

# COMP2511

## The Art of Software Design (OO Design & Programming)

### Course Introduction

Term 1, 2024

# Our Team

## Lecturer-in-charge:

- Dr Ashesh Mahidadia <a.mahidadia@unsw.edu.au>



Ashesh



Alvin



Sai

## Course Admin Team:

- Alvin Cherk
- Sai Nair
- Carl Buchanan
- Amanda Lu

## Tutors:

- 23 passionate tutors!

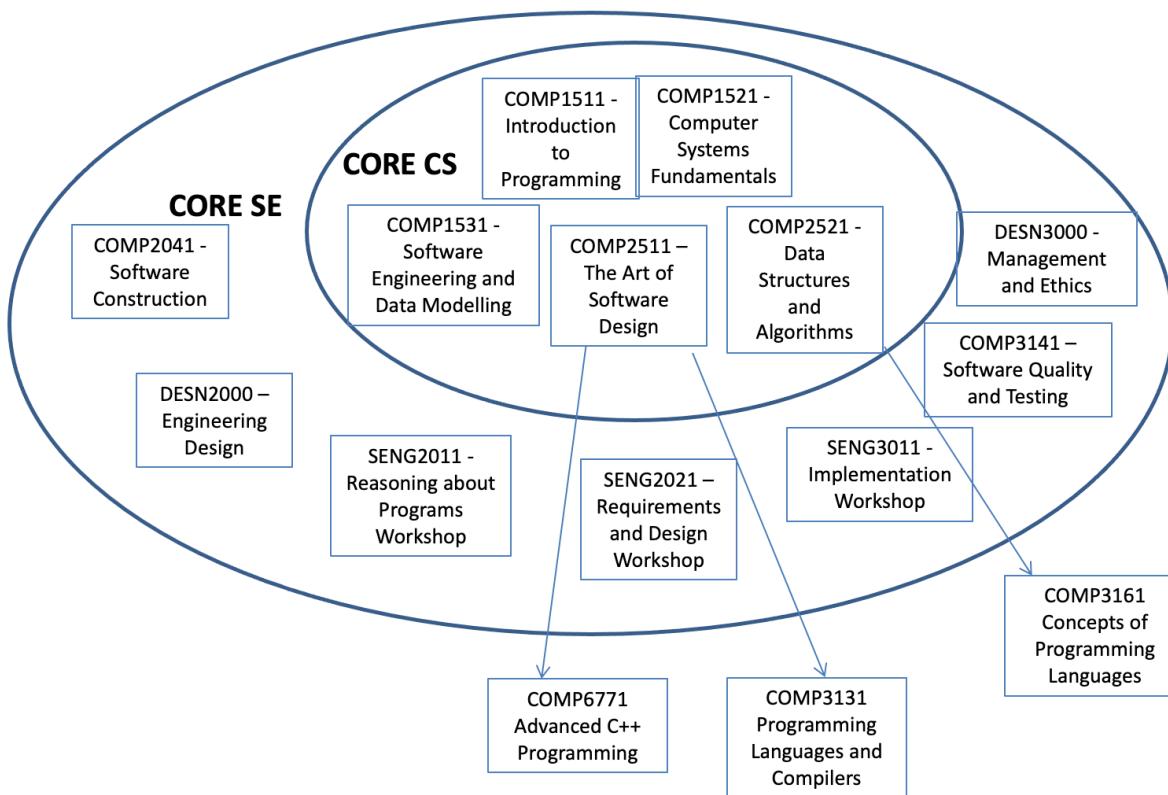
Course Account Email: **cs2511@cse.unsw.edu.au**

*(Unless you specifically require to contact a member of the admin team,  
please use the **above email** for any queries related to the course.)*

Class Web: <http://webcms3.cse.unsw.edu.au/COMP2511/24T1/>

COMP2511: Course Introduction

# Course Context



# The Story So Far: Course Context

- COMP1511: Solving problems with computers, the wonder and joy of programming
- COMP1521: Getting right down into the silicon
- COMP1531: Solving problems in a team; programming in the large
- COMP2521: Solving problems at scale using data structures and algorithms
- **COMP2511???**

# COMP2511

- We can write code, but how do we write good code?
- **Designing elegant and beautiful software.**
- Shades of Grey - things aren't clear cut; writing good software is an art
- From programmers to designers.



# COMP 2511 Major Themes

- Develop an appreciation for elegantly written software, and how to create and maintain well-designed systems;
  - Apply principles and patterns to effectively design **flexible, maintainable and reusable systems**
- Understand different design paradigms and methodologies, their background and application;
  - Object-Oriented Paradigm
  - Functional Paradigm
  - Concurrent Paradigm (introduction)

# COMP 2511 Major Themes

- Understand and apply the principles of Object-Oriented Design to solve problems;
  - Be able to follow a **systematic** OO Design process
  - Be able to interpret and use tools for OO Design
- Understand the role of and apply widely used Design Patterns to create extensible designs
  - Behavioural patterns
  - Structural patterns
  - Creational patterns
  - Programming patterns (exceptions, generic programming)
  - Testing patterns

# COMP 2511 Major Themes

- Develop skills in both creating medium-scale systems from scratch, and working on existing systems as part of the Software Development Life Cycle;
  - Be able to analyse, refactor and work with code started by someone else
  - Create medium-scale systems using Java
- Work with an enterprise programming language and IDE
  - Java language
  - VSCode IDE

# Credit teaching material

- ❖ No text book, the lecture slides cover the required topics.
- ❖ However, you are strongly encouraged to read additional material and the reference books.
- ❖ In the lecture notes, some content and ideas are drawn from:
  - *Head First Design Patterns* , by Elisabeth Freeman and Kathy Sierra, The State University of New Jersey
  - *Refactoring: Improving the design of existing code* , by Martin Fowler
  - Material from many popular websites.

# How do we obtain our educational objectives?

❖ **Lectures:** 4 hour lectures (9 weeks)

❖ **Tutorials:**

- ❖ A 1 hour tutorial session per week, which is scheduled before the lab.
- ❖ Online Tutorials/Labs will be run via **MS Teams**.
- ❖ Tutorials are understanding-driven - interactive examples to illustrate concepts discussed in lectures
- ❖ Solutions and recording to tutorials posted at the end of each week

# How do we obtain our educational objectives?

## ❖ Labs:

- ❖ 2 hours each week, straight after tutorial
- ❖ Similar to most CSE core courses
- ❖ Lab retros posted at after due date on Confluence
- ❖ Online Run via MS Teams

The above are subject to change, if required.

# Assessments

# Coursework (15%)

- ❖ Your coursework mark is made up of marks associated with the lab exercises.
- ❖ There are **seven** labs, each worth **ten** marks.
- ❖ We will cap total coursework marks at 60 (which will translate to 15%), leaving **one lab as a buffer**.
- ❖ If you attend all seven labs, we will add all seven lab marks and cap the total coursework marks to 60.
- ❖ The specific marking criteria for each lab will be outlined in the respective specifications.
- ❖ The table below offers as a general guide for the criteria that your tutor/lab assistant will use to assess you.
  - [A general guide for lab assessments](#)

# Assignment I (15%)

- ❖ The marking criteria for the assignment will be outlined in the specification which will be released Tuesday of Week 2.
- ❖ Due Friday 5pm Week 5.
- ❖ Completed **individually**.

## Assignment II (20 %)

- ❖ The marking criteria for the project will be outlined in the specification which will be released Thursday Week 5.
- ❖ **Pairs** formed within your tutorial.
- ❖ Groups formed by end of Week 3.
- ❖ **Due Friday 5pm week 9**
- ❖ Measurers in place for difficult partners (Keep your tutor informed)

## Assignment III (8 % Bonus)

- ❖ A more challenging real-world problem that incorporates Design Principles and Patterns discussed in the course.
- ❖ For students that wish to extend themselves and score highly in the course
- ❖ Can be completed in a pair or individually
- ❖ Assignment spec released **Tuesday week 8**
- ❖ Due **Sunday 5pm of week 10**

## Final Exam (50%)

- ❖ In 24T1 the COMP2511 exam will be **held in person in the CSE Labs, and invigilated.**
- ❖ **All** the **students** are required to take **the final exam in person**, even if they have enrolled in online classes. In 23T3, there will be no online exams.
- ❖ **Hurdle** : From Term 3 2023, in order to pass the course, it is required for the student to achieve a **minimum of 40%** (20 out of 50) marks **in the final examination.**
- ❖ Students are eligible for a Supplementary Exam if and only if:
  - Students cannot attend the final exam due to illness or misadventure.  
Students must formally apply for a special consideration, and it must be approved by the respective authority.

# Assumed Knowledge

- ❖ Confident programmers
  - Familiar with C and Python programming concepts
- ❖ Able to work in a team
  - Git
  - Working with others
- ❖ Understand basic testing principles
- ❖ Understand basic software engineering design principles (DRY, KISS)

# Assumed Knowledge

- ❖ What we don't assume:
  - Knowledge of Java
  - Understanding of Object-Oriented Programming
- ❖ **This is not a Java course**

# Course philosophy

- ❖ A step up from first year courses
- ❖ Challenging but achievable
- ❖ Develop skills in time management, teamwork as well as critical thinking
- ❖ Highly rewarding

# Support

- ❖ Supporting you is our job :)
- ❖ Help Sessions
  - Lots of them with fantastic tutors
  - Feedback on work, help with problems, clarifying ideas
  - You are expected to have done your own research and debugging before arriving

# Support

- ❖ Course Forum (Ed)
  - Ask questions and everyone can see the answers!
  - Make private posts for sharing code
  - Response time
- ❖ Course Account - cs2511@cse.unsw.edu.au
  - Sensitive/personal information
- ❖ During the project - your tutor

# Support

- ❖ Go to help sessions for help on concepts
- ❖ Post on the forum if you need more immediate lab feedback
- ❖ There are no late extensions on labs unless in extenuating circumstances - email [cs2511@cse.unsw.edu.au](mailto:cs2511@cse.unsw.edu.au)

# Support - UNSW

- ❖ **Special Consideration** -  
<https://student.unsw.edu.au/special-consideration>
- ❖ **Equitable Learning Services** -  
<https://student.unsw.edu.au/els>

# Mental Health & Wellbeing

- ❖ UNSW Psychology & Wellness - <https://student.unsw.edu.au/mhc>
- ❖ UNSW Student Advisors - <https://student.unsw.edu.au/advisors>
- ❖ Reach out to us at [cs2511@cse.unsw.edu.au](mailto:cs2511@cse.unsw.edu.au)
- ❖ Check in with each other
- ❖ Talk to someone

# Technology Stack

- ❖ Java Version – JDK 17
- ❖ VSCode
- ❖ Gradle 8.5
- ❖ Gitlab (+ CI pipelines)

# Feedback

- ❖ We love feedback :)
- ❖ Changes made to the course this term based on constructive student feedback
- ❖ We always want to continuously improve
  - In response to the previous term's feedback, we introduced a sample exam in the Week 10 lab and developed a framework to promote greater engagement during tutorials and laboratories.
- ❖ Feedback form
- ❖ Course account

# Respect

- ❖ Yourselves, each other, course staff

It's time to lift off for 24T1!!!!



**COMP2511**

# **Object Oriented Programming (OOP) in Java**

Prepared by

Dr. Ashesh Mahidadia

# OOP in Java

- ❖ Object Oriented Programming (OOP)
- ❖ Inheritance in OOP
- ❖ Introduction to Classes and Objects
- ❖ Subclasses and Inheritance
- ❖ Abstract Classes
- ❖ Single Inheritance versus Multiple Inheritance
- ❖ Interfaces
- ❖ Method Forwarding (Has-a relationship)
- ❖ Method Overriding (Polymorphism)
- ❖ Method Overloading
- ❖ Constructors

# Object Oriented Programming (OOP)

In procedural programming languages (like ‘C’), programming tends to be **action-oriented**, whereas in Java - programming is **object-oriented**.

In **procedural** programming,

- groups of actions that perform some task are formed into functions and functions are grouped to form programs.

In **OOP**,

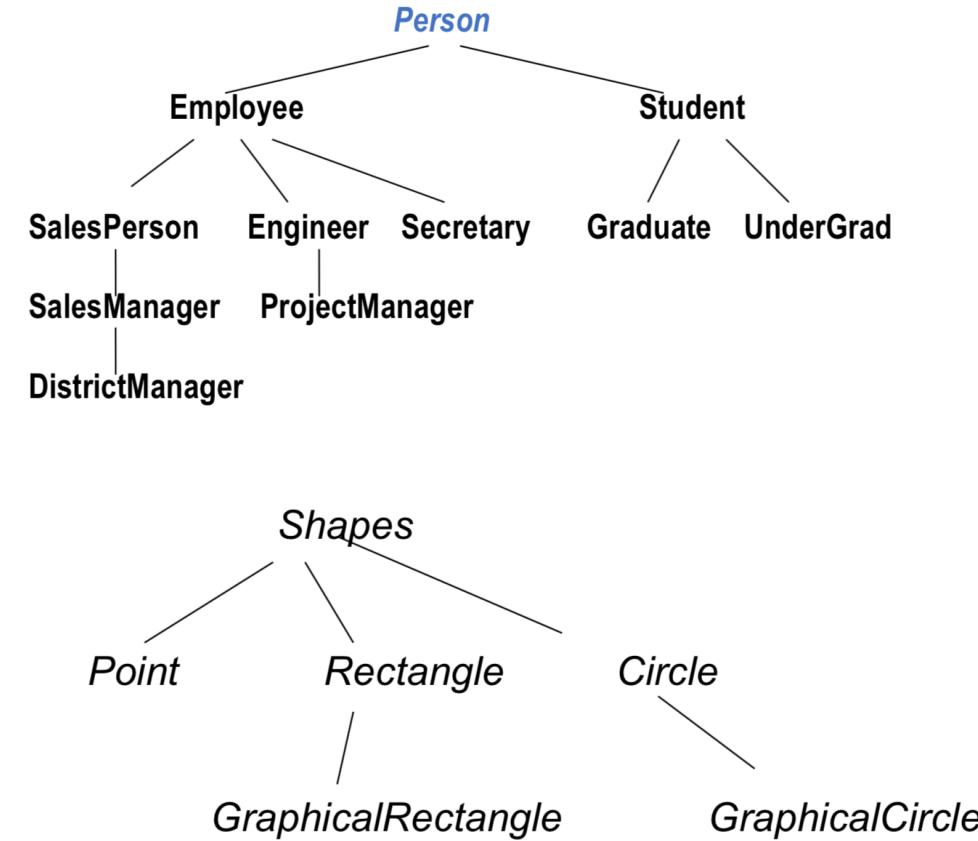
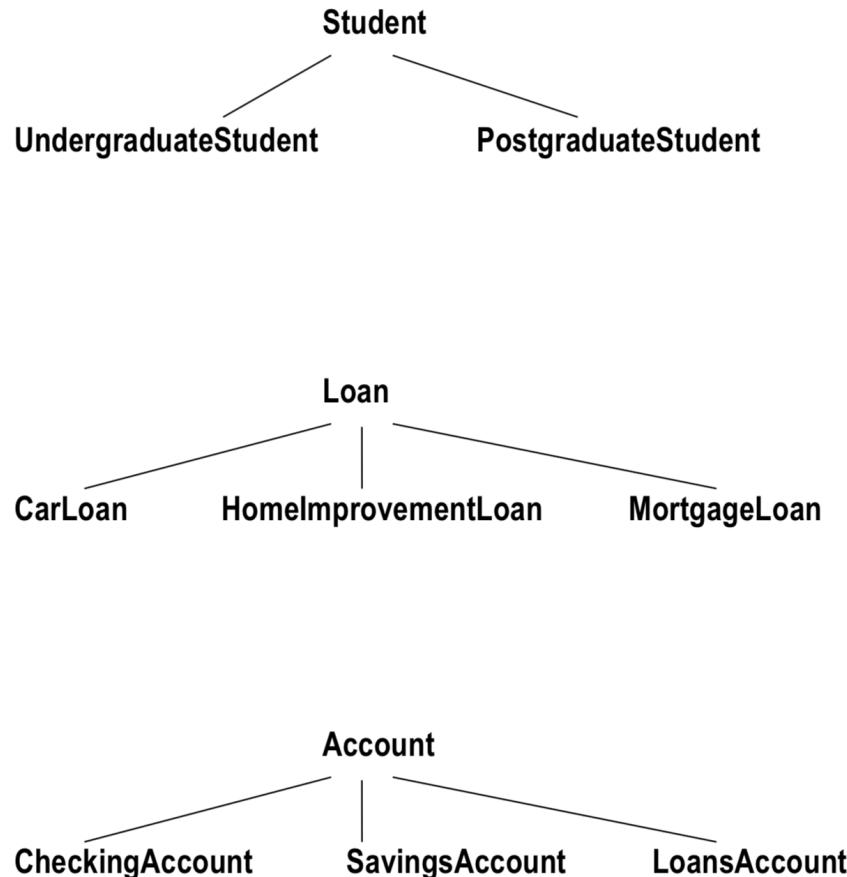
- programmers concentrate on creating their own user-defined types called **classes**.
- each **class** contains **data** as well as the set of **methods** (procedures) that manipulate the data.
- an instance of a user-defined type (i.e. a **class**) is called an **object**.
- OOP **encapsulates** data (attributes) and methods (behaviours) into **objects**, the data and methods of an object are intimately tied together.
- Objects have the property of **information hiding**.

# Inheritance in Object Oriented Programming (OOP)

- ❖ **Inheritance** is a form of software reusability in which new classes are created from the existing classes by absorbing their attributes and behaviours.
- ❖ Instead of defining completely (separate) new class, the programmer can designate that the new class is to **inherit** attributes and behaviours of the existing class (called **superclass**). The new class is referred to as **subclass**.
- ❖ Programmer can add **more attributes and behaviours** to the *subclass*, hence, normally **subclasses** have **more features than their super classes**.

# Inheritance in Object Oriented Programming (OOP)

Inheritance relationships form **tree-like hierarchical** structures. For example,



# “Is-a” - Inheritance relationship

- ❖ In an “**is-a**” relationship, an object of a subclass may also be treated as an object of the superclass.
- ❖ For example, *UndergraduateStudent* can be treated as *Student* too.
- ❖ You should use *inheritance* to model “is-a” relationship.

## Very Important:

- ❖ Don’t use inheritance unless **all or most** inherited attributes and methods **make sense**.
- ❖ For example, mathematically a *circle* is-a (an) *oval*, however you should **not** inherit a class *circle* from a class *oval*. A class *oval* can have one method to set *width* and another to set *height*.

# “Has-a” - Association relationship

- ❖ In a “has-a” relationship, a **class object has an object of another class** to store its state or do its work, i.e. it “has-a” reference to that other object.
- ❖ For example, a Rectangle Is-NOT-a Line.  
However, we may use a Line to draw a Rectangle.
- ❖ The “has-a” relationship is quite different from an “is-a” relationship.
- ❖ “Has-a” relationships are examples of creating new classes by *composition* of existing classes (as oppose to *extending* classes).

## Very Important:

- ❖ Getting “Is-a” versus “Has-a” relationships correct is both **subtle** and potentially **critical**. You should **consider** all **possible** future **usages** of the classes before finalising the hierarchy.
- ❖ It is possible that **obvious solutions may not work** for some applications.

# Designing a Class

- Think carefully about the functionality (methods) a class should offer.
- Always **try to keep data private** (local).
- Consider **different ways** an object may be **created**.
- Creating an object may require different actions such as initializations.
- Always initialize data.
- If the object is no longer in use, free up all the associated resources.
- **Break up classes with too many responsibilities.**
- In OO, classes are often closely related. “**Factor out**” common attributes and behaviours and place these in a class. Then use suitable relationships between classes (for example, “is-a” or “has-a”).

# Introduction to Classes and Objects

- ❖ A class is a collection of **data** and **methods** (procedures) that operate on that data.
- ❖ For example,  
a **circle** can be described by the **x, y** position of its centre and by its **radius**.
- ❖ We can define some useful methods (procedures) for circles,  
compute **circumference**, compute area, check whether pointes are inside the circle,  
etc.
- ❖ By defining the **Circle class** (as below), we can create a **new data type**.

# The class Circle

- ❖ For simplicity, the methods for *getter* and *setters* are not shown in the code.

```
public class Circle {  
  
    protected static final double pi = 3.14159;  
    protected int x, y;  
    protected int r;  
  
    // Very simple constructor  
    public Circle(){  
        this.x = 1;  
        this.y = 1;  
        this.r = 1;  
    }  
    // Another simple constructor  
    public Circle(int x, int y, int r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    /**  
     * Below, methods that return the circumference  
     * area of the circle  
     */  
    public double circumference( ) {  
        return 2 * pi * r ;  
    }  
    public double area ( ) {  
        return pi * r * r ;  
    }  
}
```

# Objects are Instances of a class

In Java, objects are created by instantiating a class.

For example,

```
Circle c ;  
c = new Circle ( ) ;
```

OR

```
Circle c = new Circle ( ) ;
```

# Accessing Object Data

We can access data fields of an object.

For example,

```
Circle c = new Circle();  
  
// Initialize our circle to have centre (2, 5)  
// and radius 1.  
// Assuming, x, y and r are not private  
  
c.x = 2;  
c.y = 5;  
c.r = 1;
```

# Using Object Methods

To access the methods of an object, we can use the same syntax as accessing the data of an object:

```
Circle c = new Circle( );
double a;

c.r = 2;           // assuming r is not private

a = c.area();

// Note that its not :    a = area(c);
```

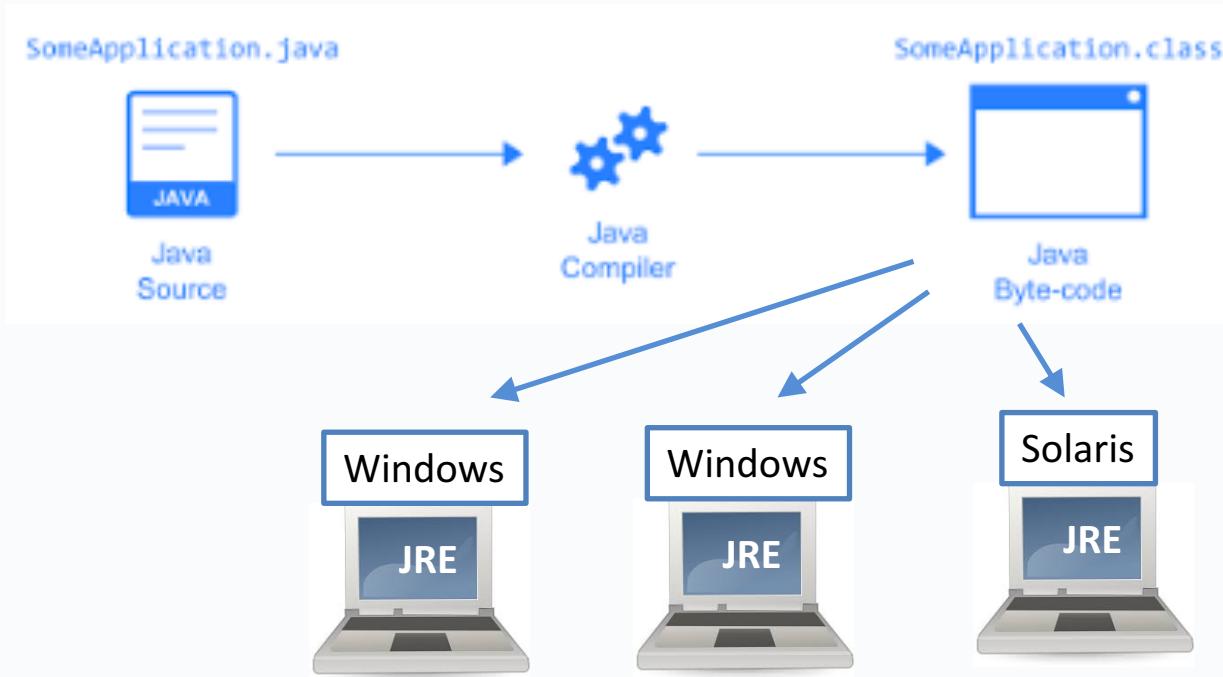
**COMP2511**

# **Object Oriented Programming (OOP) in Java**

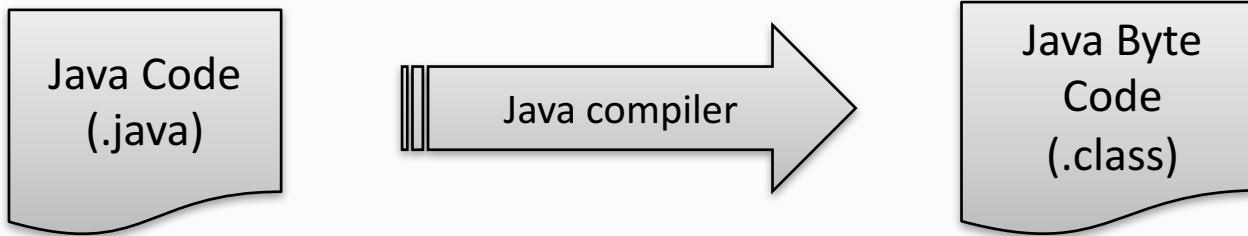
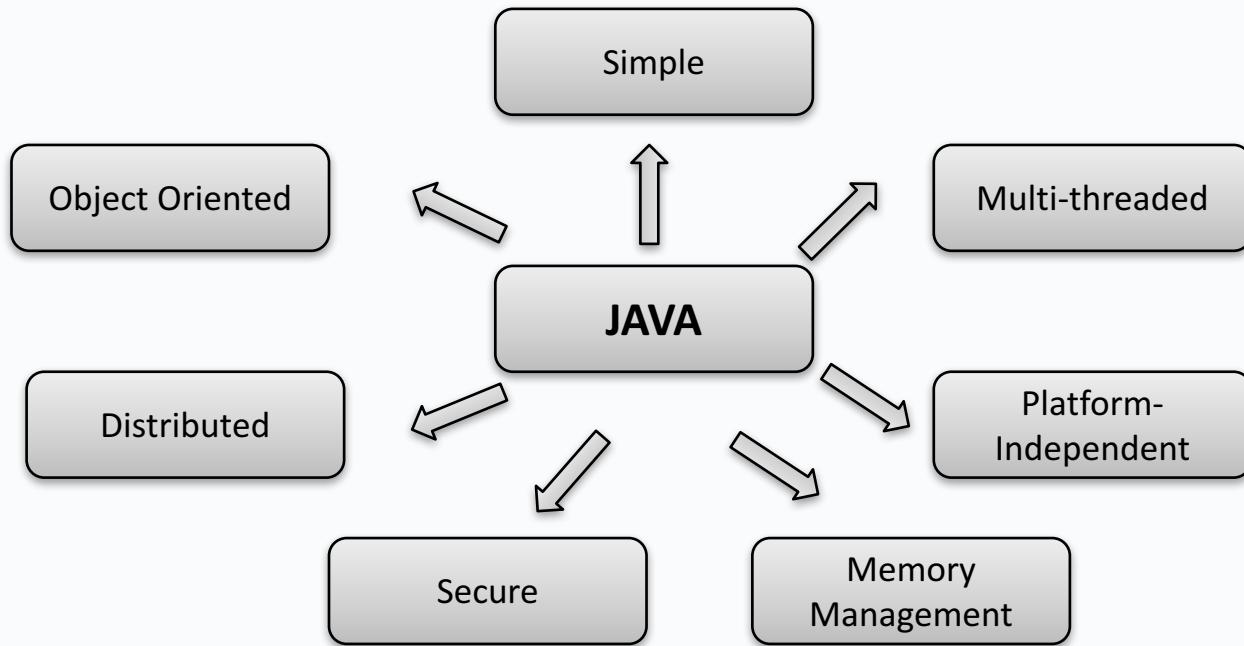
Prepared by

Dr. Ashesh Mahidadia

# Java is Platform Independent



# The Java Platform



**COMP2511**

# **Object Oriented Programming (OOP) in Java**

Prepared by

Dr. Ashesh Mahidadia

# OOP in Java

- ❖ Object Oriented Programming (OOP)
- ❖ Inheritance in OOP
- ❖ Introduction to Classes and Objects
- ❖ Subclasses and Inheritance
- ❖ Abstract Classes
- ❖ Single Inheritance versus Multiple Inheritance
- ❖ Interfaces
- ❖ Method Forwarding (Has-a relationship)
- ❖ Method Overriding (Polymorphism)
- ❖ Method Overloading
- ❖ Constructors

# Subclasses and Inheritance: *First Approach*

We want to implement *GraphicalCircle*.

This can be achieved in at least 3 different ways.

## First Approach:

- ❖ In this approach we are creating the new separate class for *GraphicalCircle* and re-writing the code already available in the class *Circle*.
- ❖ For example, we re-write the methods *area* and *circumference*.
- ❖ Hence, this approach is NOT elegant, in fact its the worst possible solution.  
Note again, its the worst possible solution!

```
// The class of graphical circles

public class GraphicalCircle {
    int x, y;
    int r;
    Color outline, fill;

    public double circumference() {
        return 2 * 3.14159 * r ;
    }
    public double area () {
        return 3.14159 * r * r ;
    }

    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }
}
```

# Subclasses and Inheritance: Second Approach

- ❖ We want to implement *GraphicalCircle* so that it can make use of the code in the class *Circle*.
- ❖ This approach uses “**has-a**” relationship.
- ❖ That means, a *GraphicalCircle* has a (mathematical) *Circle*.
- ❖ It uses methods from the class *Circle* (*area* and *circumference*) to define some of the new methods.
- ❖ This technique is also known as **method forwarding**.

```
public class GraphicalCircle2 {  
    // here's the math circle  
    Circle c;  
    // The new graphics variables go here  
    Color outline, fill;  
  
    // Very simple constructor  
    public GraphicalCircle2() {  
        c = new Circle();  
        this.outline = Color.black;  
        this.fill = Color.white;  
    }  
  
    // Another simple constructor  
    public GraphicalCircle2(int x, int y, int r,  
                           Color o, Color f) {  
        c = new Circle(x, y, r);  
        this.outline = o;  
        this.fill = f;  
    }  
  
    // draw method , using object 'c'  
    public void draw(Graphics g) {  
        g.setColor(outline);  
        g.drawOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);  
        g.setColor(fill);  
        g.fillOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);  
    }  
}
```

# Subclasses and Inheritance:

## Third Approach - Extending a Class

- ❖ We can say that *GraphicalCircle* **is-a** *Circle*.
- ❖ Hence, we can define *GraphicalCircle* as an **extension**, or *subclass* of Circle.
- ❖ The subclass *GraphicalCircle* **inherits** all the variables and methods of its superclass *Circle*.

```
import java.awt.Color;
import java.awt.Graphics;

public class GraphicalCircle extends Circle {

    Color outline, fill;
    public GraphicalCircle(){
        super();
        this.outline = Color.black;
        this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle(int x, int y,
                          int r, Color o, Color f){
        super(x, y, r);
        this.outline = o; this.fill = f;
    }

    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }
}
```

# Subclasses and Inheritance: Example

We can assign an instance of *GraphicCircle* to a *Circle* variable. For example,

```
GraphicCircle gc = new GraphicCircle();  
...  
double area = gc.area();  
...  
Circle c = gc;  
// we cannot call draw method for "c".
```

## Important:

- ❖ Considering the variable “c” is of type *Circle*,
- ❖ we can only access attributes and methods available in the class *Circle*.
- ❖ we **cannot** call *draw* method for “c”.

# Super classes, Objects, and the Class Hierarchy

- ❖ Every class has a superclass.
- ❖ If we don't define the superclass, by default, the superclass is the class **Object**.

## **Object** Class :

- ❖ Its the only class that does not have a superclass.
- ❖ The methods defined by **Object** can be called by any Java object (instance).
- ❖ Often we need to **override** the following methods:
  - **toString()**
    - read the API at [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString\(\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString())
  - **equals()**
    - read the API at  
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>equals\(java.lang.Object\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>equals(java.lang.Object))
  - **hashCode()**

# Abstract Classes

Using **abstract** classes,

- ❖ we can declare classes that define **only** part of an implementation,
- ❖ leaving extended classes to provide specific implementation of some or all the methods.

The benefit of an **abstract** class

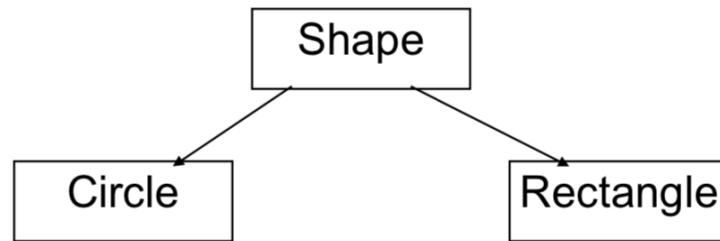
- ❖ is that methods may be declared such that the programmer knows **the interface definition** of an object,
- ❖ however, methods can be **implemented differently** in different subclasses of the abstract class.

# Abstract Classes

Some rules about abstract classes:

- ❖ An abstract class is a class that is **declared abstract**.
- ❖ If a class **includes abstract methods**, then the class itself must be declared abstract.
- ❖ An abstract class **cannot be instantiated**.
- ❖ A subclass of an abstract class can be instantiated if it overrides each of the abstract methods of its superclass and provides **an implementation** for **all** of them.
- ❖ If a subclass of an abstract class **does not implement** all the abstract methods it inherits, that subclass is itself abstract.

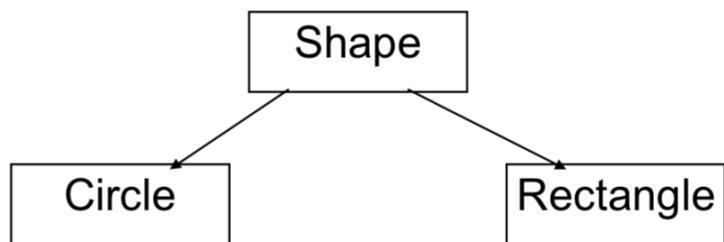
# Abstract Class: Example



```
public abstract class Shape {  
    public abstract double area();  
    public abstract double circumference();  
}
```

```
public class Circle extends Shape {  
  
    protected static final double pi = 3.14159;  
    protected int x, y;  
    protected int r;  
  
    // Very simple constructor  
    public Circle(){  
        this.x = 1;  
        this.y = 1;  
        this.r = 1;  
    }  
    // Another simple constructor  
    public Circle(int x, int y, int r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    /**  
     * Below, methods that return the circumference  
     * area of the circle  
     */  
    public double circumference( ) {  
        return 2 * pi * r ;  
    }  
    public double area ( ) {  
        return pi * r * r ;  
    }  
}
```

# Abstract Class: Example



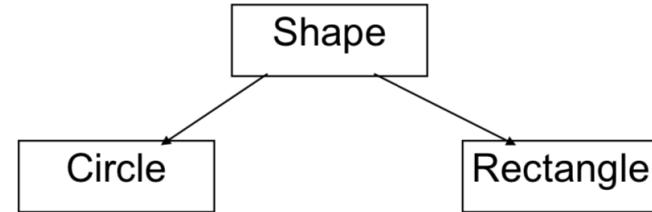
```
public abstract class Shape {  
    public abstract double area();  
    public abstract double circumference();  
}
```

```
public class Rectangle extends Shape {  
  
    protected double width, height;  
  
    public Rectangle() {  
        width = 1.0;  
        height = 1.0;  
    }  
  
    public Rectangle(double w, double h) {  
        this.width = w;  
        this.height = h;  
    }  
  
    public double area(){  
        return width*height;  
    }  
  
    public double circumference() {  
        return 2*(width + height);  
    }  
}
```

# Abstract Class: Example

Some points to note:

- ❖ As **Shape** is an abstract class, we cannot instantiate it.
- ❖ Instantiations of **Circle** and **Rectangle** can be assigned to variables of **Shape**.  
**No cast** is necessary
- ❖ In other words, subclasses of **Shape** can be assigned to elements of an array of **Shape**.  
**No cast** is necessary.
- ❖ We can invoke **area()** and **circumference()** methods for **Shape** objects.



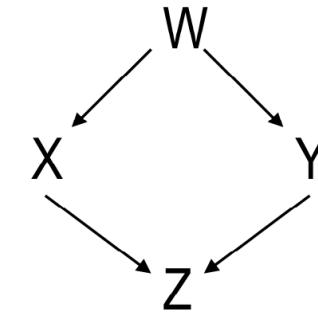
We can now write code like this:

```
// create an array to hold shapes
Shape[] shapes = new Shape[4];
shapes[0] = new Circle(4, 6, 2);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
shapes[3] = new GraphicalCircle(1, 1, 6,
    Color.green, Color.yellow);

double total_area = 0;
for(int i = 0; i < shapes.length; i++) {
    // compute the area of the shapes
    total_area += shapes[i].area();
}
```

# Single Inheritance versus Multiple Inheritance

- In Java, a new class can extend exactly one superclass - a model known as *single inheritance*.
- Some object-oriented languages employ *multiple inheritance*, where a new class can have two or more *super classes*.
- In multiple inheritance, **problems** arise when a superclass's behaviour is **inherited in two/multiple ways**.
- Single inheritance precludes some useful and correct designs.
- In Java, **interface** in the class hierarchy can be used to add multiple inheritance, more discussions on this later.

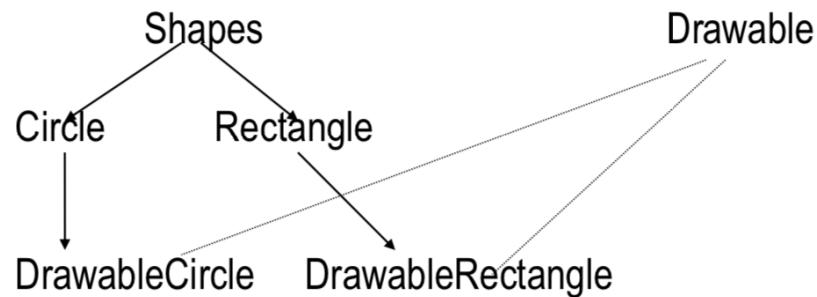


Diamond inheritance  
problem

# Interfaces in Java

- ❖ Interfaces are like abstract classes, but with few **important differences**.
- ❖ All the methods defined within an interface are **implicitly abstract**. (We don't need to use abstract keyword, however, to improve clarity one can use abstract keyword).
- ❖ **Variables** declared in an interface must be **static and final**, that means, they must be **constants**.
- ❖ Just like a class **extends** its superclass, it also can optionally **implements** an interface.
- ❖ In order to implement an interface, a class must first declare the interface in an **implements** clause, and then it must provide an implementation for all of the abstract methods of the interface.
- ❖ A class can “**implements**” more than one **interfaces**.
- ❖ More discussions on “**interfaces**” later in the course.

# Interfaces in Java: Example



```
public interface Drawable {
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(Graphics g);
}

public class DrawableRectangle
    extends Rectangle
    implements Drawable {
    private Color c;
    private double x, y;

    .....
    .....

    // Here are implementations of the
    // methods in Drawable
    // we also inherit all public methods
    // of Rectangle

    public void setColor(Color c) { this.c = c; }
    public void setPosition(double x, double y) {
        this.x = x; this.y = y; }
    public void draw(Graphics g) {
        g.drawRect(x,y,w,h,c); }
}
```

## Using Interfaces: Example

- ❖ When a class **implements** an interface, instance of that class can also be **assigned to** variables of the interface type.

```
Shape[] shapes = new Shape[3];
Drawable[] drawables = new Drawable[3];

DrawableCircle dc = new DrawableCircle(1.1);
DrawableSquare ds = new DrawableSquare(2.5);
DrawableRectangle dr = new DrawableRectangle(2.3,
4.5);

// The shapes can be assigned to both arrays
shapes[0] = dc; drawables[0] = dc;
shapes[1] = ds; drawables[1] = ds;
shapes[2] = dr; drawables[2] = dr;

// We can invoke abstract method
// in Drawable and Shapes

double total_area = 0;
for(int i=0; i< shapes.length; i++) {

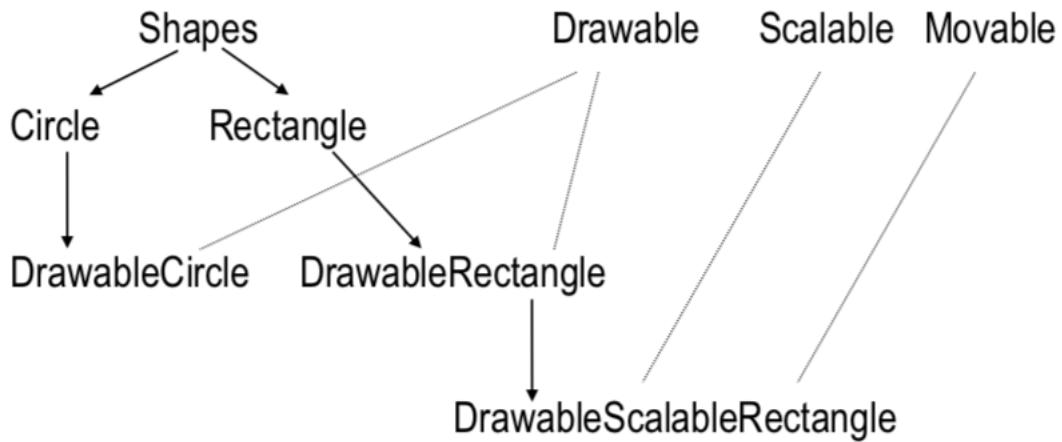
    total_area += shapes[i].area();

    drawables[i].setPosition(i*10.0, i*10.0);

    // assume that graphic area 'g' is
    // defined somewhere
    drawables[i].draw(g);
}
```

# Implementing Multiple Interfaces

A class can **implements** more than one interfaces. For example,



```
public class DrawableScalableRectangle
    extends DrawableRectangle
    implements Movable, Scalable {

    // methods go here ...

}
```

# Extending Interfaces

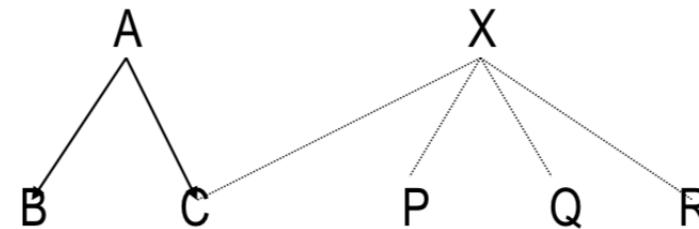
- ❖ Interfaces can have **sub-interfaces**, just like classes can have subclasses.
- ❖ A sub-interface **inherits all** the abstract methods and constants of its super-interface, and may define new abstract methods and constants.
- ❖ Interfaces **can extend** more than one interface at a time. For example,

```
public interface Transformable
    extends Scalable, Rotable, Reflectable {}

public interface DrawingObject
    extends Drawable, Transformable {}

public class Shape implements DrawingObject {
    ...
}
```

# Method Forwarding



- ❖ Suppose class C extends class A, and also implements interface X.
- ❖ As all the methods defined in interface X are abstract, class C needs to implement all these methods.
- ❖ However, there are three implementations of X (in P,Q,R).
- ❖ In class C, we may want to use one of these implementations, that means, we may want to use some or all methods implemented in P, Q or R.
- ❖ Say, we want to use methods implemented in P. We can do this by creating an object of type class P in class C, and through this object access all the methods implemented in P.
- ❖ Note that, in class C, we do need to provide required stubs for all the methods in the interface X. In the body of the methods we may simply call methods of class P via the object of class P.
- ❖ This approach is also known as **method forwarding**.

# Methods Overriding (Polymorphism)

- ❖ When a class defines a method using the **same** name, return type, and by the number, type, and position of its arguments as a method in its *superclass*, the method in the class **overrides** the method in the *superclass*.
- ❖ If a method is invoked for an object of the class, it's the **new definition** of the method that is called, and **not** the superclass's **old definition**.

## Polymorphism

- An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is usually called **polymorphism**.

# Methods Overriding: Example

In the example below,

- ❖ if `p` is an instance of class `B`,  
`p.f()` refers to `f()` in class `B`.
- ❖ However, if `p` is an instance of class `A`,  
`p.f()` refers to `f()` in class `A`.

The example also shows how to refer to the `overridden` method using `super` keyword.

```
class A {  
    int i = 1;  
    int f() { return i; }  
}  
  
class B extends A {  
    int i;                                // shadows i from A  
    int f() {  
        i = super.i + 1;                  // retrieves i from A  
        return super.f() + i;             // invokes f() from A  
    }  
}
```

# Methods Overriding: Example

Suppose class C is a subclass of class B, and class B is a subclass of class A.

Class A and class C both define method `f()`.

From class C, we can refer to the overridden method by,

`super.f()`

This is because class B inherits method `f()` from class A.

However,

- ❖ if `all the three` classes define `f()`, then calling `super.f()` in class C invokes class B's definition of the method.
- ❖ Importantly, in this case, there is `no way` to invoke `A.f()` from within class C.
- ❖ Note that `super.super.f()` is `NOT legal` Java syntax.

# Method Overloading

Defining methods with the **same name** and **different** argument or return types is called *method overloading*.

In Java,

- ❖ a method is distinguished by its **method signature** - its name, return type, and by the number, type, and position of its arguments

For example,

```
double add(int, int)
double add(int, double)
double add(float, int)
double add(int, int, int)
double add(int, double, int)
```

# Data Hiding and Encapsulation

We can **hide** the **data** within the class and make it available only through the methods.

This can help in maintaining the consistency of the data for an object, that means the state of an object.

## Visibility Modifiers

Java provides five access modifiers (for variables/methods/classes),

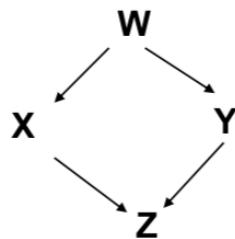
- ❖ **public** - visible to the world
- ❖ **private** - visible to the class only
- ❖ **protected** - visible to the package and all subclasses
- ❖ **No modifier (default)** - visible to the package

# Constructors

- ❖ Good practice to **define** the required constructors for **all** classes.
- ❖ If a constructor is **not defined** in a class,
  - **no-argument** constructor is **implicitly inserted**.
  - this no-argument constructor invokes the **superclass's no-argument** constructor.
  - if the parent class (superclass) doesn't have a visible constructor with no-argument, it results in a compilation **error**.
- ❖ If the **first statement** in a constructor is **not** a call to **super()** or **this()**, a call to **super ()** is **implicitly** inserted.
- ❖ If a constructor is **defined** with **one or more arguments**, **no-argument** constructor is **not inserted** in that class.
- ❖ A class can have **multiple** constructors, with **different signatures**.
- ❖ The word “**this**” can be used to call another constructor in the same class.

# Diamond Inheritance Problem: A Possible Solution

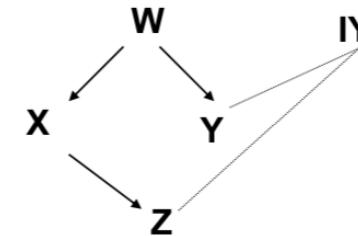
Using **multiple inheritance** (in C++):



we achieve the following:

- In class **Z**, we can use methods and variables defined in **X**, **W** **and** **Y**.
- Objects of classes **Z** and **Y** can be assigned to variables of **type Y**.
- and more ...

Using **single inheritance** in Java:



```
class W {}  
interface IY {}  
class X extends W {}  
class Y extends W implements IY {}  
class Z extends X implements IY {}
```

we achieve the following:

- In class **Z**, we can use methods and variables defined in **X** and **W**. In class **Z**, if we want to use methods implemented in class **Y**, we can use **method forwarding** technique. That means, in class **Z**, we can create an object of type class **Y**, and via this object we can access (in class **Z**) all the methods defined in class **Y**.
- Objects of classes **Z** and **Y** can be assigned to variables of **type IY** (instead of **Y**).
- and more ....

# Some References to Java Tutorials

- ❖ <https://docs.oracle.com/javase/tutorial/>
- ❖ <https://www.w3schools.com/java/default.asp>
- ❖ <https://www.tutorialspoint.com/java/index.htm>

# COMP2511

# Design by Contract

Prepared by

Dr. Ashesh Mahidadia

# Defensive Programming Vs Design by Contract

## Defensive programming:

Tries to **address unforeseen** circumstances, in order to ensure the continuing functionality of the software element. For example, it makes the software behave in a predictable manner despite unexpected inputs or user actions.

- often used where **high availability**, safety or security is needed.
- results in **redundant checks** (both client and supplier may perform checks), more **complex software** for maintenance.
- difficult to locate errors, considering there is **no clear demarcation** of responsibilities.
- may safeguard against errors that will never be encountered, thus incurring run-time and maintenance costs.

## Design by Contract:

At the design time, **responsibilities** are **clearly assigned** to different software elements, clearly documented and enforced during the development using unit testing and/or language support.

- clear demarcation of responsibilities helps **prevent redundant checks**, resulting in **simpler** code and **easier** maintenance.
- crashes if the required conditions are not satisfied! May **not be suitable** for **high availability** applications.

# Design by Contract (DbC)

- ❖ Bertrand Meyer coined the term for his design of the Eiffel programming language (in 1986). Design by Contract (DbC) has its roots in work on formal specification, formal verification and Hoare logic.
- ❖ In business, when two parties (supplier and client) *interact* with each other, often they write and sign **contracts** to clarify the **obligations** and **expectations**. For example,

	Obligations	Benefits
Client	<i>(Must ensure precondition)</i> Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	<i>(May benefit from post-condition)</i> Reach Chicago.
Supplier	<i>(Must ensure post-condition)</i> Bring customer to Chicago.	<i>(May assume pre-condition)</i> No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

The example is from <https://www.eiffel.com/values/design-by-contract/introduction/>

# Design by Contract (DbC)

Every software element should define a **specification** (or a **contract**) that governs its interaction with the rest of the software components.

A **contract** should address the following three questions:

- ❖ **Pre-condition** - what does the contract expect?

If the precondition is true, it can avoid handling cases outside of the precondition.

For example, expected argument value (`mark>=0`) and (`marks<=100`).

- ❖ **Post-condition** - what does the contract guarantee?

Return value(s) is guaranteed, provided the precondition is true.

For example: correct return value representing a grade.

- ❖ **Invariant** - what does the contract maintain?

Some values must satisfy constraints, before and after the execution (say of the method).

For example: a value of `mark` remains between zero and 100.

# Design by Contract (DbC)

A **contract** (precondition, post-condition and invariant) should be,

- ❖ declarative and must **not** include implementation details.
- ❖ as far as possible: **precise, formal** and **verifiable**.

# Benefits of Design by Contract (DbC)

- ❖ Do not need to do error checking for conditions that not satisfy the preconditions!
- ❖ Prevents redundant validation tasks.
- ❖ Given the preconditions are satisfied, clients can expect the specified post-conditions.
- ❖ Responsibilities are clearly assigned, this helps in locating errors and resulting in easier code maintenance.
- ❖ Helps in cleaner and faster development.

# Design by Contract (DbC) : Implementation Issues

- ❖ Some programming languages (like Eiffel) offer **native support** for DbC.
- ❖ Java does **not have native support** for DbC, there are various libraries to support DbC.
- ❖ In the absence of a native language support, **unit testing** is used to test the contracts (preconditions, post-conditions and invariants).
- ❖ Often preconditions, post-conditions and invariants are **included** in the **documentation**.
- ❖ As indicated earlier, **contracts** should be,
  - **declarative** and must not include implementation details.
  - as far as possible: **precise, formal and verifiable**.

# Design by Contract : Example using Eiffel

```
class DICTIONARY [ELEMENT]
feature
    put (x: ELEMENT; key: STRING) is
        -- Insert x so that it will be retrievable
        -- through key.

        require
            count <= capacity
            not key.empty

        ensure
            has (x)
            item (key) = x
            count = old count + 1
        end
    end
```

Precondition → **require**

Postcondition → **ensure**

invariant → **invariant**

... Interface specifications of other features ...

# Design by Contract: Examples in Java

```
/**  
 * @param value to calculate square root  
 * @returns sqrt - square root of the value  
 * @pre value >= 0  
 * @post value = sqrt * sqrt  
 */  
public double squareRoot ( double value );
```

```
/**  
 * @invariant age >= 0  
 */  
  
public class Student {
```

```
/**  
 * @param amount to be deposited into the account  
 * @pre amount > 0  
 * @post balance = old balance + amount  
 */  
public void deposit( double amount);
```

# Pre-Conditions

- ❖ A **pre-condition** is a condition or predicate that must always be true just **prior** to the execution of some section of code
- ❖ If a precondition is violated, the effect of the section of code becomes **undefined** and thus may or may not carry out its intended work.
- ❖ Security problems can **arise** due to **incorrect** pre-conditions.
- ❖ Often, preconditions are **included** in the **documentation** of the affected section of code.
- ❖ Preconditions are sometimes **tested** using **guards** or **assertions** within the code itself, and some languages have **specific** syntactic **constructions** for testing .
  
- ❖ In **Design by Contract**, a software element can **assume** that **preconditions are satisfied**, resulting in removal of redundant error checking code.
  
- ❖ See the next slide for the examples.

# Pre-Conditions: Examples

```
/**  
 * @pre (mark >=0) and (mark<=100)  
 * @param mark  
 */  
public void printGradeDbC(double mark) {  
  
    if(mark < 50 ) {  
        System.out.println("Fail");  
    }  
    else {  
        System.out.println("Pass");  
    }  
}
```

Incorrect behaviour if *mark* is outside the expected range

```
/**  
 * Get Student at i'th position  
 * @pre i < number_of_students  
 * @param i - student's position  
 * @return student at i'th position  
 */  
public Student getStudentDbC(int i) {  
  
    return students.get(i);  
}
```

Throws runtime exception if (*i* >= *number\_of\_students*)

## Design by Contract

No additional error checking for pre-conditions

```
/**  
 * @pre (mark >=0) and (mark<=100)  
 * @param mark  
 */  
public void printGradeDefensive(double mark) {  
  
    if( (mark < 0) || (mark > 100) ){  
        System.out.println("Error");  
    }  
  
    if(mark < 50 ) {  
        System.out.println("Fail");  
    }  
    else {  
        System.out.println("Pass");  
    }  
}
```

Defensive Programming:  
Additional error checking for pre-conditions

# Pre-Conditions in Inheritance

- ❖ An implementation or redefinition (method overriding) of an inherited method **must comply** with the **inherited contract** for the method.
- ❖ Preconditions **may be weakened** (relaxed) in a subclass, but it must comply with the inherited contract.
- ❖ An implementation or redefinition **may lessen** the obligation of the client, but not increase it.
- ❖ For example,

```
/**  
 * @pre (theta >=0) and (theta <= 90)  
 * @param theta - angle to calculate trajectory  
 * @return trajectory at angle theta  
 */  
public double calculateTrajectory(double theta) {
```

valid

Weaker Pre-condition

```
/**  
 * @pre (theta >=0) and (theta <= 180)  
 * @param theta - angle to calculate trajectory  
 * @return trajectory at angle theta  
 */  
public double calculateTrajectory(double theta) {
```

X - not valid

Stronger Pre-condition

```
/**  
 * @pre (theta >=0) and (theta <=45)  
 * @param theta - angle to calculate trajectory  
 * @return trajectory at angle theta  
 */  
public double calculateTrajectory(double theta) {
```

# Post-Conditions

- ❖ A **post-condition** is a condition or predicate that must always be true just **after** the execution of some section of code
- ❖ The **post-condition** for any routine is a declaration of the properties which are guaranteed upon completion of the routine's execution<sup>[1]</sup>.
- ❖ Often, preconditions are **included** in the **documentation** of the affected section of code.
- ❖ Post-conditions are sometimes **tested** using **guards** or **assertions** within the code itself, and some languages have **specific** syntactic **constructions** for testing .
- ❖ In **Design by Contract**, the properties declared by **the post-condition(s)** are **assured**, provided the software element is called in a state in which its pre-condition(s) were true.

[1] Meyer, Bertrand, Object-Oriented Software Construction, second edition, Prentice Hall, 1997.

```
/*
@param value to calculate square root
@returns sqrt - square root of the value
@pre value >= 0
@post value = sqrt * sqrt
*/
public double squareRoot ( double value );
```

# Post-Conditions in Inheritance

- ❖ An implementation or redefinition (method overriding) of an inherited method **must comply** with the **inherited contract** for the method.
- ❖ Post-conditions **may be strengthened** (more restricted) in a subclass, but it must comply with the inherited contract.
- ❖ An implementation or redefinition (overridden method) **may increase** the benefits it provides to the client, but **not decrease** it.
- ❖ For example,
  - ❖ the original contract requires returning a **set**.
  - ❖ the redefinition (overridden method) returns **sorted set**, offering *more* benefit to a client.

# Class Invariant

- ❖ The class invariant **constrains** the **state** (i.e. values of certain variables) stored in the object.
- ❖ Class invariants are **established** during construction and constantly **maintained** between calls to public methods. Methods of the class must make sure that the class invariants are satisfied / preserved.
- ❖ **Within a method:** code **within** a method **may break** invariants as long as the invariants are **restored** before a public method ends.
- ❖ Class invariants help programmers to rely on a valid state, avoiding risk of inaccurate / invalid data. Also helps in locating errors during testing.

## Class invariants in Inheritance

- ❖ Class invariants are **inherited**, that means,  
*"the **invariants of all the parents** of a class apply to the class itself!"* !
- ❖ A subclass can access implementation data of the parents, however, **must always satisfy** the **invariants of all the parents** – preventing invalid states!

# End

# COMP2511

# Domain Modelling

# using UML

Prepared by Dr. Robert Clifton-Everest

Updated by Dr. Ashesh Mahidadia

# Domain Models

- Domain Models are used to **visually represent** important domain **concepts** and **relationships** between them.
- Domain Models help **clarify** and **communicate** important domain specific concepts and are used during the requirements gathering and **designing phase**.
- **Domain modeling** is the activity of expressing related domain concepts into a domain model.
- Domain models are also often referred to as *conceptual models* or *domain object models*.
- We will be use Unified Modeling Language (**UML**) **class diagrams** to represent domain models.
- There are many different modelling frameworks, like: UML, Entity-Relationship, Mind maps, Context maps, Concept diagrams. etc.

# Requirements Analysis vs Domain modelling

- Requirements analysis determines *external behaviour*  
*“What are the features of the system-to-be and who requires these features (actors) ”*
- Domain modelling determines (internal behavior) –  
*“how elements of system-to-be interact to produce the external behaviour”*
- Requirements analysis and domain modelling are **mutually dependent** - domain modelling supports clarification of requirements, whereas requirements help building up the model.

# What is a domain?

- *Domain* – A sphere of knowledge particular to the problem being solved
- *Domain expert* – A person expert in the domain
- For example, in the domain of cake decorating, cake decorators are the domain experts

# Problem

A motivating example:

- Tourists have schedules that involve at least one and possibly several cities
- Hotels have a variety of rooms of different grades: standard and premium
- Tours are booked at either a standard or premium rate, indicating the grade of hotel room
- In each city of their tour, a tourist is booked into a hotel room of the chosen grade
- Each room booking made by a tourist has an arrival date and a departure date
- Hotels are identified by a name (e.g. Melbourne Hyatt) and rooms by a number
- Tourists may book, cancel or update schedules in their tour

# Ubiquitous language

- **Things in our design must represent real things in the domain expert's mental model.**
- For example, if the domain expert calls something an "order" then in our domain model (and ultimately our implementation) we should have something called an Order.
- Similarly, our domain model should not contain an OrderHelper, OrderManager, etc.
- Technical details do not form part of the domain model as they are not part of the design.

# Noun/verb analysis

- Finding the ubiquitous language of the domain by finding the **nouns** and **verbs** in the requirements
- The **nouns** are possible entities in the domain model and the **verbs** possible behaviours

# Problem

- The **nouns** and **verbs**:
  - Tourists have **schedules** that involve at least one and possibly several **cities**
  - Hotels have a variety of **rooms** of different **grades**: standard and premium
  - Tours are **booked** at either a standard or premium rate, indicating the **grade** of hotel **room**
  - In each **city** of their **tour**, a **tourist** is **booked** into a hotel **room** of the chosen **grade**
  - Each room **booking** made by a tourist has an arrival **date** and a departure **date**
  - Hotels are identified by a **name** (e.g. Melbourne Hyatt) and **rooms** by a **number**
  - Tourists may **book**, **cancel** or **update** **schedules** in their **tour**

# UML Class diagrams

- Classes



- Relationships

Dependency      

Aggregation      

Composition      

Association      

Directed  
Association      

# UML Class diagrams

Dependency



- The loosest form of relationship. A class in some way *depends* on another.

Association



- A class "uses" another class in some way. When undirected, it is not yet clear in what direction dependency occurs.

Directed  
Association



- Refines association by indicating which class has knowledge of the other

# UML Class diagrams

Aggregation



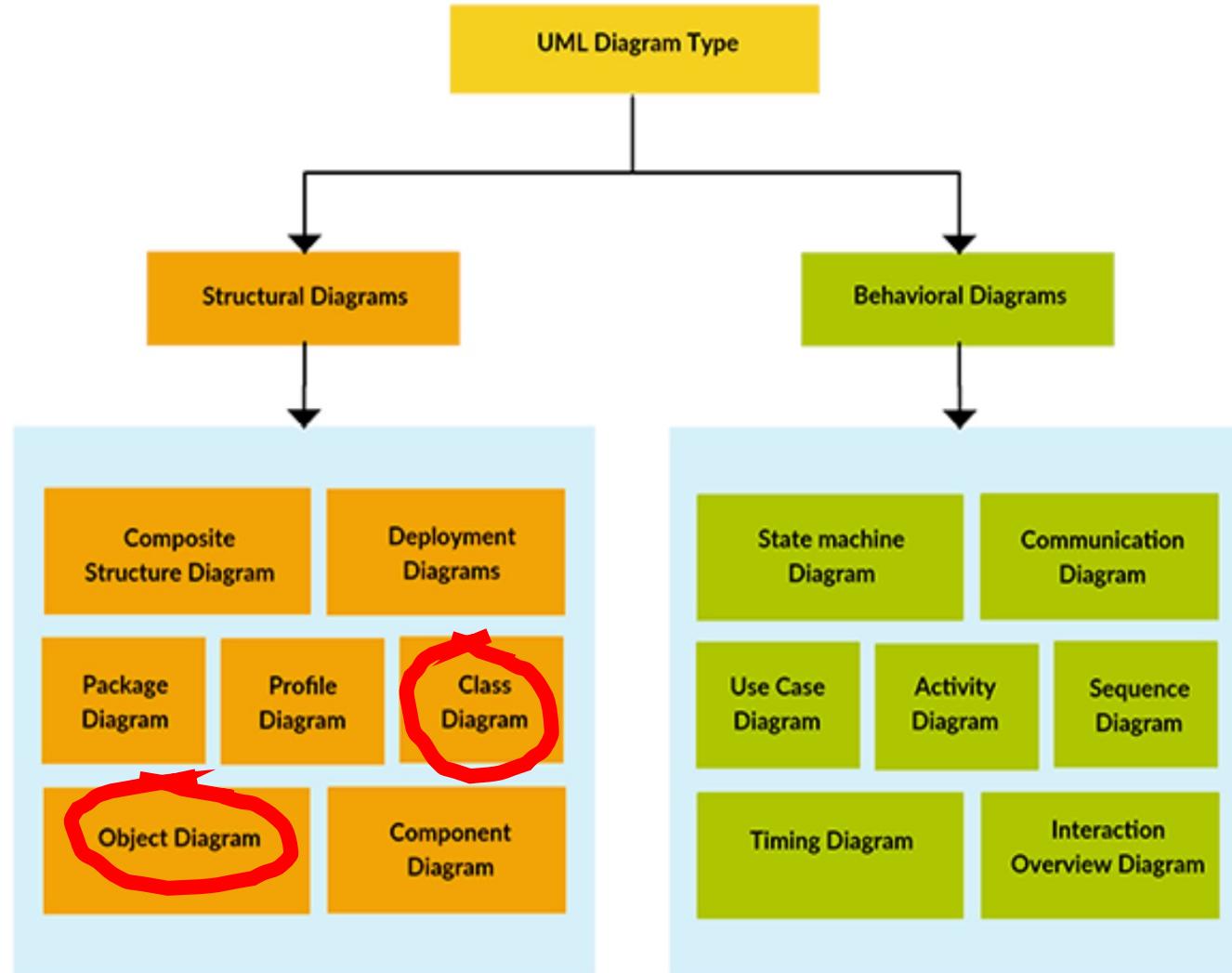
- A class contains another class (e.g. a course contains students). Note that the diamond is at the end with the containing class.

