Design and Architecture in Industry
The agile viewpoint

Ben Stopford

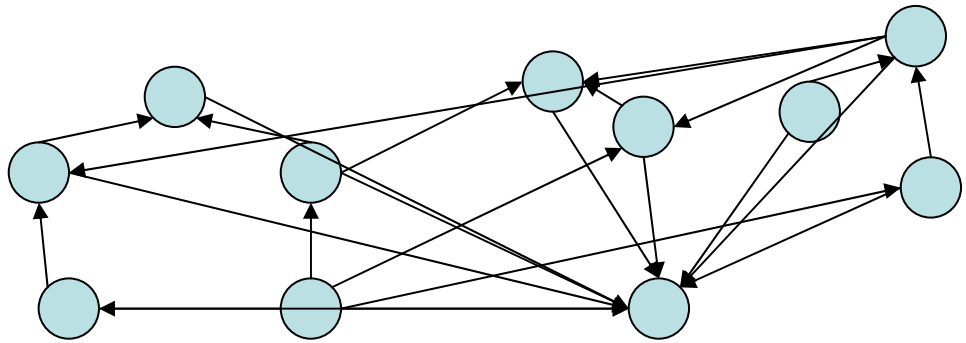
Thoughtworks

What I’ll be covering

- The importance of expecting designs to change.
- The softer side of architecture needed to successfully guide a team.
- Methods for guiding design using patterns and frameworks.
- Problems that can occur with design.

Why do we need to worry about Architecture and Design?

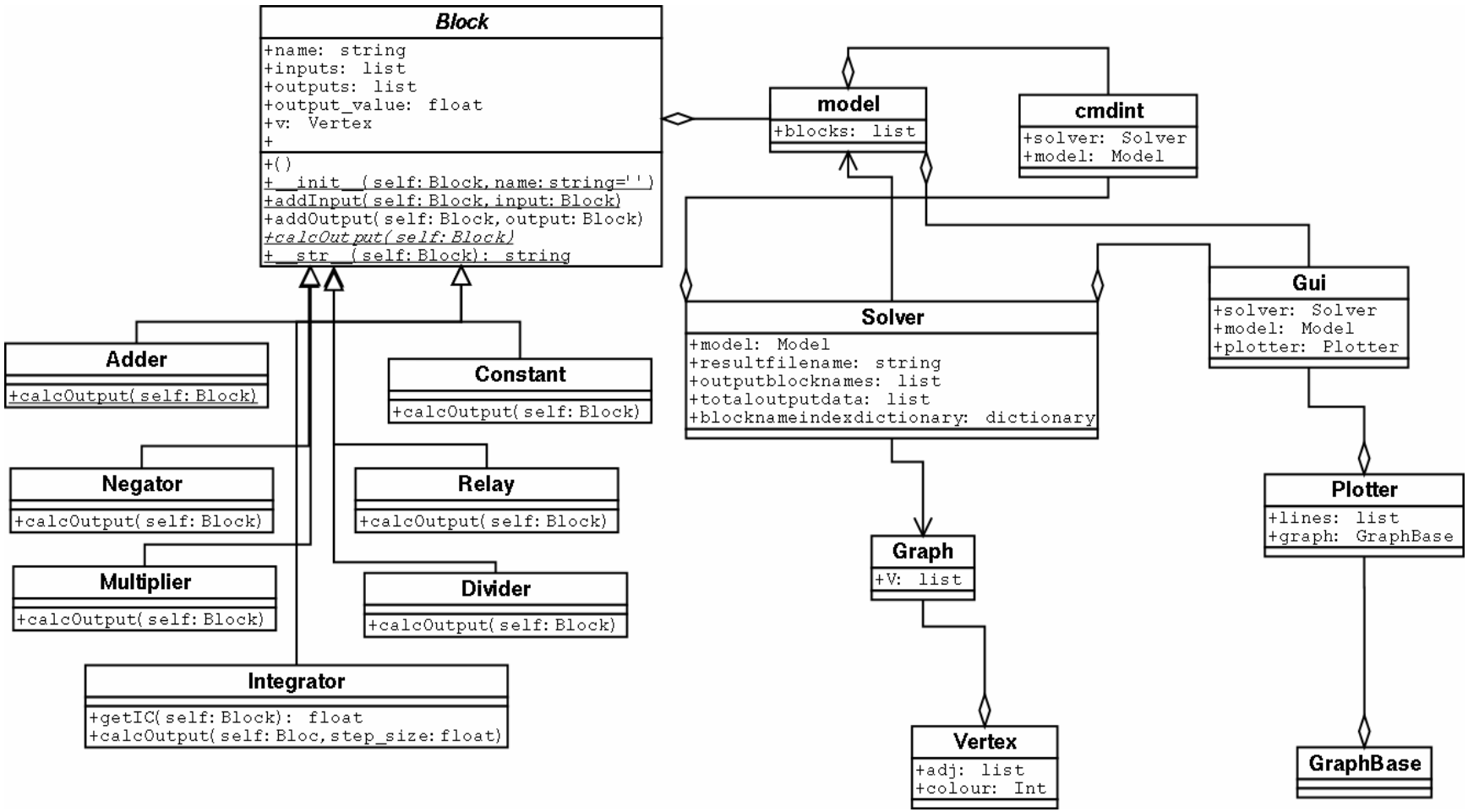
- Software evolves over time.
- Unmanaged change leads to spaghetti code bases where classes are highly coupled to one another.
- This makes them brittle and difficult to understand



So how to we avoid this?

We Architect our System

… with UML

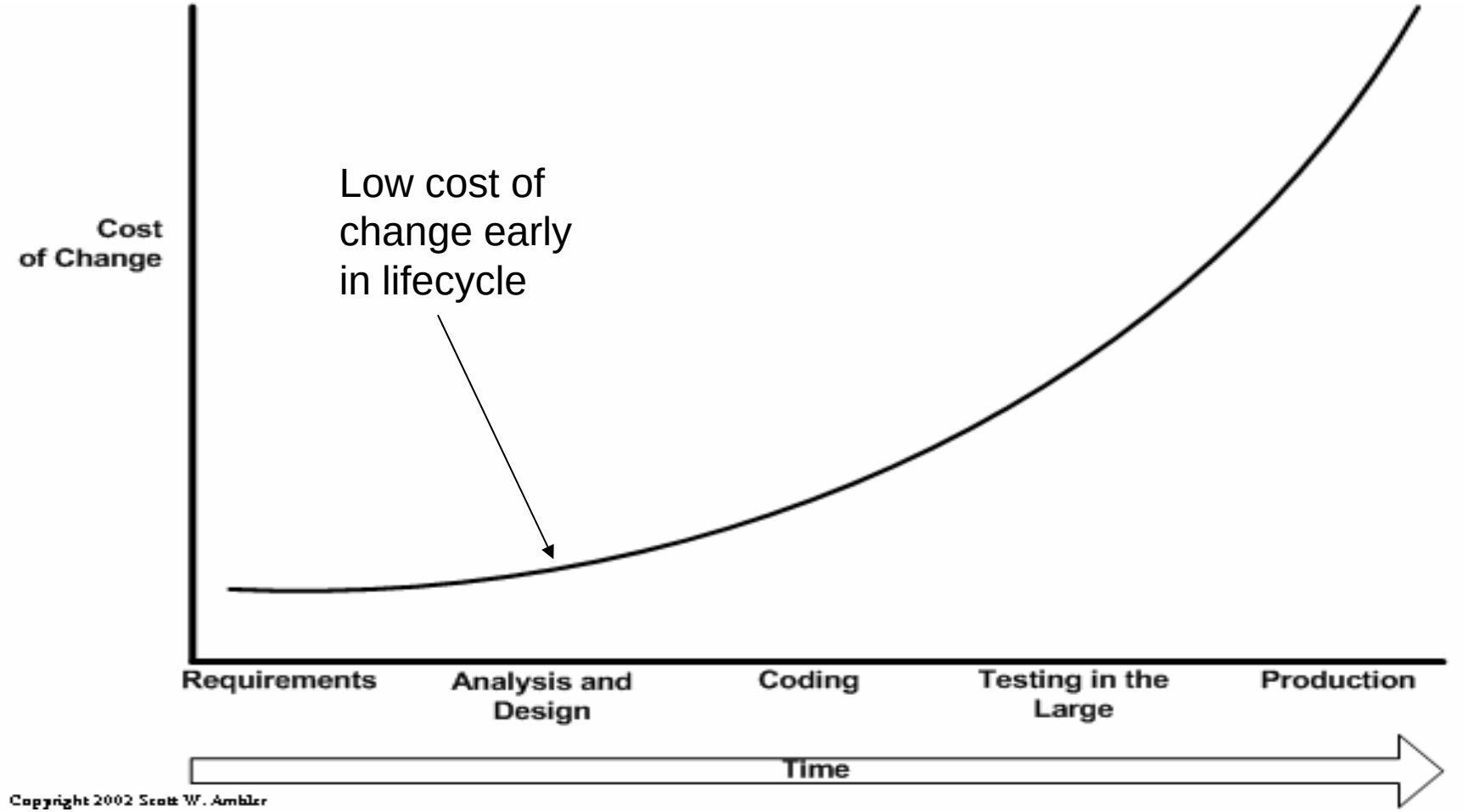

Block
+name: string
+inputs: list
+outputs: list
+output_value: float
+v: Vertex
+
+()
+__init__(self: Block, name: string='')
+addInput(self: Block, input: Block)
+addOutput(self: Block, output: Block)
+calcOutput(self: Block)
+__str__(self: Block): string
model
+blocks: list
cmdint
+solver: Solver
+model: Model
Adder
+calcOutput(self: Block)
Constant
+calcOutput(self: Block)
Negator
+calcOutput(self: Block)
Relay
+calcOutput(self: Block)
Multiplier
+calcOutput(self: Block)
Divider
+calcOutput(self: Block)
Integrator
+getIC(self: Block): float
+calcOutput(self: Bloc, step_size: float)
Solver
+model: Model
+resultfilename: string
+outputblocknames: list
+totaloutputdata: list
+blocknameindexdictionary: dictionary
Gui
+solver: Solver
+model: Model
+plotter: Plotter
Graph
+V: list
Plotter
+lines: list
+graph: GraphBase
Vertex
+adj: list
+colour: Int
GraphBase

AND…

We get to spot problems early on in the project lifecycle.

