Imperial College
London

NOTES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

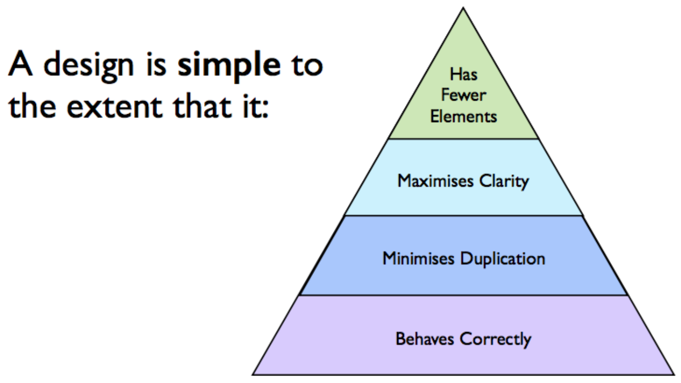
Software Engineering

Author:
Your Name (CID: your college-id number)

Date: March 31, 2018

1 Introduction

Four Elements of Simple Design



Test-Driven Development (TDD)

We start by writing a test. This helps us to specify what we want the code we are about to write to do. Once we have written a test, we write the simplest possible piece of code we can to make the test pass. After this we refactor our design to clean up, remove any duplication, improve clarity etc etc. Then we being the cycle again. When we do TDD, we should only refactor when we are in a green state.

UML sequence diagram

The exchange of messages between objects can be shown in a UML sequence diagram, sometimes known as a Message Sequence Chart.

2 Design Principles

Law of Demeter

only talk to your immediate friends.

Having an implementation that depends on knowing the implementation of pieces of the system that are further away results in a tight coupling which can cause problems with testing and with maintenance.

This is often revealed by long chains of method calls, asking other objects for a reference to their collaborators, so that the first object can traverse the object graph.

Tell Don't Ask

objects send each other messages, requesting certain actions to be carried out, but they do not expect back a return value from those calls.

The only calls that should return values are queries, that simply return data and do not cause any other change of state or interaction between objects.

Hollywood Principle

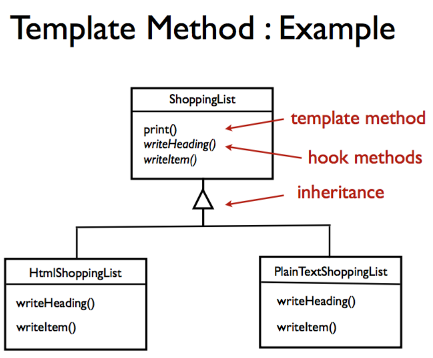
Don't call us, we'll call you.

Open-Closed Principle

The Open-Closed principle means that we should aim to produce re-usable, extensible modules that are flexible, but can be specialised without changing their original source code.

3 Design Patterns

3.1 Template Method Pattern



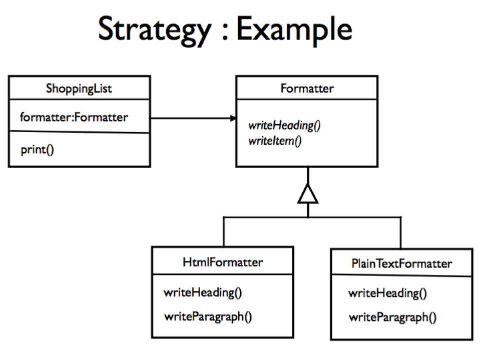
We define the main part of the algorithm in the superclass, but call out to some abstract hook methods, which will be overridden in subclasses to specify particular behaviour for the different variants.

Hollywood Principle: The concrete classes don't call up into the superclass, they just define methods that will be called when the superclass needs them.

Open-Closed Principle: The core of the algorithm is in the superclass, and does not change. The hook methods allow us to override and specialise a particular part of the algorithm in each of the subclasses, but without having to alter or recompile the superclass.

