Lecture 7 –Design Principles and Design Patterns

COMP2013 (AUT1 23-24)

Dr Marjahan Begum and Dr Horia A. Maior

Register your attendance

COMP2013: Developing Maintainable Software
Week 8 – 4:00pm Monday – 13 November 2023

valid for 65 minutes from 3:55pm
generated 2023-10-10 03:14

COMP2013: Developing Maintainable Software
Week 8 – 4:00pm Monday – 13 November 2023

valid for 65 minutes from 3:55pm
generated 2023-10-10 03:14

Overview

- Q/A with a 3rd Year Student
- Generating automatic class diagram
- SOLID principles
- Design Patterns

Topics for this Week

- Lecture 07A:
  - Design Principles and Design Patterns
- Lecture 07B:
  - Continuation of 07A
  - Review of some exam related questions
- Lab 07:
  - Continue with your coursework

From Concepts to Patterns

Design Patterns and SOLID Principles extend the Object Oriented Philosophy

From Concepts to Patterns

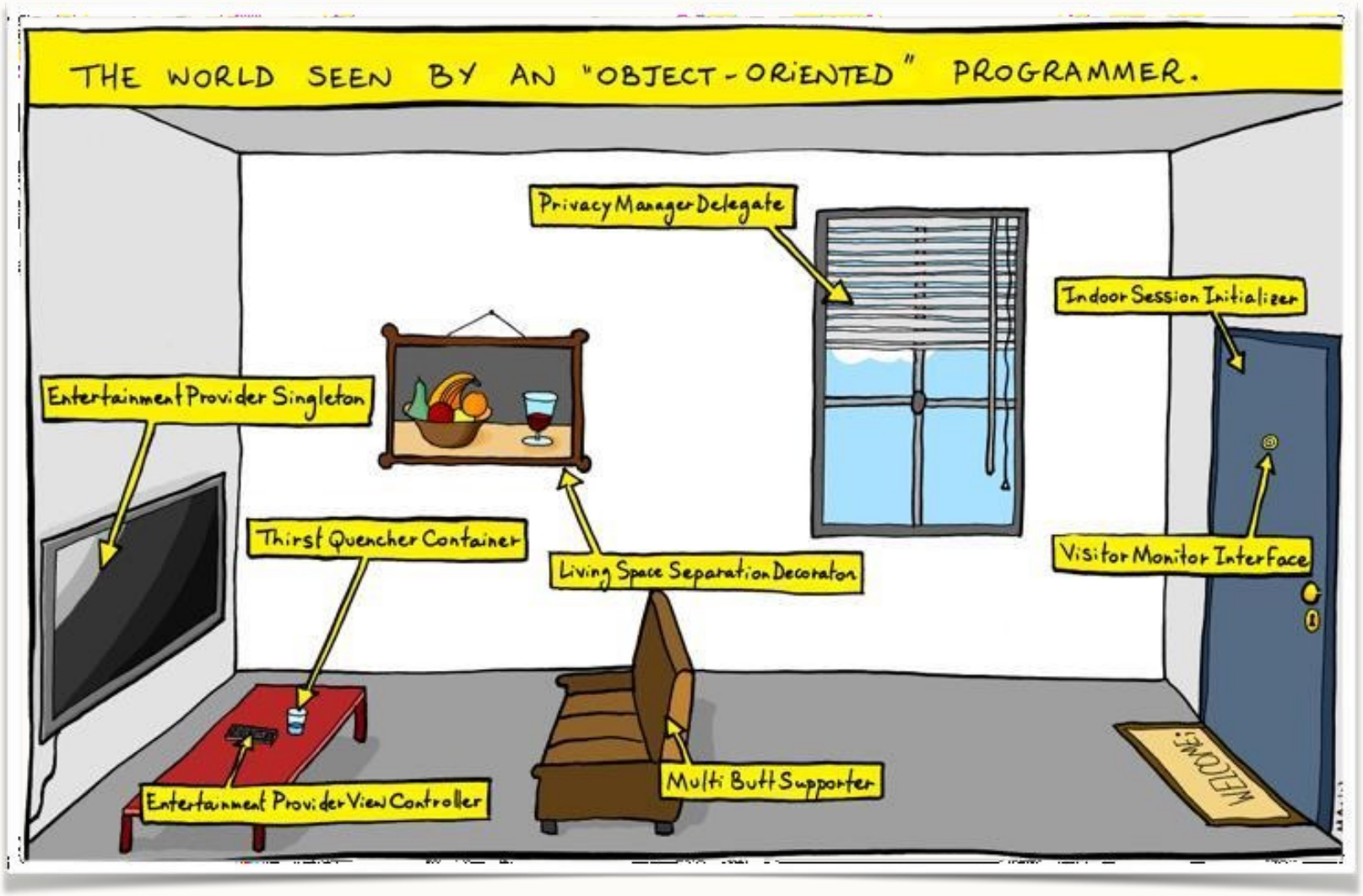
- OO Concepts
  - Object model (abstraction, encapsulation, modularity, hierarchy); data abstraction; inheritance; polymorphism; interfaces
- OO Design Principles
  - Encapsulate what varies; favour object aggregation over class inheritance; program to interfaces not implementations;
- OO Design Patterns:
  - Show how to build systems with good OO design qualities (reusable; extensible; maintainable)

Why OOD Principles and Patterns

- Software development specifications keep changing constantly.
- Your software solves real life problems and processes in real life (and your knowledge) evolve.
    - This is in particular true in the business world.
- Your software need to be smart enough to cope.
  - Easy to maintain (by applying changes with minimal effort)
  - Easy to extend (without changing existing code)
- Following systematically conventions so others can also maintain your code.

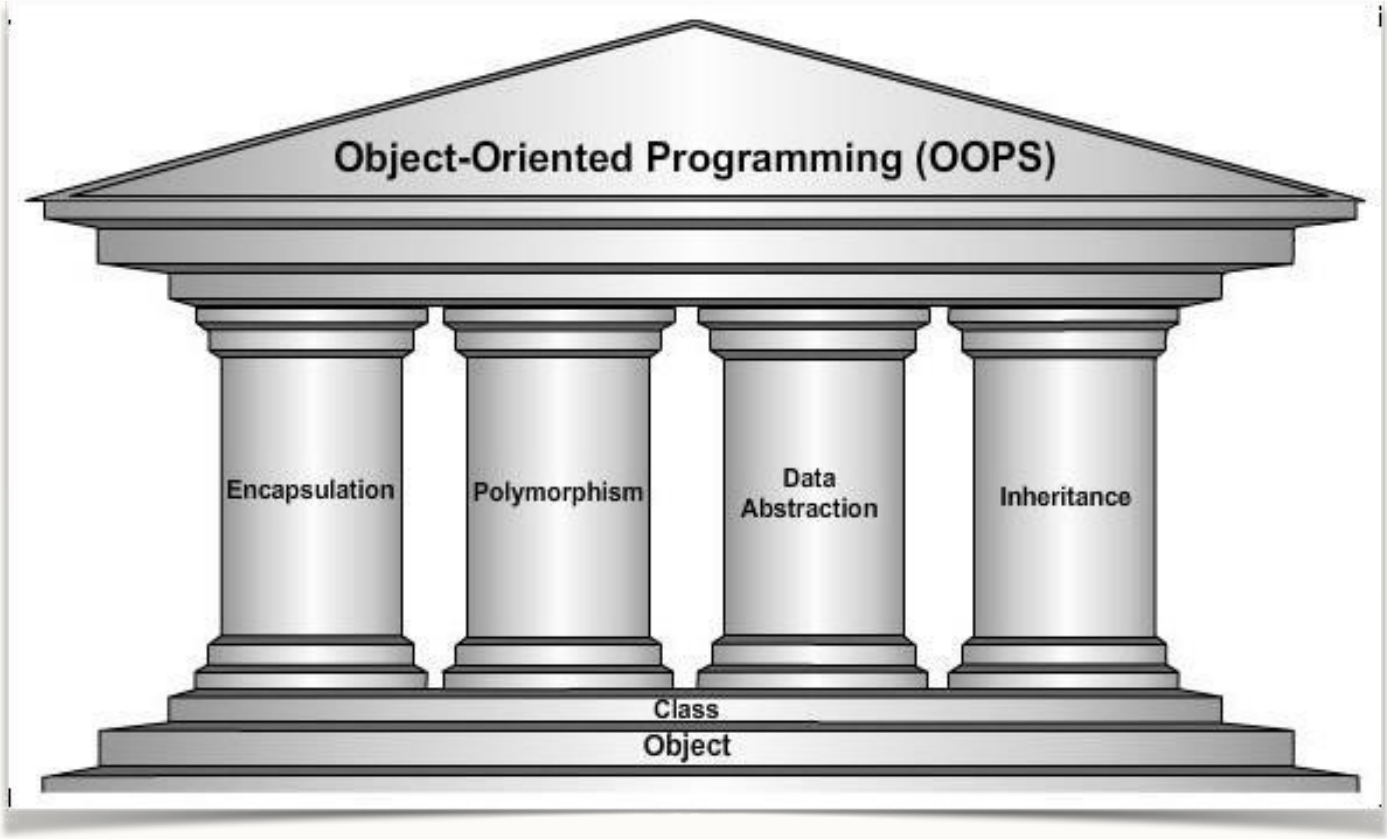
You don’t always have to Object Orientate, but when you do…

The Bottom Line: Object Orientation


THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Privacy Manager Delegate
Indoor Session Initializer
Entertainment Provider Singleton
Thirst Quencher Container
Living Space Separation Decorator
Visitor Monitor Interface
Entertainment Provider View Controller
Multi Butt Supporter
WELCOME!