Composition



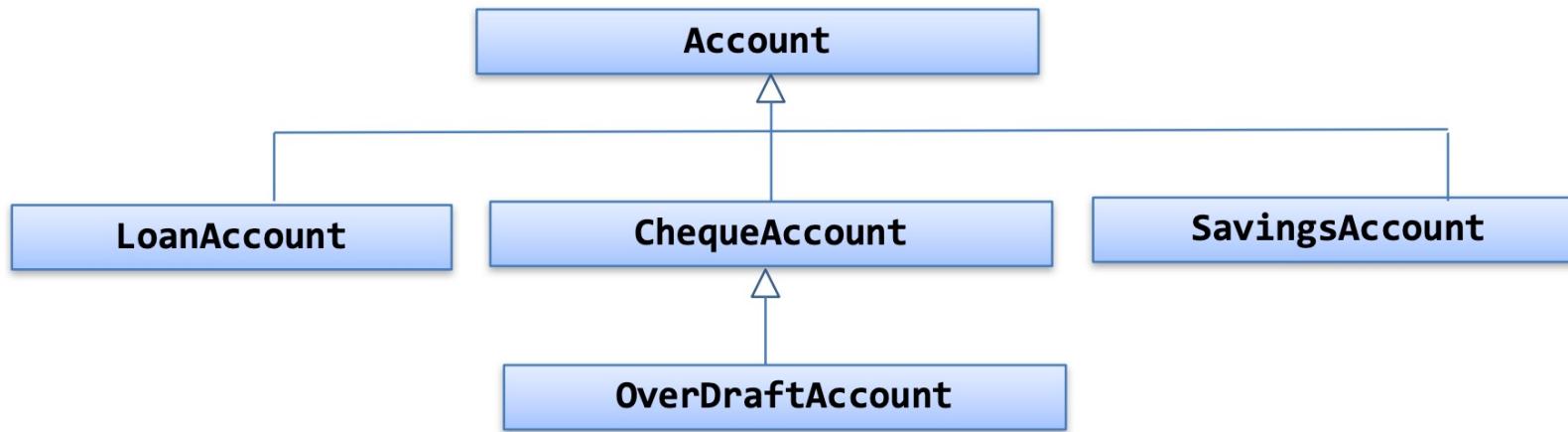
- Like aggregation, but the contained class is integral to the containing class. The contained class cannot exist outside of the container (e.g. the leg of a chair)

# UML Diagram Types



© Aarthi Natarajan, 2018

## Examples



# Representing classes in UML

## class ( class diagram )

**Account**

-name: String  
-balance: float

+getBalance(): float  
+getName() : String  
+withDraw(float)  
+deposit(float)

## object instances (object diagram)

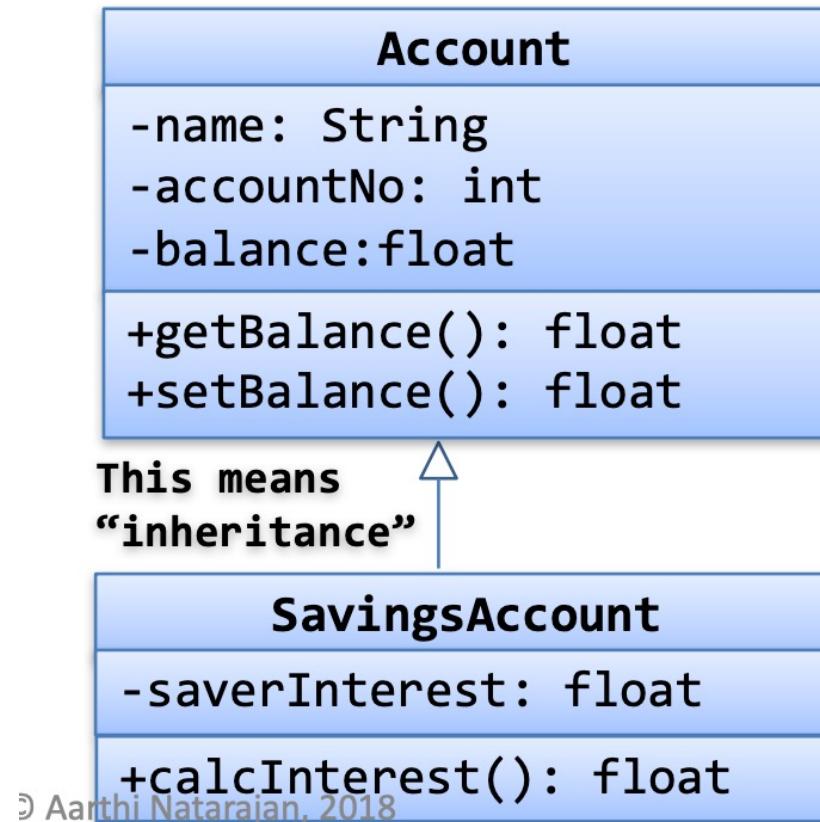
**a1:Account**

name = "John Smith"  
balance = 40000

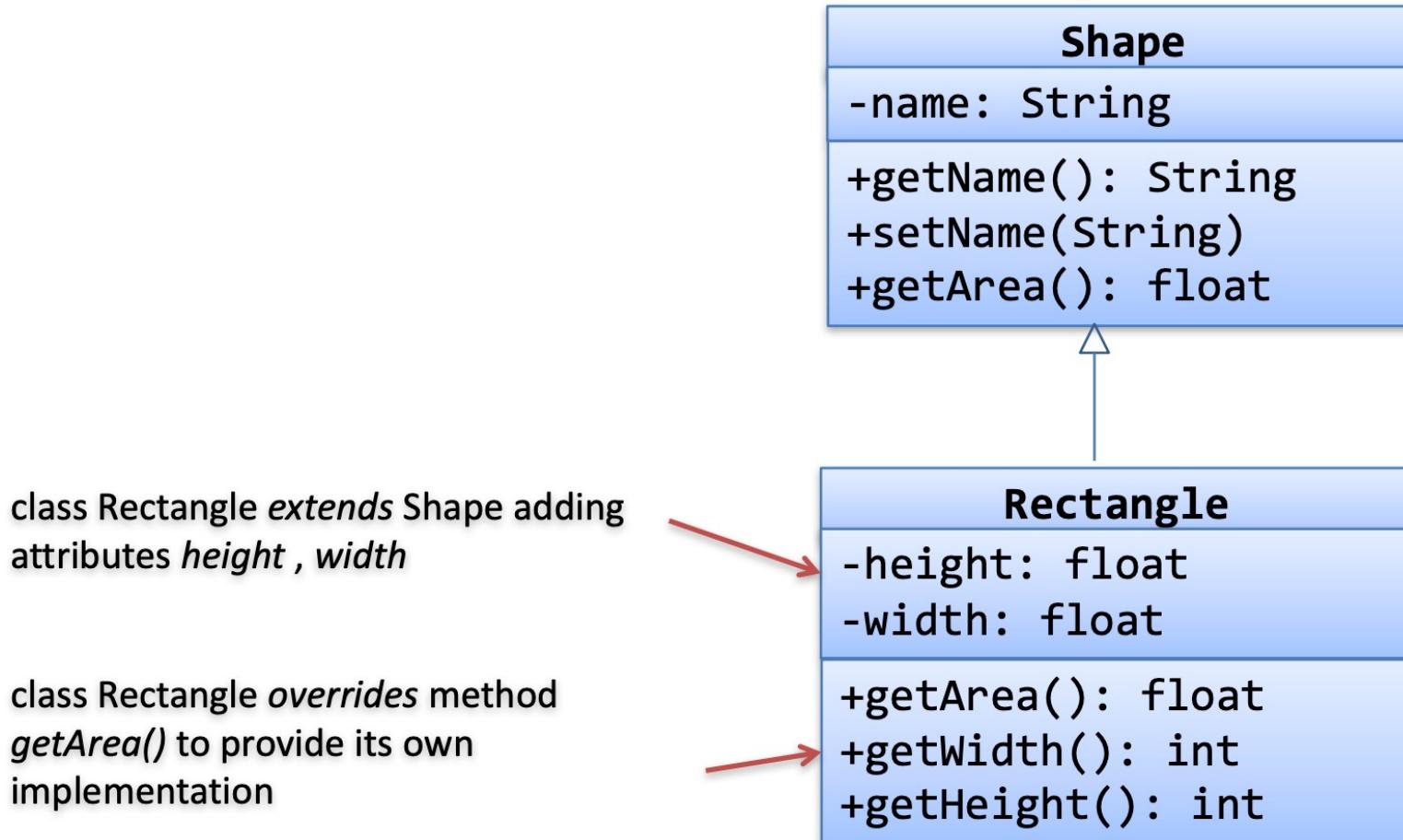
**a2:Account**

name = "Joe Bloggs"  
balance = 50000

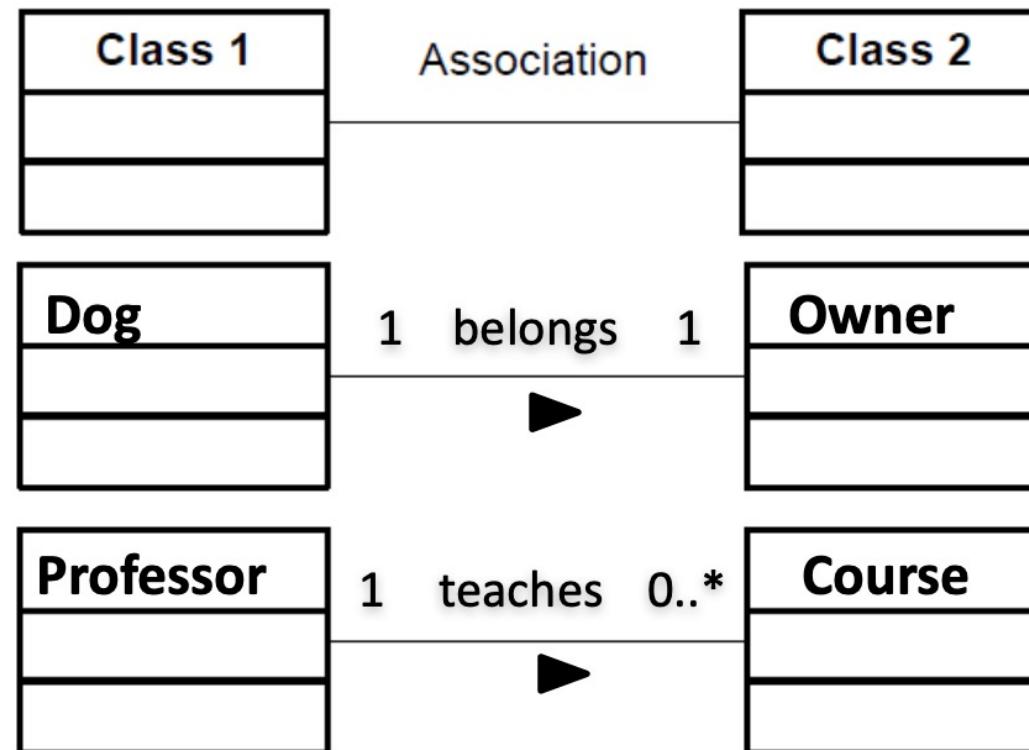
# Representing classes in UML



# Representing classes in UML



# Representing Association in UML



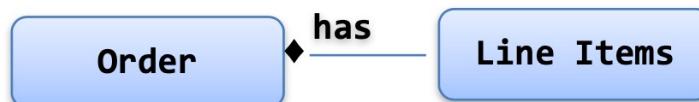
© Aarthi Natarajan, 2018

# Representing Association in UML

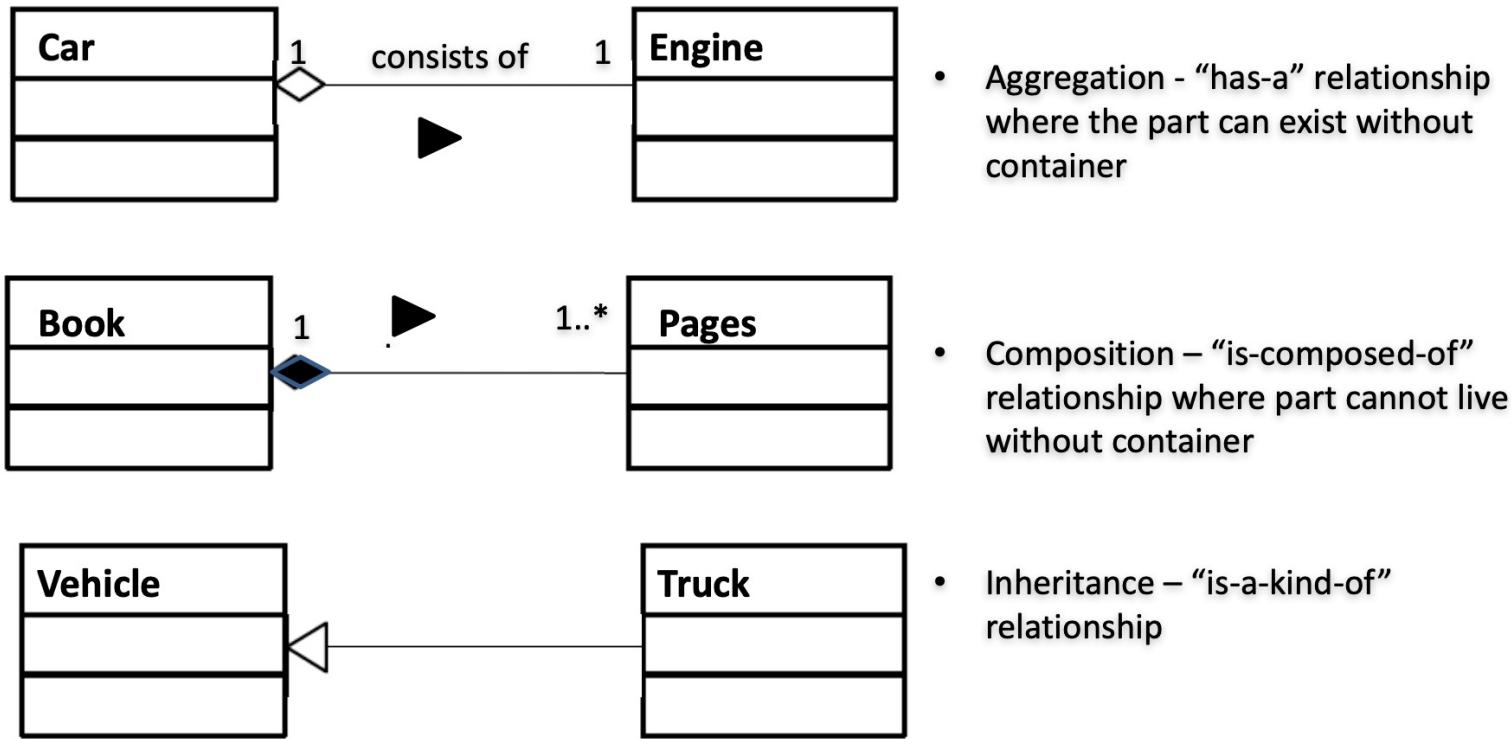
- Associations can model a “**has-a**” relationship where one class “contains” another class
- Associations can further be refined as:  
*Aggregation* relationship (hollow diamond symbol  $\diamond$ ): The contained item is an element of a collection but it can also exist on its own, e.g., a lecturer in a university or a student at a university



*Composition* relationship (filled diamond symbol  $\blacklozenge$  in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk, or engine in a car



# Representing Association in UML



# Attributes vs. Classes

- ❖ The most common confusion – *should it be an attribute or a class?*
  - when creating a domain model, often we need to decide whether to represent *something* as an *attribute* or a conceptual *class*.
- ❖ If a concept is **not** representable by a *number* or a *string*, most likely it is a *class*.
- ❖ For example:
  - a *lab mark* can be represented by a *number*, so we should represent it as an *attribute*
  - a *student* cannot be represented by a *number or a string*, so we should represent it as a *class*

# Attributes vs. Classes



# What wrong with the following?

Course
+ Title:
+ Code:
+ UOC
+ Term: String
+ Lecturers [] :
+ Sessions [] :
+ Students []:

Object:Course
Title = "Inro to computing"
Code = "COMP1511"
UOC = 6
Term = 19T1
Lecturers []
Sessions []
Students []

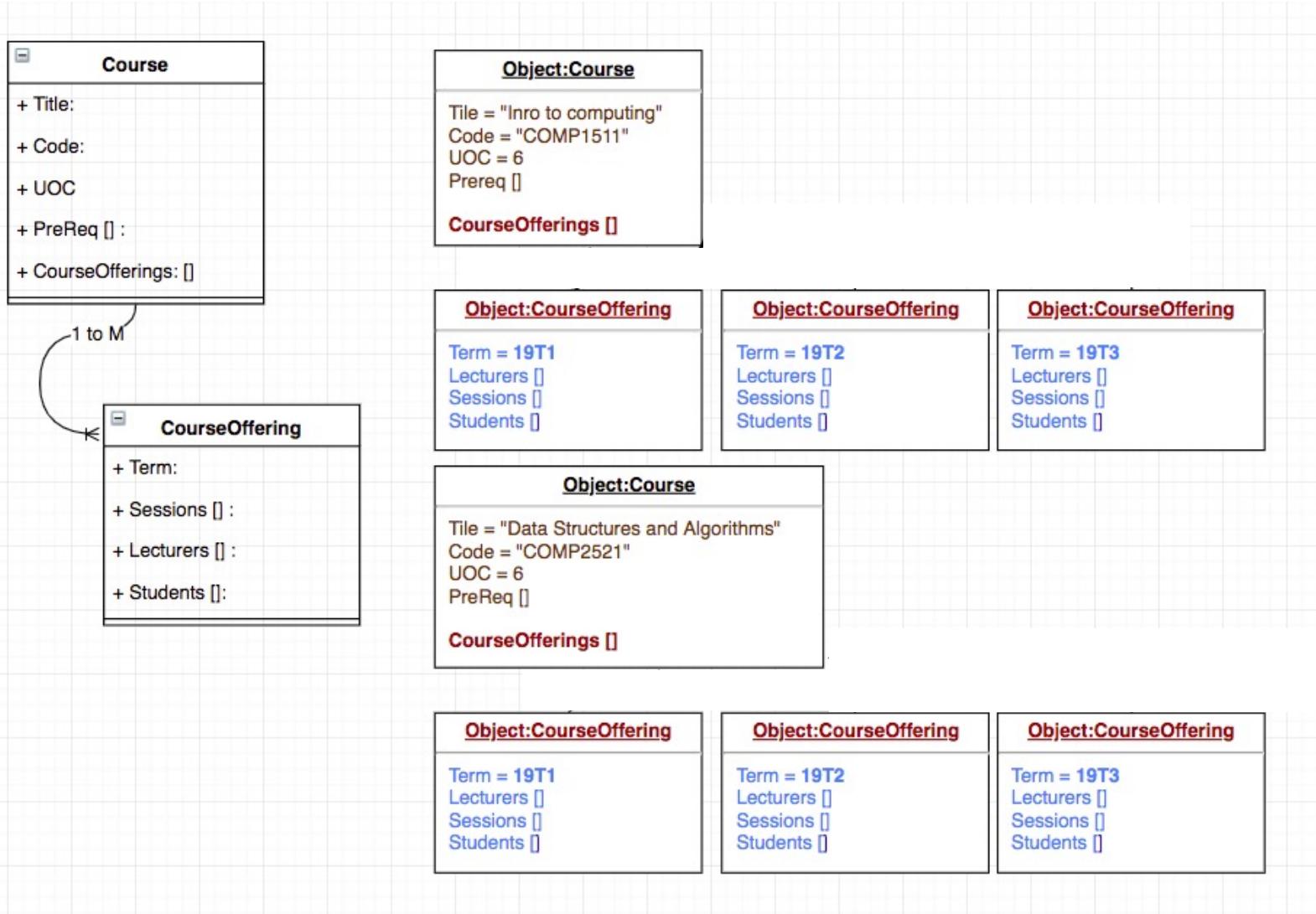
Object:Course
Title = "Inro to computing"
Code = "COMP1511"
UOC = 6
Term = 19T2
Lecturers []
Sessions []
Students []

Object:Course
Title = "Inro to computing"
Code = "COMP1511"
UOC = 6
Term = 19T3
Lecturers []
Sessions []
Students []

Object:Course
Title = "Data Structures and Algoritms"
Code = "COMP2521"
UOC = 6
Term = 19T1
Lecturers []
Sessions []
Students []

Object:Course
Title = "Data Structures and Algoritms"
Code = "COMP2521"
UOC = 6
Term = 19T2
Lecturers []
Sessions []
Students []

# A Possible solution



# References

- A very detailed description of UML
  - <https://www.uml-diagrams.org/>
- Books that go into detail on Domain Driven Design
  - *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans.
  - *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#* by Scott Wlaschin.

**COMP2511**

# **Object Oriented Programming (OOP) in Java**

Prepared by

Dr. Ashesh Mahidadia

# OOP in Java

- ❖ Object Oriented Programming (OOP)
- ❖ Inheritance in OOP
- ❖ Introduction to Classes and Objects
- ❖ Subclasses and Inheritance
- ❖ Abstract Classes
- ❖ Single Inheritance versus Multiple Inheritance
- ❖ Interfaces
- ❖ Method Forwarding (Has-a relationship)
- ❖ Method Overriding (Polymorphism)
- ❖ Method Overloading
- ❖ Constructors

# Object Oriented Programming (OOP)

In procedural programming languages (like ‘C’), programming tends to be **action-oriented**, whereas in Java - programming is **object-oriented**.

In **procedural** programming,

- groups of actions that perform some task are formed into functions and functions are grouped to form programs.

In **OOP**,

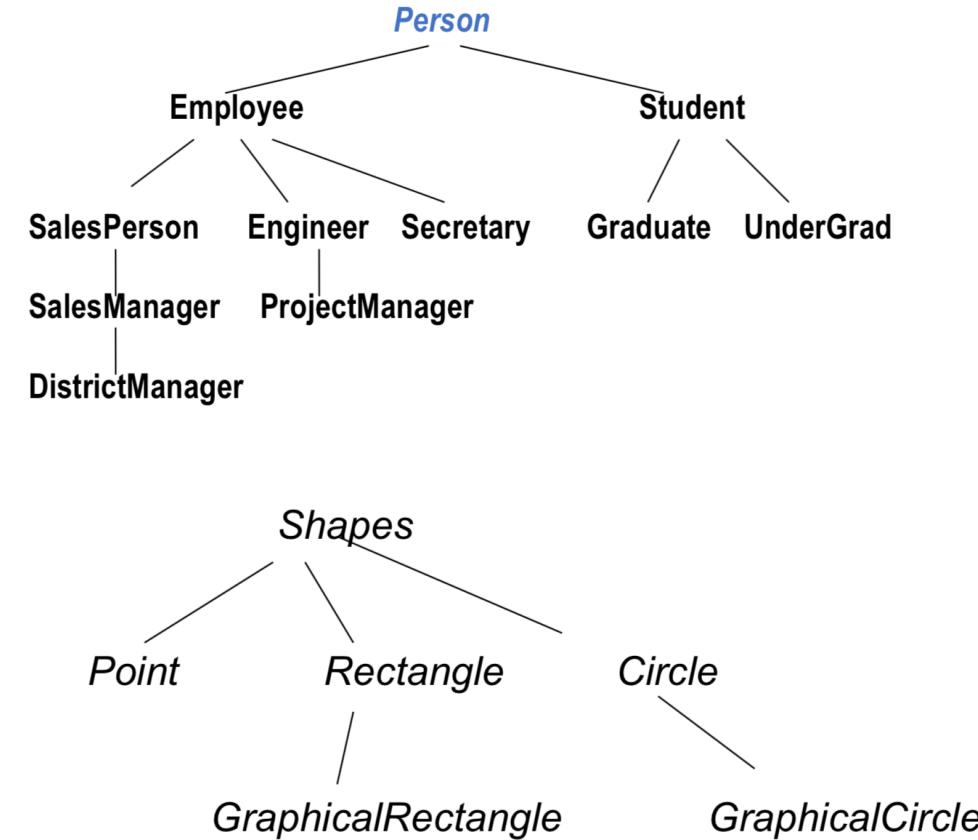
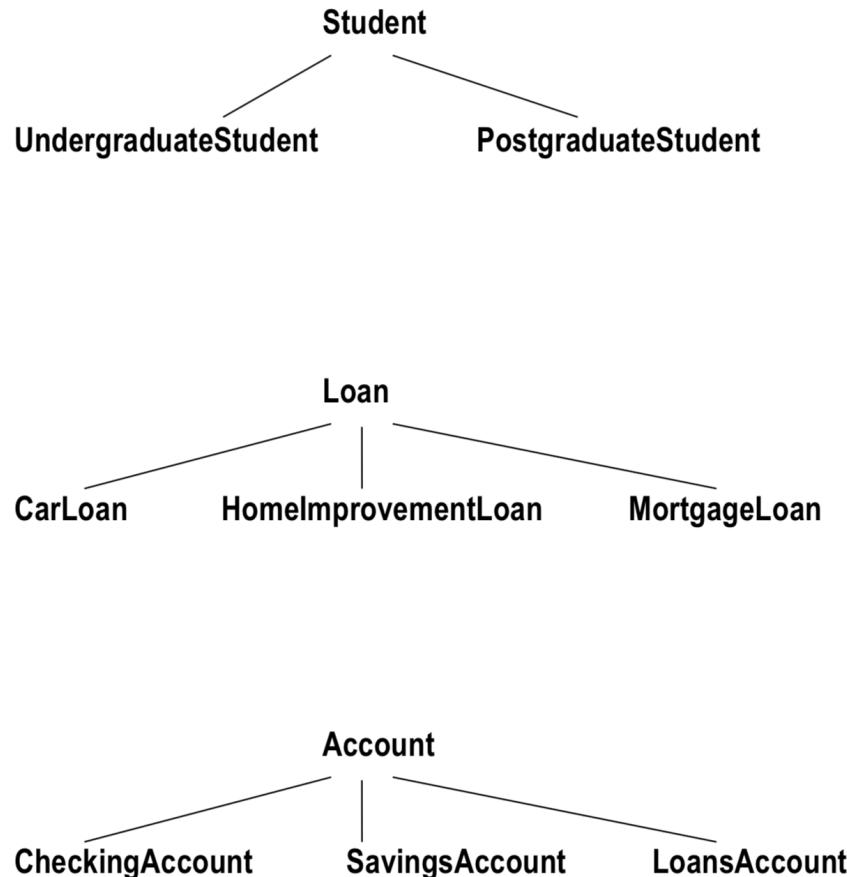
- programmers concentrate on creating their own user-defined types called **classes**.
- each **class** contains **data** as well as the set of **methods** (procedures) that manipulate the data.
- an instance of a user-defined type (i.e. a **class**) is called an **object**.
- OOP **encapsulates** data (attributes) and methods (behaviours) into **objects**, the data and methods of an object are intimately tied together.
- Objects have the property of **information hiding**.

# Inheritance in Object Oriented Programming (OOP)

- ❖ **Inheritance** is a form of software reusability in which new classes are created from the existing classes by absorbing their attributes and behaviours.
- ❖ Instead of defining completely (separate) new class, the programmer can designate that the new class is to **inherit** attributes and behaviours of the existing class (called **superclass**). The new class is referred to as **subclass**.
- ❖ Programmer can add **more attributes and behaviours** to the *subclass*, hence, normally **subclasses** have **more features than their super classes**.

# Inheritance in Object Oriented Programming (OOP)

Inheritance relationships form **tree-like hierarchical** structures. For example,



# “Is-a” - Inheritance relationship

- ❖ In an “**is-a**” relationship, an object of a subclass may also be treated as an object of the superclass.
- ❖ For example, *UndergraduateStudent* can be treated as *Student* too.
- ❖ You should use *inheritance* to model “is-a” relationship.

## Very Important:

- ❖ Don’t use inheritance unless **all or most** inherited attributes and methods **make sense**.
- ❖ For example, mathematically a *circle* is-a (an) *oval*, however you should **not** inherit a class *circle* from a class *oval*. A class *oval* can have one method to set *width* and another to set *height*.

# “Has-a” - Association relationship

- ❖ In a “has-a” relationship, a **class object has an object of another class** to store its state or do its work, i.e. it “has-a” reference to that other object.
- ❖ For example, a Rectangle Is-NOT-a Line.  
However, we may use a Line to draw a Rectangle.
- ❖ The “has-a” relationship is quite different from an “is-a” relationship.
- ❖ “Has-a” relationships are examples of creating new classes by *composition* of existing classes (as oppose to *extending* classes).

## Very Important:

- ❖ Getting “Is-a” versus “Has-a” relationships correct is both **subtle** and potentially **critical**. You should **consider** all **possible** future **usages** of the classes before finalising the hierarchy.
- ❖ It is possible that **obvious solutions may not work** for some applications.

# Designing a Class

- Think carefully about the functionality (methods) a class should offer.
- Always **try to keep data private** (local).
- Consider **different ways** an object may be **created**.
- Creating an object may require different actions such as initializations.
- Always initialize data.
- If the object is no longer in use, free up all the associated resources.
- **Break up classes with too many responsibilities.**
- In OO, classes are often closely related. “**Factor out**” common attributes and behaviours and place these in a class. Then use suitable relationships between classes (for example, “is-a” or “has-a”).

# Introduction to Classes and Objects

- ❖ A class is a collection of **data** and **methods** (procedures) that operate on that data.
- ❖ For example,  
a **circle** can be described by the **x, y** position of its centre and by its **radius**.
- ❖ We can define some useful methods (procedures) for circles,  
compute **circumference**, compute area, check whether pointes are inside the circle,  
etc.
- ❖ By defining the **Circle class** (as below), we can create a **new data type**.

# The class Circle

- ❖ For simplicity, the methods for *getter* and *setters* are not shown in the code.

```
public class Circle {  
  
    protected static final double pi = 3.14159;  
    protected int x, y;  
    protected int r;  
  
    // Very simple constructor  
    public Circle(){  
        this.x = 1;  
        this.y = 1;  
        this.r = 1;  
    }  
    // Another simple constructor  
    public Circle(int x, int y, int r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    /**  
     * Below, methods that return the circumference  
     * area of the circle  
     */  
    public double circumference( ) {  
        return 2 * pi * r ;  
    }  
    public double area ( ) {  
        return pi * r * r ;  
    }  
}
```

# Objects are Instances of a class

In Java, objects are created by instantiating a class.

For example,

```
Circle c ;  
c = new Circle ( ) ;
```

OR

```
Circle c = new Circle ( ) ;
```

# Accessing Object Data

We can access data fields of an object.

For example,

```
Circle c = new Circle();  
  
// Initialize our circle to have centre (2, 5)  
// and radius 1.  
// Assuming, x, y and r are not private  
  
c.x = 2;  
c.y = 5;  
c.r = 1;
```

# Using Object Methods

To access the methods of an object, we can use the same syntax as accessing the data of an object:

```
Circle c = new Circle( );
double a;

c.r = 2;           // assuming r is not private

a = c.area();

// Note that its not :    a = area(c);
```

**COMP2511**

# **Object Oriented Programming (OOP) in Java**

Prepared by

Dr. Ashesh Mahidadia

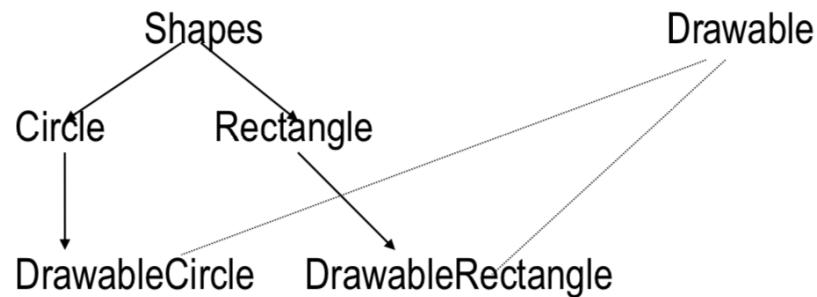
# OOP in Java

- ❖ Object Oriented Programming (OOP)
  - ❖ Inheritance in OOP
  - ❖ Introduction to Classes and Objects
  - ❖ Subclasses and Inheritance
  - ❖ Abstract Classes
  - ❖ Single Inheritance versus Multiple Inheritance
  - ❖ Interfaces
- ❖ Method Forwarding (Has-a relationship)
  - ❖ Method Overriding (Polymorphism)
  - ❖ Method Overloading
  - ❖ Constructors

# Interfaces in Java

- ❖ Interfaces are like abstract classes, but with few **important differences**.
- ❖ All the methods defined within an interface are **implicitly abstract**. (We don't need to use abstract keyword, however, to improve clarity one can use abstract keyword).
- ❖ **Variables** declared in an interface must be **static and final**, that means, they must be **constants**.
- ❖ Just like a class **extends** its superclass, it also can optionally **implements** an interface.
- ❖ In order to implement an interface, a class must first declare the interface in an **implements** clause, and then it must provide an implementation for all of the abstract methods of the interface.
- ❖ A class can “**implements**” more than one **interfaces**.
- ❖ More discussions on “**interfaces**” later in the course.

# Interfaces in Java: Example



```
public interface Drawable {
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(Graphics g);
}

public class DrawableRectangle
    extends Rectangle
    implements Drawable {
    private Color c;
    private double x, y;

    .....
    .....

    // Here are implementations of the
    // methods in Drawable
    // we also inherit all public methods
    // of Rectangle

    public void setColor(Color c) { this.c = c; }
    public void setPosition(double x, double y) {
        this.x = x; this.y = y; }
    public void draw(Graphics g) {
        g.drawRect(x,y,w,h,c); }
}
```

## Using Interfaces: Example

- ❖ When a class **implements** an interface, instance of that class can also be **assigned to** variables of the interface type.

```
Shape[] shapes = new Shape[3];
Drawable[] drawables = new Drawable[3];

DrawableCircle dc = new DrawableCircle(1.1);
DrawableSquare ds = new DrawableSquare(2.5);
DrawableRectangle dr = new DrawableRectangle(2.3,
4.5);

// The shapes can be assigned to both arrays
shapes[0] = dc; drawables[0] = dc;
shapes[1] = ds; drawables[1] = ds;
shapes[2] = dr; drawables[2] = dr;

// We can invoke abstract method
// in Drawable and Shapes

double total_area = 0;
for(int i=0; i< shapes.length; i++) {

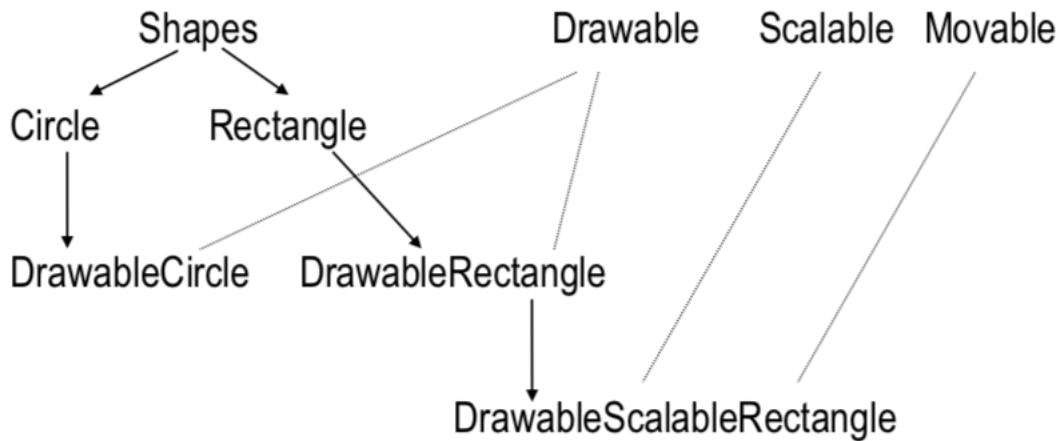
    total_area += shapes[i].area();

    drawables[i].setPosition(i*10.0, i*10.0);

    // assume that graphic area 'g' is
    // defined somewhere
    drawables[i].draw(g);
}
```

# Implementing Multiple Interfaces

A class can **implements** more than one interfaces. For example,



```
public class DrawableScalableRectangle
    extends DrawableRectangle
    implements Movable, Scalable {

    // methods go here ...

}
```

# Extending Interfaces

- ❖ Interfaces can have **sub-interfaces**, just like classes can have subclasses.
- ❖ A sub-interface **inherits all** the abstract methods and constants of its super-interface, and may define new abstract methods and constants.
- ❖ Interfaces **can extend** more than one interface at a time. For example,

```
public interface Transformable
    extends Scalable, Rotable, Reflectable {}

public interface DrawingObject
    extends Drawable, Transformable {}

public class Shape implements DrawingObject {
    ...
}
```

**COMP2511**

# **Object Oriented Programming (OOP) in Java**

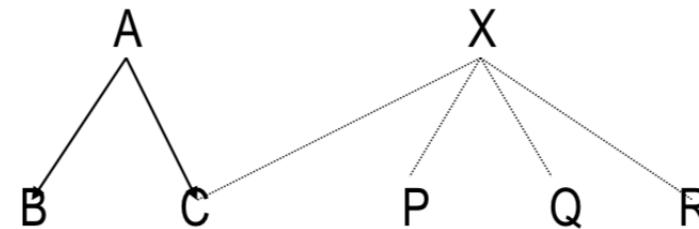
Prepared by

Dr. Ashesh Mahidadia

# OOP in Java

- ❖ Object Oriented Programming (OOP)
  - ❖ Inheritance in OOP
  - ❖ Introduction to Classes and Objects
  - ❖ Subclasses and Inheritance
  - ❖ Abstract Classes
  - ❖ Single Inheritance versus Multiple Inheritance
  - ❖ Interfaces
- ❖ Method Forwarding (Has-a relationship)
  - ❖ Method Overriding (Polymorphism)
  - ❖ Method Overloading
  - ❖ Constructors

# Method Forwarding



- ❖ Suppose class C extends class A, and also implements interface X.
- ❖ As all the methods defined in interface X are abstract, class C needs to implement all these methods.
- ❖ However, there are three implementations of X (in P,Q,R).
- ❖ In class C, we may want to use one of these implementations, that means, we may want to use some or all methods implemented in P, Q or R.
- ❖ Say, we want to use methods implemented in P. We can do this by creating an object of type class P in class C, and through this object access all the methods implemented in P.
- ❖ Note that, in class C, we do need to provide required stubs for all the methods in the interface X. In the body of the methods we may simply call methods of class P via the object of class P.
- ❖ This approach is also known as **method forwarding**.

# Methods Overriding (Polymorphism)

- ❖ When a class defines a method using the **same** name, return type, and by the number, type, and position of its arguments as a method in its *superclass*, the method in the class **overrides** the method in the *superclass*.
- ❖ If a method is invoked for an object of the class, it's the **new definition** of the method that is called, and **not** the superclass's **old definition**.

## Polymorphism

- An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is usually called **polymorphism**.

# Methods Overriding: Example

In the example below,

- ❖ if `p` is an instance of class `B`,  
`p.f()` refers to `f()` in class `B`.
- ❖ However, if `p` is an instance of class `A`,  
`p.f()` refers to `f()` in class `A`.

The example also shows how to refer to the `overridden` method using `super` keyword.

```
class A {  
    int i = 1;  
    int f() { return i; }  
}  
  
class B extends A {  
    int i;                                // shadows i from A  
    int f() {  
        i = super.i + 1;                  // retrieves i from A  
        return super.f() + i;             // invokes f() from A  
    }  
}
```

# Methods Overriding: Example

Suppose class C is a subclass of class B, and class B is a subclass of class A.

Class A and class C both define method `f()`.

From class C, we can refer to the overridden method by,

`super.f()`

This is because class B inherits method `f()` from class A.

However,

- ❖ if `all the three` classes define `f()`, then calling `super.f()` in class C invokes class B's definition of the method.
- ❖ Importantly, in this case, there is `no way` to invoke `A.f()` from within class C.
- ❖ Note that `super.super.f()` is `NOT legal` Java syntax.

# Method Overloading

Defining methods with the **same name** and **different** argument or return types is called *method overloading*.

In Java,

- ❖ a method is distinguished by its **method signature** - its name, return type, and by the number, type, and position of its arguments

For example,

```
double add(int, int)
double add(int, double)
double add(float, int)
double add(int, int, int)
double add(int, double, int)
```

# Data Hiding and Encapsulation

We can **hide** the **data** within the class and make it available only through the methods.

This can help in maintaining the consistency of the data for an object, that means the state of an object.

## Visibility Modifiers

Java provides five access modifiers (for variables/methods/classes),

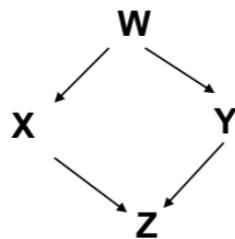
- ❖ **public** - visible to the world
- ❖ **private** - visible to the class only
- ❖ **protected** - visible to the package and all subclasses
- ❖ **No modifier (default)** - visible to the package

# Constructors

- ❖ Good practice to **define** the required constructors for **all** classes.
- ❖ If a constructor is **not defined** in a class,
  - **no-argument** constructor is **implicitly inserted**.
  - this no-argument constructor invokes the **superclass's no-argument** constructor.
  - if the parent class (superclass) doesn't have a visible constructor with no-argument, it results in a compilation **error**.
- ❖ If the **first statement** in a constructor is **not** a call to **super()** or **this()**, a call to **super ()** is **implicitly** inserted.
- ❖ If a constructor is **defined** with **one or more arguments**, **no-argument** constructor is **not inserted** in that class.
- ❖ A class can have **multiple** constructors, with **different signatures**.
- ❖ The word “**this**” can be used to call another constructor in the same class.

# Diamond Inheritance Problem: A Possible Solution

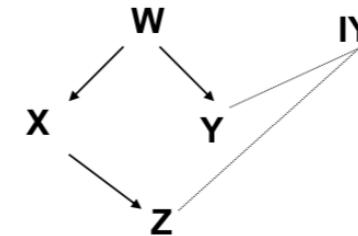
Using **multiple inheritance** (in C++):



we achieve the following:

- In class **Z**, we can use methods and variables defined in **X**, **W** **and** **Y**.
- Objects of classes **Z** and **Y** can be assigned to variables of **type Y**.
- and more ...

Using **single inheritance** in Java:



```
class W {}  
interface IY {}  
class X extends W {}  
class Y implements IY {}  
class Z extends X implements IY {}
```

we achieve the following:

- In class **Z**, we can use methods and variables defined in **X** and **W**. In class **Z**, if we want to use methods implemented in class **Y**, we can use **method forwarding** technique. That means, in class **Z**, we can create an object of type class **Y**, and via this object we can access (in class **Z**) all the methods defined in class **Y**.
- Objects of classes **Z** and **Y** can be assigned to variables of **type IY** (instead of **Y**).
- and more ....

# Some References to Java Tutorials

- ❖ <https://docs.oracle.com/javase/tutorial/>
- ❖ <https://www.w3schools.com/java/default.asp>
- ❖ <https://www.tutorialspoint.com/java/index.htm>

**COMP2511**

# **Object Oriented Programming (OOP) in Java**

Prepared by

Dr. Ashesh Mahidadia

# OOP in Java

- ❖ Object Oriented Programming (OOP)
- ❖ Inheritance in OOP
- ❖ Introduction to Classes and Objects
- ❖ Subclasses and Inheritance
- ❖ Abstract Classes
- ❖ Single Inheritance versus Multiple Inheritance
- ❖ Interfaces
- ❖ Method Forwarding (Has-a relationship)
- ❖ Method Overriding (Polymorphism)
- ❖ Method Overloading
- ❖ Constructors

# Subclasses and Inheritance: *First Approach*

We want to implement *GraphicalCircle*.

This can be achieved in at least **3 different ways**.

## First Approach:

- ❖ In this approach we are creating the **new separate class** for *GraphicalCircle* and **re-writing** the code already available in the class *Circle*.
- ❖ For example, we re-write the methods *area* and *circumference*.
- ❖ Hence, this approach is NOT elegant, in fact its the **worst** possible solution.  
Note again, its the **worst** possible solution!

```
// The class of graphical circles

public class GraphicalCircle {
    int x, y;
    int r;
    Color outline, fill;

    public double circumference() {
        return 2 * 3.14159 * r ;
    }
    public double area () {
        return 3.14159 * r * r ;
    }

    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }
}
```

# Subclasses and Inheritance: Second Approach

- ❖ We want to implement *GraphicalCircle* so that it can make use of the code in the class *Circle*.
- ❖ This approach uses “**has-a**” relationship.
- ❖ That means, a *GraphicalCircle* has a (mathematical) *Circle*.
- ❖ It uses methods from the class *Circle* (*area* and *circumference*) to define some of the new methods.
- ❖ This technique is also known as **method forwarding**.

```
public class GraphicalCircle2 {  
    // here's the math circle  
    Circle c;  
    // The new graphics variables go here  
    Color outline, fill;  
  
    // Very simple constructor  
    public GraphicalCircle2() {  
        c = new Circle();  
        this.outline = Color.black;  
        this.fill = Color.white;  
    }  
  
    // Another simple constructor  
    public GraphicalCircle2(int x, int y, int r,  
                           Color o, Color f) {  
        c = new Circle(x, y, r);  
        this.outline = o;  
        this.fill = f;  
    }  
  
    // draw method , using object 'c'  
    public void draw(Graphics g) {  
        g.setColor(outline);  
        g.drawOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);  
        g.setColor(fill);  
        g.fillOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);  
    }  
}
```

# Subclasses and Inheritance:

## Third Approach - Extending a Class

- ❖ We can say that *GraphicalCircle* **is-a** *Circle*.
- ❖ Hence, we can define *GraphicalCircle* as an **extension**, or *subclass* of Circle.
- ❖ The subclass *GraphicalCircle* **inherits** all the variables and methods of its superclass *Circle*.

```
import java.awt.Color;
import java.awt.Graphics;

public class GraphicalCircle extends Circle {

    Color outline, fill;
    public GraphicalCircle(){
        super();
        this.outline = Color.black;
        this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle(int x, int y,
                          int r, Color o, Color f){
        super(x, y, r);
        this.outline = o; this.fill = f;
    }

    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }
}
```

# Subclasses and Inheritance: Example

We can assign an instance of *GraphicCircle* to a *Circle* variable. For example,

```
GraphicCircle gc = new GraphicCircle();  
...  
double area = gc.area();  
...  
Circle c = gc;  
// we cannot call draw method for "c".
```

## Important:

- ❖ Considering the variable “c” is of type *Circle*,
- ❖ we can only access attributes and methods available in the class *Circle*.
- ❖ we **cannot** call *draw* method for “c”.

# Super classes, Objects, and the Class Hierarchy

- ❖ Every class has a superclass.
- ❖ If we don't define the superclass, by default, the superclass is the class **Object**.

## **Object** Class :

- ❖ Its the only class that does not have a superclass.
- ❖ The methods defined by **Object** can be called by any Java object (instance).
- ❖ Often we need to **override** the following methods:
  - **toString()**
    - read the API at [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString\(\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString())
  - **equals()**
    - read the API at  
[https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>equals\(java.lang.Object\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>equals(java.lang.Object))
  - **hashCode()**

# Abstract Classes

Using **abstract** classes,

- ❖ we can declare classes that define **only** part of an implementation,
- ❖ leaving extended classes to provide specific implementation of some or all the methods.

The benefit of an **abstract** class

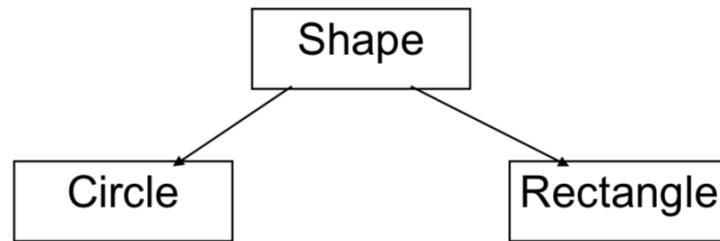
- ❖ is that methods may be declared such that the programmer knows **the interface definition** of an object,
- ❖ however, methods can be **implemented differently** in different subclasses of the abstract class.

# Abstract Classes

Some rules about abstract classes:

- ❖ An abstract class is a class that is **declared abstract**.
- ❖ If a class **includes abstract methods**, then the class itself must be declared abstract.
- ❖ An abstract class **cannot be instantiated**.
- ❖ A subclass of an abstract class can be instantiated if it overrides each of the abstract methods of its superclass and provides **an implementation** for **all** of them.
- ❖ If a subclass of an abstract class **does not implement** all the abstract methods it inherits, that subclass is itself abstract.

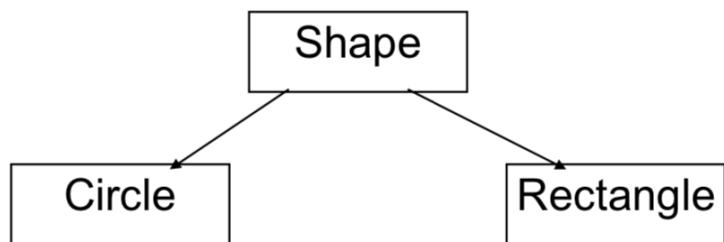
# Abstract Class: Example



```
public abstract class Shape {  
    public abstract double area();  
    public abstract double circumference();  
}
```

```
public class Circle extends Shape {  
  
    protected static final double pi = 3.14159;  
    protected int x, y;  
    protected int r;  
  
    // Very simple constructor  
    public Circle(){  
        this.x = 1;  
        this.y = 1;  
        this.r = 1;  
    }  
    // Another simple constructor  
    public Circle(int x, int y, int r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    /**  
     * Below, methods that return the circumference  
     * area of the circle  
     */  
    public double circumference( ) {  
        return 2 * pi * r ;  
    }  
    public double area ( ) {  
        return pi * r * r ;  
    }  
}
```

# Abstract Class: Example



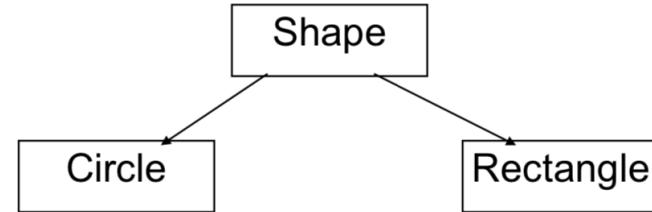
```
public abstract class Shape {  
    public abstract double area();  
    public abstract double circumference();  
}
```

```
public class Rectangle extends Shape {  
  
    protected double width, height;  
  
    public Rectangle() {  
        width = 1.0;  
        height = 1.0;  
    }  
  
    public Rectangle(double w, double h) {  
        this.width = w;  
        this.height = h;  
    }  
  
    public double area(){  
        return width*height;  
    }  
  
    public double circumference() {  
        return 2*(width + height);  
    }  
}
```

# Abstract Class: Example

Some points to note:

- ❖ As **Shape** is an abstract class, we cannot instantiate it.
- ❖ Instantiations of **Circle** and **Rectangle** can be assigned to variables of **Shape**.  
**No cast** is necessary
- ❖ In other words, subclasses of **Shape** can be assigned to elements of an array of **Shape**.  
**No cast** is necessary.
- ❖ We can invoke **area()** and **circumference()** methods for **Shape** objects.



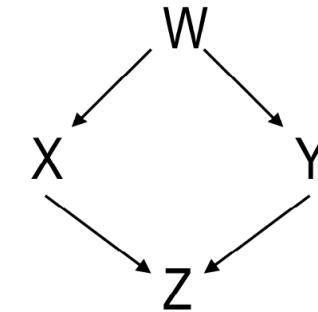
We can now write code like this:

```
// create an array to hold shapes
Shape[] shapes = new Shape[4];
shapes[0] = new Circle(4, 6, 2);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
shapes[3] = new GraphicalCircle(1, 1, 6,
    Color.green, Color.yellow);

double total_area = 0;
for(int i = 0; i < shapes.length; i++) {
    // compute the area of the shapes
    total_area += shapes[i].area();
}
```

# Single Inheritance versus Multiple Inheritance

- In Java, a new class can extend exactly one superclass - a model known as *single inheritance*.
- Some object-oriented languages employ *multiple inheritance*, where a new class can have two or more *super classes*.
- In multiple inheritance, **problems** arise when a superclass's behaviour is **inherited in two/multiple ways**.
- Single inheritance precludes some useful and correct designs.
- In Java, **interface** in the class hierarchy can be used to add multiple inheritance, more discussions on this later.



Diamond inheritance  
problem

# COMP 2511

# Design Principles

# The One Constant in software analysis and design

- What is the one thing you can always count on in writing software? - **Change**

# Building Good Software

Is all about:

- Making sure your software does what the customer wants it to do – use-case diagram, feature list, prioritise them
- Applying OO design principles to:
  - To ensure the system is flexible and extensible to accommodate changes in requirements
  - To strive for a maintainable, reusable, extensible design

- A change in requirements sometimes reveals problems with your system that you did not even know that they existed
- Remember, change is constant and your system should **continually improve** when you add these changes.....else software rots

# Why does Software Rot?

We write **bad code**

Why do we write bad code ?

- Is it because **do not know** how to write better code?
- Requirements change in ways that original design did not anticipate
- But changes are not the issue –
  - changes require **refactoring** and refactoring requires **time** and we say **we do not have the time**
  - Business pressure - changes need to be made quickly – **“quick and dirty solutions”**
  - changes may be made by developers not familiar with the original design philosophy

Bad code, in fact **slows us down**

# Design Smells

When software rots  
it smells...

A design smell

- is a symptom of poor design
- often caused by violation of key design principles
- has structures in software that suggest refactoring

# Design Smells (1)

## Rigidity

- Tendency of the software being too difficult to change even in simple ways
- A single change causes a cascade of changes to other dependent modules

## Fragility

- Tendency of the software to break in many places when a single change is made

Rigidity and fragility complement each other – aim towards minimal impact, when a new feature or change is needed

# Design Smells (2)

## Immobility

- Design is hard to reuse
- Design has parts that could be useful to other systems, but the effort needed and risk in disentangling the system is too high

## Viscosity

- Software viscosity – changes are easier to implement through ‘hacks’ over ‘design preserving methods’
- Environment viscosity – development environment is slow and in-efficient

## Opacity

- Tendency of a module to be difficult to understand
- Code must be written in a clear and expressive manner

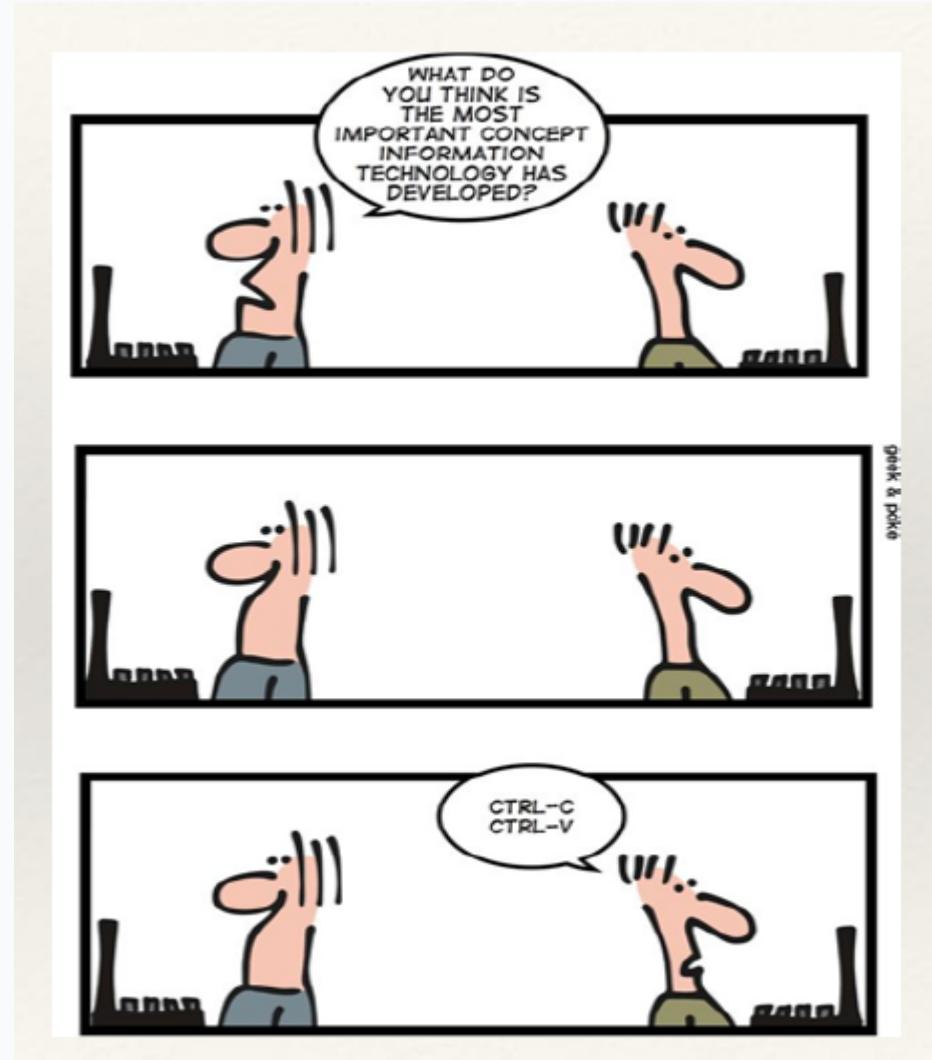
# Design Smells (3)

## Needless complexity

- Contains constructs that are not currently useful
- Developers ahead of requirements

## Needless repetition

- Design contains repeated structures that could potentially be unified under a single abstraction
- Bugs found in repeated units have to be fixed in every repetition



# Characteristics of Good Design

So, we know when our design smells...

But how do we measure if a software is well-designed?

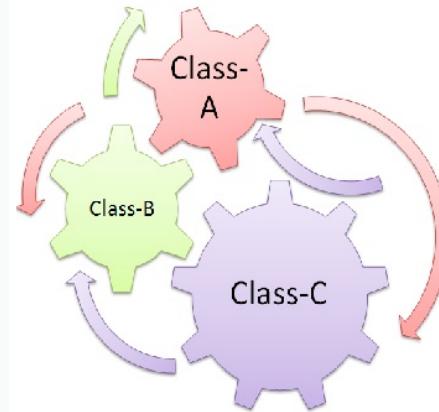
The design quality of software is characterised by

- Coupling
- Cohesion

Good software aims for building a system with **loose coupling** and **high cohesion** among its components so that software entities are:

- Extensible
- Reusable
- Maintainable
- Understandable
- Testable

# Coupling



- Is defined as the degree of **interdependence** between components or classes
- **High coupling** occurs when one component **A** depends on the internal workings of another component **B** and is affected by internal changes to component **B**
- High coupling leads to a complex system, with difficulties in maintenance and extension...eventual software rot
- Aim for **loosely coupled** classes - allows components to be used and modified independently of each other
- But “**zero-coupled**” classes are not usable – striking a balance is an art!

# Cohesion

- The degree to which all elements of a component or class or module work together as a functional unit
- **Highly cohesive** modules are:
  - much easier to maintain and less frequently changed and have higher probability of reusability
- Think about
  - How well the lines of code in a method or function work together to create a sense of purpose?
  - How well do the methods and properties of a class work together to define a class and its purpose?
  - How well do the classes fit together to create modules?
- Again, just like zero-coupling, do not put all the responsibility into a single class to avoid low cohesion!

And, applying **design principles** is the key to creating high-quality software

“Design principles are key notions considered fundamental to many different software design approaches and concepts.”

- SWEBOK v3 (2014)

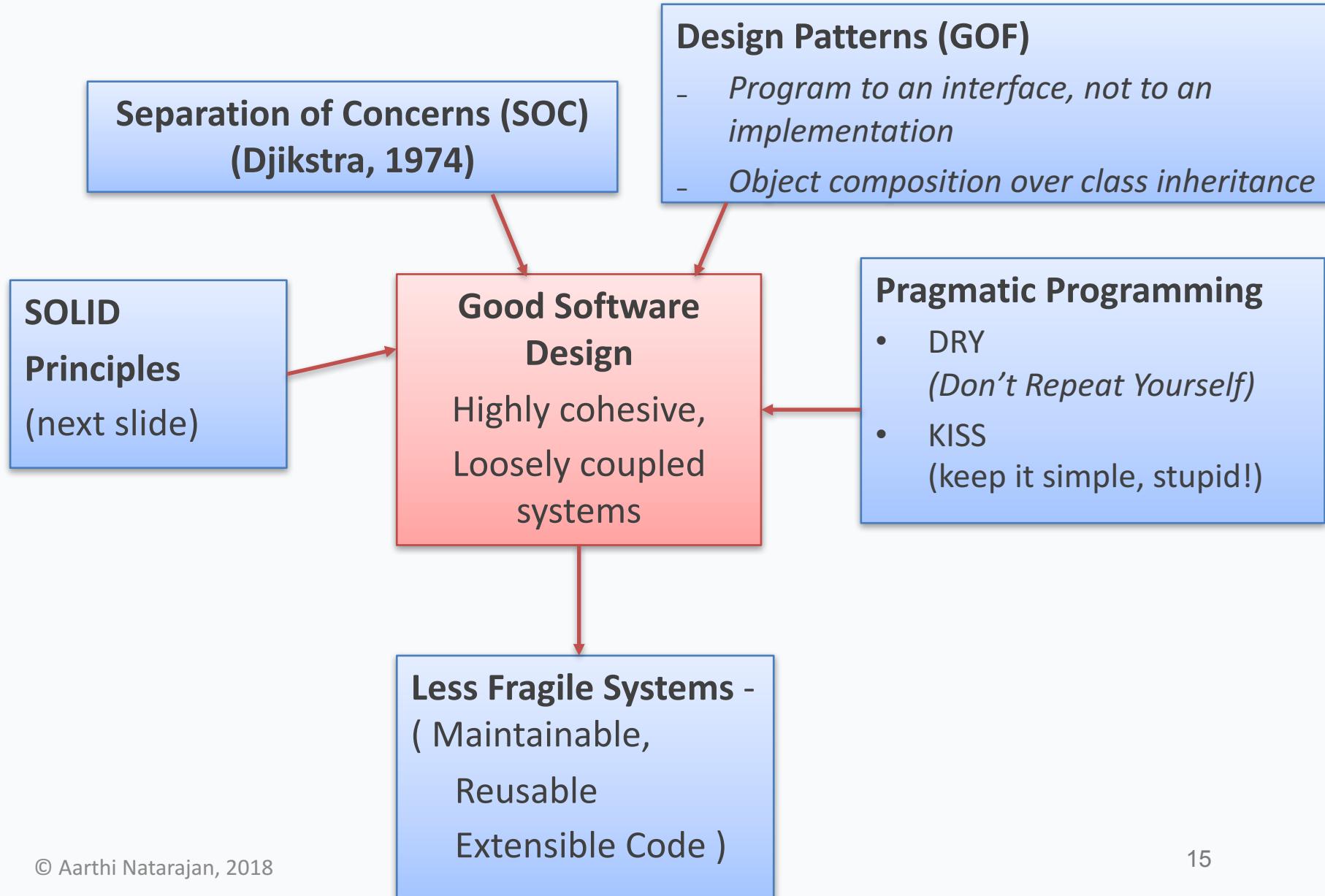
"The critical design tool for software development is a mind well educated in design principles"

- Craig Larman

# What is a “design principle”?

A basic tool or technique that can be applied to designing or writing code to make software more maintainable, flexible and extensible

# Several Design Principles...One Goal



# SOLID

- **Single responsibility principle:** A class should only have a single responsibility.
- **Open–closed principle:** Software entities should be open for extension, but closed for modification.
- **Liskov substitution principle:** Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Interface segregation principle:** Many client-specific interfaces are better than one general-purpose interface.
- **Dependency inversion principle:** One should "depend upon abstractions, [not] concretions."

# When to use design principles

- Design principles help eliminate design smells
- But, **don't apply** principles when there **no design smells**
- **Unconditionally conforming to a principle** (just because it is a principle is **a mistake**)
- Over-conformance leads to the design smell – **needless complexity**

# Design Principle #1

The Principle of Least Knowledge or Law of Demeter

# Design Principle #1

## The Principle of Least Knowledge (Law of Demeter) – Talk only to your friends

- Classes should know about and interact with as few classes as possible
- Reduce the interaction between objects to just a few close “friends”
- These friends are “immediate friends” or “local objects”
- Helps us to design “loosely coupled” systems so that changes to one part of the system does not cascade to other parts of the system
- The principle limits interaction through a set of rules

# The Principle of Least Knowledge (Law of Demeter)

A method in an object should only invoke methods of:

- The object itself
- The object passed in as a parameter to the method
- Objects instantiated within the method
- Any component objects
- And not those of objects returned by a method

Don't dig deep inside your friends for friends of friends of friends and get in deep conversations with them -- don't do

- e.g. `o.get(name).get(thing).remove(node)`

## Principle of Least Knowledge, Rule 1:

A method **M** in an object **O** can call on any other method within **O** itself

- This rule makes logical sense, a method encapsulated within a class can call any other method that is also encapsulated within the same class

```
public class M {  
    public void methodM() {  
        this.methodN();  
    }  
    public void methodN() {  
        // do something  
    }  
}
```

- Here `methodM()` calls `methodN()` as both are methods of the same class

## Principle of Least Knowledge, Rule 2:

A method **M** in an object **O** can call on any methods of parameters passed to the method **M**

- The parameter is local to the method, hence it can be called as a friend

```
public class O {  
  
    public void M(Friend f) {  
        // Invoking a method on a parameter passed to the method is  
        // legal  
        f.N();  
    }  
  
    public class Friend {  
  
        public void N() {  
            // do something  
        }  
    }  
}
```

## Principle of Least Knowledge, Rule 3:

A method **M** can call a method **N** of another object, if that object is instantiated within the method **M**

- The object instantiated is considered “local” just as the object passed in as a parameter

```
public class O {  
  
    public void M() {  
        Friend f = new Friend();  
        // Invoking a method on an object created within the  
        // method is legal  
        f.N();  
    }  
  
    public class Friend {  
        public void N() {  
            // do something  
        }  
    }  
}
```

## Principle of Least Knowledge, Rule 4:

Any method **M** in an object **O** can call on any methods of any type of object that is a direct component of **O**

- This means a method of a class can call methods of classes of its instance variables

```
public class O {  
  
    public Friend instanceVar = new Friend();  
  
    public void M4() {  
        // Any method can access the methods of the friend class  
        // F through the instance variable "instanceVar"  
        instanceVar.N();  
    }  
  
    public class Friend {  
        public void N() {  
            // do something  
        }  
    }  
}
```

# Well-designed Inheritance

# Design Principle #2

## LSP (Liskov Substitution Principle)

LSP is about well-designed inheritance

Barbara Liskov (1988) wrote:

*If for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$  then  $S$  is a subtype of  $T$ .*

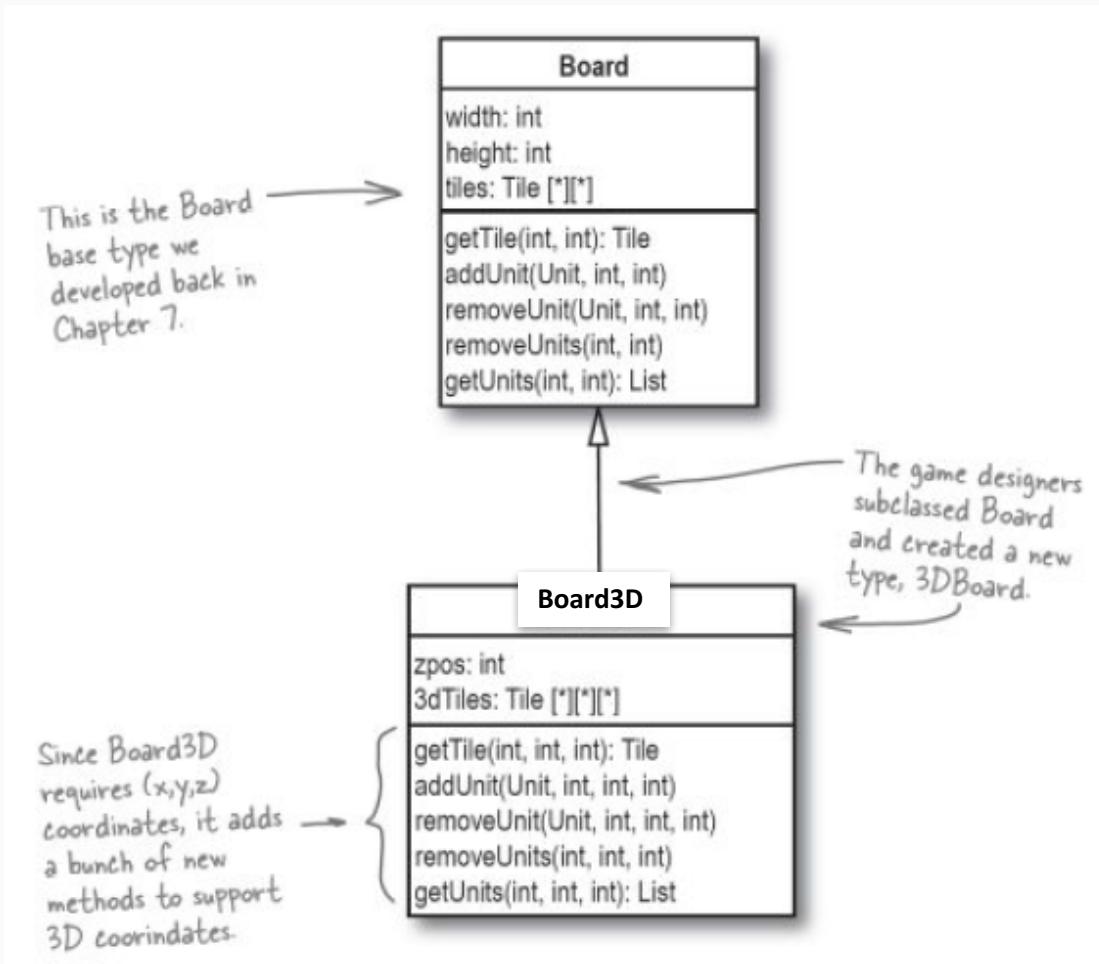
Bob wrote:

*subtypes must be substitutable for their base types*

Lecture demo: Square vs Rectangle

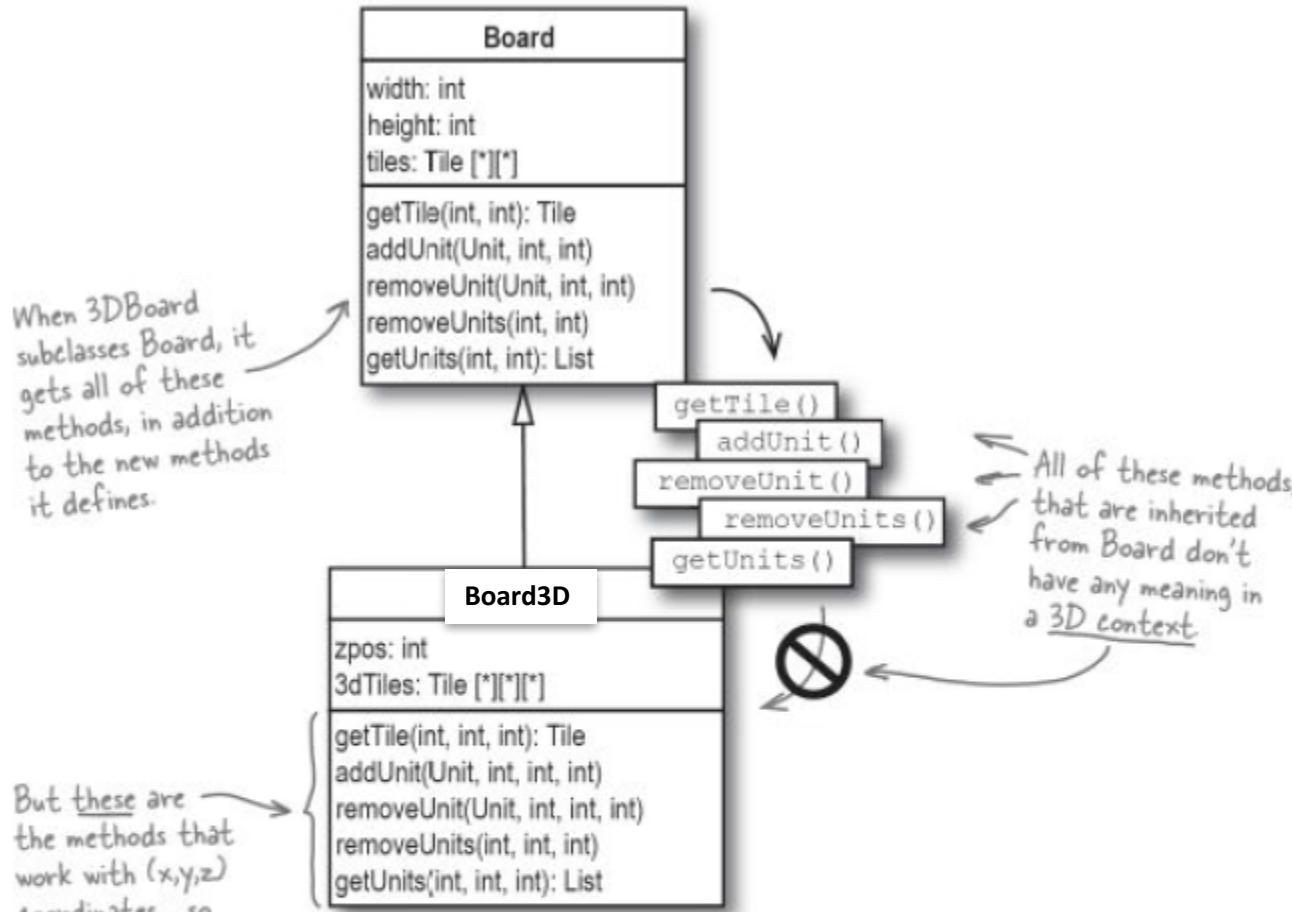
What is the problem with Square-Rectangle IS A relationship?

# Another LSP Example: A board game



LSP reveals hidden problems with the above inheritance structure

© Aarthi Natarajan, 2018



**The `Board3D` class is not substitutable for `Board`, because none of the methods on `Board` work correctly in a 3D environment. Calling a method like `getUnits(2, 5)` doesn't make sense for `3DBoard`. So this design violates the LSP.**

Even worse, we don't know what passing a coordinate like (2,5) even means to `3DBoard`. This is not a good use of inheritance.

# What are the issues?

LSP states that subtypes must be substitutable for their base types

```
Board board = new Board3D()
```

But, when you start to *use* the instance of Board3D like a Board, things go wrong

```
Artillery unit = board.getUnits(8,4)
```

*Board here is actually  
an instance of the sub-  
type Board3D*

*But, what does this  
method for a 3D board?*

Inheritance and LSP indicate that any method on Board should be able to use on a Board3D, and that Board3D can stand in for Board without any problems, so the above example clearly violates LSP

# Solve the problem without inheritance

So what options are there besides inheritance?

- Delegation – delegate the functionality to another class
- Composition – reuse behaviour using one or more classes with composition

***Design Principle: Favour composition over inheritance***

*If you favour delegation, composition over inheritance, your software will be more flexible, easier to maintain, extend*

# Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method
- The access level cannot be more restrictive than the overridden method's access level.  
E.g., if the super class method is declared **public** then the overriding method in the sub class cannot be either **private** or **protected**.
- A method declared **final** cannot be overridden.
- Constructors cannot be overridden.

# Can static methods be over-ridden?

Static methods can be defined in the sub-class with the same signature

- This is not overriding, as there is no run-time polymorphism
- The method in the derived class hides the method in the base class

Lecture demo...

