Queen Mary
University of London

ECS505U
SOFTWARE ENGINEERING

MUSTAFA BOZKURT & LORENZO JAMONE

LECTURER IN SOFTWARE ENGINEERING

Queen Mary
University of London

Week 5

Introduction to Design Patterns

INTRODUCTION

- What are design patterns?
- Why are they useful?
- Some concrete examples
- Summary of “Gang Of Four” patterns

Queen Mary University of London

WHAT ARE DESIGN PATTERNS?

"*A pattern is the outline of a reusable solution to a general problem encountered in a particular context*."

"*Design patterns are recurring solutions to design problems you see over and over.*" (Alpert et al., 1998)

First described by Christopher Alexander in 1960s

WHAT ARE DESIGN PATTERNS?

Design Patterns: Elements of Reusable Object-Oriented Software

by “the GangOfFour”

ErichGamma, RichardHelm, RalphJohnson, and JohnVlissides

ISBN 0201633612

“The GangOfFour book didn't invent all those design patterns you know. They went out and tried to document the most fundamental object oriented design patterns they could find over several years.”

EXAMPLE: MASTERING CHESS

- First learn rules
- Then learn principles
- Finally study the games of masters

www.westerlylibrary.org

MASTERING SOFTWARE DESIGN

- First learn rules
- Then learn principles
- Finally study the designs of masters

PATTERNS AT DIFFERENT LEVELS

Package level ('Architectural')

- Model View Control (MVC)

Class/Object level

- We will cover during this lecture

Method level

- If-else or loop

DESIGN PATTERN TYPES

Creational Patterns

- Deal with creating, initialising and configuring classes and objects

Structural Patterns

- Deal with decoupling the interface and implementations of classes and objects to provide more powerful structures

Behavioural Patterns

- Deal with communication between groups of objects and classes

CREATIONAL PATTERNS

Name	Description
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Factory method	Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses (dependency injection).
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
Multiton	Ensure a class has only named instances, and provide global point of access to them.
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Resource acquisition is initialization	Ensure that resources are properly released by tying them to the lifespan of suitable objects.
Singleton	Ensure a class has only one instance, and provide a global point of access to it.

STRUCTURAL PATTERNS

Name	Description
Adapter or Wrapper or Translator.	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
Bridge (Player-Role)	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of similar objects efficiently.
Front Controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Twin	Twin allows to model multiple inheritance in programming languages that do not support this feature.

BEHAVIOURAL PATTERNS

Name	Description
Blackboard	Generalized observer, which allows multiple readers and writers. Communicates information system-wide.
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Null object	Avoid null references by providing a default object.
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Servant	Define common functionality for a group of classes
Specification	Recombinable business logic in a Boolean fashion
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

ELEMENTS OF A DESIGN PATTERN

Name: A title that conveys the essence of the pattern succinctly

Context: The general situation in which the pattern applies

Problem: A statement of the problem that describes the pattern's intent and any issues to consider

Solution: The elements that make up the design: relationships, responsibilities and collaborations

Consequences: The results and trade-offs of applying the pattern

Antipattern: Examples of bad practice (things you shouldn't do)

ABSTRACTION-OCCURRENCE: PROBLEM

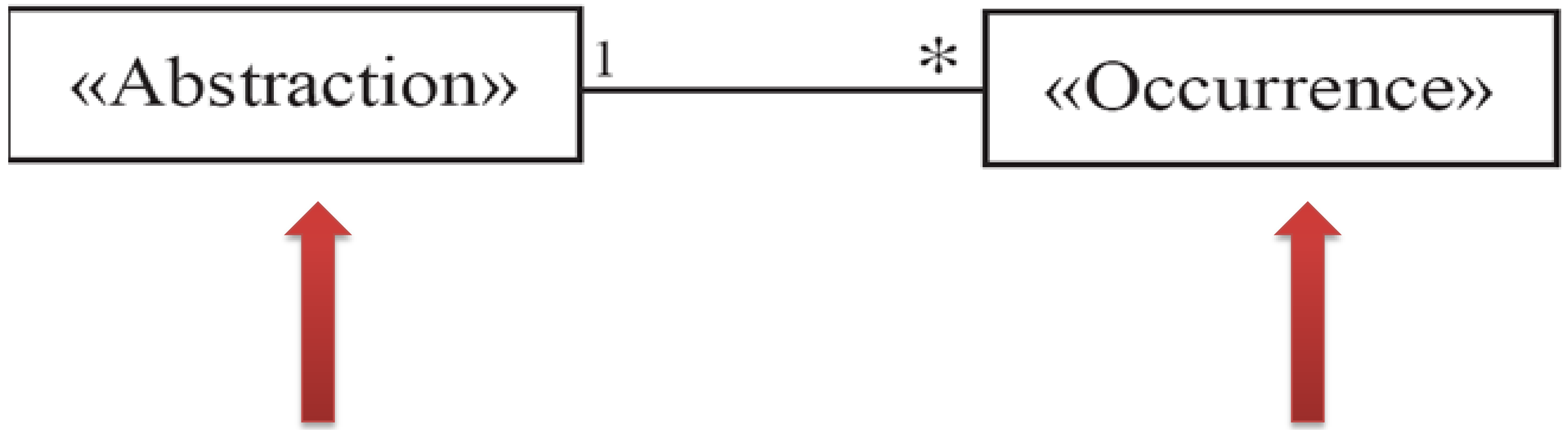
Context:

There is a set of related objects (occurrences) whose members share common information but also differ in important ways (think of multiple copies of a book)

Problem:

Need an efficient way to implement the set of objects without duplicating the common information

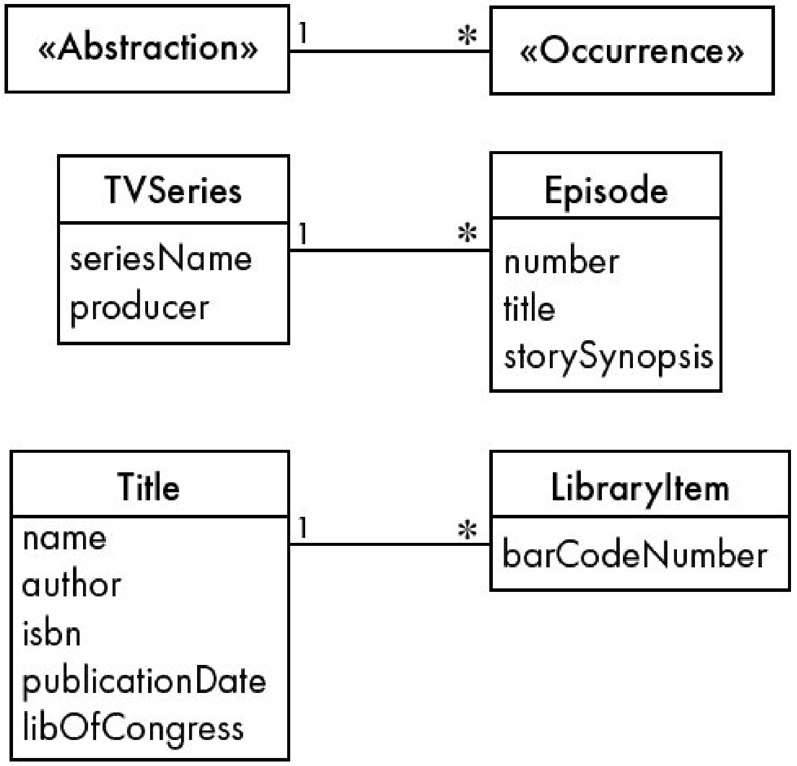
ABSTRACTION-OCCURRENCE: SOLUTION



Contains the information that is really common to all the related items

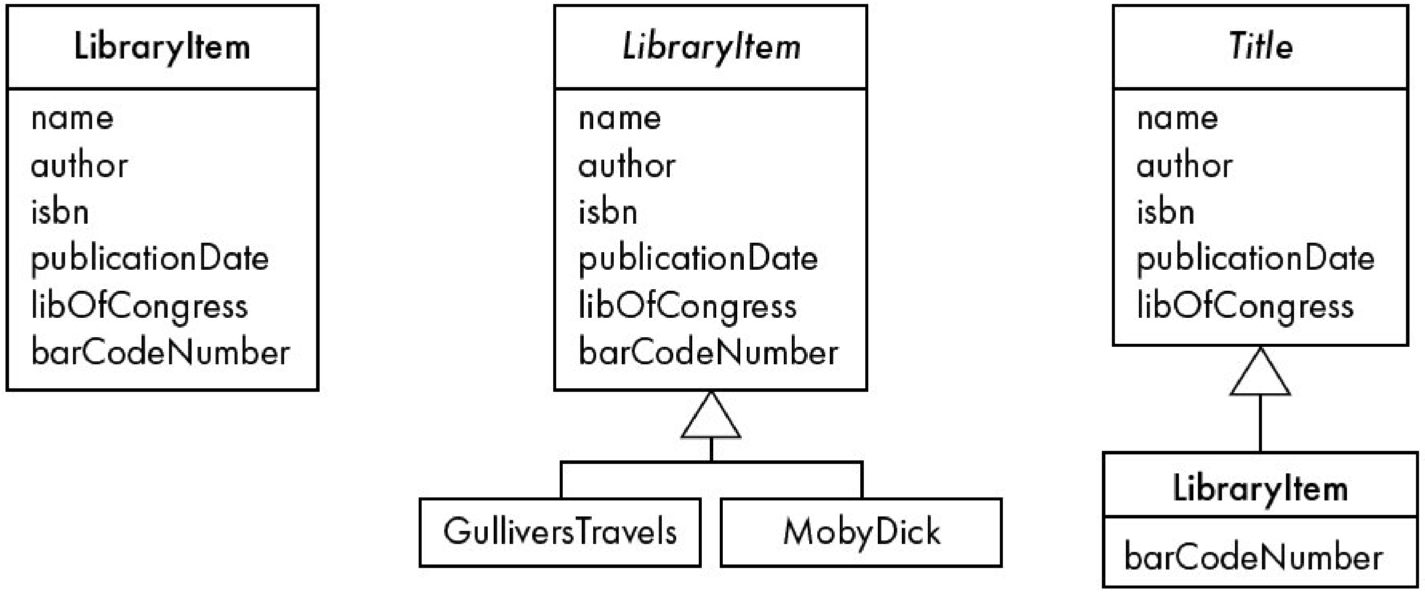
Contains the information that is unique to instances of the items