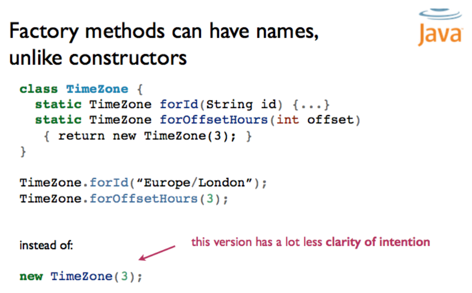
Disadvantage (immobility caused by coupling): we have a tight coupling between the subclasses and the superclass. We cannot use the subclasses independently, or use them in a different system, because they depend on the named superclass.

3.2 Strategy Pattern



The Strategy pattern gives us more flexibility here as we compose the objects together, which is a looser coupling than using inheritance.

3.3 Factory Pattern



Factory methods can have names, unlike constructors

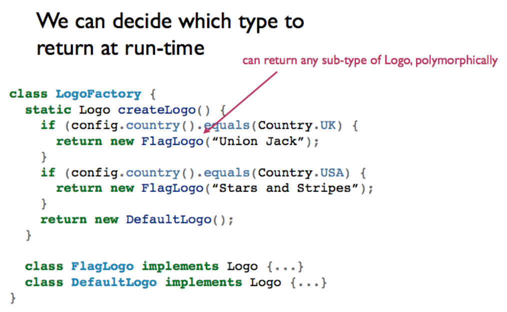
```
class TimeZone {
  static TimeZone forId(String id) {...}
  static TimeZone forOffsetHours(int offset)
   { return new TimeZone(3); }
}

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

instead of:

this version has a lot less clarity of intention

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

Sometimes it is not clear from a call to a constructor function what we will get back, or how the parameters that we pass will be interpreted. It would be helpful of we could increase the clarity of intention by giving the constructor function a name



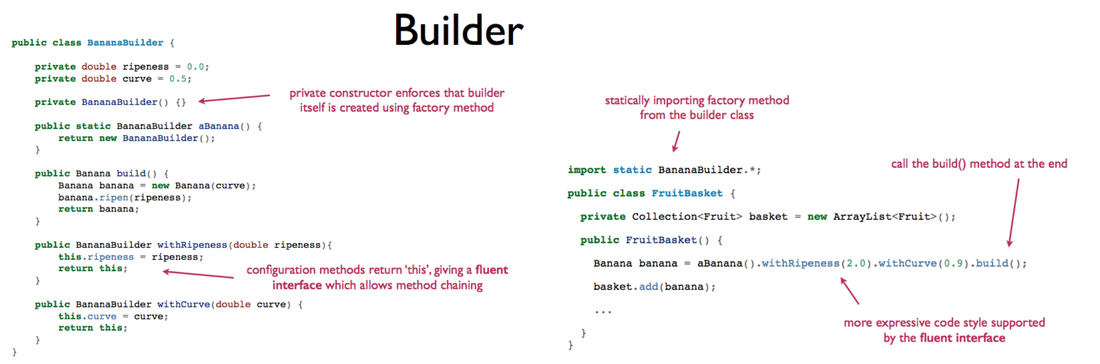
We can decide which type to return at run-time

can return any sub-type of Logo, polymorphically

```
class LogoFactory {
  static Logo createLogo() {
    if (config.country().equals(Country.UK) {
      return new FlagLogo("Union Jack");
    }
    if (config.country().equals(Country.USA) {
      return new FlagLogo("Stars and Stripes");
    }
    return new DefaultLogo();
  }