# Rules for Method Overriding

## Covariance of return types in the overridden method

- The return type in the overridden method should be the same or a sub-type of the return type defined in the super-class
- This means that return types in the overridden method may be narrower than the parent return types

```
public class AnimalShelter {  
  
    public Animal getAnimalForAdoption() {  
        return null;  
    }  
  
    public void putAnimal(Animal someAnimal){  
    }  
}
```

```
public class CatShelter extends AnimalShelter {  
  
    /*  
     * @see AnimalShelter#getAnimalForAdoption()  
     */  
    @Override  
    public Cat getAnimalForAdoption() {  
  
        //Returning a narrower type than parent  
        return new Cat();  
    }  
}
```

# Rules for Method Overriding

What about Contra-variance of method arguments in the overridden method

Can arguments to methods in sub-class be wider than the arguments passed in the parent's method ?

```
public class CatShelter extends AnimalShelter {  
  
    /*  
     * @see AnimalShelter#putAnimal(Animal)  
     */  
    // Java sees this as an unrelated method.  
    // This is not actually overriding parent method  
    public void putAnimal(Object someAnimal) {  
        // do something  
    }  
}
```

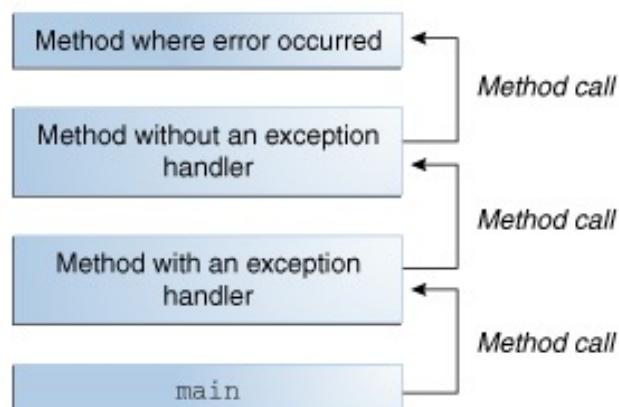
# COMP2511

# Exceptions in Java

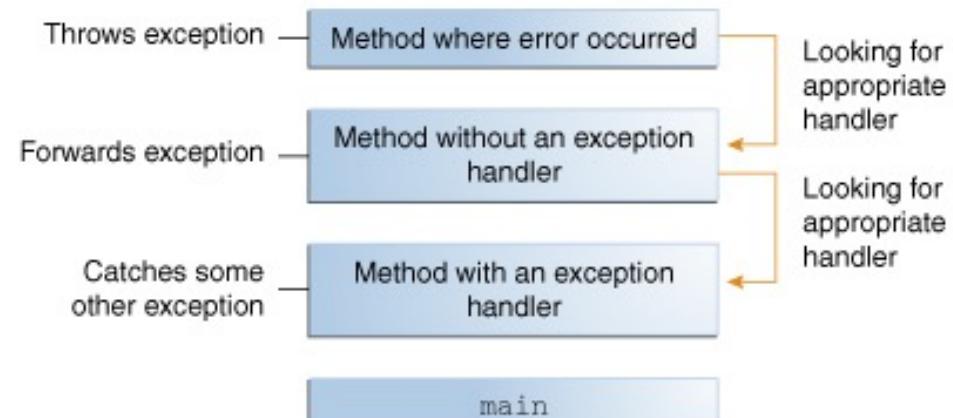
Prepared by  
Dr. Ashesh Mahidadia

# Exceptions in Java

- ❖ An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- ❖ When error occurs, an *exception object* is created and given to the runtime system, this is called **throwing** an exception.
- ❖ The runtime system searches the call stack for a method that contains a block of code that can **handle** the exception.
- ❖ The exception handler chosen is said to **catch** the exception.



The call stack.



Searching the call stack for  
the exception handler.

# Exceptions in Java

## The Three Kinds of Exceptions

- ❖ Checked exception (IOException, SQLException, etc.)
- ❖ Error (VirtualMachineError, OutOfMemoryError, etc.)
- ❖ *Runtime exception* (ArrayIndexOutOfBoundsException, ArithmeticException, etc.)

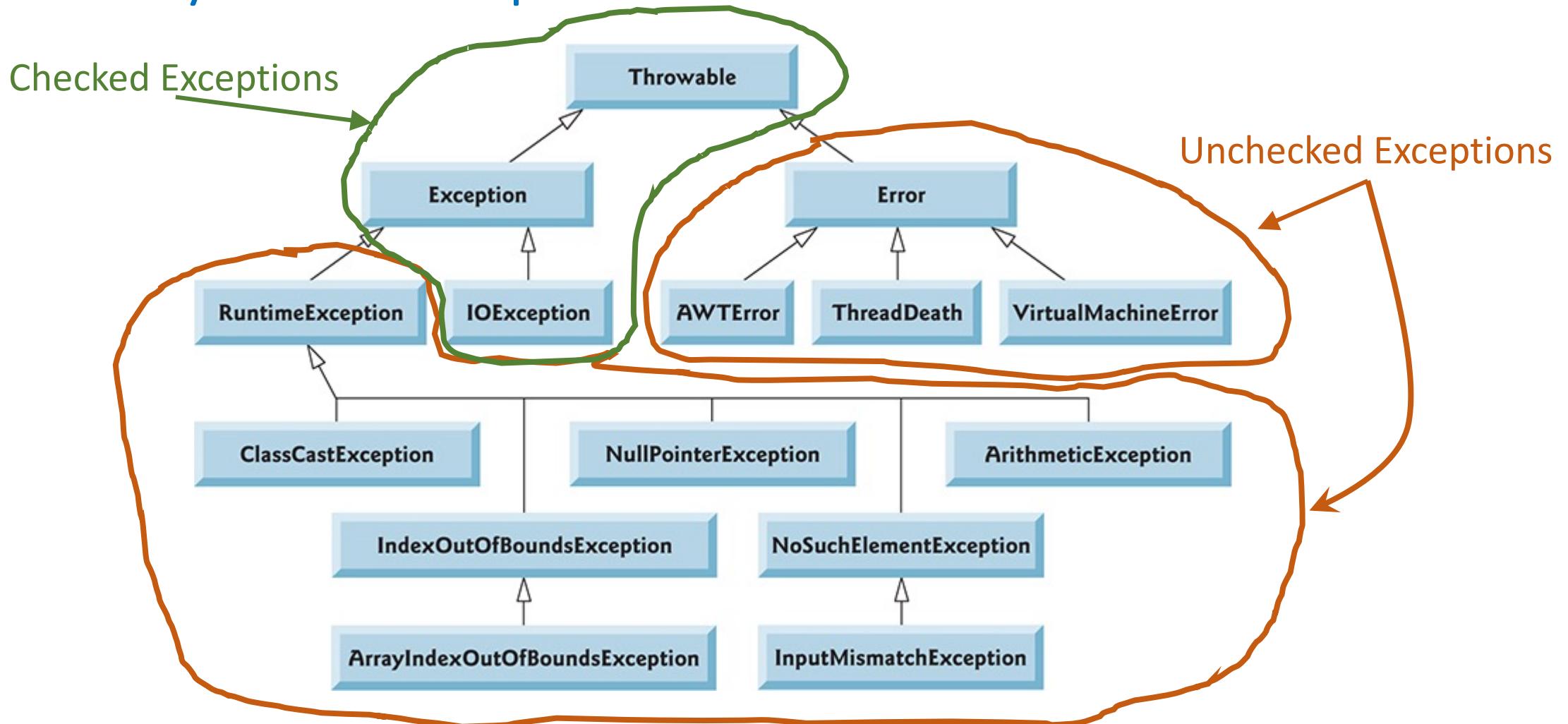
## Checked vs. Unchecked Exceptions

- ❖ An exception's type determines whether it's checked or unchecked.
- ❖ All classes that are subclasses of *RuntimeException* (typically caused by defects in your program's code) or *Error* (typically 'system' issues) are **unchecked** exceptions.
- ❖ All classes that inherit from class *Exception* but not directly or indirectly from class *RuntimeException* are considered to be **checked** exceptions.

# Exceptions in Java

- ❖ Good **introduction on Exceptions** at  
<https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>
- ❖ **Unchecked Exceptions** — The Controversy  
<https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>

# Hierarchy of Java Exceptions



From the book "Java How to Program, Early Objects", 11th Edition, by Paul J. Deitel; Harvey Deitel

# Example

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

The diagram illustrates the scope of exception handling blocks. A purple brace groups the entire try block, including its body and the catch block that follows it. A blue brace groups the two catch blocks. A green brace groups the finally block, which is executed regardless of whether an exception was caught or not.

# User Defined Exceptions in Java

- ❖ We can also create **user defined** exceptions.
- ❖ All exceptions must be a child of **Throwable**.
- ❖ A **checked** exception need to extend the **Exception** class,  
but **not** directly or indirectly from class **RuntimeException**.
- ❖ An **unchecked** exception (like a runtime exception) need to extend the  
**RuntimeException** class.

# User Defined / Custom Checked Exception

- Normally we define a *checked* exception, by extending the `Exception` class.

```
class MyException extends Exception {  
    public MyException(String message){  
        super( message );  
    }  
}
```

# User Defined / Custom Exceptions: A Simple Example

```
try {
    out = new PrintWriter(new FileWriter("myData.txt"));
    for(int i=0; i<SIZE; i++){

        int idx = i + 5;

        if(idx >= SIZE){
            throw new MyException("idx is out of index range!");
        }
        out.println(list.get(idx));
    }

}
catch(IOException e){
    System.out.println(" In writeln ....");
}
catch(MyException e){
    System.out.println(e.getMessage());
}
catch(Exception e){
    System.out.println(" In writeln, Exception ....");
}
```

# Exceptions in Inheritance

- ❖ If a subclass method **overrides** a superclass method,  
a subclass's **throws** clause can contain a subset of  
a superclass's **throws** clause.  
**It must **not** throw more exceptions!**
- ❖ Exceptions are **part of** an API documentation and **contract**.

# Demo: Exceptions in Java

Demo ...

# Assertions in Java

- An **assertion** is a statement in the Java that enables you to test your assumptions about your program. Assertions are **useful** for checking:
  - Preconditions, Post-conditions, and Class Invariants (DbC!)
  - Internal Invariants and Control-Flow Invariants
- You should **not** use assertions:
  - for argument checking in **public methods**.
  - to do any work that your application requires for correct operation.
- Evaluating assertions should **not** result in side effects.
- The following document shows how to use **assertions in Java** :  
<https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>

**Important:** for backward compatibility, by **default**, Java **disables** assertion validation feature.

It needs to be explicitly **enabled** using the following command line argument:

- **-enableassertions** command line argument, or
- **-ea** command line argument

# Assert : Example

```
/**  
 * Sets the refresh interval (which must correspond to a legal frame rate).  
 *  
 * @param  interval refresh interval in milliseconds.  
 */  
private void setRefreshInterval(int interval) {  
    // Confirm adherence to precondition in nonpublic method  
    assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;  
    ... // Set the refresh interval  
}
```



# Exceptions: Summary Points

- ❖ Consider your exception-handling and error-recovery strategy in the **design process**.
- ❖ Sometimes you can **prevent an exception** by validating data first.
- ❖ If an exception can be handled meaningfully in a method, the method should **catch** the exception **rather than declare** it.
- ❖ If a subclass method overrides a superclass method, a subclass's **throws** clause can contain a subset of a superclass's **throws** clause. It must not throw more exceptions!
- ❖ Programmers should **handle checked** exceptions.
- ❖ If **unchecked** exceptions are **expected**, you must handle them **gracefully**.
- ❖ Only the **first** matching **catch** is executed, so select your catching class(es) carefully.
- ❖ Exceptions are part of an API documentation and contract.
- ❖ Assertions can be used to check preconditions, post-conditions and invariants.

# COMP2511

## Generics and Collections in Java

Prepared by

Dr. Ashesh Mahidadia

# Generics in Java

## (Part 1)

# Generics in Java

Generics enable **types** (classes and interfaces) to be **parameters** when defining:

- classes,
- interfaces and
- methods.

## Benefits

- ❖ Removes *casting* and offers stronger type checks at compile time.
- ❖ Allows implementations of generic algorithms, that work on collections of different types, can be customized, and are type safe.
- ❖ Adds stability to your code by making more of your bugs detectable at compile time.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Without Generics

```
List<String> listG = new ArrayList<String>();
listG.add("hello");
String sg = listG.get(0); // no cast
```

With Generics

# Generic Types

- ❖ A generic type is a generic **class** or **interface** that is **parameterized** over types.
- ❖ A generic class is defined with the following format:  
`class name< T1, T2, ..., Tn > { /* ... */ }`
- ❖ The most commonly used type parameter names are:
  - ❖ E - Element (used extensively by the Java Collections Framework)
  - ❖ K - Key
  - ❖ N - Number
  - ❖ T - Type
  - ❖ V - Value
  - ❖ S,U,V etc. - 2nd, 3rd, 4th types
- ❖ For example,

```
Box<Integer> integerBox = new Box<Integer>();
```

OR

```
Box<Integer> integerBox = new Box<>();
```

```
public class Box {  
    private Object object;  
  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

```
/**  
 * Generic version of the Box class.  
 * @param <T> the type of the value being boxed  
 */  
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

# Multiple Type Parameters

- ❖ A generic class can have multiple type parameters.
- ❖ For example, the generic `OrderedPair` class, which implements the generic `Pair` interface
- ❖ Usage examples,

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}  
  
public class OrderedPair<K, V> implements Pair<K, V> {  
  
    private K key;  
    private V value;  
  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);  
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");  
....  
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);  
OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");  
....  
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

# Generic Methods

Generic methods are methods that **introduce** their **own type** parameters.

```
public class Util {  
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
        return p1.getKey().equals(p2.getKey()) &&  
               p1.getValue().equals(p2.getValue());  
    }  
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown above.

Generally, this can be left out and the compiler will **infer** the **type** that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.compare(p1, p2);
```

# Collections in Java

# Collections in Java

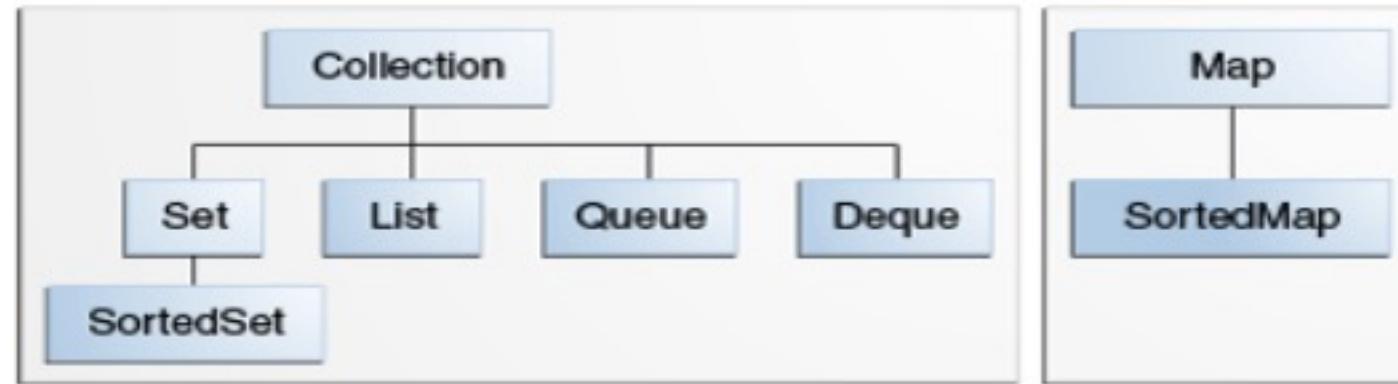
A **collections framework** is a unified architecture for representing and manipulating collections. A collection is simply an object that groups multiple elements into a single unit.

All collections frameworks contain the following:

- ❖ **Interfaces**: allows collections to be manipulated independently of the details of their representation.
- ❖ **Implementations**: concrete implementations of the collection interfaces.
- ❖ **Algorithms**: the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
  - The algorithms are said to be **polymorphic**: that is, the same method can be used on many different implementations of the appropriate collection interface.

## Core Collection Interfaces:

- ❖ The core collection interfaces encapsulate different types of collections
- ❖ The interfaces allow collections to be manipulated independently of the details of their representation.



The core collection interfaces.

# The Collection Interface

- ❖ A **Collection** represents a group of objects known as its elements.
- ❖ The **Collection interface** is used to pass around collections of objects where maximum generality is desired.
- ❖ For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument.
- ❖ The **Collection interface** contains methods that perform basic operations, such as
  - int **size()**,
  - boolean **isEmpty()**,
  - boolean **contains(Object element)**,
  - boolean **add(E element)**,
  - boolean **remove(Object element)**,
  - **Iterator<E> iterator()**,
  - ... ... **many more ...**

More at : <https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html>

# Collection Implementations

- ❖ The general purpose **implementations** are summarized in the following table:

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<a href="#">HashSet</a>		<a href="#">TreeSet</a>		<a href="#">LinkedHashSet</a>
List		<a href="#">ArrayList</a>		<a href="#">LinkedList</a>	
Deque		<a href="#">ArrayDeque</a>		<a href="#">LinkedList</a>	
Map	<a href="#">HashMap</a>		<a href="#">TreeMap</a>		<a href="#">LinkedHashMap</a>

Implemented Classes in the Java Collection,  
Read their APIs.

- ❖ Overview of the *Collections Framework* at the following page:

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>

# Wrappers for the Collection classes

- <https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html>

# Demo: Collections Framework

Demo ...

**End**

# COMP2511

## JUnit

Prepared by

Dr. Ashesh Mahidadia

# Software Testing

- ❖ Different types of testing:
  - Object Oriented Design document describes responsibilities of classes and methods (APIs) → **Unit Testing**
  - System Design Document → Integration Testing
  - Requirements Analysis Document → System Testing
  - Client Expectation → Acceptance Testing
- ❖ **Unit Testing** is also useful for **refactoring** tasks.
- ❖ In this course, we will focus on Unit testing.

# JUnit

- ❖ **JUnit** is a popular unit testing (open source) framework for testing Java programs.
- ❖ Most popular IDEs facilitate easy integration of Junit.
- ❖ Basic Junit Terminology:
  - **Test Case** – Java class containing test methods
  - **Test Method** – a method that executes the test code, annotated with @Test, in a Test Case
  - **Asserts** - asserts or assert statements check an expected result versus the actual result
  - **Test Suites** – collection of several Test Cases

## JUnit Example: assertEquals, assertTrue

```
14 /**
15  * Tests for Pineapple on Piazza
16  * @author Nick Patrikeos
17 */
18 public class PiazzaTest {
19
20     @Test
21     public void testExampleUsage() {
22         // Create a forum and make some posts!
23         PiazzaForum forum = new PiazzaForum("COMP2511");
24         assertEquals("COMP2511", forum.getName());
25
26         Thread funThread = forum.publish("The Real Question - Pineapple on Piazza", "Who likes pineapple on piazza?");
27
28         funThread.setTags(new String[] { "pizza", "coding", "social", "hobbies" });
29         assertTrue(
30             Arrays.equals(new String[] { "coding", "hobbies", "pizza", "social" }, funThread.getTags().toArray()));
31
32         funThread.publishPost("Yuck!");
33         funThread.publishPost("Yes, pineapple on pizza is the absolute best");
34         funThread.publishPost("I think you misspelled pizza btw");
35         funThread.publishPost("I'll just fix that lol");
36
37         assertEquals(5, funThread.getPosts().size());
38     }
39
40     @Test
41     public void testSearchByTag() {
42         PiazzaForum forum = new PiazzaForum("COMP2511");
43
44         Thread labThread = forum.publish("Lab 01", "How do I do the piazza exercise?");
45         Thread assignmentThread = forum.publish("Assignment", "Are we back in blackout?");
46         labThread.setTags(new String[] { "Java" });
47         assignmentThread.setTags(new String[] { "Java" });
48
49         List<Thread> searchResults = forum.searchByTag("Java");
50         assertEquals("Lab 01", searchResults.get(0).getTitle());
51         assertEquals("Assignment", searchResults.get(1).getTitle());
52     }
53 }
```

## JUnit Example: Exception

```
15 public class ArchaicFsTest {
16     @Test
17     public void testCdInvalidDirectory() {
18         ArchaicFileSystem fs = new ArchaicFileSystem();
19
20         // Try to change directory to an invalid one
21         assertThrows(UNSWNoSuchFileException.class, () -> {
22             fs.cd("/usr/bin/cool-stuff");
23         });
24     }
25
26     @Test
27     public void testCdValidDirectory() {
28         ArchaicFileSystem fs = new ArchaicFileSystem();
29
30         assertDoesNotThrow(() -> {
31             fs.mkdir("/usr/bin/cool-stuff", true, false);
32             fs.cd("/usr/bin/cool-stuff");
33         });
34     }
35
36     @Test
37     public void testCdAroundPaths() {
38         ArchaicFileSystem fs = new ArchaicFileSystem();
39
40         assertDoesNotThrow(() -> {
41             fs.mkdir("/usr/bin/cool-stuff", true, false);
42             fs.cd("/usr/bin/cool-stuff");
43             assertEquals("/usr/bin/cool-stuff", fs.cwd());
44             fs.cd("../");
45             assertEquals("/usr/bin", fs.cwd());
46             fs.cd("../bin/..");
47             assertEquals("/usr", fs.cwd());
48         });
49     }
50
51     @Test
52     public void testCreateFile() {
53         ArchaicFileSystem fs = new ArchaicFileSystem();
54
55         assertDoesNotThrow(() -> {
56             fs.writeToFile("test.txt", "My Content", EnumSet.of(FileWriteOptions.CREATE, FileWriteOptions.TRUNCATE));
57             assertEquals("My Content", fs.readFile("test.txt"));
58             fs.writeToFile("test.txt", "New Content", EnumSet.of(FileWriteOptions.TRUNCATE));
59             assertEquals("New Content", fs.readFile("test.txt"));
60         });
61     }
```

# Junit: Dynamic and parameterized tests

For information on **Dynamic and parameterized tests**,

- ❖ see the tutorial at <https://www.vogella.com/tutorials/JUnit/article.html>

For more information on JUnit, read the **user guide** at:

- ❖ <https://junit.org/junit5/docs/current/user-guide/>

**End**

# **COMP 2511**

# **Object Oriented Design &**

# **Programming**

Story so far,

Basic OO principles

Abstraction, Encapsulation, Inheritance, Polymorphism

Basic refactoring techniques

Extract method, Rename variable, Move Method, Replace  
Temp With Query

This week

OO design principles

- Encapsulate what varies
- Program to an interface, not an implementation
- Favour composition over inheritance

Design Patterns

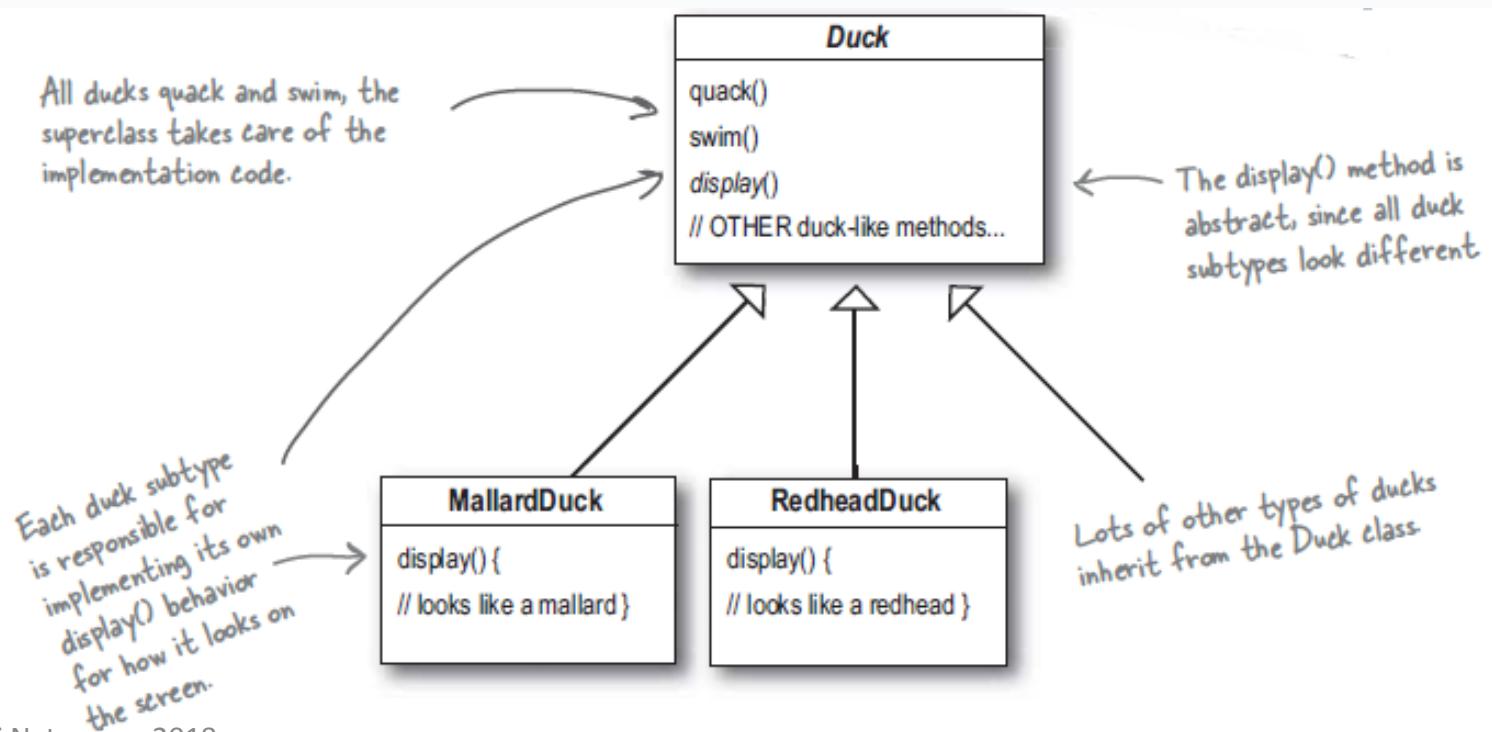
- Strategy and State Patterns

Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.



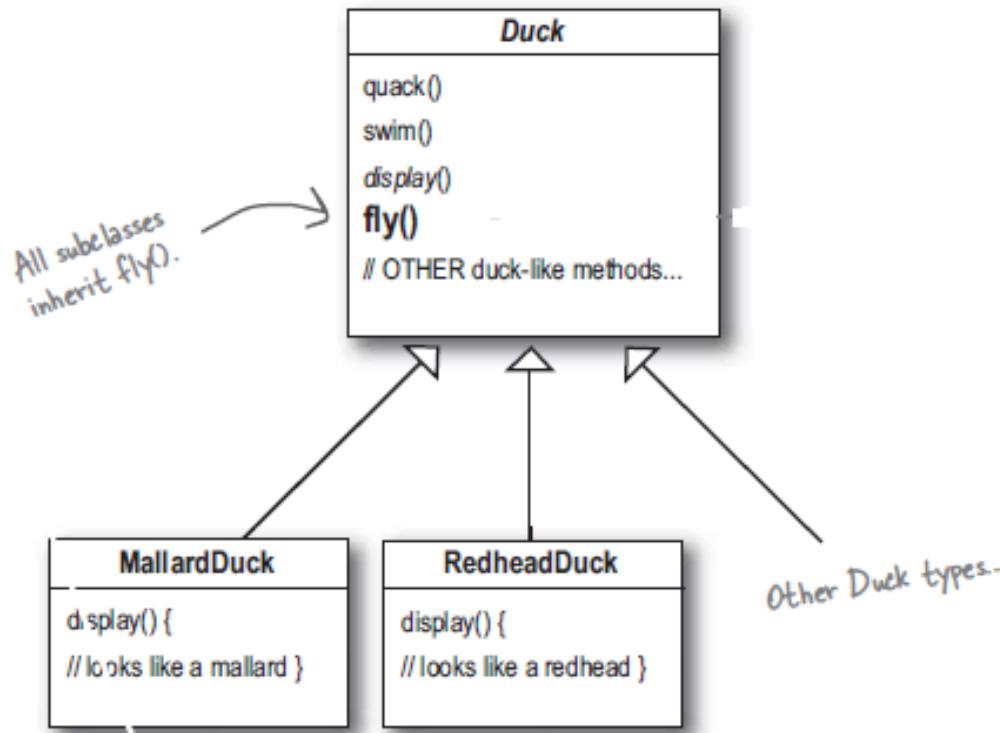
# A simple Duck Simulator App

- A game that shows a large variety of duck species swimming and making quacking sounds
- What we need is a base class Duck and sub-classes MallardDuck, RedheadDuck for the different duck species



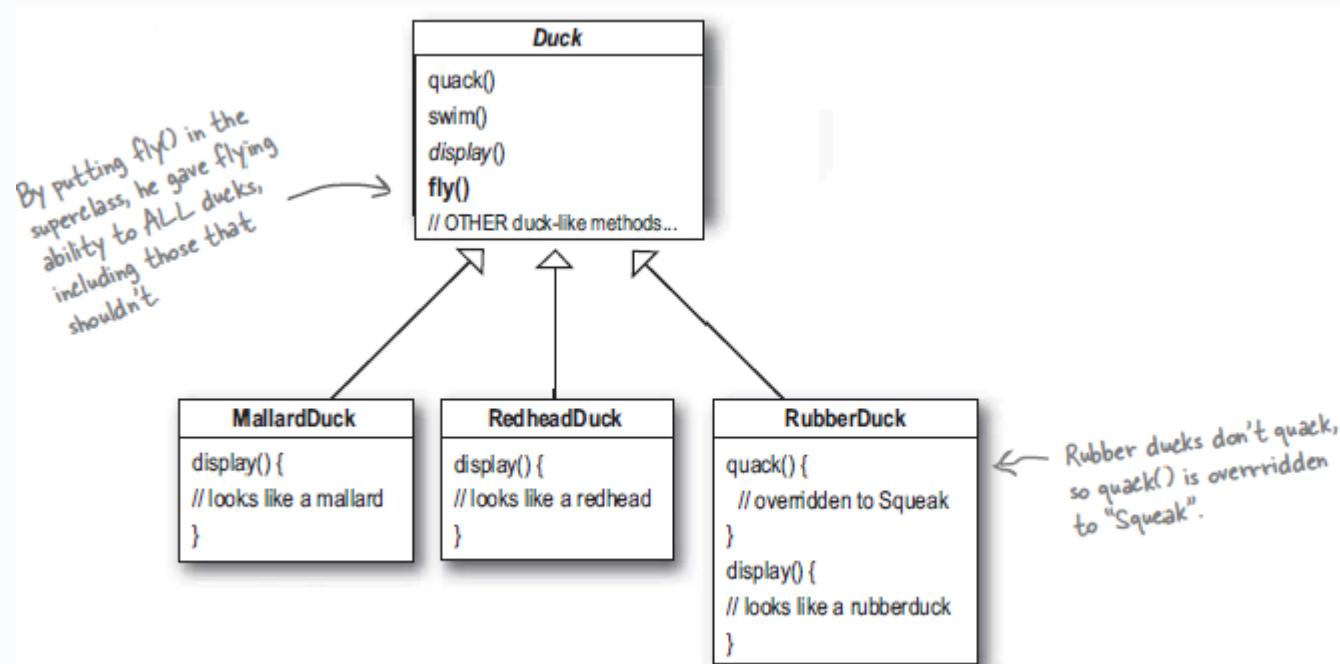
# But, now we need ducks that fly

- How hard can that be?
- Just add a `fly()` method to the class Duck and all sub-types will inherit it



# Design Flaw...

- A localised update to the code caused a non-local side effect (*flying rubber ducks*)
- What normally is thought of as a great use of inheritance for the purpose of “reuse” actually didn’t turn out so well, when it comes to maintenance

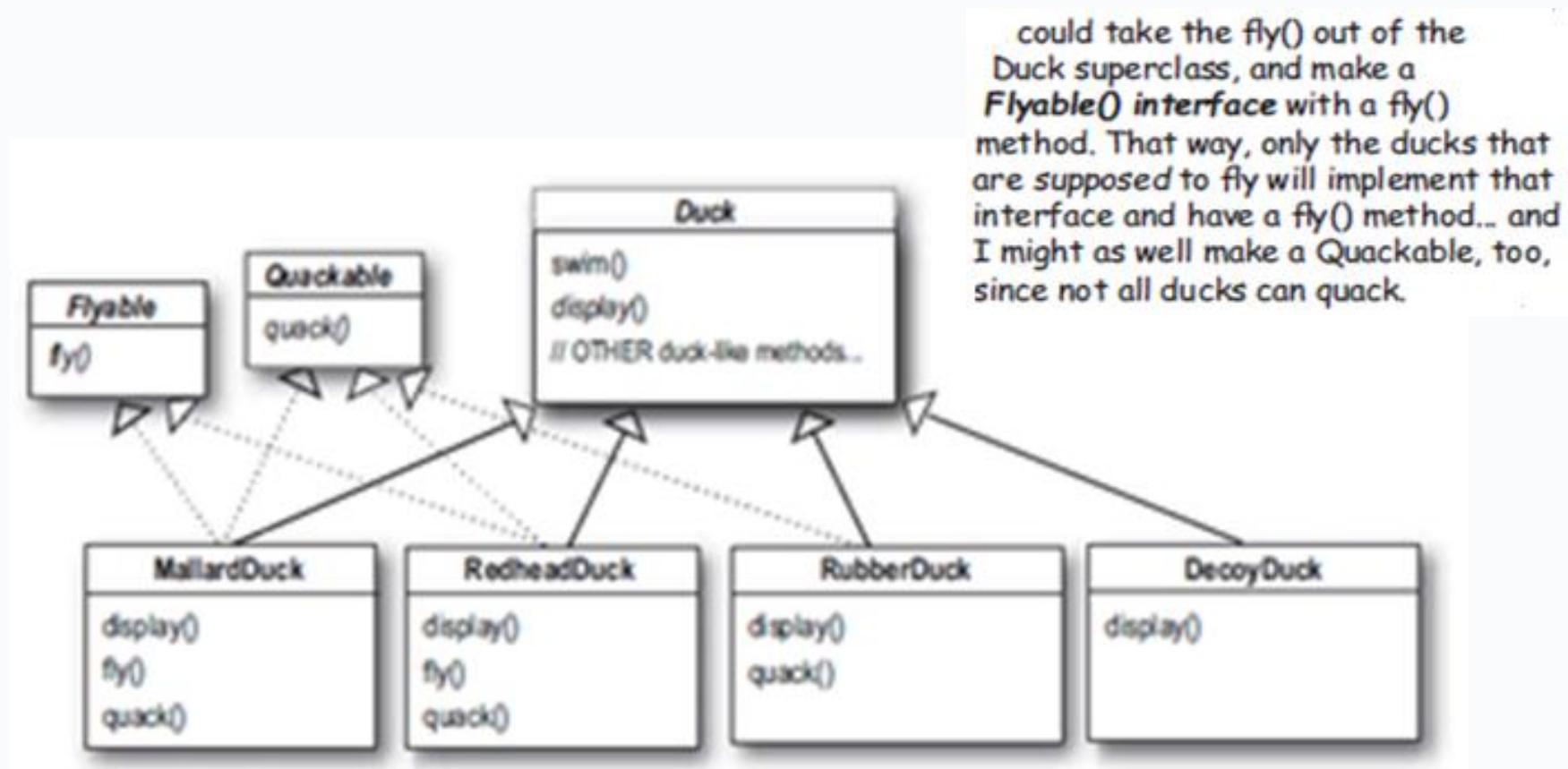


# Solution 1...

- Solution 1:
  - We could simply override `fly()` method to do nothing, just as the `quack()` method was overridden
  - But, what happens, when more different types of ducks were added that didn't quack or fly

# Solution 2...

- Solution 2:
  - Need a cleaner way, so that only some ducks fly and some quack. How about an interface?



## Solution 2...

- Using interfaces, all sub-classes that fly implement *flyable* interface and that quack implement *Quackable* interface
- But, completely destroys code reuse – every class must implement *fly()* method (perhaps not an issue in Java 8), but what if there are more than two kinds of flying behaviour among ducks that fly.
- Change every class where behaviour has changed? – **maintenance night-mare**
- Need a design pattern to come riding on a white horse and save the day.

- Is there a way to build software so that when we need to change it, we could do so with the least possible impact on the existing code?

# Solution

## Design Principle #3:

Identify aspects of your code that varies and “encapsulate” and separate it from code that stays the same, so that it won’t affect your real code.

- By separating what changes from what stays the same, the result is fewer unintended consequences from code changes and more flexibility in your software
- Another way to think about this principle: *take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.*

# So, let us pull out the duck behaviour from the duck class

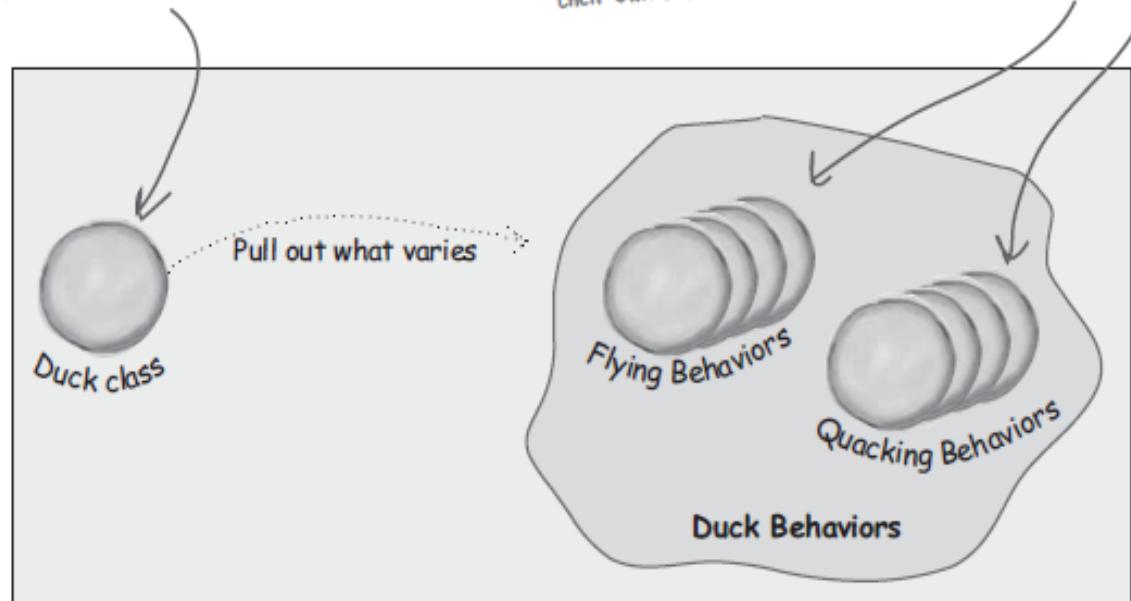
**We know that `fly()` and `quack()` are the parts of the Duck class that vary across ducks.**

**To separate these behaviors from the Duck class, we'll pull both methods out of the Duck class and create a new set of classes to represent each behavior.**

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.



- How are we going to design the set of classes that implement the *fly* and *quack* behaviour?

### Design Principle #4:

Program to a/an interface, not to an implementation

- Program to an interface, really means “program to a super-type” i.e., the declared type of the variable should be a super-type (abstract class or interface)
  - e.g., Dog d = new Dog(); d.bark(); // programming to an implementation
  - Animal a = new Dog(); a. makeSound(); // programming to an interface
- What we want is to exploit polymorphism by programming to a super-type so that actual run-time object isn’t locked into the code

# Programming to a super-type

Previously,

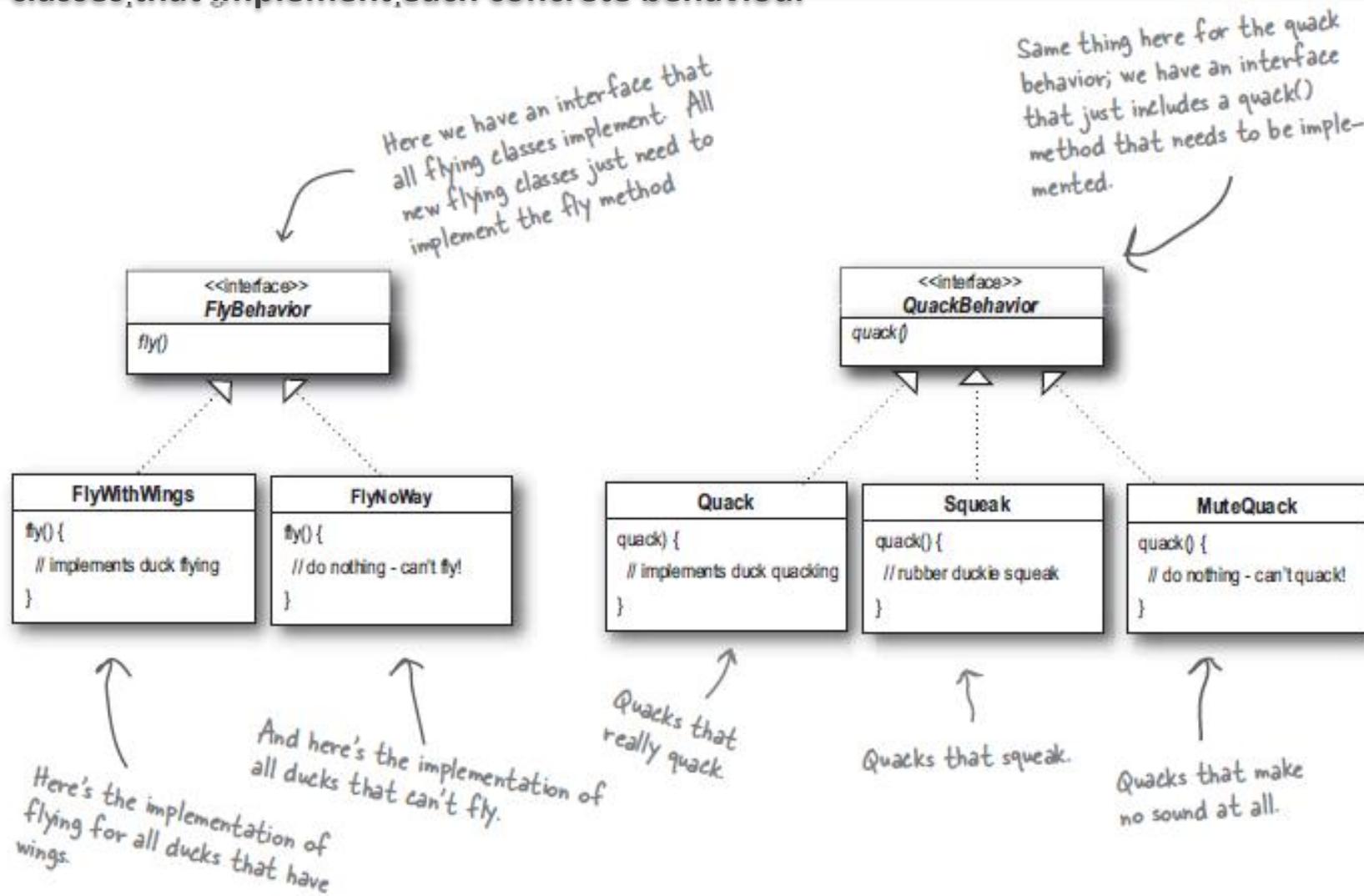
- A behaviour was locked into a **concrete implementation** in the Duck class or a **specialised implementation** in the sub-class
- Either way, we were locked into using a **specific implementation**
- There was no room for changing that behaviour

Now,

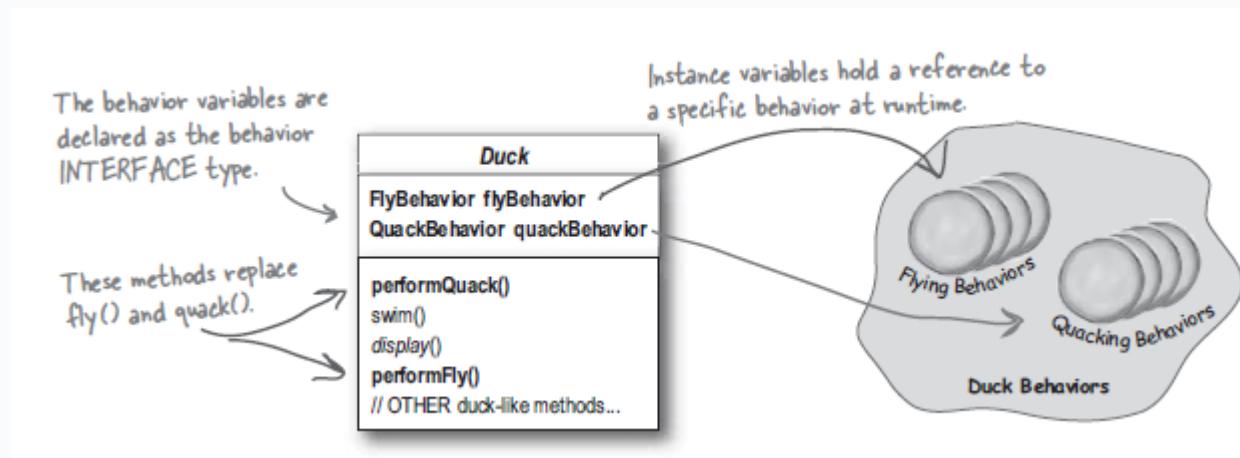
- Use an **interface** to represent the behaviour
- Implement a set of separate “**behaviour**” classes that implement this interface
- Associate a **duck instance** with a specific “**behaviour**” class
- The Duck classes won’t need to know any of the **implementation details** for their own behaviour

# Implementing the Behaviours

Here, there are two interfaces, FlyBehavior and QuackBehavior along with set of classes that implement each concrete behaviour



# Integrating Duck behaviour with the Duck instance



1. Define two instance variables in the Duck class
2. Implement **performQuack()** and **performFly()** that delegate the quacking and flying behavior to other objects
3. Assign the instance variables the right behavior

```
public class MallardDuck extends Duck {  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
}
```

Remember, **MallardDuck** inherits the **quackBehavior** and **flyBehavior** instance variables from class **Duck**.

```
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

A **MallardDuck** uses the **Quack** class to handle its quack, so when **performQuack** is called, the responsibility for the quack is delegated to the **Quack** object and we get a real quack.  
And it uses **FlyWithWings** as its **FlyBehavior** type.

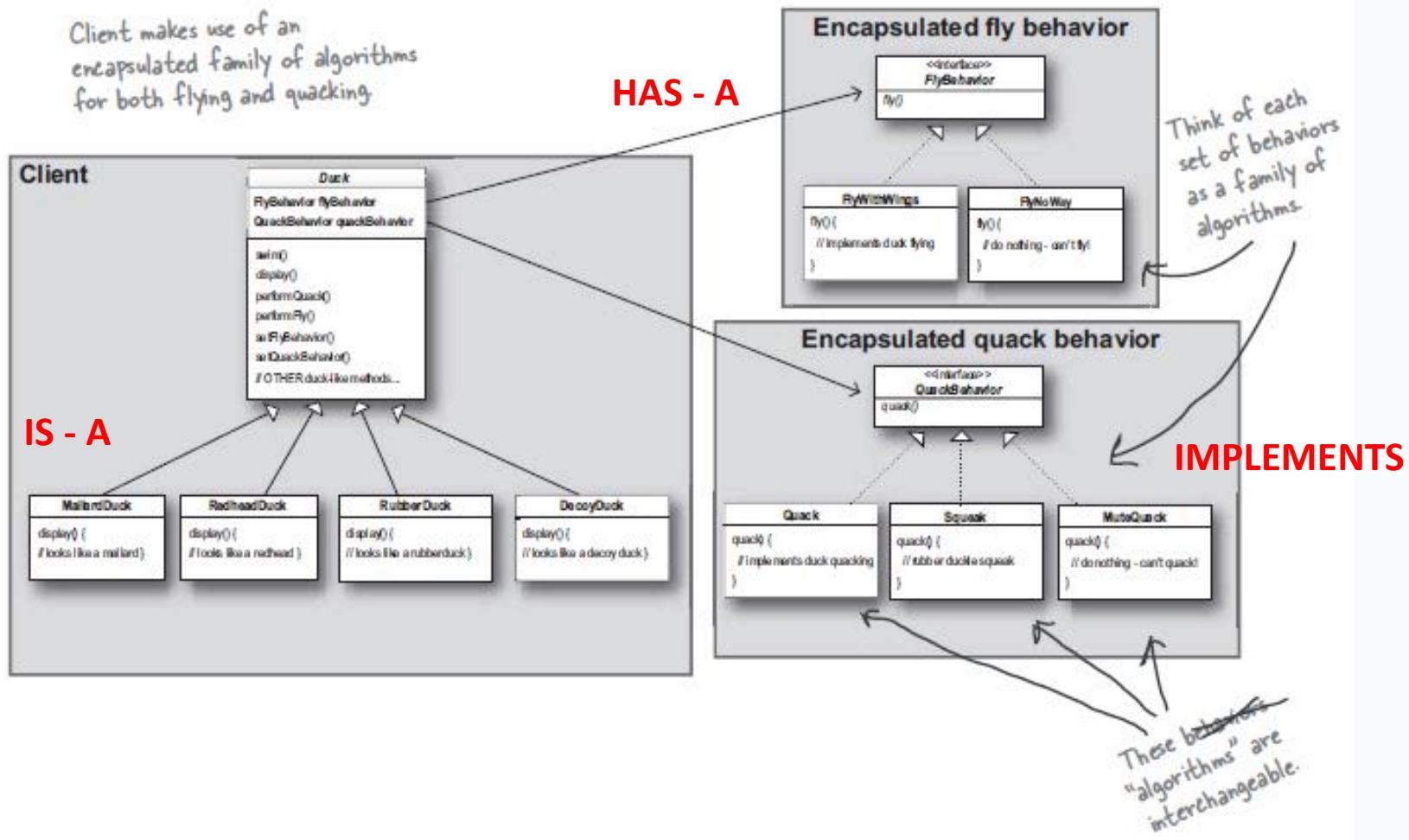
# Setting behaviour dynamically

1. Create two setter methods `setFlyBehavior()` and `setQuackBehavior()` inside class Duck
2. To change a duck's behavior at run-time, call the duck's setter method for that behavior

```
public abstract class Duck {  
    ...  
  
    // Add setter methods to change behavior at run-time  
    public void setFlyBehavior(FlyBehavior f) {  
        this.flyBehavior = f;  
    }  
    public void setQuackBehavior(QuackBehavior q) {  
        this.quackBehavior = q;  
    }  
}
```

# Our complete design

Think about the different relationships - IS-A, HAS-A, IMPLEMENTS



# HAS-A can be better than IS-A

- Each duck **has a** fly behaviour and **has a** quack behaviour. Haven't we heard of this relationship?

## COMPOSITION

- Instead of inheriting their behaviour, the ducks get their behaviour by being **composed** with the right behaviour objects and **delegate** to the behaviour objects
- This allows you to **encapsulate** a family of algorithms
- Enables you to “**change behaviour**” at run-time

**Design Principle #5:**

Favour composition over inheritance

# Our first design pattern

- We have just applied our first design pattern to design our Duck app

## STRATEGY PATTERN

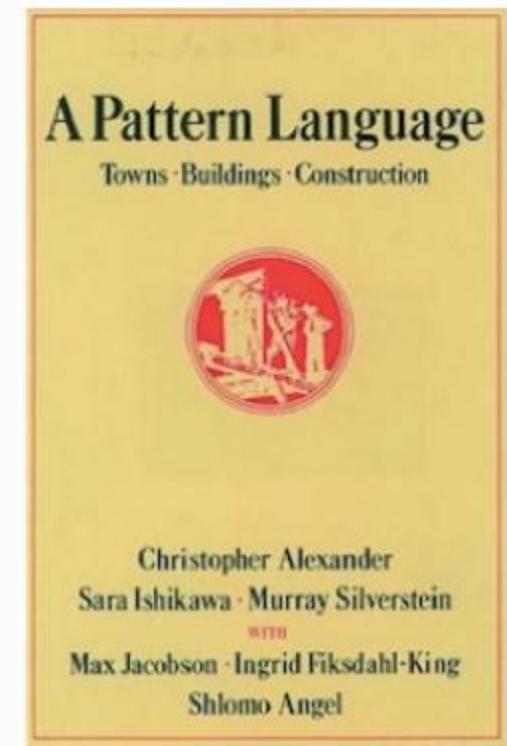
- This allows you to encapsulate a family of algorithms
- Enables you to “change behaviour” at run-time

### Design Pattern #1: Strategy Pattern

This pattern defines a family of algorithms, encapsulates each one

# Design pattern

- A **design pattern** is a tried solution to a commonly recurring problem
- Original use comes from a set of 250 patterns formulated by Christopher Alexander et al for architectural (building) design
- Every pattern has
  - A short name
  - A description of the context
  - A description of the problem
  - A prescription for a solution



# Design pattern

- In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design
- A design pattern is
  - Represents a template for how to solve a problem
  - Captures design expertise and enables this knowledge to be transferred and reused
  - Provide shared vocabularies, improve communications and eases implementation
  - Is not a finished solution, they give you general solutions to design problems

# How to use Design Patterns?

Using Design Patterns is essentially an “art & craft”

- Have a good working knowledge of patterns
- Understand the problems they can solve
- Recognise when a problem is solvable by a pattern

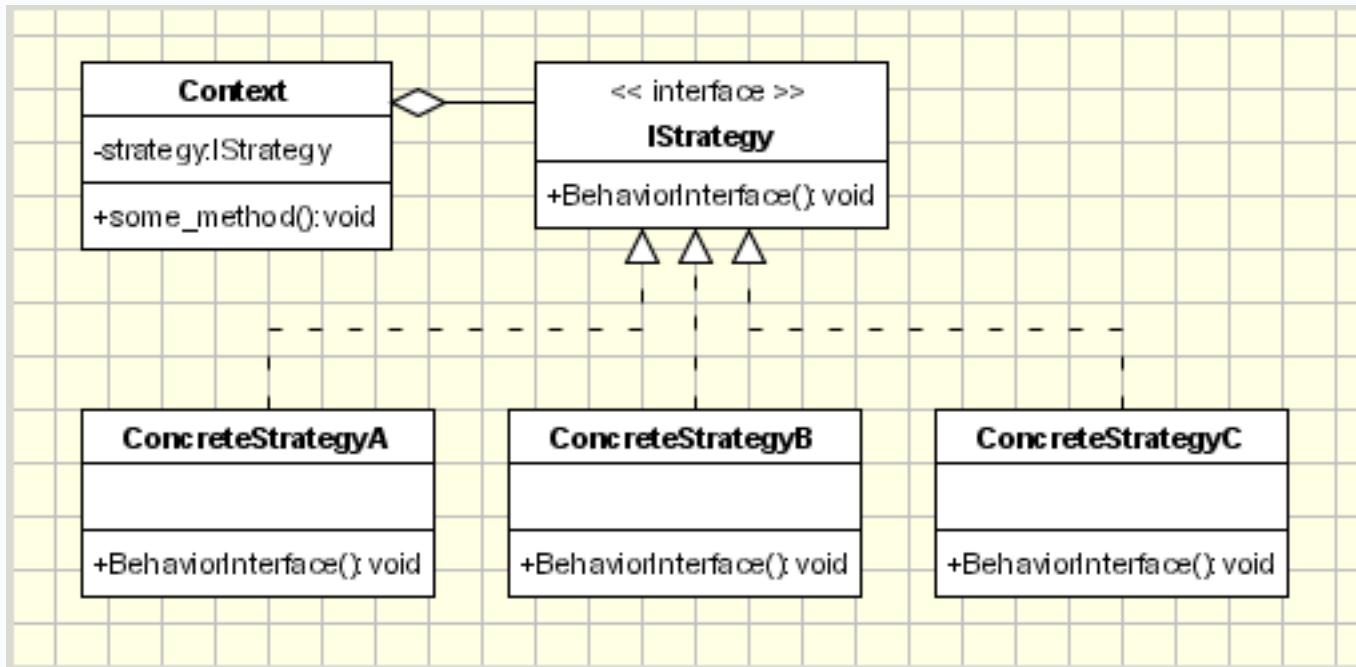
# Design Patterns Categories

- Behavioural Patterns
- Structural Patterns
- Creational Patterns

# Pattern #1: Strategy Pattern

- Motivation
  - Need a way to adapt the behaviour of an algorithm at runtime
- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Strategy pattern is a **behavioural design pattern** that lets the algorithm vary independently from the context class using it

# Strategy Pattern: Implementation



# Strategy Pattern: Uses, Benefits, Liabilities

- Applicability
  - Many related classes differ in their behaviour
  - A context class can benefit from different variants of an algorithm
  - A class defines many behaviours, and these appears as multiple conditional statements (e.g., if or switch). Instead, move each conditional branch into their own concrete strategy class
- Benefits
  - Uses composition over inheritance which allows better decoupling between the behaviour and context class that uses the behaviour
- Drawbacks
  - Increases the number of objects
  - Client must be aware of different strategies

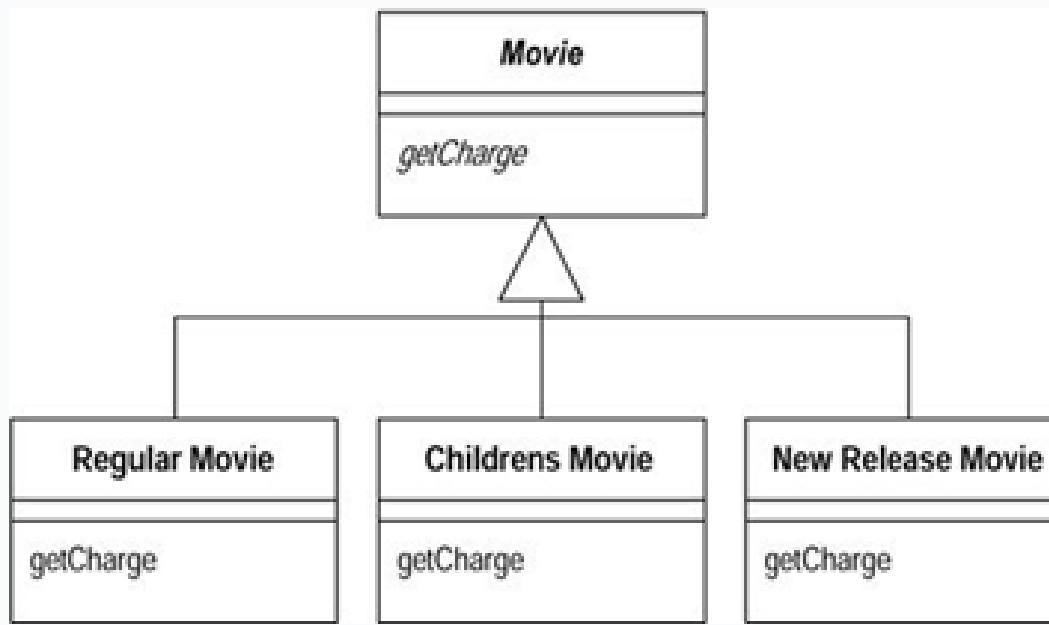
# Strategy Pattern: Examples

- Sorting a list (quicksort, bubble sort, merge-sort)
  - Encapsulate each sort algorithm into a concrete strategy class
  - Context class decides at run-time, which sorting behaviour is needed
- Search (binary search, DFS, BFS, A\*)

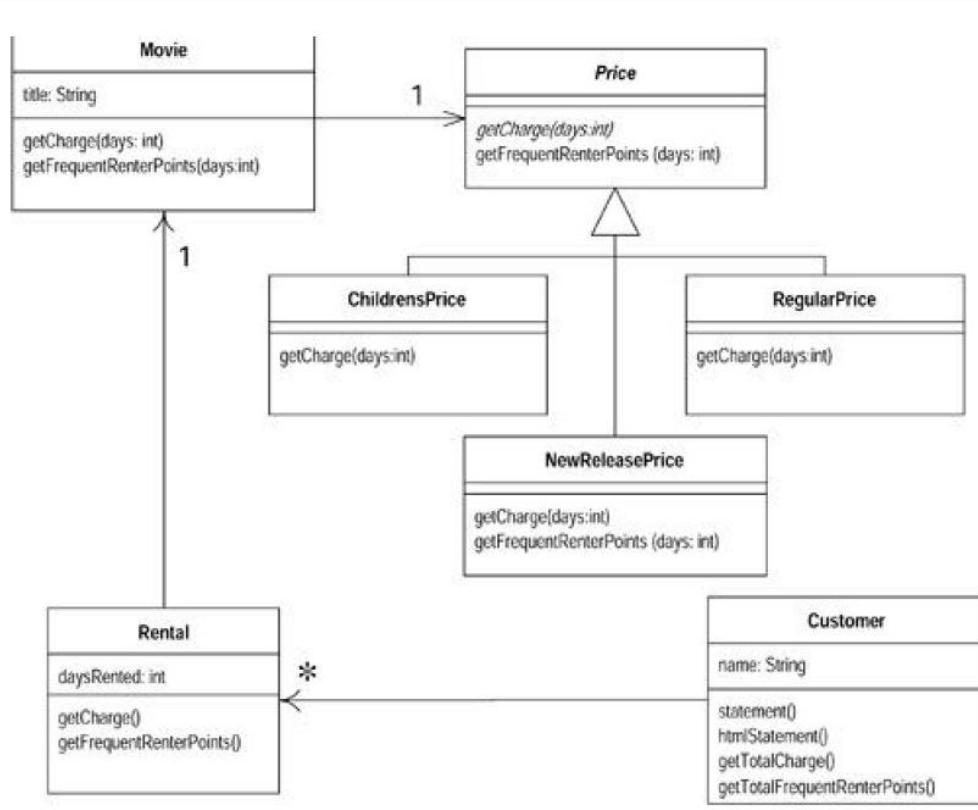
Next,

- State Pattern
- Revisit our video rental example

# Recall our Video Rental Example from Week 03



- A movie can change its classification during its life-time, hence the price of the movie would vary
- The design above is not right, for the same reason we cannot have `fly()` inside the Duck class



- Remember our design principles
  - encapsulate what varies
  - compose and delegate
- Refactoring Techniques that support these principles
  - Replace Type Code with Strategy/State Pattern
  - Replace conditional logic with polymorphism

# Summary

- Knowing OO basics does not make you a good OO designer
- Good OO designs are reusable, extensible and maintainable

## OO Basics

- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Polymorphism*

## OO Principles

- *Principle of least knowledge – talk only to your friends*
- *Encapsulate what varies*
- *Favour composition over inheritance*
- *Program to an interface, not an implementation*

## OO Patterns

- *Strategy*
- *State*

# COMP2511

## Java: Lambda Expressions, Aggregate Operations, Method References

Prepared by

Dr. Ashesh Mahidadia

# Java Lambda Expressions

- ❖ Lambda expressions allow us to
  - ❖ easily define **anonymous methods**,
  - ❖ treat **code as data** and
  - ❖ pass **functionality** as method **argument**.
- ❖ An **anonymous inner class** with **only one method** can be replaced by a **lambda expression**.
- ❖ Lambda expressions can be used to implement an interface with **only one abstract method**. Such interfaces are called *Functional Interfaces*.
- ❖ Lambda expressions offer *functions as objects* - a feature from functional programming.
- ❖ Lambda expressions are less verbose and offers more flexibility.

# Java Lambda Expressions - Syntax

A lambda expression consists of the following:

- ❖ A **comma-separated** list of formal **parameters** enclosed in parentheses. No need to provide data types, they will be inferred. For only one parameter, we can omit the parentheses.
- ❖ The **arrow token**, **->**
- ❖ A **body**, which consists of a single expression or a statement block.

```
public interface MyFunctionInterfaceA {  
    public int myCompute(int x, int y);  
}
```

```
public interface MyFunctionInterfaceB {  
    public boolean myCmp(int x, int y);  
}
```

```
public interface MyFunctionInterfaceC {  
    public double doSomething(int x);  
}
```

```
MyFunctionInterfaceA f1 = (x, y) -> x + y ;  
  
MyFunctionInterfaceA f2 = (x, y) -> x - y + 200;  
  
MyFunctionInterfaceB f3 = (x, y) -> x > y ;  
  
MyFunctionInterfaceC f4 = x -> {  
    double y = 1.5*x;  
    return y + 8.0;  
};  
  
System.out.println( f1.myCompute(10, 20) ); // prints 30  
System.out.println( f2.myCompute(10, 20) ); // prints 190  
System.out.println( f3.myCmp(10, 20) ); // prints false  
System.out.println( f4.doSomething(10) ); // prints 23.0
```

# Method References

We can treat an existing method as an instance of a Functional Interface.

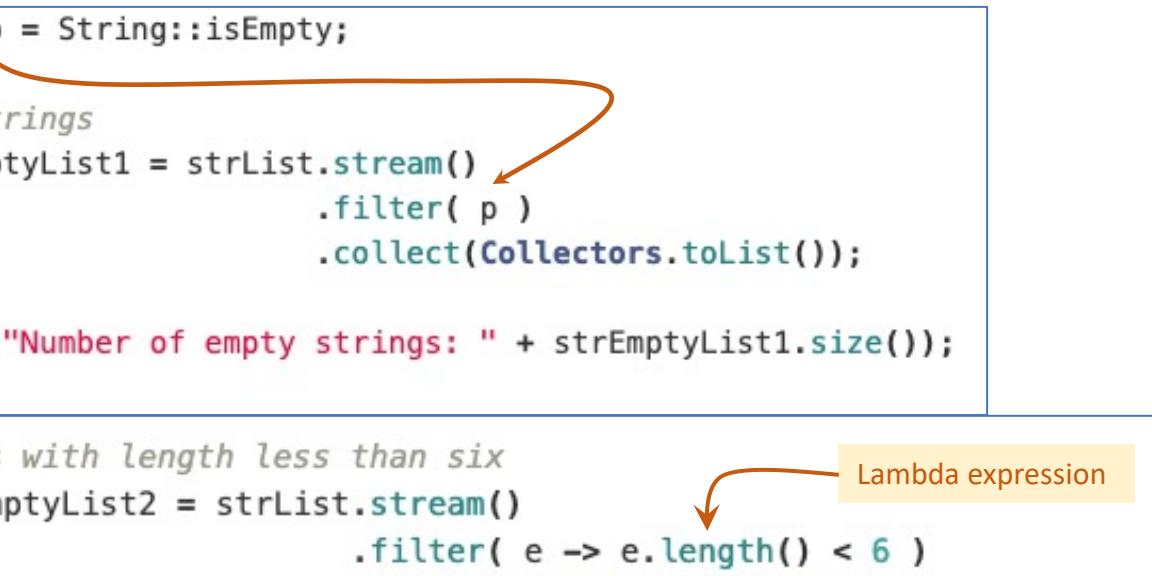
There are multiple ways to refer to a method, using `::` operator.

- ❖ A **static** method (`ClassName ::methName`)
- ❖ An **instance** method of a particular object (`instanceRef ::methName`) or  
(`ClassName ::methName`)
- ❖ A class **constructor** reference (`ClassName ::new`)
- ❖ Etc.

# Function Interfaces in Java

- ❖ Functional interfaces, in the package `java.util.function`, provide predefined **target types** for **lambda expressions** and **method references**.
- ❖ Each functional interface has a **single abstract method**, called the functional method for that functional interface, to which the **lambda expression's parameter and return types are matched** or adapted.
- ❖ Functional interfaces can provide a target type in **multiple contexts**, such as assignment context, method invocation, etc. For example,

```
Predicate<String> p = String::isEmpty;  
  
// Collect empty strings  
List<String> strEmptyList1 = strList.stream()  
    .filter( p )  
    .collect(Collectors.toList());  
  
System.out.println("Number of empty strings: " + strEmptyList1.size());  
// prints 3  
  
// Collect strings with length less than six  
List<String> strEmptyList2 = strList.stream()  
    .filter( e -> e.length() < 6 )  
    .collect(Collectors.toList());  
  
System.out.println("Number of strings with length < 6: " + strEmptyList2.size());  
// prints 4
```



# Function Interfaces in Java

- ❖ There are several basic *function shapes*, including
  - ❖ **Function** (unary function from T to R),
  - ❖ **Consumer** (unary function from T to void),
  - ❖ **Predicate** (unary function from T to boolean), and
  - ❖ **Supplier** (nilary function to R).
- ❖ More information at the package summary page  
<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

## Function Interfaces in Java: Examples

```
Function<String, Integer> func = x -> x.length();
Integer answer = func.apply("Sydney");
System.out.println(answer); // prints 6
```

```
Function<String, Integer> func1 = x -> x.length();
Function<Integer, Boolean> func2 = x -> x > 5;
Boolean result = func1.andThen(func2).apply("Sydney");
System.out.println(result);
```

```
Predicate<Integer> myPass = mark -> mark >= 50 ;
List<Integer> listMarks = Arrays.asList(45, 50, 89, 65, 10);
List<Integer> passMarks = listMarks.stream()
    .filter(myPass)
    .collect(Collectors.toList());

System.out.println(passMarks); // prints [50, 89, 65]
```

```
Consumer<String> print = x -> System.out.println(x);
print.accept("Sydney"); // prints Sydney
```

## Function Interfaces in Java: Examples

```
// Consumer to multiply 5 to every integer of a list
Consumer<List<Integer>> myModifyList = list -> {
    for (int i = 0; i < list.size(); i++)
        list.set(i, 5 * list.get(i));
};
```

```
List<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(1);
list.add(10);
```

```
// Implement myModifyList using accept()
myModifyList.accept(list);
```

```
// Consumer to display a list of numbers
Consumer<List<Integer>> myDispList = myList -> {
    myList.stream().forEach(e -> System.out.println(e));
};
```

```
// Display list using myDispList
myDispList.accept(list);
```

# Comparator using Lambda Expression: Example

```
//Using an anonymous inner class
Comparator<Customer> myCmpAnonymous = new Comparator<Customer>() {
    @Override
    public int compare(Customer o1, Customer o2) {
        return o1.getRewardsPoints() - o2.getRewardsPoints();
    }
}
custA.sort( myCmpAnonymous );
```

Only one line!