The Pillars of Object Orientation

- Abstraction
- Inheritance
- Encapsulation
- Polymorphism



The Object Model [Booch 1994]

- Abstraction
  - Denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
    - Concentrating only on essential characteristics.

- Encapsulation:
  - Process of compartmentalising the elements of an abstraction that constitute its structure and behaviour.
    - Encapsulation serves to separate the contractual interface of an abstraction and its implementation.

The Object Model [Booch 1994]

- Modularity:
  - Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules (classes).
    - Cohesive: Performing a single type of tasks
    - Loosely coupled: Highly independent

- Hierarchy:
  - A ranking or ordering of abstractions
    - Classes at the same level of the hierarchy should be at the same level of abstraction.

Core O-O Concepts: Abstraction and Inheritance

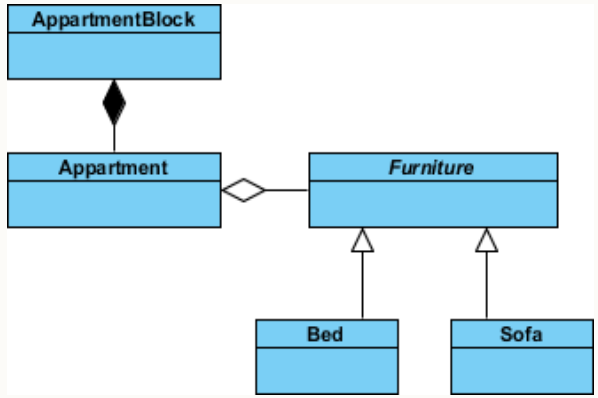
- Data Abstraction
  - Application of implementation hiding – you don't need to know all the details of how something works in order to make use of it.
  - Creating new data types that are well suited to an application; allows the creation of user defined data types, having the properties of build in data types and a set of permitted operators.
  - *Abstract Data Type*: A structure that contains both, data and the definition of actions to be performed with that data where a class is an implementation of an abstract data type.

Core O-O Concepts: Abstraction and Inheritance

- Inheritance
  - Inheritance defines implementation of a class in terms of another's
  - New data types can be defined as extensions to previously defined types
  - This allows reuse since the implementation is not repeated

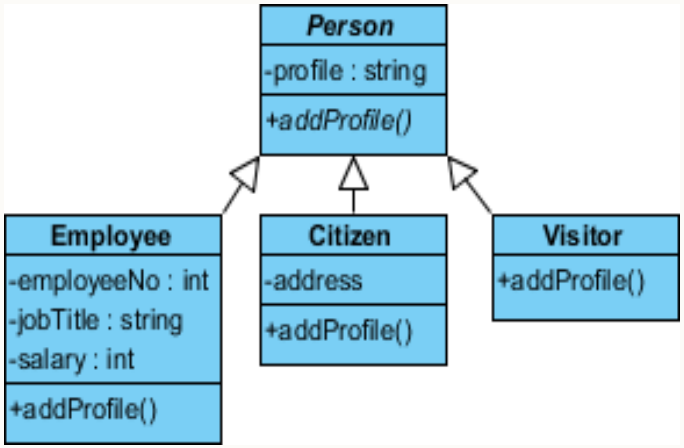
Other Object Oriented Concepts

- Composition vs Aggregation – two forms of specialised containment
  - **Composition**
    - A relationship where a child cannot exist independent of the parent e.g., an apartment cannot exist outside of the ApartmentBlock
    - Implies that the internal objects are not seen from the outside
  - **Aggregation**
    - Implies that the child can exist without the parent e.g., the bed can exist outside of the apartment.
    - Obtaining new functionality by assembling objects
      – This allows reuse as aggregated objects have a separate existence.
    - Aggregated objects might be directly accessed.



Core O-O Concept: Abstract vs Concrete Classes

- Abstract class
  - A class that cannot be directly instantiated
  - An abstract class is written with the expectation that its concrete subclasses will add to its structure and behaviour.
  - Denote by marking classes and methods as abstract
- Concrete class
  - A class that can be directly instantiated
  - A class whose implementation is complete and thus may have instances.
- Still confused? Try this: https://www.youtube.com/watch?v=hZ1EU-F-0nU



Core O-O Concept: Polymorphism

- This is treating different subclasses as the same superclass.

- Polymorphism through method overloading (design time polymorphism)
  - Create more than one function with same name but different signatures.

- Polymorphism through method overriding (run time polymorphism)
  - Create a function in the derived class with the same name and signature.

Core O-O Concept: Polymorphism

- Polymorphism through sub classing (run time polymorphism)
  - A reference variable of a base class is able to reference, instantiate and destroy objects of a derived class.
  - A reference variable may reference objects of many classes at runtime, but the compiler cannot predict which they will be at compile time.

- Still confused? Try this: https://www.youtube.com/watch?v=SwEzCBM7n-Q

Core O-O Concept: Interfaces

- In most cases interfaces are only method declarations and do not implement any of the methods.
- An interface describes the behavior or capabilities of a class without committing
  to a particular implementation of that class.
- An interface is a contract of related services and a set of conditions that must be true for the contract to be faithfully executed.
- Interfaces formalise polymorphism, they allow us to define polymorphism in a declarative way unrelated to implementation.
- Still confused? Try this: https://www.youtube.com/watch?v=fX1xNMBTMfg

Concurrency Framework in JavaFX

- Framework consist of one interface + four classes + one enum
  - **Worker** interface specifies the methods available to background (work) threads
  - **Task** class instance represents a one-shot task
  - **Service** class instance represents a reusable task
  - **ScheduledService** class instance represents a reusable task that runs repeatedly following a specified interval
  - **WorkerStateEvent** class instance represents an event that occurs as state of Worker changes
    - You can add event handlers to all three types of tasks to listen to the changes in their states
  - **State** enum constants represents different states of a worker

Self test: Things you should be able to define by now

- Inheritance is…
- Polymorphism is…
- Abstraction is…
- Encapsulation is…
- An interface is…
- Concrete classes are …
- Abstract classes are …

SOLID Principles

Object Oriented Design Principles for Clean Coders

Uncle Bob

- Robert C. Martin aka ‘Uncle Bob’
- Agile and OOP guru
- Derive principles to improve the state of the art of *software craftsmanship*
  - *“Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best.”*

“SOLID” Design Principles and Clean

- Software solves real life business problems and real life business processes evolve
  and change - always.
- A smartly designed software can adjust changes easily; it can be extended, and it is re-usable.
- SOLID Principles (by Uncle Bob)
  [http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod]
  - S = Single Responsibility Principle
  - O = Open-Closed Principle
  - L = Liskov's Substitution Principle
  - I = Interface Segregation Principle
  - D = Dependency Inversion Principle

Code Sense

- "A programmer without code-sense can look at a messy module and recognise the mess but will have no idea what to do about it"

- "A programmer with code-sense will look at a messy module and see options and variations. The code-sense will help the programmer chose the best variation and guide him or her to plot a sequence of behaviour-preserving transformations to get from here to there."

- SOLID principles help us to develop our code sense

- As with all design principles and patterns, they do not involve code reuse.
  – they "employ experience reuse"

Remembering the Mnemonic

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

- UB says: "A class should have one and only one responsibility"

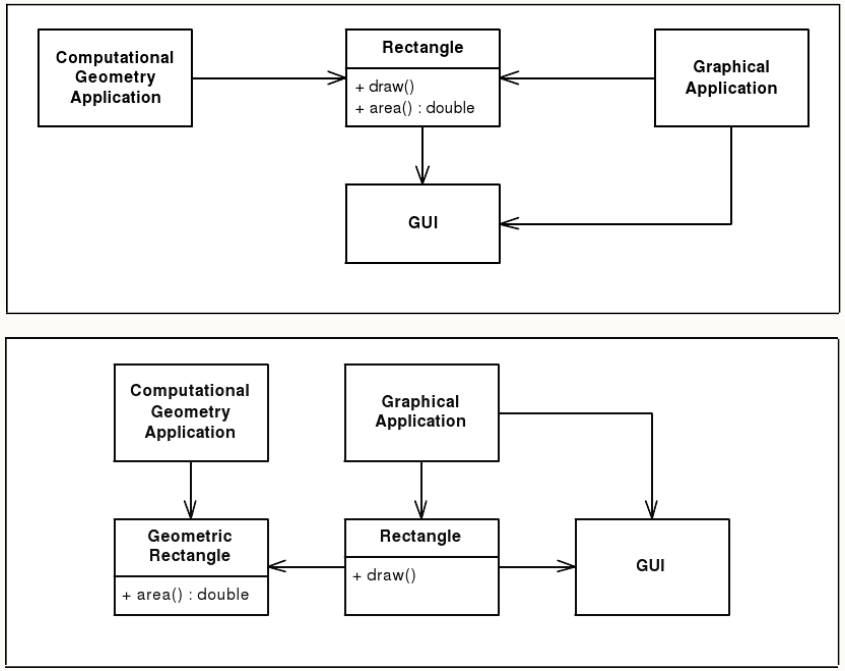
Single Responsibility Principle

- UB says: "A class should have one and only one responsibility"
  - Code becomes coupled if classes have more than one responsibility.
    - Modifications done carry the risk of affecting other parts of the class
  - Classes with more than one responsibility need to be split up.
    - Supports reuse
    - Supports modifying minimal amount of code

- A method should have one and only one responsibility.
  - Methods should be broken down so that they do only one job.
    - Supports reuse
    - Supports modifying minimal amount of code

