UNIVERSITY OF PLYMOUTH

COMP2000: Software engineering 2

Session 8: Design patterns and SOLID principles

Outline

- Design patterns – Overview
- SOLID principles – Overview

Design Patterns in Java

- A design patterns are **well-proved solution** for solving the specific problem/task.

- Design patterns are programming language independent strategies for solving the common object-oriented design problems.

- That means, a design pattern represents an idea, not a particular implementation.

- As a developer, using design patterns, can make your code more flexible, reusable and maintainable.

- It is the most important part because java internally follows design patterns.

- **Design patterns are categorised into:**

- **Creational patterns**: How you *create* objects.
- **Structural patterns**: How you *compose* objects.
- **Behavioural patterns**: How you *coordinate* object interactions.

Creational Patterns	Structural Patterns	Behavioral Patterns
•Builder	•Adapter	•Command
•Dependency Injection	•Facade	•Observer
•Singleton	•Decorator	•Strategy
•Factory	•Composite	•State

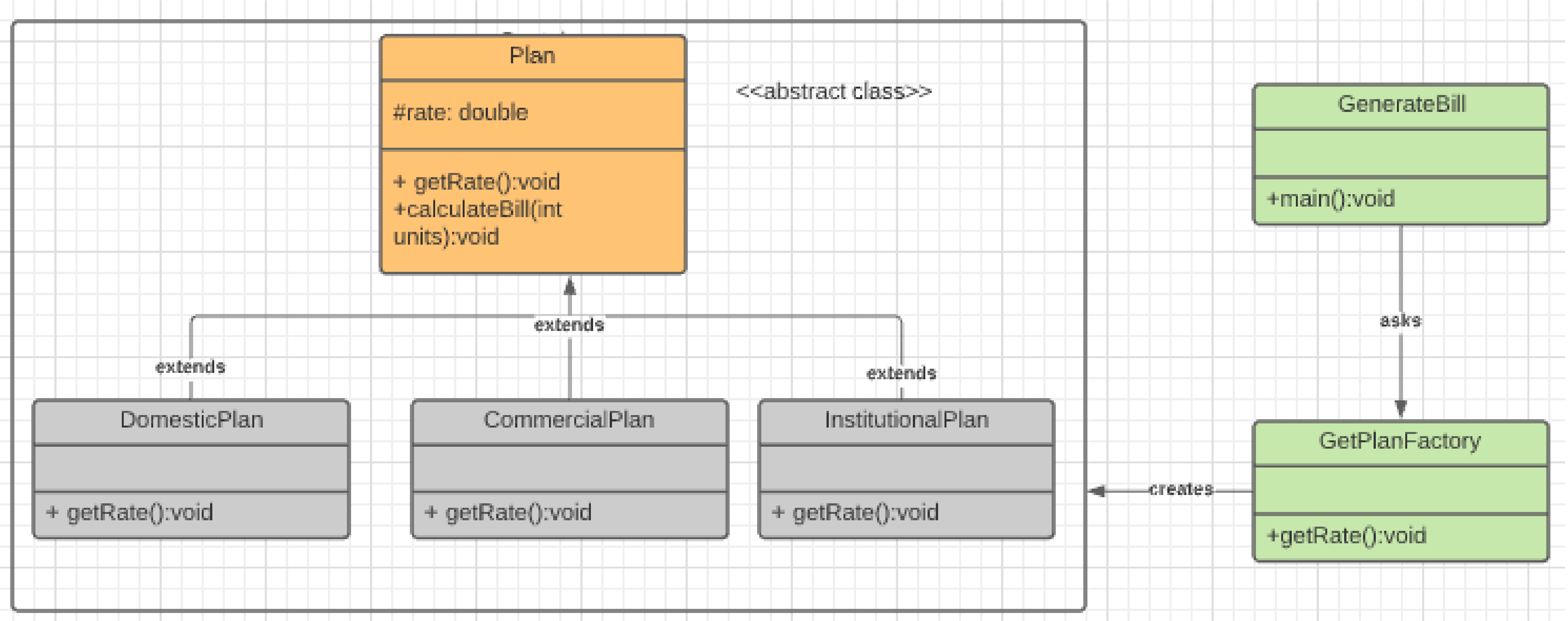
Creational Patterns:

- **Factory method**
- A Factory Pattern or Factory Method Pattern is to **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.**

- In other words, subclasses are responsible to create the instance of the class.

