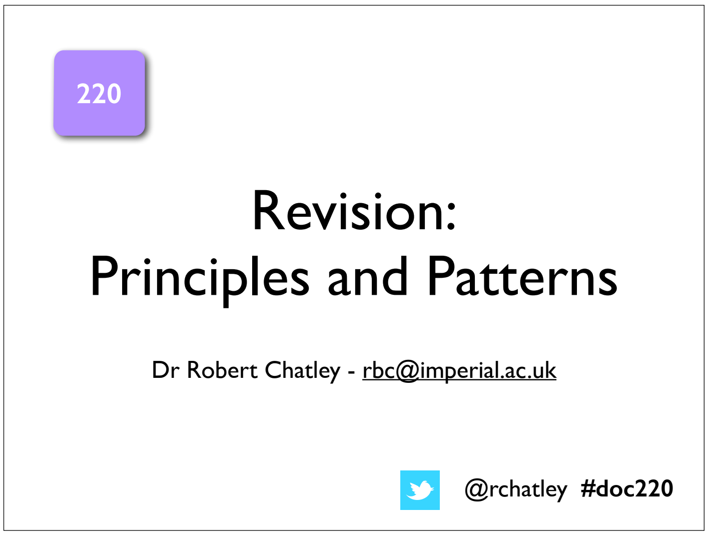
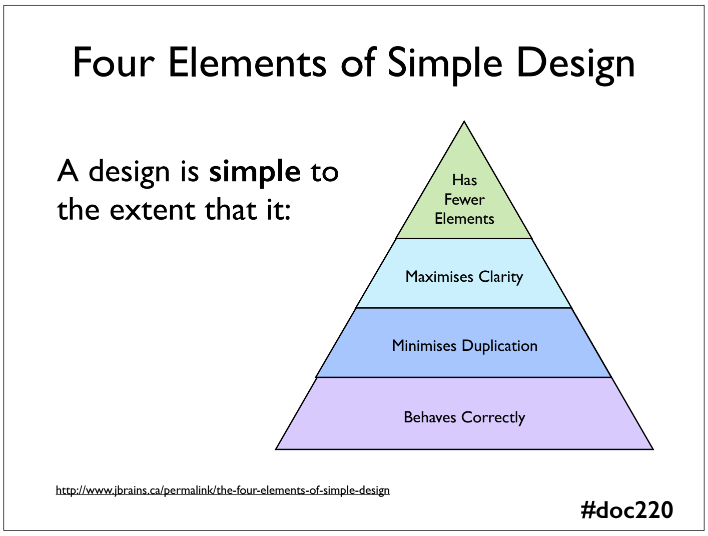


220

# Revision: Principles and Patterns

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley **#doc220**



## Four Elements of Simple Design

A design is **simple** to the extent that it:

- Has Fewer Elements
- Maximises Clarity
- Minimises Duplication
- Behaves Correctly

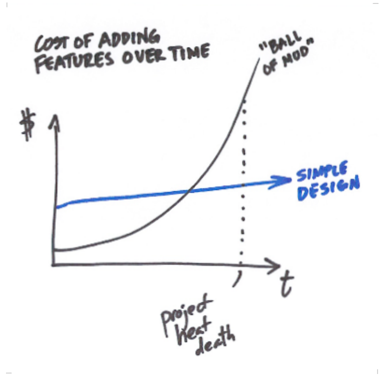
http://www.jbrains.ca/permalink/the-four-elements-of-simple-design

**#doc220**

In software, J.B. Rainsberger sets out his four elements of simple design as follows. A design is simple to the extent that it 1) passes its tests 2) minimises duplication 3) maximises clarity 4) has fewer elements.

Minimising duplication helps to make the design more maintainable and hence more robust. If code is duplicated, then making a change to it may mean making the same change in multiple places, which is more work than we would like. By aiming to maximise clarity, we work to improve comprehension of the design by humans. Rainsberger's fourth principle - to reduce the number of elements - aims to make the overall size of the system smaller, making it simpler and easier to comprehend.

## Cost of Change



http://www.jbrains.ca/permalink/the-three-values-of-software

**#doc220**

In the article linked on the slide, J.B. Rainsberger refers to design as being one of the three values of software. Without design, the marginal cost of adding a new feature will gradually increase until it becomes too hard to add anything new. Paying attention to improving the design over time will allow us to evolve and maintain the software effectively.

This is one of the most important aspects of software design. As Sandi Metz writes in her book "Practical Object Oriented Design in Ruby": "Changing requirements are the programming equivalent of friction and gravity. They introduce forces that apply sudden and unexpected pressures that work against the best-laid plans. It is the need for change that makes design matter".