```
//Using Lambda expression – simple example (only one line)
custA.sort((Customer o1, Customer o2)->o1.getRewardsPoints() - o2.getRewardsPoints());
```

```
custA.forEach( (cust) -> System.out.println(cust) );
```

Print using Lambda expression

# Comparator using Lambda Expression: Another Example

```
//Using Lambda expression - Another example (with return)
custA.sort( (Customer o1, Customer o2) -> {
    if(o1.getPostcode() != o2.getPostcode()) {
        return o1.getPostcode() - o2.getPostcode() ; }
    return o1.getRewardsPoints() - o2.getRewardsPoints() ;
});
```

Parameters – o1 and o2

Body

# Pipelines and Streams

- ❖ A **pipeline** is a sequence of **aggregate** operations.
- ❖ The following example prints the male members contained in the collection **roster** with a **pipeline** that consists of the **aggregate** operations **filter** and **forEach**:

```
roster
```

```
.stream()  
.filter( e -> e.getGender() == Person.Sex.MALE )  
.forEach( e -> System.out.println(e.getName()) );
```

Using pipeline and aggregate ops:

```
for (Person p : roster) {  
    if (p.getGender() == Person.Sex.MALE) {  
        System.out.println(p.getName());  
    }  
}
```

Traditional approach,  
using a **for-each** loop:

- ❖ Please note that, in a pipeline, **operations are loosely coupled**, they only rely on their incoming streams and can be easily rearranged/replaced by other suitable operations.
- ❖ Just to clarify, the “.” (dot) operator in the above syntax has a very different meaning to the “.” (dot) operator used with an **instance** or a **class**.

# Pipelines and Streams

- ❖ A **pipeline** contains the following components:
  - A **source**: This could be a collection, an array, a generator function, or an I/O channel. Such as *roster* in the example.
  - Zero or **more intermediate operations**. An intermediate operation, such as **filter**, produces a new stream.
- ❖ A **stream** is a sequence of elements. The method **stream** creates a stream from a collection (*roster*).
- ❖ The **filter** operation returns a new stream that contains elements that **match** its **predicate**. The **filter** operation in the example returns a stream that contains all male members in the collection *roster*.
- ❖ A **terminal** operation. A terminal operation, such as **forEach**, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of **forEach**, no value at all.

```
roster
    .stream()
    .filter( e -> e.getGender() == Person.Sex.MALE )
    .foreach( e -> System.out.println(e.getName()) );
```

# Pipelines and Streams: Example

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

- ❖ The above example calculates the average age of all male members contained in the collection `roster` with a pipeline that consists of the aggregate operations `filter`, `mapToInt`, and `average`.
- ❖ The `mapToInt` operation returns a new stream of type `IntStream` (which is a stream that contains only integer values). The operation applies the function specified in its parameter to each element in a particular stream.
- ❖ As expected, the `average` operation calculates the average value of the elements contained in a stream of type `IntStream`.
- ❖ There are many `terminal operations` such as `average` that return one value by combining the contents of a stream. These operations are called `reduction operations`; see the section Reduction for more information at <https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html>

# Pipelines and Streams: Another Example

```
double avgNonEmptyStrLen = strList.stream()
    .filter( e -> e.length() > 0 )
    .mapToInt(String::length)
    .average()
    .getAsDouble();
```

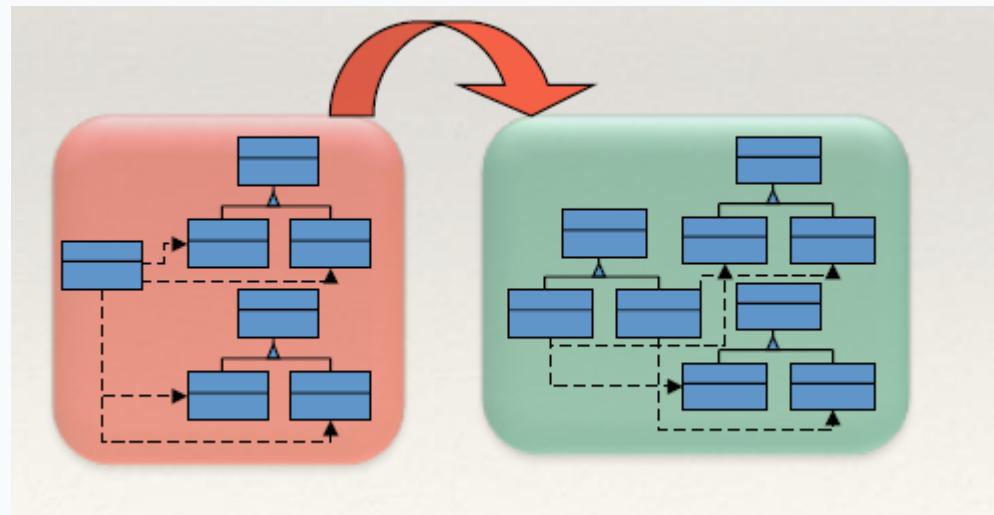
**End**

# **COMP2511**

# **Refactoring**

# Refactoring

The process of **restructuring** (changing the internal structure of software) software to make it *easier to understand* and *cheaper to modify* without changing its *external, observable behaviour*



# Why should you refactor?

- Refactoring improves design of software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster
- Refactoring helps you to conform to design principles and avoid design smells

# When should you refactor?

**Tip:** *When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature*

**Refactor** when:

- You add a function (swap hats between adding a function and refactoring)
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

# Common Bad Code Smells

- **Duplicated Code**
  - Same code structure in more than one place or
  - Same expression in two sibling classes
- **Long Method**
- **Large Class** (when a class is trying to do too much, it often shows up as too many instance variables)
- **Long Parameter List**
- **Divergent Change** ( when one class is commonly changed in different ways for different reasons )
- **Shotgun Surgery** ( The opposite of divergent change, when you have to make a lot of little changes to a lot of different classes )

# The Video Rental Example

## What is wrong with the design?

Is it wrong to write a quick and dirty solution OR is it an aesthetic judgment (dislike of ugly code ) ...

- Overly long `statement()` method , poorly designed that does far too much, tasks that should be done by other classes ([Code Smell: Long Method](#))
- What if customer wanted to generate a statement in HTML? - Impossible to reuse any of the behaviour of the current statement method for an HTML statement. ([Code Smell: Duplicated code](#))
- What about changes?
  - What happens when “charging rules” change?
  - what if the user wanted to change the way the movie was classified
- The code is a maintenance night-mare ([Design smell: Rigidity](#))

# Improving the design

Apply a series of fundamental refactoring techniques:

## Technique #1: Extract Method

- Find a logical clump of code and use Extract Method.  
Which is the obvious place? the switch statement
- Scan the fragment for any variables that are local in scope to the method we are looking at  
(`Rental r` and `thisAmount`)
- Identify the changing and non-changing local variables
- Non-changing variable can be passed as a parameter
- Any variable that is modified needs more care, if there is only one, you could simply do a return

# Improving the design

## Technique #2: Rename variable

- Is renaming worth the effort? Absolutely
- Good code should communicate what it is doing clearly, and variable names are a key to clear code. Never be afraid to change the names of things to improve clarity.

### Tip

*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

# Improving the design

## #3: Move method

- Re-examine method `calculateRental()` in class `Customer`
- Method uses the `Rental` object and not the `Customer` object
- Method is on the wrong object

### Tip

*Generally, a method should be on the object whose data it uses*

# Improving the design

What OO principles do Extract Method and Move Method use?

They make code reusable through Encapsulation and Delegation

But, isn't encapsulation about keeping your data private?

The basic idea about encapsulation is to protect information in one part of your application from other parts of the application, so

- You can protect data
- You can protect behaviour – when you break the behaviour out from a class, you can change the behaviour without the class having to change

And what is delegation?

- The act of one object forwarding an operation to another object to be performed on behalf of the first object

# Improving the design

## #4: Replace Temp With Query

- A technique to remove unnecessary local and temporary variables
- Temporary variables are particularly insidious in long methods and you can lose track of what they are needed for
- Sometimes, there is a performance price to pay

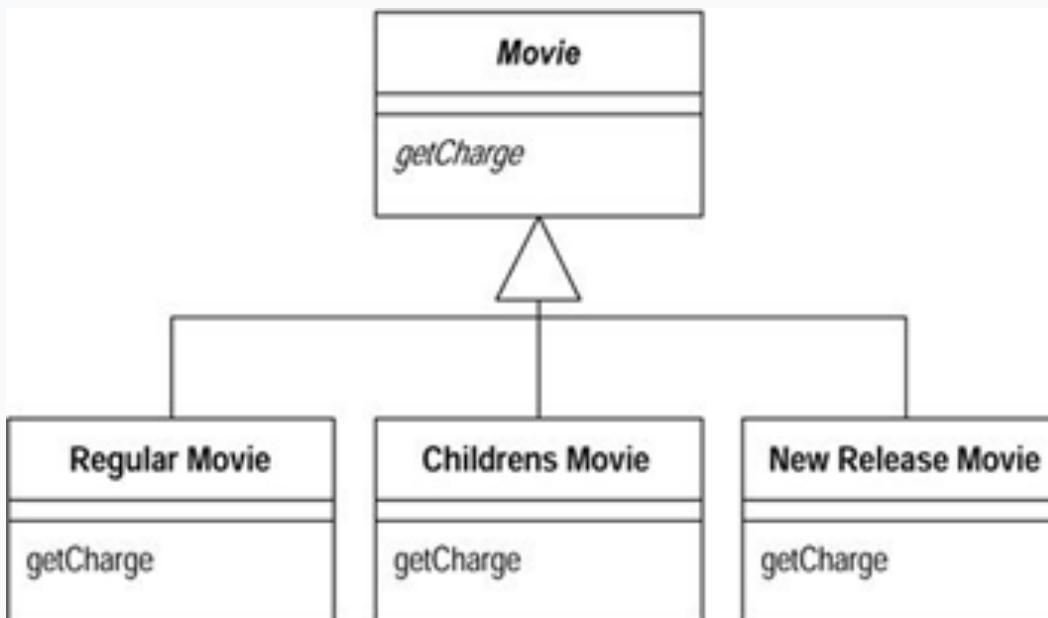
# Improving the design

## #5: Replacing conditional logic with Polymorphism

- The switch statement – an obvious problem, with two issues
- class `Rental` Is tightly coupled with class `Movie` - a switch statement based on the data of another object – not a good design
- There are several types of movies with its own type of charge, hmm... sounds like inheritance

# Improving the design

- A base class Movie class with method getPrice() and sub-classes NewRelease, ChildrenMovie and Regular
- This allows us to replace **switch** statement with **polymorphism**



- Sadly, it has one flaw...a movie can change its classification during its life-time

# So, what options are there besides inheritance ?

- Composition – reuse behaviour using one or more classes with composition
- Delegation: delegate the functionality to another class

*...this is the second time, this week we have said, we need something more than inheritance*

So, next ...

- **Design Principle:** Favour composition over inheritance
- **More refactoring techniques to solve our “switch” problem**
  - Replace type code with Strategy/State Pattern
  - Move Method
  - Replace conditional code with polymorphism

# COMP2511

# State Pattern

Prepared by

Dr. Ashesh Mahidadia

# State Pattern

These lecture notes are from the wikipedia page at: [https://en.wikipedia.org/wiki/Finite-state\\_machine](https://en.wikipedia.org/wiki/Finite-state_machine)

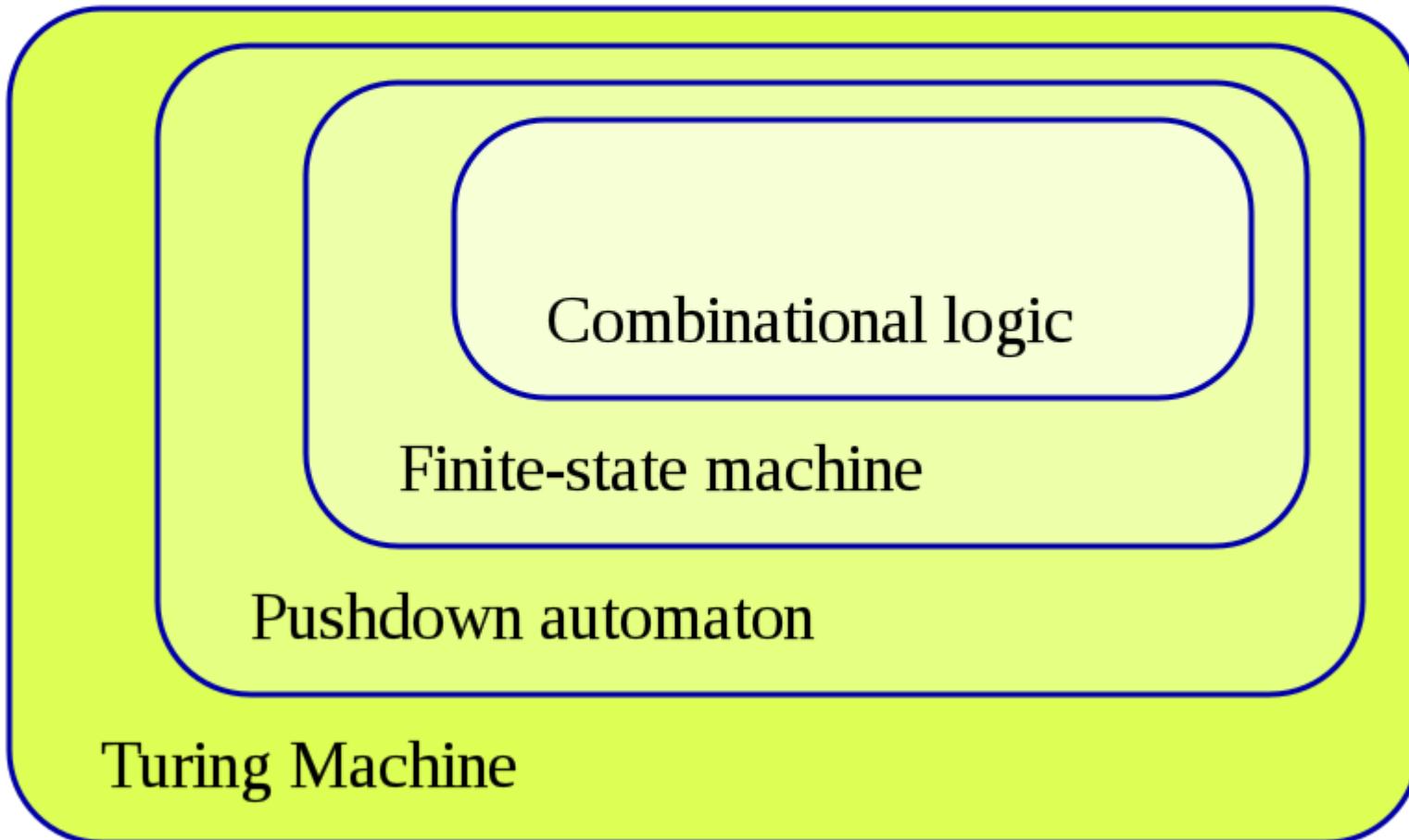
And

the reference book “Head First Design Patterns”.

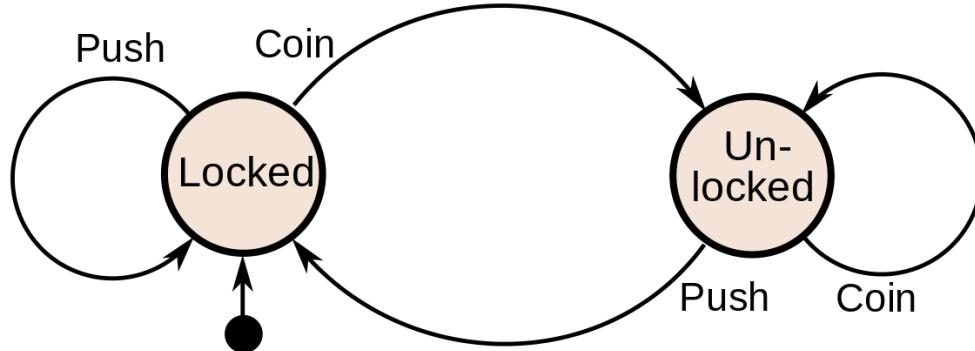
# Finite-state Machine

- A **finite-state machine (FSM)**, is an abstract machine that can be in exactly **one** of a finite number of **states** at any given time.
  - the finite-state machine can change from one state to another in response to some external **inputs**.
  - the change from one state to another is called a **transition**.
- An finite-state machine is **defined** by
  - a list of its **states**
  - the conditions for each **transition**
  - its **initial** state
- Finite-state machine also refer to as finite-state automaton, finite automaton, or state machine

# Automata theory



# Example: coin-operated turnstile



Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

**State Transition Table:** shows for each possible state, the transitions between them (based upon the inputs given to the machine) and the outputs resulting from each input

# State Machines: Simple examples

- **vending machines**, which dispense products when the proper combination of coins is deposited,
- **elevators**, whose sequence of stops is determined by the floors requested by riders,
- **traffic lights**, which change sequence when cars are waiting,
- **combination locks**, which require the input of combination numbers in the proper order.

# State Machine: Terminology

- A **state** is a description of the status of a system that is waiting to execute a *transition*.
- A **transition** is a set of actions to be executed when a condition is fulfilled or when an event is received.
- **Identical stimuli trigger different actions depending on the current state.**
- For example,
  - when using an audio system to listen to the radio (the system is in the "radio" state), receiving a "**next**" stimulus results in moving to the next station.
  - when the system is in the "CD" state, the "**next**" stimulus results in moving to the next track.
- Often, the following are also associated with a state:
  - an **entry action**: performed *when entering* the state, and
  - an **exit action**: performed *when exiting* the state.

# Representations

- The most common representation is shown below:

		State transition table		
		Current state	State A	State B
Input	Current state	State A	State B	State C
Input X		...	...	...
Input Y		...	State C	...
Input Z		...	...	...

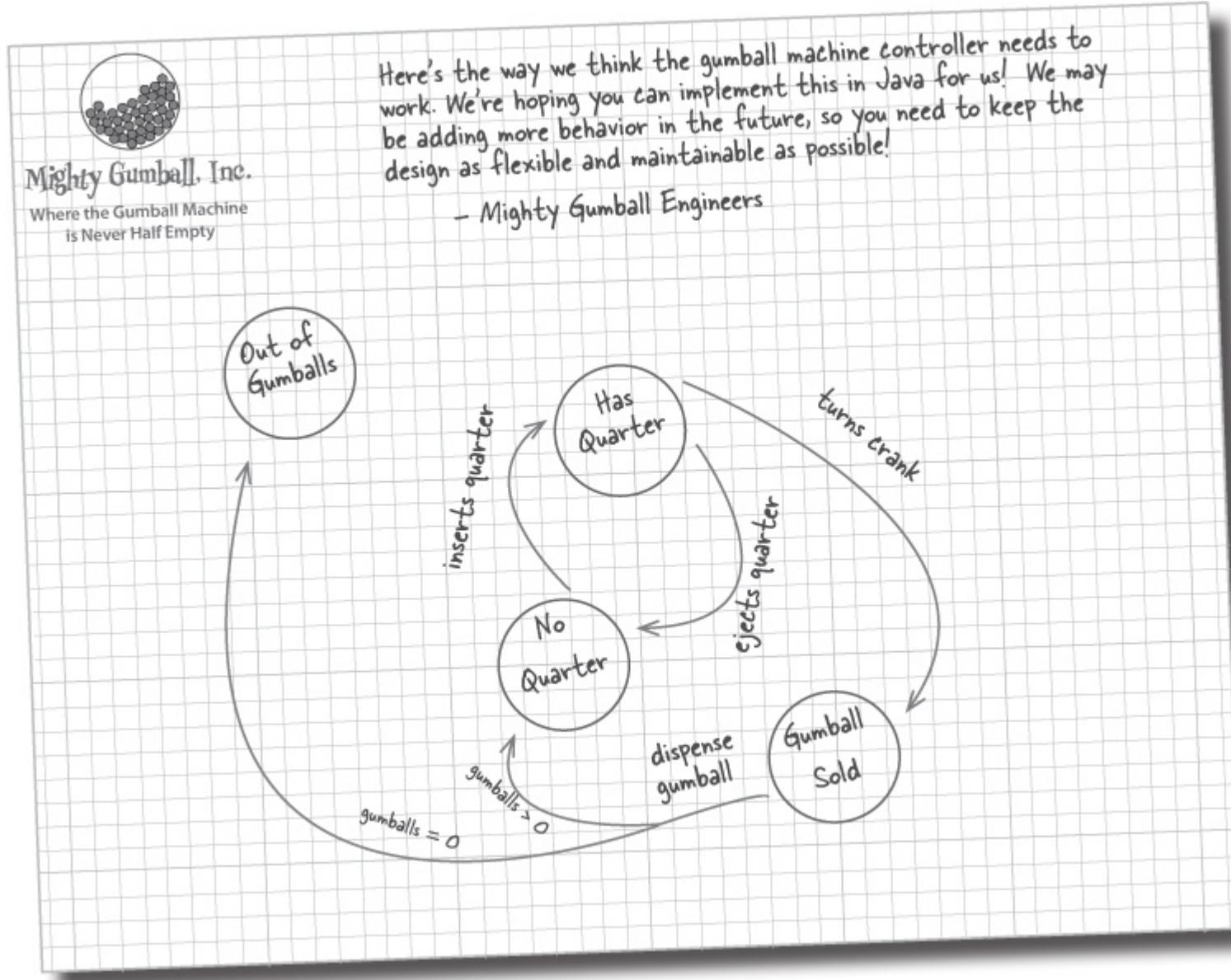
# State Machines for UI

- Examples ...

# Gumball Machine!



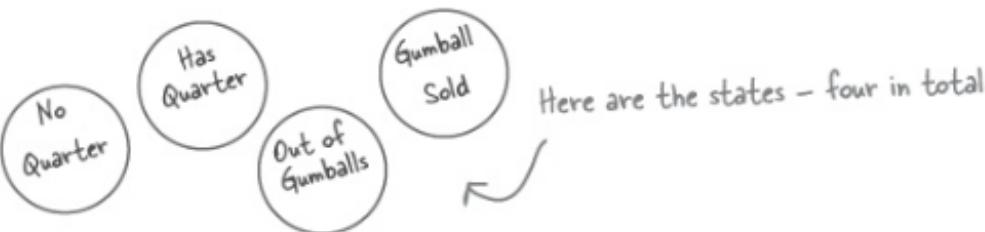
From the reference book  
“head First Design Patterns”



# State machines 101

How are we going to get from that state diagram to actual code? Here's a quick introduction to implementing state machines:

- ① First, gather up your states:



- ② Next, create an instance variable to hold the current state, and define values for each of the states:

Let's just call "Out of Gumballs"  
"Sold Out" for short

```
final static int SOLD_OUT = 0;  
final static int NO_QUARTER = 1;  
final static int HAS_QUARTER = 2;  
final static int SOLD = 3;
```

Here's each state represented  
as a unique integer...

```
int state = SOLD_OUT;
```

...and here's an instance variable that holds the  
current state. We'll go ahead and set it to "Sold  
Out" since the machine will be unfilled when it's  
first taken out of its box and turned on.

- ③ Now we gather up all the actions that can happen in the system:

inserts quarter      turns crank

ejects quarter

dispense

These actions are  
the gumball machine's  
interface – the things  
you can do with it.

Looking at the diagram, invoking any of  
these actions causes a state transition.

Dispense is more of an internal  
action the machine invokes on itself.

From the reference book  
"head First Design Patterns"

- ④ Now we create a class that acts as the state machine. For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state. For instance, for the insert quarter action, we might write a method like this:

Read the example code provided for this week

```
public void insertQuarter() {  
  
    if (state == HAS_QUARTER) {  
  
        System.out.println("You can't insert another quarter");  
  
    } else if (state == NO_QUARTER) {  
  
        state = HAS_QUARTER;  
        System.out.println("You inserted a quarter");  
  
    } else if (state == SOLD_OUT) {  
  
        System.out.println("You can't insert a quarter, the machine is sold out");  
  
    } else if (state == SOLD) {  
  
        System.out.println("Please wait, we're already giving you a gumball");  
  
    }  
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

Here we're talking about a common technique: modeling state within an object by creating an instance variable to hold the state values and writing conditional code within our methods to handle the various states.



From the reference book  
“head First Design Patterns”

Read the example code provided for this week

```

public void ejectQuarter() {
    if (state == HAS_QUARTER) { ↗ Now, if the customer tries to remove the quarter...
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}
    ↗ You can't eject if the machine is sold out; it doesn't accept quarters!
    ↗ The customer tries to turn the crank...
public void turnCrank() {
    if (state == SOLD) { ↗ Someone's trying to cheat the machine.
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}
    ↗ Called to dispense a gumball.
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}
    ↗ Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.
    ↗ We're in the SOLD state; give 'em a gumball!
    ↗ Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.
    ↗ None of these should ever happen, but if they do, we give 'em an error, not a gumball.
}
// other methods here like toString() and refill()
}
    ↗ If there is a quarter, we return it and go back to the NO_QUARTER state.
    ↗ Otherwise, if there isn't one we can't give it back.
    ↗ If the customer just turned the crank, we can't give a refund; he already has the gumball!
    ↗ We need a quarter first.
    ↗ We can't deliver gumballs; there are none.

```

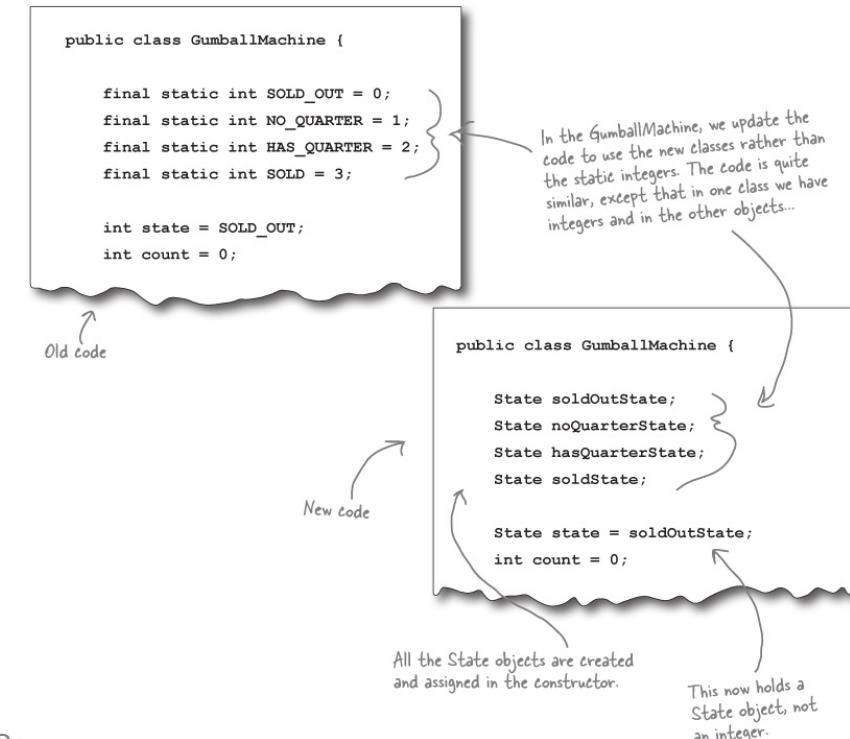
From the reference book  
"head First Design Patterns"

## The new design

It looks like we've got a new plan: instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.

We're following our design principles here, so we should end up with a design that is easier to maintain down the road. Here's how we're going to do it:

- ① First, we're going to define a State interface that contains a method for every action in the Gumball Machine.
- ② Then we're going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.
- ③ Finally, we're going to get rid of all of our conditional code and instead delegate to the State class to do the work for us.



From the reference book  
“head First Design Patterns”

Read the example  
Code provided for  
this week

```

public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine); ← Print out the state of the machine.

        gumballMachine.insertQuarter(); ← Throw a quarter in...
        gumballMachine.turnCrank(); ← Turn the crank; we should get our gumball.

        System.out.println(gumballMachine); ← Print out the state of the machine, again.

        gumballMachine.insertQuarter(); ← Throw a quarter in...
        gumballMachine.ejectQuarter(); ← Ask for it back.
        gumballMachine.turnCrank(); ← Turn the crank; we shouldn't get our gumball.

        System.out.println(gumballMachine); ← Print out the state of the machine, again.

        gumballMachine.insertQuarter(); ← Throw a quarter in...
        gumballMachine.turnCrank(); ← Turn the crank; we should get our gumball.
        gumballMachine.insertQuarter(); ← Throw a quarter in...
        gumballMachine.turnCrank(); ← Turn the crank; we should get our gumball.
        gumballMachine.ejectQuarter(); ← Ask for a quarter back we didn't put in.

        System.out.println(gumballMachine); ← Print out the state of the machine, again.

        gumballMachine.insertQuarter(); ← Throw TWO quarters in...
        gumballMachine.insertQuarter(); ← Turn the crank; we should get our gumball.

        System.out.println(gumballMachine); ← Print that machine state one more time.
    }
}

```

```

File Edit Window Help mightygumball.com
$java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there's no quarter
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven't inserted a quarter
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You can't insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out

```

From the reference book  
“head First Design Patterns”

# Demo ...

- Demo of Gumball, from the reference book “Head First Design Patterns”.

## BULLET POINTS

- The State Pattern allows an object to have many different behaviors that are based on its internal state.
- Unlike a procedural state machine, the State Pattern represents state as a full-blown class.
- The Context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.
- The State and Strategy Patterns have the same class diagram, but they differ in intent.
- Strategy Pattern typically configures Context classes with a behavior or algorithm.
- State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.

From the reference book  
“head First Design Patterns”

# COMP2511



## 5.1 - The Unknown

# In this lecture

- The unknown; types of unknowns
- The role of the unknown in Software Engineering
  - The role of the unknown in **developing new software**
  - The role of the unknown in **working with existing software**
- Software Longevity
- Collaborative Engineering

A brief step back - what makes good software?

# The Unknown

- Two types of hard problems:
  - Easy to understand, hard to solve
  - Hard to understand, easy to solve
- Two types of unknowns
  - **Known unknowns** - we know that it exists, but we don't know what it is
  - **Unknown unknowns** - that which we had never even thought to consider
- Learn to deal with unknowns gracefully as they arise

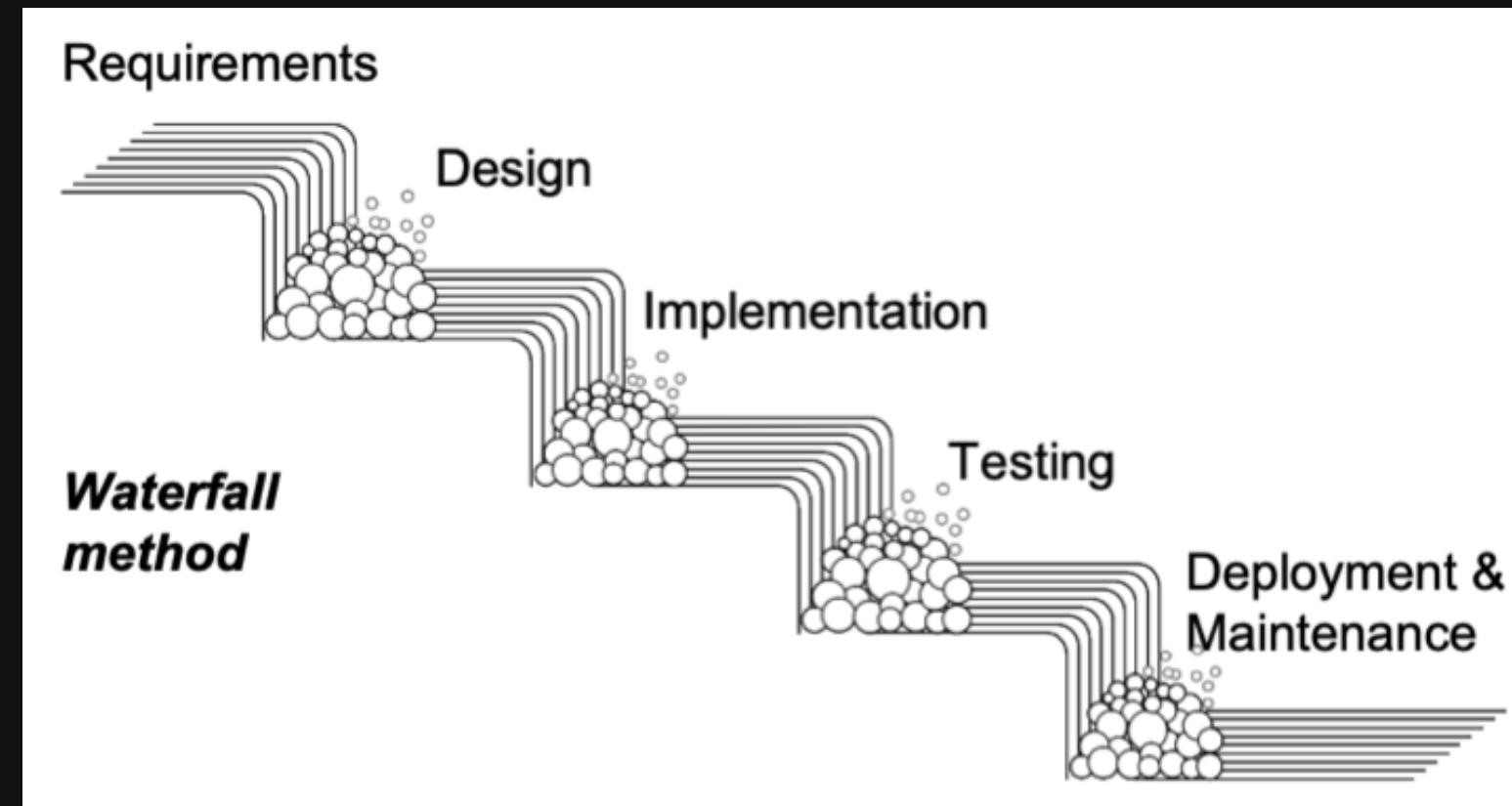
# I: The Role of The Unknown in Developing New Software

# The Software Development Lifecycle



# Traditional Engineering: The Big Design Up Front

- One step at a time
- Ensure the current step is perfect before moving onto the next one
- A big design up front
- Project can take months-years to complete



# Difficulties Presented by the Unknown

- The game changes
  - Changing market
  - Changing client expectations
  - Changing technical world
- Evolution of Requirements
- Too many unknown unknowns to be able to **design everything up-front**
- There comes a need to **learn through experience** and **iterate on design**

# Iterative Design

- Work in sprints, iterations, milestones
- 'Agile' software development
- Many variants - eXtreme programming, Rapid Application Development, Kanban, Scrum
- Design incrementally
  - Adapt to changes in requirements
  - Discover and deal with problems in design as they arise

# Problems with Purely Iterative Design

- No clear sense of direction/trajectory
- In poorly designed systems, adaptations to new requirements become smaller-scale 'workarounds' - limit functionality/decrease maintainability
- Tendency to 'make it up as we go along'

# A solution?

- **High Level Design**
  - Design a broad overview up-front
  - A framework to begin development
  - Set the trajectory and boundaries of work at the start
- Adapt and change the design during development as needed
- Design up-front a solution that is open for extension, reusable, etc.
- Complete work in **small increments** and **improve iteratively**
- A brief history of COMP2511

# Requirements Engineering

- As systems grow in size and complexity, so do their requirements
- Requirements become harder to:
  - **Communicate**, on the side of the client;
  - **Understand**, on the side of the engineer
- Domains are often highly specific, nuanced and complex
- Requirements *Analysis* might be better termed *Engineering*
- The problem space grows around us with our understanding
- Transformation of the unknown into the known
- Often, this requires us to dive into the deep end and start **actualising solutions** in order to better understand what the problem is
- We have to make **assumptions** to limit the contract to which we program

## II: The Role of The Unknown in Working with Existing Software

# Now, and Later

- What about later?
- 5 weeks, 5 months, 5 years... 15 years?
- What happens when code grows and systems get bigger?

# Complexity

- Code becomes more complex as it grows in size
- Cyclomatic complexity
- More complexity leads to:
  - More scope for errors to creep in;
  - Higher **risk** of software breaking;
  - More unknown unknowns
  - Too many possible combinations to predict up-front

# Software Becomes a Beast

- When software reaches a certain level of complexity, it becomes **alive**, and **constantly moving**
- We can no longer informally reason about it with certainty; behaviour can become unpredictable
- Changes in one class/package have far-reaching unpredictable effects due to **coupling**
- Testing becomes essential here
- **Monolith** Repositories
  - Large applications, with all the code inside a single repository
  - Slower build and Continuous Integration times
  - More *inertia* - harder to ship new features
  - More incidents, bugs and defects due to mathematical complexity
  - Industry is moving towards **microservice architecture**

# Technical Debt & Trade-offs

- In large-scale software systems, the design decisions you make today will have consequences for years to come.
- Every design decision comes with a cost (termed "technical debt")
- This cost must later be paid back in the form of maintenance, refactoring, or rewriting
- Design decisions are often a matter of picking the deal that is least-worst in the long run - no easy way out
- Opportunity cost - pick one or the other
- All software architecture and design consists of making trade-offs

# How to deal with the unknown here?

- **Code Hygiene** - take care of your code, and your code will take care of you
- Rigorous testing is essential
- A Symbiotic Relationship between Product and Engineering
  - Sacrifice time spent shipping new features **now** to maintain software hygiene, meaning we can still ship new features **later**
  - Slower today, faster forever

# Software Longevity

- Difficult challenges:
  - Dealing with code that you wrote weeks, months or years ago
  - Dealing with code that *other people* wrote weeks, months and years ago
- In large-scale software systems, many software components are treated as black-boxes since the original author is no longer present and no one understands how it works!
- Can often be a tendency to **completely migrate (move) systems** rather than **maintaining or decomposing existing legacy infrastructure**
- Software has a very fast turnover compared to other forms of engineering

# Longevity and the Open-Closed Principle

- There can be a tendency to **build around existing software** rather than going in and making changes
- In some philosophical respects, this is the open-closed principle
- The band-aid problem

# Parting Thoughts on Longevity

- Mark of well-written code:
  - It remains intact for a long time
  - *And* new engineers are able to onboard and understand how it functions
- Bugs lying dormant - Human tendency to ignore that which we do not immediately understand or must analyse - leads to problems in legacy codebases remaining unchecked before surfacing

# Collaborative Engineering

- Pair assignment - dealing with the unknown together
- Bounce ideas off one another
- Review each others' work - design reviews, test reviews, code reviews
- Pair programming
  - One person codes, one person watches

# COMP2511

# Composite Pattern

Prepared by

Dr. Ashesh Mahidadia

# Composite Pattern

These lecture notes use material from the wikipedia page at: [https://en.wikipedia.org/wiki/Composite\\_pattern](https://en.wikipedia.org/wiki/Composite_pattern)

and

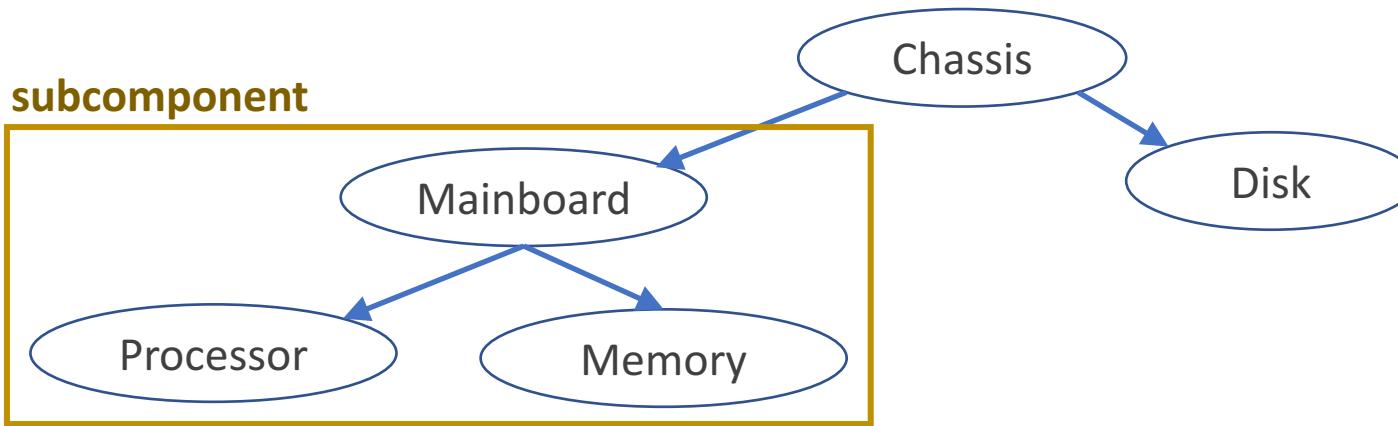
the reference book “[Head First Design Patterns](#)”.

# Composite Pattern: Motivation and Intent

- In OO programming, a **composite** is an object designed as a composition of one-or-more similar objects (exhibiting similar functionality).
- Aim is to be able to manipulate a single instance of the object just as we would manipulate a group of them. For example,
  - operation to resize a **group** of Shapes should be **same as** resizing a **single Shape**.
  - calculating size of a **file** should be **same as a directory**.
- **No discrimination** between a Single (leaf) Vs a Composite (group) object.
  - If we discriminate between a single object and a group of object, code will become more complex and therefore, more error prone.

# Composite Pattern: More Examples

Calculate the total price of an **individual part** or a complete **subcomponent** (consisting of many parts) without having to treat part and subcomponent differently.

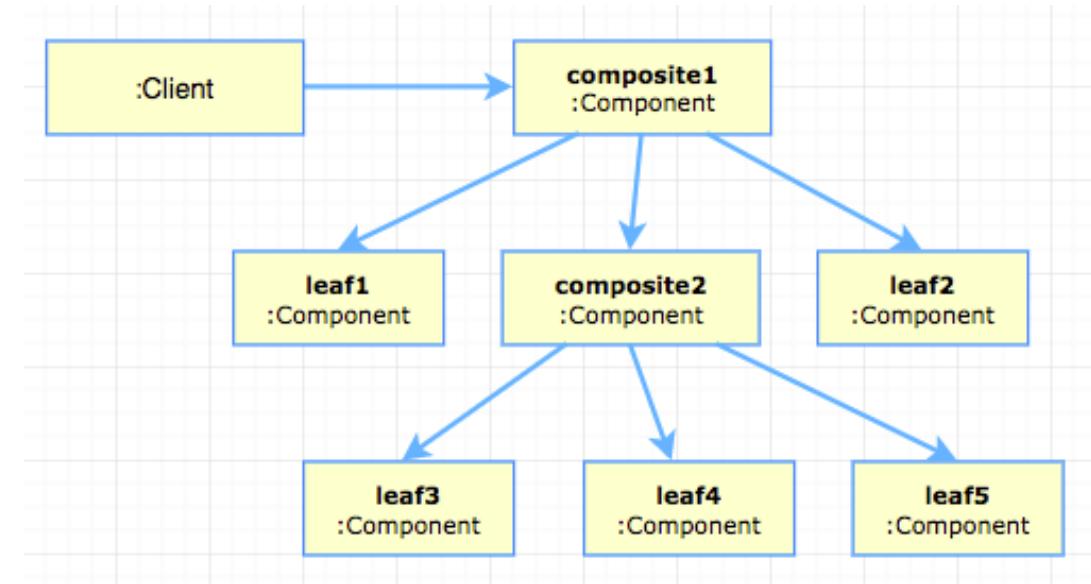
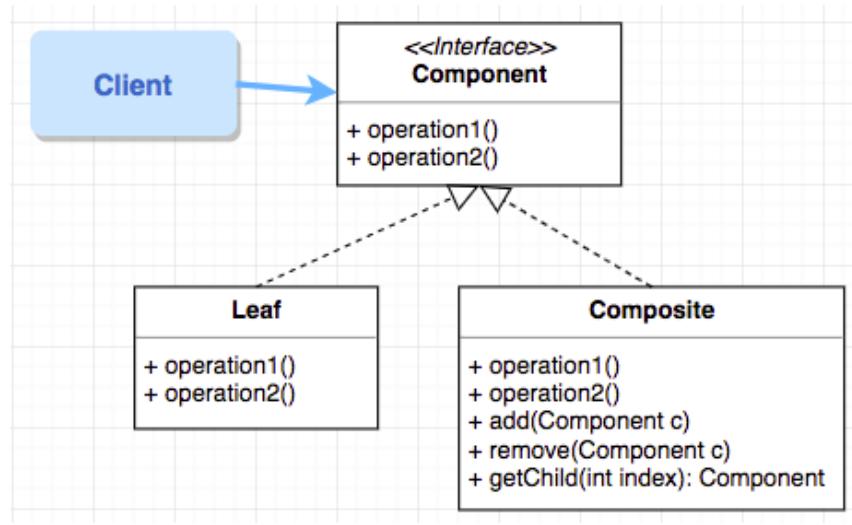


---

A **text document** can be organized as **part-whole hierarchy** consisting of

- characters, pictures, lines, pages, etc. (parts) and
- lines, pages, document, etc. (wholes).
- Display a line, page or the entire document (consisting of many pages) **uniformly** using the same operation/method.

# Composite Pattern: Possible Solution



- Define a unified **Component** interface for both **Leaf** (*single / part*) objects and **Composite** (*Group / whole*) objects.
- A **Composite** stores a **collection of children** components (either **Leaf** and/or **Composite** objects).
- Clients can **ignore the differences** between compositions of objects and individual objects, this greatly **simplifies** clients of complex hierarchies and makes them easier to implement, change, test, and reuse.

# Composite Pattern: Possible Solution

- *Tree structures* are normally used to represent part-whole hierarchies. A multiway tree structure stores a collection of say Components at each node (**children** below), to store Leaf objects and Composite (subtree) objects.
- A **Leaf** object performs operations directly on the object.
- A **Composite** object performs operations on its **children**, and if required, collects return values and derives the required answers.

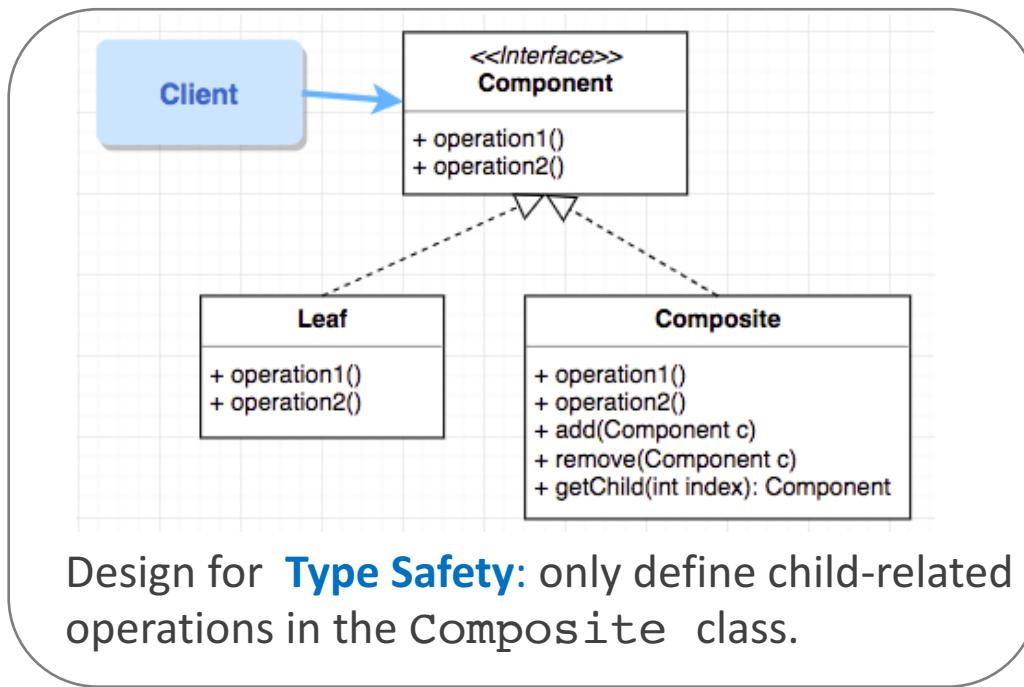
Code Segment from the **Composite** class

```
ArrayList<Component> children = new ArrayList<Component>();  
  
@Override  
public double calculateCost() {  
    double answer = this.getCost();  
    for(Component c : children) {  
        answer += c.calculateCost();  
    }  
  
    return answer;  
}
```

For more, read the example code provided for this week

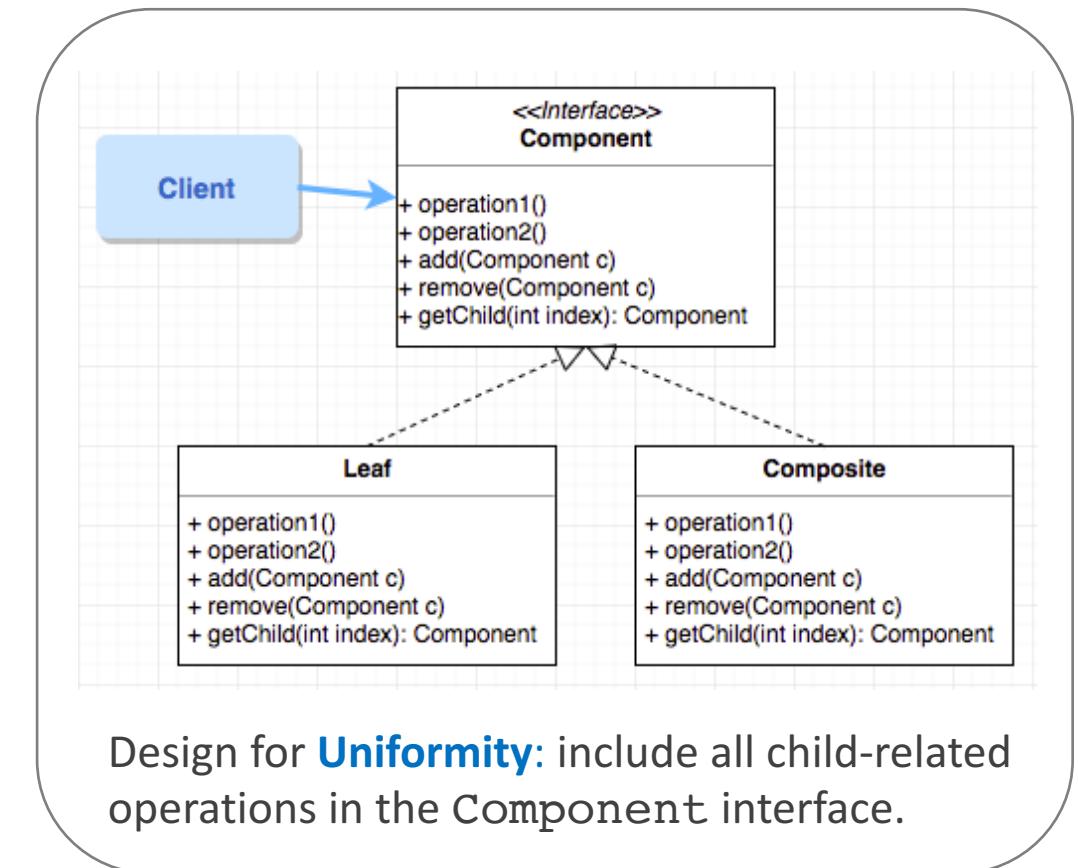
# Implementation Issue: Uniformity vs Type Safety

Two possible approaches to implement child-related operations  
(methods like add, remove, getChild, etc.):



Design for **Type Safety**: only define child-related operations in the **Composite** class.

See the [next slide](#) for more details.



Design for **Uniformity**: include all child-related operations in the **Component** interface.

# Implementation Issue: Uniformity vs Type Safety

## Design for Uniformity

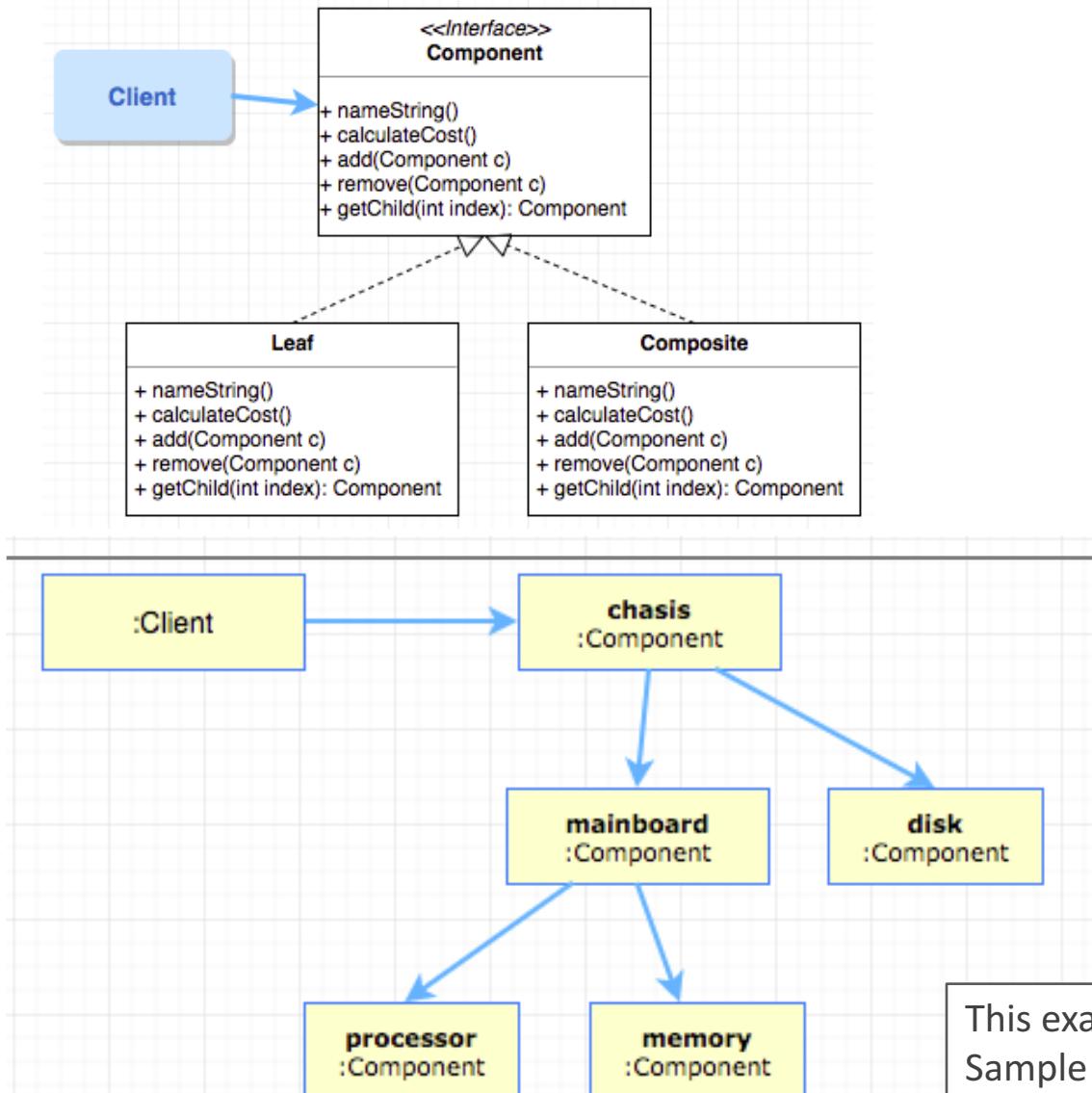
- include all child-related operations in the Component interface, this means the Leaf class needs to implement these methods with “do nothing” or “throw exception”.
- a client can treat both Leaf and Composite objects uniformly.
- we lose type safety because Leaf and Composite types are not cleanly separated.
- useful for dynamic structures where children types change dynamically (from Leaf to Composite and vice versa), and a client needs to perform child-related operations regularly. For example, a document editor application.

## Design for Type Safety

- only define child-related operations in the Composite class
- the type system enforces type constraints, so a client cannot perform child-related operations on a Leaf object.
- a client needs to treat Leaf and Composite objects differently.
- useful for static structures where a client doesn’t need to perform child-related operations on “unknown” objects of type Component.

# Composite Pattern: Demo Example

Read the example code discussed/developed in the lectures, and also provided for this week



```
Component mainboard = new Composite("Mainboard", 100);
Component processor = new Leaf("Processor", 450);
Component memory = new Leaf("Memory", 80);
mainboard.add(processor);
mainboard.add(memory);

Component chasis = new Composite("Chasis", 75);
chasis.add(mainboard);

Component disk = new Leaf("Disk", 50);
chasis.add(disk);

System.out.println("[0] " + processor.nameString());
System.out.println("[0] " + processor.calculateCost());

System.out.println("[1] " + mainboard.nameString());
System.out.println("[1] " + mainboard.calculateCost());

System.out.println("[2] " + chasis.nameString());
System.out.println("[2] " + chasis.calculateCost());
```

This example uses design for **Uniformity** (see composite.uniformity). Sample code also includes design for **Type Safety** (see composite.typesafe).

```

public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");
        MenuComponent coffeeMenu = new Menu("COFFEE MENU", "S

        MenuComponent allMenus = new Menu("ALL MENUS", "All m

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        pancakeHouseMenu.add(new MenuItem(
            "K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99));
        pancakeHouseMenu.add(new MenuItem(
            "Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99));
    }
}

```

## Composite Pattern: Demo Example

Read the example code discussed/developed in the lectures, and also provided for this week

ALL MENUS, All menus combined

PANCAKE HOUSE MENU, Breakfast

- K&B's Pancake Breakfast(v), 2.99
  - Pancakes with scrambled eggs, and toast
- Regular Pancake Breakfast, 2.99
  - Pancakes with fried eggs, sausage
- Blueberry Pancakes(v), 3.49
  - Pancakes made with fresh blueberries, and blueberry syrup
- Waffles(v), 3.59
  - Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch

- Vegetarian BLT(v), 2.99
  - (Fakin') Bacon with lettuce & tomato on whole wheat
- BLT, 2.99
  - Bacon with lettuce & tomato on whole wheat
- Soup of the day, 3.29
  - A bowl of the soup of the day, with a side of potato salad
- Hotdog, 3.05

# Demos ...

- Live Demos ...
- Make sure you **properly understand** the demo example code available for this week.

# Summary

- The Composite Pattern provides a structure to hold both individual objects and composites.
- The Composite Pattern allows clients to treat composites and individual objects uniformly.
- A Component is any object in a Composite structure. Components may be other composites or leaf nodes.
- There are many design tradeoffs in implementing Composite. You need to balance transparency/uniformity and type safety with your needs.

From the reference book: “Head First Design Pattern”

# COMP2511

# Observer Pattern

Prepared by  
Dr. Ashesh Mahidadia

# Observer Pattern

These lecture notes use material from the wikipedia page at: [https://en.wikipedia.org/wiki/Observer\\_pattern](https://en.wikipedia.org/wiki/Observer_pattern)

and

the reference book “[Head First Design Patterns](#)”.

# Observer Pattern

- The **Observer Pattern** is used to implement distributed **event handling** systems, in "event driven" programming.
- In the observer pattern
  - an object, called the **subject** (or **observable** or **publisher**) , maintains a list of its dependents, called **observers** (or **subscribers**), and
  - **notifies** the *observers* **automatically** of any state **changes** in the **subject**, usually by calling one of their methods.
- Many programming languages support the observer pattern, Graphical User Interface libraries use the observer pattern extensively.

# Observer Pattern

- The Observer Pattern defines a **one-to-many** dependency between objects so that when one object (*subject*) changes state, all of its dependents (*observers*) are notified and updated automatically.
- The aim should be to,
  - define a one-to-many dependency between objects **without** making the objects **tightly coupled**.
  - **automatically** notify/update an **open-ended** number of *observers* (dependent objects) when the *subject* changes state
  - be able to **dynamically** add and remove *observers*

# Observer Pattern: Possible Solution

- Define *Subject* and *Observer* **interfaces**, such that when a subject changes state, all registered observers are notified and updated automatically.
- The **responsibility of**,
  - a **subject** is to maintain a list of observers and to notify them of state changes by calling their **update()** operation.
  - **observers** is to register (and unregister) themselves on a subject (to get notified of state changes) and to update their state when they are notified.
- This makes subject and observers **loosely coupled**.
- Observers can be **added** and **removed** independently **at run-time**.
- This notification-registration interaction is also known as **publish-subscribe**.

# Java Observer and Observable : Deprecated

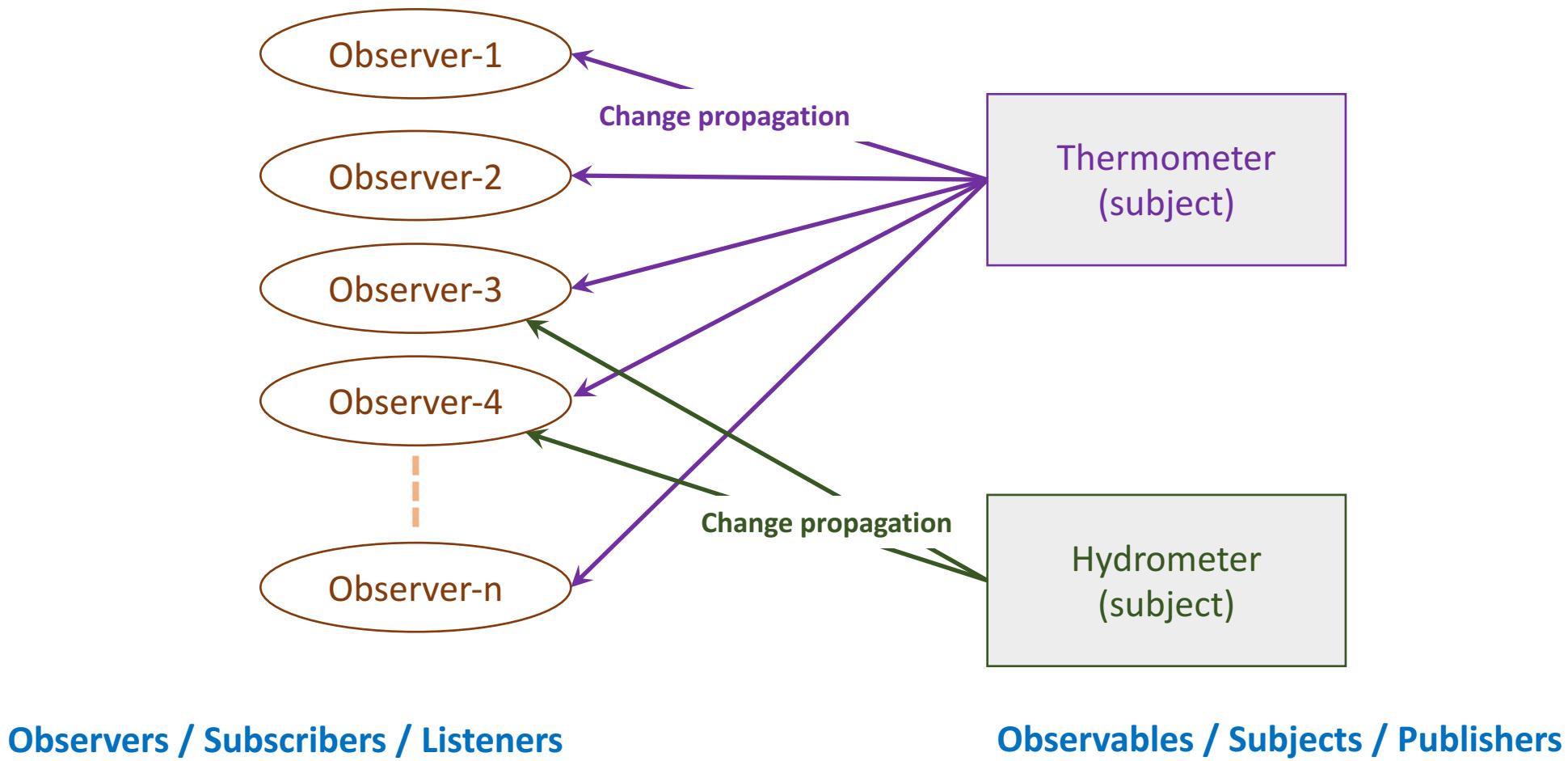
The following java library classes have been **deprecated** in Java **9** because the model implemented was quite **limited**.

- [java.util.Observer](#) and
- [java.util.Observable](#)

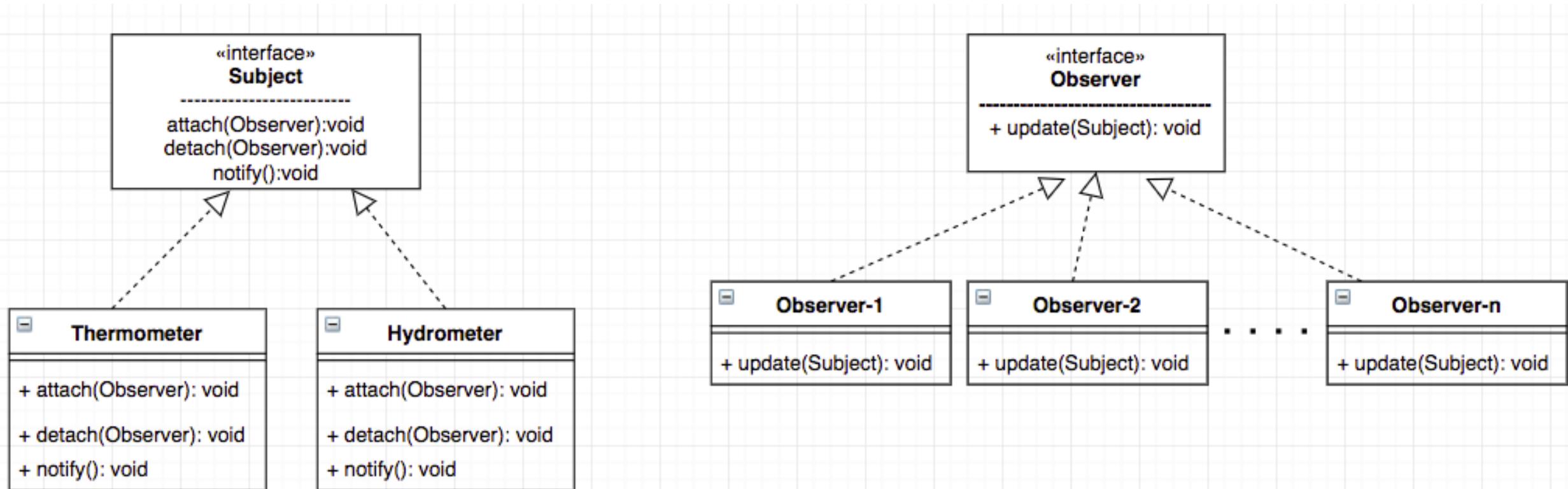
## Limitations

- *Observable* is a **class**, not an interface !
- Observable **protects** crucial methods, the `setChanged()` method is protected.
- we can't call `setChanged()` unless we subclass *Observable*! Inheritance is must, bad design 😊
- we can't add on the *Observable* behavior to an existing class that already extends another superclass.
- there isn't an *Observable* interface, for a proper custom implementation

# Multiple Observers and Subjects



# Observer Pattern: Possible Solution



```
ArrayList<Observer> list0bservers = new ArrayList<Observer>();  
  
public void notify0bservers() {  
    for( Observer obs : list0bservers) {  
        obs.update(this);  
    }  
}
```

Read the example code  
discussed/developed in the lectures,  
and also provided for this week

# Passing data: Push or Pull

The *Subject* needs to pass (change) data while notifying a change to an *Observer*. Two possible options,

## Push data

- *Subject* passes the changed data to its observers, for example:  
`update(data1, data2, ...)`
- All *observers* must implement the above update method.

