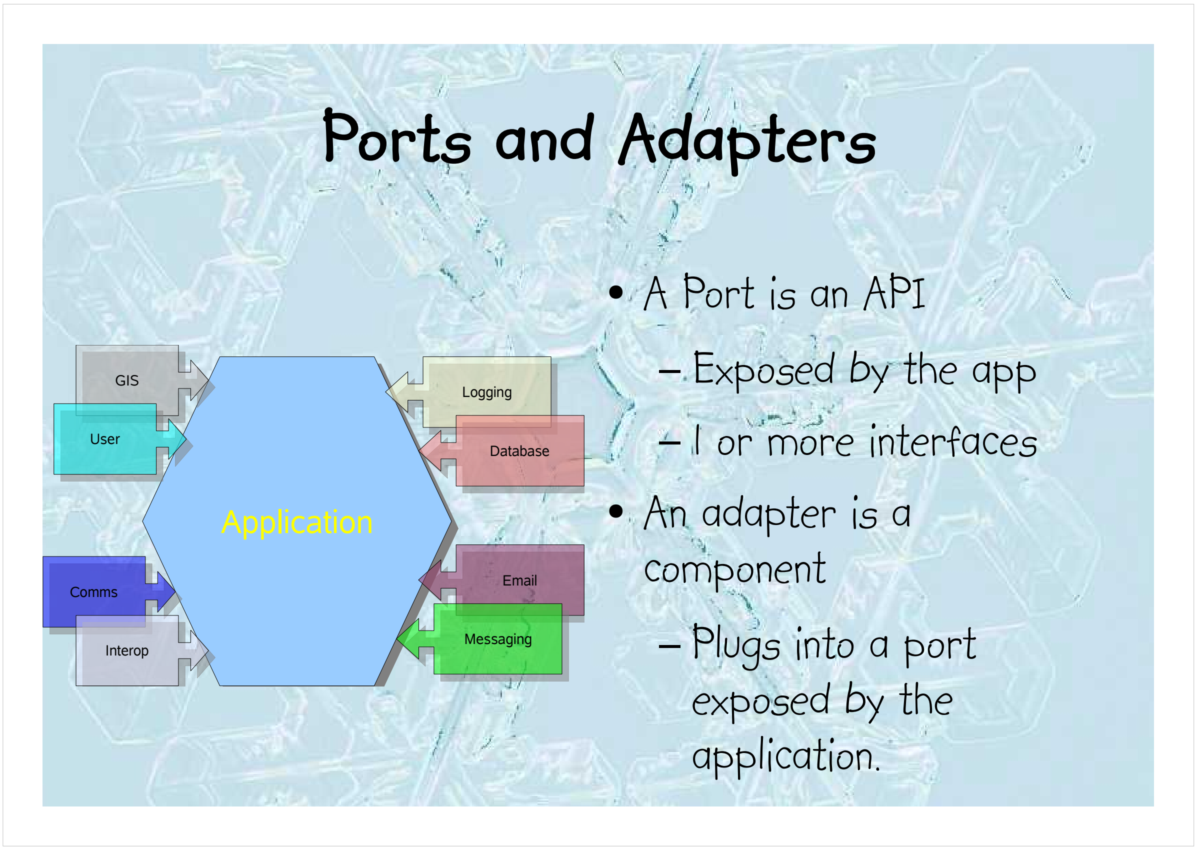


In a paper originally entitled “The Hexagonal Architecture”, Alisdair Cockburn describes an architectural style which promotes the idea that an Application itself is a service, operating on other services (later called Adapters) to provide a coherent whole. The key to the whole idea lies in a single sentence: “**The application has a semantically sound interaction with the adapters on all sides of it, without actually knowing the nature of the things on the other side of the adapters.**”

The adapters can be easily swapped out for mock objects or alternative implementations so that the application can be driven by batch process as easily as by a user, or a mocked network as easily as a real one.

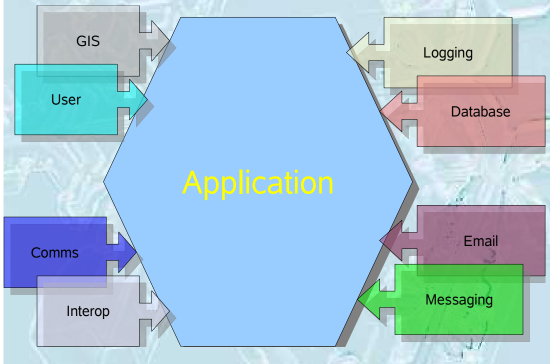
The hexagone was chosen deliberately to emphasise the symmetry between Inside and Outside, rather than above and below as in the layer cake model. It also highlights the idea that there are a number of Ports (over which Adapters communicate with the Inside App).

And, by complete coincidence, a snowflake is a Hexagon. So there's the secret to the title of this talk :-)