## What is **Bad** Design?

Rigidity

Fragility

Immobility

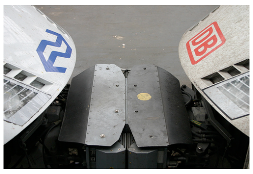
**#doc220**

We would obviously like our system to have a "good" design. But what might happen if we have a "bad" design? Robert C. Martin identifies a number of problems that a badly design software system may have.

Rigidity - where software is hard to change
Fragility - where when we change one part, other parts break unexpectedly
Immobility - where it is hard to reuse elements of the code in other applications

## Coupling and Cohesion

Aim for **low coupling** between classes



Aim for **high cohesion** within each class

**#doc220**

The web of objects should be as independent as possible. Sometimes you see systems that are written in an object-oriented language, but where objects do not have clear and separate responsibilities, and all the objects are connected tightly together, sharing data, and without a clear structure. Sometimes objects are very large, and have many different behaviours. These are not good traits. We would rather reduce the coupling between objects, have them know less about each other, and co-operate only by the exchange of messages.

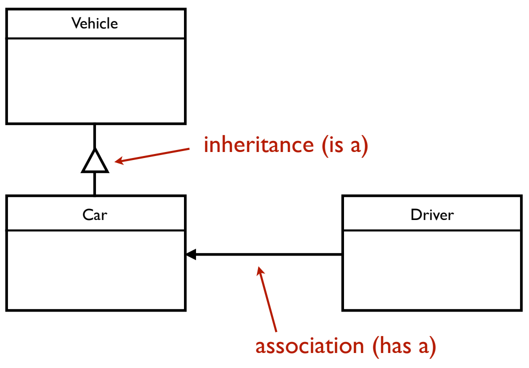
## Principles of OO Design

Program to an interface, not an implementation
Separate things that change from things that stay the same
Prefer composition over inheritance
Delegate, delegate, delegate

**#doc220**

From the patterns books we can extract four core principles of OO design. We will refer back to these a lot as we look at different design problems, and see how trying to follows these principles influences our design choices.

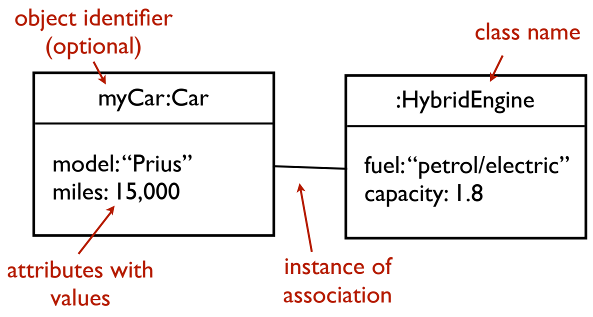
## UML: class diagram



#doc220

The UML class diagram shows the relationships between different classes in an object-oriented system. It uses the triangle to indicate inheritance, and lines between peers to indicate associations, which can be directional - the dependency indicated by an arrowhead.

## UML: object diagram

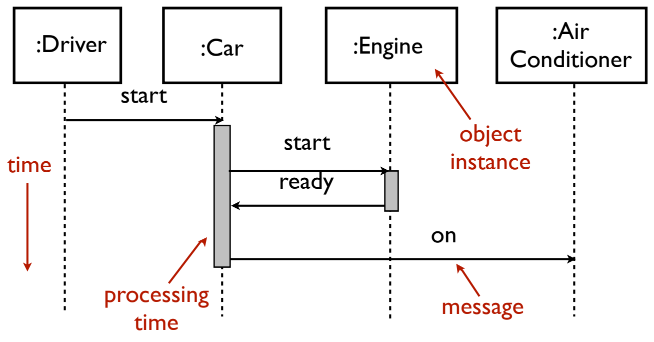


We may well have multiple instances of the same type in the same diagram

#doc220

An alternative, or supporting, type of UML diagram is the object or instance diagram. Rather than showing relationships between classes, this shows the relationships between, and the values held by, particular instances of classes. An object diagram may show how several instances of the same type interact, and give examples of particular values in their fields.

## UML: sequence diagram



#doc220

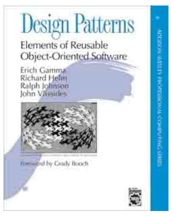
Both class and object diagrams show a snapshot at a particular moment in time. The relationships between objects (if not classes) may vary often during the execution of a program, but these diagrams do not capture any of this dynamism. An alternative style of diagram is the sequence diagram, which again shows object instances, but also shows the interactions between them, but showing the messages that they send and receive between one another over time.