Why is that advantageous?

Because it costs less



So we have a plan for how to build our application before we start coding.

All we need to do is follow the plan!!

Well this is what we used to do…

…but problems kept cropping up…

It was really hard to get the design right up front.

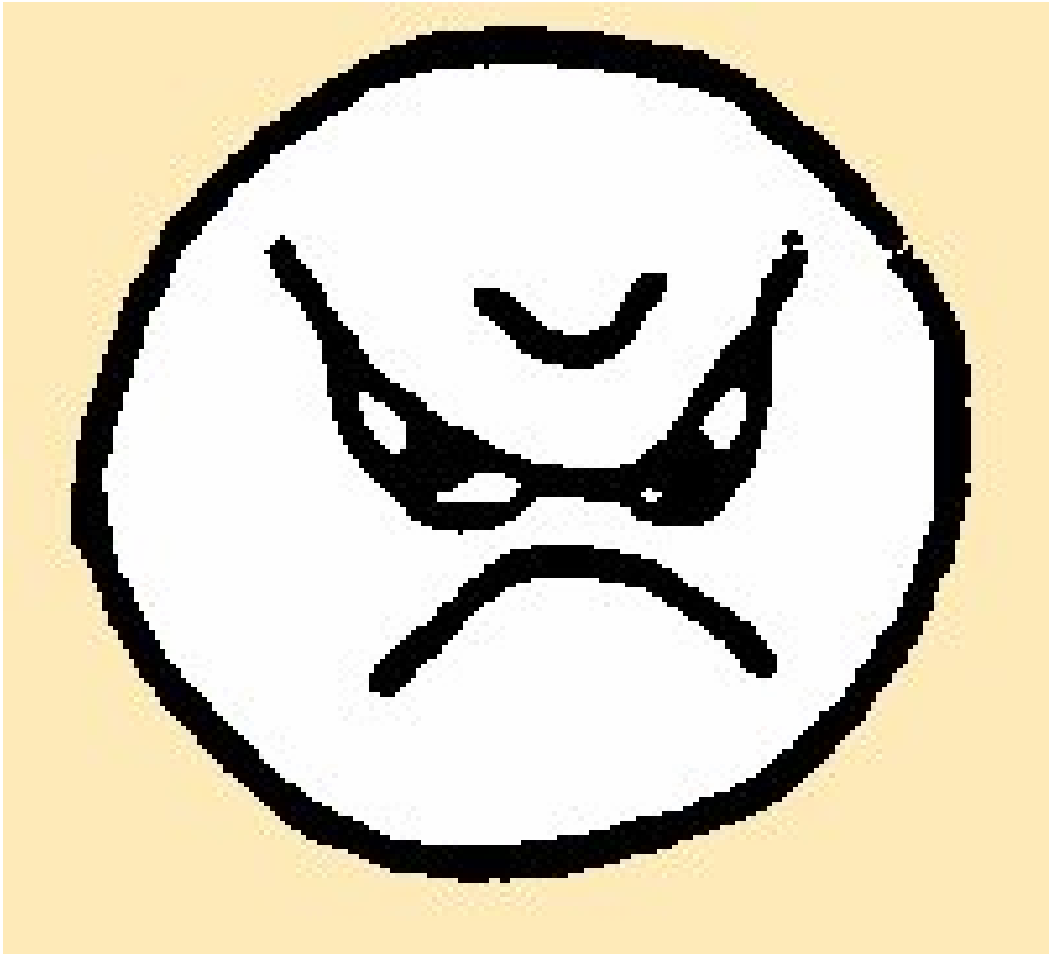
- The problems we face are hard.
- The human brain is bad at predicting all the implications of a complex solution up front.
- When we came to implementing solutions, our perspective would inevitably change and so would our design.

And then…

…when we did get the design right…

…the users would go and change the requirements and we'd have to redesign our model.

Ahhh, those pesky users, why can’t they make up their minds?



So…

…in summary…

We find designing up front hard…

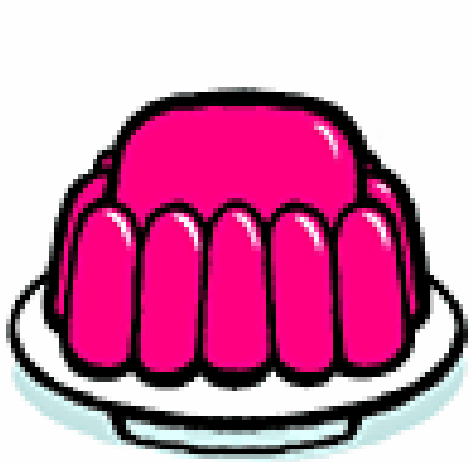
...and a bit bureaucratic

…and when we do get it right the users generally go and change the requirements…

…and it all changes in the next release anyway.

So are we taking the right approach

Software is supposed to be soft.

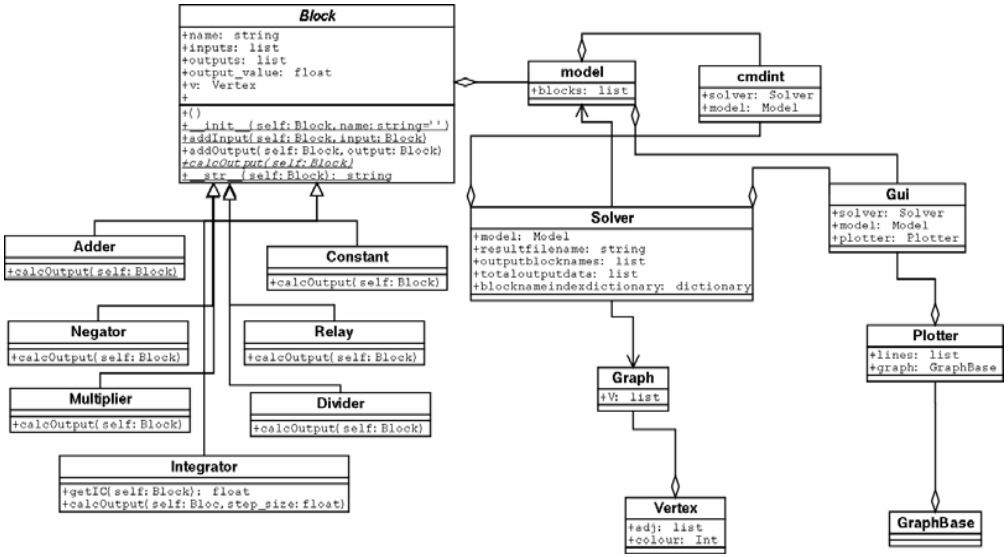


That means it is supposed to be easy to change.

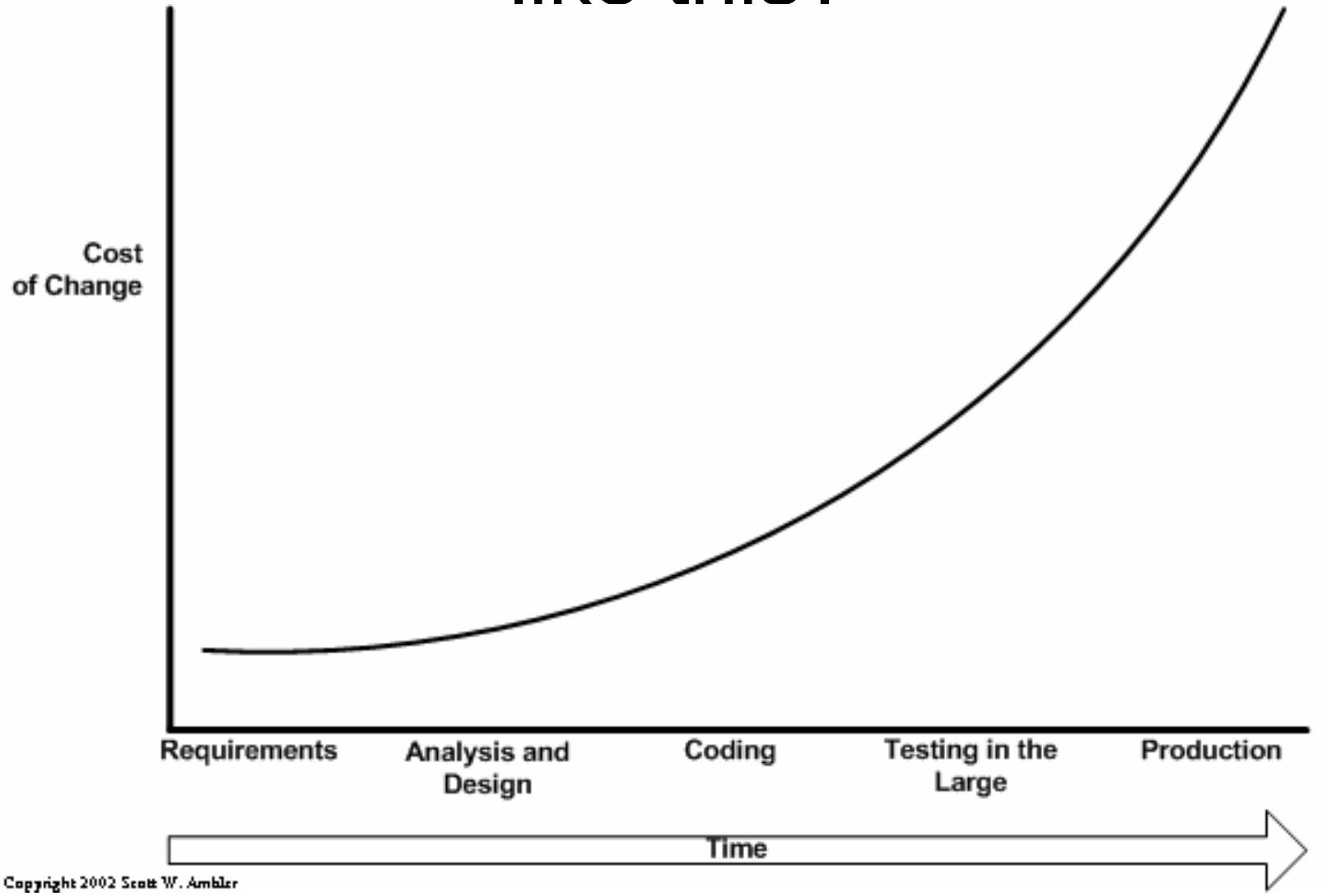
But is it?