Ports and Adapters

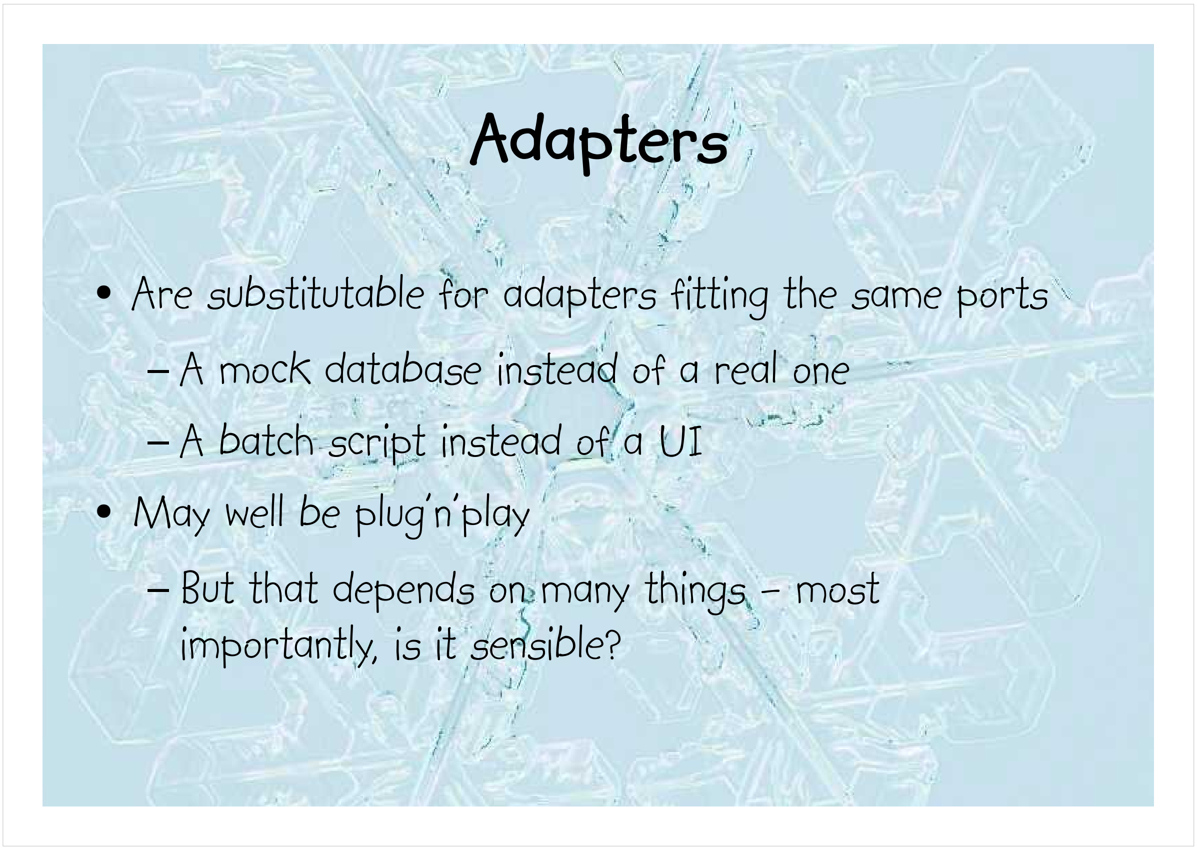
- A Port is an API
  - Exposed by the app
  - 1 or more interfaces
- An adapter is a component
  - Plugs into a port exposed by the application.



The purpose of the terms identifies how the communication happens. An application publishes/exposes *Ports*, which *Adapters* plug-in to. A Port may be an interface which is implemented by an Adapter, it may be an API which is called by an adapter.

This brings into play the idea of Active and Passive adapters. An active Adapter may call into the application, a passive one is used by the application.

So a database would normally be a passive adapter, implementing some interface. A UI would necessarily be an active Adapter, calling an API exposed by the application, which might be an interface given to the UI component.



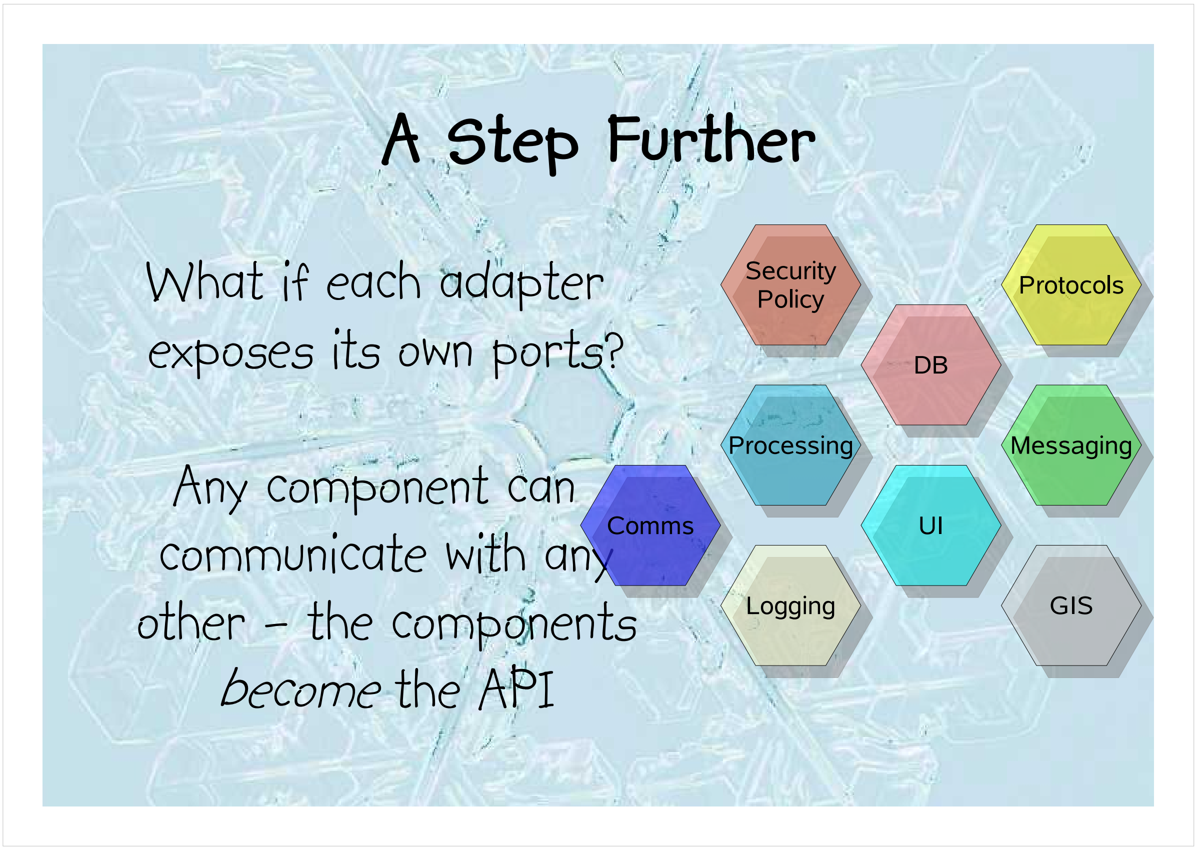
Adapters

- Are substitutable for adapters fitting the same ports
  - A mock database instead of a real one
  - A batch script instead of a UI
- May well be plug'n'play
  - But that depends on many things – most importantly, is it sensible?

Importantly, a port defines a contract, which in turn supports the idea that adapters which fit the same port ar substitutable for each other. Importantly, the application should know nothing of such a substitution, for example, using a MockObject for the database, or a batch script instead of a UI.