## Design Patterns

Original "Gang of Four" (GoF) book published 1994 - still very relevant

Presents 23 patterns, common object structures that solve particular problems

Examples in C++ and Smalltalk



#doc220

Following on from Alexander's work, the book Design Patterns, commonly known as the Gang of Four book, contains a number of patterns that are possible solutions to frequently occurring problems in object-oriented design. The Gang of Four book mostly has examples in C ++ and in SmallTalk, but the patterns apply to any object-oriented design.

## Pattern Descriptions

**Name:** handle for design problem/solution
**Problem:** intent, motivation, applicability
**Solution:** elements, relationships, collaborations
**Consequences:** results and trade-offs

GoF has a very detailed pattern description format with many sections

#doc220

There are now a lot of patterns books, but all follow a similar form for describing patterns. They give the pattern a name, so that people can refer to it and talk about it with other engineers, and show a problem that the particular pattern might solve, and example of how the solution might be implemented, and any consequences of using this pattern.

## Behavioural Patterns

**Template Method** extend via inheritance

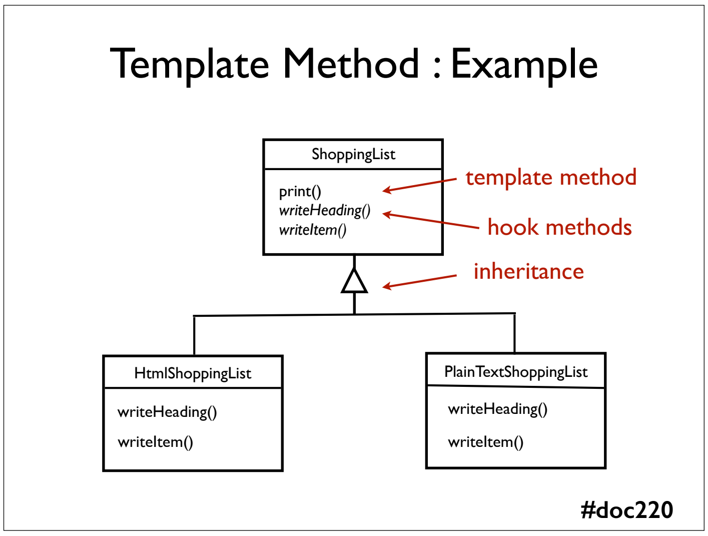
**Strategy** extend via delegation

**Iterator** process elements of aggregate

**Observer** subscribe to changes of state

**Command** package behaviour to execute later

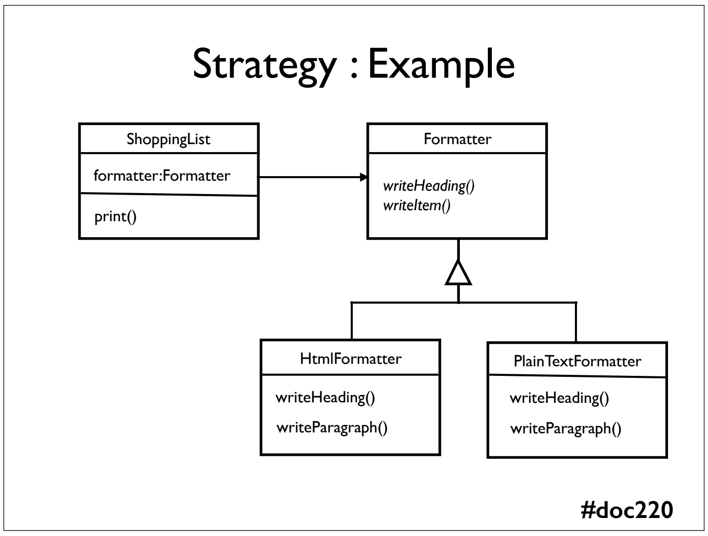
#doc220



## Template Method : Example

#doc220

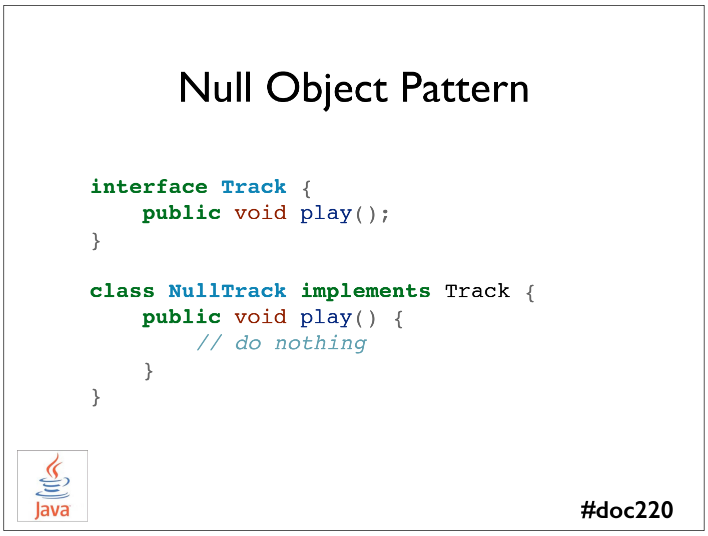
We define the main part of the algorithm in the superclass, but call out to some abstract hook methods, which will be overridden in subclasses to specify particular behaviour for the different variants. Here we separate code that stays the same for all number sequences (in the superclass) from code that varies between the different types, which we put into the subclasses.



## Strategy : Example

#doc220

We can see that we still have an inheritance tree, but this is just between the different strategies (in the example above for formatting a shopping list), not between different types of shopping list itself. This makes it much easier to, in future, apply these different types of formatter to other objects, such as Emails, Newsletters, Reports etc etc.



## Null Object Pattern

```
interface Track {
    public void play();
}

class NullTrack implements Track {
    public void play() {
        // do nothing
    }
}
```

#doc220