- Small application are easy to change.
- Large applications generally are not.

So is fixing the architecture and design up front the right way to do it?

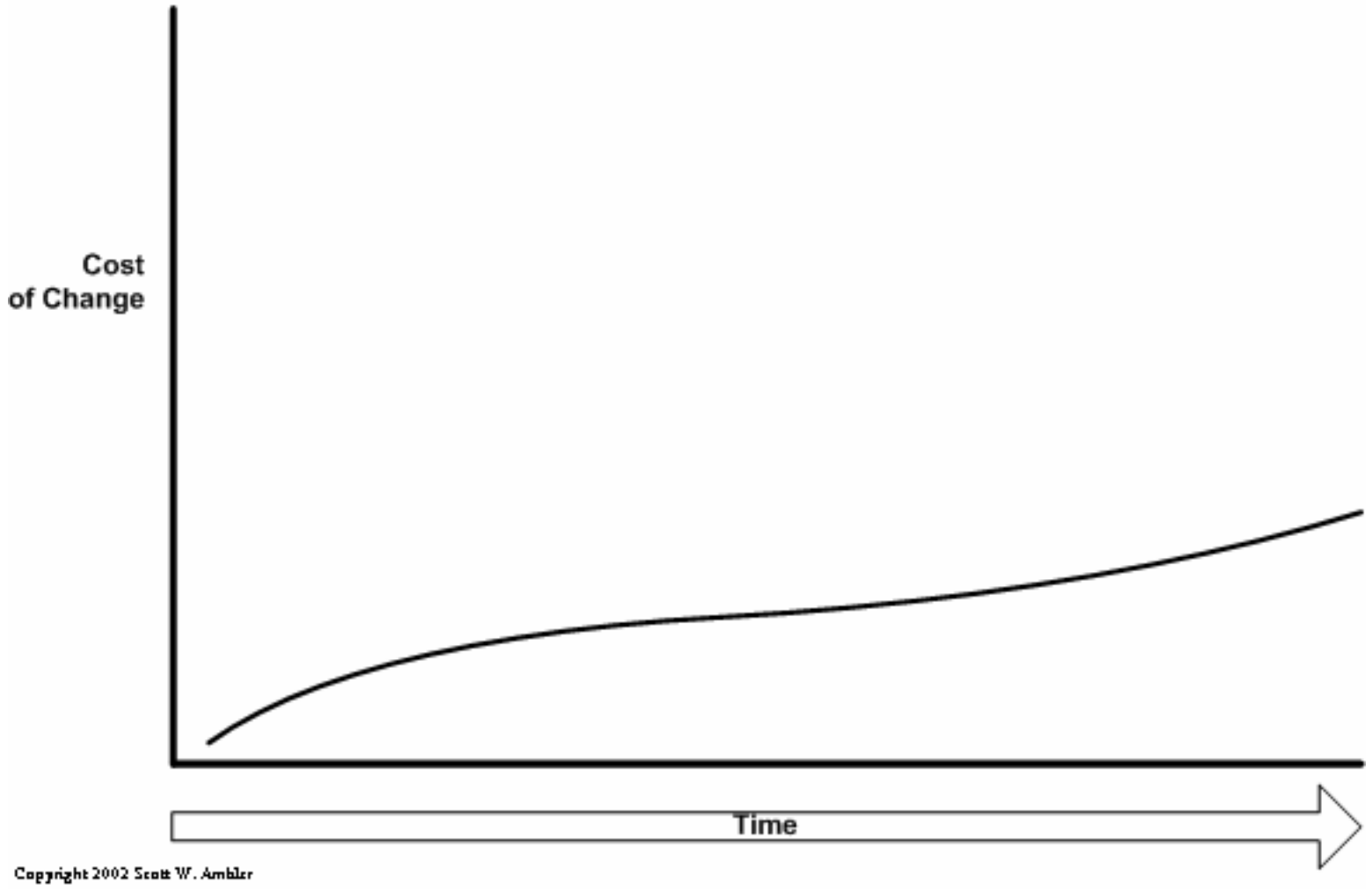


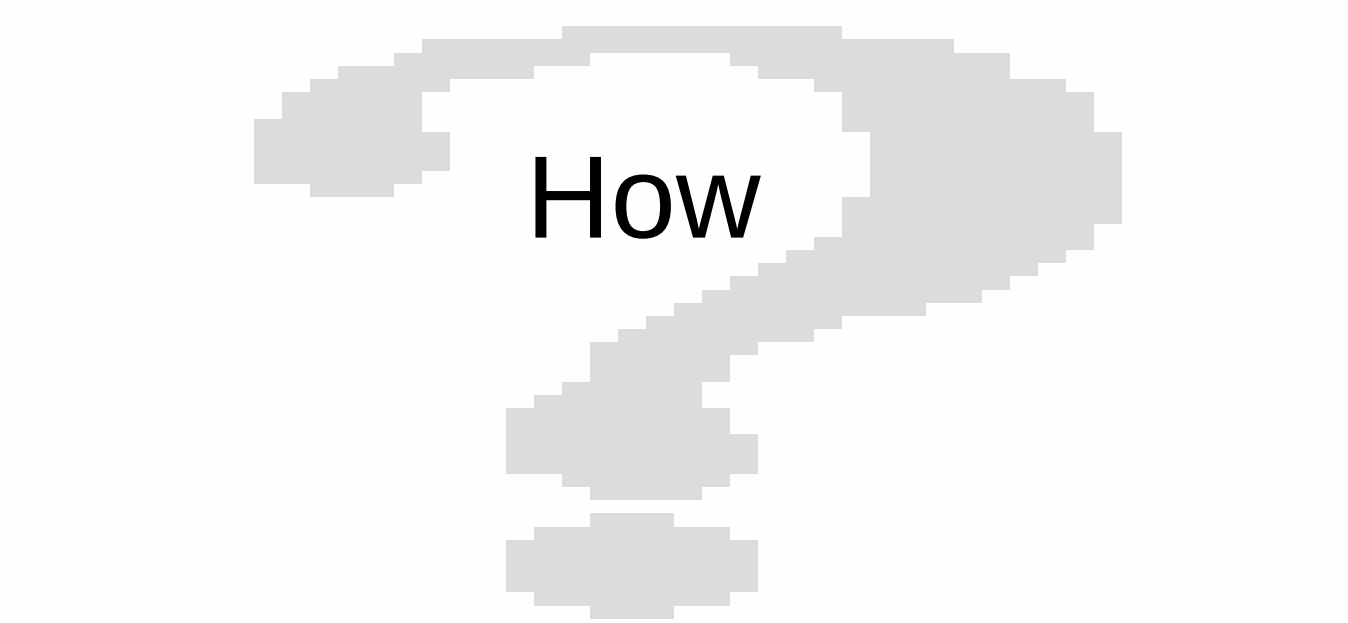
Does the cost curve have to look like this?



Can we design our systems so that we CAN change them later in the lifecycle?