Single Responsibility Principle

- A classic example from Uncle Bob



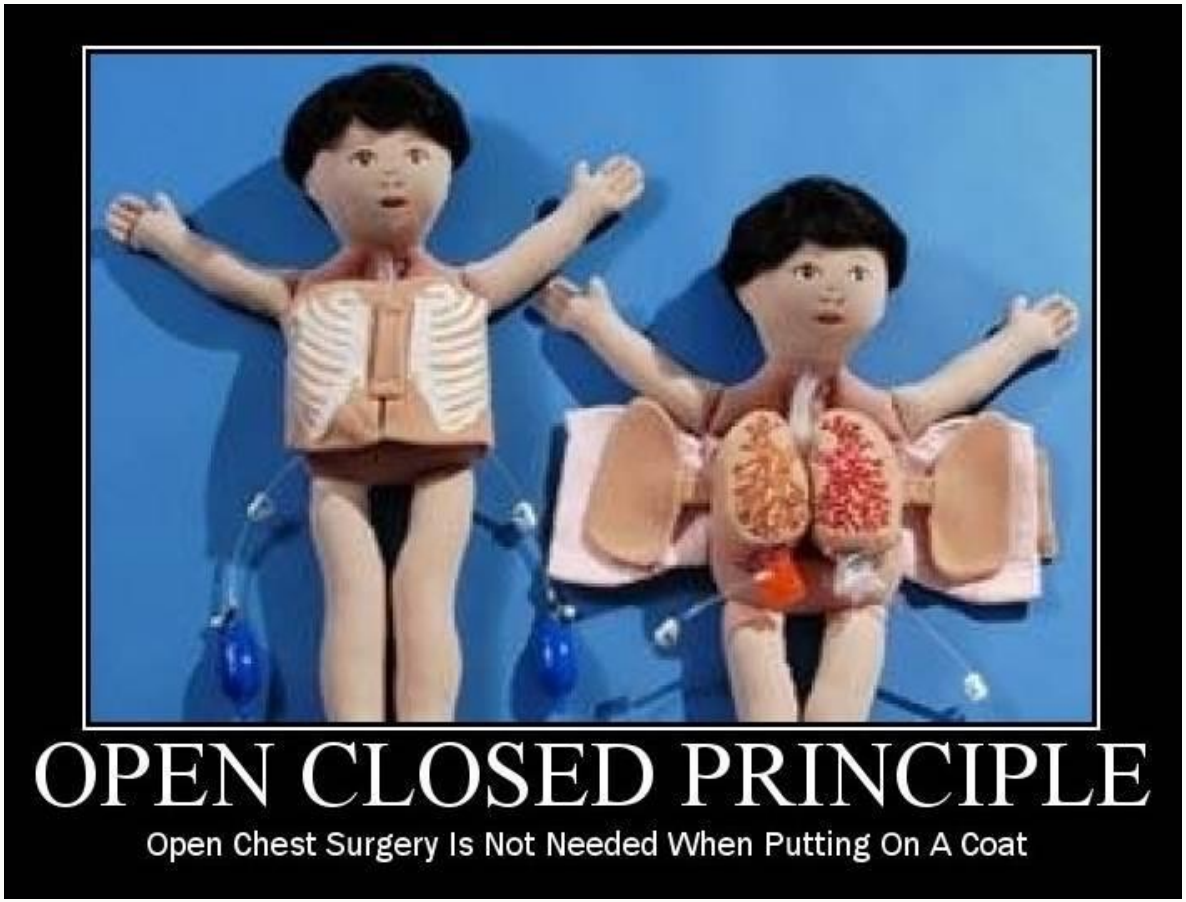
Single Responsibility Principle

- Here's an example of a Java class that does not follow the single responsibility
  principle (SRP):

```
public class Vehicle {
    public void printDetails() {}
    public double calculateValue() {}
    public void addVehicleToDB() {}
}
```

- By applying SRP, we can separate the above class into three classes with separate responsibilities.

Open-Closed Principle



- "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

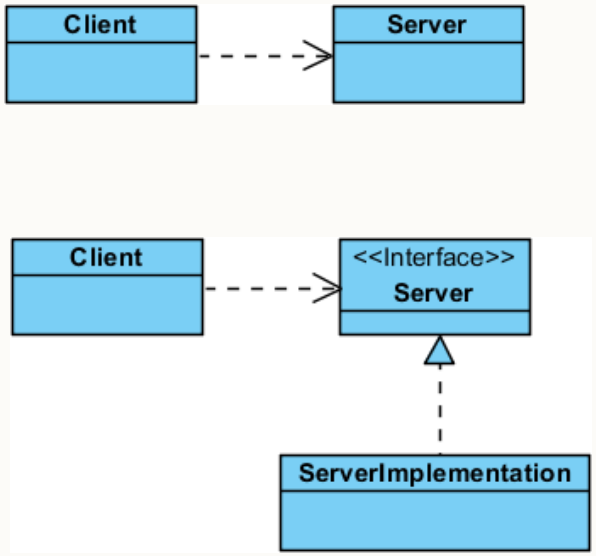
Open-Closed Principle

- "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."
  - Open for Extension
    - The behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
  - Closed for Modification
    - The source code of such a module is inviolate. No one is allowed to make source code changes to it.

"Abstraction" is the Key

Open-Closed Principle

- A classic example from Uncle Bob making good use of interfaces



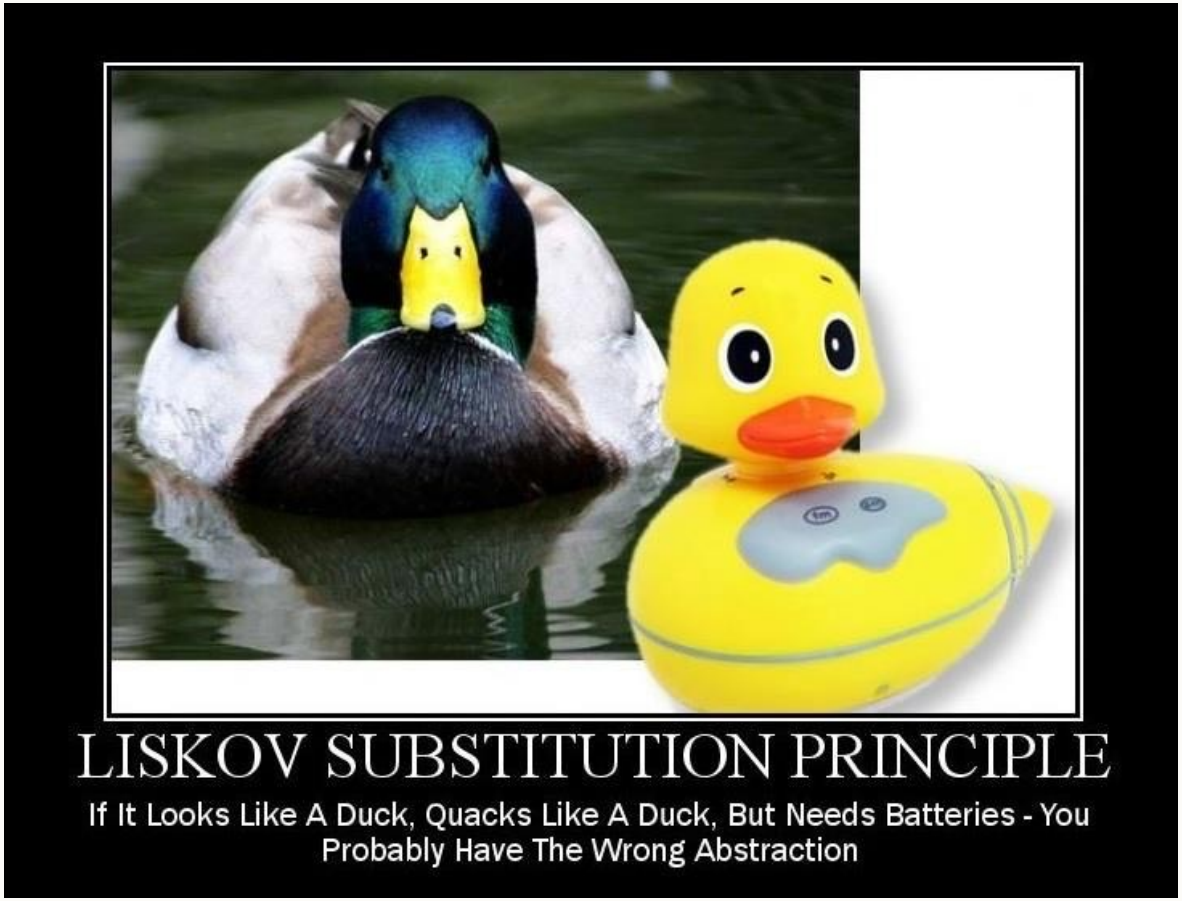
Open-Closed Principle

- Consider the below method of the class VehicleCalculations.
- A better approach would be for the subclasses Car and Truck to override the calculateValue method.

```
public class VehicleCalculations {
    public double calculateValue(Vehicle v) {
        if (v instanceof Car) {
            return v.getValue() * 0.8;
        if (v instanceof Bike) {
            return v.getValue() * 0.5;

    }
}
```

```
public class Vehicle {
    public double calculateValue() {...}
}
public class Car extends Vehicle {
    public double calculateValue() {
        return this.getValue() * 0.8;
}
public class Truck extends Vehicle{
    public double calculateValue() {
        return this.getValue() * 0.9;
}
```

Liskov's Substitution Principle



- UB says: ""Subtypes must be substitutable for their base types."