A way of solving this more cleanly is never to return null from a method. Instead we attack the problem using polymorphism, and instead of returning a null value, we return an object which when we call it, does nothing. This is the Null Object Pattern. Here we implement a NullTrack. It implements the same interface as a normal Track, but when we call play() it just does nothing, but no errors are caused. When we can't find a track in the library we return a NullTrack.

So, from the caller's point of view, all Tracks are the same, and there is no need to check for null as a special case. The calling code goes back to the original neater form.

## External Iterator

```
List<String> list = Arrays.asList("quick", "brown", "fox");

void processList() {

    for (Iterator<String> i = list.iterator(); i.hasNext(); ) {
        String element = i.next();
        process(element);
    }
}
```

Alternatively we can use the same iterator methods, but back in a for() structure. This is often a better formation, as the scope of the iterator variable is local to the loop, whereas in the while() example it is in scope for the rest of the method.

## Internal Iterator

```
square :: Int -> Int
square x = x * x

map square [1,2,3,4]
```

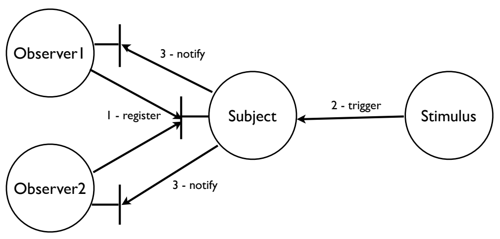
```
def square(x)
  x * x
end

list = [1,2,3,4]
list.map { | x | square x }
```

Ruby

In Haskell we can use a higher-order function - map - define the square function as a first-class entity, and pass it to map as a parameter, for map to apply it to every element of the list. In this way the square() function and the map() function are defined separately and can be used separately.

In Ruby a method isn't a first class citizen like it is in Haskell, but we can pass a *block* as a parameter to another method. In the example above the block is defined in curly braces and says: given an x, square that x. The list type defines a map method that takes the block, applies it to each element, and returns a new list.

## Observer

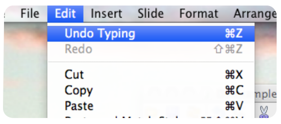


Commonly used with GUI components:
do X when someone presses button Y

#doc220

Here is another style of diagram showing the communications between the observer and the subjects.

## Command



Wrap up a piece of behaviour in a object
to do now, or later, or repeatedly.
Operations may be queued or undone.

A command is an instruction to do something.

#doc220

Both Runnable and Callable are examples of a device where we can wrap up a piece of behaviour in an object, so that we can pass that behaviour around, execute in a different context etc. We can also append command objects to a queue or log, so that they can be executed in a batch, or we can keep a history of past commands executed to replay them later (for example as an undo/redo function).