Adapters may even be plug and play – substitutable at runtime via some external means, perhaps a configuration file. Dependency Injection frameworks often provide this facility, but it's not always required, or even sensible to have that extra complexity “built-in” from the ground up.

A suitably decoupled architecture such as the Ports and Adapters idea here, supports the idea of runtime substitution without actually enforcing it.



A Step Further

What if each adapter exposes its own ports?

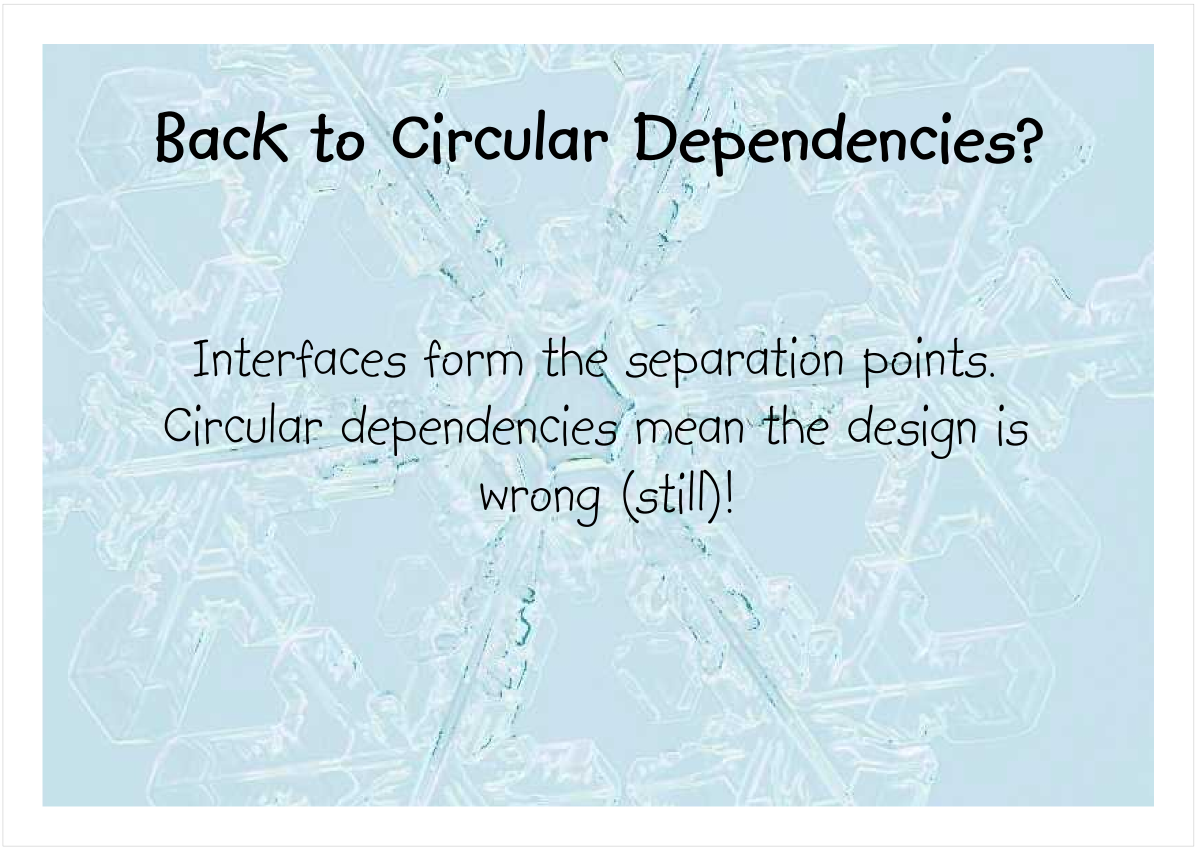
Any component can communicate with any other – the components *become* the API

Security Policy
Protocols
DB
Processing
Messaging
Comms
UI
Logging
GIS

Extending the idea of the “application” just a bit, what if each of the adapters exposed its own ports, so that other adapters could use it? This would be kind of like SOA, but much less formally. An application becomes a network of services, and building an application requires wiring those component services together...

Well, yes and no. This is indeed what SOA promises, and it fails to deliver for the same reasons as the OO hype of off-the-shelf components did – design happens on *your* side of the keyboard. Programming is still a skilled occupation.

To a point, however, the design of an application comprising services in other components is attractive. The components themselves become the applications API – its ports.



Back to Circular Dependencies?

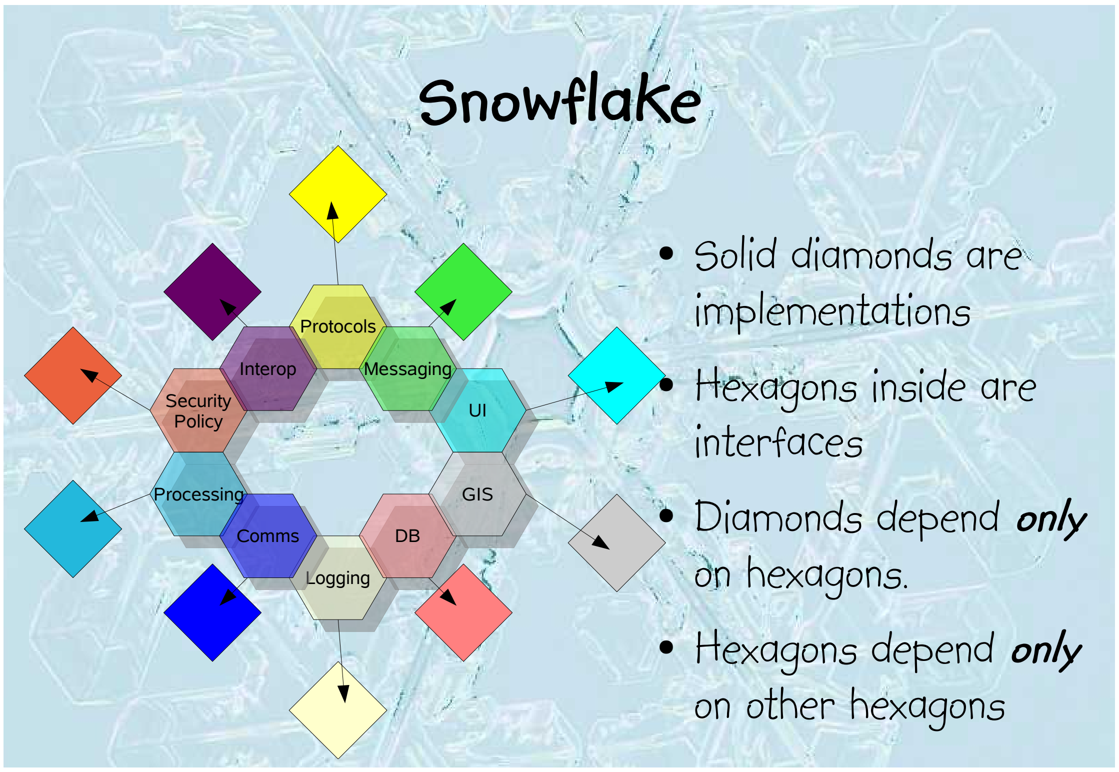
Interfaces form the separation points.
Circular dependencies mean the design is wrong (still)!