The Cost of Change in an agile application



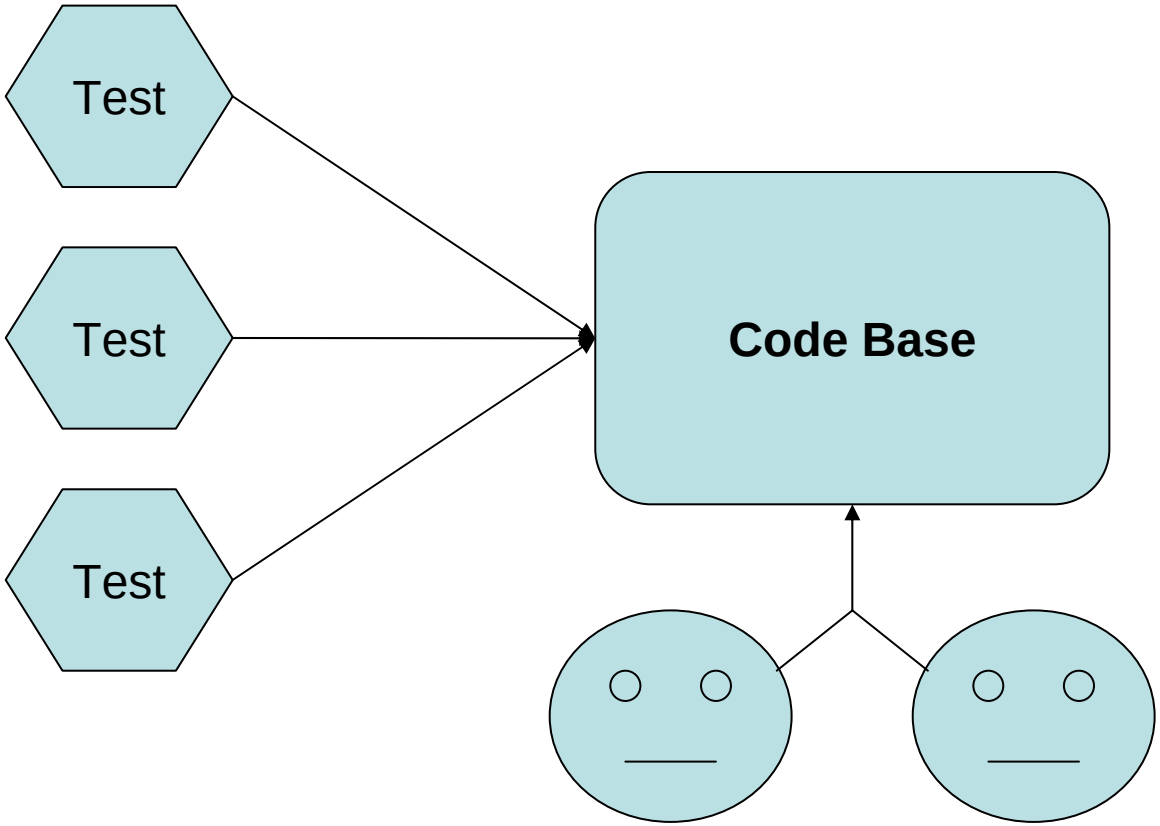

How

By architecting and designing for change

**** But not designing for any specific changes**

***** And writing lots and lots of tests**

Agile Development facilitates this



Dynamic Design

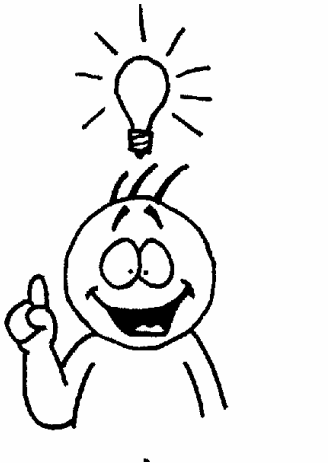
- Similar to up-front design except that it is done little and often.
- Design just enough to solve the problem we are facing now, and NO MORE.
- **Refactor** it later when a more complex solution is needed.

This means that your system’s design is constantly evolving.

Making our key values:

- Changeability – because most software projects involve change.

- Comprehensibility – because the easier it is to understand, the easier it is to change.



So what does this imply for the Architect?

Architecture becomes about steering the application so that it remains easy to understand and easy to change.

With everyone being responsible for the design.

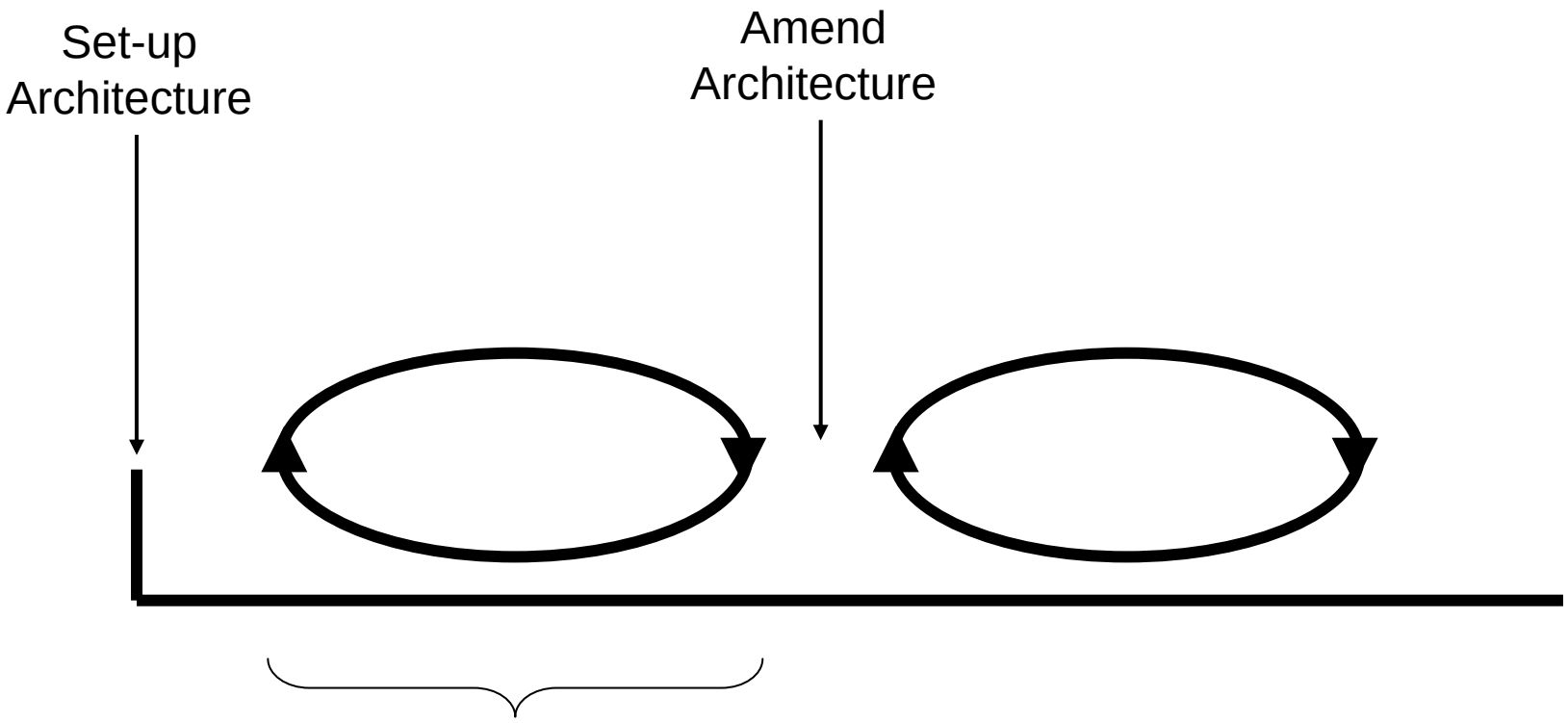
So the architects role becomes about steering the applications design through others.

Shepherding the team!

Shepherding the team

- People will always develop software in their own way. You can’t change this.
- You use the techniques you know to keep the team moving in the right direction.
- Occasionally one will run of in some tangential direction and when you see this you move them back.

Timeline


Set-up
Architecture
Amend
Architecture
Push patterns and reuse
Watch for architectural breakers

Aims:

- Encourage preferred patterns.
- Encourage reuse.

Tools:

- Communication
- Frameworks

1. Architecture through reuse

- Good OO Design
- Common Libraries
- A Domain Model

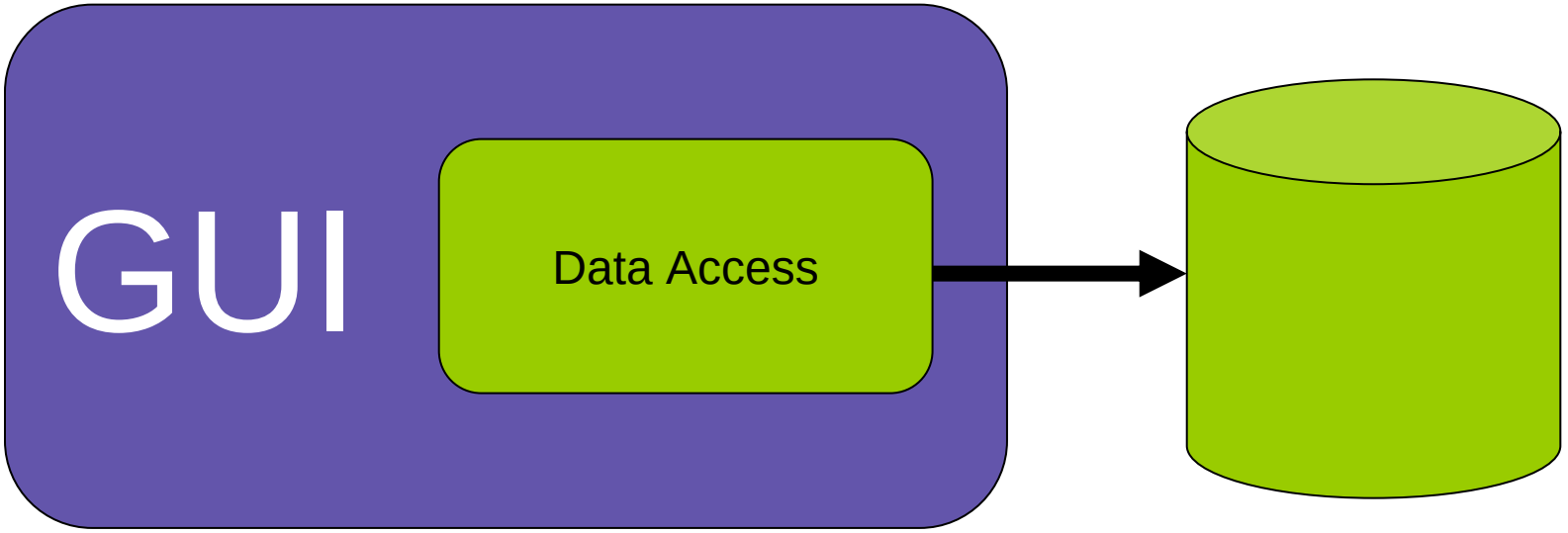
The importance of a Domain Model

- Simulate the business problem in software.
- Separate from any technically implied dependencies.