ABSTRACTION-OCCURRENCE: SOLUTION



```
public class TVSeries {
    String title;
    String producer;
    Collection<Episode> episodes; //Many episodes of a TV series
}

class Episode {
    TVSeries series; //One series for all episodes
    String title;
    int season;
    int number;
    Collection<Actor> actors;
}


class Actor{

}
```

ABSTRACTION-OCCURRENCE: ANTIPATTERNS



THE PLAYER–ROLE PATTERN: PROBLEM (AKA BRIDGE)

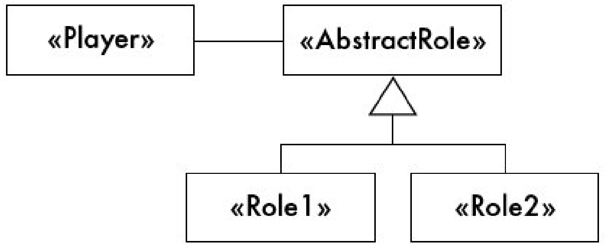
Context:

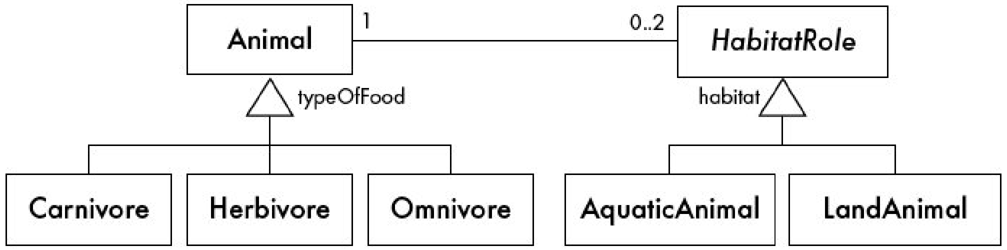
If your objects change roles or have different roles based on the context.

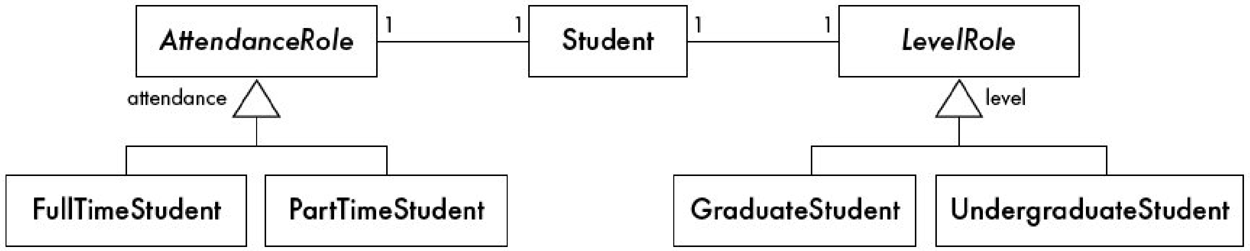
Problem:

Need an efficient way to model players and roles in a way to enable role changing.

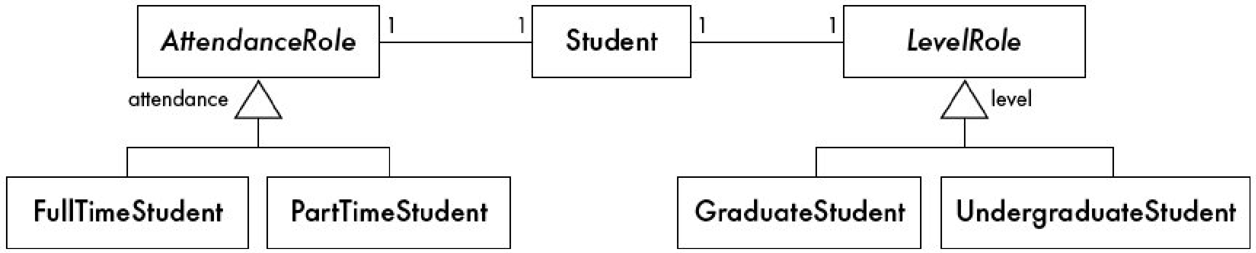
THE PLAYER–ROLE PATTERN: SOLUTION







THE PLAYER–ROLE PATTERN: SOLUTION



```
interface AttendanceRole {

}

class FullTimeStudent implements AttendanceRole{

}

class PartTimeStudent implements AttendanceRole{

}
```

```
public class Student {
    String id;
    LevelRole level;
    AttendanceRole attendance;
}

abstract class LevelRole {

}

class GraduteStudent extends LevelRole{
    Collection<Publication> publications;
}

class UnderGraduteStudent extends LevelRole{
    int yearOfStudy;
    void changeProgramme(int progCode){}
}
```

THE PLAYER–ROLE PATTERN: ANTIPATTERNS

- Merge all the features into a single class
- Subclasses of the «Player» class

THE SINGLETON PATTERN: PROBLEM

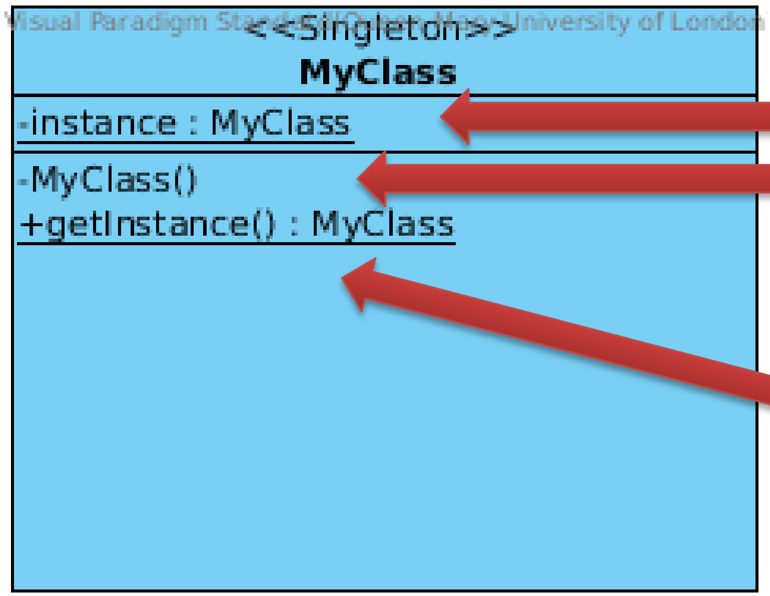
Context:

It is very common to find classes for which only one instance should exist (singleton)

Problem:

- Ensure not possible to create more than one instance
- The use of a public constructor cannot guarantee that no more than one instance will be created.
- The singleton instance must also be accessible to all classes that require it