Sometimes it makes sense for components to be mutually dependent, at the conceptual level. Breaking that circular dependency is *still* a matter of using interfaces such that the interfaces themselves are *not* mutually dependent.

It's probably possible to imagine a scenario where even interfaces become mutually dependent, and as before the answer to that problem is to introduce another layer of indirection.

Interfaces are the separtion points – the variation points – in an application, a way to provide a parameterised approach to the architecture, vis. Mock objects and alternative implementations.

It's important to re-emphasise the idea of parameterising from ***outside*** (rather than ***above***) and attempt to ensure that where mutual dependencies exist, they are confined to the realm of *implementation*, not the interface.



Snowflake

- Solid diamonds are implementations
- Hexagons inside are interfaces
- Diamonds depend ***only*** on hexagons.
- Hexagons depend ***only*** on other hexagons

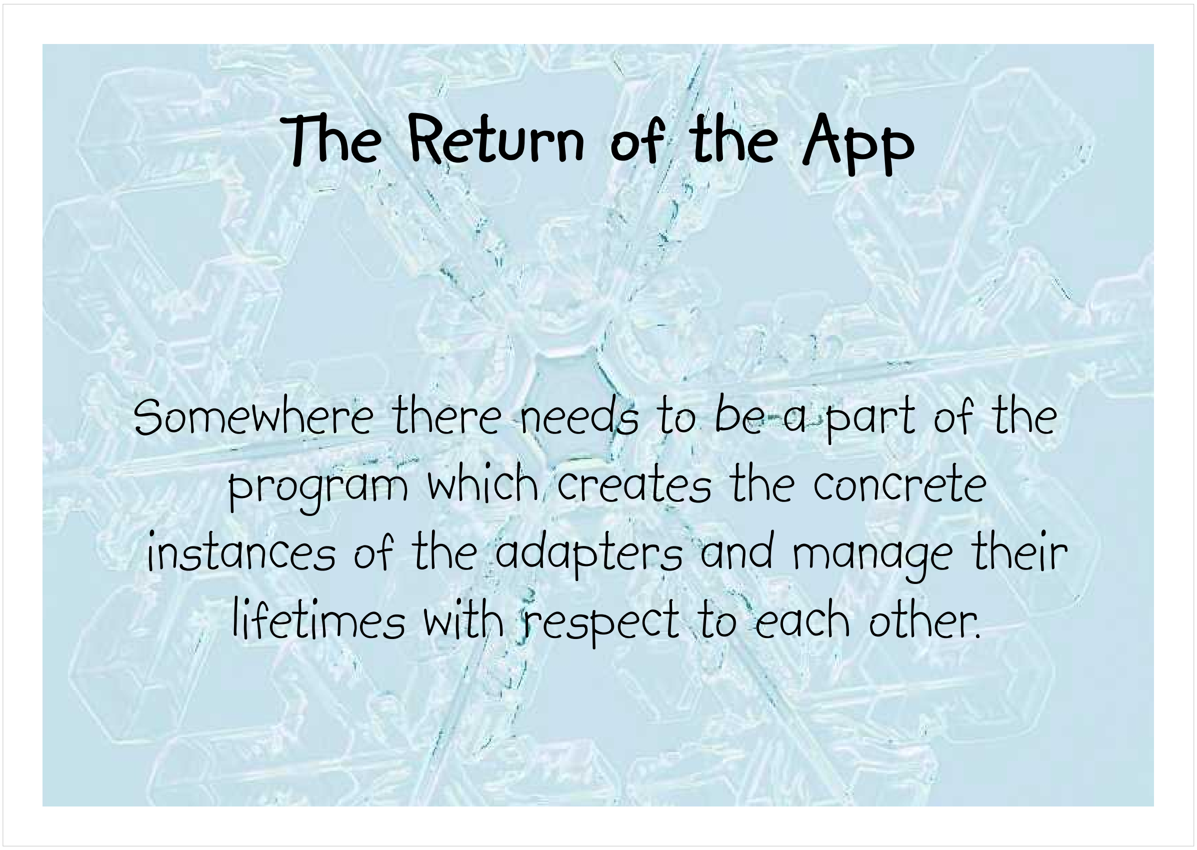
Where Alisdair Cockburn's Hexagonal Architecture had distinct left and rights sides, representing Input and Output (or Active and Passive), this model has no such distinction. Its purpose is to relate the deployment distinction between interface and implementation, and how the communication channels between interfaces (hexagons) can be direct, the communication between implementations is indirect, via interfaces.

It is important to note here that *NO* implementation diamon is a dependency of *anything* else – they should be entirely stand-alone. All dependencies are on Hexagon Interfaces – whether that dependency is implementation-to-interface or interface-to-interface.

It therefore conforms to the Dependency Inversion Principle,

“High level modules should not depend upon low level modules. Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.”