## Pull data

- *Subject* passes reference to itself to its observers, and the observers need to get (pull) the required data from the subject, for example:  
`update(this)`
- Subject needs to provide the required access methods for its observers.  
For example,    `public double getTemperature() ;`

```
public interface Subject {  
  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}
```

```
public class Thermometer implements Subject {  
  
    ArrayList<Observer> listObservers = new ArrayList<Observer>();  
    double temperatureC = 0.0;  
  
    @Override  
    public void registerObserver(Observer o) {  
        if(! listObservers.contains(o)) { listObservers.add(o); }  
    }  
  
    @Override  
    public void removeObserver(Observer o) {  
        listObservers.remove(o);  
    }  
  
    @Override  
    public void notifyObservers() {  
        for( Observer obs : listObservers) {  
            obs.update(this);  
        }  
    }  
  
    public double getTemperatureC() {  
        return temperatureC;  
    }  
  
    public void setTemperatureC(double temperatureC) {  
        this.temperatureC = temperatureC;  
        notifyObservers();  
    }  
}
```



Notify Observers  
after every update

Read the example code  
discussed/developed in the lectures,  
and also provided for this week

```
public interface Observer {  
    public void update(Subject obj);  
}
```

Update for  
Multiple Subjects

Display after an update

Read the example code  
discussed/developed in the lectures,  
and also provided for this week

```
public class DisplayUSA implements Observer {  
    Subject subject;  
    double temperatureC = 0.0;  
    double humidity = 0.0;  
  
    @Override  
    public void update(Subject obj) {  
  
        if(obj instanceof Thermometer) {  
            update( (Thermometer) obj);  
        }  
        else if(obj instanceof Hygrometer) {  
            update((Hygrometer) obj);  
        }  
    }  
  
    public void update(Thermometer obj) {  
        this.temperatureC = obj.getTemperatureC();  
        display();  
    }  
    public void update(Hygrometer obj) {  
        this.humidity = obj.getHumidity();  
        display();  
    }  
  
    public void display() {  
        System.out.printf("From DisplayUSA: Temperature is %.2f F, "  
                         + "Humidity is %.2f\n", convertToF(), humidity);  
    }  
  
    public double convertToF() {  
        return (temperatureC *(9.0/5.0) + 32);  
    }  
}
```

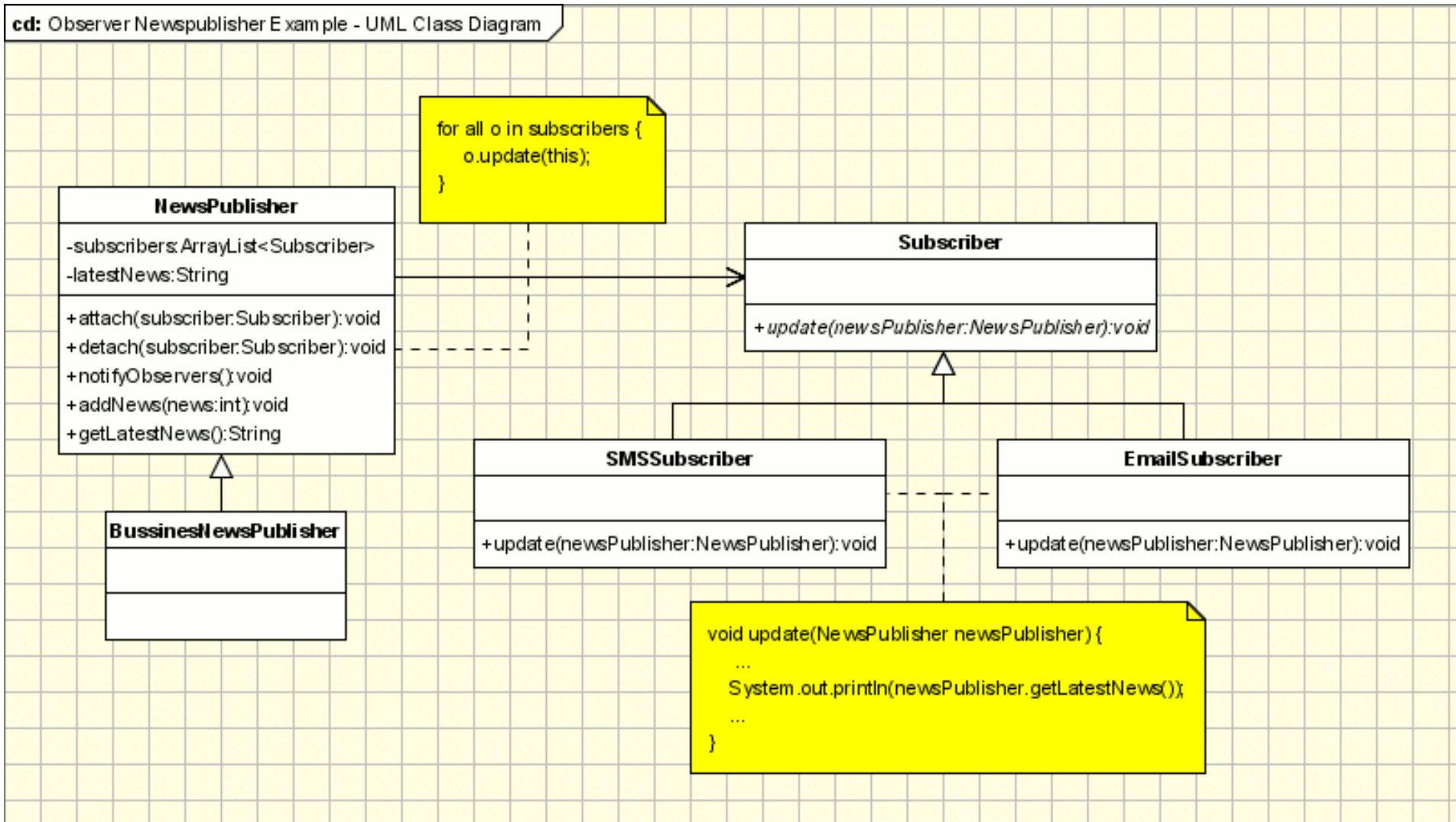
Read the example code  
discussed/developed in the lectures,  
and also provided for this week

```
public class Test1 {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        Thermometer thermo = new Thermometer();  
        Observer usaDisplay = new DisplayUSA();  
        thermo.registerObserver(usaDisplay); ← add / register  
  
        Observer ausDisplay = new DisplayAustralia();  
        thermo.registerObserver(ausDisplay);  
  
        System.out.println("\n----- thermo.setTemperatureC(30) ----- ");  
        thermo.setTemperatureC(30);  
        System.out.println("\n----- thermo.setTemperatureC(12) ----- ");  
        thermo.setTemperatureC(12); ← change state  
  
        Hygrometer hyg = new Hygrometer();  
        hyg.registerObserver(usaDisplay);  
  
        System.out.println("\n----- hyg.setHumidity(77) ----- ");  
        hyg.setHumidity(77);  
        System.out.println("\n----- hyg.setHumidity(96) ----- ");  
        hyg.setHumidity(96);  
        System.out.println("\n----- thermo.setTemperatureC(35) ----- ");  
        thermo.setTemperatureC(35);  
  
        thermo.removeObserver(usaDisplay); ← remove  
        System.out.println("\n----- thermo.removeObserver(usaDisplay) ----- ");  
  
        System.out.println("\n----- thermo.setTemperatureC(41) ----- ");  
        thermo.setTemperatureC(41);  
        System.out.println("\n----- ----- ");  
    }  
}
```

# Demos ...

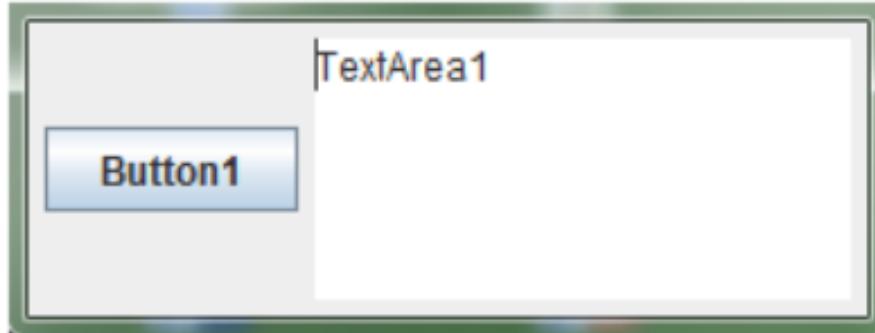
- Live Demos ...
- Make sure you **properly understand the demo example code** available for this week.

# Observer Pattern: Example



The above image is from <https://www.odesign.com/observer-pattern.html>

# Observer Pattern: UI Example



# Summary

## Advantages:

- Avoids tight coupling between *Subject* and its *Observers*.
- This allows the *Subject* and its *Observers* to be at different levels of abstractions in a system.
- **Loosely coupled** objects are easier to maintain and reuse.
- Allows **dynamic** registration and deregistration.

## Be careful:

- A change in the subject may result in a chain of updates to its observers and in turn their dependent objects – resulting in a **complex update behaviour**.
- Need to properly manage such dependencies.

# Summary

## BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects, or as we also know them, Observables, update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer interface.
- You can push or pull data from the Observable when using the pattern (pull is considered more “correct”).
- Don’t depend on a specific order of notification for your Observers.
- Java has several implementations of the Observer Pattern, including the general purpose `java.util.Observable`.
- Watch out for issues with the `java.util.Observable` implementation.
- Don’t be afraid to create your own Observable implementation if needed.
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You’ll also find the pattern in many other places, including JavaBeans and RMI.

From the reference book: “Head First Design Pattern”

# COMP2511

## Refactoring

Prepared by  
Ashesh Mahidadia

# Refactoring: Motivation

- ❖ Code refactoring is the process of restructuring existing computer code **without changing its external behavior**.
- ❖ Originally Martin Fowler and Kent Beck defined refactoring as,  
“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior... It is a disciplined way to clean up code that minimizes the chances of introducing bugs.”
- ❖ **Advantages:** improved code readability, reduced complexity; improved maintenance and extensibility
- ❖ **If done well**, helps to identify *hidden* or *dormant* bugs or vulnerabilities, by simplifying code logic.
- ❖ **If done poorly**, may change external behavior, and/or introduce new bugs!
- ❖ Refactoring is **different to** adding features and debugging.

## Refactoring: Motivation

- ❖ Refactoring is usually motivated by noticing a *code smell* (*possible bad design/coding practices*).
- ❖ Code Smell is a **hint** that something might be wrong, **not** a certainty.
- ❖ Identifying a Code Smell allows us to **re-check** the implementation details and consider possible *better* alternatives.
- ❖ Automatic **unit tests** should be set up before refactoring to ensure routines still behave as expected.
- ❖ Refactoring is an **iterative** cycle of making a small program **transformation**, **testing** it to ensure correctness, and making another small **transformation**.

# Software Maintenance

- ❖ Software Systems **evolve over time** to meet new requirements and features.
- ❖ Software maintenance involve:
  - Fix bugs
  - Improve performance
  - Improve design
  - Add features
- ❖ Majority of software maintenance is for the last three points!
- ❖ **Harder to maintain** code than write from scratch!
- ❖ **Most** of the development **time** is spent in **maintenance**!
- ❖ **Good design**, coding and planning can reduce maintenance pain and time!
- ❖ **Avoid** code smells to reduce maintenance pain and time!

## Code Smells: Possible Indicators

- ❖ Duplicated code
- ❖ Poor abstraction (change one place → must change others)
- ❖ Large loop, method, class, parameter list; deeply nested loop
- ❖ Class has too little cohesion
- ❖ Modules have too much coupling
- ❖ Class has poor encapsulation
- ❖ A subclass doesn't use majority of inherited functionalities
- ❖ A “data class” has little functionality
- ❖ Dead code
- ❖ Design is unnecessarily general
- ❖ Design is too specific

# Low-level refactoring

- ❖ **Names:**
  - ❖ Renaming (methods, variables)
  - ❖ Naming (extracting) “magic” constants
- ❖ **Procedures:**
  - ❖ Extracting code into a method
  - ❖ Extracting common functionality (including duplicate code) into a class/method/etc.
  - ❖ Changing method signatures
- ❖ **Reordering:**
  - ❖ Splitting one method into several to improve cohesion and readability (by reducing its size)
  - ❖ Putting statements that semantically belong together near each other
- ❖ For more, see <http://www.refactoring.com/catalog/>

# IDEs support low-level refactoring

- ❖ Renaming:
  - Variable, method, class.
- ❖ Extraction:
  - Method, constant
  - Repetitive code snippets
  - Interface from a type
- ❖ Inlining: method, etc.
- ❖ Change method signature.
- ❖ Warnings about inconsistent code.

## Higher-level refactoring

- ❖ Refactoring to design patterns.
- ❖ Changing language idioms (safety, brevity).
- ❖ Performance optimization.
- ❖ Generally high-level refactoring is **much more important**,  
but unfortunately **not** well-supported by tools.

# Code Smells

# Code and Design Smells

**Smells** : Design aspects that violate fundamental design principles and impact software quality

Smells occur at different levels of granularity

- Code Smells: Structures in implementation of code such as large methods, classes with multiple responsibilities, complex conditional statements that lead to poor code
- Design Smells: Design aspects at a higher level of abstraction (class level abstractions) such as classes with multiple responsibilities, refused bequest

Regardless of the granularity, smells in general indicate violation of software design principles, and eventually lead to code that is rigid, fragile and require “refactoring”

# Smells

**Bloaters:** Code, Methods and classes that have grown in size, that they are hard to work with

- Long Method, Large Class, Long Parameter List, Data Clumps

**OO Abusers:** Result from incorrect or incomplete application of OO principles

- Switch statements, Refused Bequest

**Change Preventers:** Code changes are difficult (rigid code)

- Divergent change, Shot Gun Surgery

**Dispensables:** Code that is pointless and unnecessary

- Comments, Data Class, Lazy Class, Duplicate code

**Couplers:** Excessive coupling between classes

- Feature Envy, Inappropriate intimacy, Message Chains

# Smell: Long Method

## Fix smell, long method

- Reduce length of a method body via [Extract Method](#)
  - More readable, Less code duplication
  - Isolates independent parts of code, - errors are less likely
- If local variables and parameters interfere with extracting a method, use
  - [Replace Temp With Query](#)
  - [Introduce Parameter Object](#)
  - [Preserve Whole Object](#)
- If the above doesn't work, try moving the entire method to a separate object via [Replace Method with Method Object](#)
- [Replace Method with Method Object](#)
- Conditional operators and loops are a good clue that code can be moved to a separate method.

# Refactoring Techniques – Extract Method

- More readable code (The new method name should describe the method's purpose)
- Less code duplication, more reusability
- Isolates independent parts of code, meaning that errors are less likely
- A very common refactoring technique for code smells

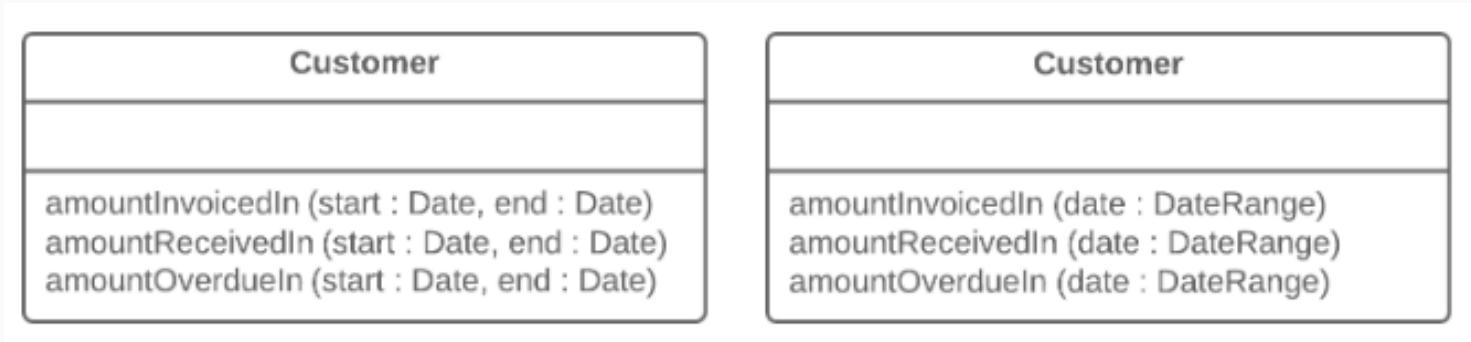
```
public void debit(float amount) {  
    // deducts amount by balance  
    balance -= amount;  
  
    // records transaction  
    transactions.add(new Transaction(amount, true));  
  
    // record last transaction date  
    lastTransactionDate = LocalDate.now().toString();  
}
```



```
public void debit(float amount) {  
    deductBalance(amount);  
    recordTransaction(amount, true);  
    recordLastTransaction();  
}  
  
private void deductBalance(float amount) {  
    balance -= amount;  
}  
private void recordTransaction(float amount, boolean isDebit) {  
    transactions.add(new Transaction(amount, isDebit));  
}  
private void recordLastTransaction() {  
    lastTransactionDate = LocalDate.now().toString();  
}
```

# Refactoring Techniques: Introduce Parameter Object

- Methods contain a repeating group of parameters, causing code duplication
- Consolidate these parameters into a separate class
  - Also helps to move the methods for handling this data
  - Beware, if only data is moved to a new class and associated behaviours are not moved, this begins to smell of a **Data Class**



- Eliminates smell such as Long Parameter List, Data Clumps, Primitive Obsession, Long Method

# Refactoring Technique: Replace Temp With Query

Often, we place the result of an expression in a local variable for later use in the code

With Replace Temp With Query we:

- Move the entire expression to a separate method and return the result from it.
- Query the method instead of using a variable
- Reuse the new method in other methods

```
double calculateTotal() {  
    double basePrice = quantity * itemPrice;  
    if (basePrice > 1000) {  
        return basePrice * 0.95;  
    }  
    else {  
        return basePrice * 0.98;  
    }  
}
```

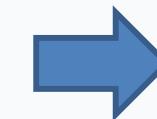
```
double calculateTotal() {  
    if (basePrice() > 1000) {  
        return basePrice() * 0.95; }  
    else {  
        return basePrice() * 0.98; } }  
double basePrice() {  
    return quantity * itemPrice;  
}
```

- Eliminates smell such as Long Method, Duplicate Code

# Refactoring Technique: Extract Class

- Having all the phone details in class Customer is not a good OO design and also breaks SRP
- Refactor into two separate classes, each with its appropriate responsibility

```
public class Customer {  
    private String name;  
    private String workPhoneAreaCode;  
    private String workPhoneNumber;  
    private String homePhoneAreaCode;  
    private String homePhoneNumber;  
}
```



```
public class Customer {  
    private String name;  
    private Phone workPhone;  
    private Phone homePhone;  
}  
  
public class Phone {  
    public String areaCode;  
    public String phoneNumber;  
}
```

# Smell: Large Class

- Problem:
- Similar to Long Method
- Usually violates Single Responsibility Principle
- May have
  - A large number of instance variables
  - Several methods
- Typically lacks cohesion and potential for duplicate code smell

## Solution:

- Bundle group of variables via [Extract Class](#) or [Extract Sub-Class](#)

# Code Smell: Long Parameter List

**Problem:** Calling a query method and passing its results as the parameters of another method, while that method could call the query directly

- Too many parameters to remember
- Bad for readability, usability and maintenance

```
public String getSummary() {
    return buildCustomerSummary(getFirstName(), getLastName(), getTitle(),
        address.getCity(), address.getPostCode());
}

private String buildCustomerSummary(String firstName, String lastName,
    String title, String city, String postCode) {
    return title + " " + firstName + " " + lastName + "," + city + "," + "postcode";
}
```

## Solution:

- try placing a query call inside the method body via **replace**

```
// 1. Apply replace parameter with method call
// 2. Apply change method signature to remove the first three parameters
// 3. Preserve whole object - passing in the entire object instead of object data
private String buildCustomerSummary(Address address) {
    return getTitle() + " " + getFirstName() + " " + getLastName() + "," + address.getCity() + ","
        + address.getPostCode();
}
```

# Code Smell: Data Clumps

## Problem:

- Different parts of the code contain identical groups of variables e.g., fields in many classes, parameters in many method signatures
- Can lead to code smell Long Parameter List

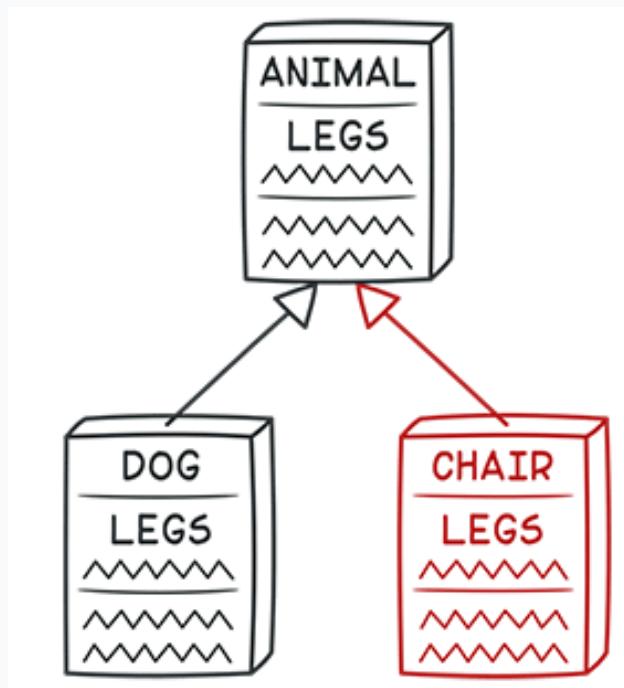
## Solution: Move the behaviour to the data class via **Move Method**

- If repeating data comprises the fields of a class, use **Extract Class** to move the fields to their own class.
- If the same data clumps are passed in the parameters of methods, use **Introduce Parameter Object** to set them off as a class.
- If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields **Preserve Whole Object** will help with this.

# Code Smell: Refused Bequest

## Problem:

- A subclass uses only some of the methods and properties inherited from its parents
- The unneeded methods may simply go unused or be redefined and give off exceptions
- Often caused by creating inheritance between classes only by the desire to reuse the code in a super-class



# Code Smell: Refused Bequest

## Solution:

- If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favour of **Replace Inheritance with Delegation**
- If inheritance is appropriate, but super class contains fields and methods not applicable to all classes, then consider the following options
  - Create a new subclass
  - Apply **Push Down Field** to move field relevant only to subclass from superclass
  - Apply **Push Down Method** to move behaviour from super class to sub class, as behaviour makes sense only to sub class
  - Often, you may apply an **Extract Sub-Class Class** to combine the above steps

# Refused Bequest Example

class Camel does not use field model. It should be pushed down to class Car

```
public abstract class Transport {  
    // Push Down Field  
    private String model;  
    // Push Down Method  
    public String getModel() throws Exception  
{  
    return model;  
}  
...  
}  
public class Car extends Transport { ... }  
public class Camel extends Transport {  
...  
    public String getModel() {  
        throw new NotSupportedException();  
    }  
}  
public abstract class Transport {
```

```
// Use Push Down Field to move field and  
// Push Down Method to move behaviour  
// only relevant to sub class  
// from super class to sub class
```

```
public abstract class Transport {  
    ...  
}  
public class Car extends Transport {  
    private String model;  
    public String getModel()  
    {  
        return model;  
    }  
    ...  
}  
public class Camel extends Transport {  
    ...  
}
```

# Code Smell: Duplicate Code

Code Fragments look similar

- If the same code is found in two or more methods in the same class:  
use **Extract Method** and place calls for the new method in both places
- If the same code is found in two subclasses of the same level:
  - Use **Extract Method** for both classes, followed by **Pull Up Field** for the fields used in the method that you are pulling up.
  - If the duplicate code is inside a constructor, use **Pull Up Constructor Body**
  - If the duplicate code is similar but not completely identical, use **Form Template Method**
  - If two methods do the same thing but use different algorithms, select the best algorithm and apply **Substitute Algorithm**
- If duplicate code is found in two different classes:
  - If the classes are not part of a hierarchy, use **Extract SuperClass** in order to create a single superclass for these classes that maintains all the previous functionality

# Code Smell: Feature Envy

**Problem:** A method that is more interested in a class other than the one it actually is

- Invokes several methods on another object to calculate some value
- Creates unnecessary coupling between the classes

**Solution:** A goal of OO design is to put the methods with its associated data

- So the method must moved to the relevant class via **Move Method**
- If only part of a method accesses the data of another object, use **Extract Method** followed by **Move Method** to move the part in question
- If a method uses functions from several other classes, first determine which class contains most of the data used. Then place the method in this class along with the other data.

# Code Smell: Divergent Change, Shot Gun Surgery

**Divergent Change:** One class is changed in different ways for different reasons

- **Solution:** Any change to handle a variation should change a single class, and all the typing in the new class should express the variation.
- To clean this up you identify everything that changes for a particular cause and use Extract Class to put them all together

**Shot Gun Surgery:** A small change in the code forces lots of little changes to different classes

- **Solution:**
  - Use **Move Method** or **Move Field** to put all the changes into a single class
  - Often you can use **Inline Class** to bring a whole bunch of behaviour together.
- Divergent change is one class that suffers many kinds of changes, and shotgun surgery is one change that alters many classes.

# Code Smell: Data Classes

**Problem:** Classes that just have attributes with setters and getters and no behaviour

One of the goals of OO design is to put behaviour where the data is

```
public class CustomerSummaryView {  
    private Customer customer;  
    public CustomerSummaryView(Customer customer) {  
        this.customer = customer;  
    }  
    public String getCustomerSummary() {  
        Address addr = customer.getAddress();  
        return customer.getTitle() + " " + customer.getFirstName() + " "  
        + customer.getLastName() + "," + addr.getCity() + "," + addr.getPostCode();  
    }  
}
```

**Solution:** Move the behaviour to the data class via [Move Method](#)

```
// 1. Apply move method inside method getCustomerSummary() in address class  
//     to move method to class Customer  
// 2. Extract the address summary  
// 3. Move the address summary to the class Address  
  
public String getCustomerSummary() {  
    return getTitle() + " " + getFirstName() + " "  
    + getLastName() + "," + address.getAddressSummary();  
}
```

# Code Smell: Lazy classes

**Problem:** Classes that aren't doing much to justify their existence (maintenance overhead)

Subclasses without any overridden methods or additional fields can be lazy classes as well

```
public class PostCode {  
    private String postcode;  
  
    public PostCode(String postcode) {  
        this.postcode = postcode;  
    }  
  
    public String getPostcode() {  
        return postcode;  
    }  
  
    public String getPostcodeArea(){  
        return postcode.split("")[0];  
    }  
}
```

```
public class Address {  
  
    private final String number;  
    private final String street;  
    private final String city;  
    private final String country;  
    private final PostCode postcode;  
  
    public Address(String no, String st, String city,  
                  PostCode pCode, String country) {  
        this.number = no;  
        this.street = st;  
        this.city = city;  
        this.postcode = pCode;  
        this.country = country;  
    }  
}
```

**Solution:**

- Move the data (postcode) from lazy class PostCode to the class Address
- Delete the lazy class

# Code Smell: Switch Statements

**Problem:** Switch statements are bad from an OO design point of view

```
public RiskFactor calculateMotoristRisk() {  
    if (motorist.getPointsOnLicense() > 3 || motorist.getAge() < 25) {  
        return RiskFactor.HIGH;  
    }  
    if (motorist.getPointsOnLicense() > 0 ) {  
        return RiskFactor.MEDIUM;  
    }  
    return RiskFactor.LOW;  
}  
  
public double calculateInsurancePremium(double insuranceValue) {  
    RiskFactor riskFactor = calculateMotoristRisk();  
    switch(riskFactor) {  
        case LOW:  
            return insuranceValue * 0.02;  
        case MEDIUM:  
            return insuranceValue * 0.04;  
        default:  
            return insuranceValue * 0.06;  
    }  
}
```

**Solution:** Replace switch statements with a polymorphic solution based on [Strategy Pattern](#) applying a series of refactoring techniques (*Extract Method, Move Method, Extract Interface etc., Refer lecture demo for complete solution*)

# List of Refactoring Techniques to be familiar

Move Field/Method

Extract Class/Inline Class

Extract Method

Inline Method/Temporary Variable

Replace Temp with Query

Replace Method with Method Object

Rename Method

Substitute Algorithm

Introduce Parameter Object

Preserve Whole Object

Extract Sub Class/Super Class/Interface

Extract Method

Pull Up Field/Method/Constructor Body

Form Template Method

Replace Inheritance with Delegation

Replace Conditional with Polymorphism

# Useful Links

<https://refactoring.guru/refactoring/smells>

<https://www.refactoring.com/catalog/>

# COMP2511

## Creational Pattern: Singleton Pattern

Prepared by

Dr. Ashesh Mahidadia

# Creational Patterns

Creational patterns provide various **object creation** mechanisms, which increase flexibility and reuse of existing code.

## ❖ Factory Method

- provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

## ❖ Abstract Factory

- let users produce families of related objects without specifying their concrete classes.

## ❖ Singleton

- Let users ensure that a class has only one instance, while providing a global access point to this instance.

# Singleton Pattern

# Singleton Pattern

**Intent:** Singleton is a creational design pattern that lets you ensure that a class has **only one instance**, while providing a global access point to this instance.

**Problem:** A client wants to,

- ❖ ensure that a class has just a **single instance**, and
- ❖ provide a **global** access point to that instance

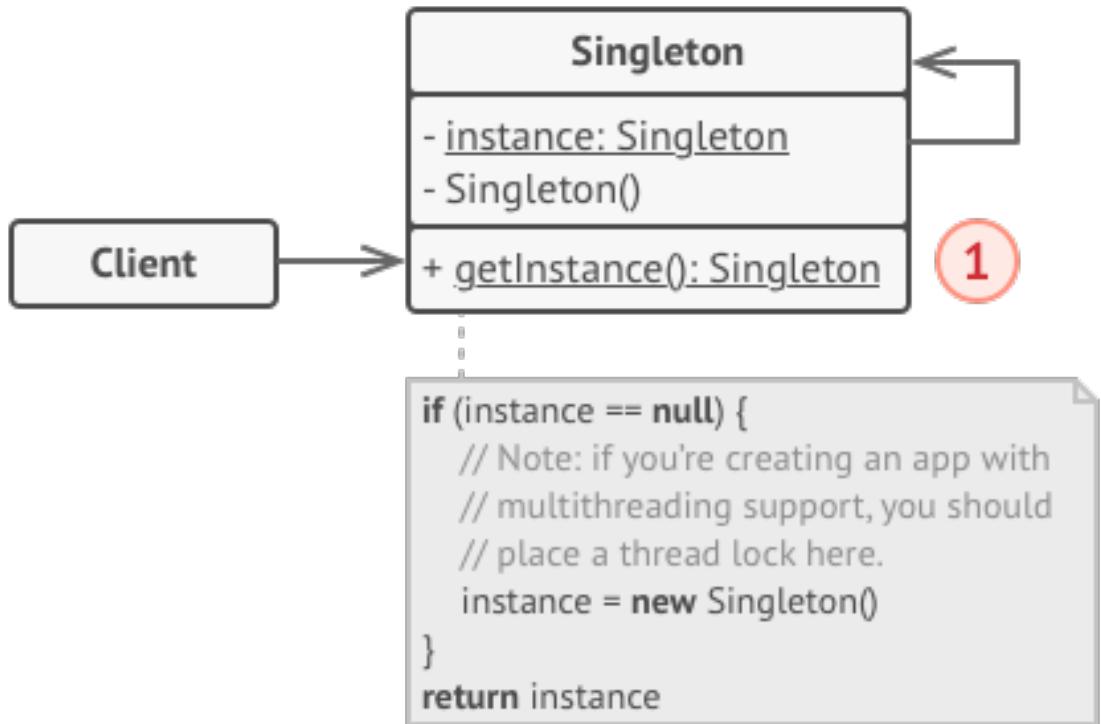
**Solution:**

All implementations of the Singleton have these two steps in common:

- ❖ Make the **default constructor private**, to prevent other objects from using the new operator with the Singleton class.
- ❖ Create a **static creation method** that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the **cached object**.
- ❖ If your code has access to the Singleton class, then it's able to **call the Singleton's static method**.
- ❖ Whenever Singleton's static method is called, the **same object** is always returned.

# Singleton: Structure

- ❖ The **Singleton** class declares the **static** method *getInstance* (1) that returns the same instance of its own class.
- ❖ The Singleton's constructor should be hidden from the client code.
- ❖ Calling the *getInstance* (1) method should be the only way of getting the Singleton object.



# Singleton: How to Implement

- ❖ Add a **private static field** to the class for storing the singleton instance.
- ❖ Declare a **public static creation method** for getting the singleton instance.
- ❖ Implement “lazy initialization” inside the static method.
  - It should create a **new object** on its first call and put it into the static field.
  - The method should always return that instance on all **subsequent calls**.
- ❖ Make the **constructor of the class private**.
  - The static method of the class will still be able to call the constructor, but not the other objects.
- ❖ In a client, call singleton’s static creation method to access the object.

Example in Java (**MUST read**):

<https://refactoring.guru/design-patterns/singleton/java/example>

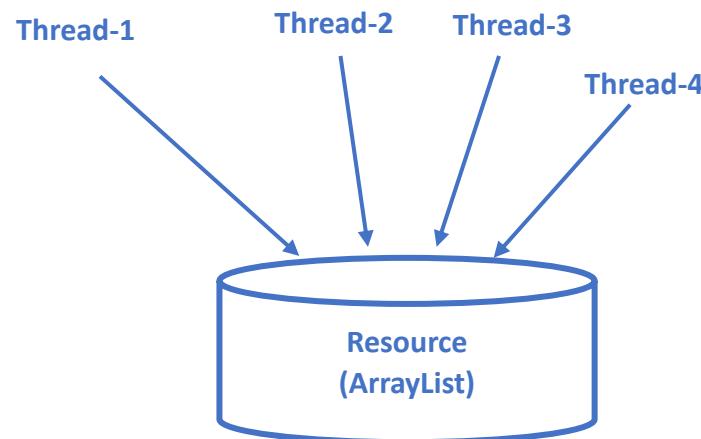
# Singleton Pattern

For more information, read:

<https://refactoring.guru/design-patterns/singleton>

# Introduction to Concurrency

- Several modern languages, including Java, allow for concurrent execution of multiple threads.
- To make the most of today's multi-core hardware, we must create applications that employ multiple threads.
- Therefore, having a fundamental understanding of concurrency is essential!
- *Thread safe:* **many threads** can access the **same resources** without exposing incorrect behaviour or causing unpredictable outcomes.
- Unfortunately, many Java libraries lack thread safety. Examples are ArrayList, StringBuilde, etc.



# Thread Safety

- **Time slicing** in Java refers to the process of allocating time to threads.
- The order in which the threads run is **uncertain**. It is also unpredictable how many statements of one thread run before some of the other thread's statements run.
- Two threads modifying the same object (data) may operate in parallel.

```
public class Account {  
    private balance int = 500;  
  
    ...  
    ...  
  
    public void withdraw(int amt){  
        int old_balance = balance;  
        ...  
        // checking ... takes 500 mil secs  
        ...  
        balance = old_balance - amt;  
        ...  
    }  
    ...  
}
```

Let's create an object for the account number 1234,

```
Account a1 = new Account(1234);
```

If two threads **thread-1** and **thread-2** call the following method 100 mil secs apart (for example from two browser windows), both may be successful, even if the balance is only \$500!

```
a1.withdraw(400);
```

# A possible solution using synchronized

- A **synchronized** method acquires the lock of the object or class at the start, executes the method, and then releases the lock at the end.
- The use of **synchronized** key word allows **only one thread** to execute the method, avoiding concurrency issues.
- To make the most efficient use of the several CPUs available, a portion of the code (under synchronized) that accesses a shared resource must be **kept to a minimum**.
- We can also *synchronized* a set of statements, however it is a good practice to synchronize a method.
- Java offers **thread-safe collection wrappers**, using static methods. For example,  
`Collections.synchronizedList(list)`
- `Java.util.concurrent` package contains collections that are suitable and optimized for multiple threads.

```
public class Account {  
    private balance int = 500;  
  
    ...  
  
    public synchronized void withdraw(int amt){  
        int old_balance = balance;  
        ...  
        // checking ... takes 500 mil secs  
        ...  
        balance = old_balance - amt;  
        ...  
    }  
    ...  
}
```



## Need to avoid ....

- When two or more threads are stuck waiting for each other indefinitely, the condition is referred to as a **deadlock**.
- When two or more threads are caught in an endless cycle of reacting to one another, this is known as a **livelock**.
- **Starvation** occurs when one or more threads are unable to progress due to another "greedy" thread.

## A lot more to concurrency ..

- There is much more to concurrency than what we have briefly addressed here; however, it is beyond the scope of this course.

# An example of Singleton pattern using synchronized

Demo: Let's see what happens when we use synchronized and when we don't.

**End**

COMP2511

# Decorator Pattern

Prepared by

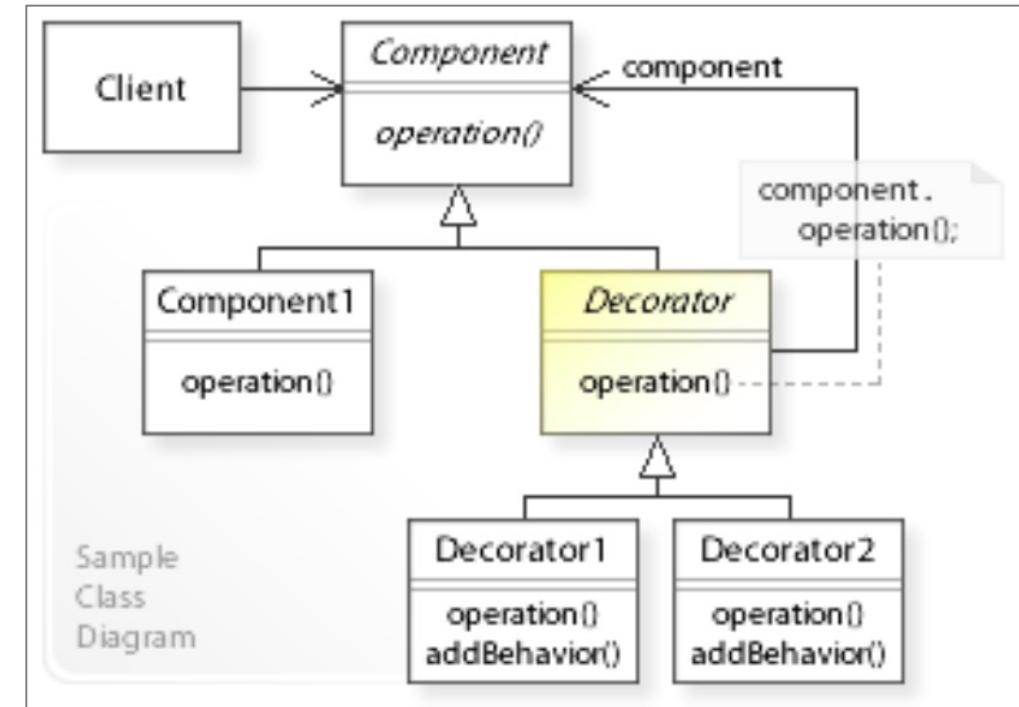
Dr. Ashesh Mahidadia

# Decorator Pattern: Intent

- "Attach additional responsibilities to an object **dynamically**.  
Decorators provide a flexible alternative to sub-classing for extending functionality."  
[GoF]
- **Decorator design patterns** allow us to selectively add **functionality to an object** (not the class) at runtime, based on the requirements.
- Original class is **not** changed (Open-Closed Principle).
- **Inheritance** extends behaviors at **compile time**, additional functionality is bound to **all** the instances of that class for their life time.
- The **decorator** design pattern prefers a **composition** over an inheritance.  
Its a **structural pattern**, which provides a **wrapper** to the existing class.
- Objects can be **decorated multiple times**, in different order, due to the **recursion** involved with this design pattern. See the example in the Demo.
- Do not need to implement all possible functionality in a single (complex) class.

# Decorator Pattern: Structure

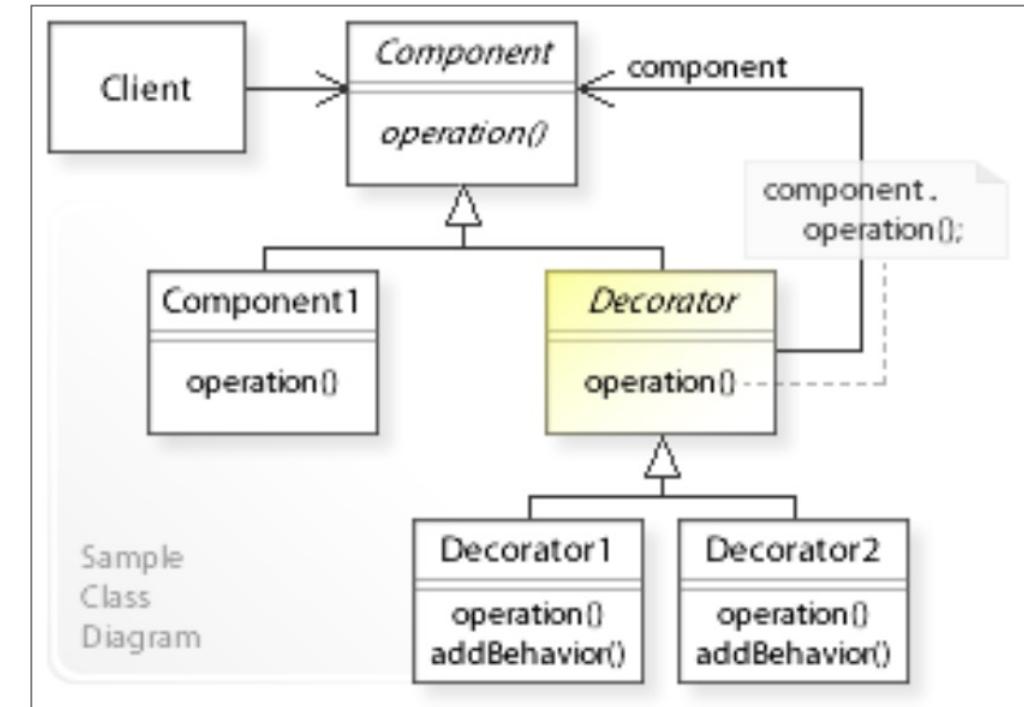
- ❖ *Client* : refers to the Component interface.
- ❖ *Component*: defines a common interface for *Component1* and *Decorator* objects
- ❖ *Component1* : defines objects that get decorated.
- ❖ *Decorator*: maintains a reference to a *Component* object, and forwards requests to this component object (*component.operation()*)
- ❖ *Decorator1, Decorator2, ...* :  
Implement additional functionality (*addBehavior()*) to be performed before and/or after forwarding a request.



See the example in the Demo.

# Decorator Pattern: Structure

- ❖ Given that the decorator has the same supertype as the object it decorates, we can pass around a **decorated** object in place of the **original** (wrapped) object.
- ❖ The **decorator adds its own** behavior either before and/or after delegating to the object it decorates to do the rest of the job.



From the book “Head First Design Pattern”.

See the example in the Demo.

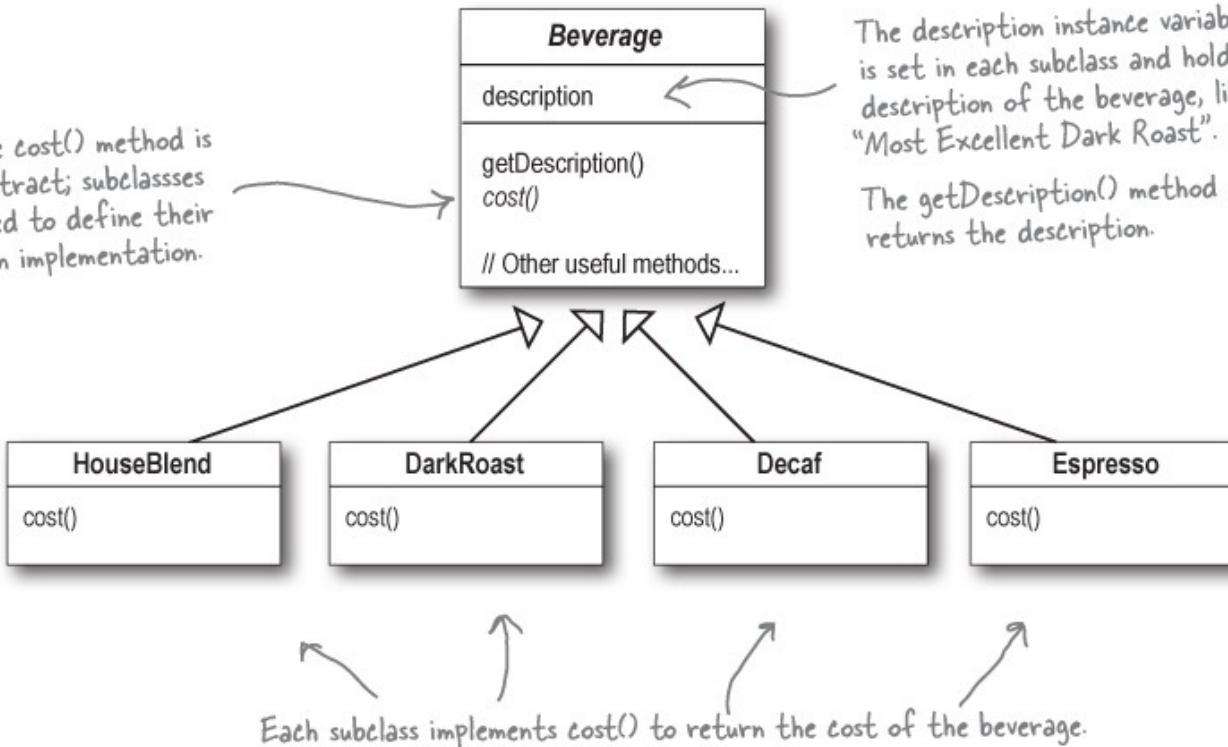
# Decorator Pattern: Example

Welcome to Starbuzz Coffee



Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The cost() method is abstract; subclasses need to define their own implementation.

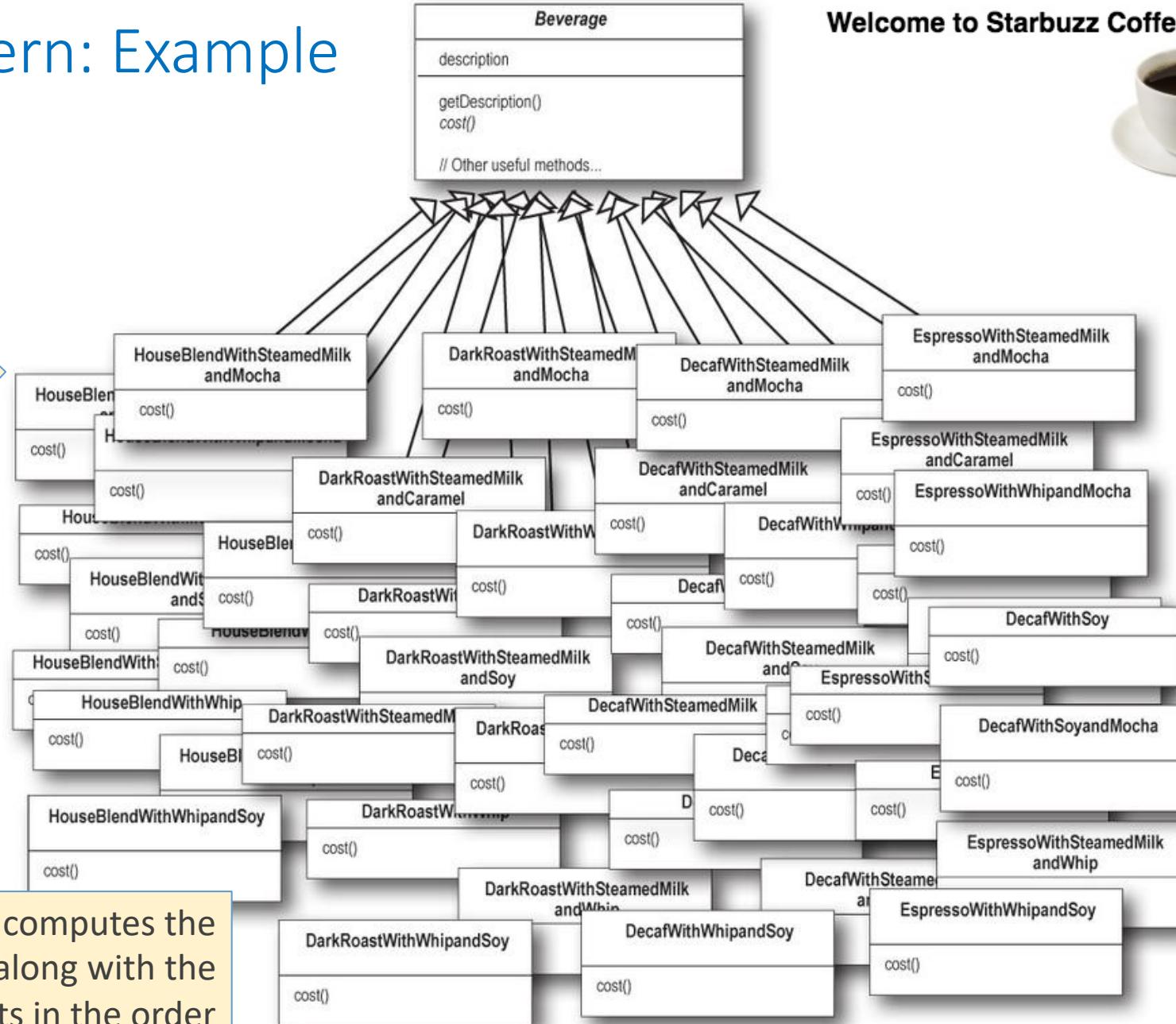


# Decorator Pattern: Example

Welcome to Starbuzz Coffee

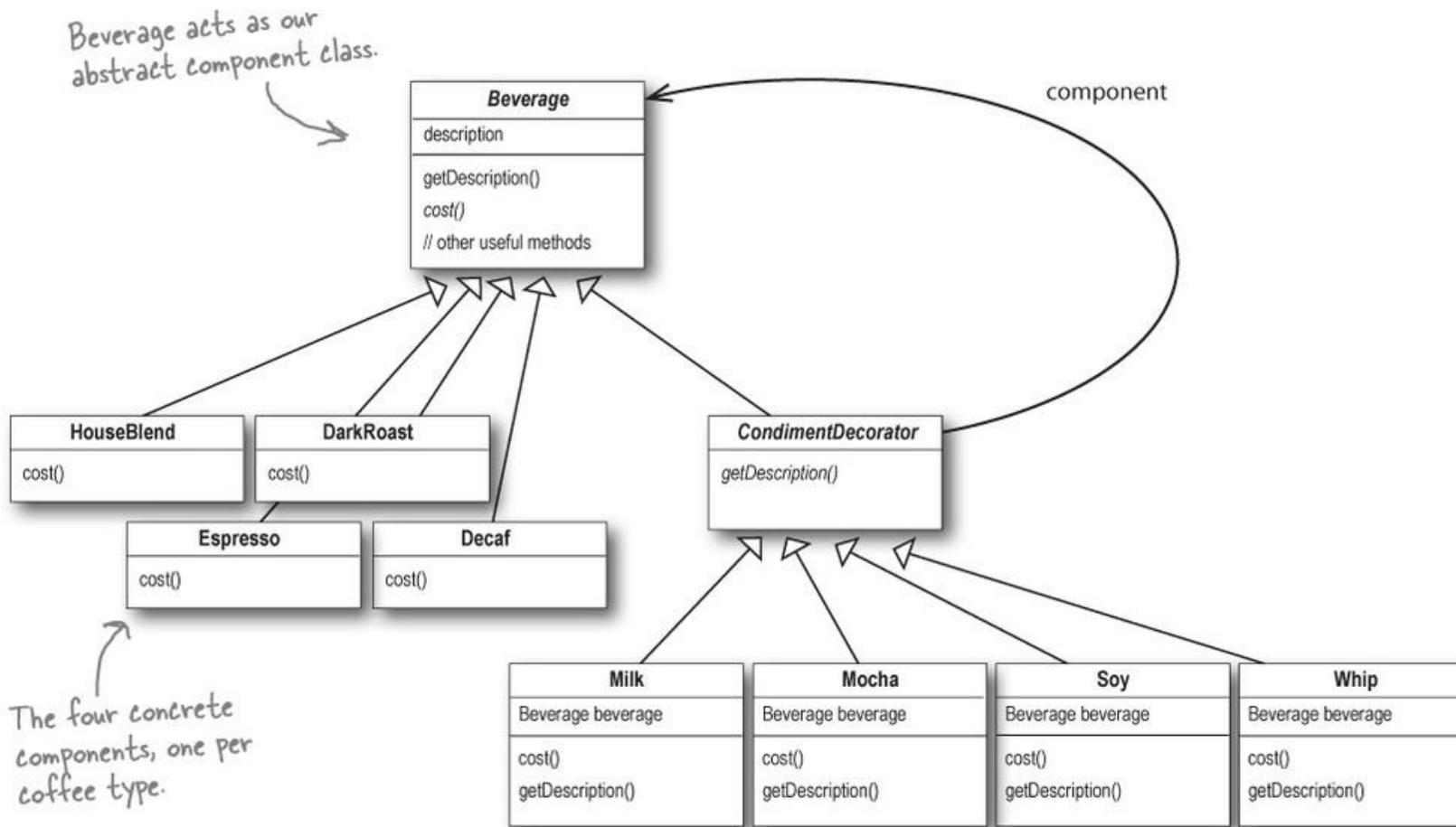


Too many combinations  
to consider!



# Decorator Pattern: Example

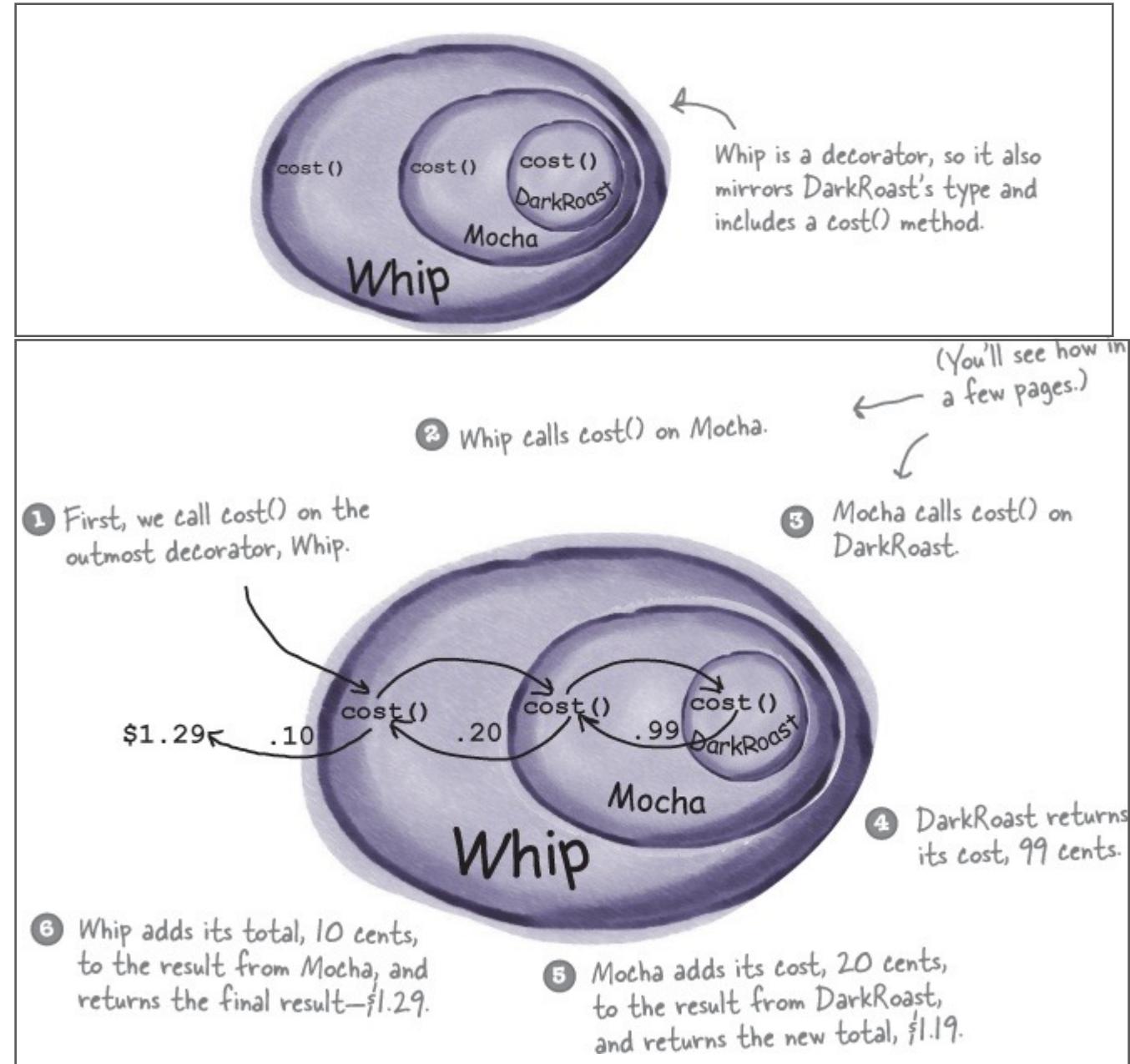
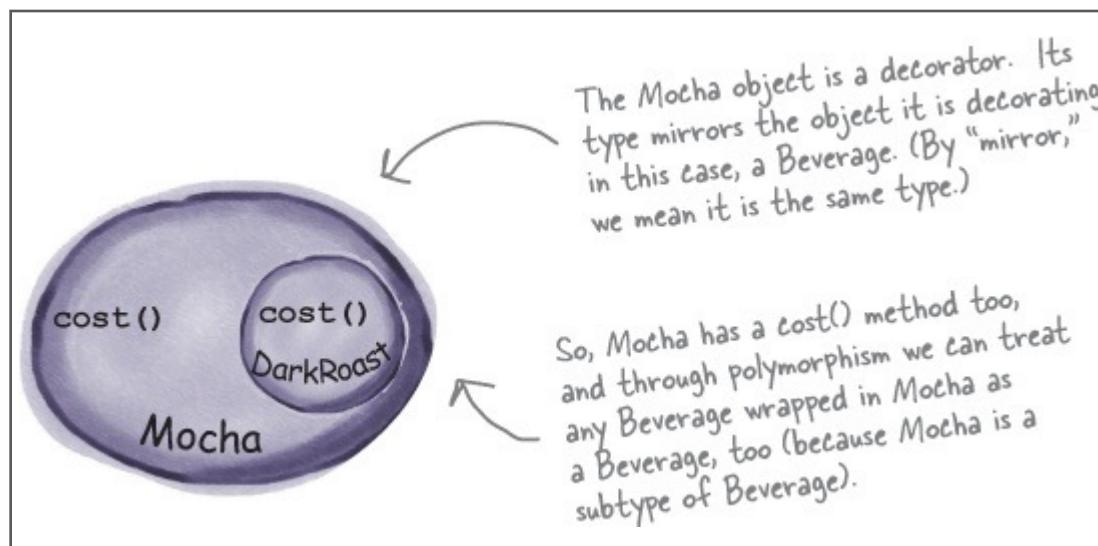
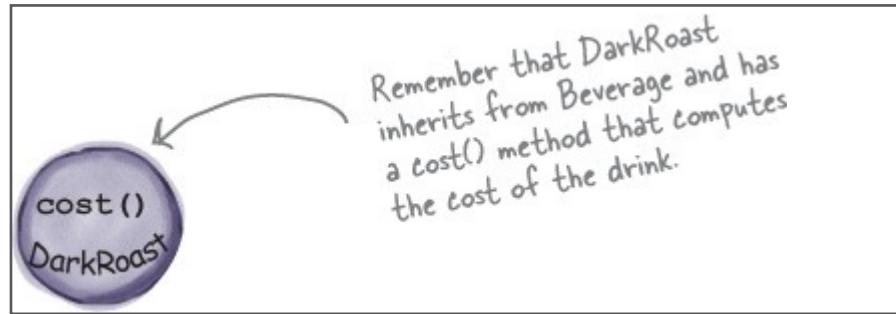
Welcome to Starbuzz Coffee



And here are our condiment decorators; notice they need to implement not only `cost()` but also `getDescription()`. We'll see why in a moment...

# Decorator Pattern: Example

## Constructing a drink order with Decorators



# Decorator Pattern: Code

```
Beverage beverage = new Espresso();
System.out.println(beverage.getDescription()
    + " $" + beverage.cost());
System.out.println("-----");
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
    + " $" + beverage2.cost());

System.out.println("-----");

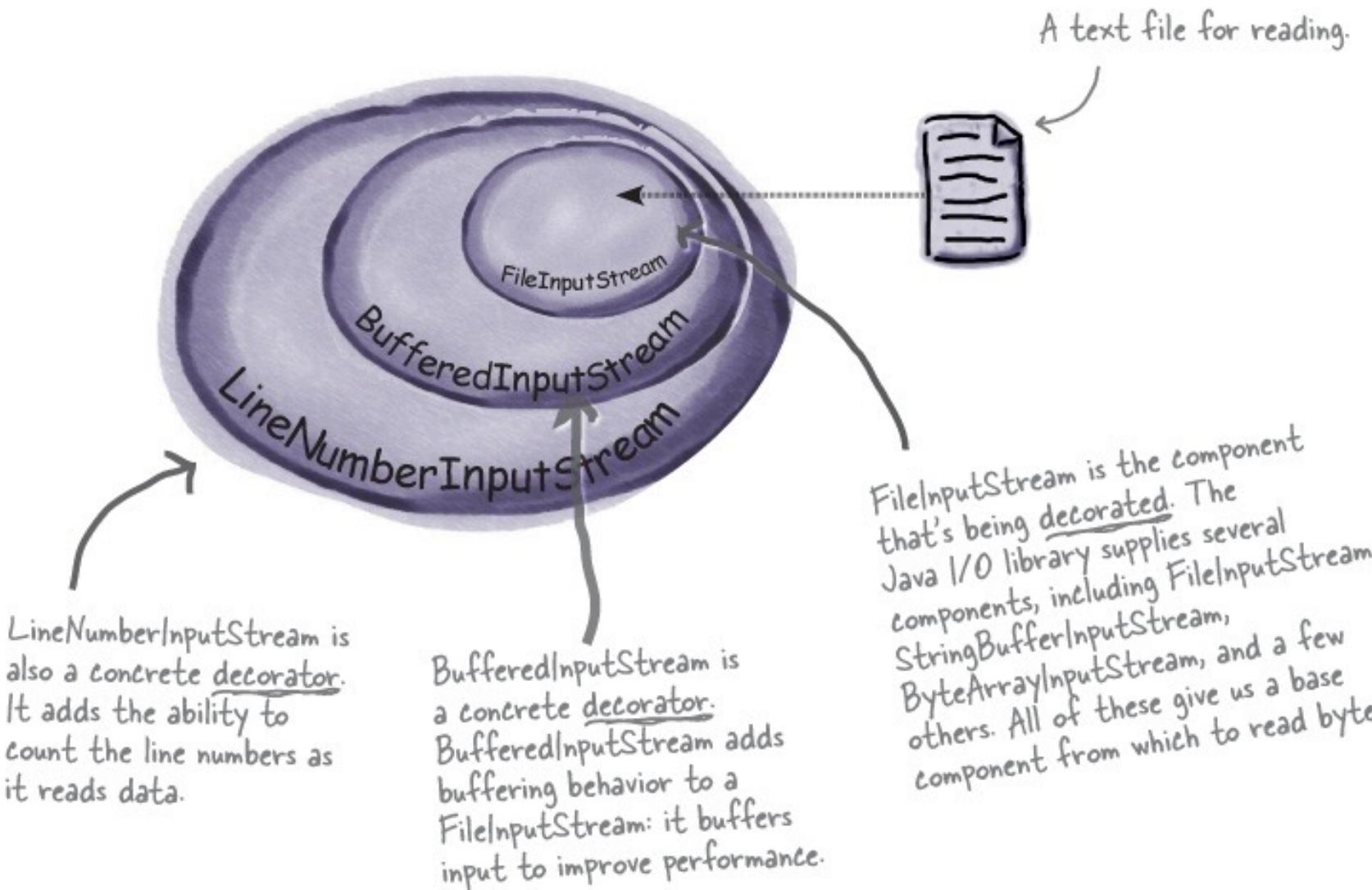
Beverage beverage3 = new HouseBlend();
beverage3 = new Soy(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
System.out.println(beverage3.getDescription()
    + " $" + beverage3.cost());
System.out.println("-----")
```

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Whipe: beverage.cost() is: " + beverage_cost);
    System.out.println(" - adding One Whip cost of 0.10c ");
    System.out.println(" - new cost is: " + (0.10 + beverage_cost) );
    return 0.10 + beverage_cost ;
}
```

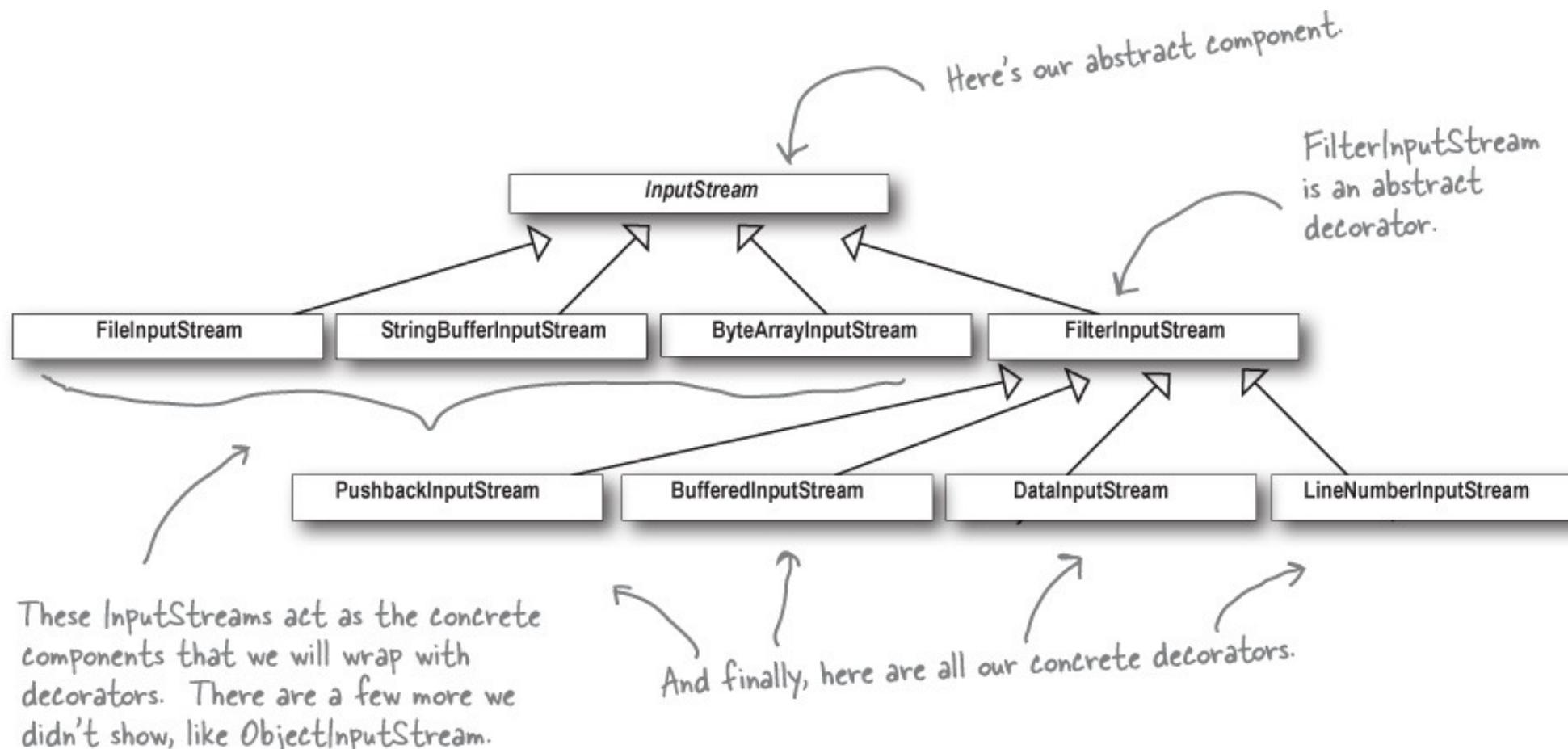
Read the example code  
discussed/developed in the lectures,  
and also provided for this week

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Mocha: beverage.cost() is: " + beverage_cost );
    System.out.println(" - adding One Mocha cost of 0.20c ");
    System.out.println(" - new cost is: " + (0.20 + beverage_cost) );
    return 0.20 + beverage_cost ;
}
```

# Decorator Pattern: Java I/O Example



# Decorator Pattern: Java I/O Example



# Decorator Pattern: Code

```
InputStream f1 = new FileInputStream(filename);
InputStream b1 = new BufferedInputStream(f1);
InputStream lCase1 = new LowerCaseInputStream(b1);
InputStream rot13 = new Rot13(b1);

while ((c = rot13.read()) >= 0) {
    System.out.print((char) c);
}
```

Read the example code  
discussed/developed in the lectures,  
and also provided for this week

## Decorator Pattern:

- Demo ...

**End**

# COMP2511

## Creational Patterns:

Factory Method  
Abstract Factory Pattern

Prepared by  
Dr. Ashesh Mahidadia

# Creational Patterns

# Creational Patterns

Creational patterns provide various **object creation** mechanisms, which increase flexibility and reuse of existing code.

## ❖ Factory Method

- provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

## ❖ Abstract Factory

- let users produce families of related objects without specifying their concrete classes.

## ❖ Builder

- let users construct complex objects step by step. The pattern allows users to produce different types and representations of an object using the same construction code.

## ❖ Singleton

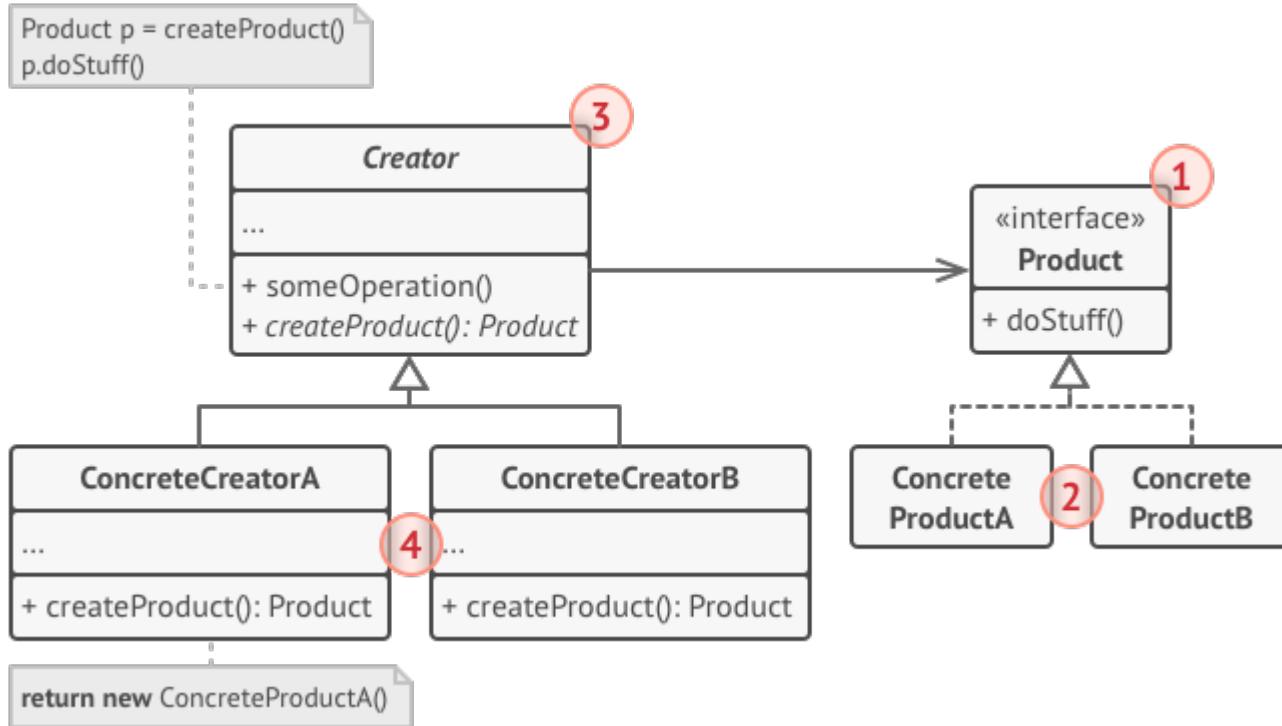
- Let users ensure that a class has only one instance, while providing a global access point to this instance.