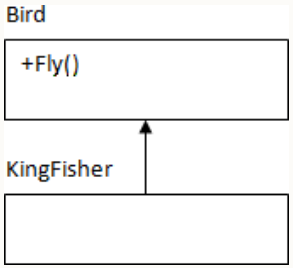
Liskov's Substitution Principle

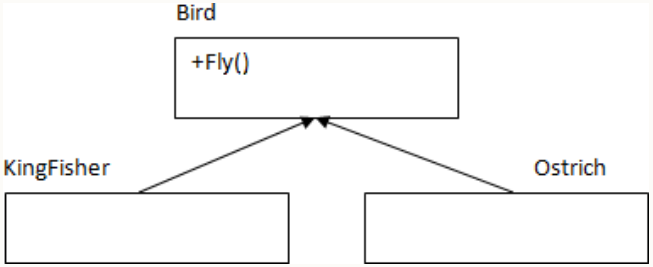
- UB says "Subtypes must be substitutable for their base types."
  - Methods that use references to base classes must be able to use objects of derived classes without knowing it.
  - The "is a" technique of determining inheritance relationships is simple and useful, but occasionally results in bad use of inheritance.

This is just a way of ensuring that "inheritance" is used correctly

Liskov's Substitution Principle

- A classic example from Uncle Bob by thinking about the functionality of classes.





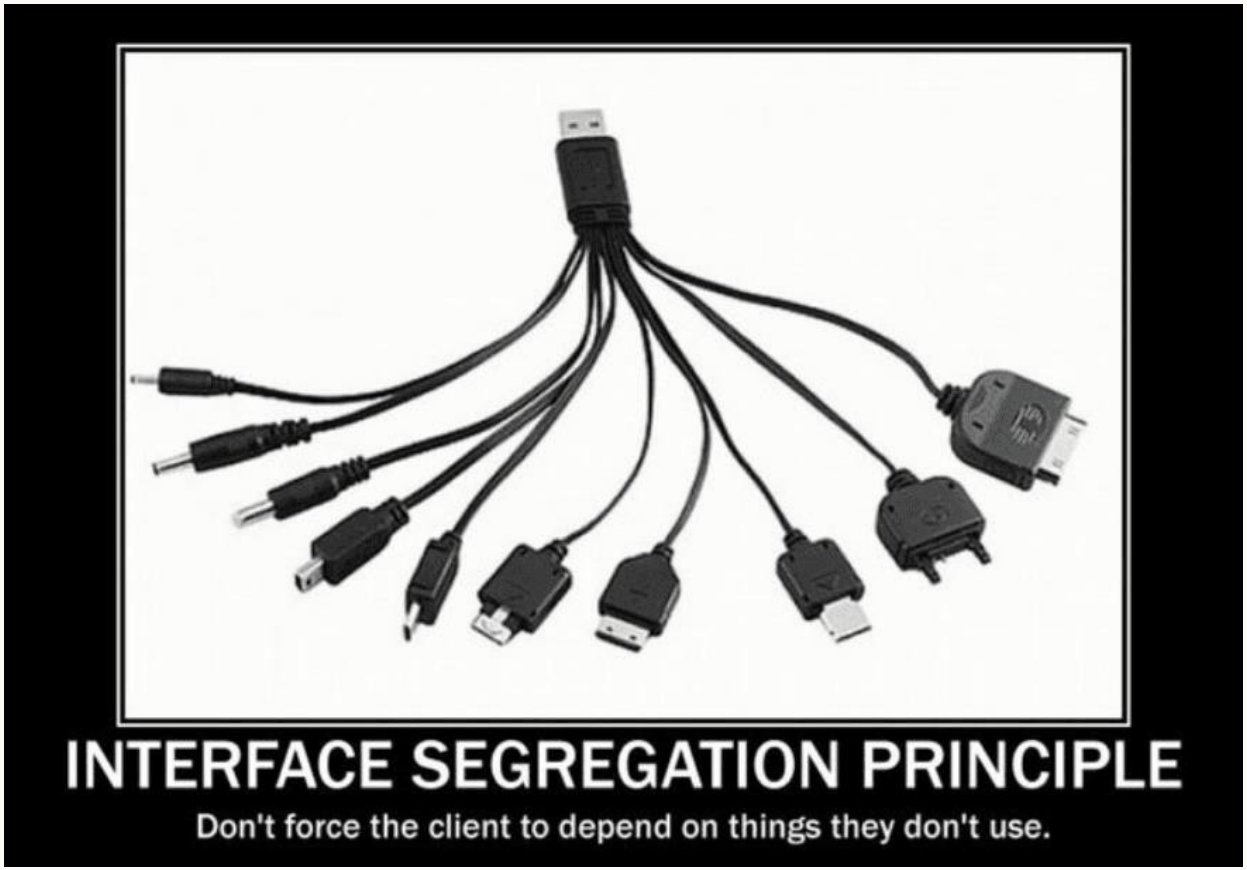
  - There should be a separate class for birds that can't really fly and Ostrich should inherit that.
  - The subclass needs to be able to properly use the methods of the superclass, else the abstraction is wrong

Liskov's Substitution Principle

- The below classes violated the Liskov substitution principle because the Square class has extra constraints i.e., height and weight that must be the same.

```
public class Rectangle {
    private double height;
    private double width;
    public void setHeight(double h) { height = h; }
    public void setWidht(double w) { width = w; }
    ...
}
public class Square extends Rectangle {
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
    public void setWidth(double w) {
        super.setHeight(w);
        super.setWidth(w);
    }
}
```

Interface Segregation Principle



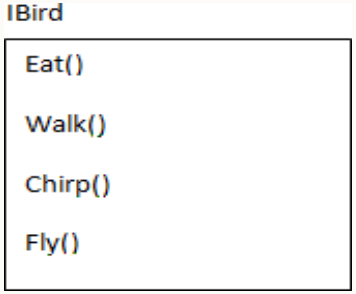
- UB says: "Clients should not be forced to depend upon interfaces that they do not use."

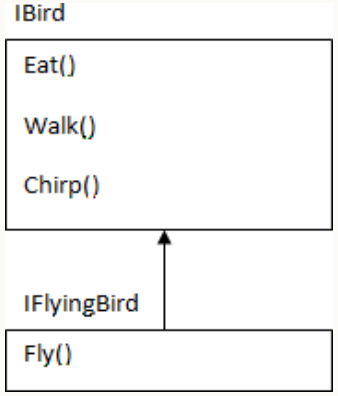
Interface Segregation Principle

- UB says "Clients should not be forced to depend
  upon interfaces that they do not use."
  - If a class wants to implement the interface it has to implement all the methods.
  - Fat interfaces should be broken down.
  - The principle ensures that interfaces are developed so that each of them have their own responsibility and thus they are specific, easily understandable and reusable.
  - This is also an extension of the *single responsibility principle.*
    - Nice example: https://dzone.com/articles/solid- principles-interface-segregation-principle

Interface Segregation Principle

- A classic example from Uncle Bob





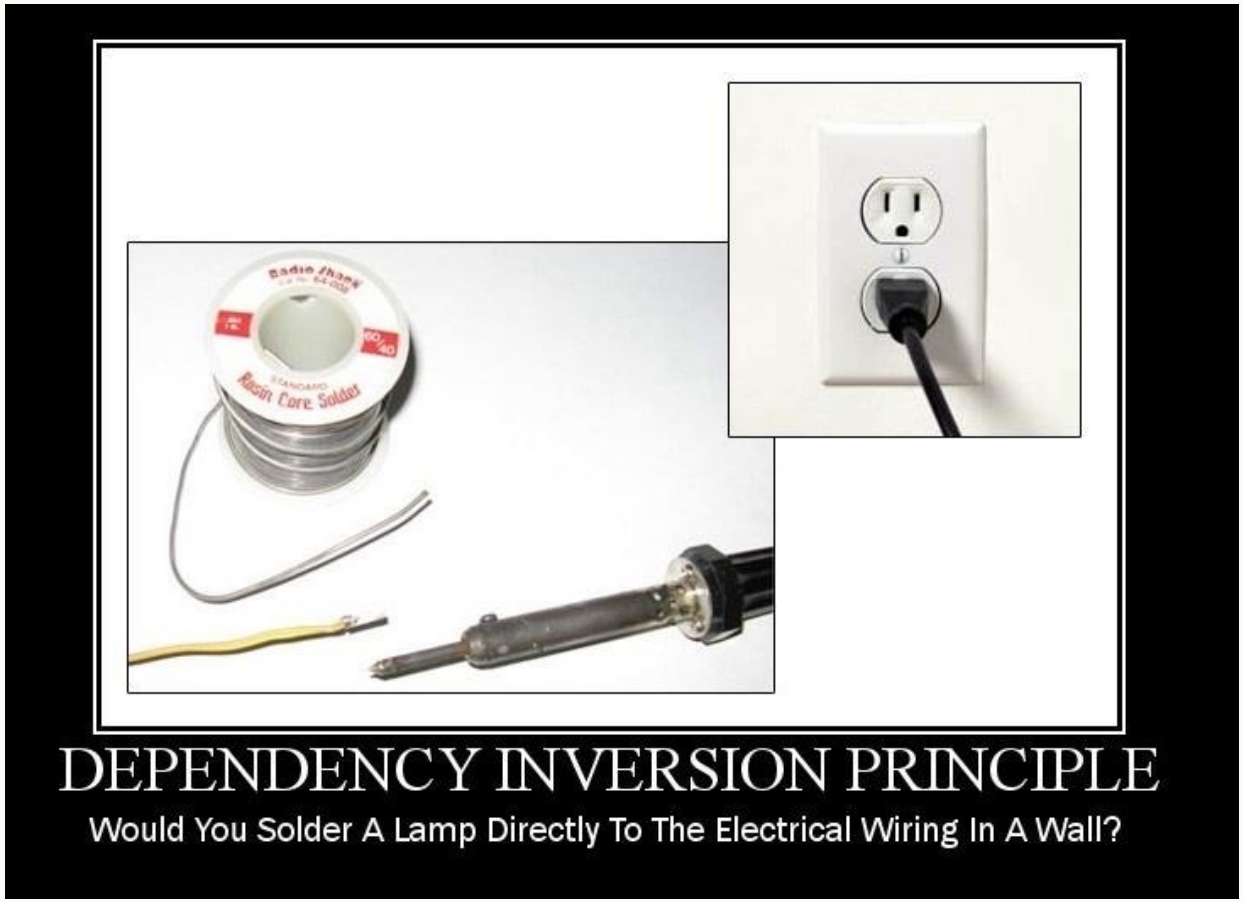
  - A SeaGull would implement the IFlyingBird interface
  - An Penguin would implement the IBird interface