The Return of the App

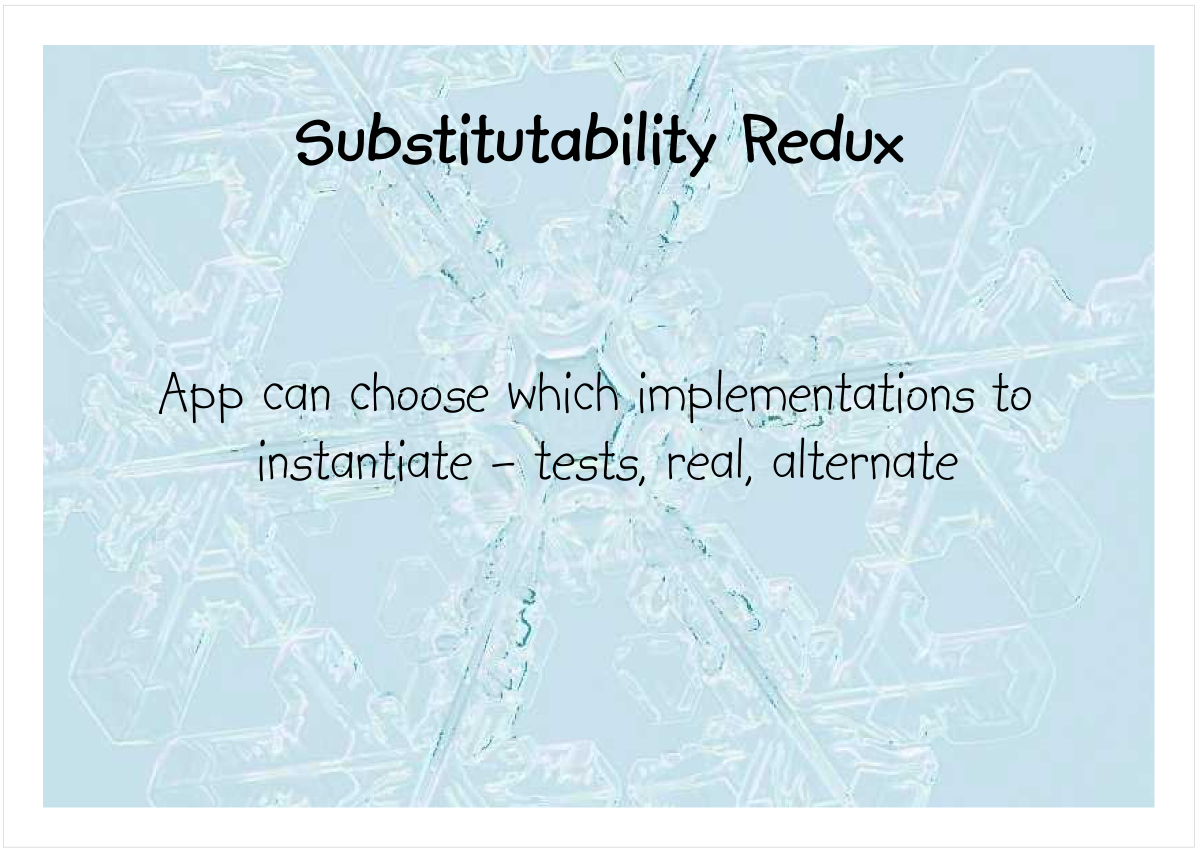
Somewhere there needs to be a part of the program which creates the concrete instances of the adapters and manage their lifetimes with respect to each other.

We've already explored the idea that sometimes it makes complete sense for services to be mutually inter-dependent; an example might be a database component that writes to an external logging component, which in turn has a database connection as one of its writers.

It becomes up to the application to manage those dependencies, to instantiate the implementations, and where necessary (it usually is, one way or another) manage their lifetimes.

There needs to be some aspect of an application that does this – even with a DI framework, there needs to be “An Application” that marshalls the communication between components to *form* a system.

A truly decoupled architecture is not an application or system, just a pile of independent parts.



Substitutability Redux

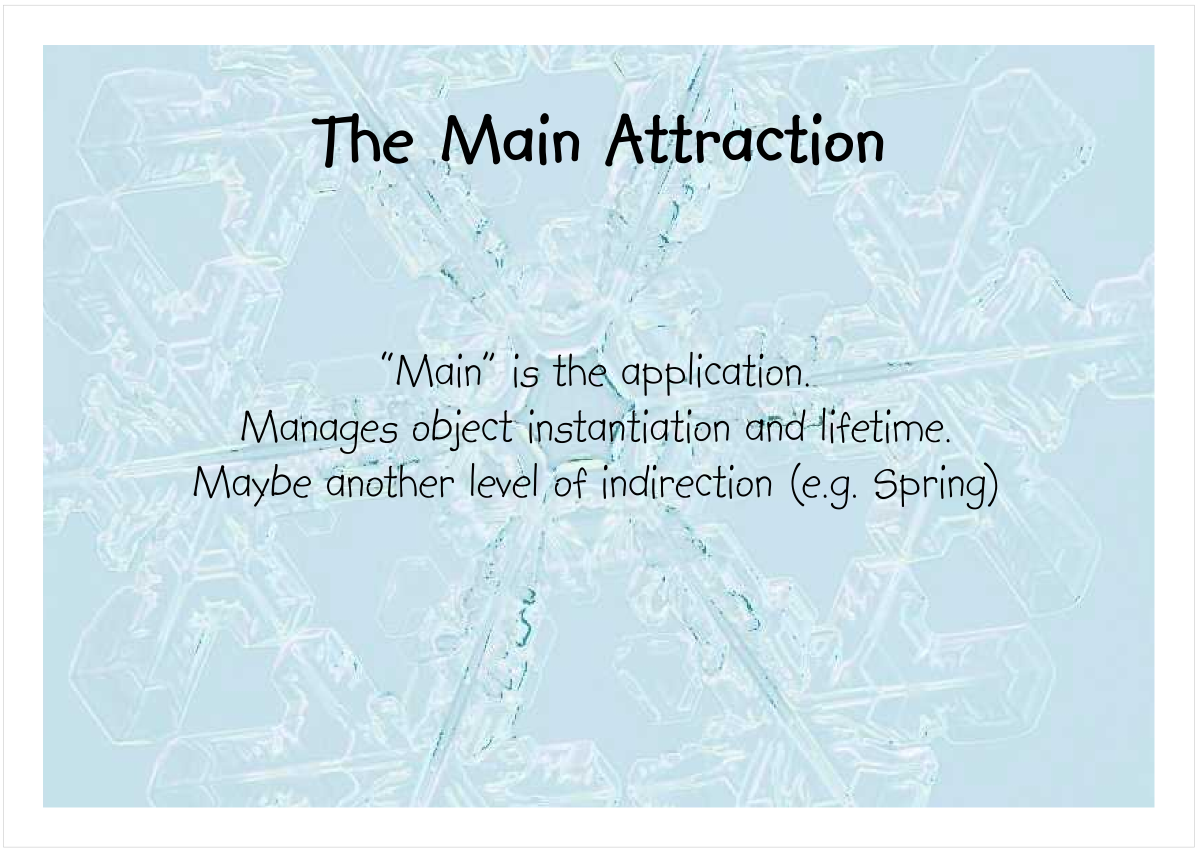
App can choose which implementations to instantiate – tests, real, alternate

Part of the importance of substitutable components is that the application can now choose which implementations to instantiate. A test application intended to drive components or other code for testing will do things differently from a production application, so there may end up being seceral applications of your software.