- The Factory Method Pattern is also known as **Virtual Constructor.**

Example


Plan
#rate: double
+ getRate():void
+calculateBill(int units):void
<<abstract class>>
extends
extends
extends
DomesticPlan
+ getRate():void
CommercialPlan
+ getRate():void
InstitutionalPlan
+ getRate():void
GenerateBill
+main():void
asks
GetPlanFactory
+getRate():void
creates

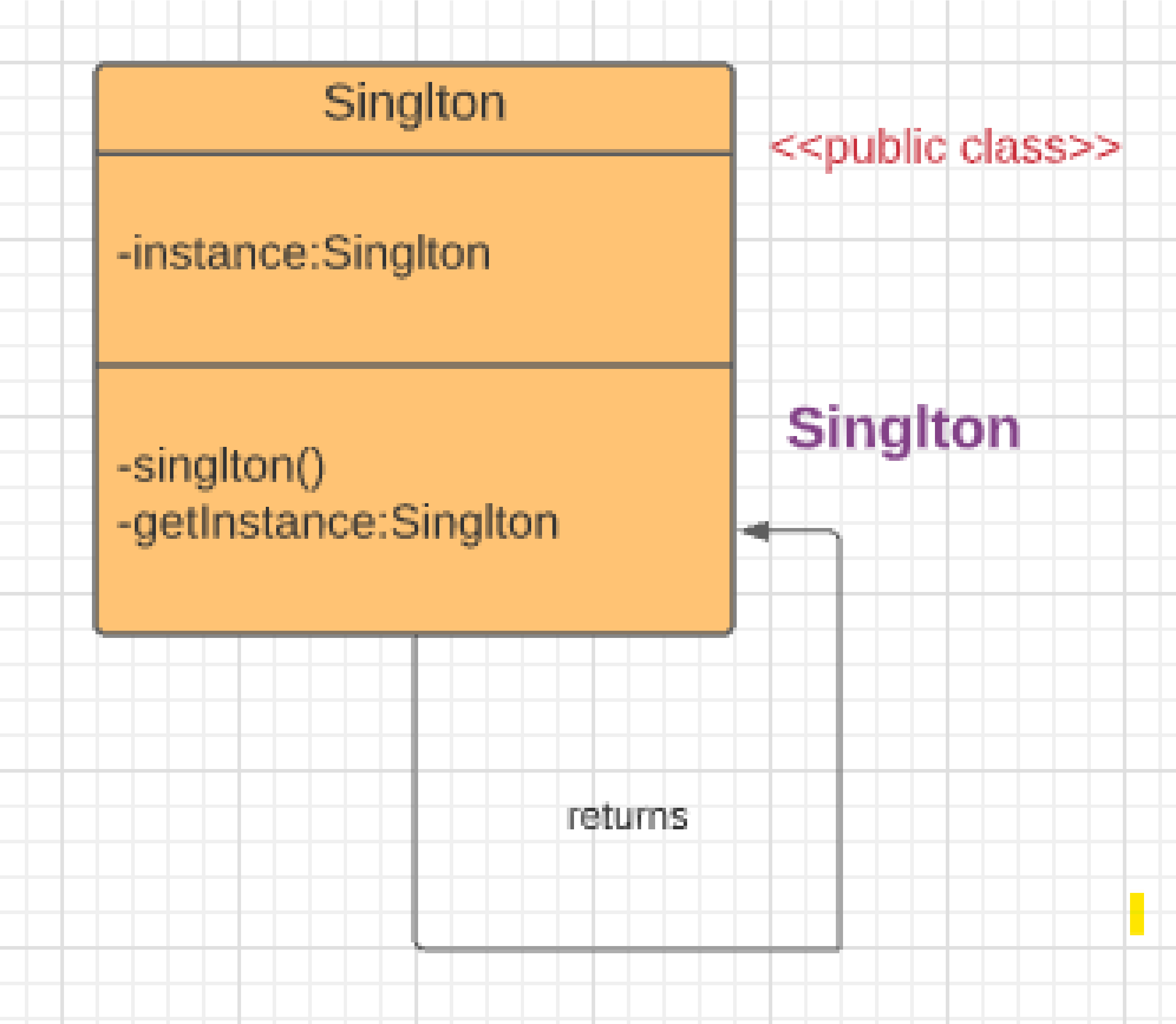
Singleton design pattern

- Singleton Pattern is to “**define a class that has only one instance and provides a global point of access to it**".

- In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

- There are two forms of singleton design pattern
- **Early Instantiation**: *creation of instance at load time*.
- **Lazy Instantiation**: *creation of instance when required*.

- Singleton pattern is mostly used in multi-threaded and database applications.
- It is used in logging, caching, thread pools, configuration settings etc.
-


Singlton
-instance:Singlton
-singlton()
-getInstance:Singlton
<<public class>>
Singlton
returns

Instantiation of Singleton Pattern

```
class A{
private static A obj=new A();//Early, instance will be created at load time
 private A() {

}
    public static A getA(){
     return obj;
    }

  public void doSomething(){
   //write your code
  }
}
```

lazy Instantiation of Singleton Pattern

```
class A{
 private static A obj;
 private A(){}
   public static A getA(){
      if (obj == null){
      synchronized(Singleton.class){
          if (obj == null){
         obj = new Singleton();     //instance will be created at request time
      }
   }
   }  return obj;
 }
  public void doSomething(){
   //write your code
 }  }
```

```
private MySingleton(Context context) {
    ctx = context;
    requestQueue = getRequestQueue();

        }
```

```java
public static synchronized MySingleton getInstance(Context context) {
    if (instance == null) {
        instance = new MySingleton(context);
    }
    return instance;
}
```