THE SINGLETON PATTERN: SOLUTION



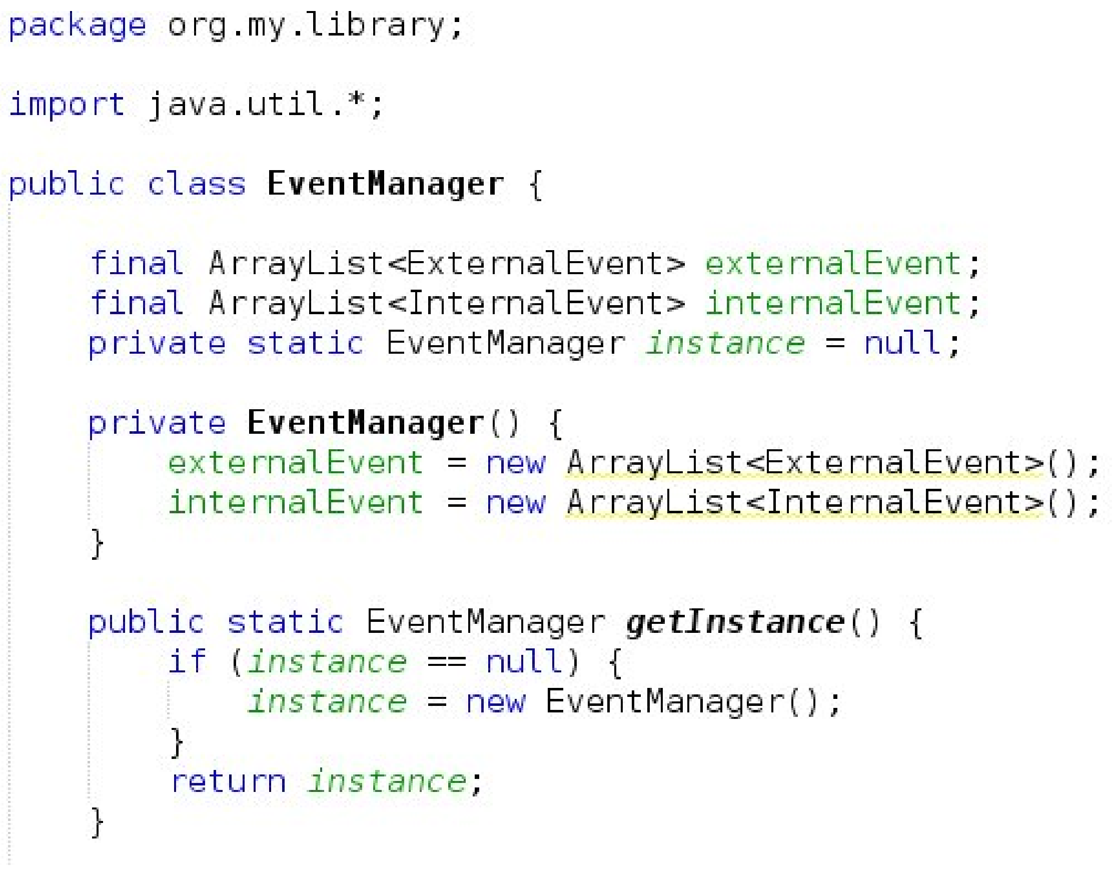
Private (static) instance

Private constructor prevents creation of instances from outside the class

Public (static) method to get the instance

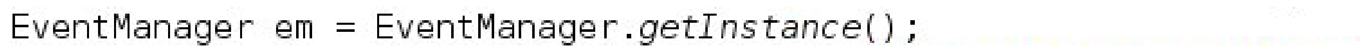
```
public class MyClass {

    private static MyClass instance = null;

    private MyClass(){
    }

    public static MyClass getInstance(){
        if(instance == null){
            instance = new MyClass();
        }
        return instance;
    }
}
```

THE SINGLETON PATTERN: JAVA EXAMPLE



```java
package org.my.library;

import java.util.*;

public class EventManager {

    final ArrayList<ExternalEvent> externalEvent;
    final ArrayList<InternalEvent> internalEvent;
    private static EventManager instance = null;

    private EventManager() {
        externalEvent = new ArrayList<ExternalEvent>();
        internalEvent = new ArrayList<InternalEvent>();
    }

    public static EventManager getInstance() {
        if (instance == null) {
            instance = new EventManager();
        }
        return instance;
    }
```

This is how you create an instance of a singleton



```java
EventManager em = EventManager.getInstance();
```

THE OBSERVER PATTERN: PROBLEM

Context:

- A number of objects need to know about a single object and must be told when it changes
- Frequent in GUI programs e.g. share prices shown as a list and a graph

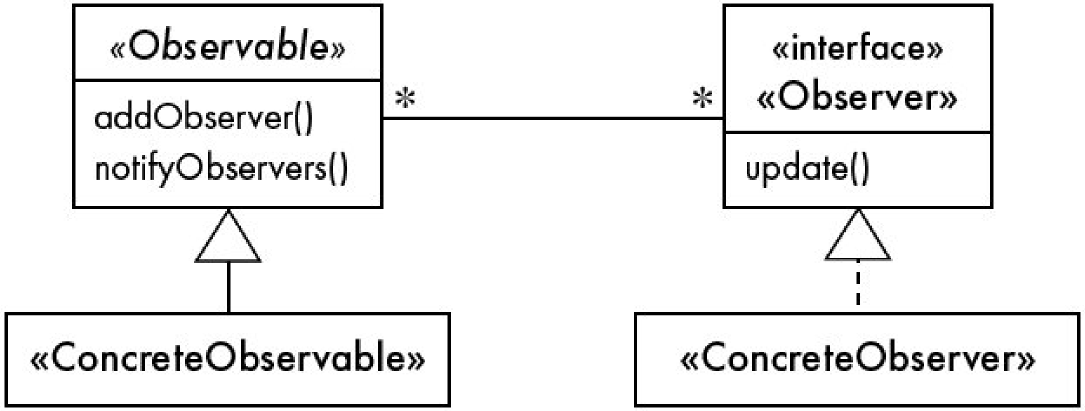
Problem:

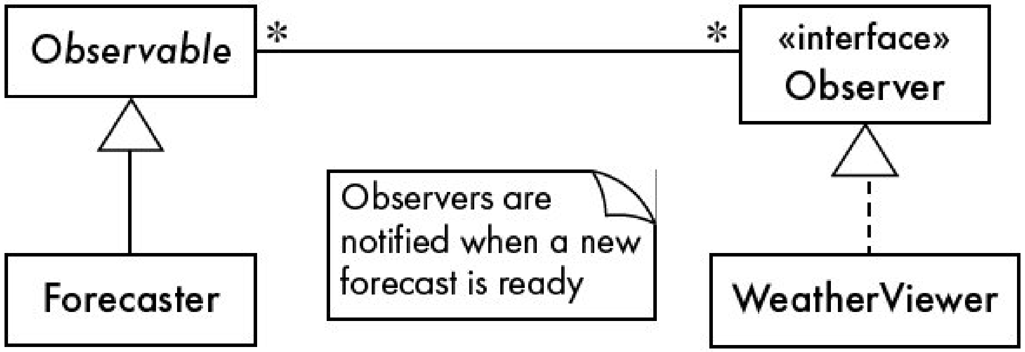
- Querying the object manually all the time to look for changes is inefficient (especially if the changing object has hard-coded notifications)
- A change to one object requires changes to others but you don't know how many objects need to be changed
- An object should be able to notify other objects without making assumptions about who they are

THE OBSERVER PATTERN: SOLUTION

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated
- Maintains consistency between related objects but without enforcing tight coupling
  - Observers can add themselves dynamically to the Subject
  - Subject doesn’t care about details of Observers
- Used throughout Swing e.g. ActionListener, TableModel

THE OBSERVER PATTERN: SOLUTION





THE OBSERVER PATTERN: JAVA EXAMPLE

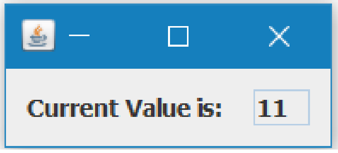
```java
public class Subject extends Observable {

    private int number = 0;

    public void setNumber(int number) {
        this.number = number;
        setChanged();
        notifyObservers();
    }

    public int getNumber() {
        return number;
    }
}
```

```java
import java.util.Observable;
import java.util.Observer;

public class ConsoleObserver implements Observer{

    @Override
    public void update(Observable o, Object arg) {
        System.out.println("Current Value is: " + ((Subject)o).getNumber());
    }

}
```

```
run:
Current Value is: 5
Current Value is: 11
```

```java
public static void main(String[] args){
    Subject s = new Subject();
    s.addObserver(new SwingObserver());
    s.addObserver(new ConsoleObserver());
    s.setNumber(5);
    s.setNumber(11);
}
```

```java
public class SwingObserver implements Observer{

    SwingObserverFrame frame = null;
    public SwingObserver(){
        frame = new SwingObserverFrame();
        frame.setTitle("Observer");
        frame.setVisible(true);
    }
    @Override
    public void update(Observable o, Object arg) {
        frame.setText(String.valueOf(((Subject)o).getNumber()));
    }
}
```


Current Value is: 11

Java has an Observer interface and an Observable class. The Java mechanism is a specific implementation of this pattern

THE OBSERVER PATTERN: CONSEQUENCES

- Abstract and minimal coupling between Subject and Observer
- Support for broadcast communication
- Observer independence can cause unexpected behaviour

THE ADAPTER PATTERN: PROBLEM

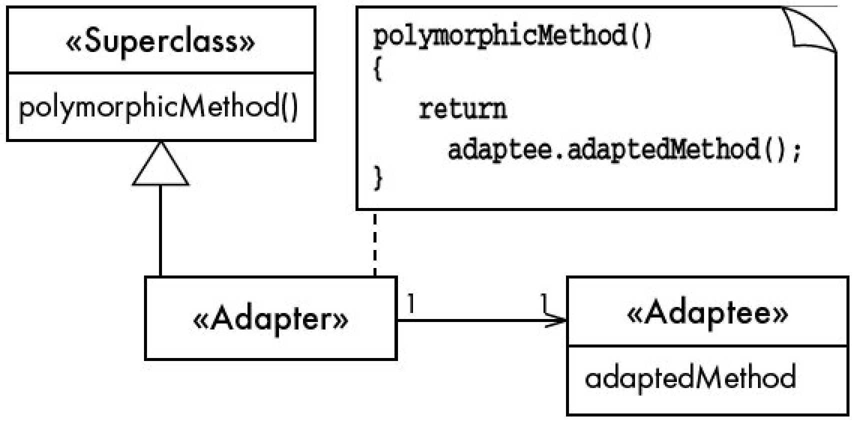
Context:

- You are building an inheritance hierarchy and you want to incorporate into it a n existing class.
- Methods of the reused class do not have the same name or argument types as the methods in the hierarchy.
- The reused class is also often already part of its own inheritance hierarchy.