  class FlagLogo implements Logo {...}
  class DefaultLogo implements Logo {...}
}
```

Another use for a factory is to defer the decision about the exact type of the object that is to be created until runtime.

3.4 Builder Pattern


Builder
private constructor enforces that builder itself is created using factory method
configuration methods return 'this', giving a fluent interface which allows method chaining
statically importing factory method from the builder class
call the build() method at the end
more expressive code style supported by the fluent interface

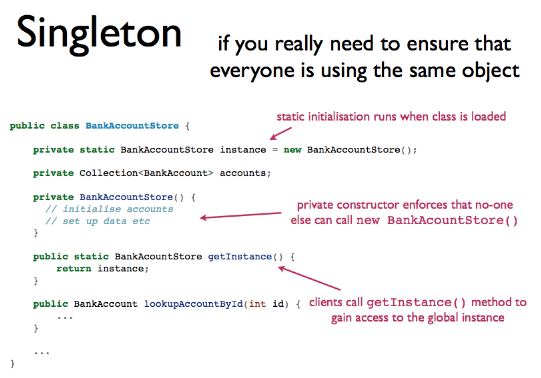
```
public class BananaBuilder {

    private double ripeness = 0.0;
    private double curve = 0.5;

    private BananaBuilder() {}

    public static BananaBuilder aBanana() {
        return new BananaBuilder();
    }

    public Banana build() {
        Banana banana = new Banana(curve);
        banana.ripen(ripeness);
        return banana;
    }

    public BananaBuilder withRipeness(double ripeness){
        this.ripeness = ripeness;
        return this;
    }

    public BananaBuilder withCurve(double curve) {
        this.curve = curve;
        return this;
    }
}
```

```
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);

    ...

  }
}
```

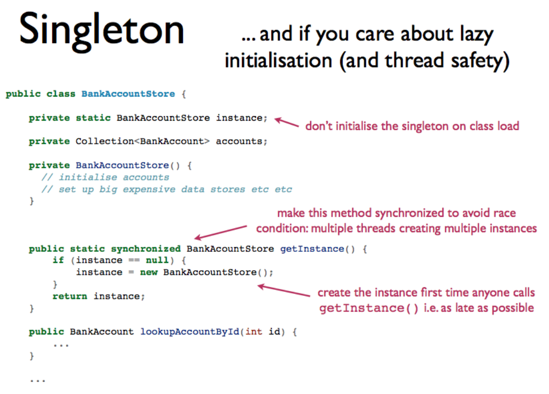
A builder is an object separate from the one being created that takes responsibility for gathering configuration parameters, and then, once all of that information has been collected, producing a fully formed object in a valid state.

We call the with methods to set the values that we want for the properties, although they may have default values, and then call the build() method at the end to create the object.

3.5 Singleton Pattern


Singleton
if you really need to ensure that everyone is using the same object
static initialisation runs when class is loaded
private constructor enforces that no-one else can call new BankAcountStore()
clients call getInstance() method to gain access to the global instance

```
public class BankAccountStore {

    private static BankAccountStore instance = new BankAccountStore();

    private Collection<BankAccount> accounts;

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

    public static BankAcountStore getInstance() {
        return instance;
    }

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

    ...
}
```

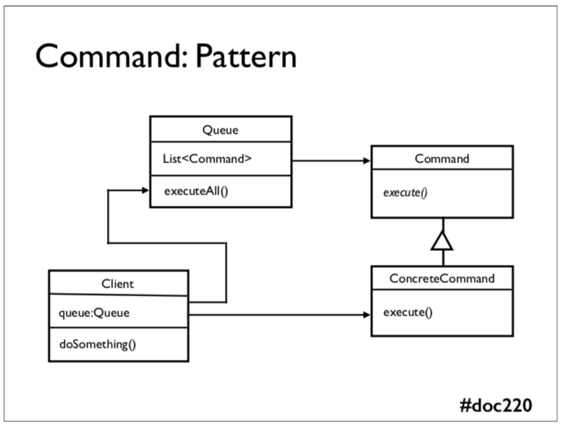

Singleton
... and if you care about lazy initialisation (and thread safety)
don't initialise the singleton on class load
make this method synchronized to avoid race condition: multiple threads creating multiple instances
create the instance first time anyone calls getInstance() i.e. as late as possible

```
public class BankAccountStore {

    private static BankAccountStore instance;

    private Collection<BankAccount> accounts;

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

    public static synchronized BankAcountStore getInstance() {
        if (instance == null) {
            instance = new BankAccountStore();
        }
        return instance;
    }