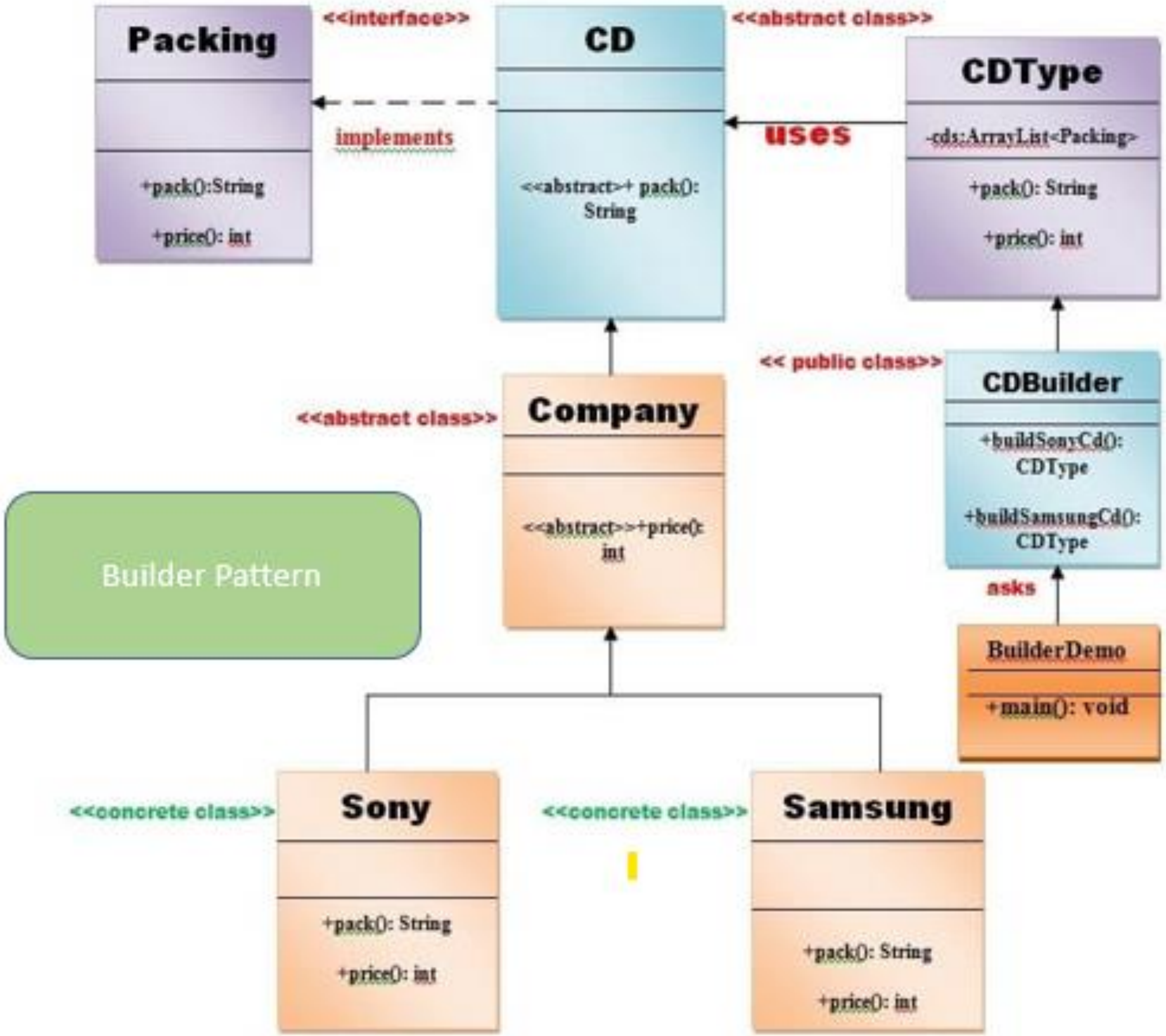
```java
public RequestQueue getRequestQueue() {
    if (requestQueue == null) {
requestQueue = Volley.newRequestQueue(ctx.getApplicationContext());
    }
    return requestQueue;
}
```

```
public <T> void addToRequestQueue(Request<T> req) {
    getRequestQueue().add(req);
}
```

Builder Design Pattern

- Builder Pattern is to "**construct a complex object from simple objects using step-by-step approach**“

- It is mostly used when object can't be created in single step like in the de-serialization of a complex object.



https://www.javatpoint.com/builder-design-pattern

Example

```
// Instantiate an Alert Dialog with its constructor
        builder = new AlertDialog.Builder(this);

      // Chain together various setter methods to set the dialog characteristics
        builder.setMessage("This is a dialog Alert")
               .setTitle("Dialog Alert");

//  Get the AlertDialog
        AlertDialog dialog = builder.create();
```