# Factory Method

# Factory Method

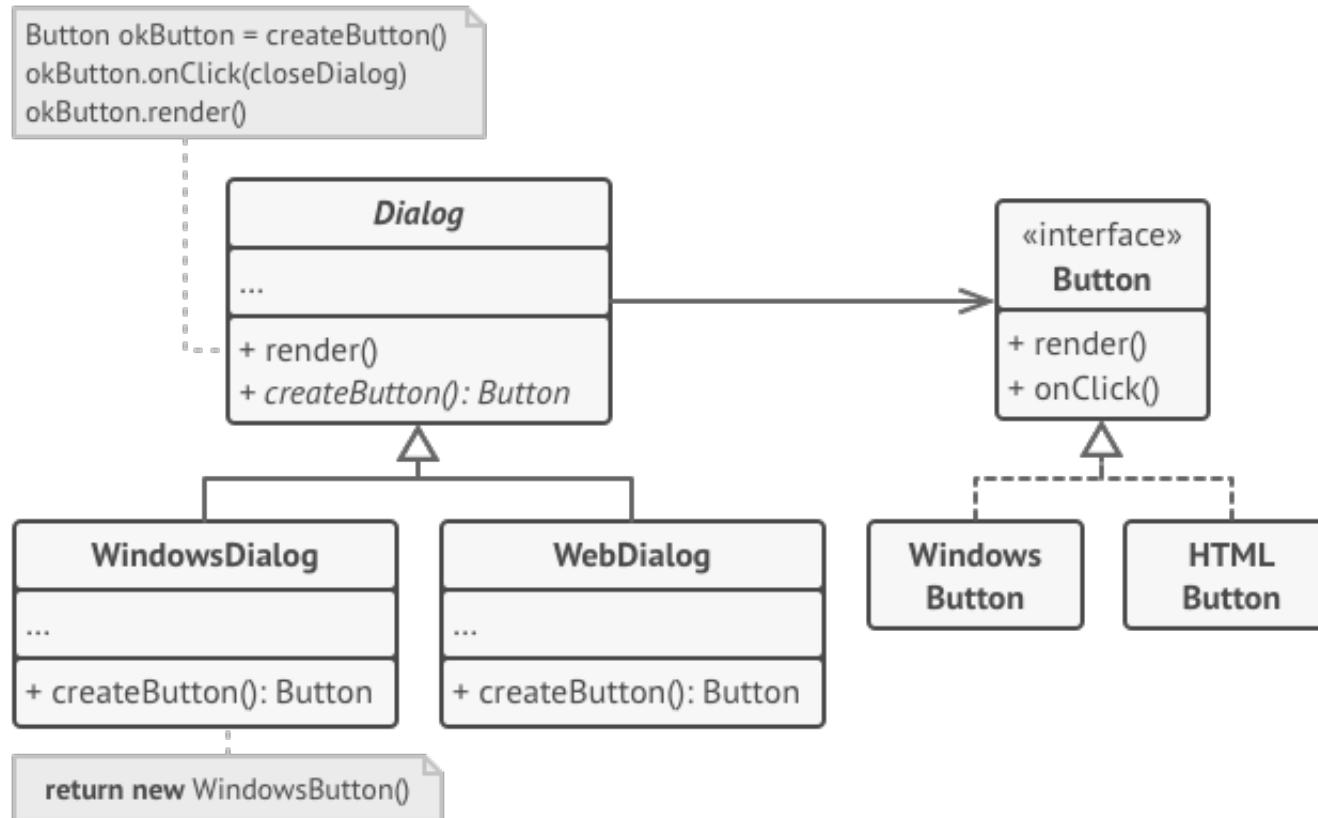
- ❖ **Factory Method** is a creational design pattern that uses factory methods to deal with the problem of creating objects **without** having to **specify the exact class** of the object that will be created.
- ❖ **Problem:**
  - creating an object directly within the class that requires (uses) the object is **inflexible**
  - it **commits** the class to a particular object and
  - makes it **impossible to change** the instantiation independently from (without having to change) the class.
- ❖ **Possible Solution:**
  - Define a **separate** operation (**factory method**) for creating an object.
  - Create an object by calling a **factory method**.
  - This enables writing of subclasses to change the way an object is created (to redefine which class to instantiate).

# Factory Method : Structure



1. The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
2. **Concrete Products** are different implementations of the product interface.
3. The **Creator** class declares the factory method that returns new product objects.
4. **Concrete Creators** override the base factory method so it returns a different type of product.

# Factory Method : Example



Example in Java (**MUST** read):

<https://refactoring.guru/design-patterns/factory-method/java/example>

# Factory Method

For more, read the following:

<https://refactoring.guru/design-patterns/factory-method>

# Abstract Factory Pattern

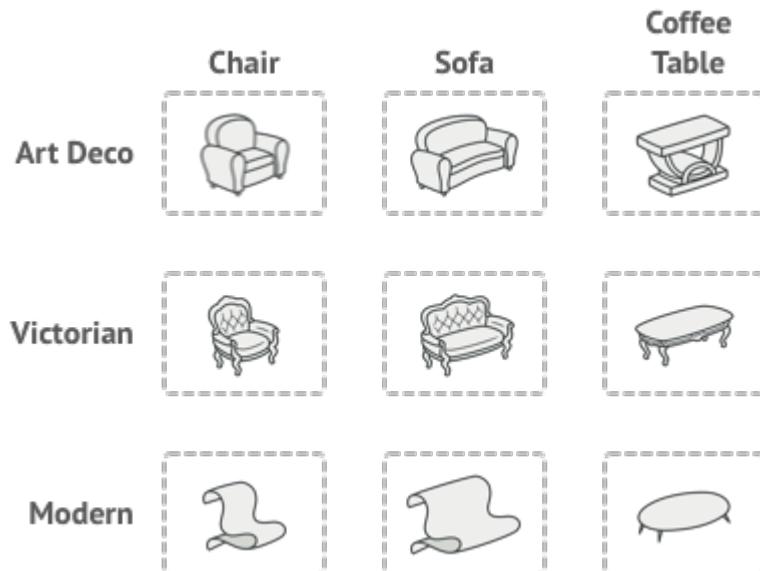
# Abstract Factory Pattern

**Intent:** Abstract Factory is a creational design pattern that lets you produce **families of related objects** without specifying their concrete classes.

## Problem:

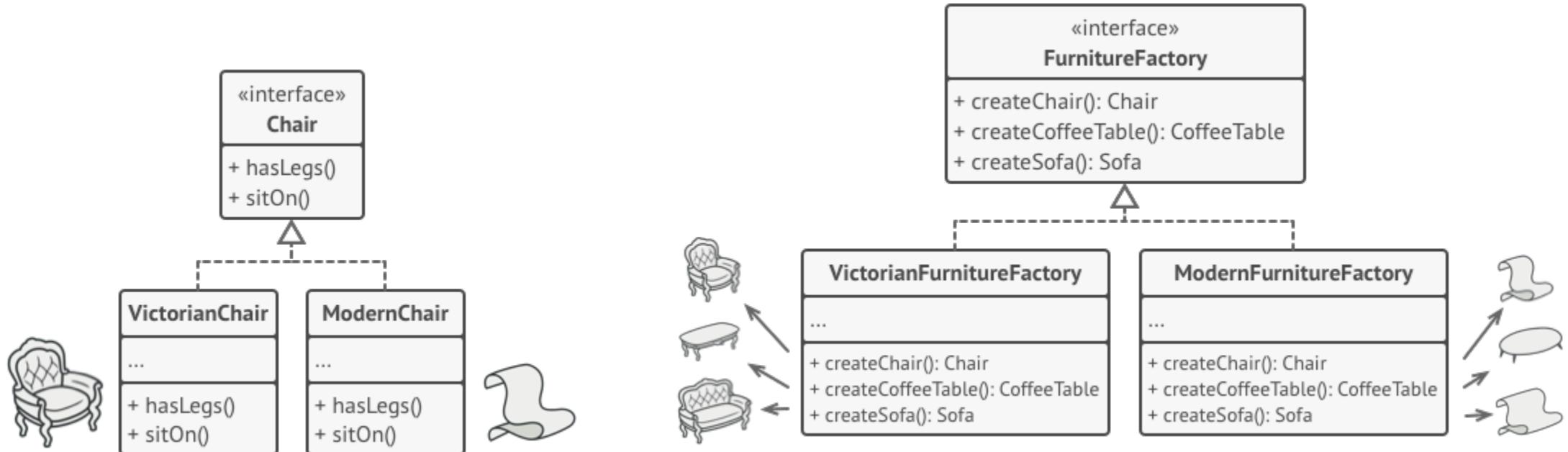
Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

- ❖ A family of related products, say: **Chair + Sofa + CoffeeTable**.
- ❖ Several variants of this family.
- ❖ For example, products **Chair + Sofa + CoffeeTable** are available in these **variants**:

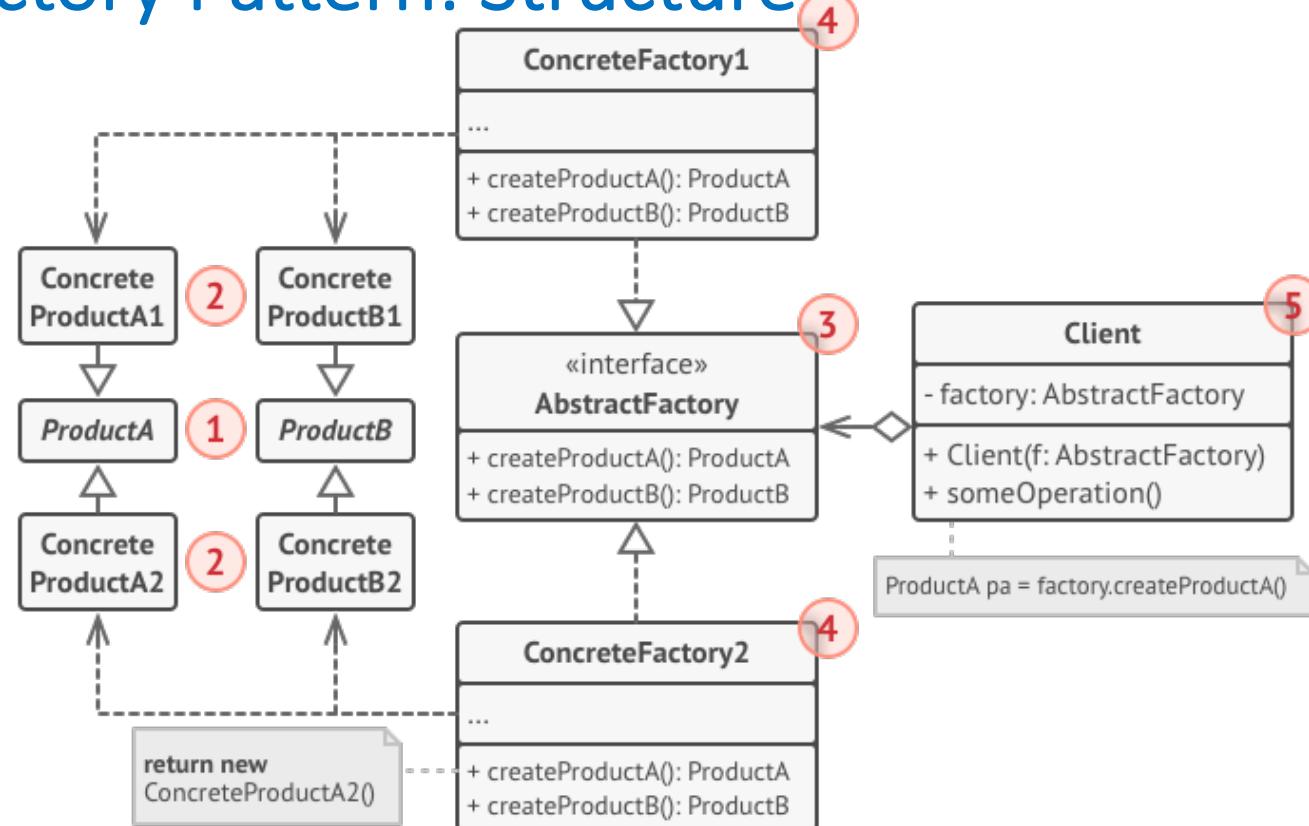


# Abstract Factory Pattern:

Possible Solution:

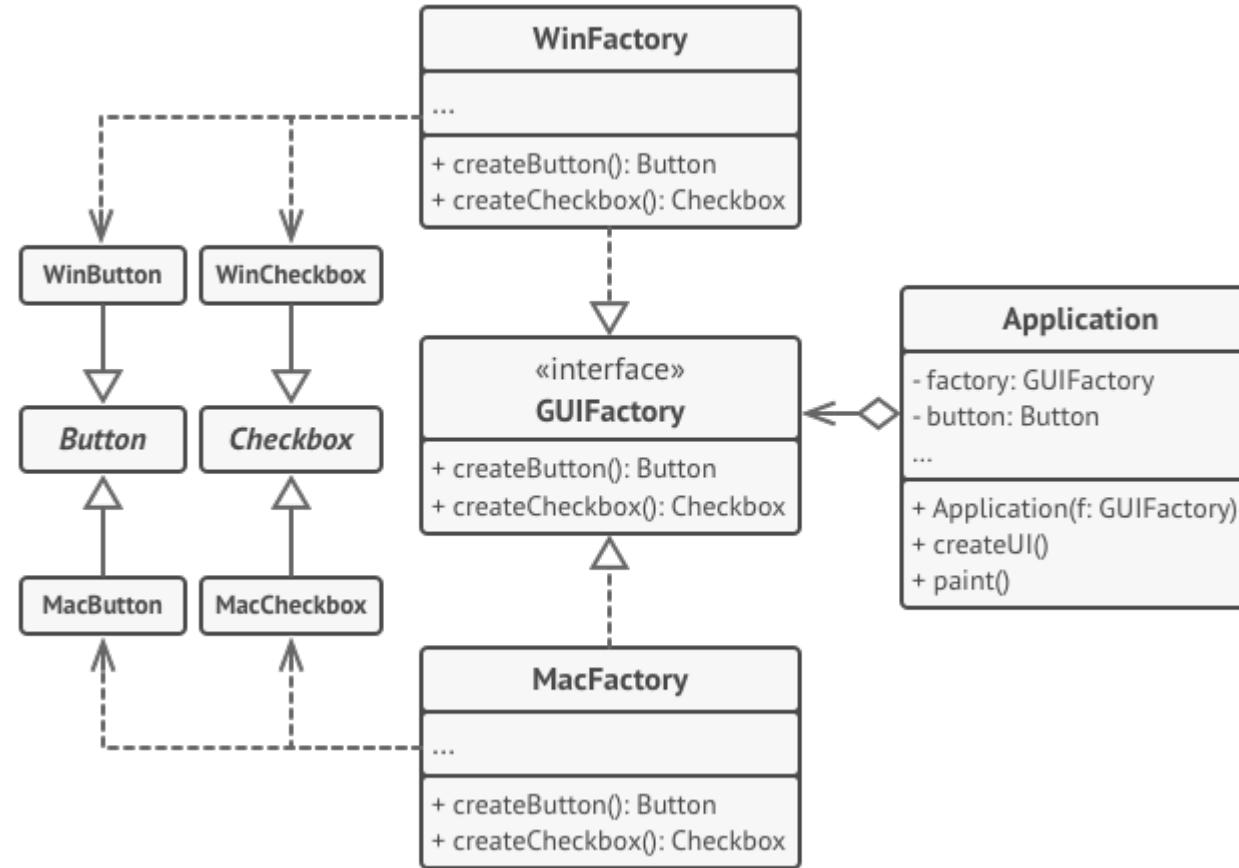


# Abstract Factory Pattern: Structure



1. **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
2. **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).
3. The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
4. **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.
5. The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

# Abstract Factory Pattern: Example



Example in Java (**MUST** read):

<https://refactoring.guru/design-patterns/abstract-factory/java/example>

# Abstract Factory Pattern

For more, read the following:

<https://refactoring.guru/design-patterns/abstract-factory>

**End**

# COMP2511

## Generics in Java (Part 2)

Prepared by

Dr. Ashesh Mahidadia

# Generics in Java: Java Tutorial

- ❖ Good introduction at the following web page, Oracle's official Java Tutorial, you must **read all the relevant pages!**

<https://docs.oracle.com/javase/tutorial/java/generics/index.html>

- ❖ The following lecture slides cover only some parts of the above tutorial, however, you should read all the relevant sections (pages) in the above tutorial.

# Generics in Java (Recap)

Generics enable **types** (classes and interfaces) to be **parameters** when defining:

- classes,
- interfaces and
- methods.

## Benefits

- ❖ Removes *casting* and offers stronger type checks at compile time.
- ❖ Allows implementations of generic algorithms, that work on collections of different types, can be customized, and are type safe.
- ❖ Adds stability to your code by making more of your bugs detectable at compile time.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Without Generics

```
List<String> listG = new ArrayList<String>();
listG.add("hello");
String sg = listG.get(0); // no cast
```

With Generics

# Generic Types (Recap)

- ❖ A generic type is a generic **class** or **interface** that is **parameterized** over types.
- ❖ A generic class is defined with the following format:  
`class name< T1, T2, ..., Tn > { /* ... */ }`
- ❖ The most commonly used type parameter names are:
  - ❖ E - Element (used extensively by the Java Collections Framework)
  - ❖ K - Key
  - ❖ N - Number
  - ❖ T - Type
  - ❖ V - Value
  - ❖ S,U,V etc. - 2nd, 3rd, 4th types
- ❖ For example,

```
Box<Integer> integerBox = new Box<Integer>();
```

OR

```
Box<Integer> integerBox = new Box<>();
```

```
public class Box {  
    private Object object;  
  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

```
/**  
 * Generic version of the Box class.  
 * @param <T> the type of the value being boxed  
 */  
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

# Multiple Type Parameters (Recap)

- ❖ A generic class can have multiple type parameters.
- ❖ For example, the generic `OrderedPair` class, which implements the generic `Pair` interface
- ❖ Usage examples,

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}  
  
public class OrderedPair<K, V> implements Pair<K, V> {  
  
    private K key;  
    private V value;  
  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);  
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");  
....  
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);  
OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");  
....  
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

# Generic Methods (Recap)

Generic methods are methods that **introduce** their **own type** parameters.

```
public class Util {  
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
        return p1.getKey().equals(p2.getKey()) &&  
               p1.getValue().equals(p2.getValue());  
    }  
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown above.

Generally, this can be left out and the compiler will **infer** the **type** that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.compare(p1, p2);
```

## Bounded Type Parameters

- ❖ There may be times when you want to **restrict the types** that can be used as type arguments in a parameterized type.
- ❖ For example, a method that operates on numbers might only want to accept instances of **Number** or its subclasses.

```
public <U extends Number> void inspect(U u){  
    System.out.println("U: " + u.getClass().getName());  
}
```

```
public class NaturalNumber<T extends Integer> {
```

# Multiple Bounds

- ❖ A type parameter can have multiple bounds:

< T extends B<sub>1</sub> & B<sub>2</sub> & B<sub>3</sub> >

- ❖ A type variable with multiple bounds is a subtype of **all** the **types** listed in the bound.
- ❖ Note that B<sub>1</sub>, B<sub>2</sub>, B<sub>3</sub>, etc. in the above refer to **interfaces** or a **class**. There can be at most one class (single inheritance), and the rest (or all) will be **interfaces**.
- ❖ If **one** of the bounds is a **class**, it must be specified **first**.

# Generic Methods and Bounded Type Parameters

```
public static <T> int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
    for (T e : anArray)  
        if (e > elem) // compiler error X - invalid  
            ++count;  
    return count;  
}
```

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

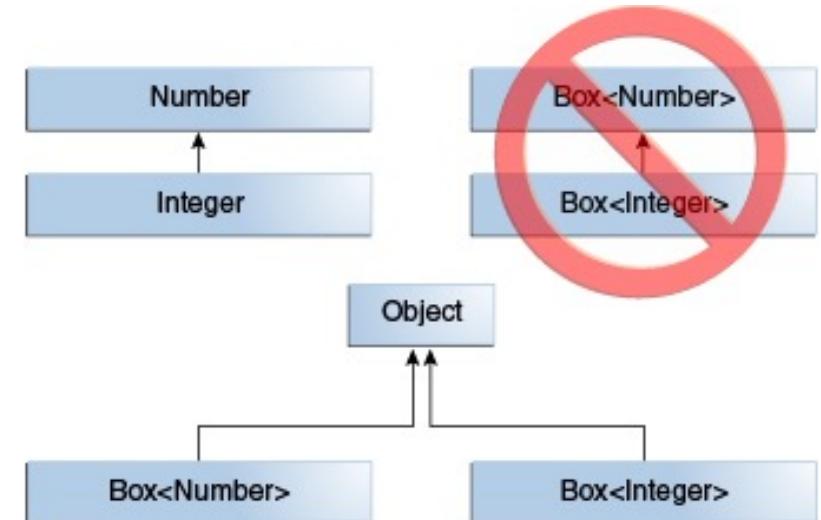
```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem)  
    int count = 0;  
    for (T e : anArray)  
        if (e.compareTo(elem) > 0) Valid  
            ++count;  
    return count;  
}
```

# Generics, Inheritance, and Subtypes

- ❖ Consider the following method:

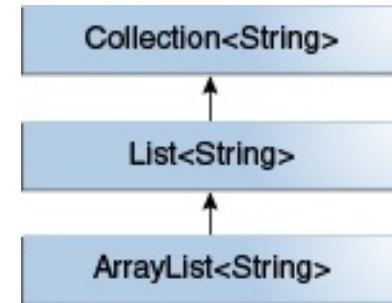
```
public void boxTest( Box<Number> n ) { /* ... */ }
```

- ❖ What type of argument does it accept?
- ❖ Are you allowed to pass in `Box<Integer>` or `Box<Double>` ?
- ❖ The answer is "no", because `Box<Integer>` and `Box<Double>` are **not** subtypes of `Box<Number>`.
- ❖ This is a **common misunderstanding** when it comes to programming with generics.



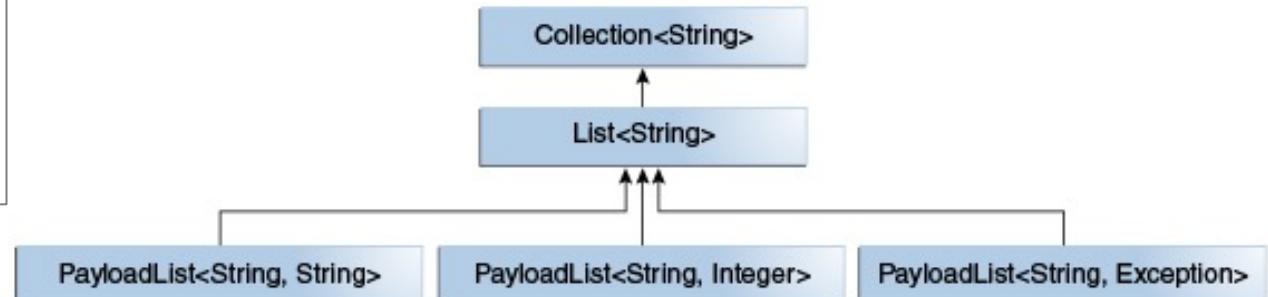
# Generic Classes and Subtyping

- ❖ You **can subtype** a generic class or interface by extending or implementing it.
- ❖ The relationship between the type parameters of one **class** or **interface** and the type parameters of another are determined by the **extends** and **implements** clauses.
- ❖ `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`.
- ❖ So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`.
- ❖ So long as you **do not vary the type argument**, the subtyping relationship is preserved between the types.



```
interface PayloadList<E,P> extends List<E> {  
    void setPayload(int index, P val);  
    ...  
}
```

```
PayloadList<String, String>  
PayloadList<String, Integer>  
PayloadList<String, Exception>
```



# Wildcards: Upper bounded

- ❖ In generic code, the question mark (?), called the **wildcard**, represents an unknown type.
- ❖ The wildcard can be used in a **variety of situations**: as the type of a parameter, field, or local variable; sometimes as a return type.
- ❖ The **upper bounded wildcard**, `< ? extends Foo >`, where Foo is any type, matches Foo and any subtype of Foo .
- ❖ You can specify an upper bound for a wildcard, or you can specify a lower bound, but you **cannot** specify **both**.

```
public static void process(List<? extends Foo> list) {  
    for (Foo elem : list) {  
        // ...  
    }  
}
```

```
public static double sumOfList(List<? extends Number> list) {  
    double s = 0.0;  
    for (Number n : list)  
        s += n.doubleValue();  
    return s;  
}
```

# Wildcards: Unbounded

- ❖ The **unbounded wildcard** type is specified using the wildcard character (?), for example, `List< ? >`. This is called a list of unknown type.

```
public static void printList(List<Object> list) {  
    for (Object elem : list)  
        System.out.println(elem + " ");  
    System.out.println();  
}
```

It prints only a list of Object instances;  
it **cannot** print `List<Integer>`, `List<String>`,  
`List<Double>`, and so on

```
public static void printList(List<?> list) {  
    for (Object elem: list)  
        System.out.print(elem + " ");  
    System.out.println();  
}
```

To write a generic `printList`  
method, use `List<?>`

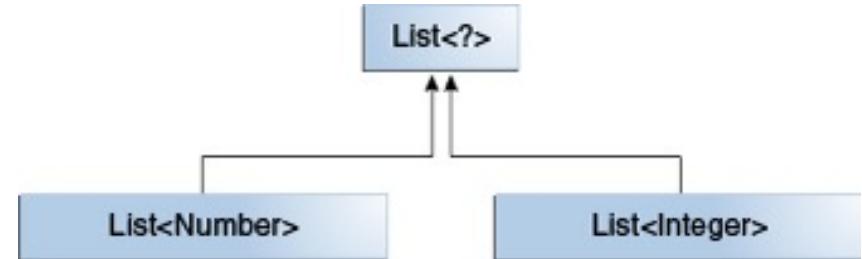
# Wildcards: Lower Bounded

- ❖ An **upper bounded wildcard** restricts the unknown type to be a specific type or a subtype of that type and is represented using the **extends** keyword.
- ❖ A **lower bounded wildcard** is expressed using the wildcard character ('?'), following by the **super** keyword, followed by its lower bound: `< ? super A >`.
- ❖ To write the method that works on lists of Integer and the super types of **Integer**, such as **Integer**, **Number**, and **Object**, you would specify `List<? Super Integer>`.
- ❖ The term `List<Integer>` is more **restrictive** than `List<? super Integer>`.

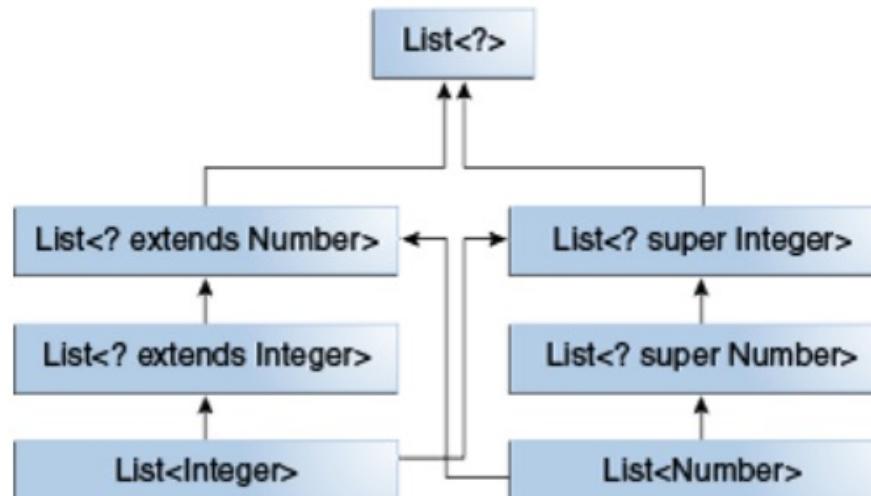
```
public static void addNumbers(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```

# Wildcards and Subtyping

- ❖ Although Integer is a subtype of Number, List<Integer> is **not** a **subtype** of List<Number> and, these two types are **not related**.



- ❖ The **common parent** of List<Number> and List<Integer> is **List<?>**.



A hierarchy of several generic List class declarations.

**End**

# COMP2511

## Creational Pattern: Singleton Pattern

Prepared by

Dr. Ashesh Mahidadia

# Creational Patterns

Creational patterns provide various **object creation** mechanisms, which increase flexibility and reuse of existing code.

## ❖ Factory Method

- provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

## ❖ Abstract Factory

- let users produce families of related objects without specifying their concrete classes.

## ❖ Singleton

- Let users ensure that a class has only one instance, while providing a global access point to this instance.

## ❖ Builder

- let users construct complex objects step by step. The pattern allows users to produce different types and representations of an object using the same construction code.

# Singleton Pattern

# Singleton Pattern

**Intent:** Singleton is a creational design pattern that lets you ensure that a class has **only one instance**, while providing a global access point to this instance.

**Problem:** A client wants to,

- ❖ ensure that a class has just a **single instance**, and
- ❖ provide a **global** access point to that instance

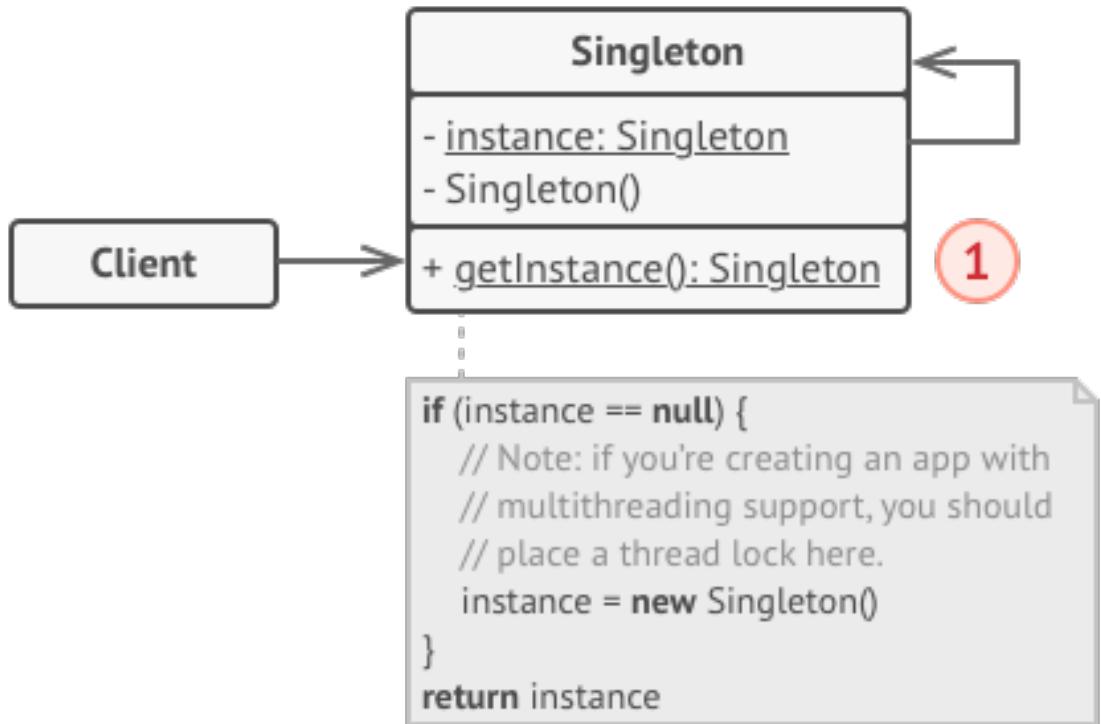
**Solution:**

All implementations of the Singleton have these two steps in common:

- ❖ Make the **default constructor private**, to prevent other objects from using the new operator with the Singleton class.
- ❖ Create a **static creation method** that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the **cached object**.
- ❖ If your code has access to the Singleton class, then it's able to **call the Singleton's static method**.
- ❖ Whenever Singleton's static method is called, the **same object** is always returned.

# Singleton: Structure

- ❖ The **Singleton** class declares the **static** method *getInstance* (1) that returns the same instance of its own class.
- ❖ The Singleton's constructor should be hidden from the client code.
- ❖ Calling the *getInstance* (1) method should be the only way of getting the Singleton object.



# Singleton: How to Implement

- ❖ Add a **private static field** to the class for storing the singleton instance.
- ❖ Declare a **public static creation method** for getting the singleton instance.
- ❖ Implement “lazy initialization” inside the static method.
  - It should create a **new object** on its first call and put it into the static field.
  - The method should always return that instance on all **subsequent calls**.
- ❖ Make the **constructor of the class private**.
  - The static method of the class will still be able to call the constructor, but not the other objects.
- ❖ In a client, call singleton’s static creation method to access the object.

Example in Java (**MUST read**):

<https://refactoring.guru/design-patterns/singleton/java/example>

# Singleton Pattern

For more information, read:

<https://refactoring.guru/design-patterns/singleton>

**End**

# COMP2511

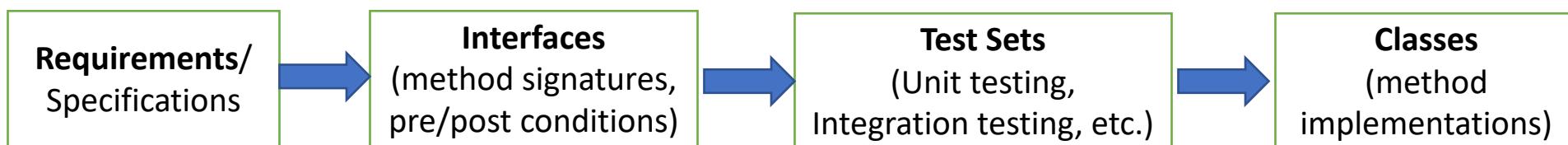
# Test Design

“Testing shows the presence, not the absence of bugs.”

*—Edsger W. Dijkstra*

# Software Testing: Test-Driven Development (TDD)

- ❖ Every iteration in the software development process **must be preceded** with a plan to properly verify (test) that the developed software meets the requirements (i.e., post conditions).
- ❖ A software developer must **not** create a software artifact and later think of how to test it!
- ❖ **Testing** is an **essential integral** part of developing a software solution. It must **not** be considered as an afterthought !
- ❖ **Incremental** development must be tested against test suites, during every iteration. Every code **modification** and/or **refactoring** must be followed by a proper testing, using the predefined test suites.
- ❖ Testing must be **setup**, based on the requirement specifications, **before** you start **implementing** your solution.



# Software Testing: Input Space Coverage

- ❖ Testing must **not** be conducted **haphazardly** by trial-and-error.
- ❖ Testing must be conducted **systematically**, with a well thought out **testing plan**.
- ❖ The aim should be to consider a possible *input space* and cover it as much as possible.
- ❖ Often this is achieved by **dividing** the *input space* into “*equivalence groups*” and selecting a representative input from each equivalence group. Here, the assumption is: from the same equivalence group, a program is expected to behave similarly on each input.
- ❖ For example, for the method `boolean isSorted (list);` possible input cases to consider,
  - Input list: 34, 12, 15, 21, 5, 21. [random all positive]
  - Input list: -13, -12, -77, -60, -55. [random all negative]
  - Input list: 10, -11, 17, 31, 50, 42. [random mix positive/negative]
  - Input list: 22, 22, 22, 22, 22, 22. [all same]
  - Input list: 3, 7, 34, 41, 53, 99. [increasing order]
  - Input list: 99, 45, 0, -10, -34, -89. [decreasing order]
  - Etc.
  - Etc.

# Software Testing: Input Space Coverage

- ❖ Consider **borderline** cases, often called **boundary testing**.
- ❖ For example, for the method `String getGrade( marks );` possible input cases to consider,
  - 0
  - 50
  - 65
  - 75
  - 85
  - 100
- ❖ For **multiple input values**, consider possible **input combinations**, **prioritise** them and consider as many as possible, given the available time and resources. Again, divide possible combinations into *homogenous* subsets and select representative combinations.

# Software Testing: Code Coverage

- ❖ Code coverage is a useful metric that can help you assess the quality of your test suite.
- ❖ Code coverage measures the degree to which a software is verified by a test suite, by determining the number of lines of code that is successfully validated by the test suite.
- ❖ The common metrics in most coverage reports include:
  - Function coverage: how many of the functions defined have been called.
  - Statement coverage: how many of the statements in the program have been executed.
  - Branches coverage: how many of the branches of the control structures (if statements for instance) have been executed.
  - Condition coverage: how many of the boolean sub-expressions have been tested for a true and a false value.
  - Line coverage: how many of lines of source code have been tested.
- ❖ For more, see <https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>

# Randomness in Software Testing and Simulation

*Randomness* is also useful!

## ❖ Software **Testing**:

- random data is often seen as *unbiased data*
  - gives average performance (e.g. in sorting algorithms)
- *stress test* components by bombarding them with random data

## ❖ Software **Simulation**:

- generating random behaviours/movements.  
For example, may want players/enemies to move in a random pattern.  
Possible approach: **randomly generate a number between 0 to 3**,
  - **0** means **front** movement, **1** means **left** movement,  
**2** means **back** movement, **3** means **right** movement.
- the layout of a dungeon may be **randomly generated**
- may want to introduce **unpredictability**

# Random Numbers

- ❖ How can a computer pick a number at random?
  - it **cannot** !
- ❖ Software can only produce *pseudo random numbers*.
  - a **pseudo random number** is one that is **predictable**!
    - (although it may appear unpredictable)
- ❖ Implementation may deviate from expected theoretical behaviour.

# Generating Random Numbers in Java

Using `random` class,

- ❖ Need to import the class `java.util.Random`
- ❖ **Option-1:** Creates a new random number generator.
  - ❖ `Random rand = new Random();`
- ❖ **Option-2:** Creates a new random number generator using a single long `seed`.
  - *Important:* Every time you run a program with the `same seed`, you get exactly the `same sequence` of 'random' numbers.  
`Random rand = new Random(long seed);`
- ❖ To vary the output, we can give the random seeder a starting point that varies with time. For example, a starting point (`seed`) is the current time.
- ❖ Go to the API for more information at  
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html>

# Basic Test Template

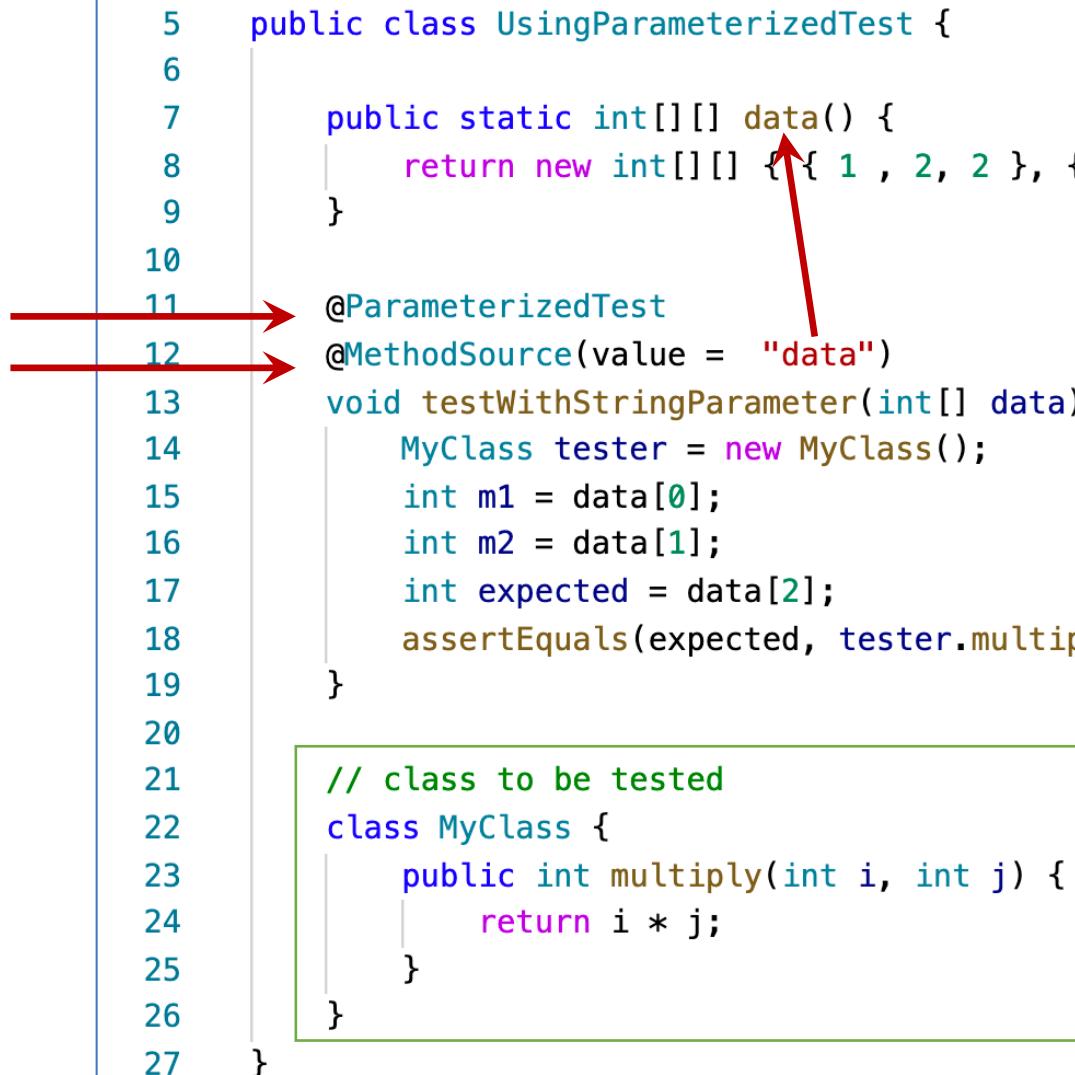
- 1) Set up Precondition (i.e. @BeforeEach, etc.)
  - 2) Act (call the method)
  - 3) Verify Post condition (i.e. @AfterEach, Asserts, etc.)
- Normally, each test should run independently, order of execution should not be important.
  - However, if required, you can order execution of the tests using @TestMethodOrder

# Avoid Repetition in Test Suites: Parameterized Tests

- JUnit offers *Parameterized Tests*.
- **Parameterized test** executes the same test over and over again using different input values and tests output against the corresponding expected results.
- A **data source** can be used to retrieve data for input values and expected results.
- The **@Before** annotation can be used if you want to execute some statement such as preconditions before each test case.
- The **@After** annotation can be used if you want to execute some statements after each Test Case for e.g. resetting variables, deleting temporary files, variables, etc.
- For more information, see [JUnit 5 tutorial](#) .

# Parameterized Test: An Example

```
5  public class UsingParameterizedTest {  
6  
7      public static int[][] data() {  
8          return new int[][] {{ 1 , 2 , 2 }, { 5 , 3 , 15 }, { 121 , 4 , 484 } };  
9      }  
10  
11     @ParameterizedTest  
12     @MethodSource(value = "data")  
13     void testWithStringParameter(int[] data) {  
14         MyClass tester = new MyClass();  
15         int m1 = data[0];  
16         int m2 = data[1];  
17         int expected = data[2];  
18         assertEquals(expected, tester.multiply(m1, m2));  
19     }  
20  
21     // class to be tested  
22     class MyClass {  
23         public int multiply(int i, int j) {  
24             return i * j;  
25         }  
26     }  
27 }
```



For more information, see [JUnit 5 tutorial](#) at  
<https://www.vogella.com/tutorials/JUnit/article.html>.

# Avoid Repetition in Test Suites: Dynamic Tests

- JUnit also offer **Dynamic Tests**.
- For more information, see [JUnit 5 tutorial](#) .

```
12  class DynamicTestCreationTest {  
13  
14  @TestFactory  
15  Stream<DynamicTest> testDifferentMultiplyOperations() {  
16      MyClass tester = new MyClass();  
17      int[][] data = new int[][] { { 1, 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };  
18      return Arrays.stream(data).map(entry -> {  
19          int m1 = entry[0];  
20          int m2 = entry[1];  
21          int expected = entry[2];  
22          return dynamicTest(m1 + " * " + m2 + " = " + expected, () -> {  
23              assertEquals(expected, tester.multiply(m1, m2));  
24          });  
25      });  
26  }  
27  
28  // class to be tested  
29  class MyClass {  
30      public int multiply(int i, int j) {  
31          return i * j;  
32      }  
33  }  
34 }
```

For more information, see [JUnit 5 tutorial](#) at  
<https://www.vogella.com/tutorials/JUnit/article.html>.

# Types of tests

Spectrum of Tests: **Unit Tests, Integration Tests, Systems Tests, Usability tests / acceptance tests**

## Unit test

- Test a single piece of functionality.
- Ideally, should test in isolation – scientific method, keep all other variables controlled.
- Minimise dependencies on other functionality.
- Therefore, difficult to write black-box unit tests
  - We can make our tests unit-like by reducing the number of dependencies as much as possible.
  - Mock testing and mock objects can be used, though this requires knowledge of what functionality the method relies on.
  - Not easy to say whether changes took place testing a single method without calling another method!

# Types of tests

## Integration test

- Test a web of dependencies (coupling) that catches any bugs “lurking in the cracks” that the unit tests didn’t pick up.
- Every failing integration test should be able to be written as a failing unit test.
- Tests interactions (couplings) between software components.

## System test

- Perform a black-box test on the entire system as a whole.
- This can be done at different levels of abstraction, for COMP2511 we system test at the controller level.

# Types of tests

## Usability tests / acceptance tests

- “Test that it works on the frontend”
- Does the functionality achieve the intended goal?
- Is it usable?

## Property-based tests

- Test individual properties of code rather than testing the output directly

# Creating a Testing Plan

Need to properly devise a way for testing that ensure:

- high coverage.
- there is a mix of different types of tests.

## Beware!

- Writing too many tests is **bad** – if you have unit, integration and system tests all for the same thing then the test suite becomes **tightly coupled** and hard to maintain.
- Need to strike a **balance** – one way is to test everything with unit tests, but only test the main flows / use cases of the program with integration/system tests – for the project this will be a team decision documented in your testing plan.

# Principles of writing test code

- Everything applies to test code as it does to normal code, DRY, KISS.
- You can use design patterns in test code.
- However, in writing test code there are some other things to consider:
  - You want to make your test code **as simple as possible**, otherwise you end up having something that is more complex (and as a result bug-prone) than the software you are testing in the first place!
  - Conditionals, loops, any control flow should be kept to a minimum to reduce test complexity.
- Factory pattern is often very useful in test design since you can write a factory to produce dummy objects for testing.

# Software Testing: Summary

- Always follow Test Driven Development.
- Software Testing is hard!
- Not possible to completely test a nontrivial software system, given the limited available resources.
- We assume that a selected *representative* test cases capture system behavior of test cases not considered.

# End

# COMP2511

## Creational Pattern: Builder Pattern

Prepared by

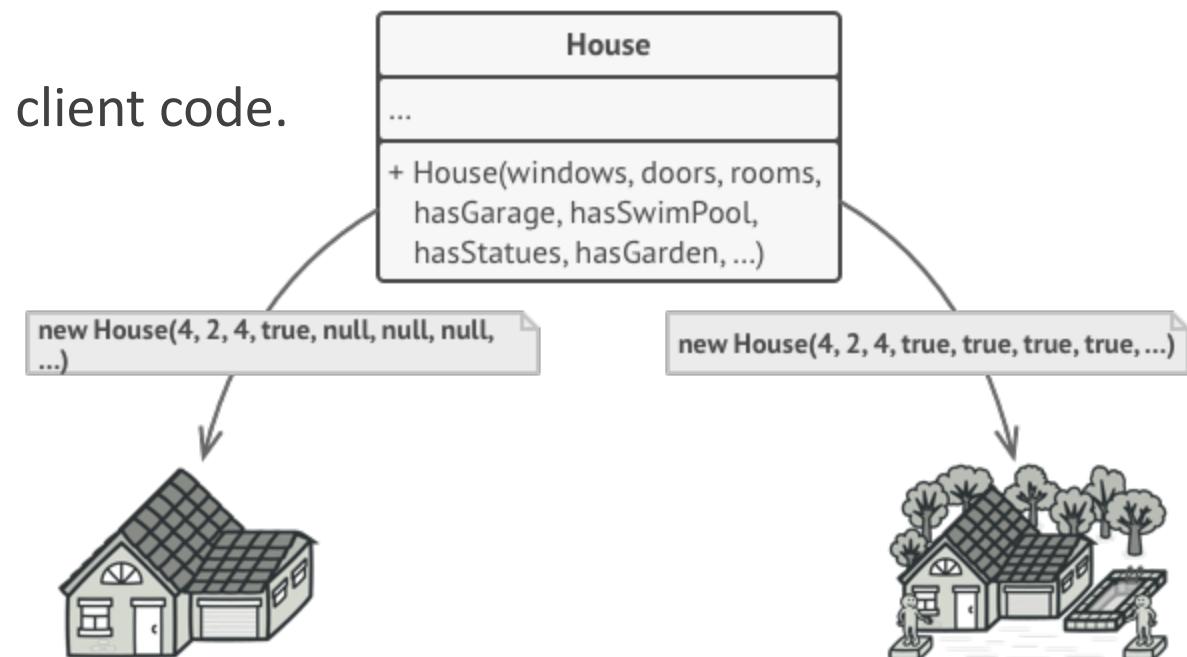
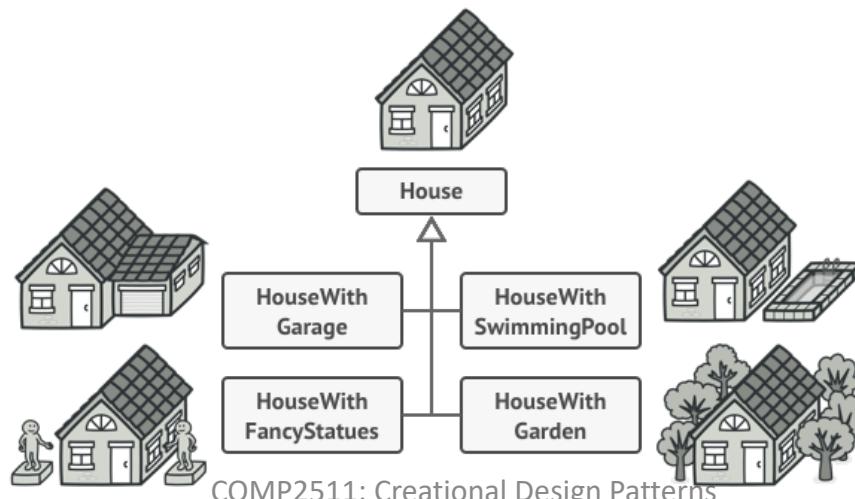
Dr. Ashesh Mahidadia

# Builder Pattern

**Intent:** Builder is a creational design pattern that lets you **construct complex objects** step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

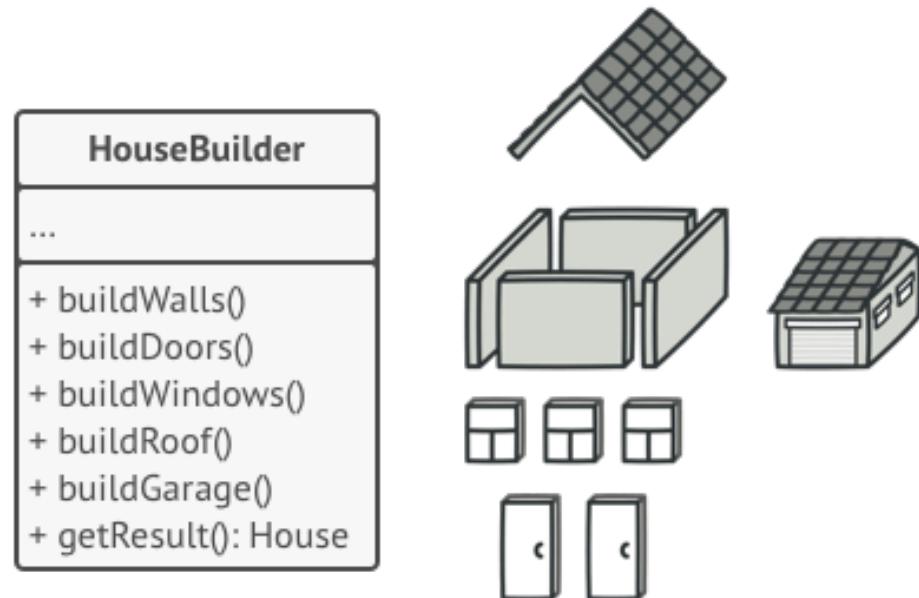
## Problem:

- ❖ Imagine **a complex object** that requires laborious, **step-by-step initialization/construction** of many fields and nested objects.
- ❖ Such initialization/construction code is usually buried inside a monstrous **constructor** with **lots of parameters**.
- ❖ Or even worse: **scattered** all over the client code.



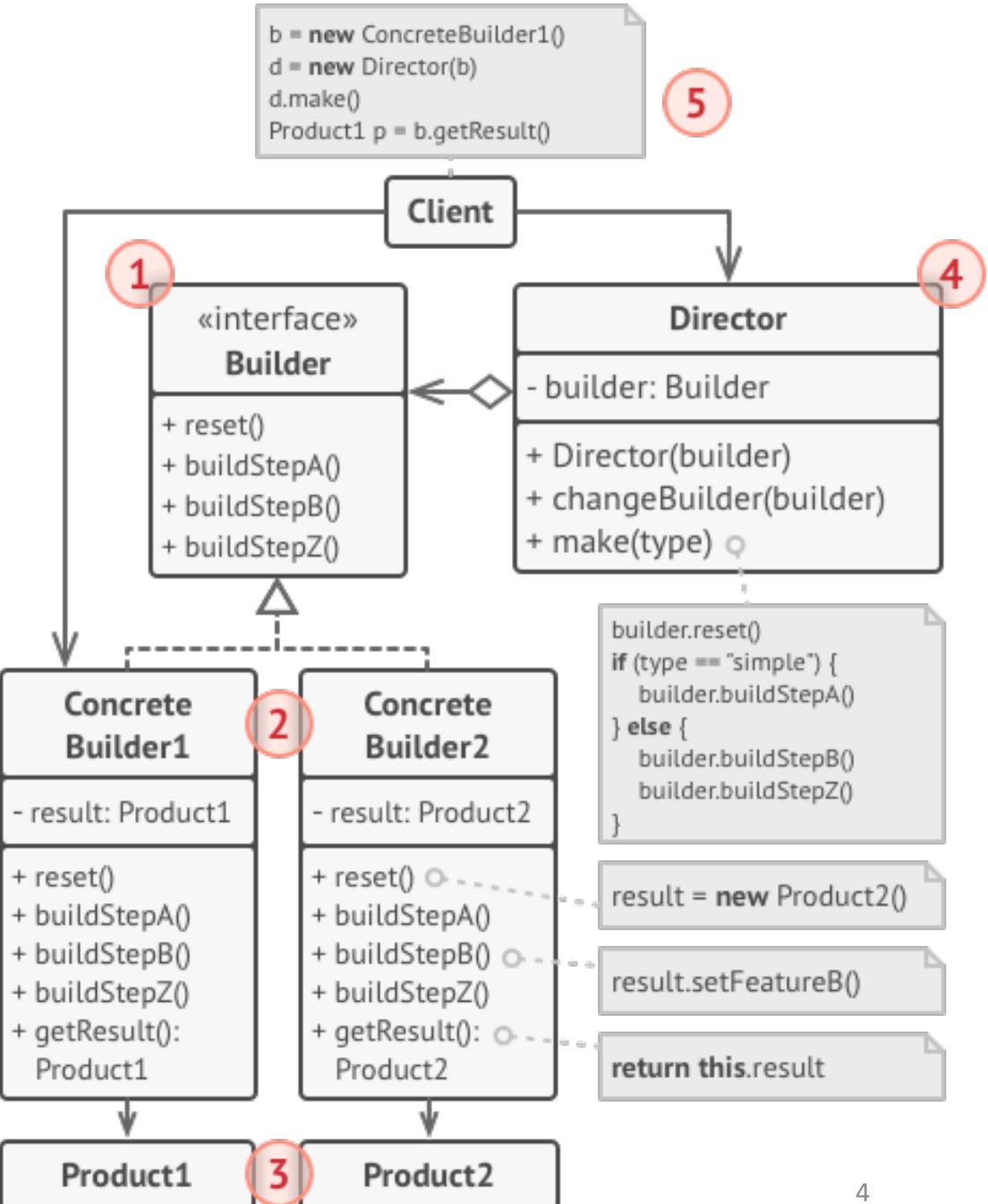
# Builder Pattern

- ❖ The Builder pattern suggests that you **extract** the **object construction code** out of its own class and move it to separate objects called **builders**.
- ❖ The Builder pattern lets you **construct** complex objects **step by step**.
- ❖ The Builder **doesn't allow** other objects to access the product **while it's being built**.
- ❖ **Director**: The **director class** defines the **order** in which to execute the building steps, while the **builder provides the implementation** for those steps.



# Builder Pattern: Structure

- ❖ The **Builder** interface declares product construction steps that are common to all types of builders.
- ❖ **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
- ❖ **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
- ❖ The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
- ❖ The **Client** must associate one of the builder objects with the director.



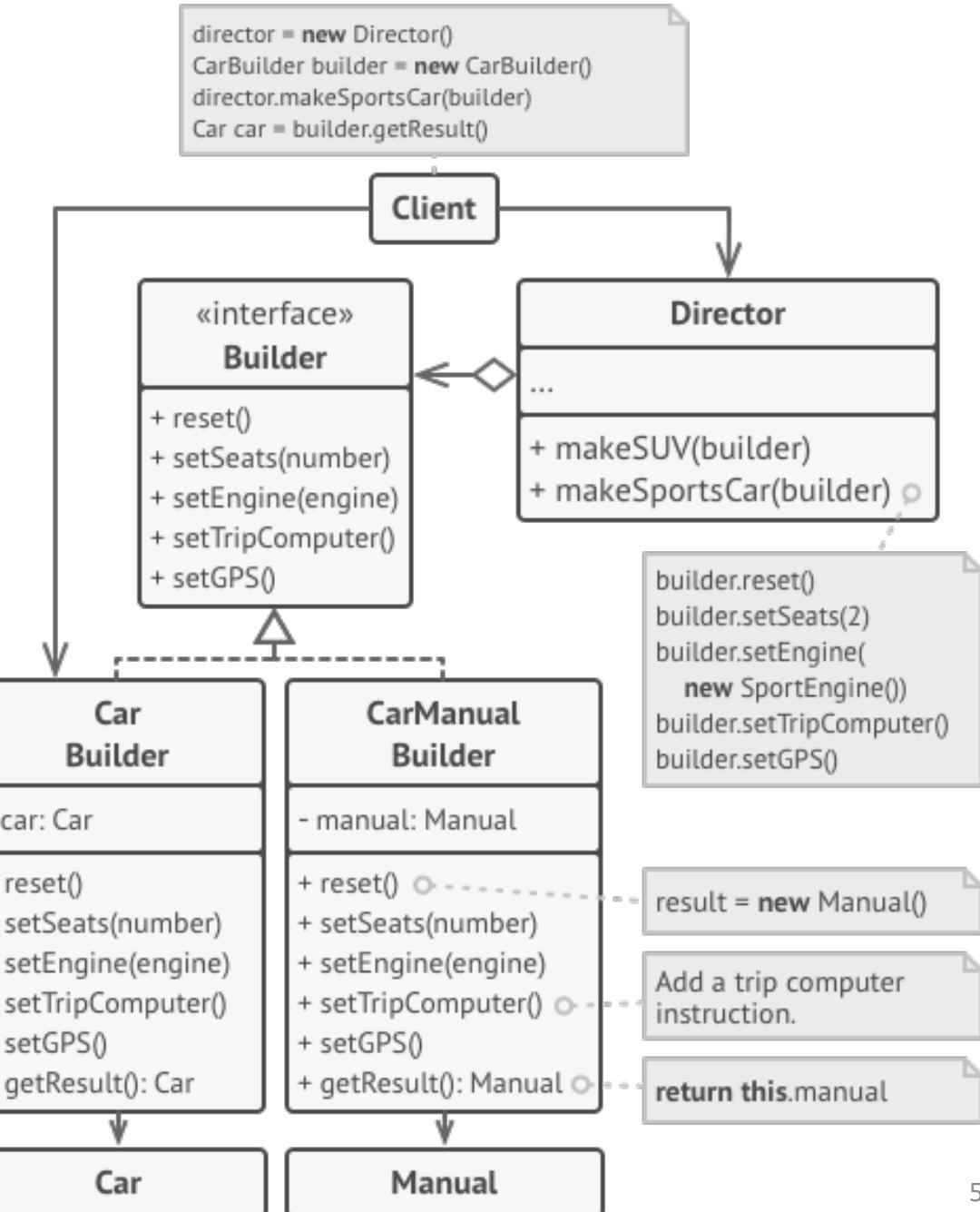
# Builder Pattern: Example

This example illustrates how you can **reuse the same object construction code** when,

- ❖ building different types of **cars**, and
- ❖ creating the corresponding **manuals** for them.

**Example in Java (MUST read):**

<https://refactoring.guru/design-patterns/builder/java/example>



# Relations with Other Patterns

- ❖ Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and **evolve** toward **Abstract Factory**, or **Builder** (more flexible, but more complicated).
- ❖ **Builder** focuses on constructing complex objects step by step.
- ❖ **Abstract Factory** specializes in creating families of related objects.
- ❖ **Abstract Factory** returns the product immediately, whereas **Builder** lets you run some additional construction steps before fetching the product.

# Builder Pattern

For more information, read:

<https://refactoring.guru/design-patterns/builder>

**End**

# COMP2511



## 8.2 - Event-Driven & Asynchronous Design, Part 2

# Asynchronous Paradigm

## What is asynchronous programming?

- Asynchronous programming, broadly speaking is any form of programming where processes are not necessarily sequentially executed
- Different forms of asynchronous implementation
  - Event loops (seen commonly in JavaScript, which is single-threaded)
  - Parallelism (multi-threading)
- We will focus mainly on parallelism in this course
- Parallelism vs Concurrency:
  - **Parallelism:** The simultaneous execution of computations
  - **Concurrency:** The art of managing parallelism

# Asynchronous Paradigm

## Why program asynchronously?

- Performance - sequential execution is too slow
- We don't want programs to become bottlenecked on **blocking** operations (e.g. reading a file, opening a web socket)
- Requirements to have real-time updates (e.g. instant messaging)
- Allows for better software design (we'll see soon)

# Asynchronous Paradigm

In Week 7 we discussed concurrency, synchronisation and the Singleton Pattern - ways of managing the issues arise when programming asynchronously.

Today, we'll discuss applications of asynchronous programming in software design.

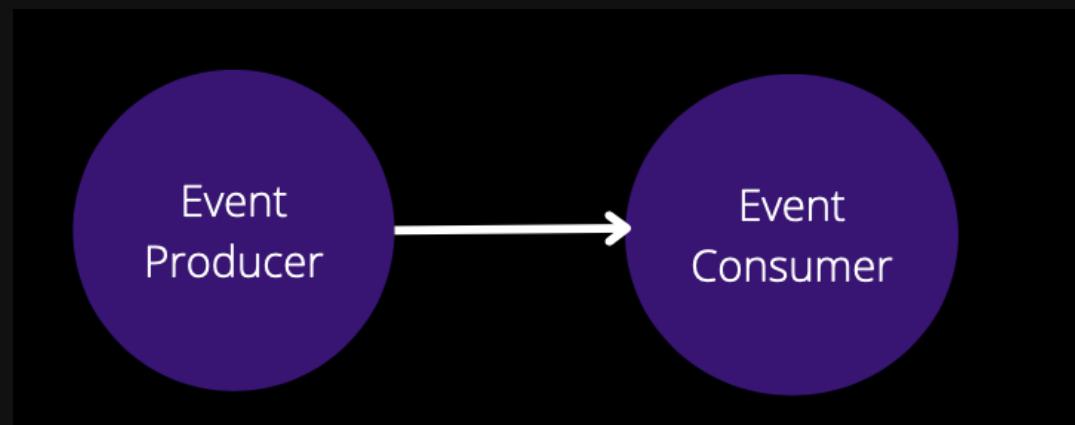
The primary application is Event-Driven Programming and the Observer Pattern.

# Promises

<https://www.digitalocean.com/community/tutorials/understanding-the-event-loop callbacks-promises-and-async-await-in-javascript>

# Event-Driven Revisited

- The Observer Pattern - way of *propagating changes in state* between software components
- **Producers** (AKA subjects) produce events
- **Consumers** (AKA subscribers, observers, listeners) consume events
- Producer/Consumer relationships can be 1:1, 1:M, M:1, M:M
- But first, some background



# 1. The Event-Driven Paradigm

- Event: A change in state
- In the event-driven paradigm, events are first class citizens

```
1 const element = document.getElementById("myBtn");
2 element.addEventListener("click", myFunction);
3
4 function myFunction(event) {
5     console.log(event.target.value)
6 }
```

## 2. State as a Series of Events

- We can think of the current state of a system to be *the result of playing out all the events that have occurred*
- Where have we seen this before?
  - In the project?
  - In your programming experience?

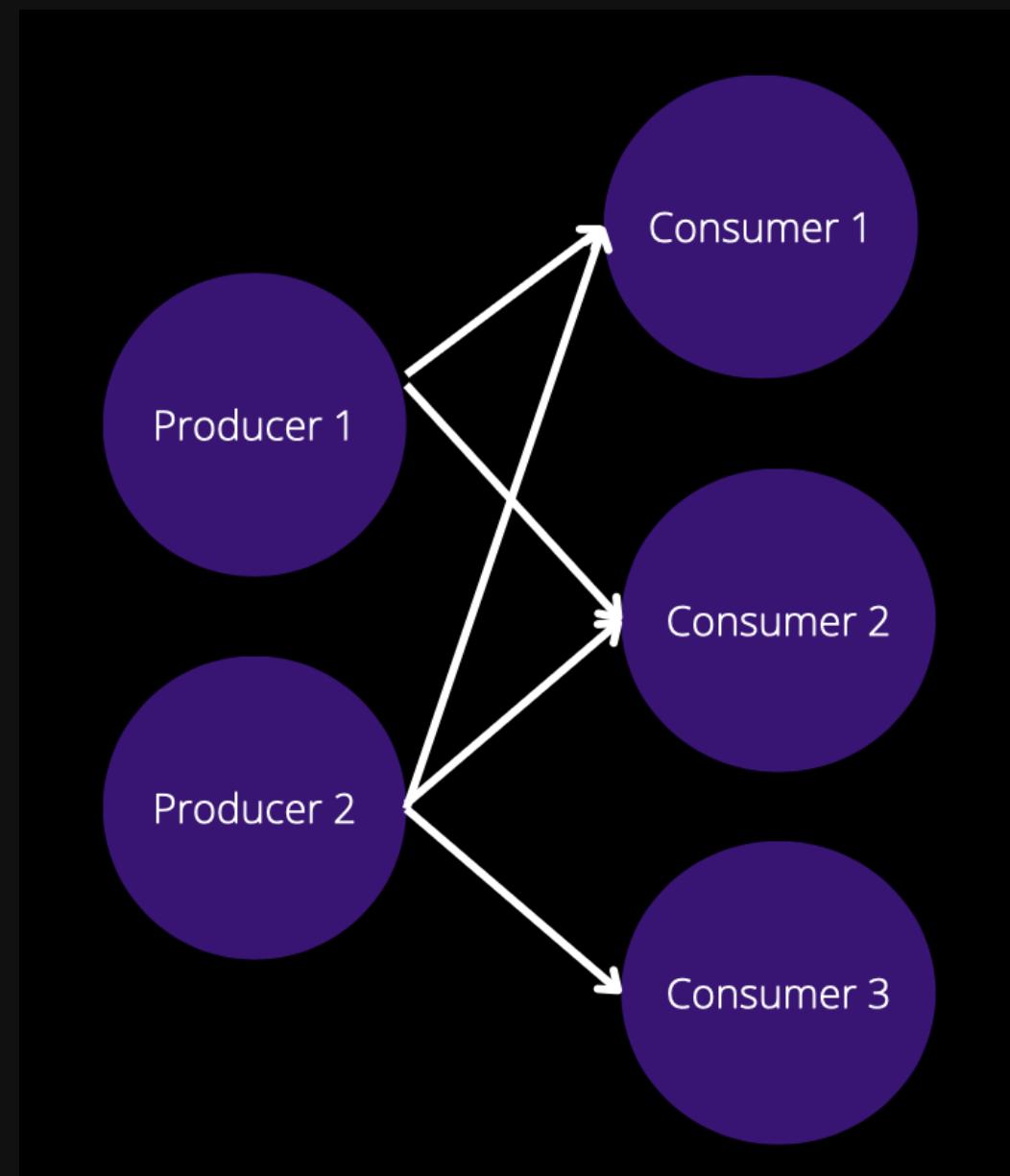
## 3. Streams Revisited

```
1 List<Integer> ints =  
2     strings2.stream()  
3         .map(Integer::parseInt)  
4         .collect(Collectors.toList());
```

- Minimise coupling by letting data "flow down the stream"

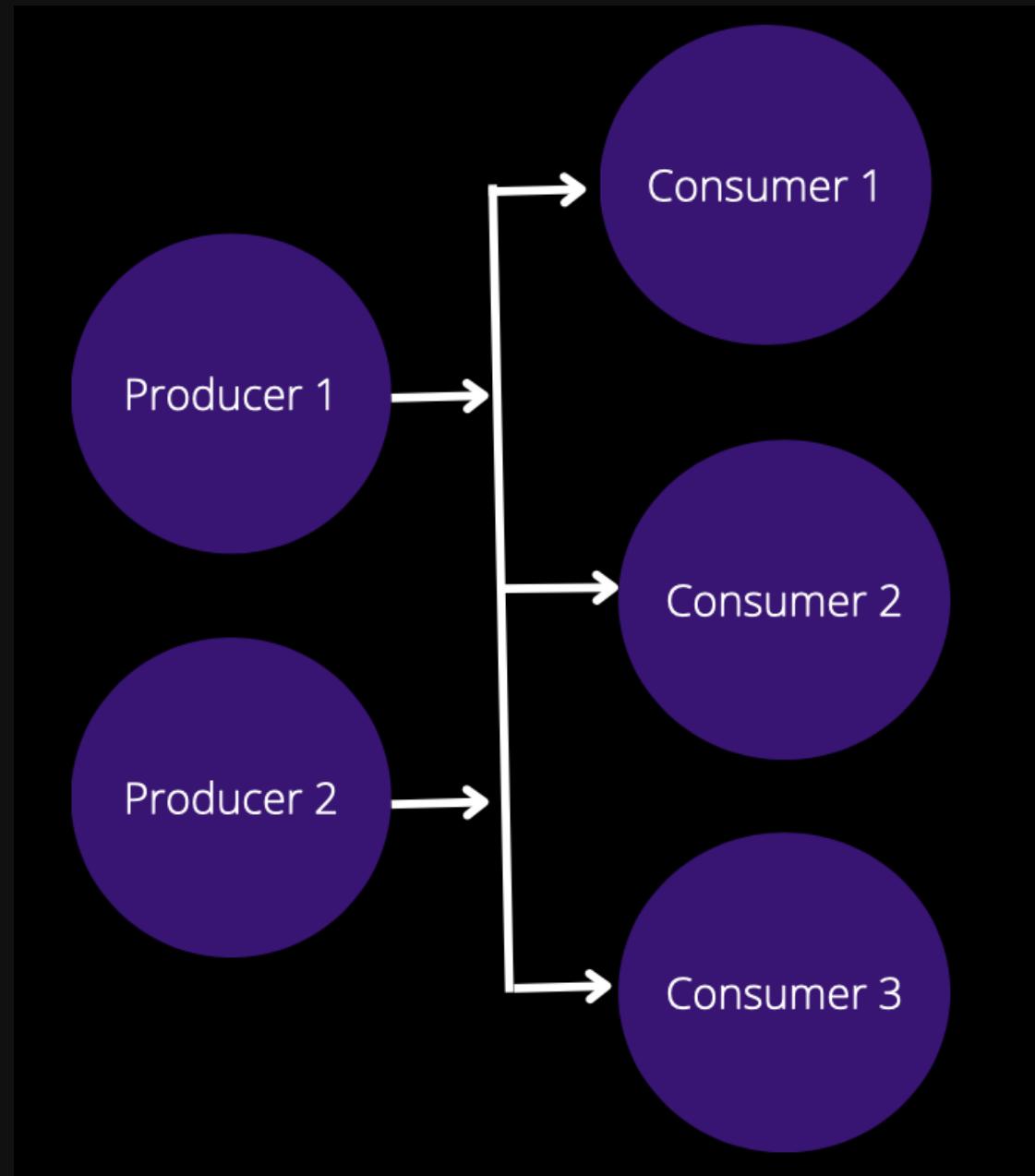
# Problems with the Synchronous Observer Pattern

- Problem 1 - What if while we are processing the first event, the second event comes along?
- Problem 2 - What if we start to increase the number of relationships:
  - More producers
  - More consumers
  - More types of events
- Leads to high coupling
- The postman problem



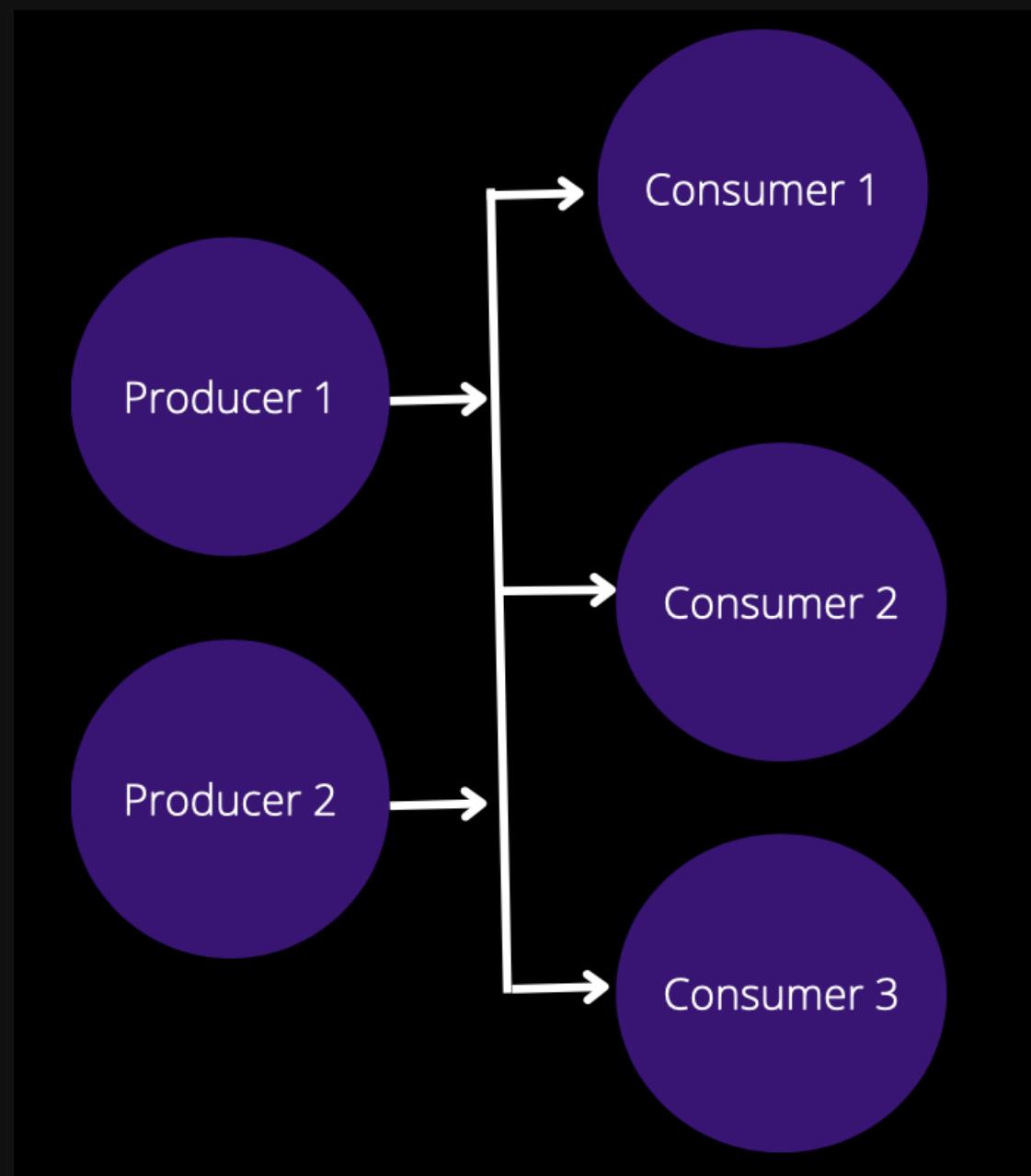
# Asynchronous Observer Pattern

- Instead of producers and consumers being directly coupled, they instead communicate over a **shared channel**
- Producers deposit events into the channel
- Events "flow through the channel"
- Consumers read from the shared channel and process the events
- Producers and consumers are all run in parallel



# Asynchronous Observer Pattern

- What if a consumer's not interested?
  - Can choose to consume certain types of events, and ignore others
- Good design
  - Principle of Least Knowledge
  - No coupling between producers and consumers
  - Can add/remove producers or consumers at will

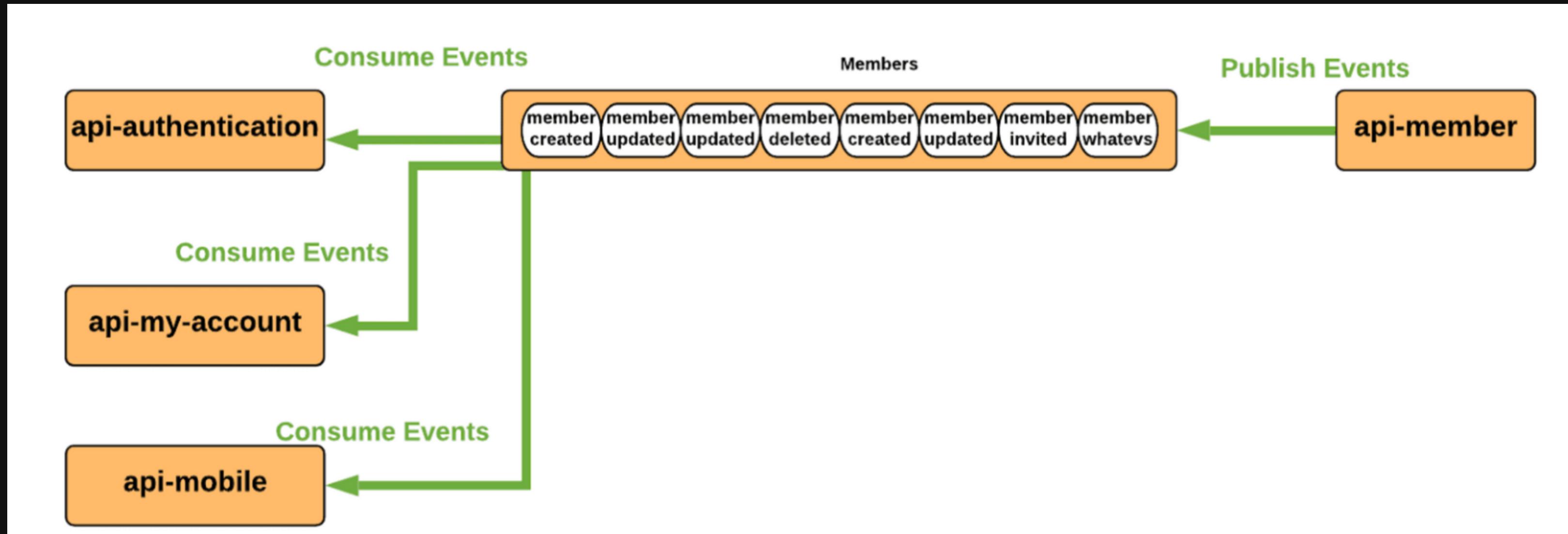


# Implementation: Go Channels

- Go provides very good lightweight support for this design pattern with **goroutines** (you can think of them as lightweight threads) and **channels** (essentially a buffer)
  - <https://go.dev/play/p/MZo57Ei1NKW>
- Need to consider blocking of goroutines on channels and problems of:
  - Deadlock
  - Starvation

# Implementation: Apache Kafka

- Communication between repositories or APIs in a service ecosystem



# Event Pipelines

- We can "play" events through a pipeline using this model
- What if we wanted to **play back** events?
  - Process interrupted
  - Data malformed
  - Fix up and replay the events
- Need to design consumers for **idempotency** - what if they consume the same event twice?
- Need to determine if consumers **rely on a particular ordering** of events to be processed in



# Final Thoughts

- It all just seems like glue?
- Trends of industry and modern software architecture
  - Microservices and abstraction
  - The need for event driven systems
- The role of design patterns in real-world software

# COMP2511



## 8.2 - Iterator Pattern

# In this lecture

## Why?

- Understand the concepts of iterators and iterables
- Understand the motivation for the Iterator Pattern
- Discuss implementation of the Iterator Pattern in different languages

# How does a for loop actually work?