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

    ...
```

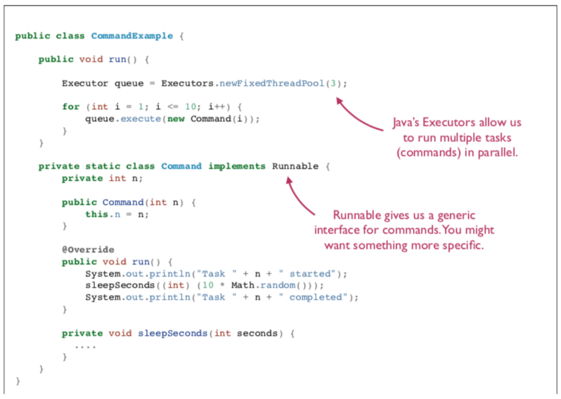
If you do need to use a singleton, you can implement it in Java by having a class that guards the single instance, creates it once on initialisation, and allows other objects to get a reference to it through a static method typically named getInstance().

Sometimes we want to initialise the singleton lazily, perhaps because doing so it very expensive and we don?t want to do it until we need it. In that case we can use this other design, but be careful here about what happens if two threads enter the getInstance() method at the same time. To prevent the race condition where multiple instances are created, we have to make this method **synchronized**.

3.6 Command Pattern



A client creates a command, and adds it to a queue of commands, which may be executed later as a batch.

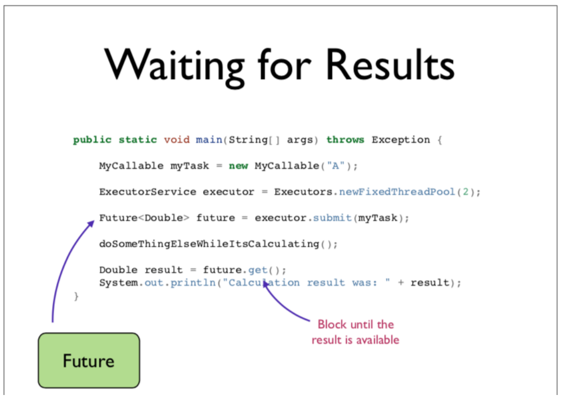


```
public class CommandExample {

    public void run() {

        Executor queue = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 10; i++) {
            queue.execute(new Command(i));
        }
    }

    private static class Command implements Runnable {
        private int n;

        public Command(int n) {
            this.n = n;
        }

        @Override
        public void run() {
            System.out.println("Task " + n + " started");
            sleepSeconds((int) (10 * Math.random()));
            System.out.println("Task " + n + " completed");
        }

        private void sleepSeconds(int seconds) {
          ....
        }
    }
}
```

Java's Executors allow us to run multiple tasks (commands) in parallel.

Runnable gives us a generic interface for commands. You might want something more specific.

We can exploit the possibility of parallelism by using more threads by using an Executor. Executor can manage a pool of threads, and use these to execute our tasks concurrently.

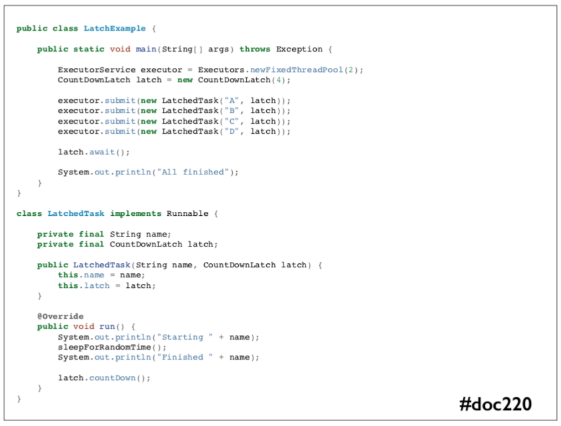


Waiting for Results

```
public static void main(String[] args) throws Exception {

    MyCallable myTask = new MyCallable("A");