Structural Patterns

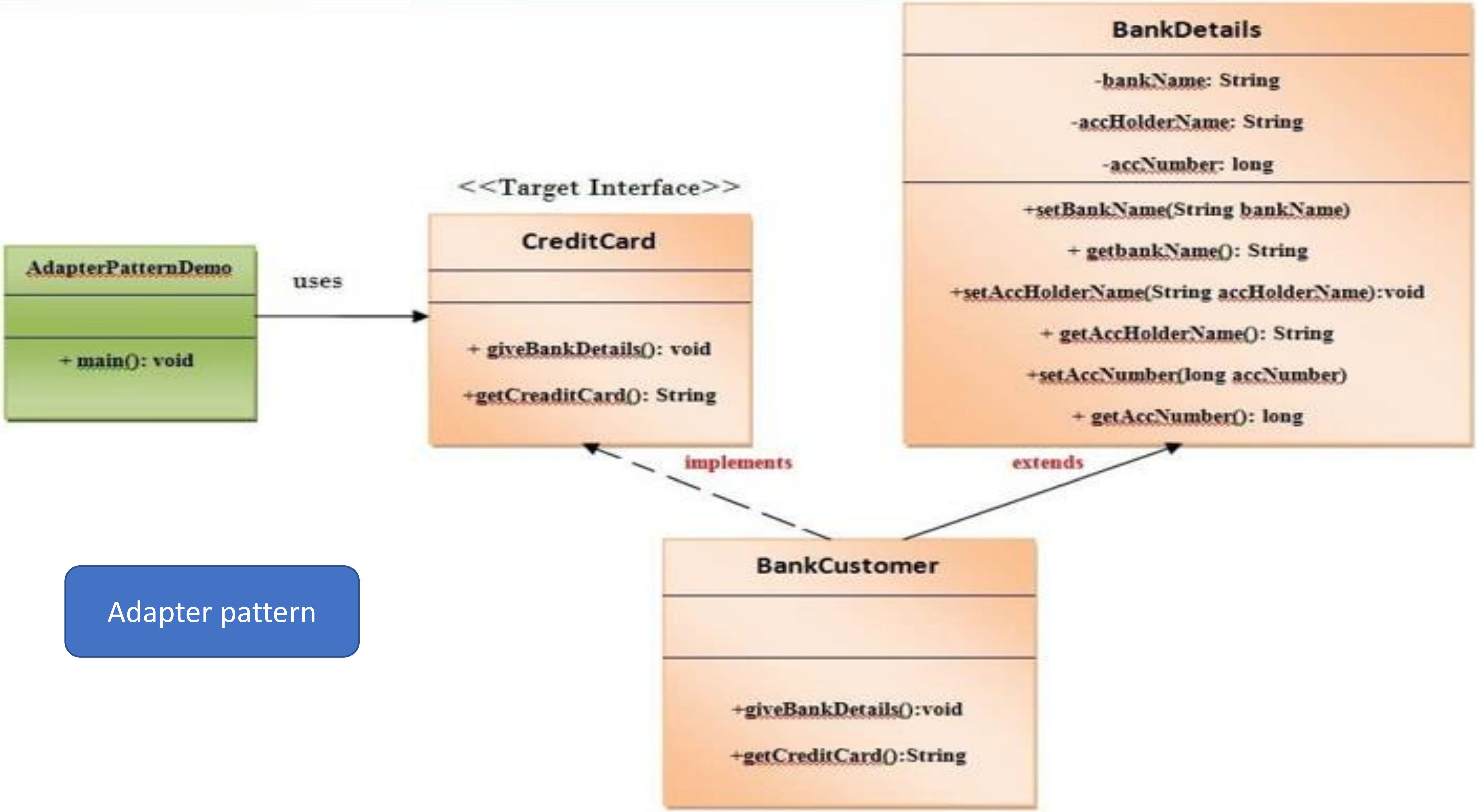
- Adapter Pattern

- An Adapter Pattern is to "**converts the interface of a class into another interface that a client wants**".

- This pattern lets two incompatible classes work together by converting a class's interface into the interface the client expects.