But we still have the open question – what *is* the application?

We are back to the idea of the application being a component itself, but maybe one which has only outward dependencies – nothing depends upon *it*.

Fortunately, in many languages, the facility for doing this already exists; it is in fact mandated for many of them...



The Main Attraction

"Main" is the application.
Manages object instantiation and lifetime.
Maybe another level of indirection (e.g. Spring)

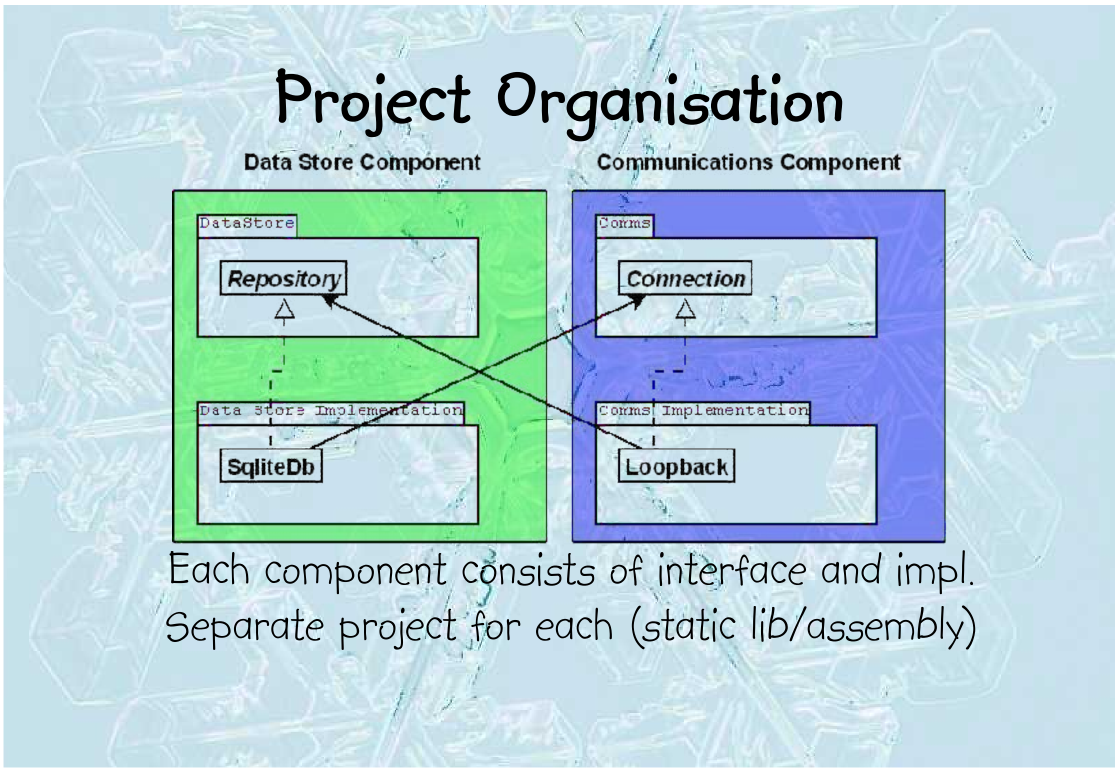
If component implementations are instantiated in main() or Main() or whatever it is called, there is an obvious way to manage both the number, type and lifetime of objects created.

If a component requires to be a single instance, instead of wrapping it up as a Singleton, create *just one instance of it*. The Main entry point to the program defines and implements the policy of the components it creates and manages, including when they get destroyed, and in what order.

It is the one point in the system which knows about concrete implementations, what their interdependencies are, and how to create and destroy them.

There may be other forces at play, too, such as lengthy argument lists to some components, or the requirement to be able to vary implementations whilst the program is running.

Each is manageable at this level of the application – it is the dirtiest, least encapsulated, and maybe the most complex of all code – but it is all in one place.