```
1 List<String> shoppingList = new ArrayList<String>(  
2     Arrays.asList(new String[ ] {  
3         "apple", "banana", "pineapple", "orange"  
4     } ));  
5  
6 for (String item : shoppingList) {  
7     System.out.println(item);  
8 }
```

# Under the hood

```
1 Iterator<String> iter = shoppingList.iterator();
2 while (iter.hasNext()) {
3     String item = iter.next();
4     System.out.println(item);
5 }
```

# Iterators

- An **iterator** is an object that enables a programmer to traverse a container
- Allows us to access the contents of a data structure while abstracting away its underlying representation
- In Java, for loops are an abstraction of iterators
- Iterators can tell us:
  - Do we have any elements left?
  - What is the next element?

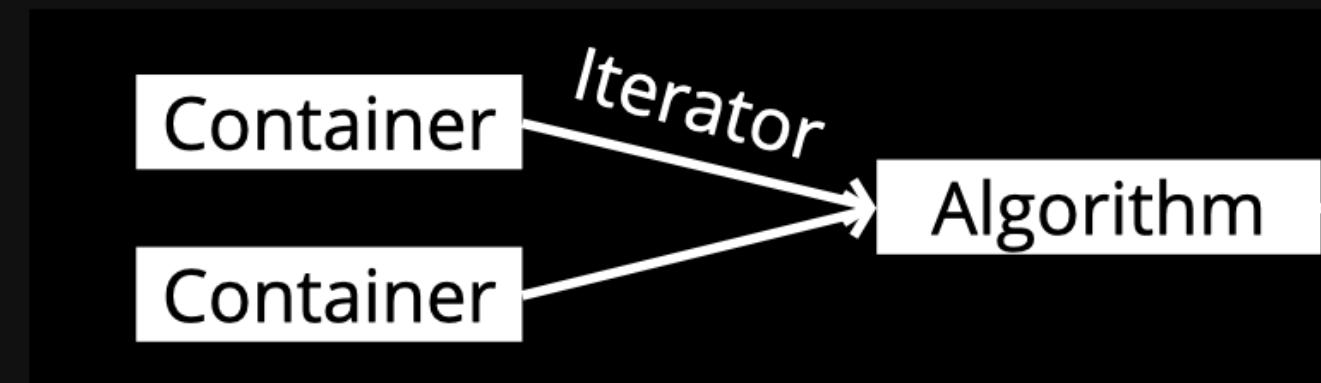
# Custom Iterators

# Traversing a Data Structure

- Aggregate entities (Containers)
  - Stacks, Queues, Lists, Trees, Graphs, Cycles
- How do we traverse an aggregate entity without exposing its underlying representation?
- Maintain abstraction and encapsulation
- Initial solution - a method in the interface
  - What if we want multiple ways to traverse the container?

# Abstracting the Traversal

- Separate Containers, Iterators and Algorithms
- Allows for many possible ways of traversal
- Avoid bloating interfaces with different traversal methods
- Client (Algorithm) requests an iterator from the container
- Container needs to provide a method for creating an iterator, to show that it is *iterable*



# Iterators vs Iterables

- An **iterable** is an object that can be iterated over
- All iterators are iterable, but not all iterables are iterators
- For loops only need to be given something *iterable*

```
public interface Iterator<E> {  
    /**  
     * Returns {@code true} if the iteration has more elements.  
     * (In other words, returns {@code true} if {@link #next} would  
     * return an element rather than throwing an exception.)  
     *  
     * @return {@code true} if the iteration has more elements  
     */  
    boolean hasNext();  
  
    /**  
     * Returns the next element in the iteration.  
     *  
     * @return the next element in the iteration  
     * @throws NoSuchElementException if the iteration has no more elements  
     */  
    E next();
```

```
public interface Iterable<T> {  
    /**  
     * Returns an iterator over elements of type {@code T}.  
     *  
     * @return an Iterator.  
     */  
    Iterator<T> iterator();
```

# Example: Custom Iterator

```
Hashtable<String, MenuItem> menuItems =  
    new Hashtable<String, MenuItem>();  
  
public Iterator<MenuItem> createIterator() {  
    return menuItems.values().iterator();  
}
```

Using or forwarding an **iterator** method from  
a collection (i.e. Hashtable, ArrayList, etc.)

Implement **Iterator** interface, and provide the  
required methods (and more if required).

```
public class DinerMenuIterator implements Iterator<MenuItem> {  
    MenuItem[] list;  
    int position = 0;  
  
    public DinerMenuIterator(MenuItem[] list) {  
        this.list = list;  
    }  
  
    public MenuItem next() {  
        MenuItem menuItem = list[position];  
        position = position + 1;  
        return menuItem;  
    }  
  
    public boolean hasNext() {  
        if (position >= list.length || list[position] == null) {  
            return false;  
        } else {  
            return true;  
        }  
    }  
  
    public void remove() {  
        if (position <= 0) {  
            throw new IllegalStateException  
                ("You can't remove an item until you've done at least one next()");  
        }  
        if (list[position-1] != null) {  
            for (int i = position-1; i < (list.length-1); i++) {  
                list[i] = list[i+1];  
            }  
            list[list.length-1] = null;  
        }  
    }  
}
```

Read the example code  
discussed/developed in the  
lectures, and also provided  
for this week

# Iterator Invalidation

- What happens when we modify something we're iterating over?

```
1 numbers = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
2
3 for number in numbers:
4     if number == 3 or number == 4:
5         numbers.remove(number)
6
7 print(numbers)
```

# Design by Contract

- In many languages, part of the **postconditions** of iterators is that modifying the container in certain ways causes the iterator to become **invalidated** (the behaviour of the iterator is undefined)
  - Python
  - C++

# Iterator Invalidation: Java

- What happens when we modify something we're iterating over?

```
1 List<Integer> numbers = new ArrayList<Integer>(
2     Arrays.asList(new Integer[] {1, 2, 3, 4, 5, 6, 7, 8, 9})
3 );
4
5 for (Integer number : numbers) {
6     if (number.equals(3) || number.equals(4)) {
7         numbers.remove(number);
8     }
9 }
10
11 System.out.println(numbers);
```

# Iterator Invalidation: Java

- What happens when we modify something we're iterating over?

```
1 Exception in thread "main" java.util.ConcurrentModificationException
2         at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
3         at java.base/java.util.ArrayList$Itr.next(ArrayList.java:997)
4         at dungeonmania.DungeonManiaController.main(IterExample.java:120)
```

# Generators

- A functional way of writing iterators
- Defined via generator functions instead of classes
- Example generator

```
1 def shopping_list():  
2     yield 'apple'  
3     yield 'orange'  
4     yield 'banana'  
5     yield 'pineapple'  
6  
7 for item in shopping_list():  
8     print(item)
```

# Iterator Categories (C++)

- Output (Write-only)
- Input (Read-only)
- Forward (most iterators, standard Java iterators)
- Bidirectional (forward and backwards)
- Random Access (iterators which function as arrays)

# COMP2511

## Template Pattern

Prepared by

Dr. Ashesh Mahidadia

# Template Pattern: Motivation and Intent

- "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*." [GoF]*

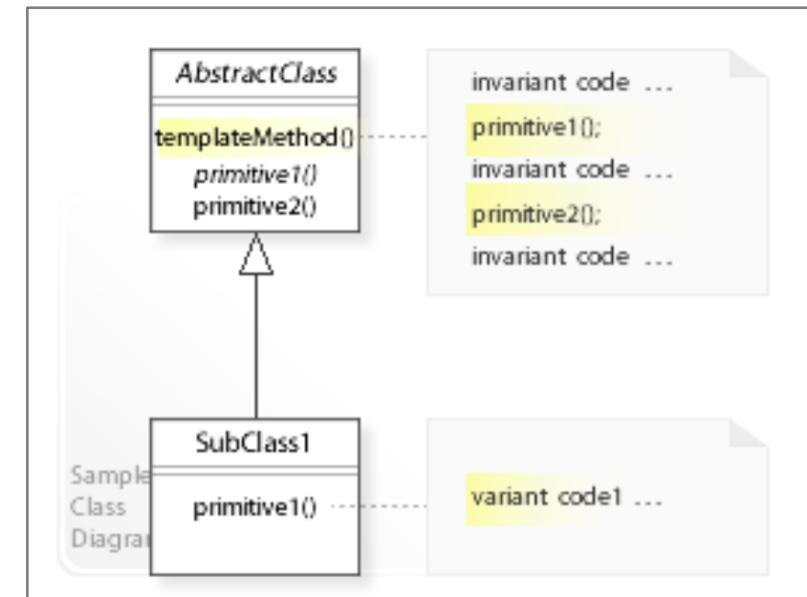
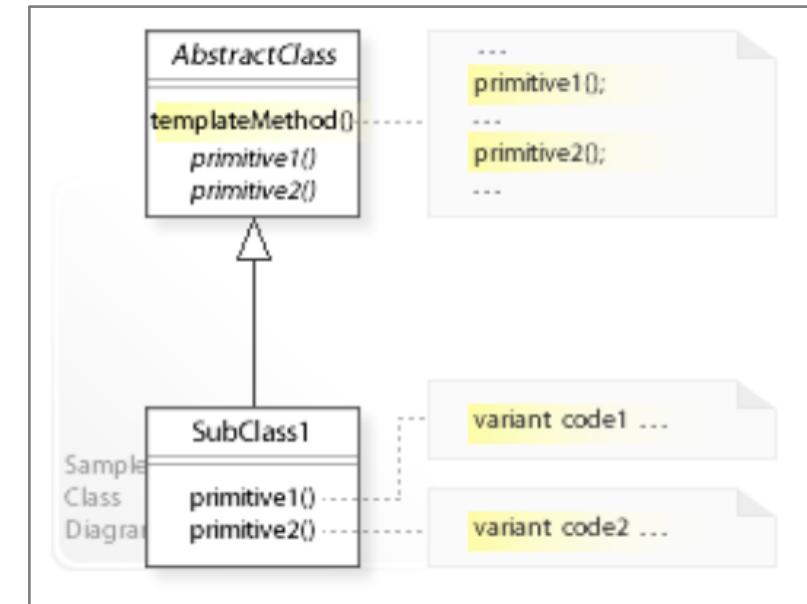
- A *template Method* defines the skeleton (structure) of a behavior (by implementing the invariant parts).
- A *template Method* calls *primitive operations*, that could be implemented by sub classes OR has default implementations in an abstract super class.
- Subclasses can redefine only certain parts of a behavior *without changing* the other parts or the *structure* of the behavior.

# Template Pattern: Motivation and Intent

- ❖ Subclasses do not control the behavior of a parent class,  
a parent class calls the operations of a subclass and not the other way around.
- ❖ **Inversion of control:**
  - ❖ when using a **library** (reusable classes), we call the code we want to reuse.
  - ❖ when using a **framework** (like Template Pattern), we write subclasses and implement the variant code the framework calls.
- ❖ Template pattern implement the **common (invariant) parts of a behavior** once "and leave it up to subclasses to implement the behavior that can vary." [GoF, p326]
- ❖ Invariant behavior is in one class (localized)

# Template Pattern: Structure

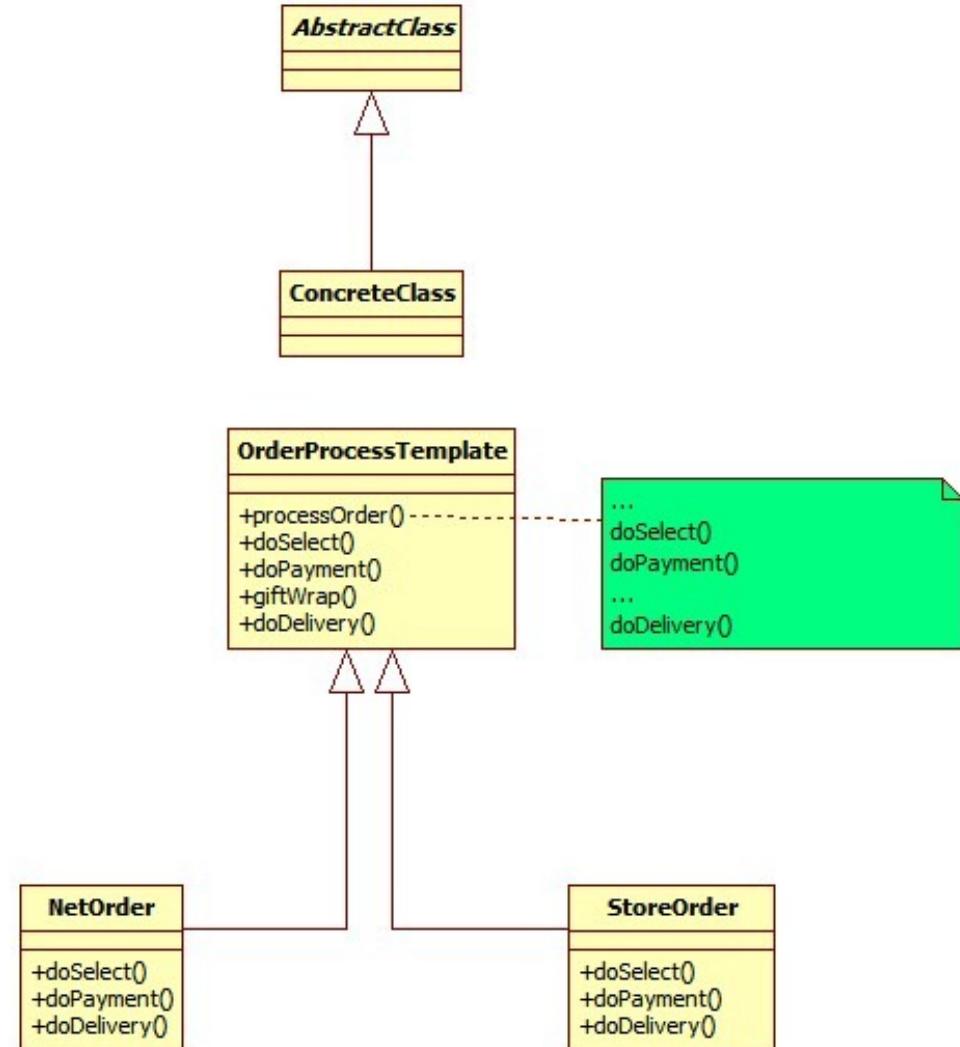
- Abstract class defines a *templateMethod()* to implement an invariant structure (behaviour)
- *templateMethod()* calls methods defined in the abstract class (abstract or concrete) - like primitive1, primitive2, etc.
- **Default** behaviour can be implemented in the abstract class by offering concrete methods
- Importantly, **sub classes** can implement primitive methods for **variant behaviour**



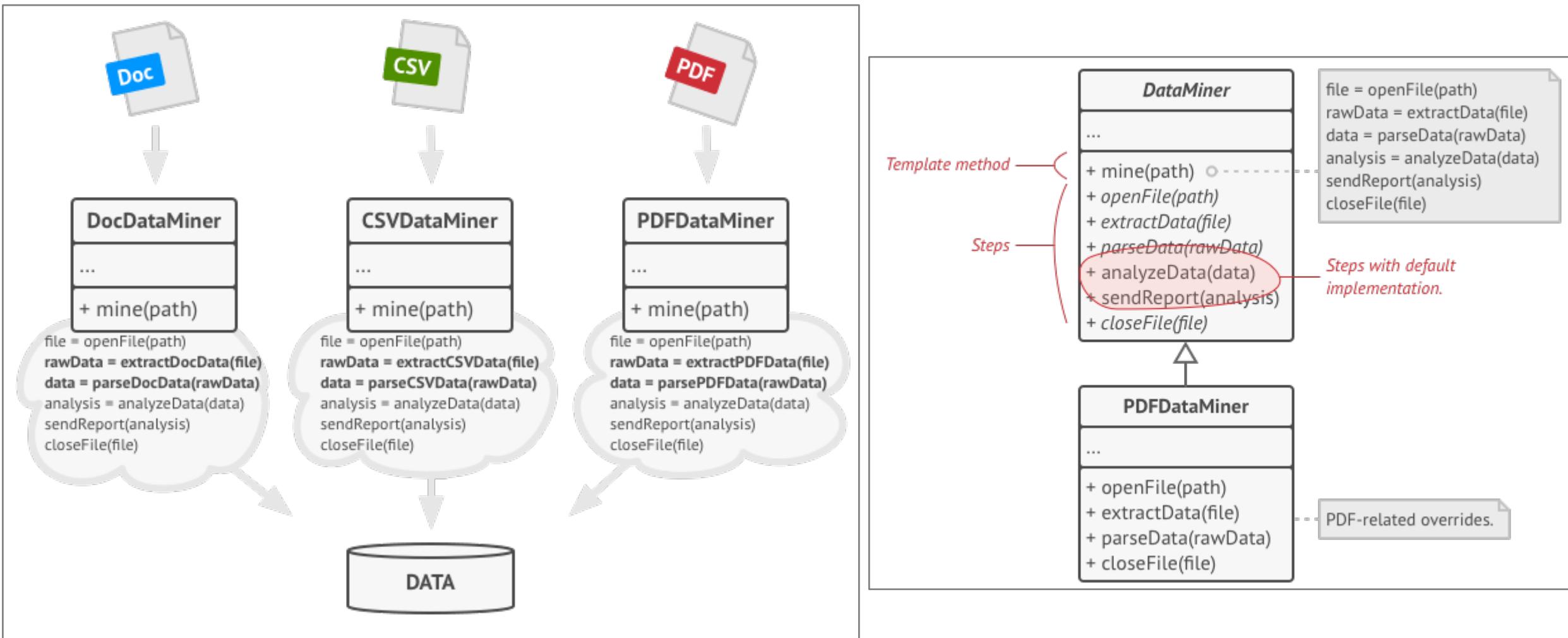
# Template Pattern: Structure

- ❖ "To reuse an abstract class effectively, **subclass writers must understand** which operations are designed for overriding." [GoF, p328]
- ❖ **Primitive operations** : operations that have default implementations or must be implemented by sub classes.
- ❖ **Final operations**: concrete operations that cannot be overridden by sub classes.
- ❖ **Hook operations**: concrete operations that do nothing by default and can be redefined by subclasses if necessary. This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore the hook. (see the example)

# Template Pattern: Example



# Template Pattern: Example



- From <https://refactoring.guru/design-patterns/template-method>

# Template Pattern: Example

```
public abstract class MyReportTemplate {
```

```
    public void genReport() {
```

Template method

```
        InputStream f1 = openFile();
```

Step 1

```
        SortedMap<String, ArrayList<String>> data = parseFile( f1 );
```

Step 2

```
        generateReport ( data );
```

Step 3

```
        if( isRequestedSummary()) {  
            generateSummary(data);  
        }
```

Step 4

```
}
```

Read the example code discussed/developed in the lectures, and also provided for this week

```
    public void generateSummary(SortedMap<String, ArrayList<String>> data) {  
        System.out.println("generating Summary (default from MyReportTemplat ...");
```

```
    }  
  
    public boolean isRequestedSummary() {  
        return false;  
    }
```

```
    public void generateReport(SortedMap<String, ArrayList<String>> data) {  
        System.out.println("generating report (default from MyReportTemplat ...");
```

```
    }  
  
    protected abstract SortedMap<String, ArrayList<String>> parseFile(InputStream f1);  
    public abstract String getFilename();
```

Abstract methods

```
    public InputStream openFile() {  
  
        String filename = getFilename();  
        InputStream f1 = null;
```

Default method

```
        try {  
            f1 = new FileInputStream(filename);  
        } catch (FileNotFoundException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
        return f1;
```

# Template Pattern: Example

Read the example code discussed/developed in the lectures, and also provided for this week

```
public class CSVReport extends MyReportTemplate{
    private String fname = "";
    private boolean reqSummary = false;

    public CSVReport() {
        super();
        fname = "src/example/data.csv";
        reqSummary = false;
    }
    public CSVReport(String filename, boolean requestSummary) {
        this.fname = filename;
        this.reqSummary = requestSummary;
    }

    @Override
    protected SortedMap<String, ArrayList<String>> parseFile(InputStream f1) {
        // CSV parsing code here
        System.out.println("parsing CSV data file: " + getFilename());
        TreeMap<String, ArrayList<String>> data =
            new TreeMap<String, ArrayList<String>>();
        // populate data object in this method ..
        return data;
    }

    @Override
    public String getFilename() {
        // ask user for a file name.. or get from a constructor
        return fname;
    }

    @Override
    public boolean isRequestedSummary() {

        return this.reqSummary;
    }
}
```

Step 2

→

Part of Step 1

→

Hook

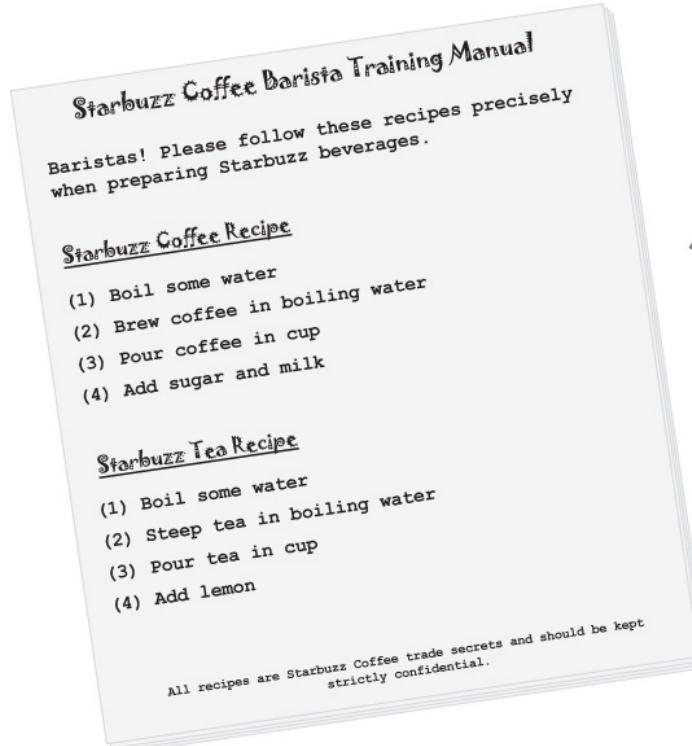
→

# Template Pattern: Example

Read the example code discussed/developed in the lectures, and also provided for this week

```
public class Test1 {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        System.out.println("\n*** Generate CSV report . . . . .");  
  
        CSVReport rep1 = new CSVReport("src/example/data.csv", true);  
        rep1.genReport();  
  
        System.out.println("\n*** Generate XML report . . . . .");  
  
        XMLReport rep2 = new XMLReport("src/example/data.xml");  
        rep2.genReport();  
  
        System.out.println("\n*** Generate CSV with Summarys report . . . . .");  
  
        CSVReportWithSummary rep3 = new CSVReportWithSummary();  
        rep3.genReport();  
  
    }  
  
}
```

# Template Pattern: Example



```
Here's our Coffee class for making coffee.
```

```
public class Coffee {
```

```
    void prepareRecipe() {
```

```
        boilWater();
```

```
        brewCoffeeGrinds();
```

```
        pourInCup();
```

```
        addSugarAndMilk();
```

```
}
```

```
public void boilWater() {
```

```
    System.out.println("Boiling water");
```

```
}
```

```
public void brewCoffeeGrinds() {
```

```
    System.out.println("Dripping Coffee through filter");
```

```
}
```

```
public void pourInCup() {
```

```
    System.out.println("Pouring into cup");
```

```
}
```

```
public void addSugarAndMilk() {
```

```
    System.out.println("Adding Sugar and Milk");
```

```
}
```

Here's our Coffee class for making coffee.

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup, and add sugar and milk.

# Template Pattern: Example



```
public class Tea {  
  
    void prepareRecipe() {  
        boilWater();  
        steepTeaBag();  
        pourInCup();  
        addLemon();  
    }  
  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    public void steepTeaBag() {  
        System.out.println("Steeping the tea");  
    }  
  
    public void addLemon() {  
        System.out.println("Adding Lemon");  
    }  
  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

This looks very similar to the one we just implemented in Coffee; the second and fourth steps are different, but it's basically the same recipe.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

These two methods are specialized to Tea.

# Template Pattern: Example

## Starbuzz Coffee Recipe

- (1) Boil some water
- (2) Brew coffee in boiling water
- (3) Pour coffee in cup
- (4) Add sugar and milk

## Starbuzz Tea Recipe

- (1) Boil some water
- (2) Steep tea in boiling water
- (3) Pour tea in cup
- (4) Add lemon

```
public abstract class CaffeineBeverage {  
  
    void final prepareRecipe() {  
  
        boilWater();  
  
        brew();  
  
        pourInCup();  
  
        addCondiments();  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        // implementation  
    }  
  
    void pourInCup() {  
        // implementation  
    }  
}
```

prepareRecipe() is our template method. Why?

Because:

- (1) It is a method, after all.
- (2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

# Template Pattern: Example (hook)

```
public abstract class CaffeineBeverageWithHook {  
  
    final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        if (customerWantsCondiments()) {  
            addCondiments();  
        }  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
  
    boolean customerWantsCondiments() {  
        return true;  
    }  
}
```

We've added a little conditional statement that bases its success on a concrete method, `customerWantsCondiments()`. If the customer WANTS condiments, only then do we call `addCondiments()`.

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a hook because the subclass can override this method, but doesn't have to.

# Template Pattern: Example (hook)

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {  
  
    public void brew() {  
        System.out.println("Dripping Coffee through filter");  
    }  
  
    public void addCondiments() {  
        System.out.println("Adding Sugar and Milk");  
    }  
  
    public boolean customerWantsCondiments() {  
  
        String answer = getUserInput();  
  
        if (answer.toLowerCase().startsWith("y")) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    private String getUserInput() {  
        String answer = null;  
  
        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");  
  
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));  
        try {  
            answer = in.readLine();  
        } catch (IOException ioe) {  
            System.err.println("IO error trying to read your answer");  
        }  
        if (answer == null) {  
            return "no";  
        }  
        return answer;  
    }  
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false, depending on the input.

This code asks the user if he'd like milk and sugar and gets his input from the command line.

# Template Vs Strategy Patterns

- Template Method works at the class level, so it's **static**.
- Strategy works on the object level, letting you switch behaviors at **runtime**.
- Template Method is based on **inheritance**: it lets you alter parts of an algorithm by extending those parts in subclasses.
- Strategy is based on composition: you can alter parts of the **object's** behavior by supplying it with different strategies that correspond to that behavior at runtime.

# COMP2511

## Adapter Pattern

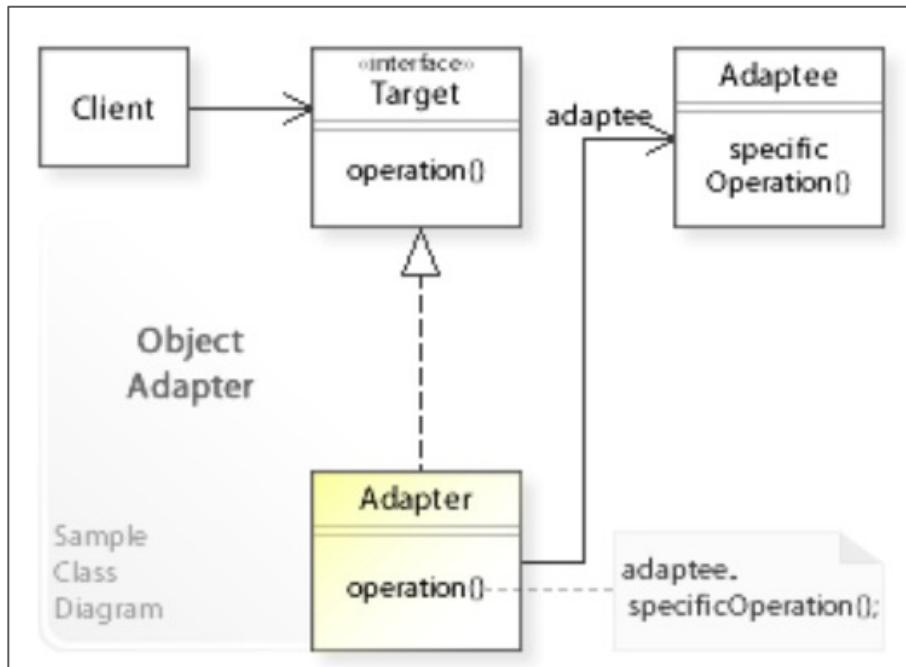
Prepared by

Dr. Ashesh Mahidadia

# Adapter Pattern : Intent

- ❖ "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces." [GoF]
- ❖ The adapter pattern allows the **interface** of an **existing class** to be used as **another interface**, **suitable** for a **client** class.
- ❖ The adapter pattern is often used to make **existing classes** (APIs) work **with** a **client** class **without modifying** their source code.
- ❖ The **adapter class** maps / joins functionality of two different types / interfaces.
- ❖ The adapter pattern offers a wrapper around an existing useful class, such that a client class can use functionality of the existing class.
- ❖ The adapter pattern do **not offer additional** functionality.

# Adapter Pattern: Structure



- ❖ The adapter contains an instance of the class it wraps.
- ❖ In this situation, the adapter makes calls to the instance of the wrapped object.

# Adapter: Example

```
interface LightningPhone {
    void recharge();
    void useLightning();
}

interface MicroUsbPhone {
    void recharge();
    void useMicroUsb();
}
```

```
class Iphone implements LightningPhone {
    private boolean connector;

    @Override
    public void useLightning() {
        connector = true;
        System.out.println("Lightning connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect Lightning first");
        }
    }
}
```

```
class Android implements MicroUsbPhone {
    private boolean connector;

    @Override
    public void useMicroUsb() {
        connector = true;
        System.out.println("MicroUsb connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect MicroUsb first");
        }
    }
}
```

# Adapter: Example

```
public class AdapterDemo {  
    static void rechargeMicroUsbPhone(MicroUsbPhone phone) {  
        phone.useMicroUsb();  
        phone.recharge();  
    }  
  
    static void rechargeLightningPhone(LightningPhone phone) {  
        phone.useLightning();  
        phone.recharge();  
    }  
  
    public static void main(String[] args) {  
        Android android = new Android();  
        Iphone iPhone = new Iphone();  
  
        System.out.println("Recharging android with MicroUsb");  
        rechargeMicroUsbPhone(android);  
  
        System.out.println("Recharging iPhone with Lightning");  
        rechargeLightningPhone(iPhone);  
  
        System.out.println("Recharging iPhone with MicroUsb");  
        rechargeMicroUsbPhone(new LightningToMicroUsbAdapter(iPhone));  
    }  
}
```

```
class LightningToMicroUsbAdapter implements MicroUsbPhone {  
    private final LightningPhone lightningPhone;  
  
    public LightningToMicroUsbAdapter(LightningPhone lightningPhone) {  
        this.lightningPhone = lightningPhone;  
    }  
  
    @Override  
    public void useMicroUsb() {  
        System.out.println("MicroUsb connected");  
        lightningPhone.useLightning();  
    }  
  
    @Override  
    public void recharge() {  
        lightningPhone.recharge();  
    }  
}
```

## Output

```
Recharging android with MicroUsb  
MicroUsb connected  
Recharge started  
Recharge finished  
Recharging iPhone with Lightning  
Lightning connected  
Recharge started  
Recharge finished  
Recharging iPhone with MicroUsb  
MicroUsb connected  
Lightning connected  
Recharge started  
Recharge finished
```

# Design Patterns: Discuss Differences

## ❖ Creational Patterns

- ❖ Abstract Factory
- ❖ Factory Method
- ❖ Singleton

## ❖ Structural Patterns

- ❖ Adapter discussed
- ❖ Composite discussed
- ❖ Decorator discussed

## ❖ Behavioral Patterns

- ❖ Iterator discussed
- ❖ Observer discussed
- ❖ State discussed
- ❖ Strategy discussed
- ❖ Template
- ❖ Visitor

We plan to discuss the rest of the design patterns above in the following weeks; and many more other topics.

**End**

# COMP2511

## Synchronous vs Asynchronous Software Design

Prepared by

Dr. Ashesh Mahidadia

# What is Synchronous programming?

- In *synchronous* programming, operations are carried out **in order**.
- The execution of an operation is **dependent upon** the completion of the **preceding** operation.
- Tasks (functions) A, B, and C are executed in a **sequence**, often using one thread.



# What is Asynchronous programming?

- In *asynchronous programming*, operations are carried out **independently**.
- The execution of an operation is **not dependent upon** the completion of the preceding operation.
- Tasks (functions) A, B, and C are executed **independently**, can use multiple threads/resources.



# Example: Synchronous vs Asynchronous programming

Synchronous

```
function getRecord(key) {  
    establish database connection  
    retrieve the record for key  
    return record;  
}
```

```
function display(rec){  
    display rec on the web page  
}
```

```
rec = getRecord('Rita');  
display(rec)  
  
rec = getRecord('John');  
display(rec)
```

Asynchronous

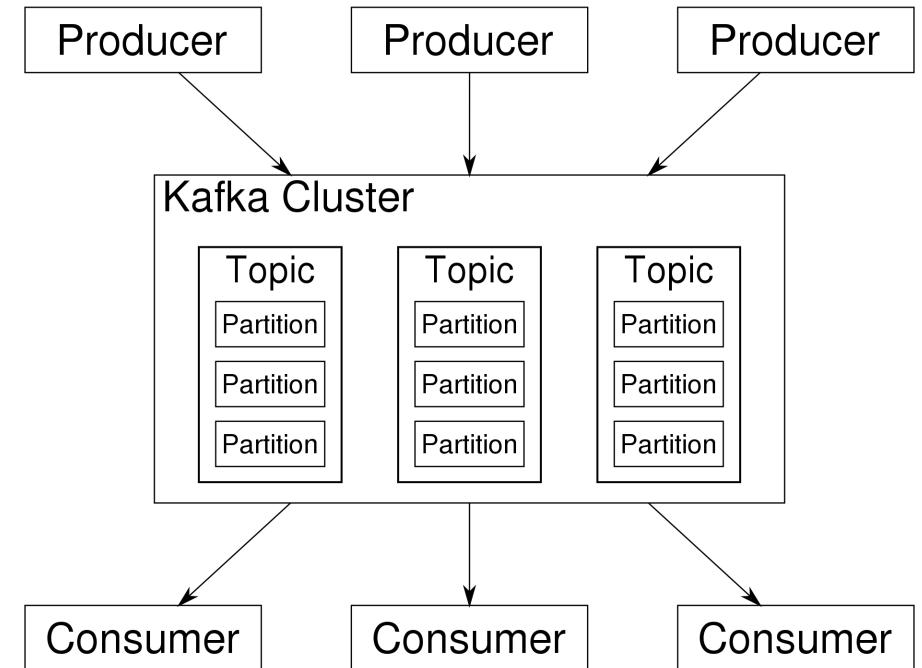
```
function getRecord(key, callback) {  
    establish database connection  
    retrieve the record for key  
    callback(record);  
}
```

```
function display(rec){  
    display rec on the web page  
}
```

```
getRecord('Rita', display) → A  
getRecord('John', display) → A → B
```

# Kafka: An Example of Asynchronous Software Design

- ❖ Today, streams of data records, including [streams of events](#), are continuously generated by many online applications.
  - ❖ A [streaming platform](#) enables the development of applications that can continuously and easily consume and process streams of data and events.
  - ❖ Apache [Kafka](#) (Kafka) is a free and open-source distributed [streaming platform](#) useful for building, *real time* or [asynchronous](#), event-driven applications.
  - ❖ Kafka offers [loose coupling](#) between *producers* and *consumers*.
  - ❖ Consumers have the option to either [consume](#) an event [in real time](#) or [asynchronously](#) at a later time.
  - ❖ Kafka maintains the [chronological order](#) of records/events, ensuring fault tolerance and durability.
  - ❖ To increase [scalability](#), Kafka separates a topic and stores each [partition](#) on a different node.
- ❖ [Producer API](#) – Permits an application to [publish](#) streams of records/events.
  - ❖ [Consumer API](#) – Permits an application to [subscribe](#) to topics and processes streams of records/events.



# COMP2511

## Command and Facade Patterns

Prepared by

Dr. Ashesh Mahidadia

# Design Patterns

## Creational Patterns

- ❖ Factory Method
- ❖ Abstract Factory
- ❖ Builder
- ❖ Singleton

## Structural Patterns

- ❖ Adapter
- ❖ Composite
- ❖ Decorator
- ❖ Façade

## Behavioral Patterns

- ❖ Iterator
- ❖ Observer
- ❖ State
- ❖ Strategy
- ❖ Template
- ❖ Visitor
- ❖ Command Pattern

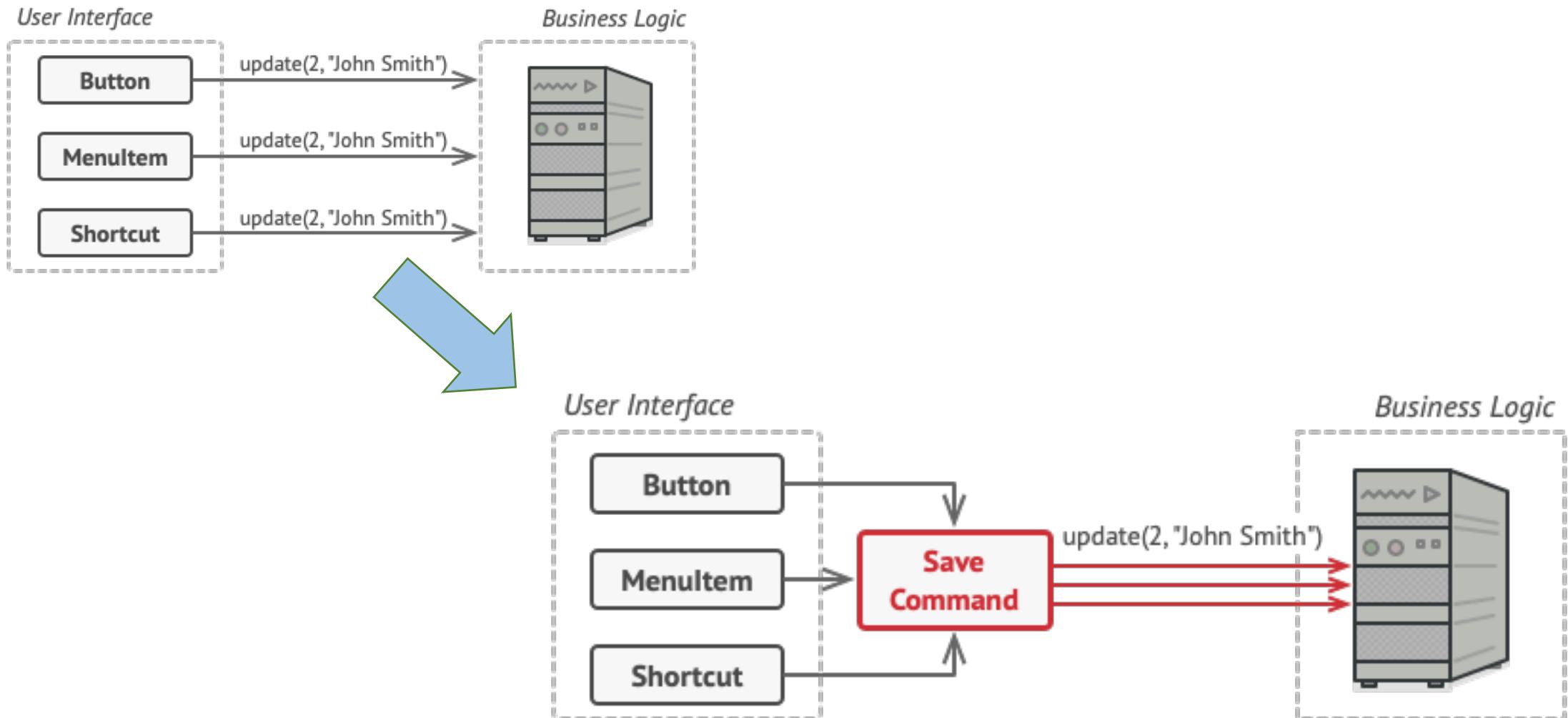
The lecture slides use material from the websites  
<https://refactoring.guru/design-patterns/>  
and the Head First Design Patterns reference book.

# Command Pattern

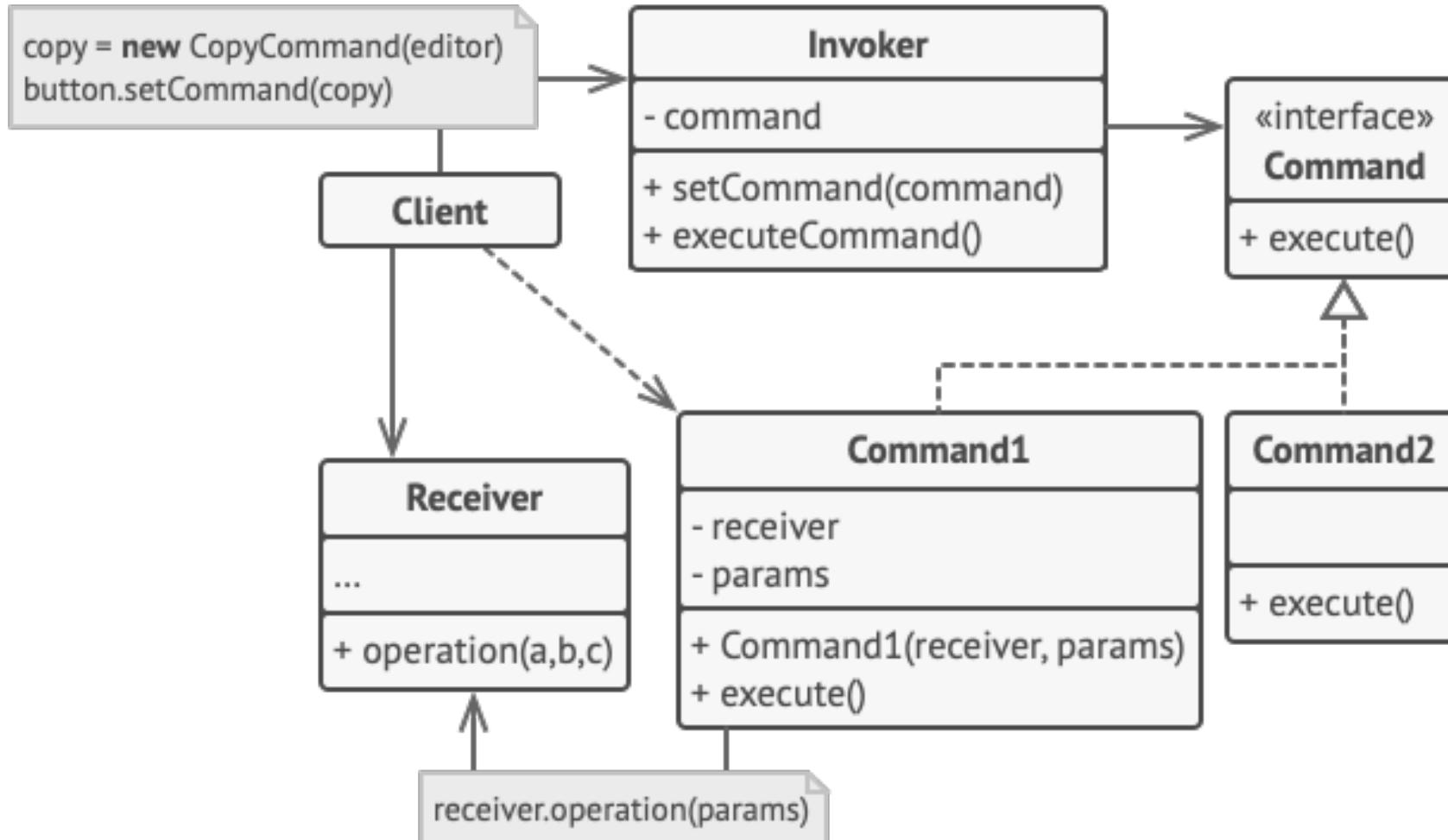
# Command Pattern

- ❖ The **Command Pattern** allows you to **decouple** the requester of an action from the object that actually performs the action.
- ❖ A **command object** encapsulates a request (i.e., turn on light) on a specific object (say, the living room light object).
- ❖ A command object is **associated** with an invoker (say a button).
- ❖ An invoker executes a **predefined** method on a command object, that in turn performs actions as per the associated request.
- ❖ An **invoker** (say a button) is **decoupled** from the original request (turn on light).
- ❖ We can easily change / substitute a command object, resulting in a different action.
- ❖ Command pattern is a **behavioral pattern**, it transforms a request into an object, allowing it to be passed as method arguments, serialized it, log it, queue it for delayed execution, etc.

# Command Pattern



# Command Pattern



# Command Pattern: Remote Control Example

```
public interface Command {  
    public void execute();  
}
```

```
public class LightOffCommand implements Command {  
    Light light;  
  
    public LightOffCommand(Light light) {  
        this.light = light;  
    }  
  
    public void execute() {  
        light.off();  
    }  
}
```

```
public class LightOnCommand implements Command {  
    Light light;  
  
    public LightOnCommand(Light light) {  
        this.light = light;  
    }  
  
    public void execute() {  
        light.on();  
    }  
}
```

```
public class Light {  
    String location = "";  
  
    public Light(String location) {  
        this.location = location;  
    }  
  
    public void on() {  
        System.out.println(location + " light is on");  
    }  
  
    public void off() {  
        System.out.println(location + " light is off");  
    }  
}
```

```
public class StereoOnWithCDCommand implements Command {  
    Stereo stereo;  
  
    public StereoOnWithCDCommand(Stereo stereo) {  
        this.stereo = stereo;  
    }  
  
    public void execute() {  
        stereo.on();  
        stereo.setCD();  
        stereo.setVolume(11);  
    }  
}
```

# Command Pattern: Remote Control Example

```
public class RemoteControl {
    // This is the invoker

    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }
}
```

# Command Pattern: Remote Control Example

Demo .....

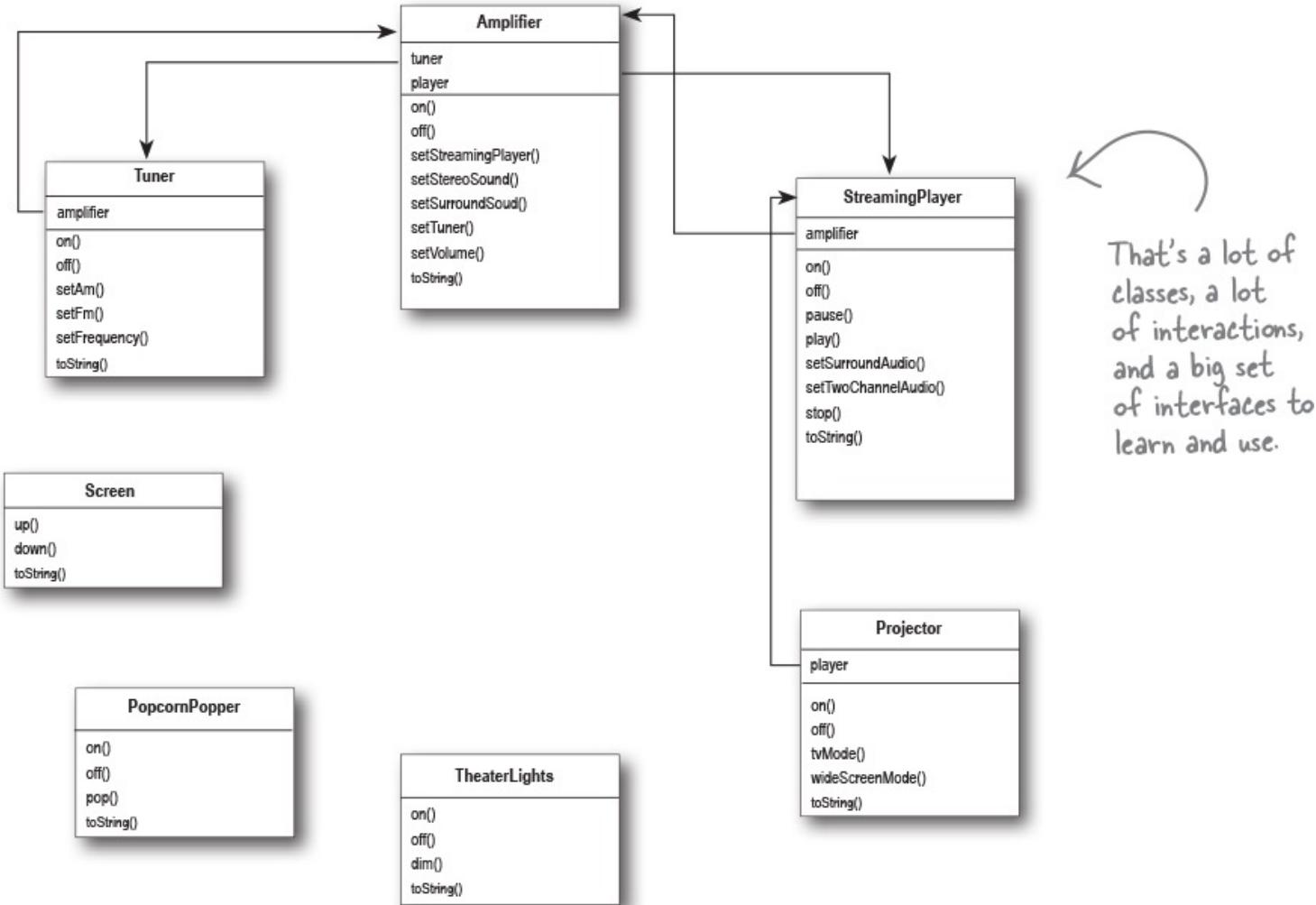
# Façade Pattern

- ❖ Façade offers a simplified interface (façade) to hide all the complexity of one or more classes .
- ❖ Adapter Vs Façade Patterns:
  - *Adapter Pattern*: Converts one interface to another (one a client is expecting)
  - *Façade Pattern*: Makes an interface simpler to a *complex* class/classes (subsystem)
- ❖ Facades offers a simplified interface to the underlying class/classes.
- ❖ Importantly, facades do NOT “*encapsulate*” the subsystem classes.
- ❖ The underlying **subsystem classes** and their **methods** are **still available** for direct use by clients. For example, in the *Home Theatre* example, methods of a projector, amplifier, etc.

# Example: Home Theatre

To watch a movie, you **need to perform a few tasks**:

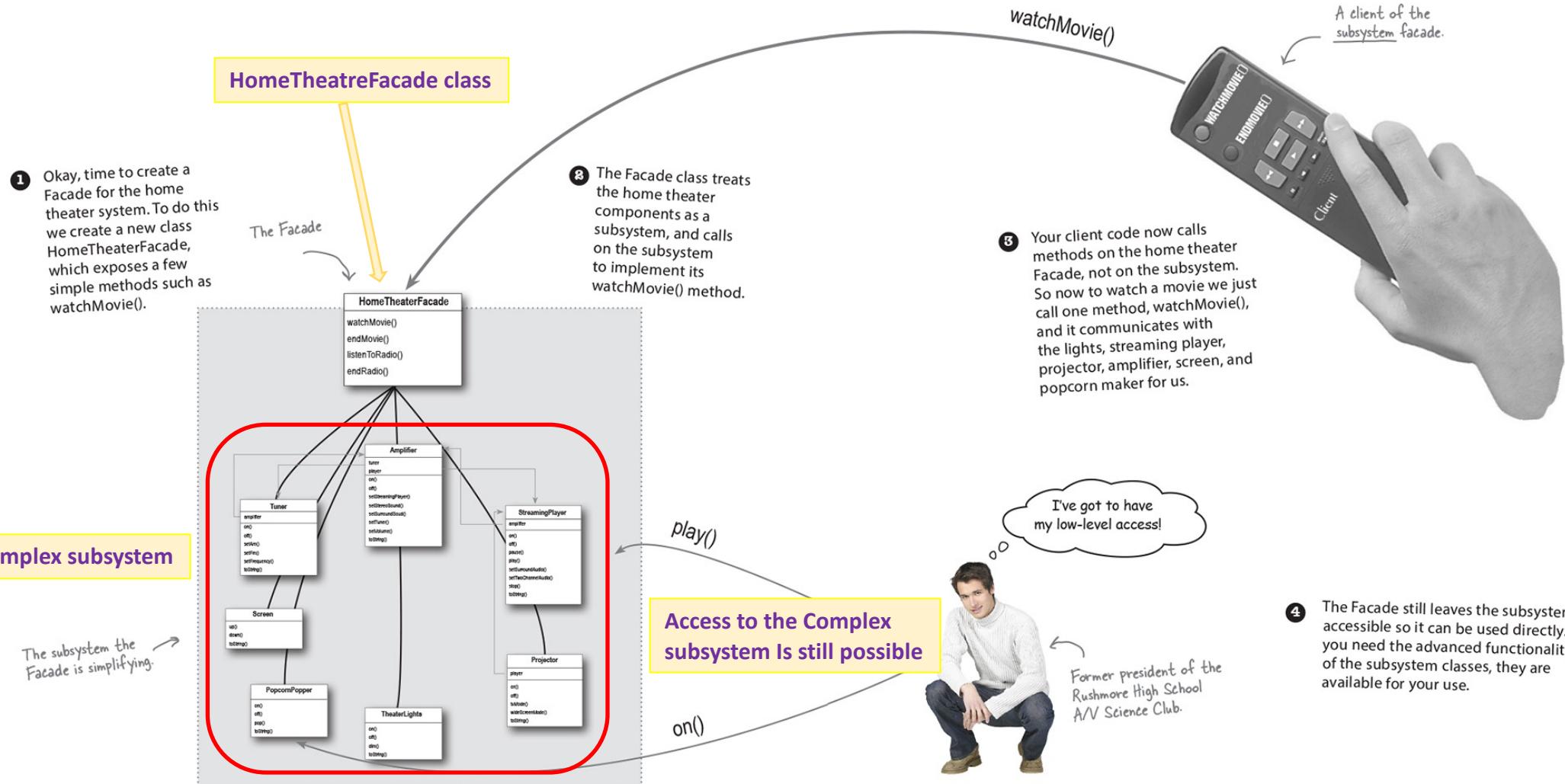
- ❖ Turn on the popcorn popper
- ❖ Start the popper popping
- ❖ Dim the lights
- ❖ Put the screen down
- ❖ Turn the projector on
- ❖ Set the projector input to streaming player
- ❖ Put the projector on widescreen mode
- ❖ Turn the sound amplifier on
- ❖ Set the amplifier to streaming player input
- ❖ Set the amplifier to surround sound
- ❖ Set the amplifier volume to medium (5)
- ❖ Turn the streaming player on
- ❖ Start playing the movie



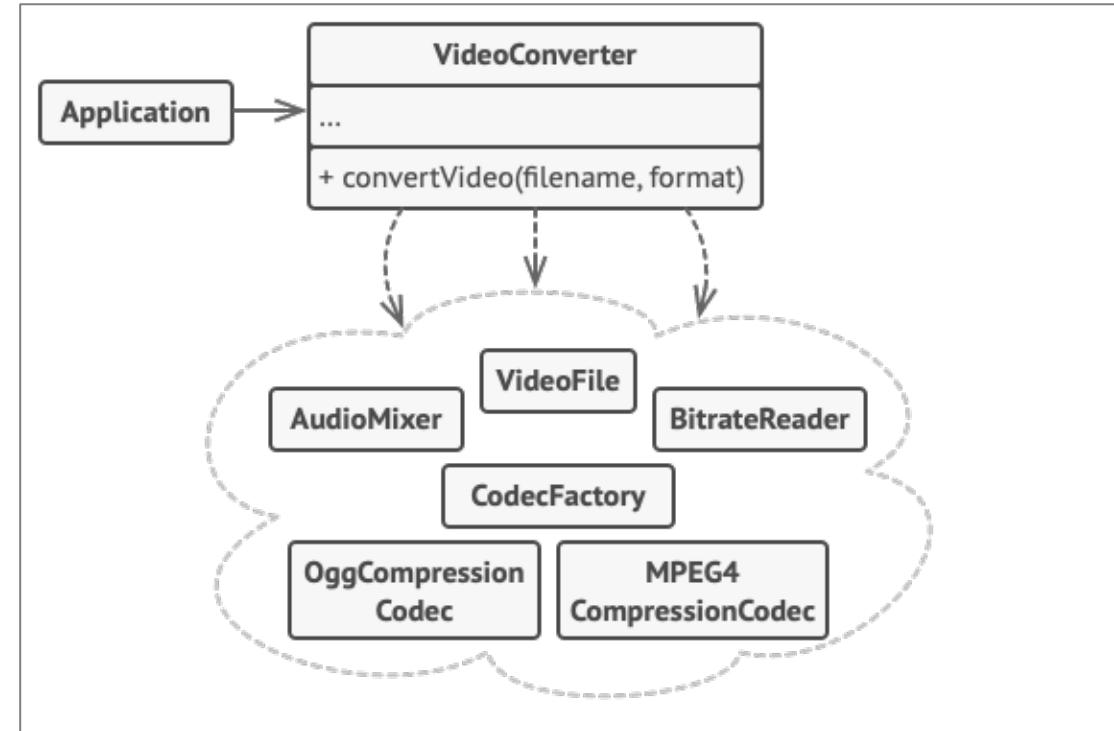
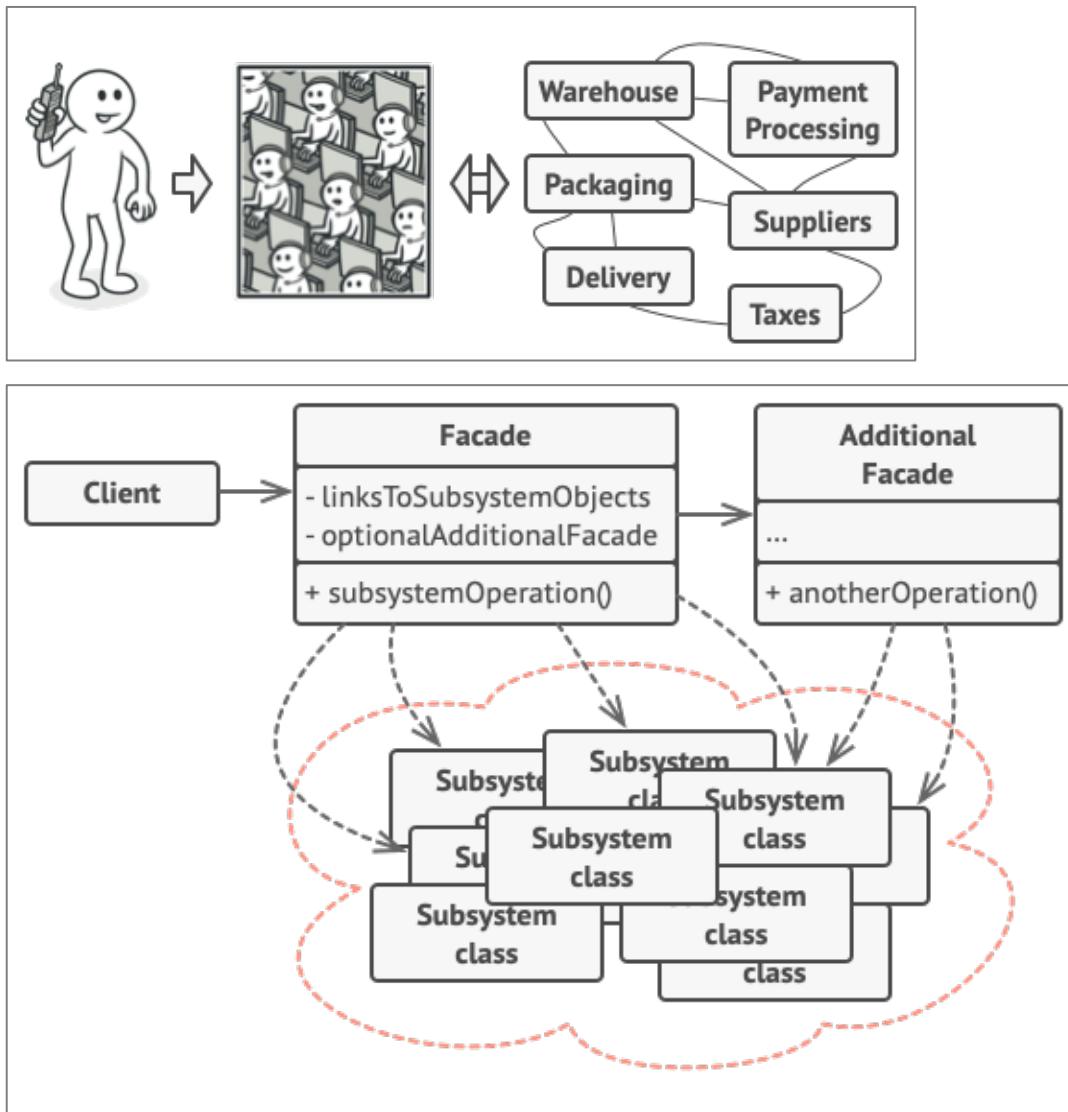
**Lot of interfaces to deal with!**

- ❖ Projector, Screen, Streaming Player, Theatre lights, Amplifier, Tuner, Theatre lights

# Example: Home Theatre with Façade class



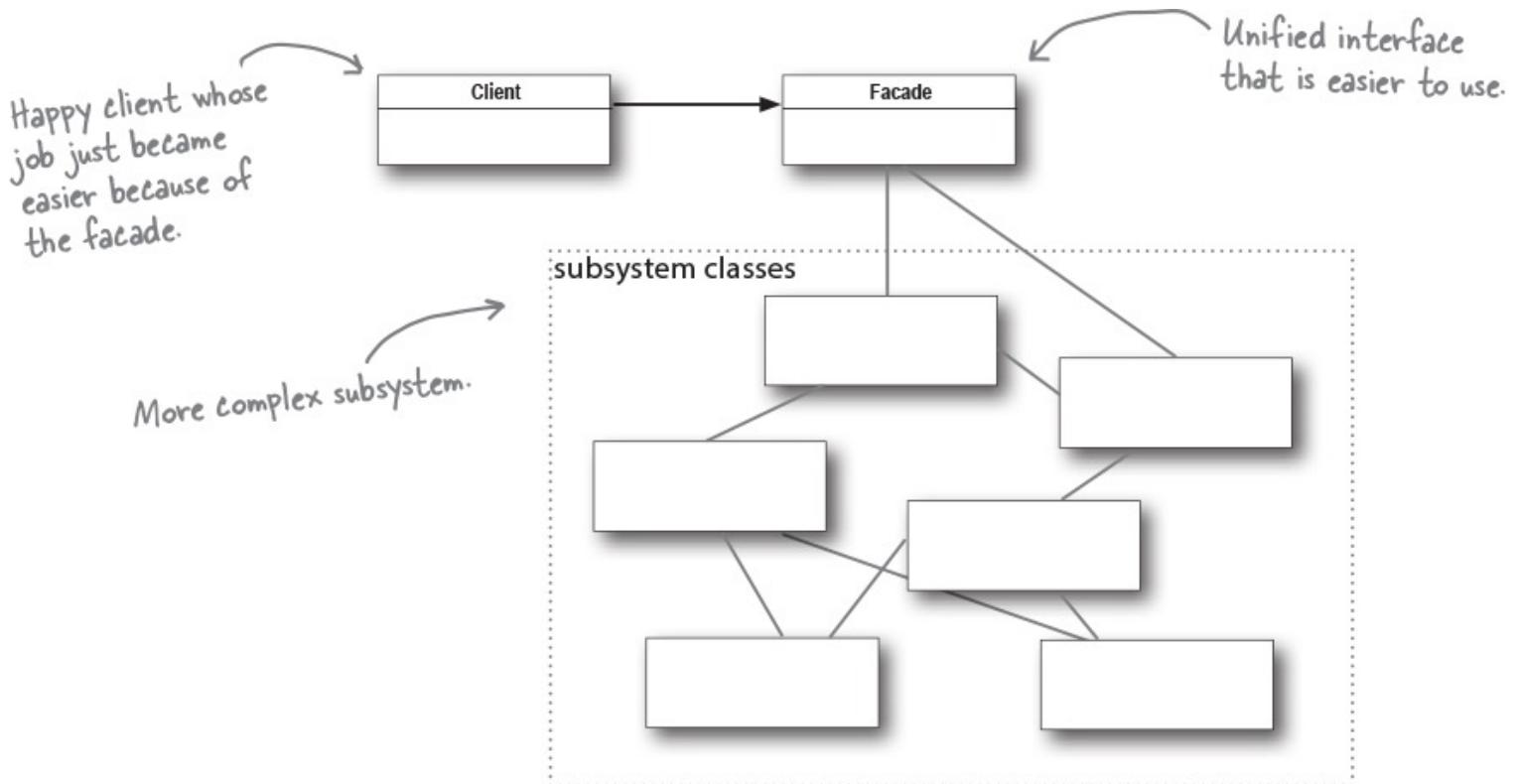
# Façade: Other Examples



From <https://refactoring.guru/design-patterns/facade>

# Façade Pattern

- ❖ **Important:** A facade can add **domain knowledge** to improve client experiences (i.e., set light intensity depending on a time of a day).
- ❖ A complex subsystem can have **multiple facades**, for different clients.
- ❖ The Façade Pattern **decouples** a client interface from any one of the subsystems. For example, you can change a type of your streaming player without changing the façade interface used by clients.