## Creational Patterns

**Factory Method** allows naming constructors

**Builder** collect object’s configuration parameters

**Singleton** ensure only one instance exists

#doc220

## Factory

Factory methods can have names,
unlike constructors

```
class TimeZone {
  static TimeZone forId(String id) {...}
  static TimeZone forOffsetHours(int offset)
   { return new TimeZone(3); }
}

TimeZone.forId(“Europe/London”);
TimeZone.forOffsetHours(3);
```

instead of:

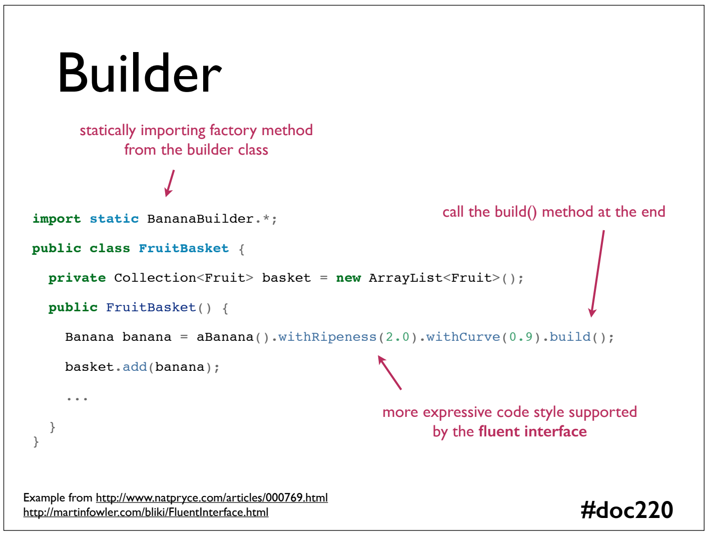
this version has a lot less **clarity of intention**

```
new TimeZone(3);
```

http://joda-time.sourceforge.net/apidocs/org/joda/time/DateTimeZone.html

#doc220

The can do this by applying the *factory* pattern. We can add a factory method to a class to use instead of a constructor. This is a named static method on a Java class that internally calls the constructor and returns the object. We may have many factory methods with different names. We can prevent use of the new operator with this class by making the constructors themselves private.



## Builder

statically importing factory method from the builder class

call the build() method at the end

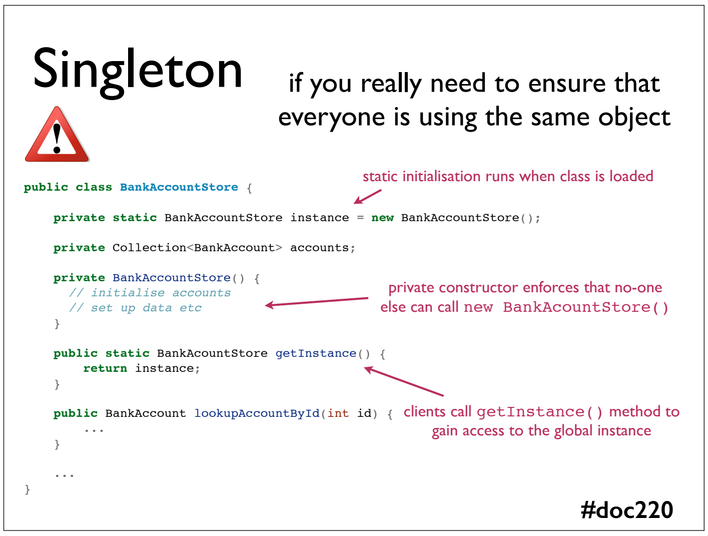
```java
import static BananaBuilder.*;

public class FruitBasket {

  private Collection<Fruit> basket = new ArrayList<Fruit>();

  public FruitBasket() {

    Banana banana = aBanana().withRipeness(2.0).withCurve(0.9).build();

    basket.add(banana);

    ...

  }
}
```

more expressive code style supported by the **fluent interface**

Example from http://www.natpryce.com/articles/000769.html
http://martinfowler.com/bliki/FluentInterface.html

**#doc220**

In the usage example we see how we can chain the calls to the *with* methods together, as each of them returns the receiver this. This is known as a fluent interface. At the end of the chain we call build() to create the object.