    ExecutorService executor = Executors.newFixedThreadPool(2);

    Future<Double> future = executor.submit(myTask);

    doSomeThingElseWhileItsCalculating();

    Double result = future.get();
    System.out.println("Calculation result was: " + result);
}
```

Block until the result is available

Sometimes we would like to execute functions in parallel that compute a value. If we use a Callable, then when we submit the task to the Executor, it gives us back a Future, which is something that we can use to track the execution of our task.

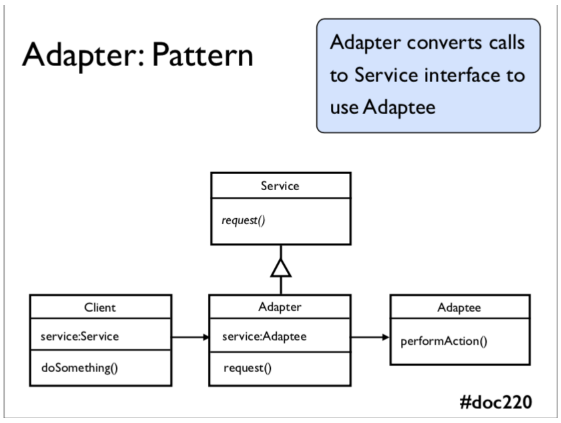


```java
public class LatchExample {

    public static void main(String[] args) throws Exception {

        ExecutorService executor = Executors.newFixedThreadPool(2);
        CountDownLatch latch = new CountDownLatch(4);

        executor.submit(new LatchedTask("A", latch));
        executor.submit(new LatchedTask("B", latch));
        executor.submit(new LatchedTask("C", latch));
        executor.submit(new LatchedTask("D", latch));

        latch.await();

        System.out.println("All finished");
    }
}

class LatchedTask implements Runnable {

    private final String name;
    private final CountDownLatch latch;

    public LatchedTask(String name, CountDownLatch latch) {
        this.name = name;
        this.latch = latch;
    }

    @Override
    public void run() {
        System.out.println("Starting " + name);
        sleepForRandomTime();
        System.out.println("Finished " + name);