Interface Segregation Principle

- As you can see, it does not make sense for a Bike class to implement the openDoors() method as a bike does not have any doors!

```
public interface Vehicle {
    public void drive();
    public void stop();
    public void refuel();
    public void openDoors();
}
public class Bike implements Vehicle {

    // Can be implemented
    public void drive() {...}
    public void stop() {...}
    public void refuel() {...}

    // Can not be implemented
    public void openDoors() {...}
}
```

Dependency Inversion Principle



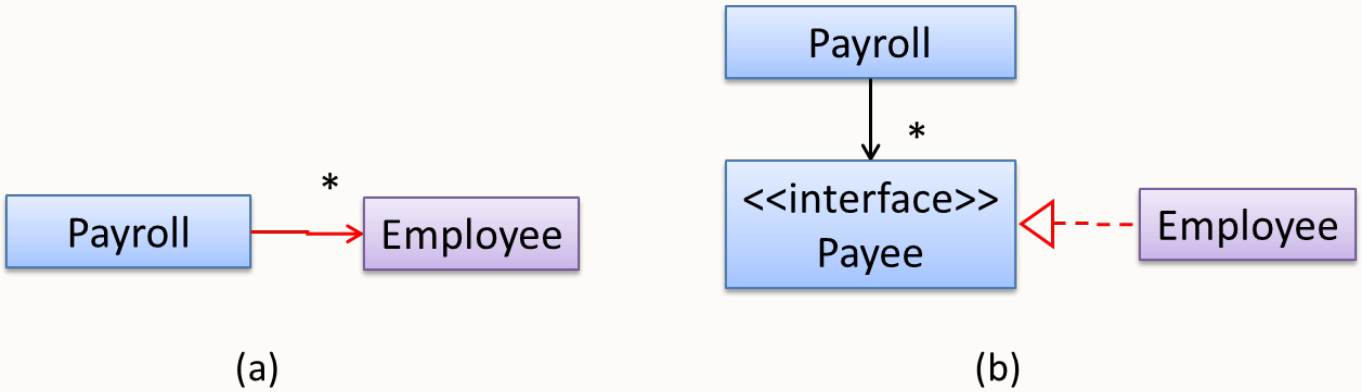
- UB says: "High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions."

Dependency Inversion Principle

- UB says "High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions."
  - Goal is to reduce coupling between different pieces of code by adding a layer of abstraction.
  - Make the higher-level modules depend on abstractions rather than concrete implementation of lower-level modules.
  - This for me is the key principle in SOLID otherwise maintainable code is not achievable.

Dependency inversion is a generalisation for the Open-Close and Liskov Substitution Principles

Dependency Inversion Principle



In design (a), the higher-level class Payroll depends on the lower-level class Employee, which is a violation of DIP. In design (b), both Payroll and Employee depend on the Payee interface (note that inheritance is a dependency).

Design (b) is more flexible (and less coupled) because now the Payroll class need not change
when the Employee class changes.

Dependency Inversion Principle

- Consider the example below. We have a Car class that depends on the concrete Engine class; therefore, it is not obeying DIP.
- The code will work, for now, but what if we wanted to add another engine type, let's say a diesel engine? This will require refactoring the Car class.

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class Engine {
   public void start() {...}
}
```

Dependency Inversion Principle

- Instead of Car depending directly on Engine, let's add an interface.
- Now we can connect any type of Engine that implements the Engine interface to the Car class.

```
public interface Engine {
    public void start();
}
```

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class PetrolEngine implements Engine {
   public void start() {...}
}
public class DieselEngine implements Engine {
   public void start() {...}
}
```

Brace Yourselves, Design Patterns Are Coming…

‘Gang of Four’Design Patterns

Promoting experience reuse

Design Patterns formalise and extend O-O principles

- What is a Design Pattern?
  - A pattern describes a problem which occurs over and over again and then describes the core of the solution to that problem in such a way that it can use the solution over and over again without ever doing it the same way again.
  - A pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.
  - A design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.
- A design pattern exists in code, not in your brain.

Design Patterns

A pattern specification has four essential elements:

1. Name: A handle that we can use to describe a design problem, its solution and the consequences in one or two words; having a vocabulary for patterns lets us talk about them with our colleagues and in our documentation.
2. Problem: Describes when to apply a pattern; sometimes the problem includes a list of conditions that must be met before it makes sense to apply a pattern.
3. Solution: Describes the elements that make up the design and their relationships.
4. Consequences: Describe the results and trade-offs (time and space) of applying the pattern; critical for evaluating design alternatives.

Design Patterns

- Design patterns are organised in two ways:
  - Purpose: Reflects what the pattern does
    - **Creational:** Concern the process of object creation.
    - **Structural:** Deal with the composition of classes and objects.
    - **Behavioural:** Characterise the way in which classes and objects interact and distribute responsibility.
  - Scope: Specifies whether the pattern applies to classes or objects
    - **Class:** These patterns deal with relationships between classes and sub-classes (which are fixed at compile time)
    - **Object:** Deal with object relationships (which can be changed at runtime)

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Just Enough Design Patterns To Get By

- How many design patterns do you really need to know?
  - The most useful should be known intimately;
  - The lesser used should be known well enough to be able to discuss them (even if implementation requires a quick refresher)
  - The remaining can be postponed until relevant (although not forgotten completely)

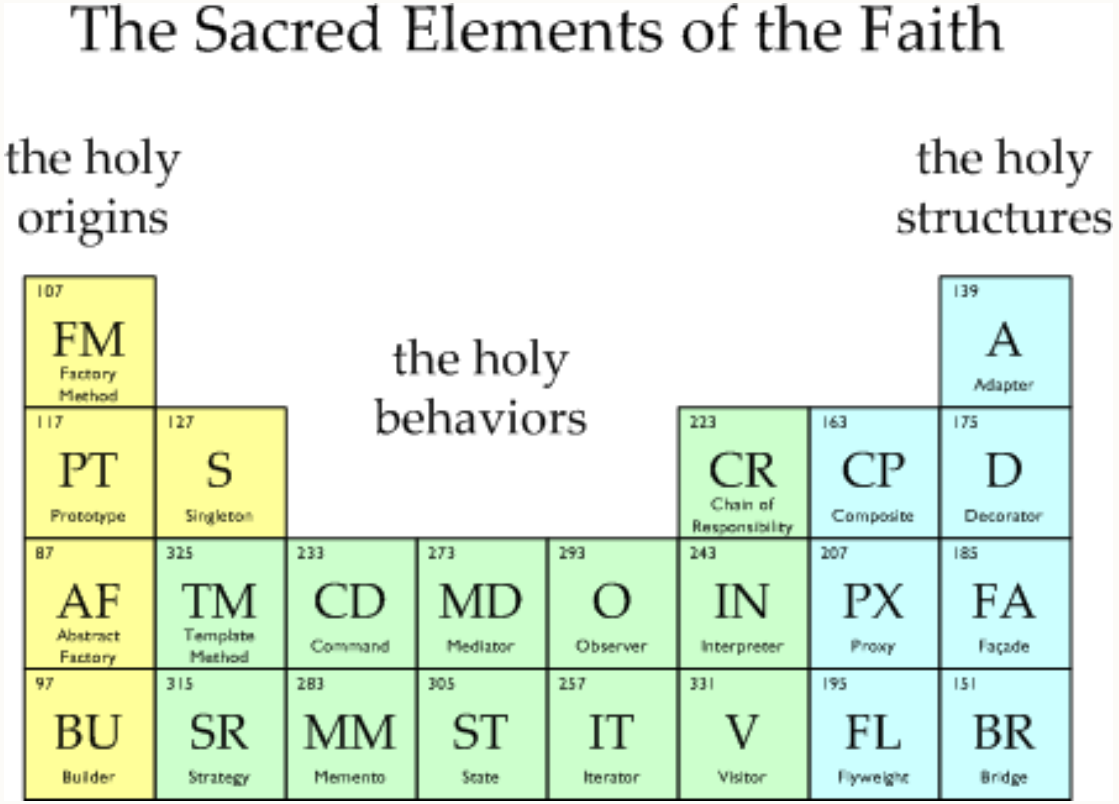
- Most useful design patterns
  - Adapter; Command; Decorator; Facade; Factory; Abstract Factory; Observer;
    Strategy
  - State (this is Peer's opinion – works on agent-based simulation)
  - Factory and Singleton (in Julie's opinion – works on sensor data in cybersecurity)

Design Pattern Examples

http://www.tutorialspoint.com/design_pattern

Design Pattern Categories

- The Gang of Four produced twenty-three patterns which are either:
  - Creational
  - Structural
  - Behavioural
- In this lecture we will look at some of the most commonly used patterns



Creational Patterns

Abstract factory: interface to create related objects without declaring the concrete class

Builder: a Builder class builds the final object step by step

Factory Method: defers instantiation to subclasses

Prototype: implementing a prototype interface which tells to create a clone of the current object

Singleton: ensures that one and only one instance of a class is created

Structural Patterns

Adapter: provides compatible interfaces for classes that couldn't work together otherwise

Bridge: involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes

Composite: put objects into a tree structure to represent the hierarchies

Decorator: add new functionality to an existing object without altering its structure

Facade: involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes

Flyweight: use sharing to support large numbers of complex objects

Proxy: provide a placeholder for another object to access its controls

Behavioural Patterns - 1

Chain of responsibility: give more than one object the chance to handle a request

Command: encapsulate a request as an object

Interpreter: convert problems expressed in natural language into a representation

Iterator: access to objects without exposing underlying representation

Mediator: promotes loose coupling by preventing objects from referring to each other

Behavioural Patterns -2

Memento: Capture and display an object's internal state

Observer: define a one to many relationship so that when one object changes state, all its dependents are notified

State: allow an object to alter its behaviour when its internal state changes.