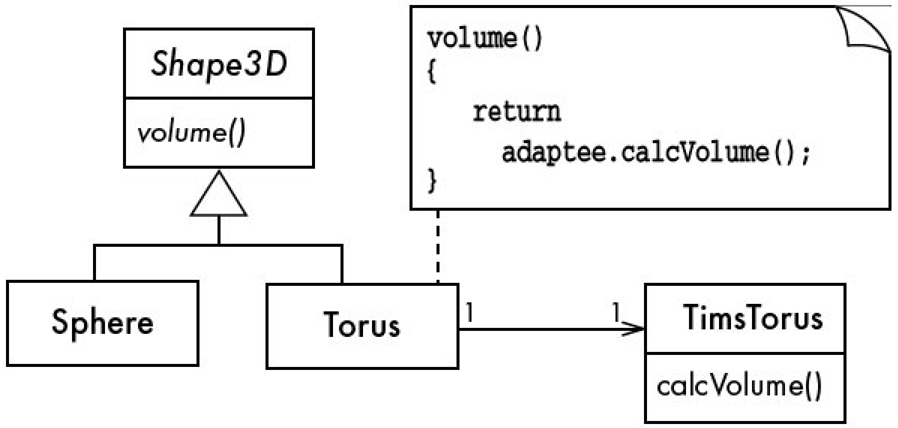
Problem:

- How do you obtain the power of polymorphism when reusing a class?
- You do not have access to multiple inheritance or you do not want to use it.

THE ADAPTER PATTERN: SOLUTION

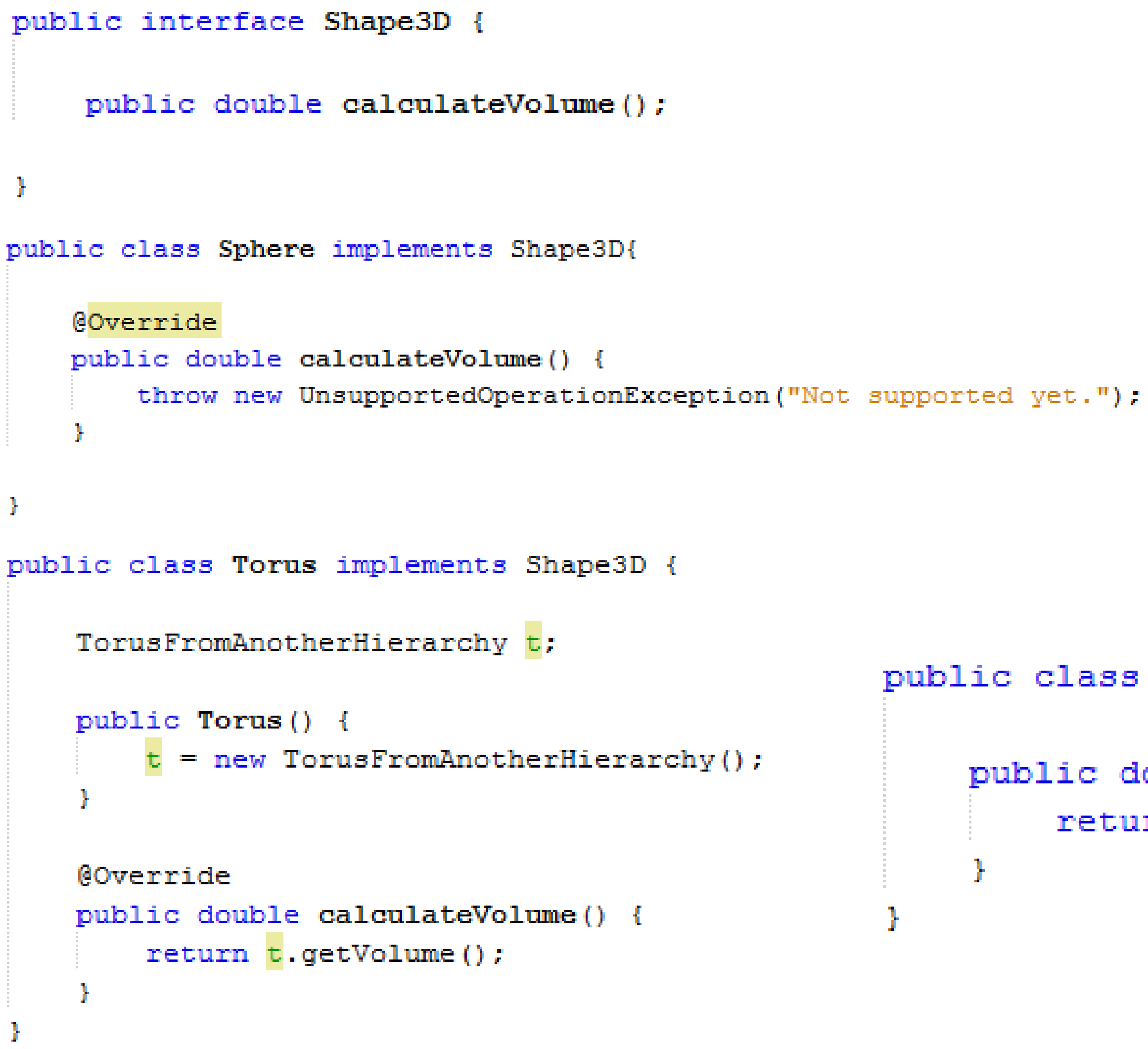


```
polymorphicMethod()
{
   return
     adaptee.adaptedMethod();
}
```

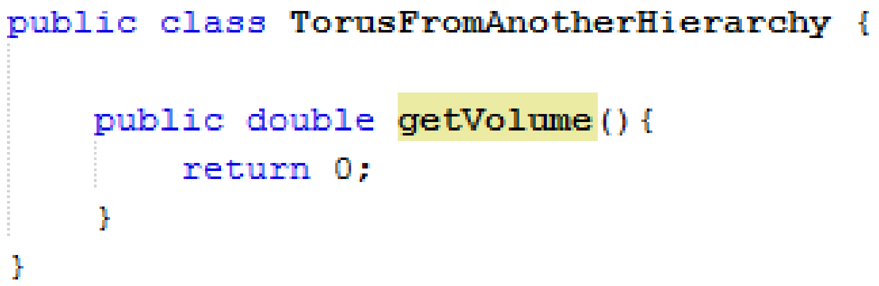


```
volume()
{
   return
     adaptee.calcVolume();
}
```

THE ADAPTER PATTERN: SOLUTION



```
public interface Shape3D {

    public double calculateVolume();

}
public class Sphere implements Shape3D{

    @Override
    public double calculateVolume() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}
public class Torus implements Shape3D {

    TorusFromAnotherHierarchy t;

    public Torus() {
        t = new TorusFromAnotherHierarchy();
    }

    @Override
    public double calculateVolume() {
        return t.getVolume();
    }
}
```



```
public class TorusFromAnotherHierarchy {

    public double getVolume(){
        return 0;
    }
}
```

THE FACTORY PATTERN: PROBLEM

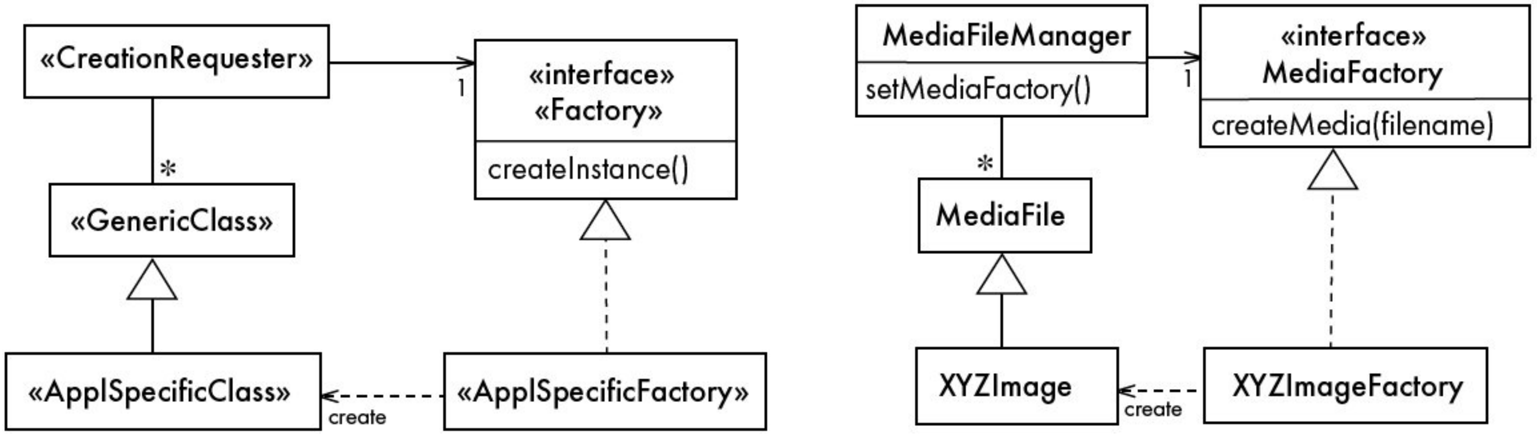
Context:

- You have a reusable framework that needs to create objects as part of its work. However, the class of the created objects will depend on the application.