2. Architecture through patterns

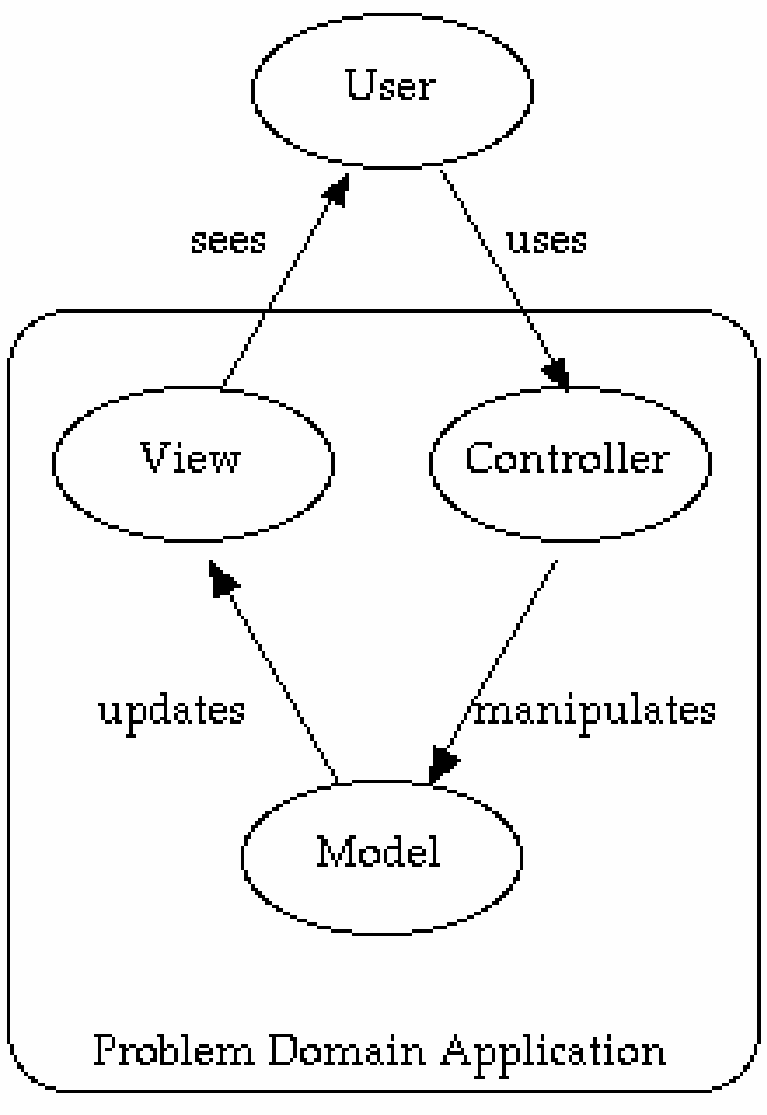
Separation of Concerns:
Layers and Services

Example



- Scaling such a solution is problematic?
- Untangling the database code from the UI code makes each easier to understand. This might not matter for small applications but the effect is very noticeable as the application grows.

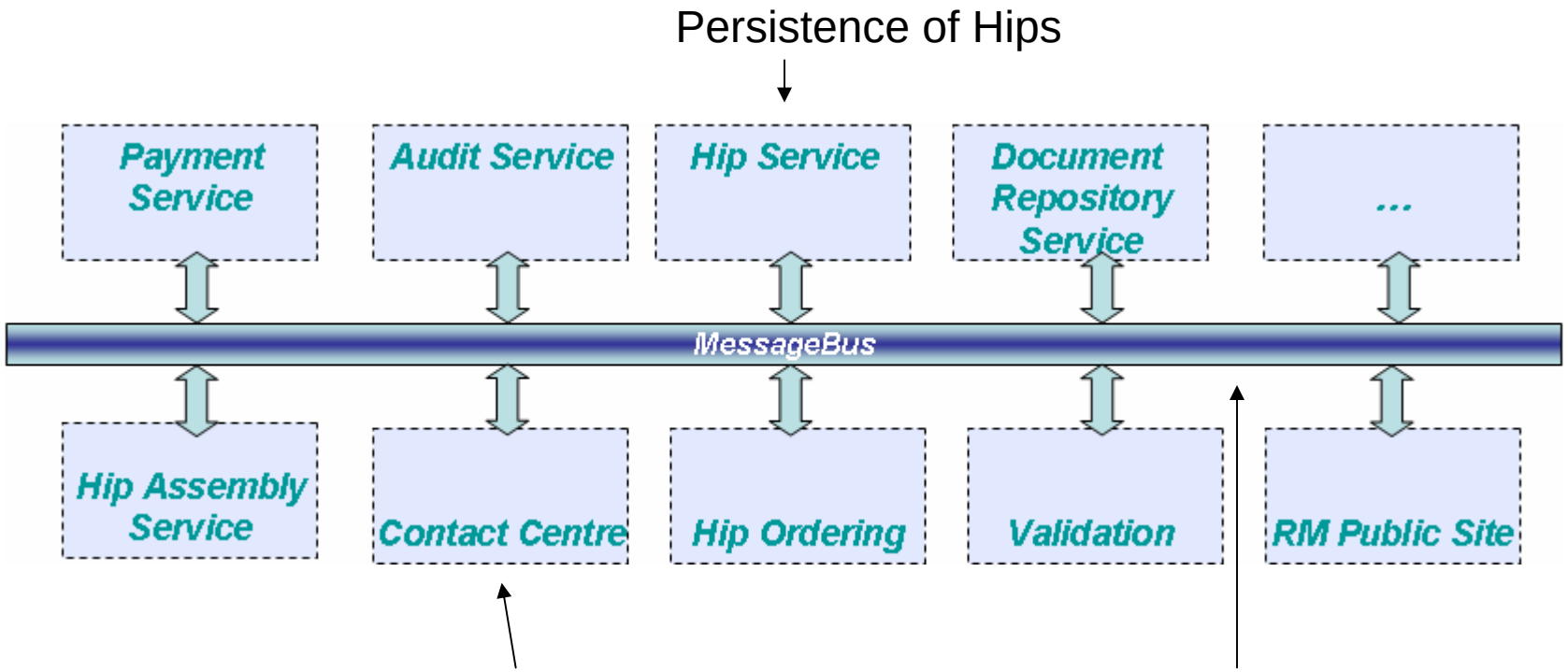
Model-View-Controller



A Real SOA and Layered System

Rightmove.com

Business Services Communicating Asynchronously over a Bus



SOA

- Architectural Pattern (Functional)
- Design Pattern (Technical)

SOA as an *Architectural Pattern*

Provides separation between the implementations of different business services giving:

- Scalability
- Fungibility

SOA as a *Design Pattern*

- Encourages separation of responsibilities into the different services.
- Forces communication between services to be at a business level => Promotes tight encapsulation.
- Asynchronous communication promotes statelessness of services.

So how does SOA compare to a Component Based Model

- CBS using Corba is very similar:
  - Breakdown into component services
  - Language neutral protocol
- The key differences are:
  - Making services valid at a business level with the aim being to integrate across the enterprise (mapping tools can be used to ensure the services match the business model).
  - Communication only through business significant messages – this has interesting architectural implications.