## Singleton

if you really need to ensure that everyone is using the same object

static initialisation runs when class is loaded

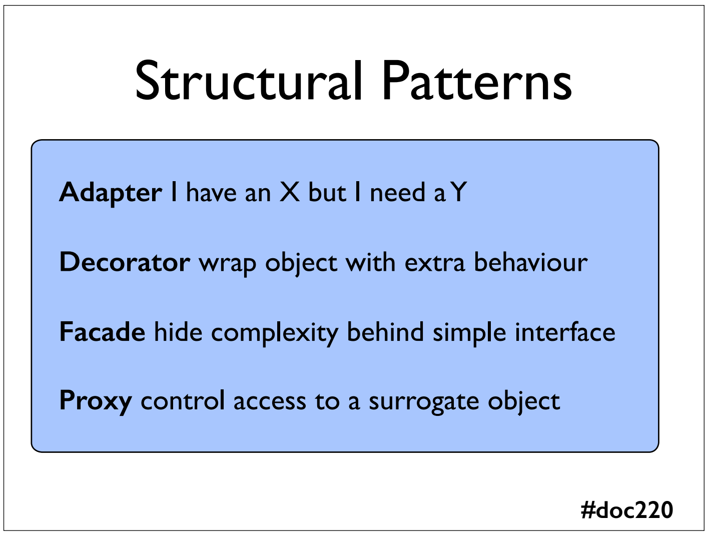
```java
public class BankAccountStore {

    private static BankAccountStore instance = new BankAccountStore();

    private Collection<BankAccount> accounts;

    private BankAccountStore() {
      // initialise accounts
      // set up data etc
    }

    public static BankAccountStore getInstance() {
        return instance;
    }

    public BankAccount lookupAccountById(int id) {
        ...
    }

    ...
}
```

private constructor enforces that no-one else can call `new BankAccountStore()`

clients call `getInstance()` method to gain access to the global instance

**#doc220**

If you do need to use a singleton, you can implement it in Java by having a class that guards the single instance, creates it once on initialisation, and allows other objects to get a reference to it through a static method typically named getInstance(). This static method can be called anywhere in your codebase, which is what makes this similar to a global variable. We make the constructor private to prevent clients using the new operator to create extra instances.



## Structural Patterns

**Adapter** I have an X but I need a Y

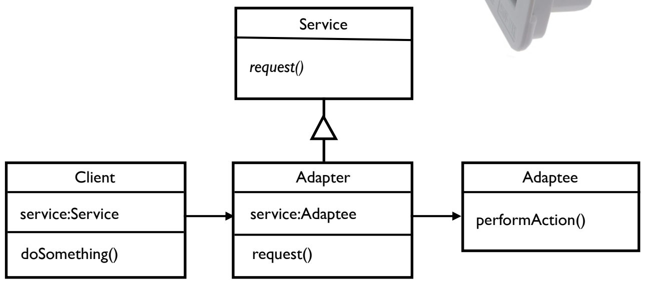
**Decorator** wrap object with extra behaviour

**Facade** hide complexity behind simple interface

**Proxy** control access to a surrogate object

**#doc220**

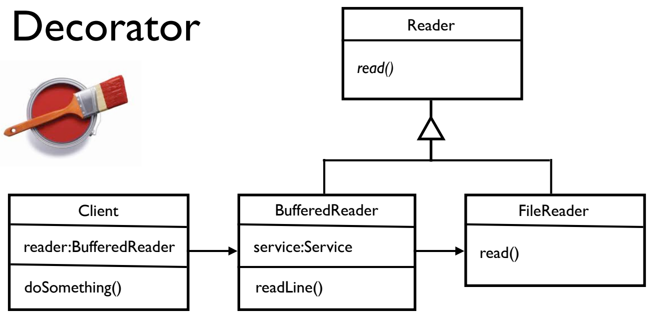
## Adapter



#doc220

Our client requires a particular service. The adapter implements the service interface but has no behaviour of its own, it translates calls through to the adaptee.

## Decorator



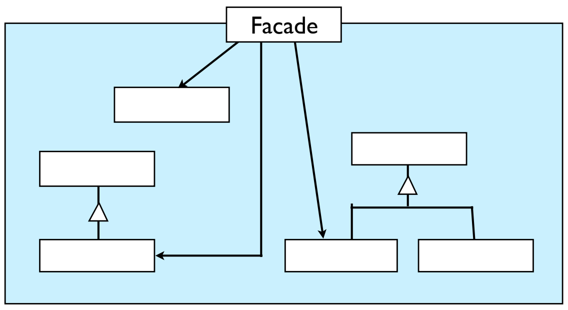
Java I/O uses decorators:

```
BufferedReader in
   = new BufferedReader(new FileReader("foo.in"));
```

#doc220

The I/O classes in the Java standard library make extensive use of the decorator pattern. For example, we can read a file byte by byte using a FileReader, but it's often useful to buffer these bytes rather than reading from the file directly all the time. We can wrap a BufferedReader around our FileReader to add this behaviour. The BufferedReader just deals with buffering, the rest of the reading operations are handled by the underlying FileReader.