Strategy: Define and encapsulate a family of algorithms. Let the algorithm vary independently of who is using it

Template Method: Define the skeleton of an algorithm, and let sub classes redefine certain steps without changing the structure of the algorithm

Visitor: define a new operation without changing the classes

The Singleton

"Create a one of a kind object for which there is only one instance"

Wow, this sounds really simple?

- Has a single class in its class diagram
  - *BUT how do we ensure that there is **only** one instance??*
- To ensure a class only has one instance, and provide a global point of access to it
  - By using static methods and private classes
- Sometimes we want just a single instance of a class to exist in the system.
  - we want just one window manager or print spooler
- We need to have that one instance easily accessible, **and** we want to ensure that additional instances of the class can not be created.

- Example : https://www.youtube.com/watch?v=8vOLNHMjgXM

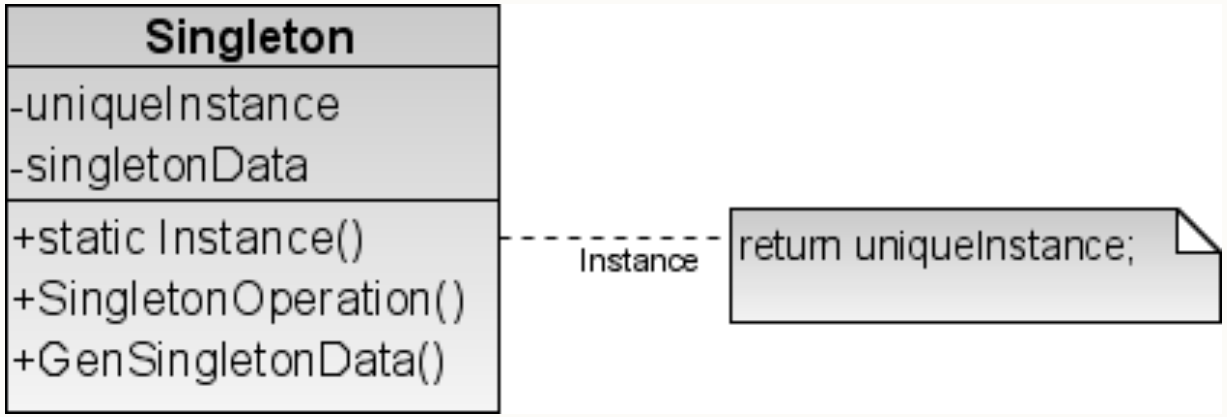
```
public class Singleton {
        private static Singleton uniqueInstance;

        private Singleton ( ) { }

        public static Singleton getInstance( ) {
                if ( uniqueInstance == null){
                        uniqueInstance = new Singleton( );
                }
                return uniqueInstance;
        }
}
```

Static variable to
hold single instance

Private
Constructor

Instantiate class and
return a single instance of it

Benefits of the Singleton



- Controlled access to sole instance
- Reduced namespace
- Can easily transform into concrete factory by permitting a variable number of instances

Singleton has some issues though

- There are even more things to consider around how to ensure that there is only a single instance.
  - concurrent requests
  - threading
  - might not have all the information needed at the point of static initialisation
  - interfacing with subclasses

The Factory Patterns

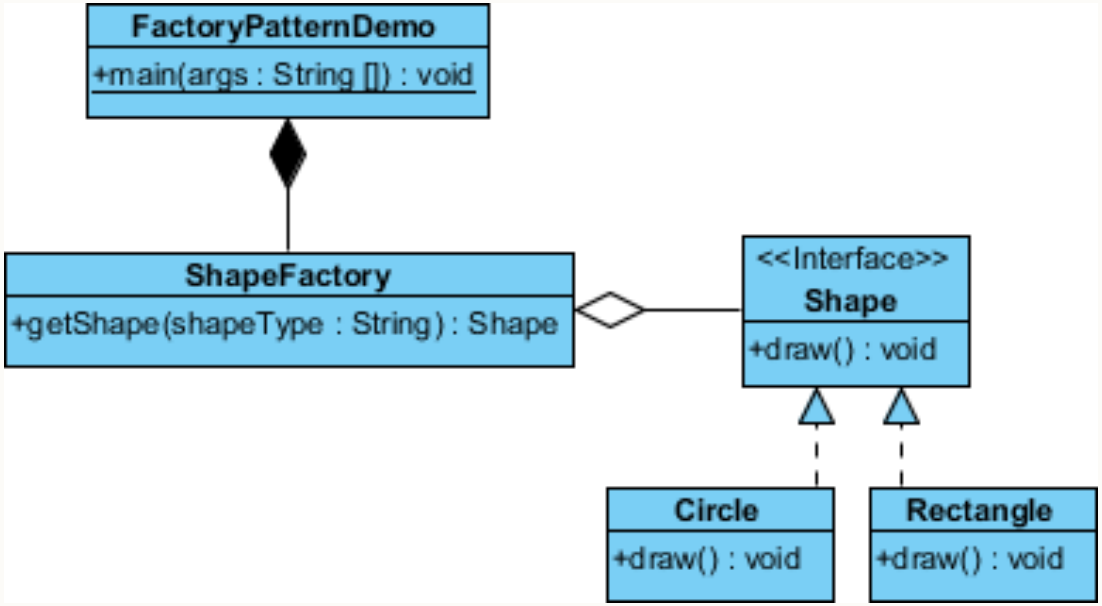
With the Factory Method pattern we create objects without exposing the creation logic to the client and refer to newly created object using a common interface.

About the Factories- The Virtual Constructor

- Creational patterns which abstract the object instantiation process.
- They hide how objects are created and help make the overall system independent of how its objects are created.
- Class creational patterns focus on the use of inheritance to decide the object to be instantiated in the Factory pattern.
- The factory method pattern can save you from those awkward dependencies in OOD by letting you define an interface for creating an object, but let its related subclasses decide which classes to instantiate.
- Used when the class does not know precisely which class of objects it must create.

Factory Method Pattern Example

- We want to create different types of Shape objects



```java
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

   }
}
```



```java
public interface Shape {
   void draw();
}
```

```java
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

```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();