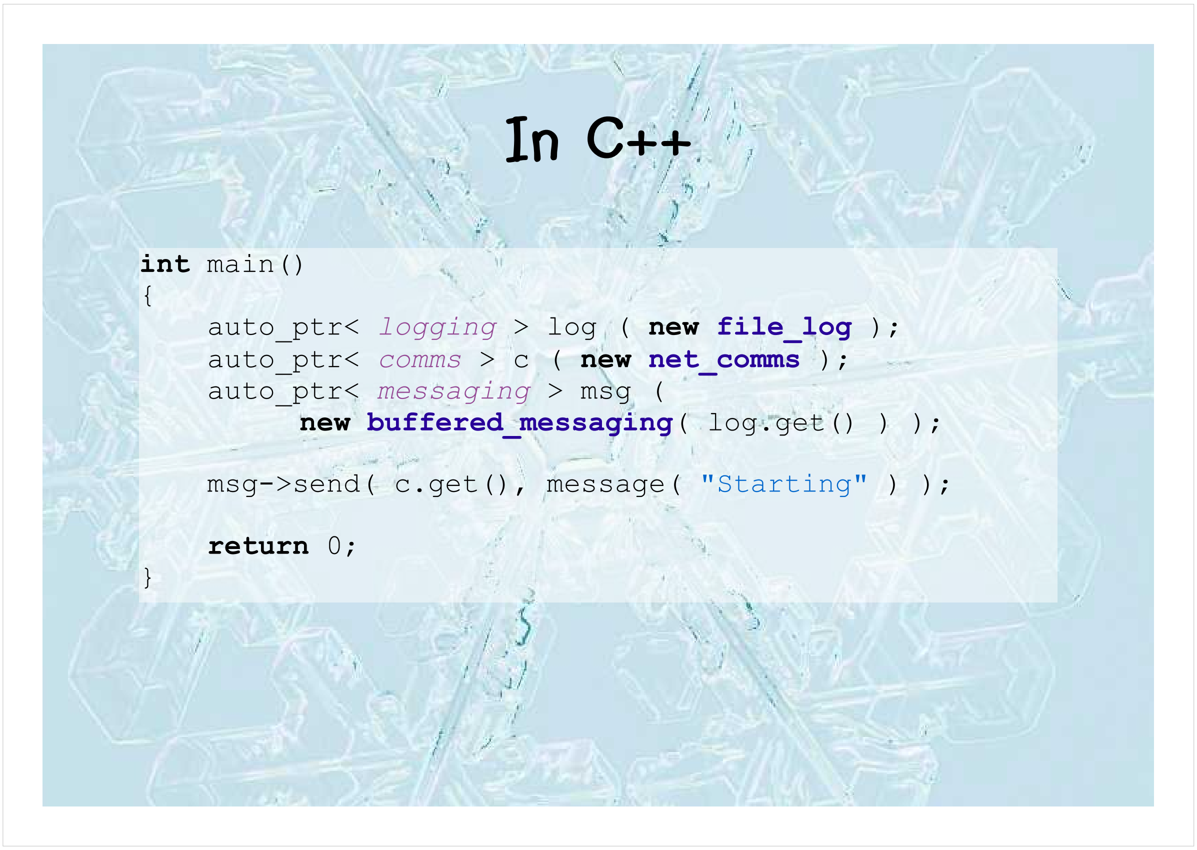
Project Organisation

Each component consists of interface and impl.
Separate project for each (static lib/assembly)

In order to ensure that a component doesn't require its clients to add dependencies they don't use, it can be split into separate modules (perhaps static or dynamic libraries, .Net assemblies, etc.) one for just the interface, and one or more for implementations.

In the example shown, two components are mutually dependent at the implementation level. If the separation of modules had not been done, any client of either component would depend upon both, even if that client required only one of them. Note there is *no* dependency between the two interface modules, so client code which requires the data store component does *not* require a reference to the communications project, provided it depends only on the interface.

The part of the application which creates the implementations will bring all the needed modules into play, instantiate objects and connections in the right order to satisfy their dependencies, and manage their lifetimes accordingly.



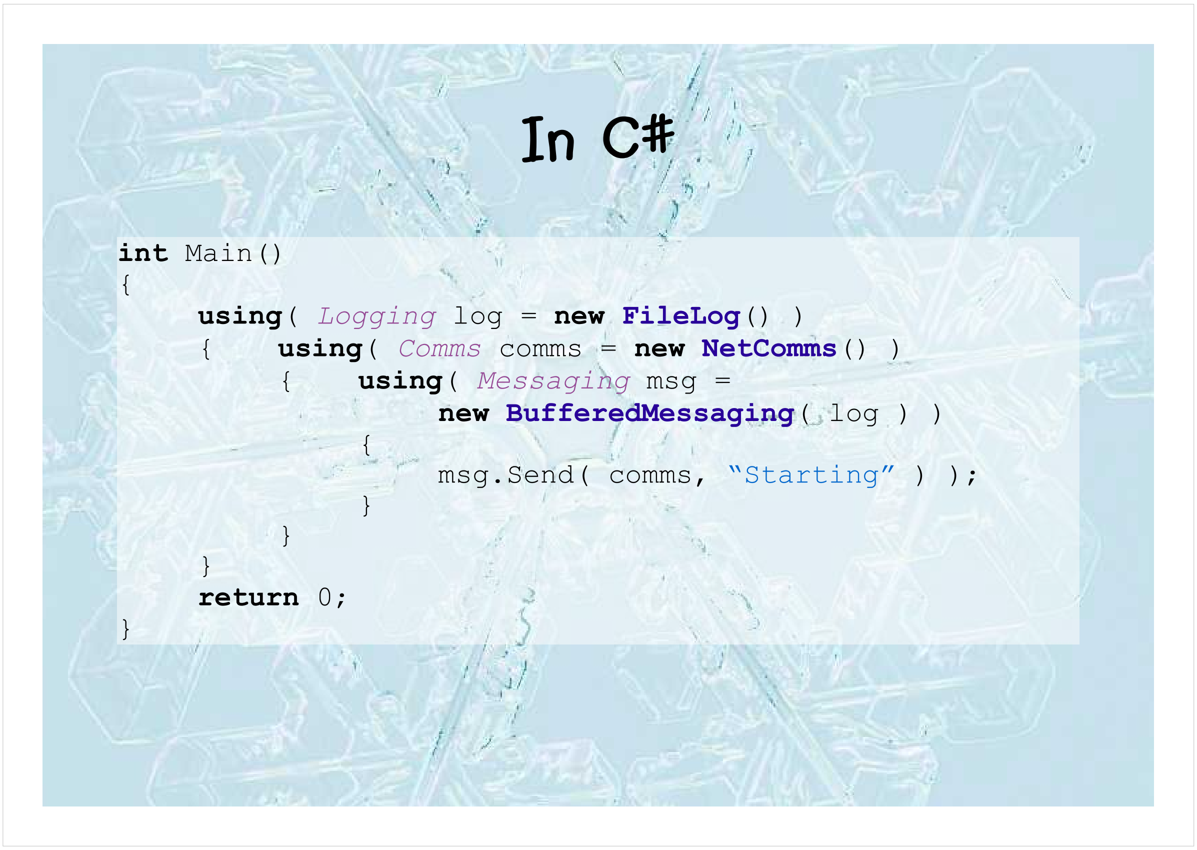
In C++

```
int main()
{
    auto_ptr< logging > log ( new file_log );
    auto_ptr< comms > c ( new net_comms );
    auto_ptr< messaging > msg (
         new buffered_messaging( log.get() ) );

    msg->send( c.get(), message( "Starting" ) );

    return 0;
}
```

For example, a simple C++ application may look a little bit like this.

Note that the `msg` object depends on the `log` object already created, and the `comms` object in the call to `send()`.

C++ guarantees that objects are destroyed in the reverse order to which they were created, so this code takes advantage of that fact. Using `auto_ptr` to manage the lifetime ensures that correct destruction takes place whatever happens (pretty much!). It's used here as an example – there are other and better ways of managing shared objects' lifetimes.