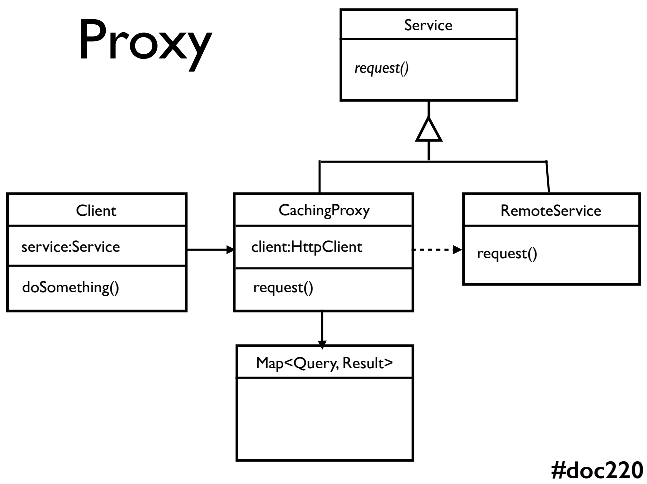
## Facade



#doc220

We can achieve this using a Facade pattern. Again this involves wrapping a number of underlying objects that actually provide the service, and delegating to them as necessary. The facade doesn't normally add behaviour, it just co-ordinates between underlying objects, or hides some of their complexity by not revealing all of the possible methods.

## Proxy



#doc220

If we are using a remote service, and it has high latency, we may be able to cache results locally to reduce the latency of subsequent calls to the service. We use a proxy that maintains a local map of query parameters to results. For each request we look it up in the map, if it is there, we return the result without going to the remote service. If the map does not contain an entry for the query, we go to the remote service, perform the query and then put the result into the cache before we return it. We may need to expire data from the cache when it gets stale, or when the cache gets too large.

## Architectural Styles

**MVC** separation of concerns in interactive apps

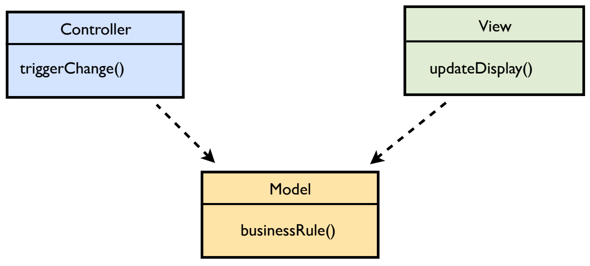
**PAC** for GUIs with hierarchical structure

**Publish-Subscribe** message passing: queues/topics

**Map Reduction** for large scale data processing

#doc220

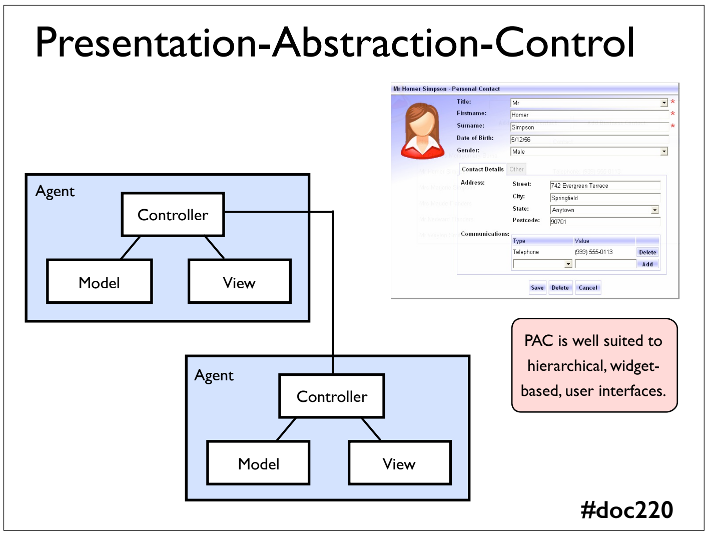
## Model-View-Controller



MVC separates interactive apps into 3 parts: data; display; user input

#doc220

One of the most common architectures for a GUI application is the Model-View-Controller (MVC). MVC splits the data model from the view which displays the data. You can build several different views on top of the same data. The controller makes updates to the model by calling methods on its objects in response to external events (e.g. clicks on the UI). It may be that the controller also triggers the view to redraw itself after an event.