      }

      return null;
   }
}
```

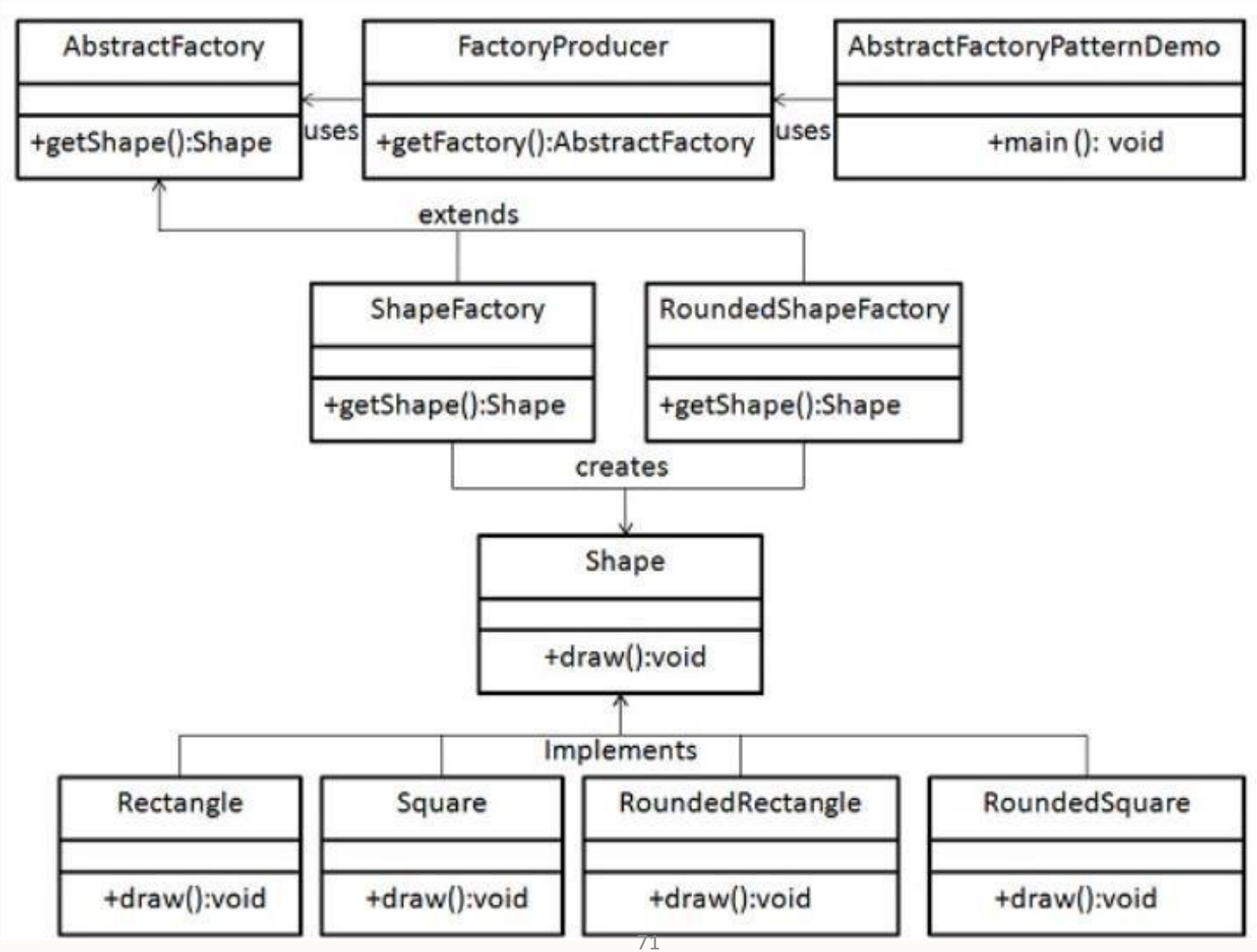
Abstract Factories

What we just discussed were concrete factories

Go one step further with the Abstract Factory

An Abstract Factory is a Factory of Factories

Used when you need an interface for creating related objects without specifying their classes


AbstractFactory
+getShape():Shape
uses
FactoryProducer
+getFactory():AbstractFactory
uses
AbstractFactoryPatternDemo
+main (): void
extends
ShapeFactory
+getShape():Shape
RoundedShapeFactory
+getShape():Shape
creates
Shape
+draw():void
Implements
Rectangle
+draw():void
Square
+draw():void
RoundedRectangle
+draw():void
RoundedSquare
+draw():void

Things to note…

- The AbstractFactory class determines the actual type of the concrete object and creates it.
- But it returns an abstract pointer to the concrete object just created.
- This stops the client from knowing anything about the actual creation of the
  object.
- The objects of the concrete type, created by the factory, are accessed by the client only through the abstract interface.
- To add new concrete types, modify the client code and make it use a different factory.

The Infamous Pizza Factory Example

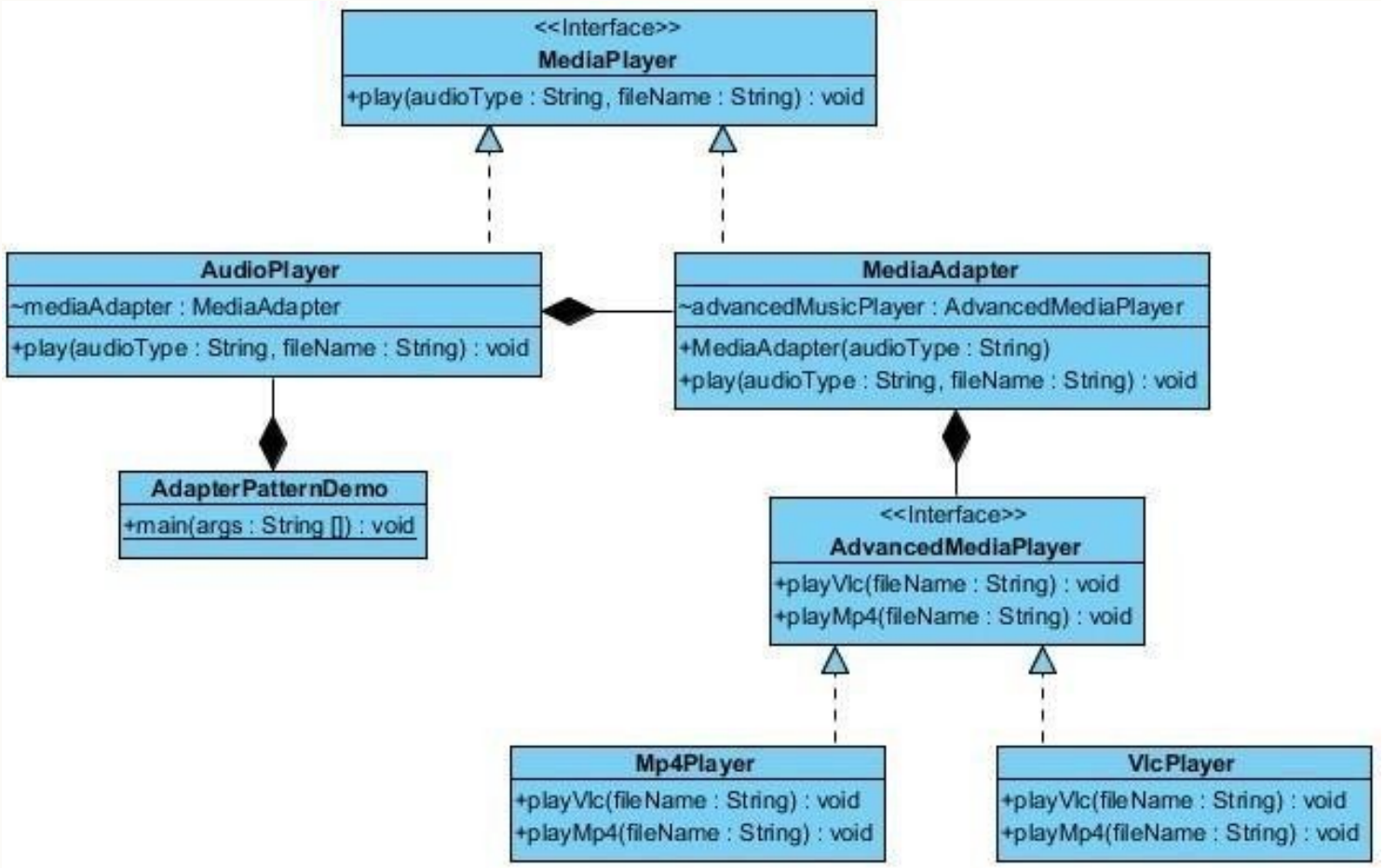
- The Pizza Factory which defines method names and returns types to make different kinds of pizza.
- The abstract factory can be named AbstractPizzaFactory.
- RomeConcretePizzaFactory and MilanConcretePizzaFactory being two extensions of the abstract class.
- The abstract factory will define types of toppings for pizza, like pepperoni, sausage or anchovy.
- The concrete factories will implement only a set of the toppings, which are specific for the area.

- More info: https://www.youtube.com/watch?v=pt1IbV1aSZ4

Adapter Pattern (Structural)

- This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.
  - **Class adapter**: Relies on multiple inheritance
  - **Object adapter**: Relies on object composition >> we use this one!

Adapter Pattern Example



- An audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files
- More info: https://www.youtube.com/watch?v=5-xqFjo_jC8

Observer Pattern

- Observer pattern is used when there is one- to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

Based on the model of old style newspaper subscriptions

- A newspaper publisher goes into business and begins publishing newspapers.
- You subscribe to a particular publisher and every time there is a new edition, it gets delivered to you for as long as you subscribe.
- You unsubscribe when you don't want any more
  newspapers, delivery also stops.
- While the publisher remains in business, other entities such as other people, hospitals, planes etc subscribe and unsubscribe.

Simple Relationships in the Observer

Publishers + Subscribers = Observer

Publishers = SUBJECT

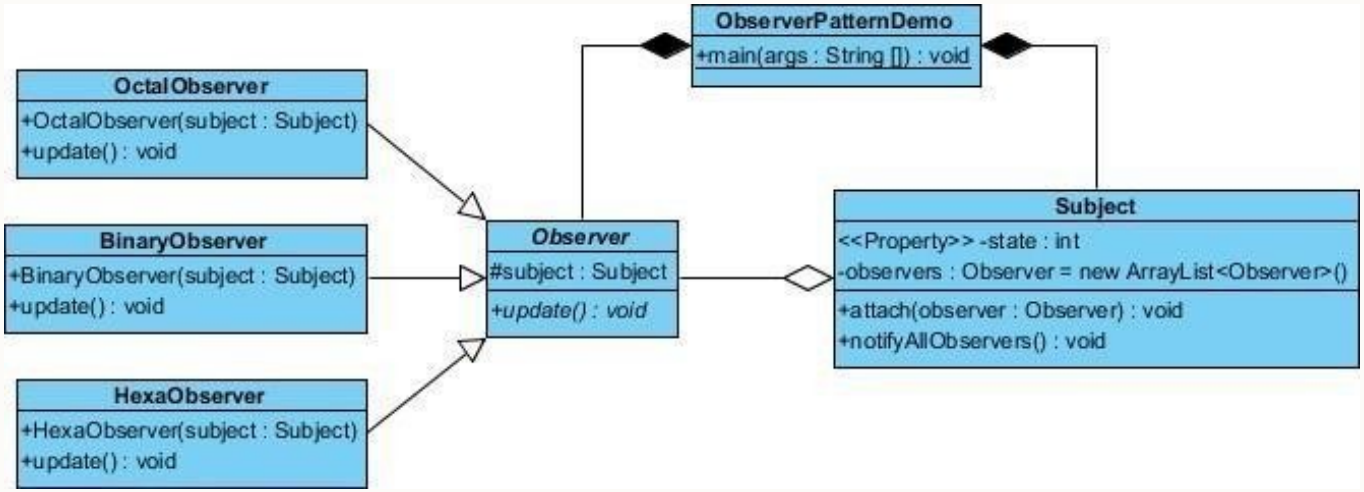