        latch.countDown();
    }
}
```

#doc220

When we don't care so much about an individual return value, but we want to wait for all tasks to complete. We can use a Latch instead.

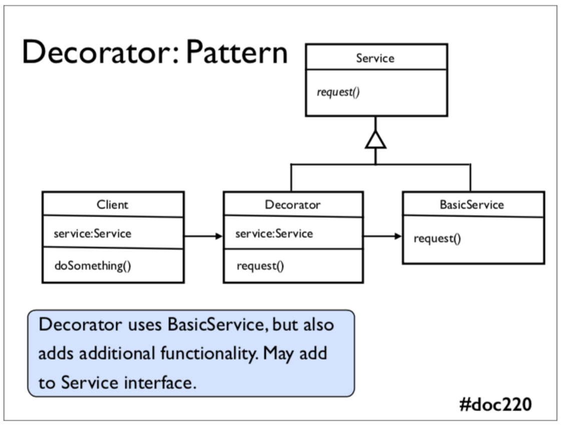
3.7 Adapter Pattern



I have X, but I need a Y.

When we have a class, or a service, that can do what we need, but we need it to present a different interface in order to work with the rest of our system we can use an Adapter pattern to convert from one interface to another.

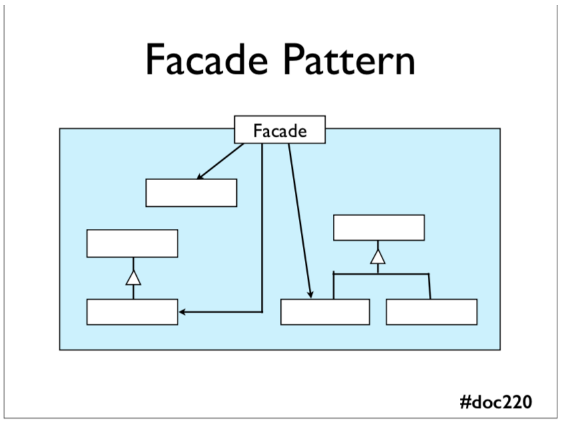
3.8 Decorator Pattern



I have X, but I want a better X

We have something that we could use - it has the right interface - but we want to add in some extra behaviour.

3.9 Facade Pattern

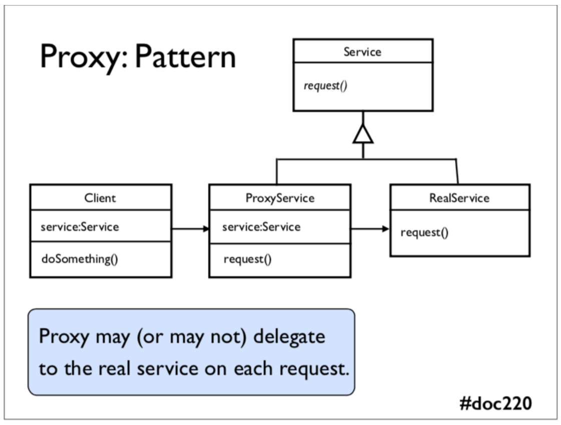


I have X, but I want a simpler X

Sometimes we have a very complex object, or sub- system, that has many possible behaviours - but we only need to use part of it.

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.

3.10 Proxy Pattern



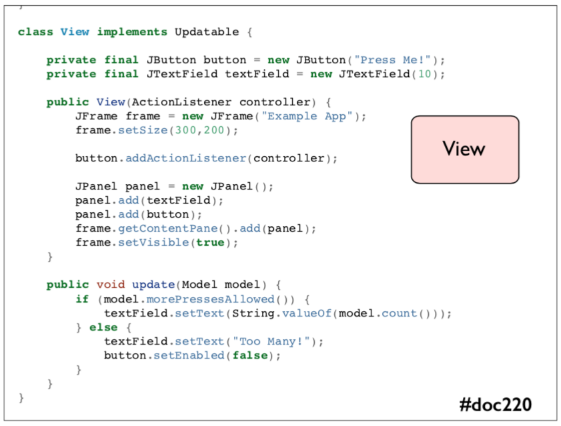
I have an X, but it is too slow.

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.

4 GUI

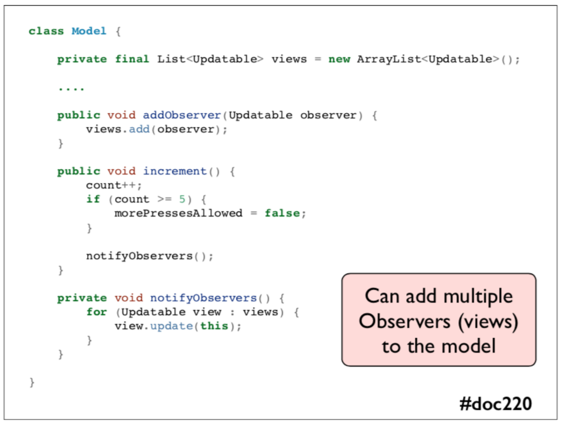
4.1 Model-View-Controller (MVC)

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.



```
class View implements Updatable {

    private final JButton button = new JButton("Press Me!");
    private final JTextField textField = new JTextField(10);

    public View(ActionListener controller) {
        JFrame frame = new JFrame("Example App");
        frame.setSize(300,200);

        button.addActionListener(controller);