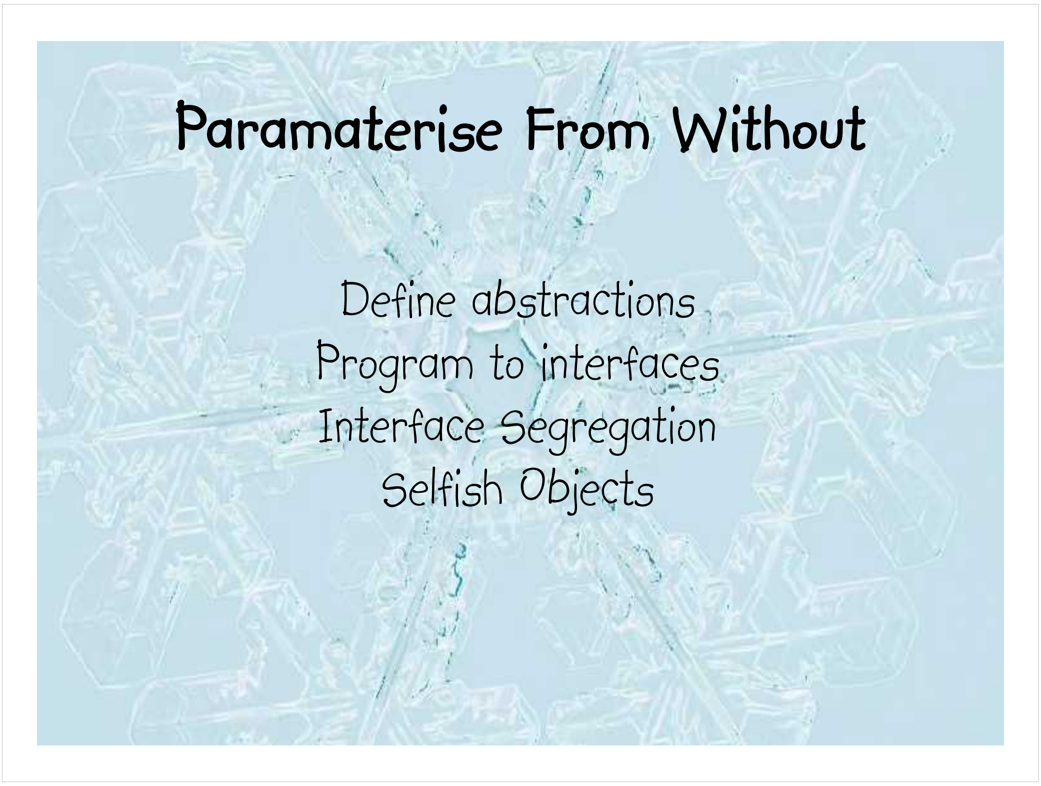


In C#

```
int Main()
{
    using( Logging log = new FileLog() )
    {   using( Comms comms = new NetComms() )
        {   using( Messaging msg =
                    new BufferedMessaging( log ) )
            {
                msg.Send( comms, "Starting" ) );
            }
        }
    }
    return 0;
}
```

Here is the same example, but in C#.

The real key here is that dependencies are passed *in* to objects requiring them. Keeping with the spirit of Alisdair Cockburn's Ports and Adapters/Hexagonal Architecture, I refer to this as Paramaterise From Without.



Paramaterise From Without

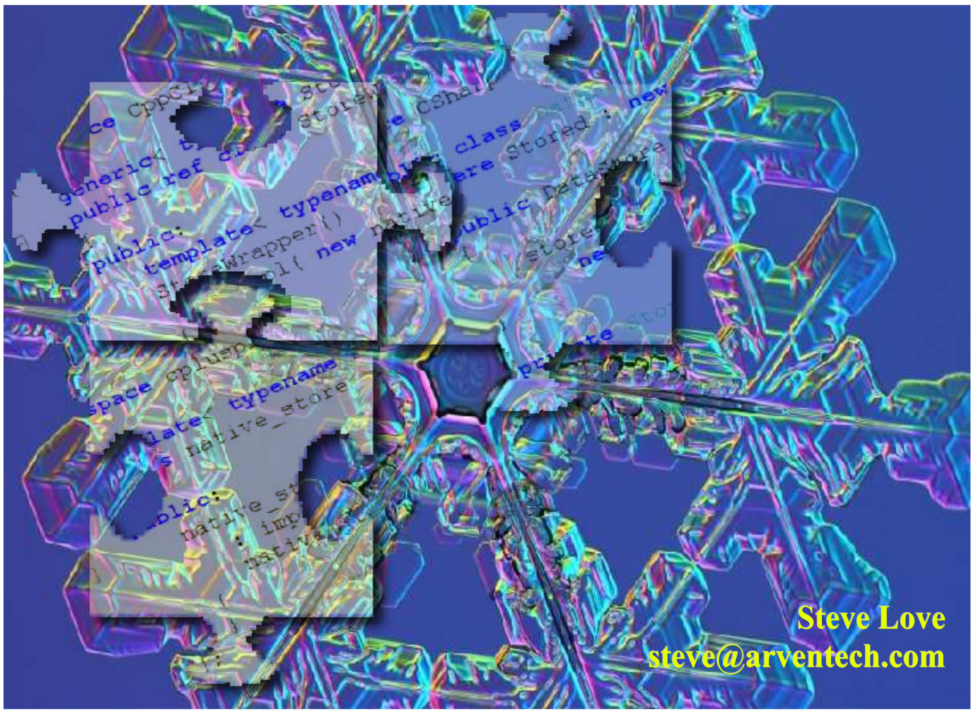
Define abstractions
Program to interfaces
Interface Segregation
Selfish Objects

In conclusion then, architecture and design are practically indistinguishable from each other at a certain level. Without having thought a bit about design at a fairly low level, we cannot consider the architecture to be stable, and conversely, architecture itself influences design because it changes the way we think about it.

Abstractions are – by their nature – slippery things, but it's important to identify the main actors in a system because that can influence the architecture.

Expose those abstractions through interfaces, and ensure that client code can and does depend on the interface alone. The Interface Segregation Principle proposes *more* interfaces with *thinner* APIs – even if some implementations end up implementing several of them (service objects).

Finally, objects should be selfish, and use only what they are given. If they can't do the job with what they are given, then either they're in the wrong job, or the job hasn't been *designed* properly – fix the design.



Steve Love
steve@arventech.com