Design Patterns

• What is design patterns?
  – The design patterns are language-independent strategies for solving common object-oriented design problems.

Outline

• Commonly recurring patterns of OO design
  • A simple example
  • Adapter pattern
  • Composite pattern
  • Solving complex problems

Commonly recurring patterns of OO design

• Creational Patterns
• Structural patterns
• Behavioural patterns
• J2EE patterns
• ...

  o Adaptor
  o Composite
  o Strategy
  o Decorator
  o Observer
  o Factory method
  o Iterator

Bicycles Simulator

• Various bicycles
  – Mountain bicycles
  – Road bicycles
  – Child bicycles …

• Various actions
  – Speed up
  – Changing gear …
Bicycle’s Inheritance

- Many different bicycles
- Classic case for inheritance
- Base-class of Bicycle, from which specialized Bicycle classes inherit

Bicycle class

- UML Representation for a Class
- UML stands for Unified Modeling Language

Bicycle class Inheritance

- Abstract classes: cannot be instantiated.
- Abstract methods: without an implementation.

Bicycle class

```java
// source file name bicycle.java
public abstract class bicycle{
private int gear;
private boolean front_brake;
private boolean rear_brake;
public void setGear(int newValue);
public void getMaximumSpeed();
...}
public class road_bicycle extends bicycle{
...
}
public class mountain_bicycle extends bicycle{
...
}
```
Inheritance summary

- A class inherits fields and methods from all its superclasses, whether direct or indirect. A subclass can override methods that it inherits, or it can hide fields or methods that it inherits.
- An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods.
- You can prevent a class from being subclassed by using the final keyword in the class's declaration. Similarly, you can prevent a method from being overridden by declaring it as a final method.

Bicycles Simulator v2

- More features of bicycles
  - Foldable
  - With motor

Bicycles Simulator v2

```java
public abstract class bicycle{
    ...
    public void setGear(int newValue);
    public void setFold(boolean newState );
    public void setMotor(double newSpeed);
    ...
}
```

Bicycles Simulator v2

- Some features are not appropriate for all subclasses of Bicycle.
- If those features are implemented in the methods of superclass, subclasses inherit inappropriate behaviour.
- Need to override the methods.
- Is Inheritance the right model here?

Bicycle Inheritance

- Disabling operations on derived classes
- Maintenance problem
  - Every new bicycle subclass will require you to consider overriding setFold()/setMotor()
  - Danger of not overriding methods on new bicycles...

Interface

- Does the behaviour belong in the baseclass in the first place?
- Put it in an interface
- Interfaces are a collection of abstract methods that an Object implements
- Object can support multiple interfaces (no multiple inheritance!)
Object Interface

- Client can ask an object if it supports an interface.
- If it does it can use the interface, if not try something else.
- Interfaces are declarations, they contain no code to implement methods.

Usage of interfaces

```java
interface foldable {
    void setFold(boolean newState);
}
interface with_motor {
    void setMotor(double newSpeed);
}
class myBike1 extends bicycle implements foldable {
    public void setFold(boolean newState) { … }
    …
}
class myBike2 extends bicycle implements foldable, with_motor {
    …
}
```

Bicycle class hierarchy

Duplicate Code

- Interfaces have no code
- Each bicycle subclass must have its own implementation for each interface.
- Any change to an implementation must be made to every sub-class that supports it

Adapter

- The intent of Adapter is to provide the interface that a client expects while using a existing class.
- A new subclass of Bicycle: Bicycle_without_pedals
  ```java
  interface no_pedal {
    void setNoPedal();
  }
class myBike1 extends bicycle implements no_pedal {
    …
  }
  ```

Adapter

- When a developer of client code thoughtfully defines the client’s needs, you may be able to fulfill the interface by adapting existing code.
Object Adapter

- ExistingClass is instantiated in NewClass so that UsefulMethod can be used. It can be dangerous if you don’t override all the methods that might be called.

Object Adapter example

```java
class SampleAdapter implements ClientClass {
  private AdapterClass mInstance;
  public SampleAdapter(AdapterClass instance) {
    mInstance = instance;
  }
  @Override
  public void ClientClassMethod() {
    // call AdapterClass's method to implement ClientClassMethod
  }
}
```

Adapter summary

- The Adapter pattern lets you use an existing class to meet new requirements.
- Class adapter: create new class that implements new interface and subclasses an existing class.
- Object adapter: create a new client subclass that uses an instance of the existing class.

Change

- Change is the one constant in software development
- Don’t make rigid designs
- Changing something should be easy and require the least amount of effort

Design Principle

“Identify the aspects of your application that vary and separate them from what stays the same”
Changing Bicycles

- Thought we had a good design for our bicycle simulator
- But changing features may require major changes
- How do we make it a good design?

Bicycle (re)design

- Bicycle class seems to work
  - Apart from methods e.g. setFold()
- Separate these methods off into new classes or interfaces
  - Foldable behaviour
  - With Motor behaviour
  - Without Pedal behaviour
  - ...

Bicycle Behaviour

- Keep things flexible
- Instances of Bicycle are assigned behaviours
- Can we change a behaviour dynamically?
  - To make a foldable bicycle non-foldable?
- Mutators to change Bicycle’s behaviour at runtime

Design Principle

“Program to an interface, not an implementation”

Program to an interface

Interface A { method();...;}
Class B implement A { method();...;};

// program 1:
Class X {
  private A a1;
  public void useA (A b1){
    a1 = b1;
    a1.method();
  }
}

// program 2:
Class X {
  private B b1;
  public void useB(){
    b1.method();
  }
}

Behaviour Interfaces

- FoldableBehaviour and WithMotorBehaviour
- Gives us flexibility
  - Don’t inherit any implementation
  - Bicycle’s aren’t tied to an implementation
Integrating Behaviours

- Interface type — don’t want to bind this to an implementation
- Add instance variables for the behaviours
  - FoldableBehaviour
  - WithMotorBehaviour
  - ...
- Add methods to call the behaviour

What is happening?

- When myBike is created:
  - Creates objects for correct behaviours
  - Sets variables to instantiate existing classes (interfaces)
- When setFoldable() called, setFold() is called on the newly created object
- Equivalent for setMotor()...

Strategy Pattern

- Example of the ‘Strategy’ Pattern
- Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it

Exercise

```java
interface Monster { void menace();}
interface DangerousMonster extends Monster { void destroy();}
interface Lethal { void kill();}

class Zombie implements DangerousMonster {
    public void menace() {} // some code here
    public void destroy() {} // some code here
}

class Vampire extends DangerousMonster, Lethal {
    public void menace() {} // some code here
    public void destroy() {} // some code here
    public void kill() {} // some code here
    public void drinkBlood() {} // some code here
}
```

** You need create three new classes:
  - Monster1: is a zombie and has kill() method
  - Monster2: has the methods of kill(), destroy(), and a new method fly()
  - Monster3: is a very bad vampire but does not have kill() method
Reading

• Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang Of Four), Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional, 1994

• Steven J. Metsker and William C. Wake, Design Patterns in Java, Addison Wesley, 2006

• http://www.javacamp.org/designPattern/