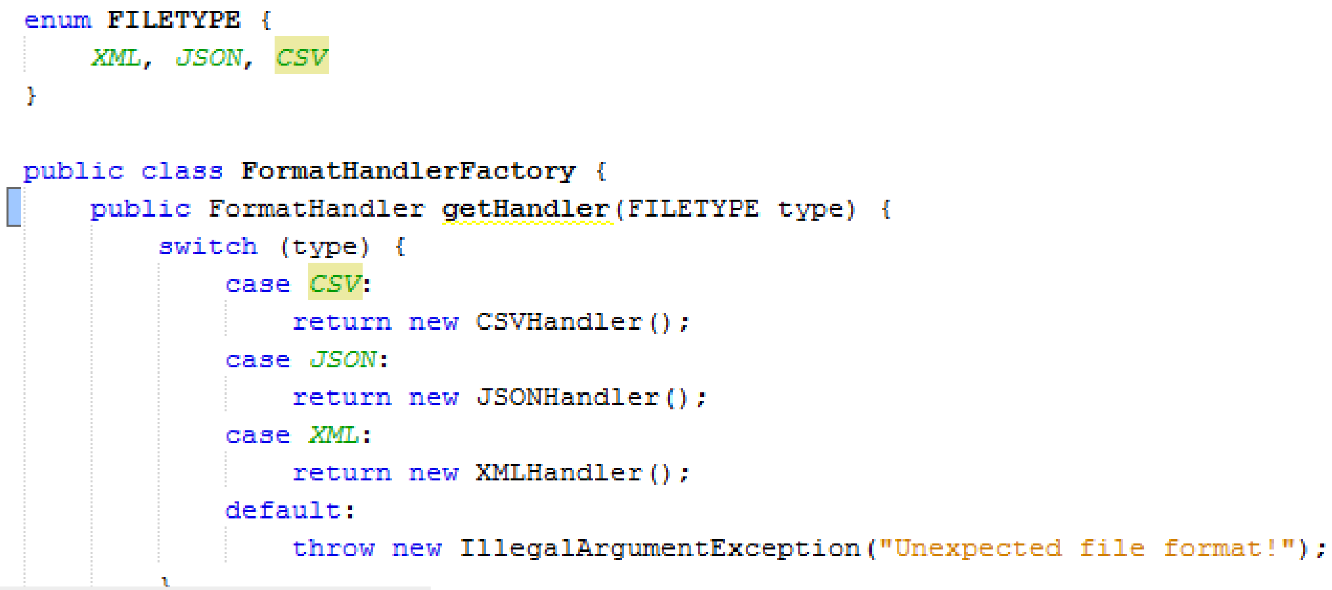
Problem:

- How do you enable a programmer to add a new application-specific class «ApplSpecificClass» into a system built on such a framework?
- How do you arrange for the framework to instantiate this class, without modifying the framework?

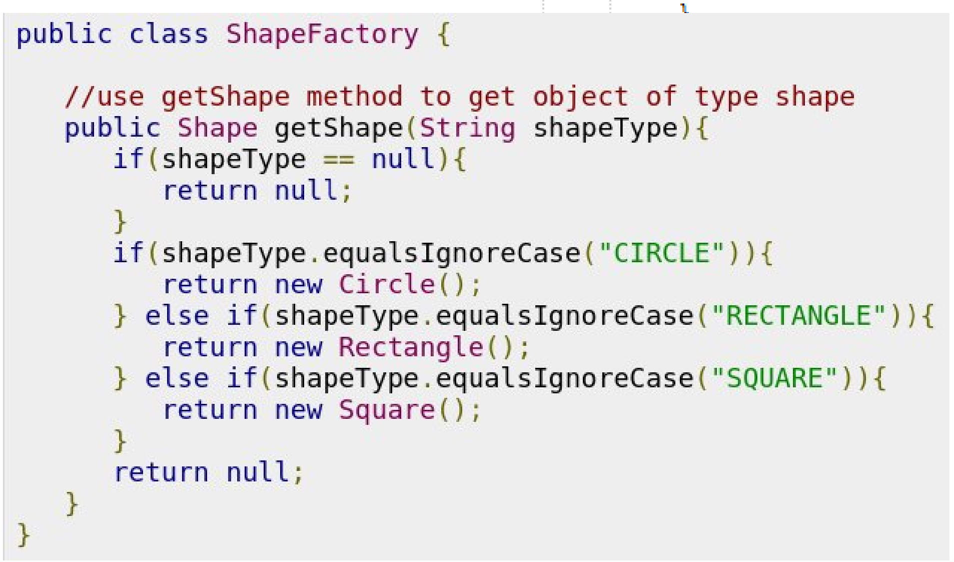
THE FACTORY PATTERN: SOLUTION



THE FACTORY PATTERN: JAVA EXAMPLE



```java
enum FILETYPE {
    XML, JSON, CSV
}

public class FormatHandlerFactory {
    public FormatHandler getHandler(FILETYPE type) {
        switch (type) {
            case CSV:
                return new CSVHandler();
            case JSON:
                return new JSONHandler();
            case XML:
                return new XMLHandler();
            default:
                throw new IllegalArgumentException("Unexpected file format!");
```



```java
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

THE FAÇADE PATTERN: PROBLEM

Context:

During development, classes are added to a component as new functionality is needed leading to an increase in the number of classes the component client has to use and know about

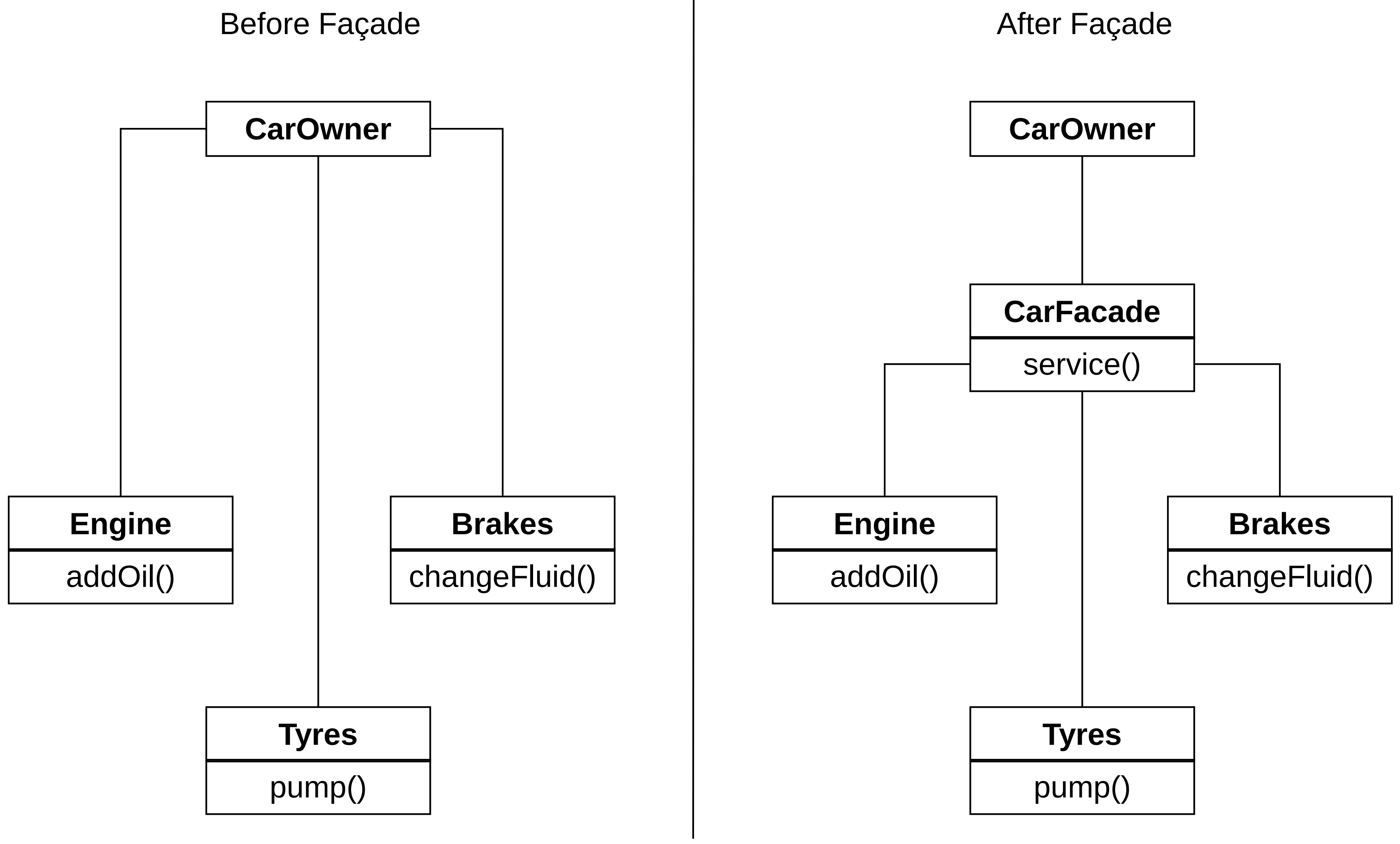
Problem:

The client code can get very complex and tightly coupled with the component. Need to ensure that changes to the component details do not always need to be ‘known’ by the client

Solution:

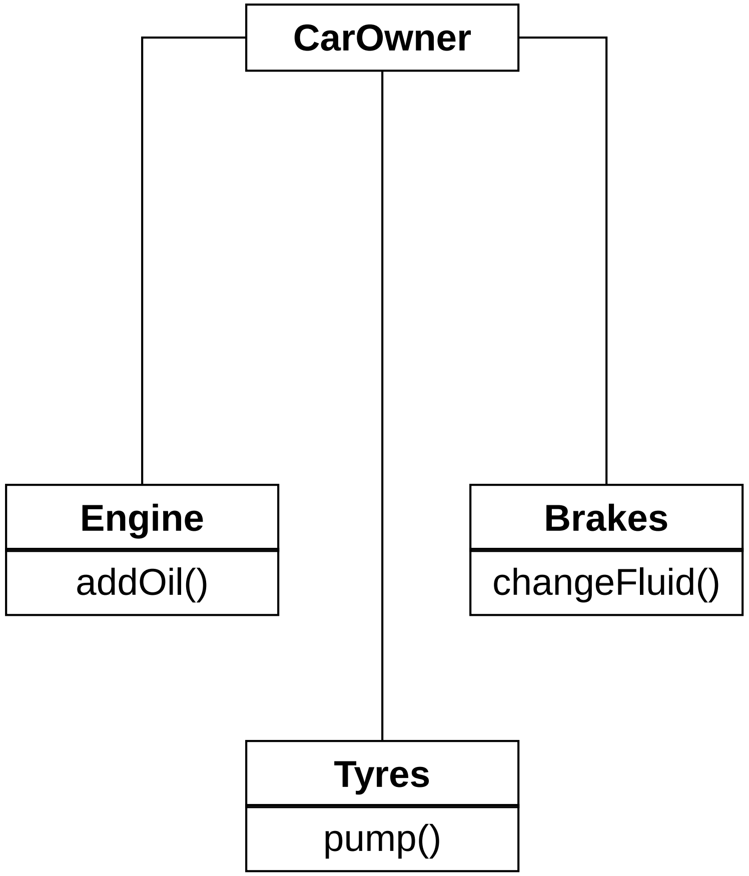
Provide a unified interface to relevant classes/methods in the component

THE FAÇADE PATTERN: SOLUTION



THE FAÇADE PATTERN: SOLUTION

Before Façade