Subscribers = OBSERVERS

Loose Coupling

- Two objects are loosely coupled if they interact but have very little knowledge of each other.
- It does not need to know the class of the observer.
- We can add new observers at any time irrespective of the state of the subject object and without modifying the subject object.
- Changes to either subject or object will not affect the other.

Observer Pattern Example

- Number Converter for Programmers



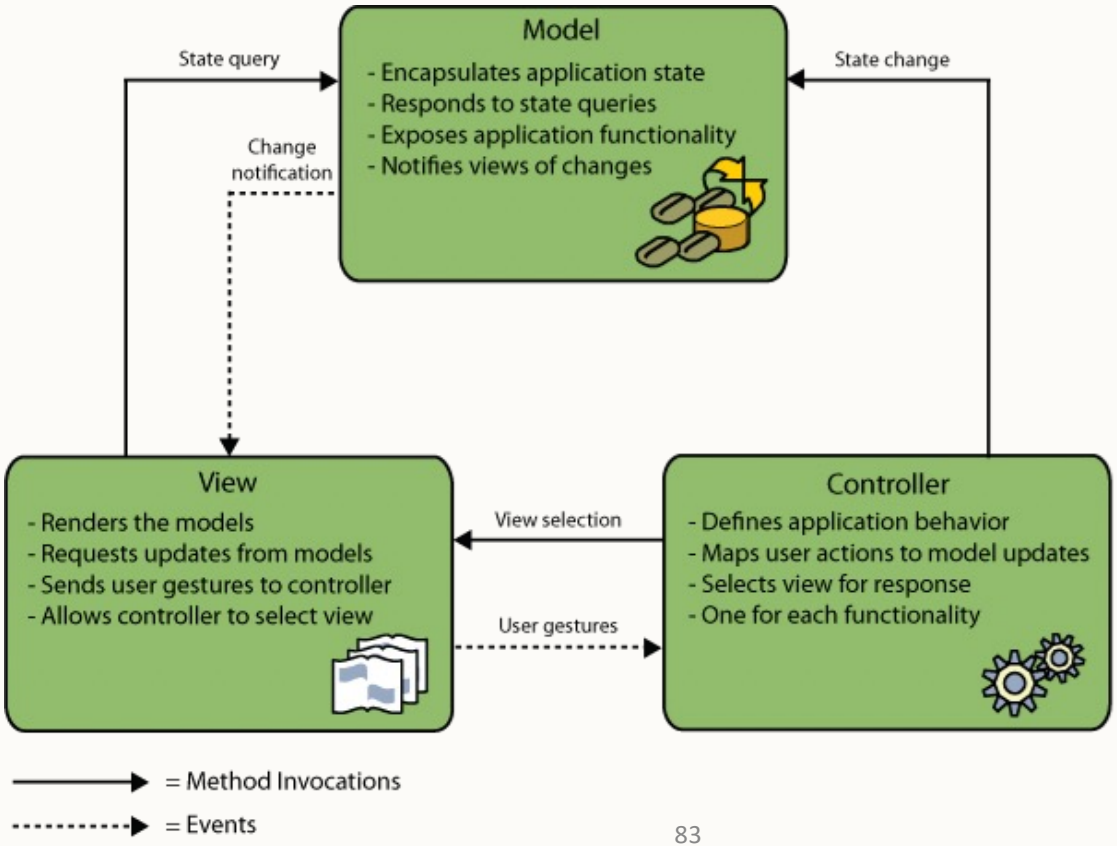
Weather Station Example

- If we have a IoT box connected to multiple weather stations, how can we set up the observer pattern and make the sensors observable in order to efficiently construct an OO version of this software?
- The weather station will be based on a WeatherData object which tracks current weather conditions including temp, humidity and pressure.
- Create an application which has three display elements:
  - current conditions, weather stats and simple forecast.
- The displays must be updated in real time.
- Have it expandable so that an API can be used to create new displays.

Conveniently Java Loves The Observer

- Inbuilt into the java.util and in JDK, in JavaBeans and Swing.
- Jbutton - addActionListener method is the subject's register observer method.
- Smalltalk **Model View Controller** user interface framework.

Oooh and remember MVC?



Good Software Maintenance Avoids 'Antipatterns'

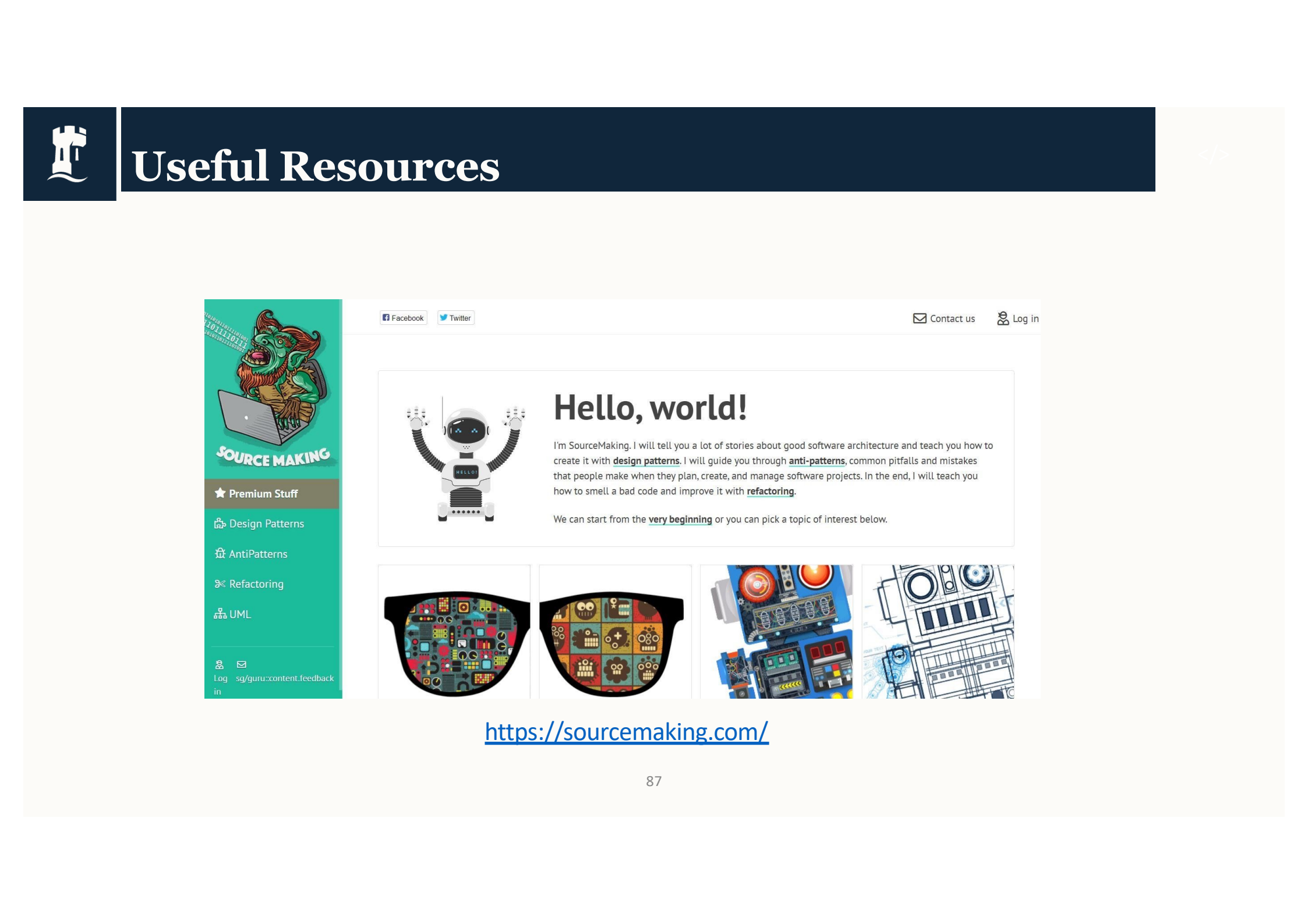
- Experience re-use to avoid common problems in software maintenance.
- Notable examples include:
  - *The Blob:* giving one object the greatest share of the responsibility.
  - *Lava Flow:* where dead or obsolete code is deeply ingrained in the functionality.
  - *Poltergeists:* classes with highly limited roles.
  - *Dead End:* achieved by modifying a reusable component when that component is no longer supported by the vendor.
  - *Spaghetti Code:* code developed in an ad hoc manner which is never refactored.
  - *Cut and Paste programming:* overuse of stackoverflow etc, see last week's every programming tutorial video ever.
- Still confused? Try this: https://sourcemaking.com/antipatterns/software-development-antipatterns

Summary

- Recap of the philosophy underlying object orientation.
- SOLID Principles to implement OO in a maintainable manner.
- The standard Design Patterns
  - Overview
  - Singleton
  - Factory(s)
  - Adapter
  - Observer/observable
- Look at practical implementations of design patterns.

Additional Resources

http://www.tutorialspoint.com/design_pattern



Useful Resources

https://sourcemaking.com/

Acknowledgements

We thank Julie Greensmith and Dan Lipsa for lecture material.

More information can be found in:

- Hans van Vliet**, Software Engineering: Principles and Practice,** 3rd Edition, 2008, John Wiley & Sons
- FreeTechBooks.com

Questions?