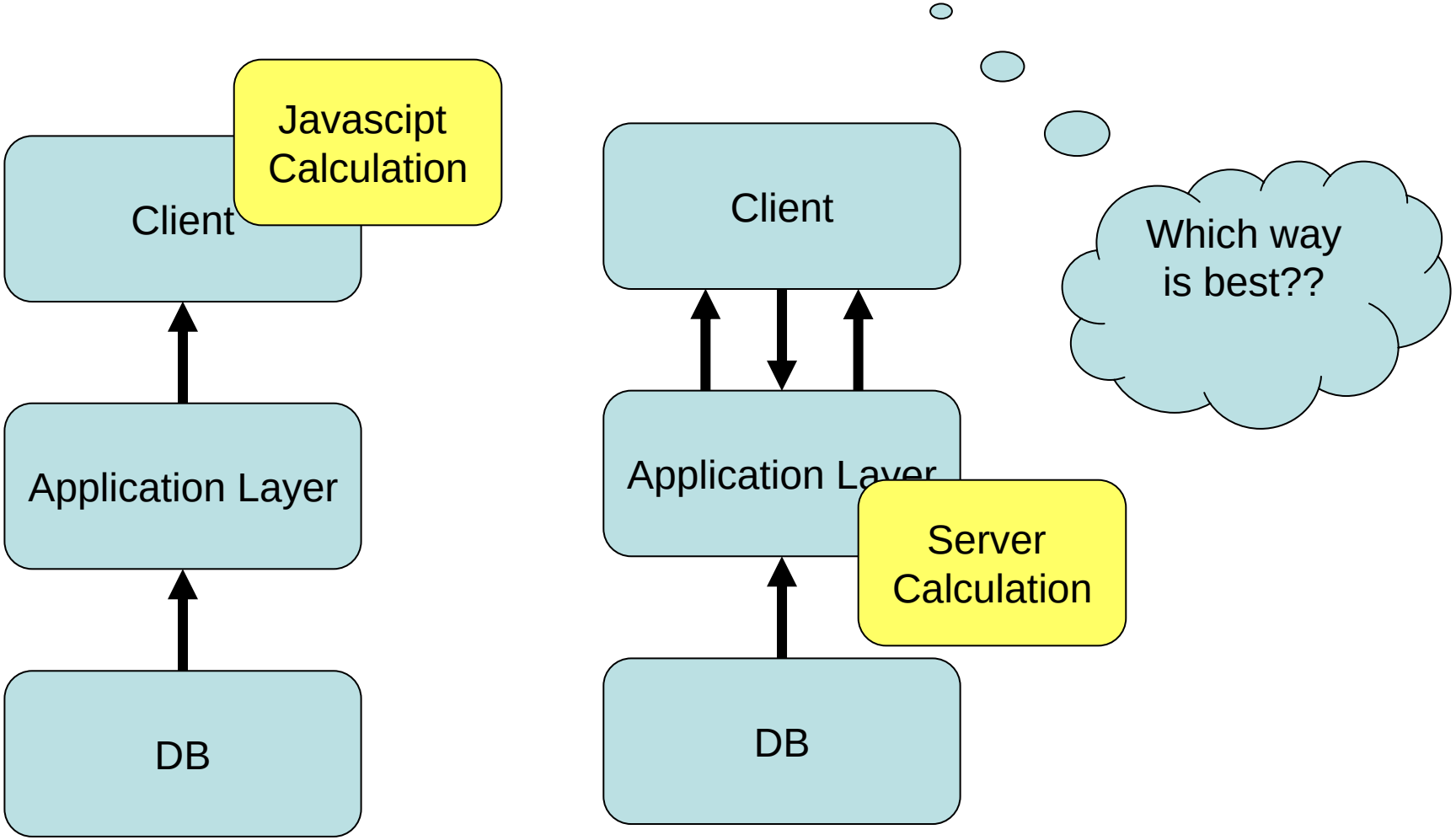
Layers and Tiers

- **Layers/Tiers** provide a pattern that promotes separation between ***technical responsibilities***.
- **Services** provide a pattern that promotes separation between ***functions at a business level***.

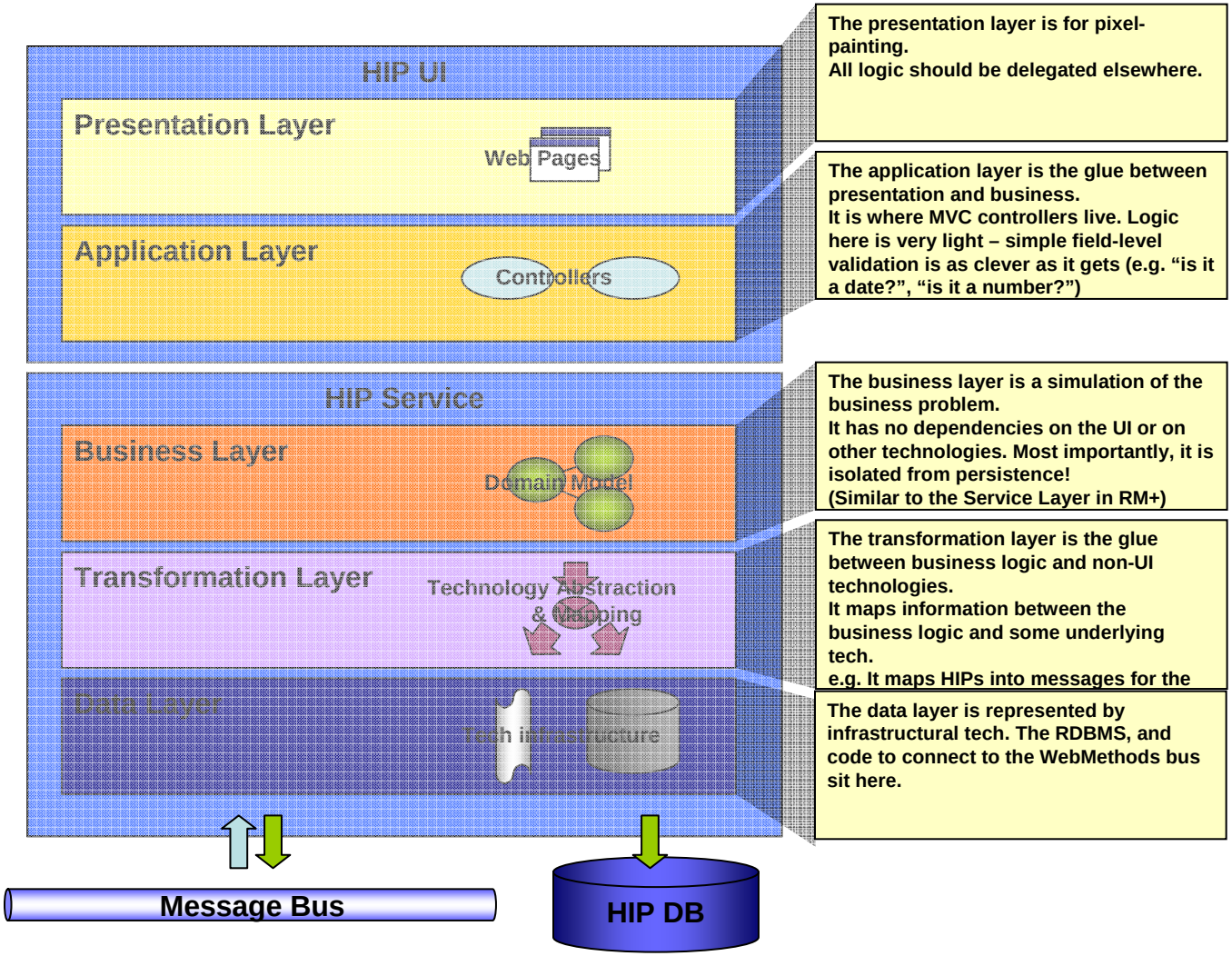
Layers vs. Services

- Services are generally wrapped behind well defined and controlled interfaces.
- Conversely the separation between layers is generally logical.