```
public class CarOwner {
    Breaks breaks;
    Engine engine;
    Tyres tyres;

    public void changeBreakFluid(){
        breaks.chageFluid();
    }

    public void addEngineOil(){
        engine.addOil();
    }

    public void pumpTyres(){
        tyres.pump();
    }
}
```

```
public class Breaks {

    public void chageFluid(){

    }
}
```

```
public class Engine {

    public void addOil(){

    }
}
```

```
public class Tyres {

    public void pump(){

    }
}
```

THE FAÇADE PATTERN: SOLUTION

```
public class CarOwner {
    CarFacade car;

    void serviceCar(){
        car.service();
    }
}
```

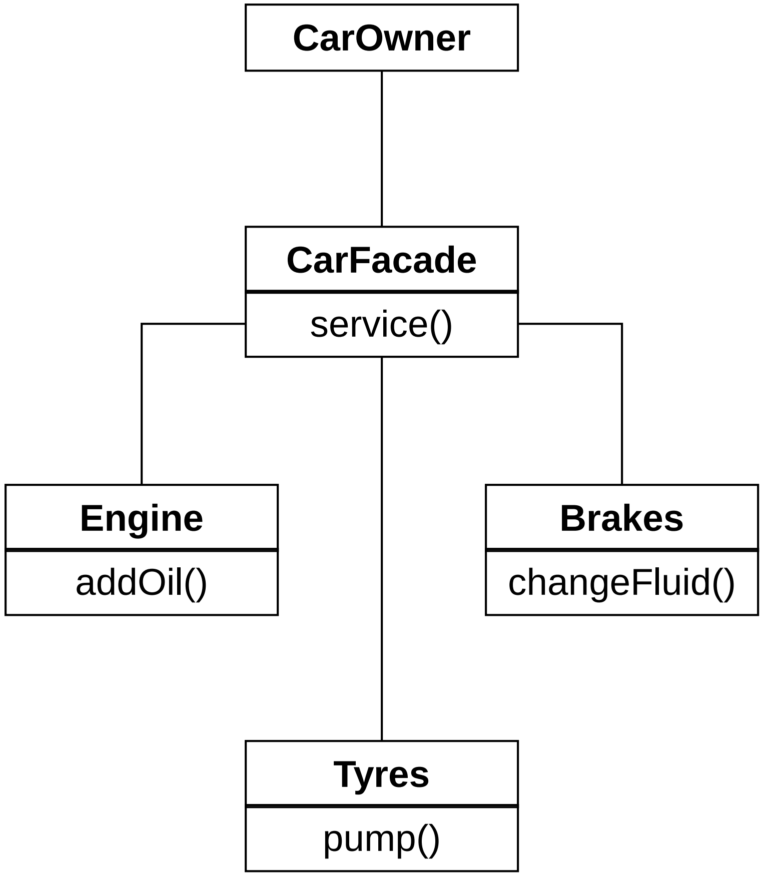
```
public class CarFacade {

    private Breaks breaks;
    private Engine engine;
    private Tyres tyres;

    public void service() {
        breaks.chageFluid();
        engine.addOil();
        tyres.pump();
    }
}
```