## Presentation-Abstraction-Control

Mr Homer Simpson - Personal Contact

Title: Mr *
Firstname: Homer *
Surname: Simpson *
Date of Birth: 5/12/56
Gender: Male

Contact Details | Other

Address: Street: 742 Evergreen Terrace
City: Springfield
State: Anytown
Postcode: 90701

Communications:

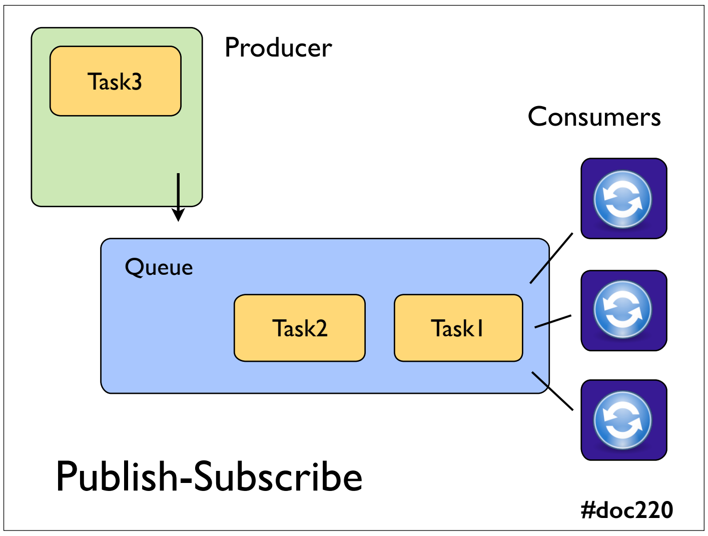
| Type | Value | |
|---|---|---|
| Telephone | (939) 555-0113 | Delete |
| | | Add |

PAC is well suited to hierarchical, widget-based, user interfaces.

#doc220

An alternative to MVC, although one that is used rather less often, is Presentation-Abstraction-Control (PAC). PAC architectures are well suited to GUIs where there is a hierarchy to the user interface - so panels contain sub-panels, each of which has a group of controls or displays for a certain item of data.

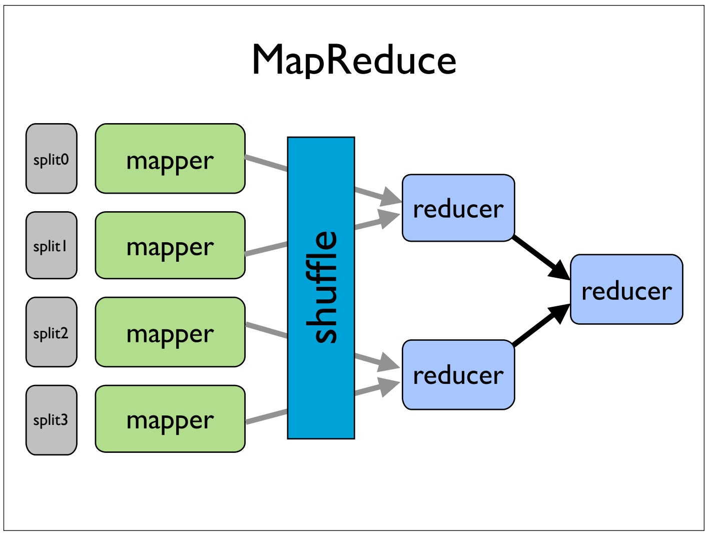
The way that PAC works is to define a set of "mini-MVC" agents. We form these agents into a tree, and communication between different widgets on the page goes via connections between the controllers. We should only communicate up and down the tree, rather than jumping across to another branch.



## Publish-Subscribe

#doc220

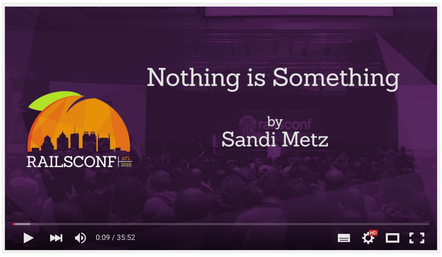
Here we see the basic concept of producers and consumers. Producers create work items, tasks, and put them onto the queue for someone else to process. The consumers monitor the queue and if there is a task there they will pick it up. The notion of a queue is one of the key concepts in publish-subscribe architectures, together with producers and consumers. In this diagram we see multiple consumers consuming from the same queue.



## MapReduce

When MapReduce runs it splits the input data into a number of "splits" and processes each split with a separate mapper, most likely running on a separate computer. The outputs are then put through the reducer phase. There are normally fewer reducers than mappers, but it depends on the complexity of the various parts of the job. The magic of MapReduce is in the shuffle. This is where all of the key-value pairs output from the map for a given key value are gathered up and supplied to the same reducer, so that it can do its work.

## Two Good Videos



https://www.youtube.com/watch?v=8bZh5LMaSmE

https://www.youtube.com/watch?v=29MAL8pJImQ