**End**

# COMP2511



## 9.1 - Risk Engineering

# In this lecture

- What is risk in Software Engineering?
- Mitigating risk
- Designing for Risk

# The Flaw in the Plan

- Why do plans (designs) not go according to plan? What went wrong?
  - Flaws in the implementation / execution of the plan/design
  - Flaws in the design/plan itself
- We can't always plan for everything up front
- Design flaws are often hard to spot; Risk is invisible
- Can only tell through design smells / red flags
- Over time, we learn to become better at recognising warning signs and identifying flaws earlier on
- It's not what happened right before things went wrong that was the problem - it is what happened **every step along the way** that got us to that point

# Design Debt, or Design Risk?

- **Risk** - the probability of a bad outcome occurring
- Design decisions come with a cost - "technical debt", the more technical debt, the more risk we accumulate
- Greater software complexity leads to more risk
- The design decisions and trade-offs we make are often the flaws in the plan - risks are inevitable
- How does this manifest itself?
  - Design problems often build in a "slow burn" fashion
  - Incidents, defects, bugs
  - Resistance to changes in software
  - These in turn present Business Risks

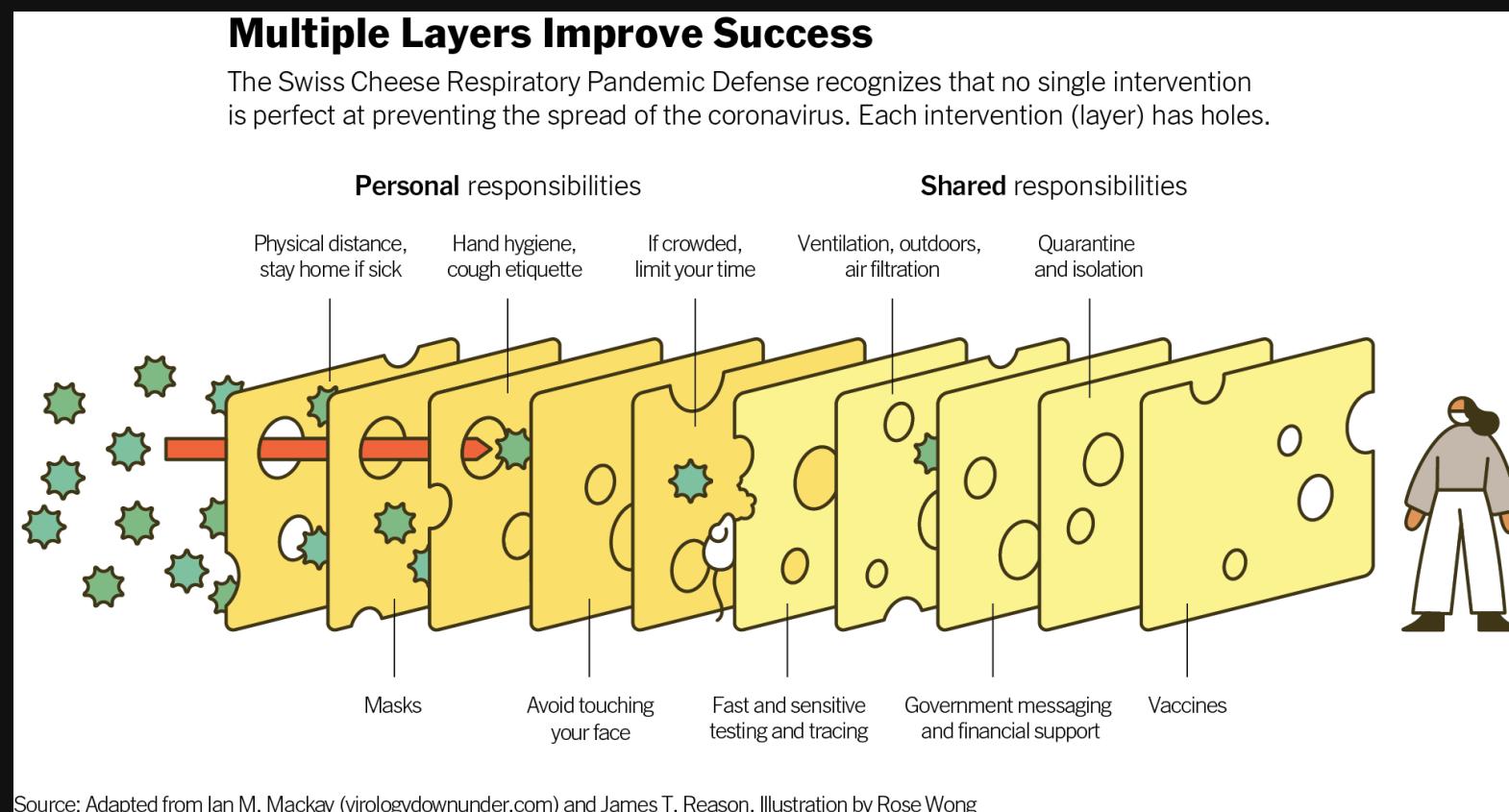
# Mitigating Risk

- Risks are centred around events, e.g. software breaking.
- Risk is often assessed in terms of **probability** and impact
- Mitigations of **probability**
  - Preventative measures that lower the chance of a bad outcome occurring
  - E.g. Looking both ways before crossing the street
- Mitigations of **impact**
  - Reactive measures that decrease the negative outcome in the event that something bad does occur
  - E.g. Wearing a bike helmet
- This is often termed **Quality Assurance**

How do we design for risk?

# Designing for Risk: Swiss Cheese Model

- James Reason - Major accidents and catastrophes reveal multiple, smaller failures that allow hazards to manifest as risks
- Each slice of cheese represents a barrier, each one of which can prevent a hazard from turning into consequences
- No single barrier is foolproof - each slice of cheese has "holes"
- When the holes all align, a risk event manifests as negative consequences

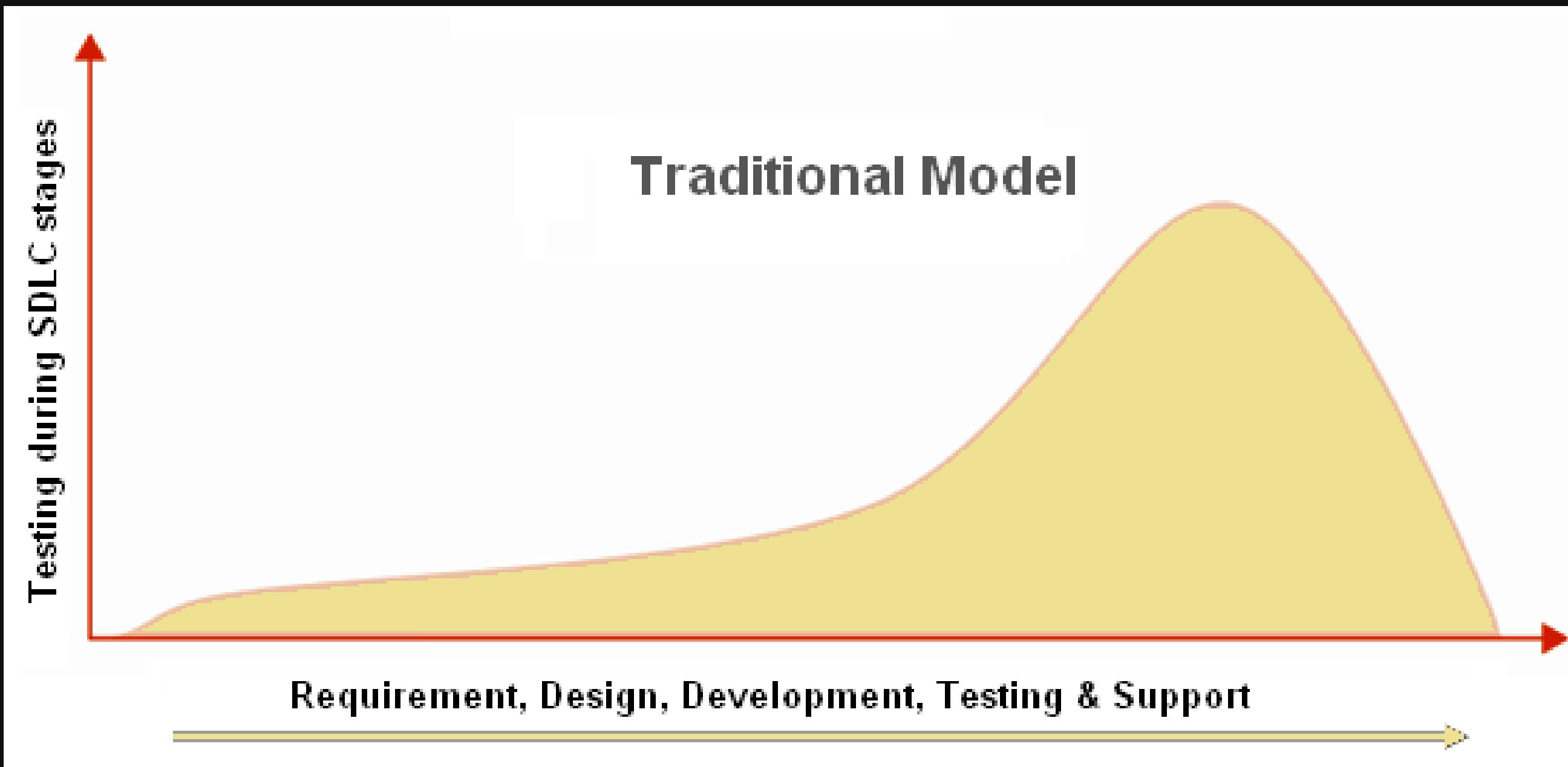


# Designing for Risk: Swiss Cheese Model

- Taking a layered approach to Software Safety
- Testing at multiple levels:
  - Static verification
  - Unit and integration tests
  - Usability tests
  - Design and code reviews
  - CI pipelines
- Sometimes referred to as **containment barriers**
- A defensive approach; multiple checks and balances in place
- Probability is **multiplicative** ( $X \text{ AND } Y \text{ AND } Z = P(X) * P(Y) * P(Z)$ )

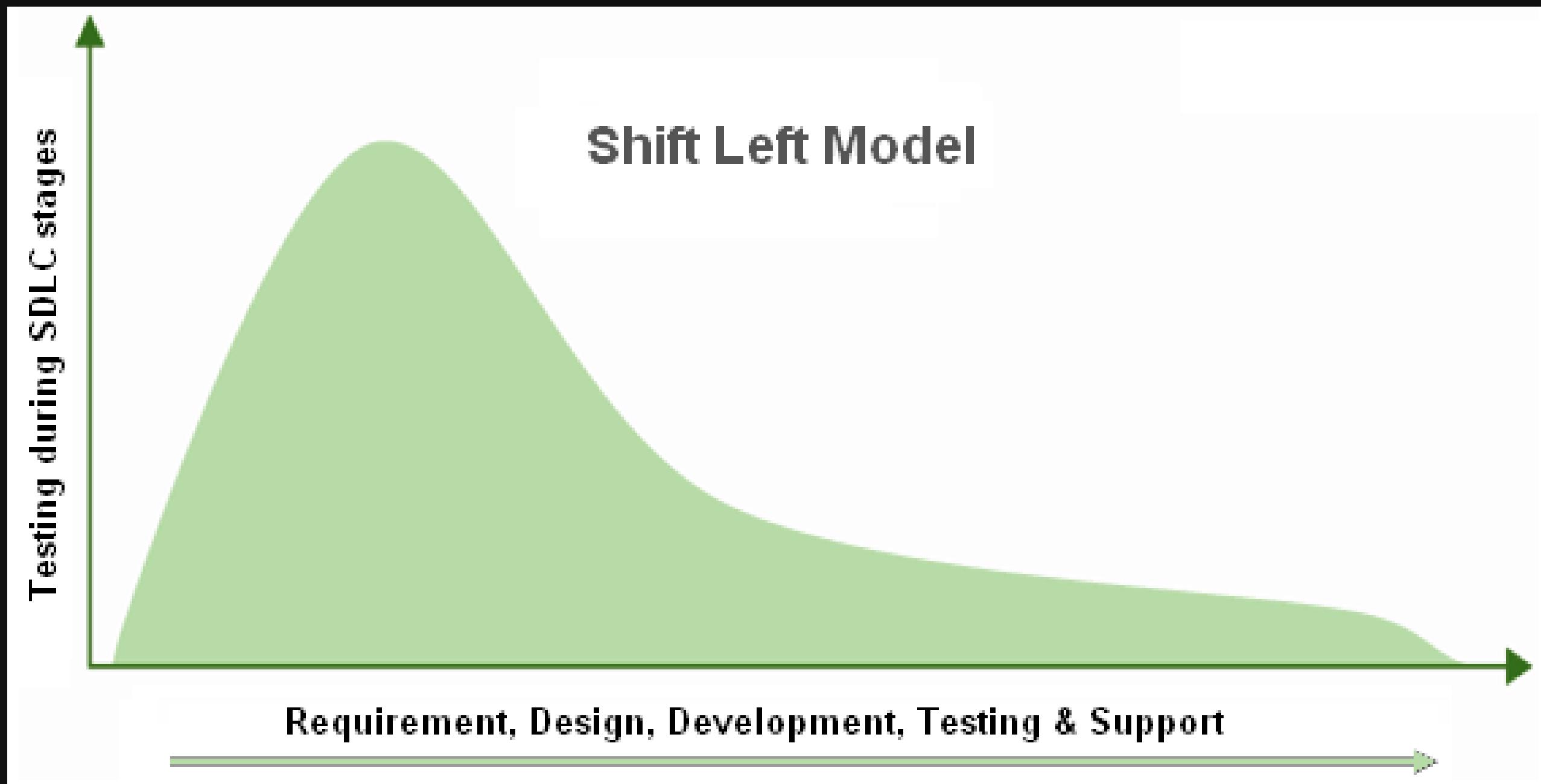
# Designing for Risk: Shifting Left

A waterfall / big design up front approach to quality assurance.



# Designing for Risk: Shifting Left

Shift Left: A practice intended to find and prevent problems early in the engineering process.



# Designing for Risk: Shifting Left

- Shifting Left in principle: Moving risk forward in the software development timeline and designing systems and processes that are built for continuous testing
- What does shifting left involve in practice?
  - Automated testing over manual testing
  - Continuous Integration
  - Test-Driven Development

# Shifting Left: An Example

- Let's take an example - a python script which runs on a remote server
- There is an error in the code, and the code fails when attempting to run a usability test

```
1 $ python3 -m svc.create_repo test
2 Traceback (most recent call last):
3   File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/runpy.py", line 194, in _run_module_as_main
4     return _run_code(code, main_globals, None,
5   File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/runpy.py", line 87, in _run_code
6     exec(code, run_globals)
7   File "/Users/nicholaspatrikeos/Desktop/COMP2511-22T3/administration/svc/create_repo.py", line 11, in <module>
8     PROJECT = gl.projects.get(f'{NAMESPACE}/{TERM}/STAFF/repos/{REPO}')
9 NameError: name 'REPO' is not defined
```

- How could we shift left here?

# Shifting Left: Dynamic Verification + CI

- We can dynamically verify the correctness of the code and automatically run the tests in a pipeline:

```
1 $ pytest
2 ===== test session starts =====
3 platform darwin -- Python 3.8.8, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
4 rootdir: /Users/nicholaspatrickos/Desktop/COMP2511-22T3/administration
5 plugins: hypothesis-6.1.1, xdist-2.2.1, timeout-1.4.2, forked-1.3.0
6 collected 1 item
7
8 create_repo_test.py F
[100%]
```

checks



test

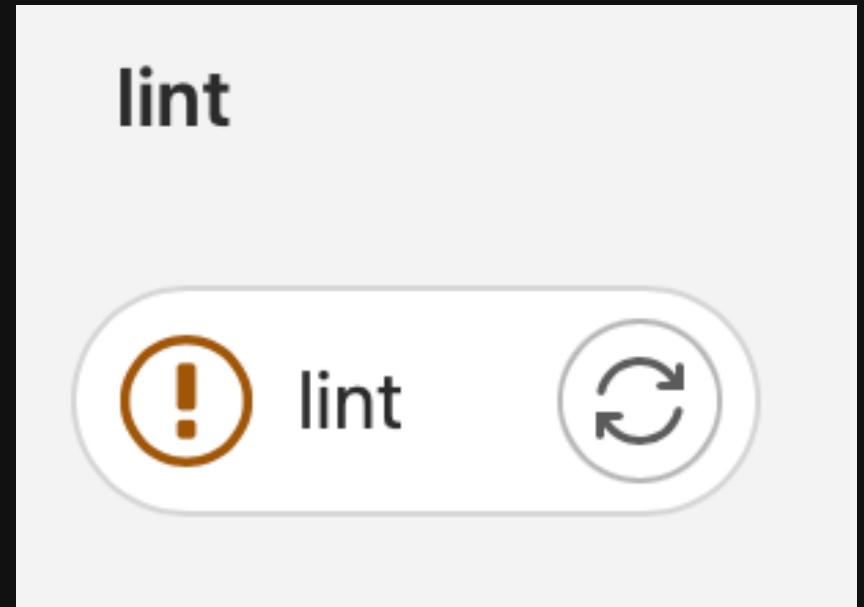


- Problem here - we are still having to run our tests in order to pick up a simple name error, this takes a long time to catch a small problem

# Shifting Left: Static Verification + CI

- We can statically verify the correctness of our code, which is faster than running all the tests using a linter or a type checker:

```
1 $ pylint svc/*.py
2 **** Module svc.create_repo
3 svc/create_repo.py:11:64: E0602: Undefined variable 'REPO' (undefined-variable)
4 svc/create_repo.py:19:31: E0602: Undefined variable 'REPO' (undefined-variable)
5 svc/create_repo.py:27:24: E0602: Undefined variable 'REPO' (undefined-variable)
6
7 -----
8 Your code has been rated at 9.61/10 (previous run: 10.00/10, -0.39)
```



- Problem here - we are still having to push to the CI for our breaking changes to be contained. Can we enforce running them before?

# Shifting Left: Local Configurations

- Pre-commit hooks and IDE tools can give us more friendly experiences that detect these problems earlier in the development loop, e.g.

A screenshot of a code editor showing a Python script. The code includes a pre-commit hook for usage validation and a main block for interacting with GitLab. A pylint error is highlighted in the main block, indicating an undefined variable 'REPO'. A tooltip provides the error message and a 'Peek Problem' link.

```
if len(sys.argv) != 2:  
    print('Usage python3 -m svc.create_repo [repo]')  
  
if __name__ == '__main__':  
    gl = configure_gitlab()  
    PROJECT = gl.projects.get('f1$NAMESPACE1/STEAM1/STAFF/repos/SPEDOL1')
```

Undefined variable 'REPO' pylint(undefined-variable)  
Peek Problem (F8) No quick fixes available

- Ideally, static verification is "baked in" to our programming language rather than added on...

# Shifting Left: Type Safety

- Types are **statically verifiable** - meaning that we can ensure correctness **earlier on in the development process**, shifting left
- In Java, code that doesn't adhere to the rules of the type system fails to compile
  - a significant containment barrier
- Extensions like mypy and TypeScript allow for an add-on of type checking
- Unlike Java however, type safety wasn't part of the Big Design Up Front for Python and JS
- Modern software design is favouring **statically typed languages** for these reasons

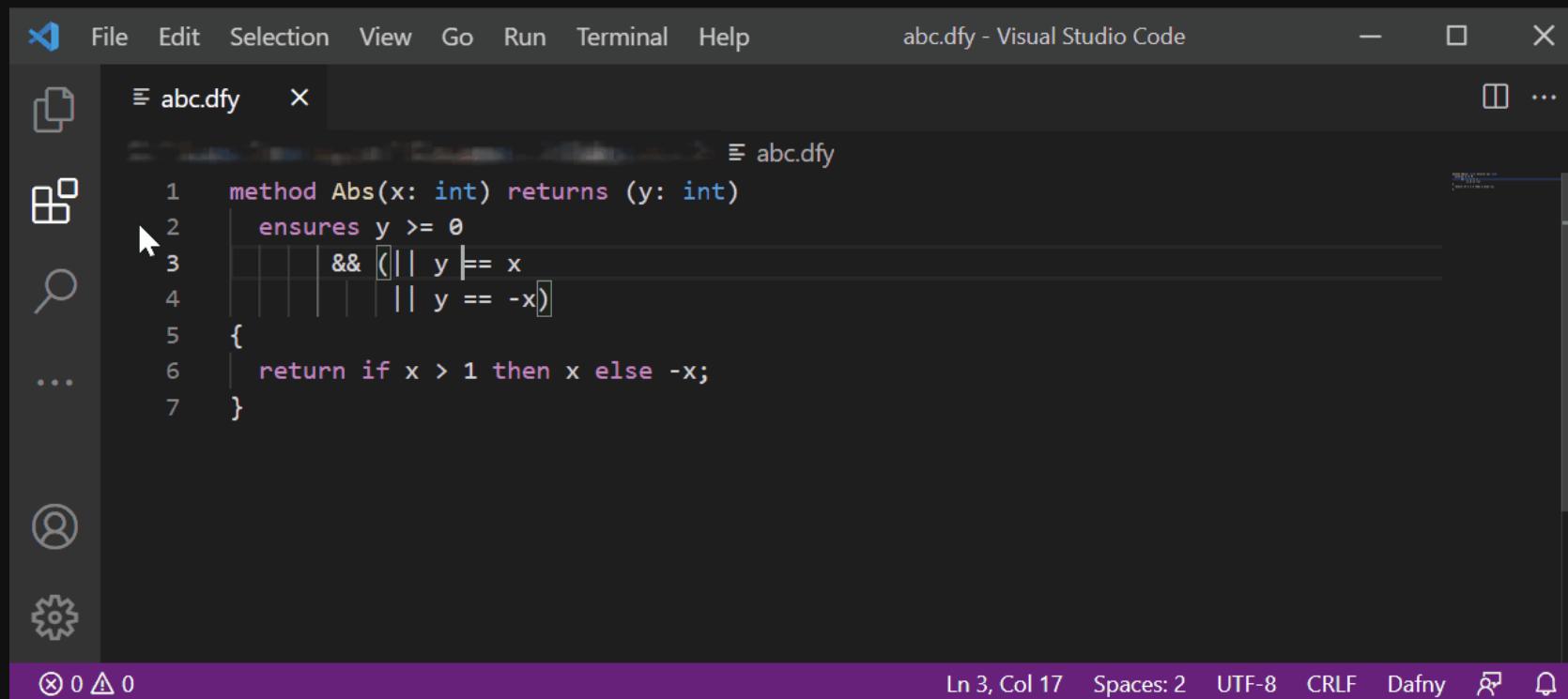
```
1 def my_function(message):
2     if message == 'hello':
3         return 1
4
5     return '0'
6
7 result = my_function('goodbye')
```

# Shifting Left: Type Safety

- Features of type systems:
  - Ability to define custom types (typedefs)
  - Inheritance, Subtypes and Supertypes
  - Interfaces
  - Generics
  - Unit types
  - Enums
- Well-designed type systems allow us to verify more of our code statically

# Shifting Left: More Static Verification & Design by Contract

- Some programming languages (e.g. Dafny) allow for more static verification than just type checking - they can prove or disprove code according to a **declarative contract** where preconditions, postconditions and invariants are specified
- Dafny makes use of a theorem prover which checks how well the implementation matches the specification (contractual correctness)



A screenshot of Visual Studio Code showing a Dafny file named "abc.dfy". The code defines a method `Abs` that takes an integer `x` and returns an integer `y`. The method ensures that `y` is non-negative and satisfies either `y == x` or `y == -x`. The implementation returns `x` if `x > 0` and `-x` otherwise. The code is annotated with a cursor highlighting the logical operator `&&`.

```
1  method Abs(x: int) returns (y: int)
2    ensures y >= 0
3    && [| y == x
4    || y == -x|]
5    {
6      return if x > 0 then x else -x;
7    }
```

# Summary

- Risk forms a large part of modern-day Software Engineering
- Designing for risk:
  - Considering risks in the design process;
  - Designing processes to accomodate for risk.
- Murphy's law: Anything that can go wrong, will.

# COMP2511



## 9.2 - Introduction to Microservices (Bonus)

# In this lecture

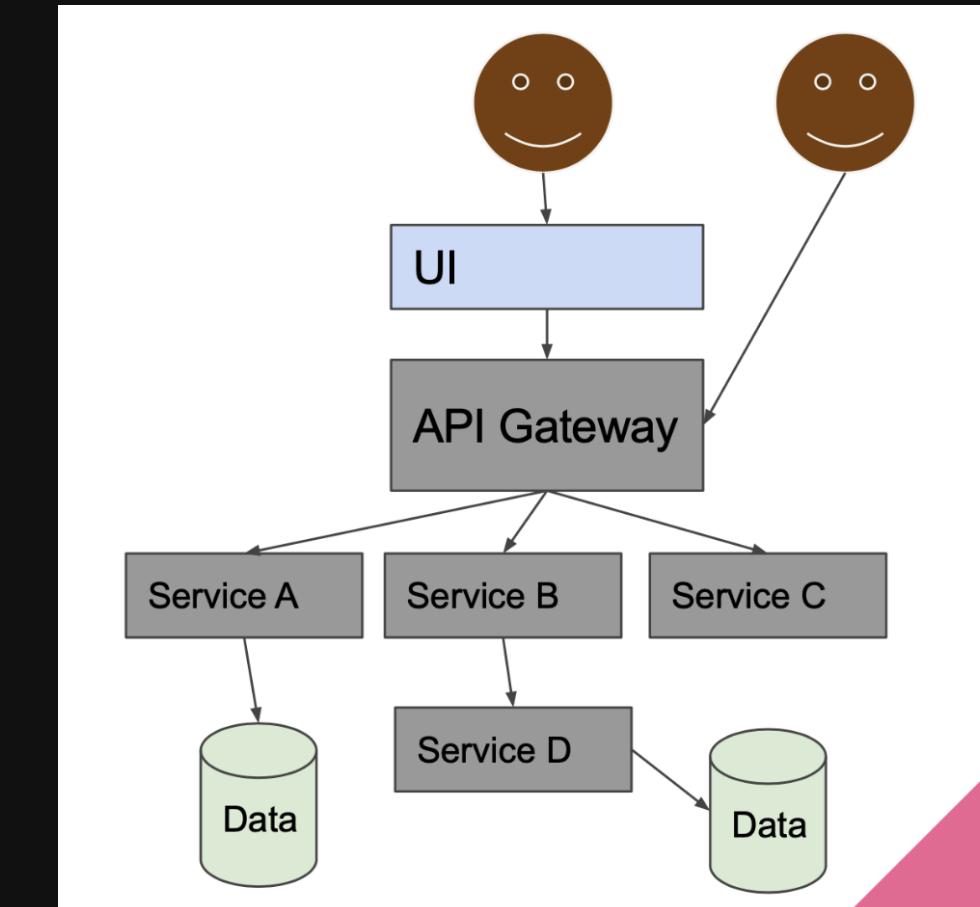
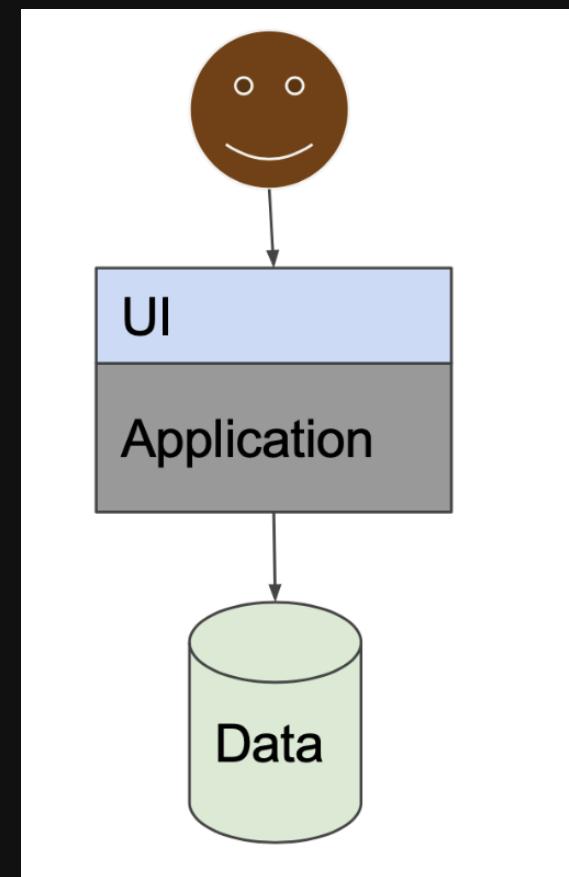
- Service-Oriented Architecture
- Monolith vs Microservices
- Microservice Ecosystem and technologies
- Trade-offs of a Microservice Approach

# Service-Oriented Computing

- The era of cloud computing - a move from software as **products** to software as **services**
- **Infrastructure as a Service** - physical / virtual machines to run code on is provided as a service
- **Platform as a Service** - hardware and operating system are provided and accessed remotely by developers
- **Software as a Service** - hardware, operating system and software are outsourced and accessed remotely and used by users
- Platform layers and **platformisation** in PaaS

# Monolith vs Microservices

- **Monolith:** a single large application that contains the entire software solution
  - One service to rule them all
- **Microservices:** A series of small-scale services that communicate with one another
  - Each service does *one task well*
- Where have we seen this before?



## HOW STANDARDS PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)

SITUATION:  
THERE ARE  
14 COMPETING  
STANDARDS.

14?! RIDICULOUS!  
WE NEED TO DEVELOP  
ONE UNIVERSAL STANDARD  
THAT COVERS EVERYONE'S  
USE CASES.

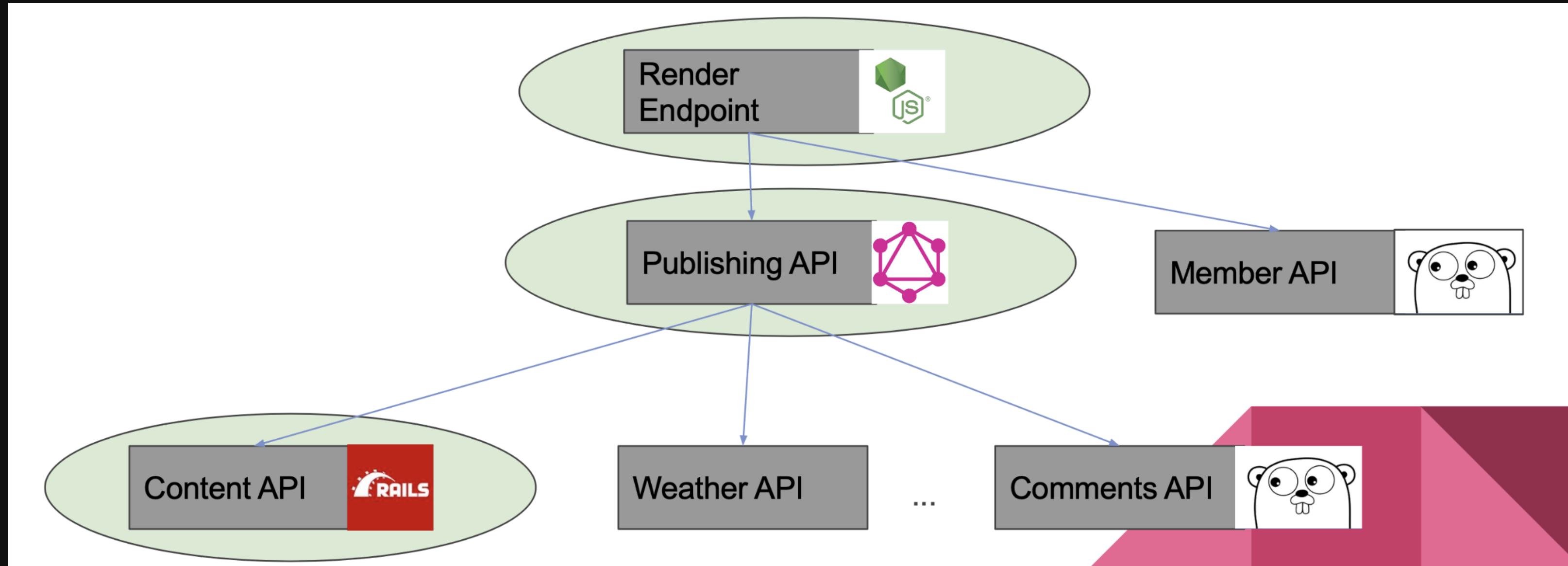


YEAH!

SOON:

SITUATION:  
THERE ARE  
15 COMPETING  
STANDARDS.

# Microservices Example



# Microservice Ecosystem: Data Interchange

- Services need to be able to communicate to one another over a **common interface**
- Synchronous interchange technologies
  - REST
  - gRPC - uses Protobuf, good for Service to Service communication
  - GraphQL - a query languages allowing for dynamic querying of data - good for public-facing APIs
- Asynchronous interchange technologies
  - Apache Kafka
  - Amazon SQS / SNS
- **Eventual consistency** - propagation of state so that Services have same logical state model

# Microservice Ecosystem: Deployment

- Amazon & AWS - IaaS
  - Applications - Elastic Beanstalk
  - Compute - EC2
  - Compute - Lambdas - Serverless Deployment
  - Storage - S3
  - Database - DynamoDB / RDS
- **Containerisation** - Docker images
- **Observability** - tools to help you understand what's happening inside a deployed application
- **Feature Flags** - switchboards to toggle and incrementally release new parts of code
- ... and much, much more

# Trade-offs: Microservice Benefits

- Freedom for service-specific programming languages / technology stacks
- Less responsibility, less coupling
- Easier to test
- Faster build and release cycles
- Lower risk per-microservice
- Not a single point of failure
- Easier to scale individual services

# Trade-offs: Microservice Costs

- Either everything breaks, or the glue breaks - how much time and money is actually saved?
- Dealing with distributed systems
  - Reliance on network connections
  - Communication latency
  - Consistency between services running in parallel
- Overhead, complexity and risk in orchestrating services in an end-to-end use case
- More complex deployment
- Security - now need to authenticate for every service, not just one
- Debugging is more difficult as control flows over different services (distributed tracing)

# Summary

- All Software Architecture is making trade-offs
- Monoliths grow too large, complex and risky and too difficult to scale
- Microservices present an alternative, which have their own challenges

# COMP2511

## Design Patterns Summary

Prepared by

Dr. Ashesh Mahidadia

# Design Patterns

## Creational Patterns

- ❖ Factory Method
- ❖ Abstract Factory
- ❖ Builder
- ❖ Singleton

## Structural Patterns

- ❖ Adapter
- ❖ Composite
- ❖ Decorator
- ❖ Façade

## Behavioral Patterns

- ❖ Iterator
- ❖ Observer
- ❖ State
- ❖ Strategy
- ❖ Template
- ❖ Visitor
- ❖ Command Pattern

The lecture slides use material from the Head First Design Patterns reference book.

# Quiz...

## Description

---

Wraps an object and provides a different interface to it.

Subclasses decide how to implement steps in an algorithm.

Subclasses decide which concrete classes to create.

Ensures one and only one object is created.

Encapsulates interchangeable behaviors and uses delegation to decide which one to use.

Clients treat collections of objects and individual objects uniformly.

Encapsulates state-based behaviors and uses delegation to switch between behaviors.

Provides a way to traverse a collection of objects without exposing its implementation.

# Quiz...

- Simplifies the interface of a set of classes.
- Wraps an object to provide new behavior.
- Allows a client to create families of objects without specifying their concrete classes.
- Allows objects to be notified when state changes.
- Encapsulates a request as an object.

# Design Patterns Summary

- ❖ Let Design Patterns emerge in your designs; **don't force** them in just for the sake of using a pattern.
- ❖ Design Patterns **aren't set in stone**; adapt and tweak them to meet your needs.
- ❖ Always use the **simplest solution** that meets your needs, even if it doesn't include a pattern.
- ❖ Study Design Patterns **catalogs** to familiarize yourself with patterns and the relationships among them.
- ❖ **Pattern classifications** (or categories) provide groupings for patterns. When they help, use them.
- ❖ Remember, most patterns you encounter will be adaptations of existing patterns, not new patterns.
- ❖ Build your team's **shared vocabulary**. This is one of the most powerful benefits of using patterns.

**End**

**COMP2511**

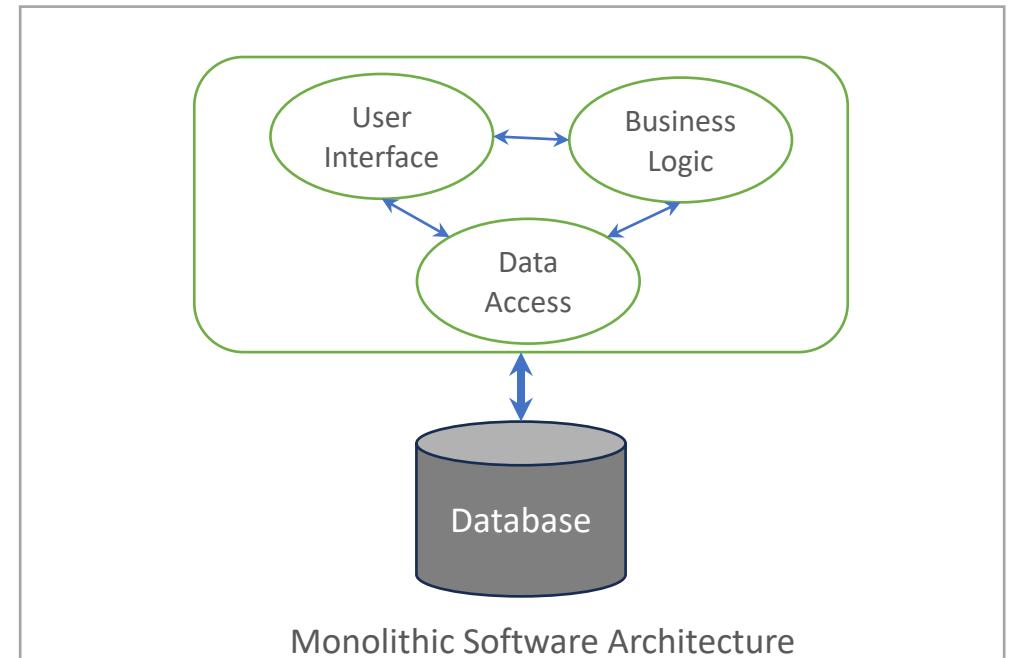
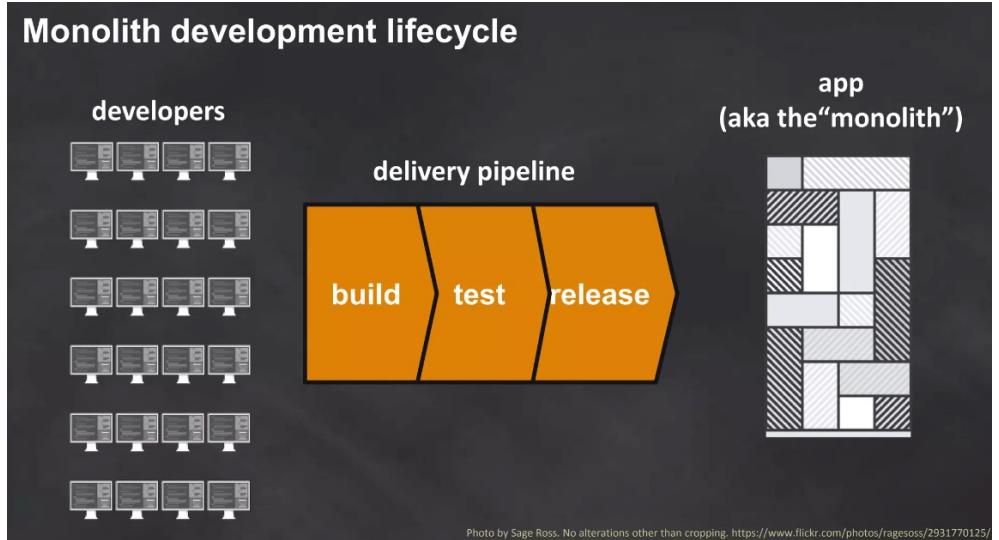
# Microservices Software Design

Prepared by

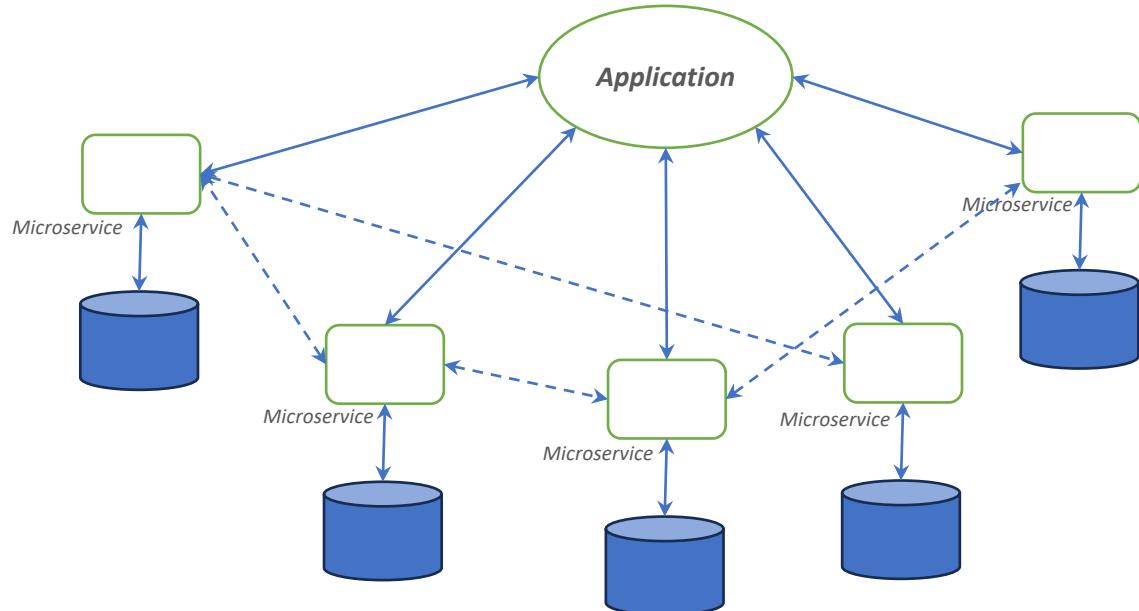
Dr. Ashesh Mahidadia

# Monolithic Architecture

- ❖ Long cycle times for building, testing, and releasing.
- ❖ Lack of agility.
- ❖ The absence of agility hinders the progress of innovations.
- ❖ Due to significant coupling, reusability is difficult.
- ❖ Often difficult to scale.



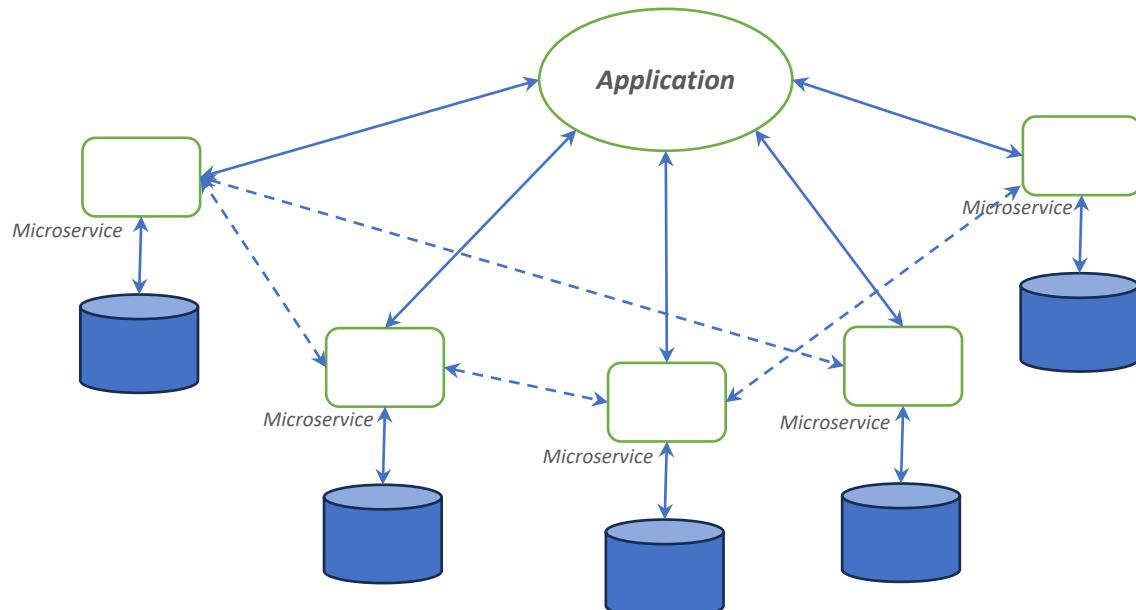
# Microservices Architecture



Microservices Software Architecture

- ❖ Microservices architecture is an architectural pattern that arranges an application as a collection of **loosely coupled** services.
- ❖ Each service is **independently** designed, developed, deployed, and maintained.
- ❖ Microservices are often developed **based on functionality**. For example, a service to manage shipping, an order management service, an inventory management service, and so on.

# Microservices Architecture



Microservices Software Architecture

- ❖ To accomplish **loose coupling**, services **only utilise** the appropriate **APIs** to communicate with other services.
- ❖ To enable the service to be utilised in **a variety of ways**, patterns such as **adapter** and **facade** are often used to offer **multiple interfaces** for the same service.
- ❖ A service offers encapsulation and abstraction.

# Advantages of Microservices

- ❖ Individual services can be added, updated or replaced without affecting other services, provided that the service contracts (APIs) are upheld.
- ❖ Different software and hardware platforms can be used by different services; for example, Java on Windows 10 on Azure, Python on Linux on AWS, Javascript on Nodejs on local server, etc.
- ❖ Only the most in-demand services need to be scaled, there is no need to scale the entire system.
- ❖ A service could be reused easily.
- ❖ Software complexity could be minimised.

# Things to Consider

- ❖ Interservice communication [latency](#).
- ❖ [Idempotency](#) must be considered in design. That is, performing the same action several times leads in the same outcome.
- ❖ Avoid using [shared data repositories/databases](#) and instead design for [data locality](#).
- ❖ The final system should handle [individual failures gracefully](#).
- ❖ It is necessary to plan for [\*eventual consistency\*](#).
- ❖ Maintaining a diverse set of services [could be a challenge](#), and we need to orchestrate deployment and maintenance carefully.

End

# COMP2511

## Visitor Pattern

Prepared by

Dr. Ashesh Mahidadia

# Design Patterns

## Creational Patterns

- ❖ Factory Method
- ❖ Abstract Factory
- ❖ Builder
- ❖ Singleton

## Behavioral Patterns

- ❖ Iterator
- ❖ Observer
- ❖ State
- ❖ Strategy
- ❖ Template
- ❖ Visitor

## Structural Patterns

- ❖ Adapter
- ❖ Composite
- ❖ Decorator

# Visitor Pattern

Some of the material is from the websites <https://refactoring.guru/design-patterns/> and the wikipedia pages.

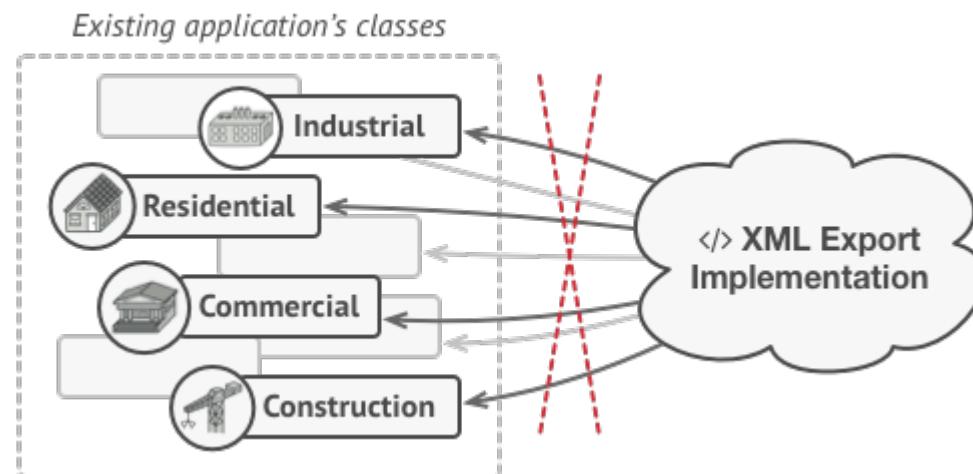
# Visitor Pattern

- ❖ Visitor is a **behavioral** design pattern that adds new operations/behaviors to the existing objects, without modifying them.
- ❖ The visitor design pattern is a way of **separating** an algorithm from an object structure on which it operates.
- ❖ A practical result of this separation is the ability to **add new operations** to existing object structures without modifying the structures.
- ❖ It is one way to follow the **open/closed principle**.
- ❖ A **visitor class** is created that **implements** all of the appropriate specializations of the **virtual operation/method**.
- ❖ The visitor **takes** the **instance reference** as input, and implements the goal (additional behavior).
- ❖ Visitor pattern can be added to **public APIs**, allowing its clients to perform operations on a class without having to modify the source.

# Visitor Pattern

## Problem:

- A geographic information **structured** as one colossal **graph**.
- Each **node** of the graph may represent **a city, an industry, a sightseeing area, etc.**
- Each node type is represented by its own class, while each specific node is an object.
- **Task:** you want to **export** the graph into **XML** format.



*The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.*

# Visitor Pattern

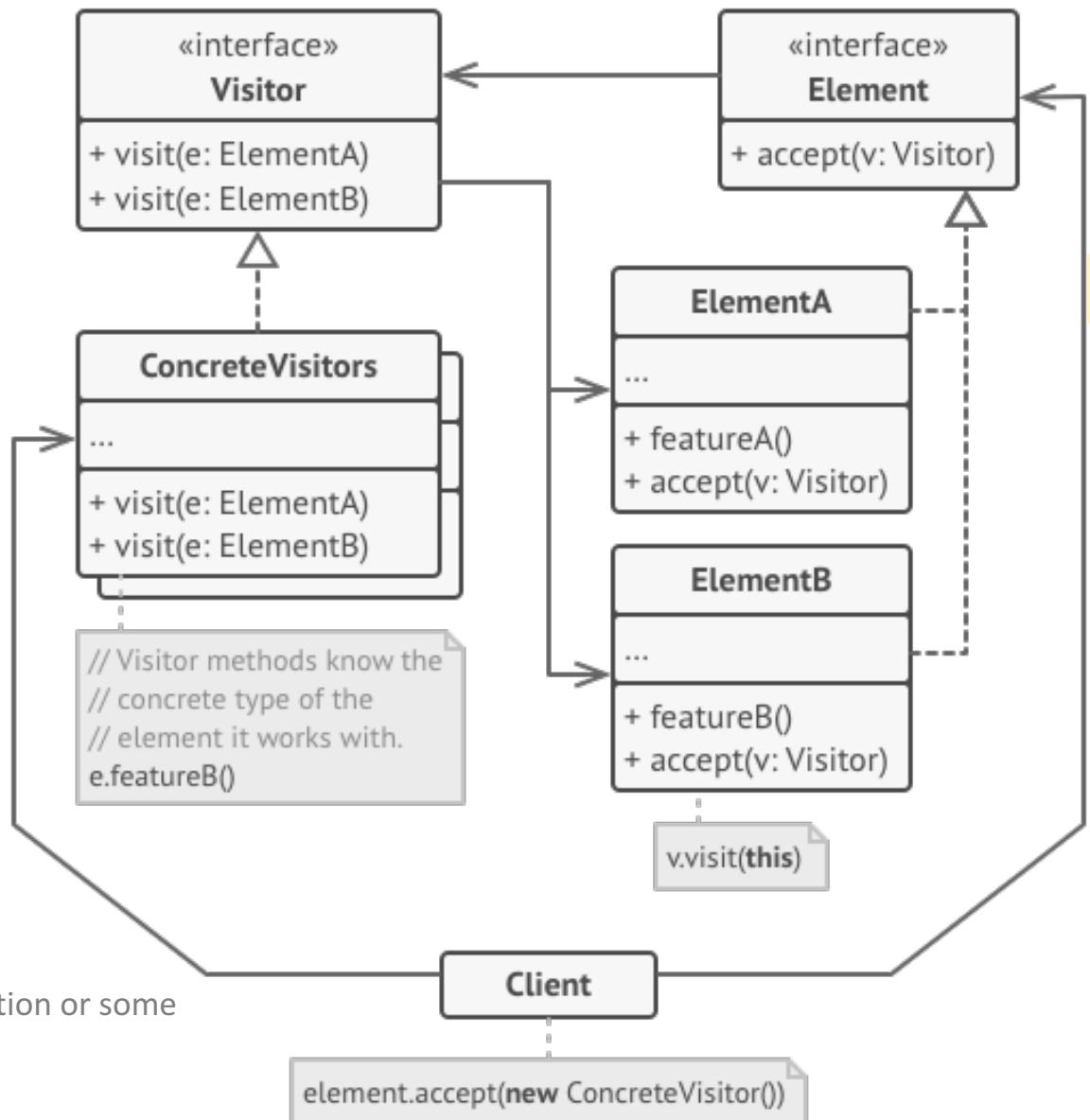
## Solution:

- ❖ The Visitor pattern suggests that you place the new behavior into a **separate class** called **visitor**, instead of trying to integrate it into existing classes.
- ❖ The original object that had to perform the behavior is now passed to one of the **visitor's methods as an argument**, providing the method access to all necessary data contained within the object (see the example for more clarification).
- ❖ The **visitor class** need to define **a set of methods**, one for each type.  
For example, a city, a sightseeing place, an industry, etc.
- ❖ The visitor pattern uses a technique called “**Double Dispatch**” to execute a suitable method on a given object (of different types).
  - An object “**accepts**” a **visitor** and tells it what visiting method should be executed. See the example for more clarifications.
  - One additional method allows us to **add further behaviors without** further altering the code.

# Visitor Pattern: Structure

1 The **Visitor** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments.

2 Each **Concrete Visitor** implements several versions of the same behaviors, tailored for different concrete element classes.



3 The **Element** interface declares a method for “accepting” visitors. This method should have one parameter declared with the type of the visitor interface.

4 Each **Concrete Element** must implement the acceptance method. The purpose of this method is to **redirect the call to the proper visitor's method corresponding to the current element class**. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.

5

The Client usually represents a collection or some other complex object (for example, a Composite tree).

# Visitor Pattern: Example-1

```
public interface CarElementVisitable {  
    void accept(CarElementVisitor visitor);  
}
```

Read the **example code** discussed in the lectures,  
and also **provided** for this week

```
public class Wheel implements CarElementVisitable {  
  
    private final String name;  
    private double cost = 210;  
  
    public Wheel(final String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
@Override  
public void accept(CarElementVisitor visitor) {  
    visitor.visit(this);  
}
```

“accept” method for  
Simple Object

```
public class Car implements CarElementVisitable {  
  
    private ArrayList<CarElementVisitable> elements =  
        new ArrayList<CarElementVisitable>();  
  
    private double cost = 6000;  
  
    public Car() {  
        this.elements.add( new Wheels() );  
        this.elements.add( new Body() );  
        this.elements.add( new Engine() );  
    }  
}
```

```
@Override  
public void accept(CarElementVisitor visitor) {  
  
    for (CarElementVisitable element : elements) {  
        element.accept(visitor);  
    }  
    visitor.visit(this);  
}
```

“accept” method for  
Composite Object

# Visitor Pattern: Example-1

```
public interface CarElementVisitor {  
  
    void visit(Body body);  
    void visit(BodyPart1 bodyPart1);  
    void visit(BodyPart2 bodyPart2);  
  
    void visit(Wheels wheels);  
    void visit(Wheel wheel);  
  
    void visit(Car car);  
    void visit(Engine engine);  
}
```

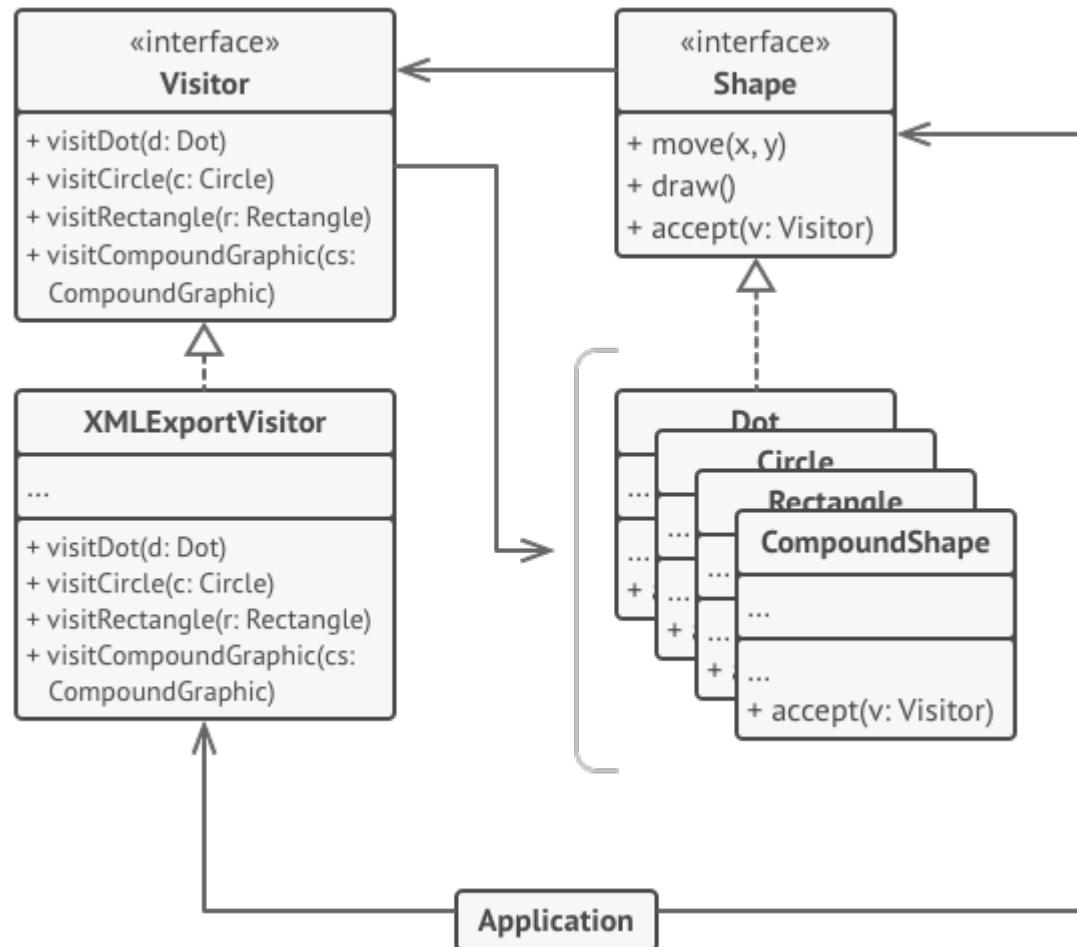
The visitor class need to define a set of visit methods, each of which could take arguments of different types. For example, "Body", "Engine", "Car", "Wheel", etc. Need to implement visit methods for each type.

```
public class CarElementPrintVisitor implements CarElementVisitor {  
  
    @Override  
    public void visit(Body body) {  
        System.out.println("Visiting body");  
    }  
  
    @Override  
    public void visit(Car car) {  
        System.out.println("Visiting car");  
    }  
  
    @Override  
    public void visit(Engine engine) {  
        System.out.println("Visiting engine");  
    }  
  
    @Override  
    public void visit(Wheel wheel) {  
        System.out.println("Visiting " + wheel.getName() + " wheel");  
    }  
  
    @Override  
    public void visit(BodyPart1 bodyPart1) {  
        System.out.println("Visiting bodyPart1");  
    }  
}
```

```
public static void main(final String[] args) {  
  
    Car car = new Car();  
  
    System.out.println("\n ----- From CarElementPrintVisitor -----");  
    car.accept(new CarElementPrintVisitor());  
}
```

Read the example code discussed in the lectures, and also provided for this week

## Visitor Pattern: Example-2



For more see: <https://refactoring.guru/design-patterns/visitor/java/example>

# Visitor Pattern: Applicability and Limitation

## Applicability:

Moving operations into visitor classes is beneficial when,

- ❖ many **unrelated operations** on an object structure are required,
- ❖ the classes that make up the object structure are known and **not expected to change**,
- ❖ **new** operations need to be added **frequently**,
- ❖ an **algorithm** involves several classes of the object structure, but it is desired to manage it **in one single location**,
- ❖ an **algorithm** needs to work **across** several independent class hierarchies.

## Limitation:

- ❖ extensions to the class hierarchy more difficult,  
as a **new class** typically require a **new visit method** to be added to each visitor.

End

# COMP2511

## Course Review Exam Structure

Prepared by

Dr. Ashesh Mahidadia

# Course Review

# Object Oriented Programming in Java: Introduction

- Abstraction
- Encapsulation
- Inheritance (single vs multiple)
- Polymorphism
- Objects, Classes, Interfaces
- Method Forwarding
- Method Overriding
- Generics
- Exceptions
  
- Domain Modeling

# Object Oriented Design : Principles

- Encapsulate what varies
- Favour composition over inheritance
- Program to an interface, not an implementation
- Principle of least knowledge (Law of Demeter)
- Liskov's Substitution Principle
- Classes should be (OCP) open for extension and closed for modification
- Avoid multiple/diverse responsibilities for a class
- Strive for loosely coupled designs between objects that interact

# Code Smells and Refactoring

- ❖ **Smells**: design aspects that violate fundamental design principles and impact software quality
- ❖ **Design Smells vs Code Smells**
- ❖ **Code smells** are usually not bugs; they are not technically incorrect and do not prevent the program from functioning.
- ❖ They indicate **weaknesses** in design that may slow down development or increase the risk of bugs or failures in the future.
- ❖ Regardless of the granularity, smells in general indicate violation of software design principles, and eventually lead to code that is rigid, fragile and require “**refactoring**”
- ❖ **Code refactoring** is the process of **restructuring** existing computer code **without changing** its external **behavior**.

# Design Patterns

## ❖ Creational Patterns

- ❖ Abstract Factory
- ❖ Factory Method
- ❖ Builder
- ❖ Singleton

## ❖ Behavioral Patterns

- ❖ Iterator
- ❖ Observer
- ❖ State
- ❖ Strategy
- ❖ Template
- ❖ Visitor

## ❖ Structural Patterns

- ❖ Adapter
- ❖ Composite
- ❖ Decorator

# Design Patterns: Creational Patterns

**Creational design** patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Four well-known creational design patterns:

- ❖ **Factory method pattern:** allows a class to defer instantiation to subclasses.
- ❖ **Abstract factory pattern:** provides an interface for creating related or dependent objects without specifying the objects' concrete classes.
- ❖ **Singleton pattern:** ensures that a class only has one instance, and provides a global point of access to it.

# Design Patterns: Structural Patterns

Structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships among entities.

Three well-known structural design patterns:

- ❖ **Adapter pattern:** '*adapts*' one interface for a class into one that a client expects
- ❖ **Composite pattern:** a tree structure of objects where every object has the **same interface** (leaf and composite nodes)
- ❖ **Decorator pattern:** add additional functionality to a class at **runtime** where subclassing would result in an exponential rise of new classes.

from the corresponding wikipedia page.

# Design Patterns: Behavioral Patterns

**Behavioral design patterns** are design patterns that identify common communication patterns among objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Six well-known structural design patterns:

- ❖ **Iterator pattern:** Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation.
- ❖ **Observer pattern:** Objects register to observe an event that may be raised by another object. Also known as Publish/Subscribe or Event Listener.
- ❖ **Strategy pattern:** Algorithms can be selected at runtime, using composition.
- ❖ **State pattern:** A clean way for an object to partially change its type at runtime.
- ❖ **Template method pattern:** Describes the program skeleton of a program; algorithms can be selected at runtime, using inheritance.
- ❖ **Visitor pattern:** A way to separate an algorithm from an object.

from the corresponding wikipedia page.

# Other Topics

We also briefly discussed the following topics:

- ❖ Event-Driven Programming
- ❖ Asynchronous Design
- ❖ Introduction to Microservices

# Exam Structure

# Final Exam : Structure

Three parts,

- ❖ Part 1: **Multiple Choice questions** (25 marks)
- ❖ Part 2: **Short Answer questions** (25 marks)
- ❖ Part 3: **Design and Programming Questions** (50 marks)

# Final Exam Information

- The Sample Final Exam will be available in the exam environment during the tutorial/lab period in Week 10. Please make sure to attend the Week 10 tutorial/lab.
- See [Exam Info](#) , available on the [Lecture Schedule / Recordings](#) web page, under week-10 material.

## Evaluation

- ❖ *myExperience* feedback is available via myUNSW.
- ❖ Tell us what you like/dislike about the course, we do take your input seriously.
- ❖ Thanks ...

And Finally ... ...

That's All Folks

Good Luck with the Exams

and with your future computing studies



**End**