```
public class Breaks {

    public void chageFluid(){

    }
}
```

```
public class Engine {

    public void addOil(){

    }
}
```

```
public class Tyres {

    public void pump(){

    }
}
```

After Façade



THE FAÇADE PATTERN: CONSEQUENCES

- Shields clients from subsystem components and makes subsystem easier to use
- Promotes weak coupling between subsystem and its clients

THE MODEL-VIEW-CONTROLLER PATTERN

Queen Mary
University of London

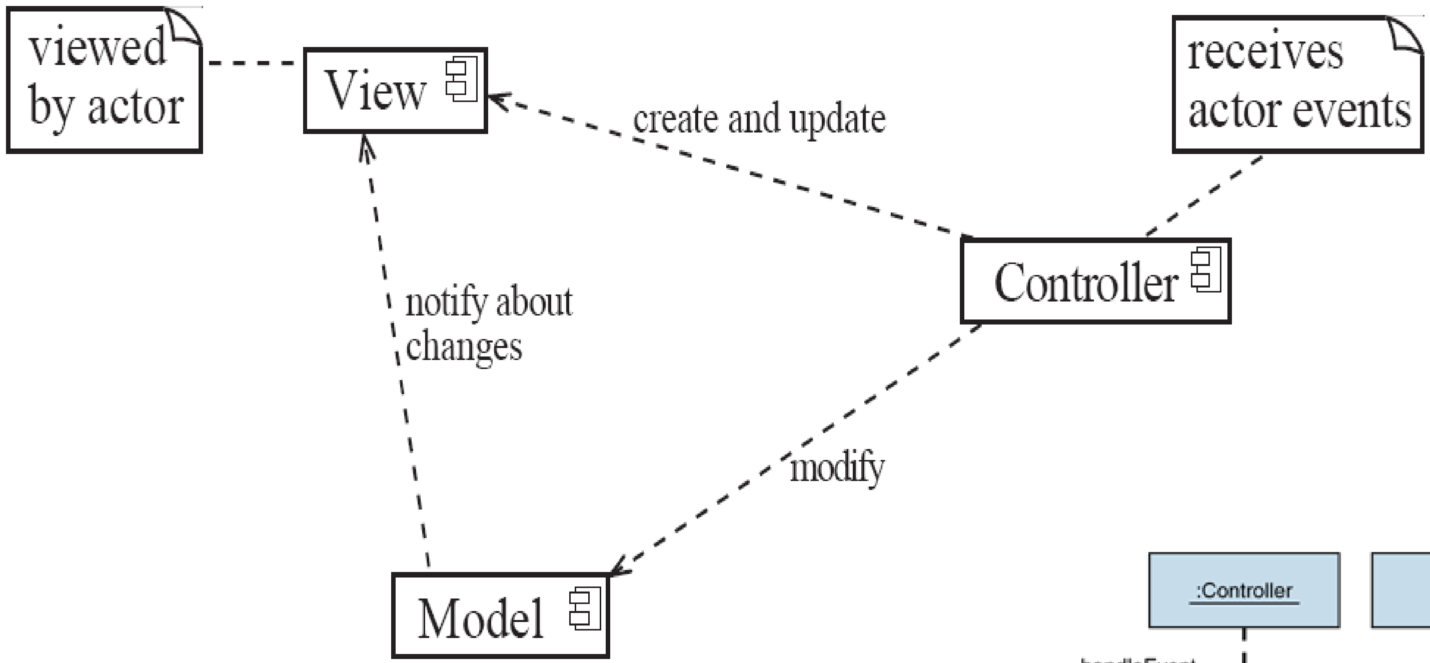
Context:

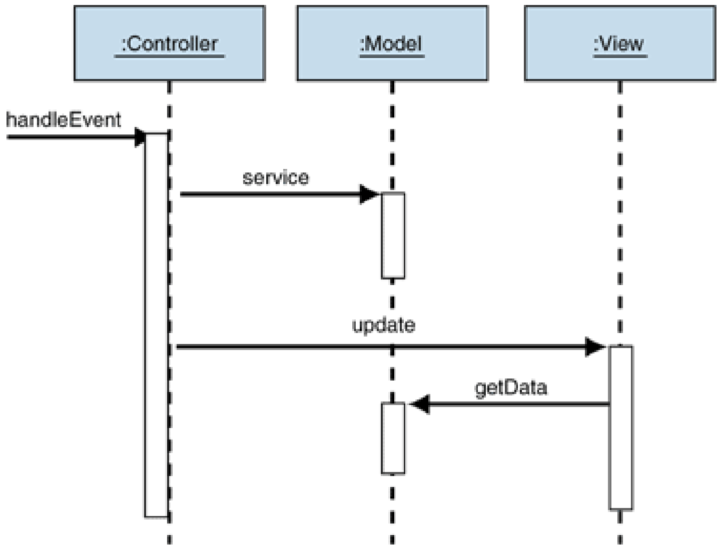
You need a GUI (view) to interact with the core program logic (model)

Problem:

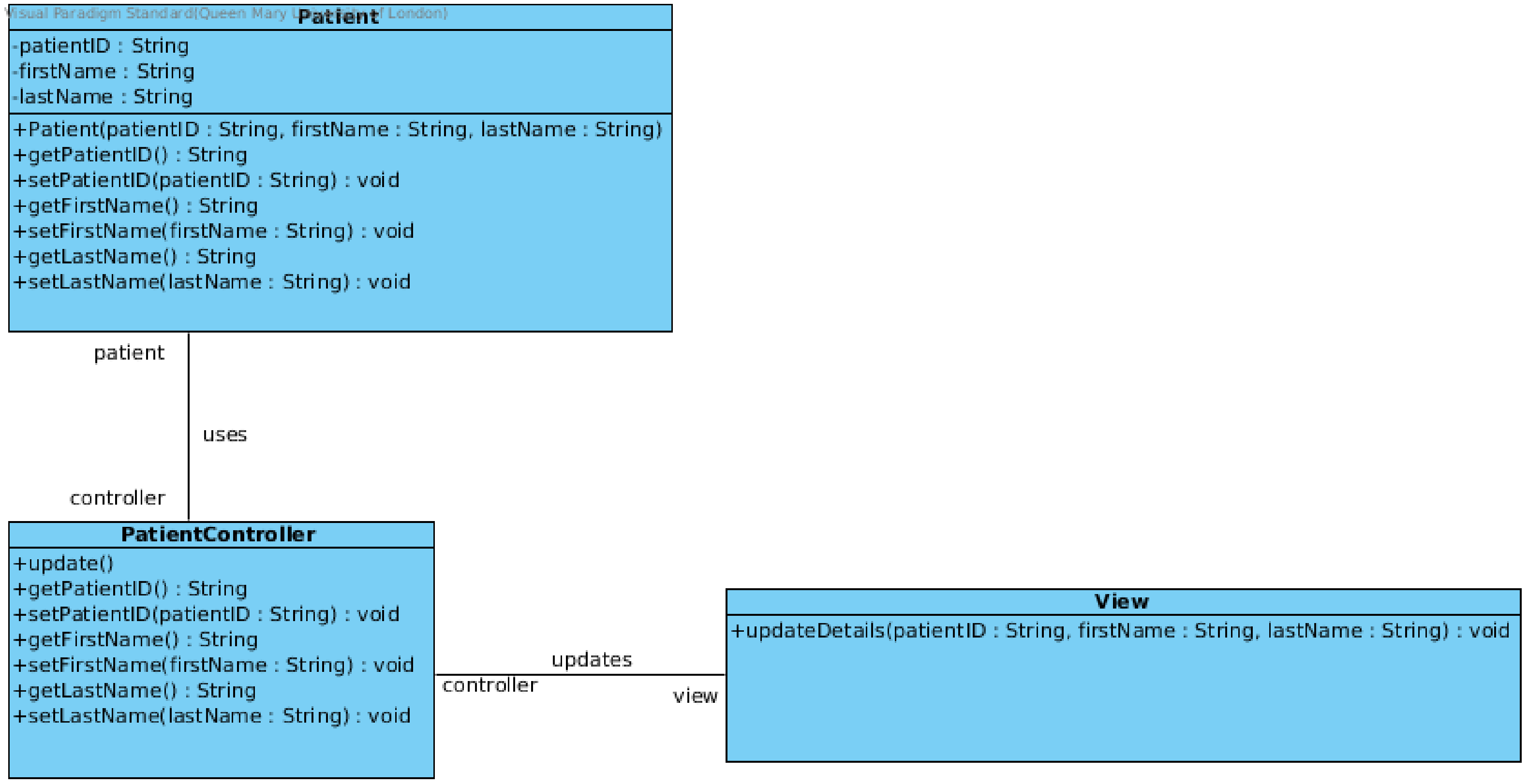
If the logic is intricately dependent on GUI representation of it, then change becomes difficult especially if multiple different views needed. So we need to make the GUI and logic as independent as possible

THE MVC PATTERN: SOLUTION

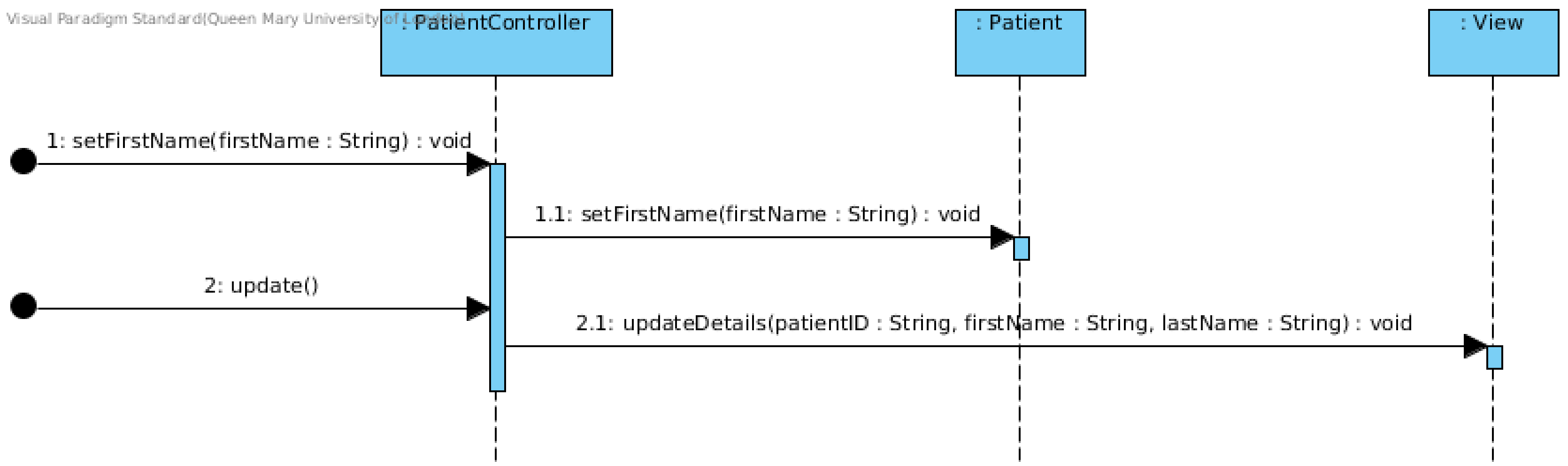




THE MVC PATTERN: SOLUTION

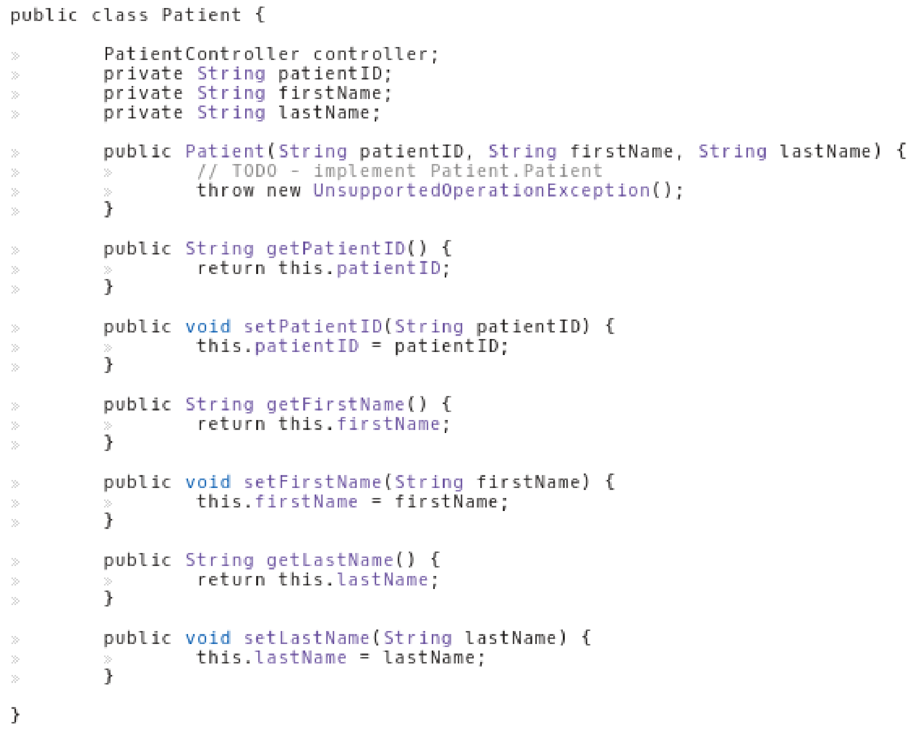

Visual Paradigm Standard(Queen Mary University of London)
Patient
-patientID : String
-firstName : String
-lastName : String
+Patient(patientID : String, firstName : String, lastName : String)
+getPatientID() : String
+setPatientID(patientID : String) : void
+getFirstName() : String
+setFirstName(firstName : String) : void
+getLastName() : String
+setLastName(lastName : String) : void
patient
uses
controller
PatientController
+update()
+getPatientID() : String
+setPatientID(patientID : String) : void
+getFirstName() : String
+setFirstName(firstName : String) : void
+getLastName() : String
+setLastName(lastName : String) : void
updates
controller
view
View
+updateDetails(patientID : String, firstName : String, lastName : String) : void

THE MVC PATTERN: SOLUTION

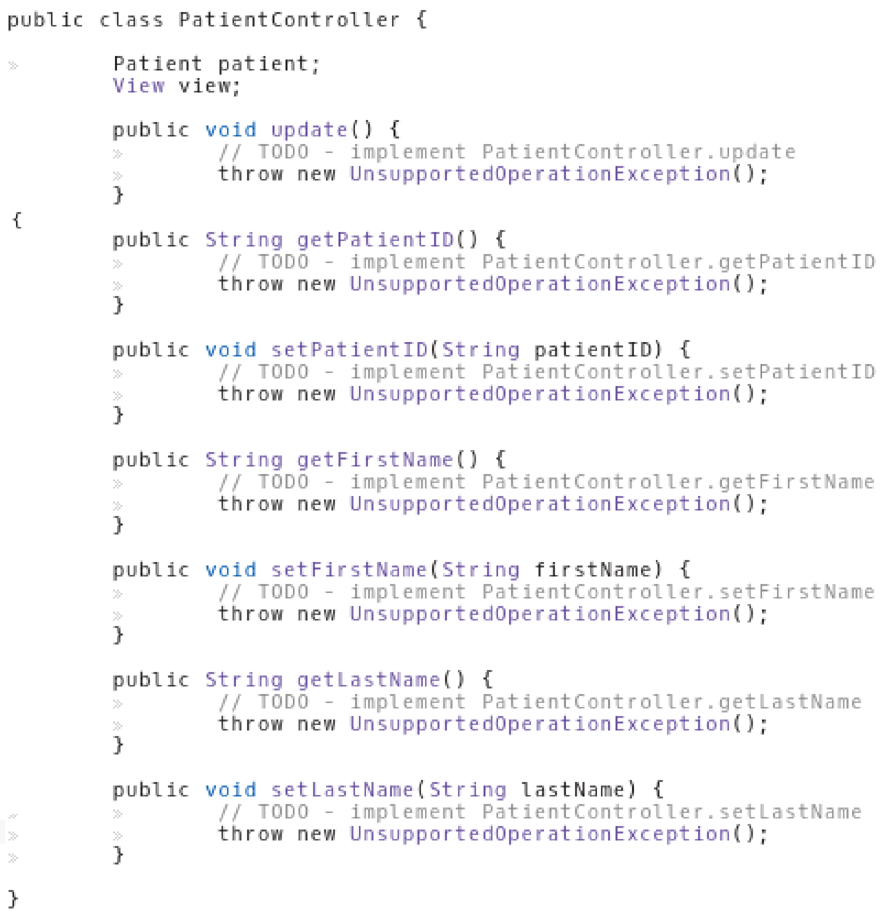


Visual Paradigm Standard(Queen Mary University of London)
: PatientController
: Patient
: View
1: setFirstName(firstName : String) : void
1.1: setFirstName(firstName : String) : void
2: update()
2.1: updateDetails(patientID : String, firstName : String, lastName : String) : void

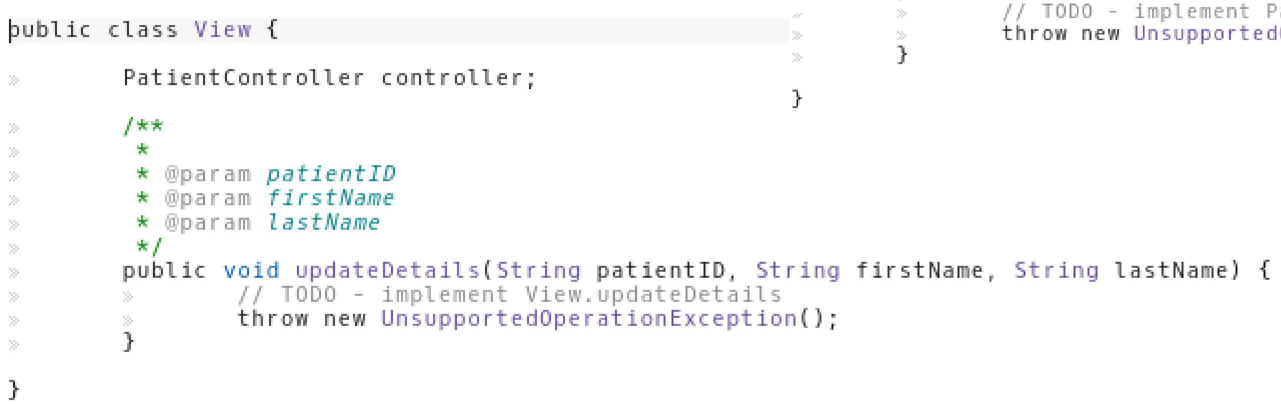
THE MVC PATTERN: JAVA EXAMPLE