Aside – Bob's Website



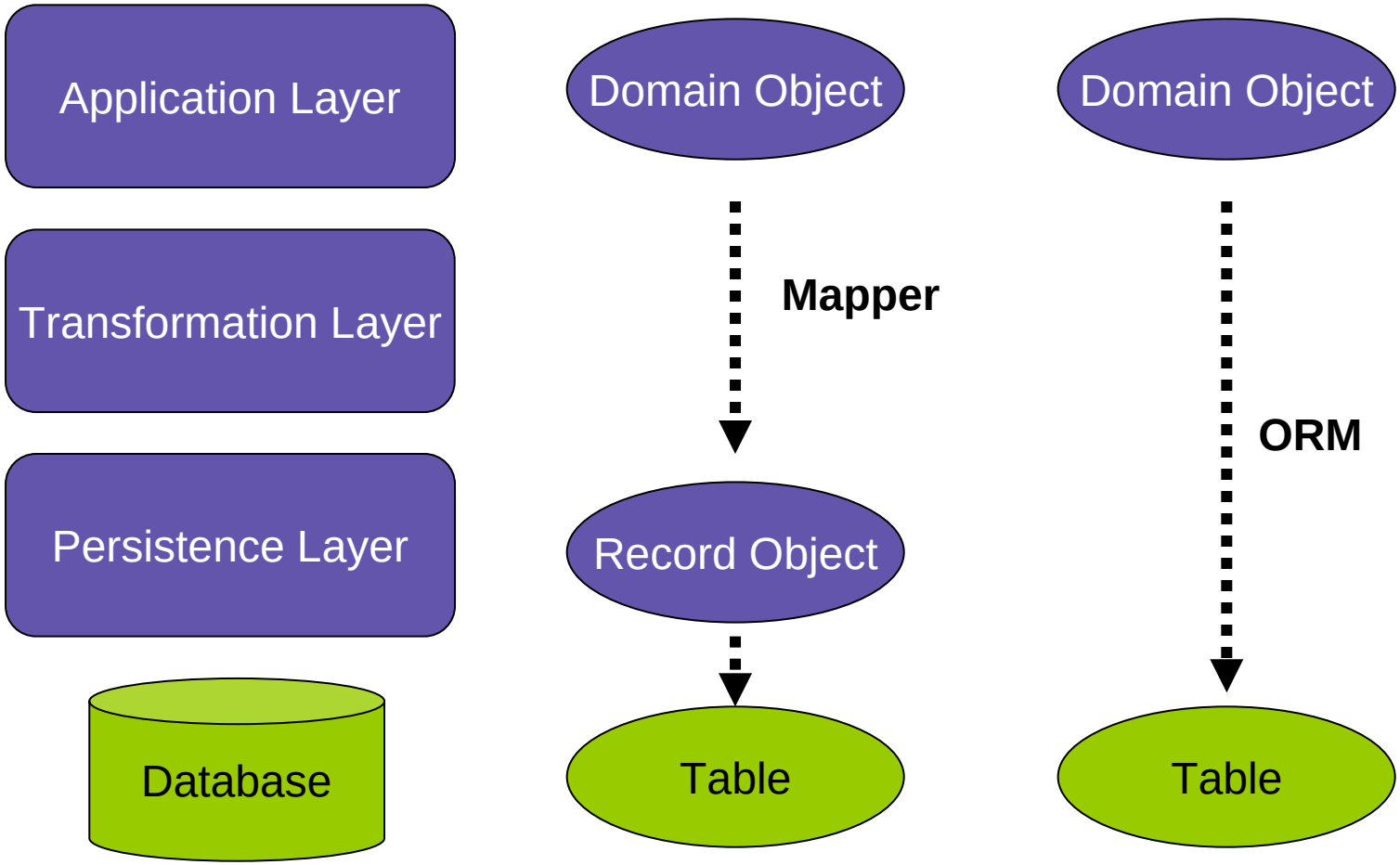
Layered Architecture

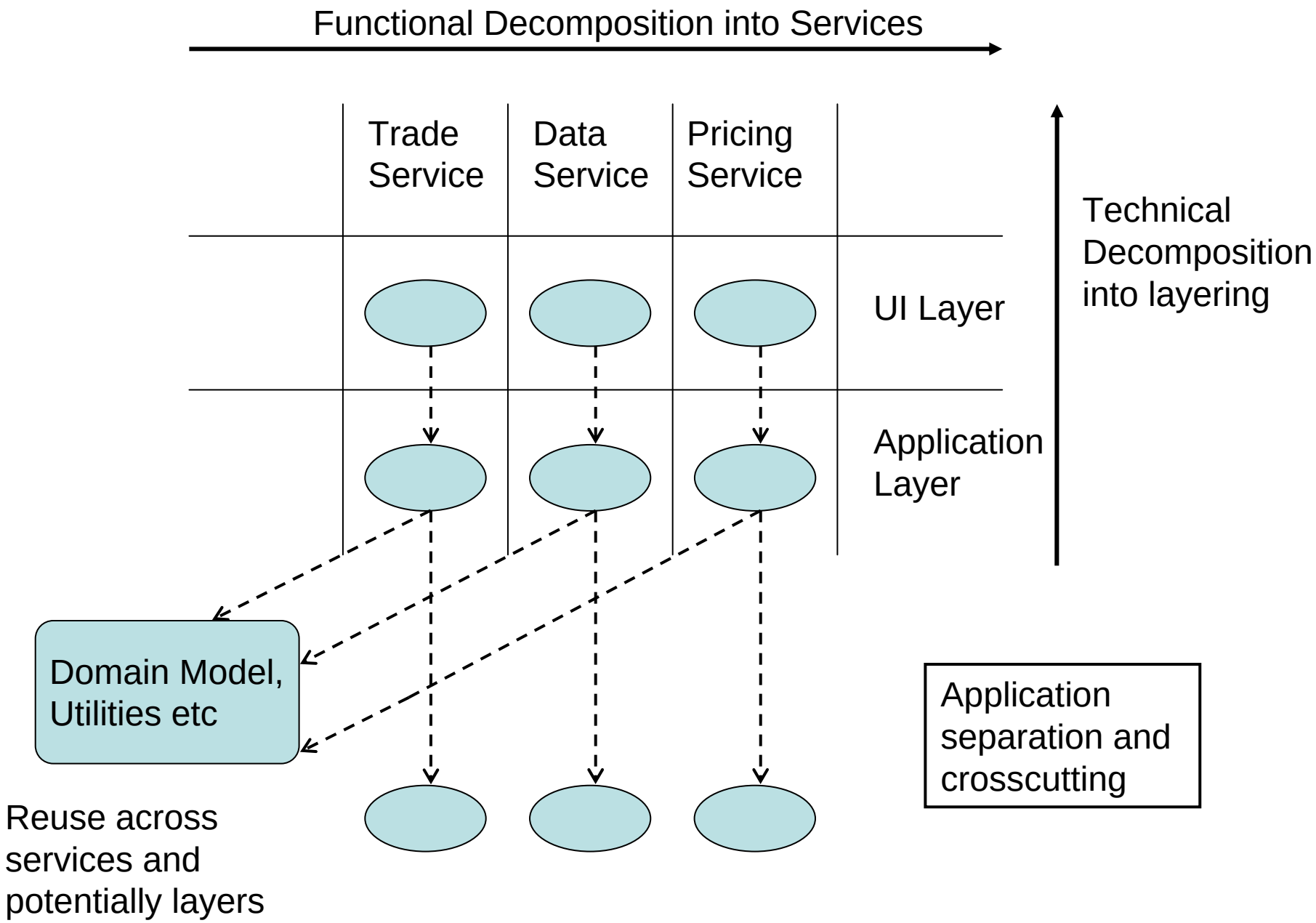


What is the point of having layers?

Separation of Technical Concerns

Example: Is a transformation layer a good idea?




Functional Decomposition into Services
Trade Service
Data Service
Pricing Service
UI Layer
Application Layer
Technical Decomposition into layering
Domain Model, Utilities etc
Reuse across services and potentially layers
Application separation and crosscutting

Questions

- Is reuse across layers and services a good idea or does it break the encapsulation of those services or layers?
- What about if the services span multiple teams?

3. Use of frameworks to enforce design principals

For Example

- Inversion of Control and Spring, Picocontainer
- Functional separation with OO, CBS, SOA
- Layering with Hibernate, IBatis, Webwork

Aside: What is wrong with the singleton pattern?

It’s very hard to test applications when singletons are around

```
public void foo()
   {
      x = Fred.getInstance().getX();
      …
      y = George.getInstance().getY();
      …
      z = Arthur.getInstance().getZ();
   }
```

Dependency Injection

```
public void foo(X x, Y y, Z z)
    {
        ...
    }
```

Spring

- Inversion of control/Dependency injection makes code easy to test.
- Helps you organise your middle tier.
- Get rid of (the static aspects of) singletons.

Config.xml – Define Dependencies

```
<bean id="CurrencySpreadRecordDAO"
 class="com.dkib.gf.dao.CurrencySpreadRecordDAOImpl">
    <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>

<bean id="MarketDataDao"
 class="com.dkib.gf.dao.MyMarketDataFacade">
    <constructor-arg index="0" ref="DrivenPairRecordDAO"/>
    <constructor-arg index="1" ref="VolSmileRecordDAO"/>
    <constructor-arg index="2" ref="SpotRateRecordDAO"/>
    <constructor-arg index="3" ref="VolSpreadRecordDAO"/>
    <constructor-arg index="4" ref="CurrencySpreadRecordDAO"/>
</bean>
```

Spring performs construction

```
public class MyMarketDataFacade implements
    MarketDataFacade {
…
    public MyMarketDataFacade(
            DrivenPairRecordDAO drivenPairsDAO,
            VolSmileRecordDAO volSmileRecordDAO,
            SpotRateRecordDAO spotRateRecordDAO,
            VolSpreadRecordDAO volSpreadRecordDAO,
          CurrencySpreadRecordDAO
    currencySpreadRecordDAO) {
…
```

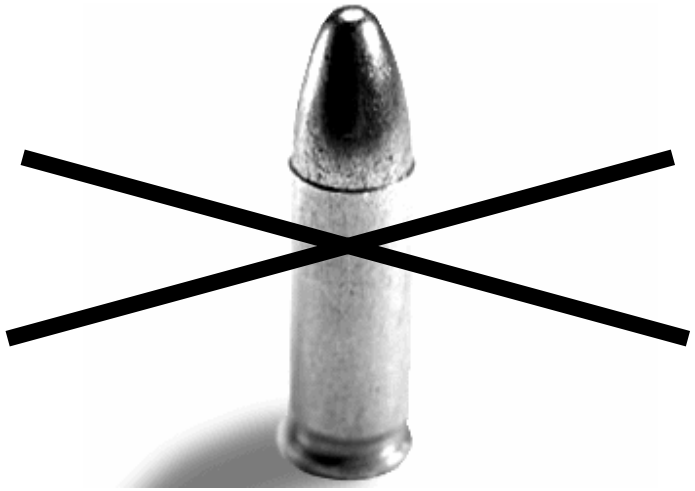
Look Up Service

```
public class DataLayer {
    private final ApplicationContext ctx;

    public DataLayer() {
        ctx = new ClassPathXmlApplicationContext("config.xml");
    }
    public MarketDataFacade getMarketDataFacade() {
        return (MarketDataFacade) ctx.getBean("MarketDataDao");
    }
}
```

But…

These frameworks are not silver bullets.
They each have their own problems.



Avoiding Architectural Breakers

Course grained decisions that are hard to refactor away from. For example embedding business logic in a UI.

Summary So Far

- Software is soft so design is an evolving process not a prescribed one.
- Architecture is about controlling the limits of design.
- Architecture is also about pushing a group of developers in a certain direction. It is a soft skill as much as a technical one.
- Patterns and frameworks are the tools the architect uses to do this.