        JPanel panel = new JPanel();
        panel.add(textField);
        panel.add(button);
        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }

    public void update(Model model) {
        if (model.morePressesAllowed()) {
            textField.setText(String.valueOf(model.count()));
        } else {
            textField.setText("Too Many!");
            button.setEnabled(false);
        }
    }
}
```

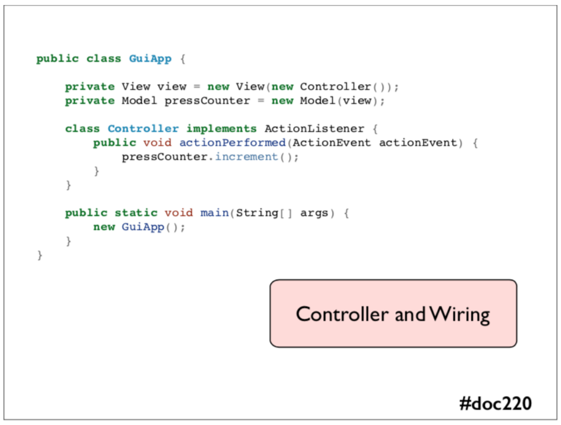
The View contains the button, and we set up the link to the Controller, so that the controller code fires when the button is pressed.



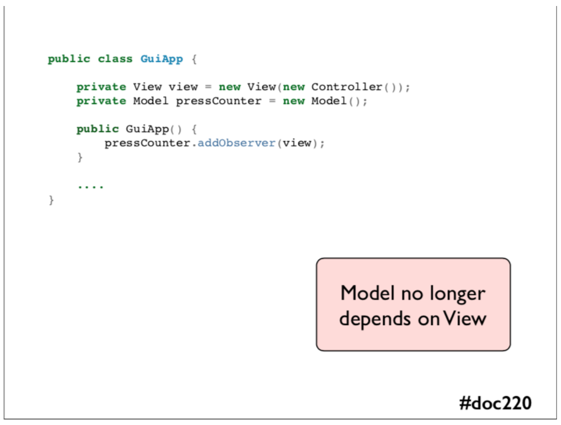
```java
class Model {

    private final List<Updatable> views = new ArrayList<Updatable>();

    ....

    public void addObserver(Updatable observer) {
        views.add(observer);
    }

    public void increment() {
        count++;
        if (count >= 5) {
            morePressesAllowed = false;
        }

        notifyObservers();
    }

    private void notifyObservers() {
        for (Updatable view : views) {
            view.update(this);
        }
    }

}
```

The model class just stores the data, and contains the logical operations that act upon this data.

One of the strengths of MVC is that we can create multiple different Views to allow the Model to be displayed in different ways. The Model is coded to allow us to add a number of different Updatables as observers, and have each one be notified when there is a change to the Model's state.



```java
public class GuiApp {

    private View view = new View(new Controller());
    private Model pressCounter = new Model(view);

    class Controller implements ActionListener {
        public void actionPerformed(ActionEvent actionEvent) {
            pressCounter.increment();
        }
    }

    public static void main(String[] args) {
        new GuiApp();
    }
}
```



```java
public class GuiApp {

    private View view = new View(new Controller());
    private Model pressCounter = new Model();

    public GuiApp() {
        pressCounter.addObserver(view);
    }

    ....
}
```

The GuiApp class wires everything together, creating the Model, View and Controller objects and wiring them together to set up the structure that we want.

4.2 Presentation-Abstraction-Control

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.

5 Richardson Maturity Model

The Richardson Maturity Model is used to describe and categorise webservices based on the degree to which they take advantage of each of URIs for identifying resources, HTTP and the various methods that it supports, and hypermedia in terms of linked data.

At the lowest level of the Richardson Maturity Model we have Level 0 services. These services use HTTP (and in that sense, the web) as a means of transporting data, but do not take advantage of URIs to identity resources, different HTTP methods to describe actions, or hypermedia.

Level 1 services make use of more URIs to represent different types of resources in the system, but typically do not take advantage of all of the available HTTP methods.

Level 2 services use URIs to represent different types of resources, and also respond to different HTTP methods (typically GET, POST, PUT and DELETE) in order to update the state of these resources. They also send appropriate HTTP status codes with their responses, which allow the client to track the effects of the calls they have made.

Level 3 is the highest level in the Richardson model, and characterises fully RESTful services. It builds on Level 2. The key point about Level 3 services is that the representations that they use contain hyperlinks to other resources that the client can follow.