```java
public class Patient {

        PatientController controller;
        private String patientID;
        private String firstName;
        private String lastName;

        public Patient(String patientID, String firstName, String lastName) {
                // TODO - implement Patient.Patient
                throw new UnsupportedOperationException();
        }

        public String getPatientID() {
                return this.patientID;
        }

        public void setPatientID(String patientID) {
                this.patientID = patientID;
        }

        public String getFirstName() {
                return this.firstName;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public String getLastName() {
                return this.lastName;
        }

        public void setLastName(String lastName) {
                this.lastName = lastName;
        }

}
```



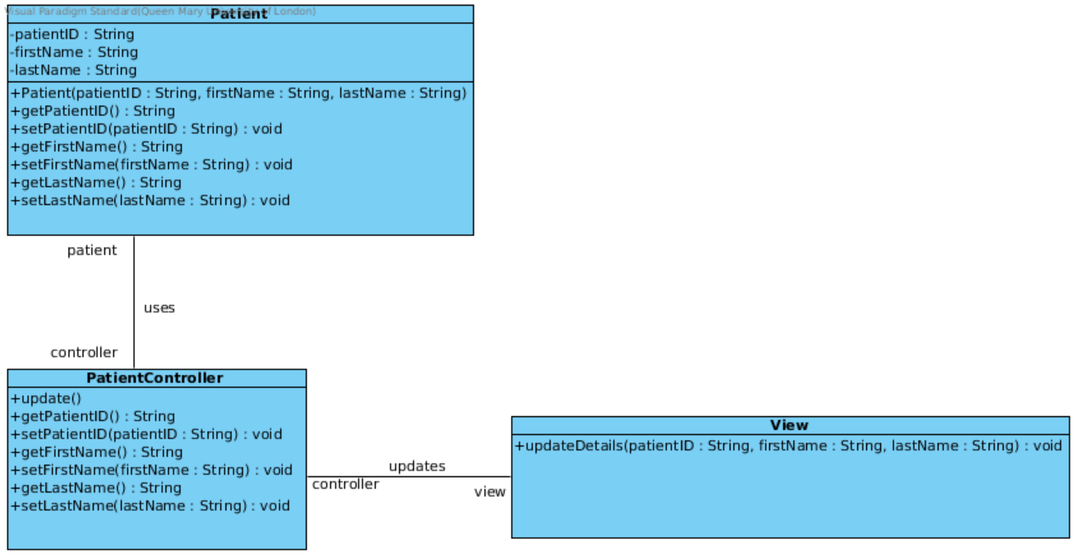
```java
public class PatientController {

        Patient patient;
        View view;

        public void update() {
                // TODO - implement PatientController.update
                throw new UnsupportedOperationException();
        }

        public String getPatientID() {
                // TODO - implement PatientController.getPatientID
                throw new UnsupportedOperationException();
        }

        public void setPatientID(String patientID) {
                // TODO - implement PatientController.setPatientID
                throw new UnsupportedOperationException();
        }

        public String getFirstName() {
                // TODO - implement PatientController.getFirstName
                throw new UnsupportedOperationException();
        }

        public void setFirstName(String firstName) {
                // TODO - implement PatientController.setFirstName
                throw new UnsupportedOperationException();
        }

        public String getLastName() {
                // TODO - implement PatientController.getLastName
                throw new UnsupportedOperationException();
        }

        public void setLastName(String lastName) {
                // TODO - implement PatientController.setLastName
                throw new UnsupportedOperationException();
        }

}
```



```java
public class View {

        PatientController controller;

        /**
         *
         * @param patientID
         * @param firstName
         * @param lastName
         */
        public void updateDetails(String patientID, String firstName, String lastName) {
                // TODO - implement View.updateDetails
                throw new UnsupportedOperationException();
        }

}
```

THE MVC PATTERN: CONCEQUENCES

- Modular design (allows divide and conquer)
- Increased cohesion.
- Reduced coupling.
- Increased reuse.
- Design for flexibility.
- Design for testability.



Visual Paradigm Standard(Queen Mary (University of London)
Patient
-patientID : String
-firstName : String
-lastName : String
+Patient(patientID : String, firstName : String, lastName : String)
+getPatientID() : String
+setPatientID(patientID : String) : void
+getFirstName() : String
+setFirstName(firstName : String) : void
+getLastName() : String
+setLastName(lastName : String) : void
patient
uses
controller
PatientController
+update()
+getPatientID() : String
+setPatientID(patientID : String) : void
+getFirstName() : String
+setFirstName(firstName : String) : void
+getLastName() : String
+setLastName(lastName : String) : void
updates
controller
view
View
+updateDetails(patientID : String, firstName : String, lastName : String) : void

DIFFICULTIES AND RISKS WHEN USING DESIGN PATTERNS

- Patterns are not solution to all problems
- Design with patterns requires experience

SUMMARY

- Proven solutions to OOP problems
- Common vocabulary for designers
- Good way for beginners to become better designers
- Makes standard libraries (Java, C++ etc.) easier to use
- MAY help you at university
- WILL help you in industry
- BUT be careful not to overuse them