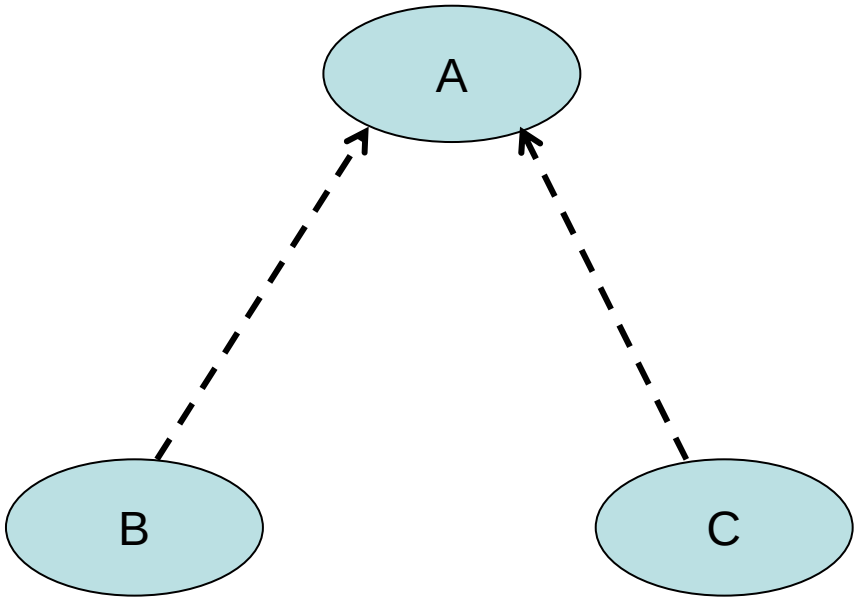
How design can go wrong

The overuse of design patterns

=> Obtuse code

Fragile Base Class Problem

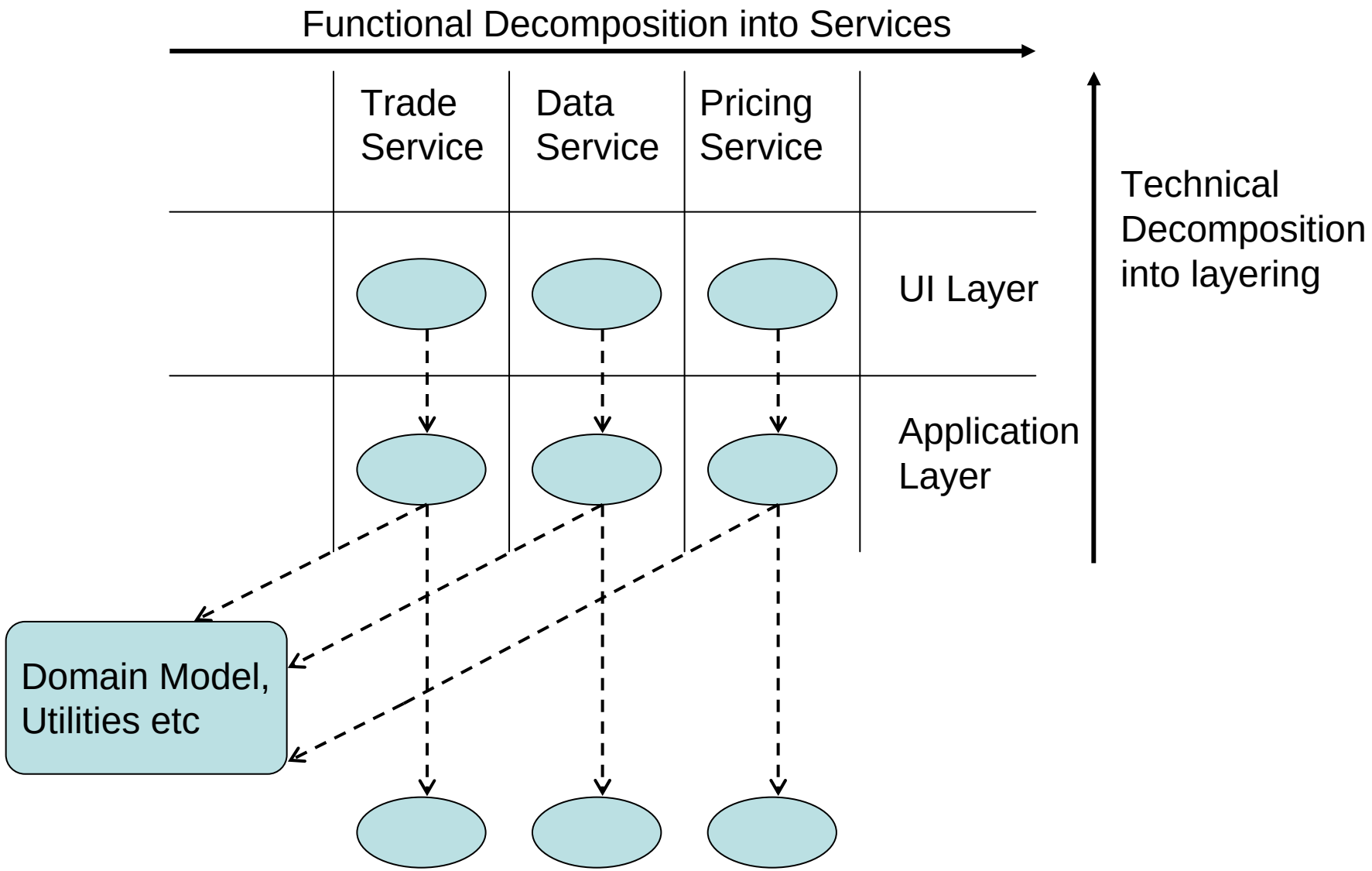
- Why does this occur?



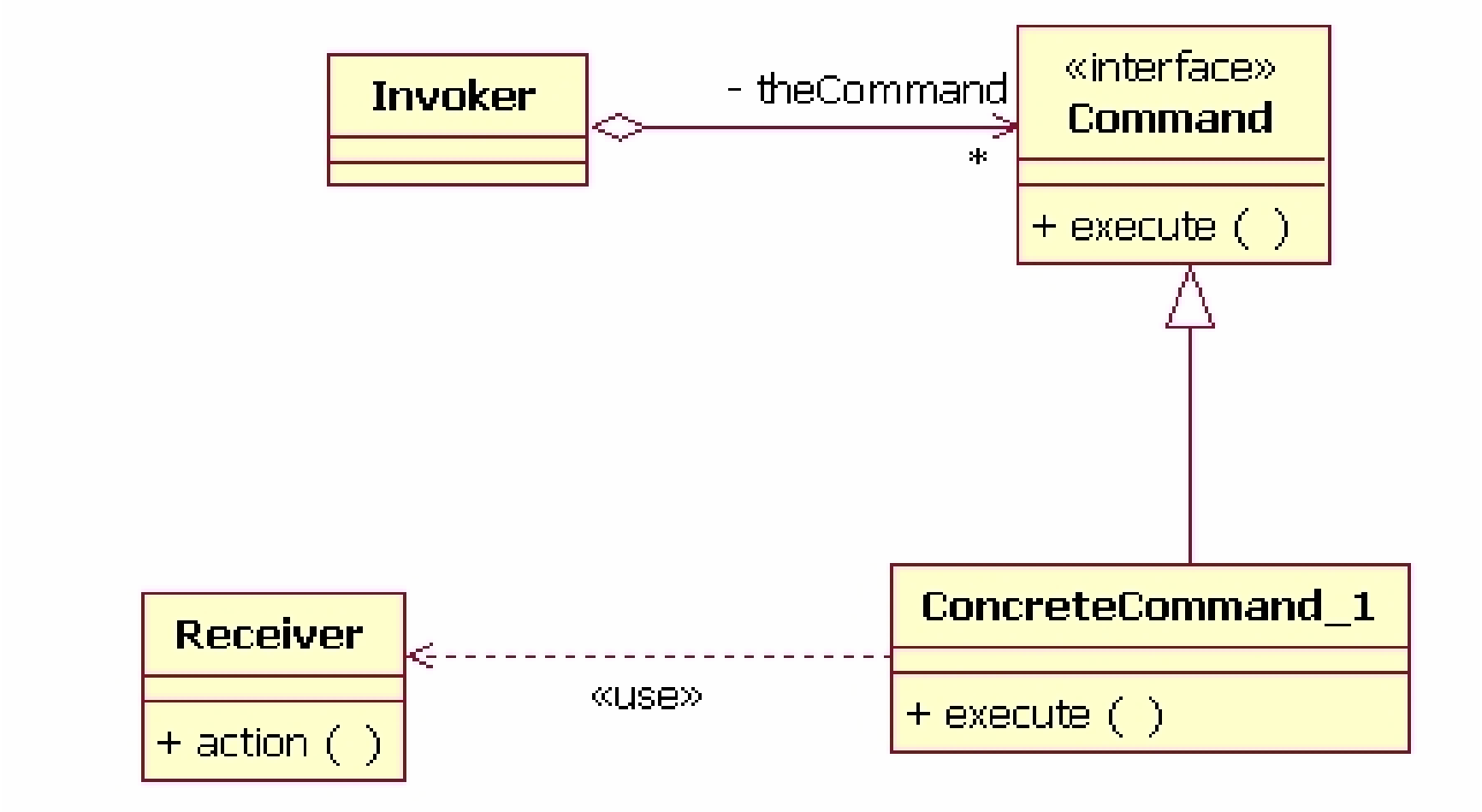
Solutions

- Favour composition over inheritance.
- Superclasses should call subclasses not the other way around

Service Duplication Problem



Command/Executor Problem



Solution

Be wary of functional decomposition and its tendency to push you away from reuse

In Conclusion

- Comprehensibility is the goal of design (followed by changeability).
- An architects role is primarily one of communicating and coordinating a common vision.
- If design is to be dynamic unit tests are mandatory.

And finally…

Never listen to an architect who does not write code.

Conversely if you are an architect, make sure you get your hands dirty.