- The Adapter Pattern is also known as **Wrapper.**


dreamstime



Adapter pattern

https://www.javatpoint.com/adapter-pattern

Example – List View adapter

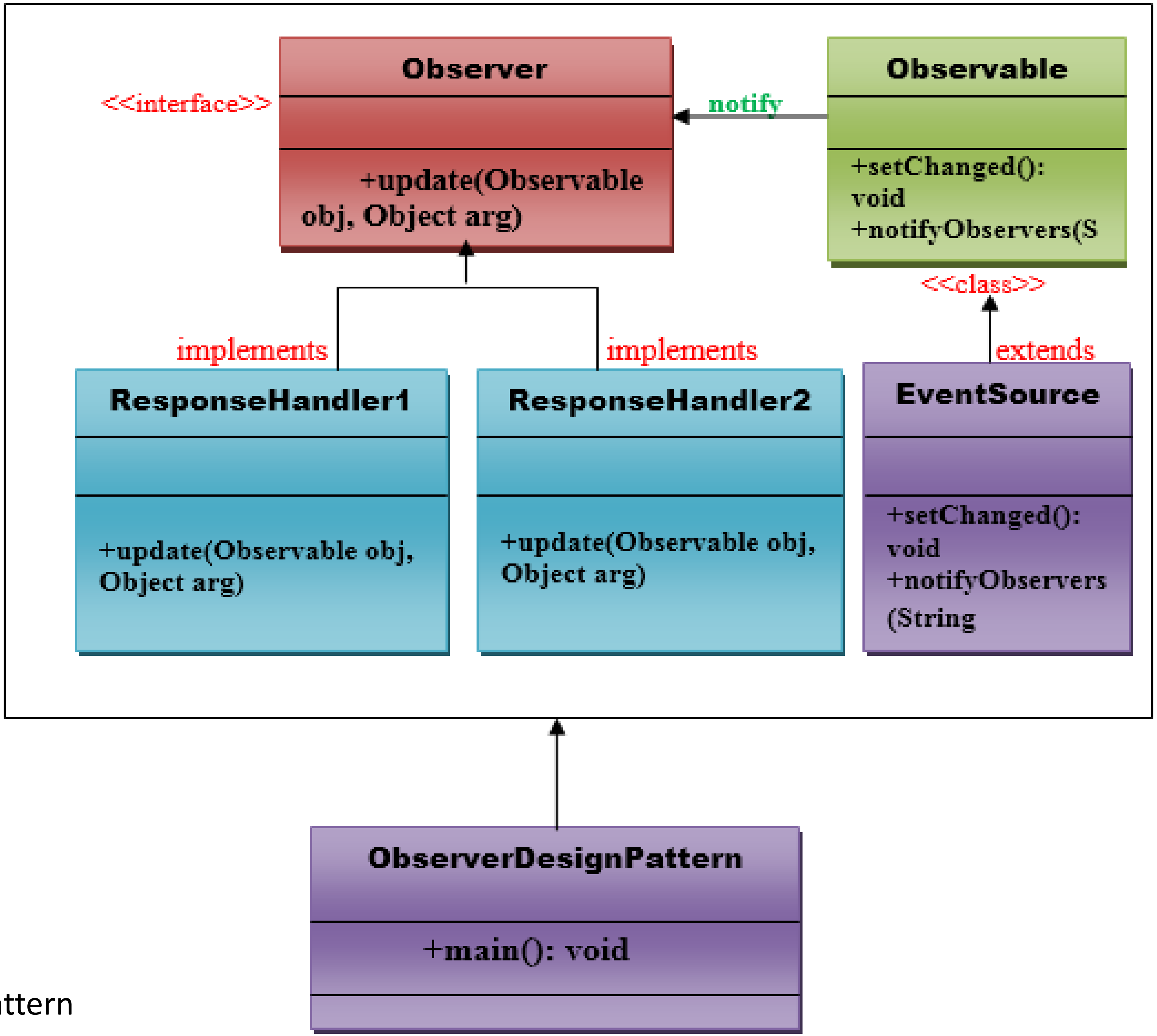
```
ArrayAdapter adapter = new ArrayAdapter<String>(this,
      R.layout.list_view, buildingArray);

ListView listView = (ListView) findViewById(R.id.buildings_list);
listView.setAdapter(adapter);
```

Behavioural Patterns: The Observer pattern

- The **Observer** pattern defines a one-to-many dependency between objects.
- When one object changes state, its dependents get a notification and updates automatically.

- you can use it for operations of indeterminate time, such as API calls. You can also use it to respond to user input.



https://www.javatpoint.com/observer-pattern

```java
public class EventSource extends Observable implements Runnable {
    @Override
    public void run() {
        try {
            final InputStreamReader isr = new InputStreamReader(System.in);
            final BufferedReader br = new BufferedReader(isr);
            while (true) {
                String response = br.readLine();
                setChanged();
                notifyObservers(response);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Activity

Problem Given:

Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

Solution:

- **Singleton design pattern** is the best solution of above specific problem.

The MVC design pattern

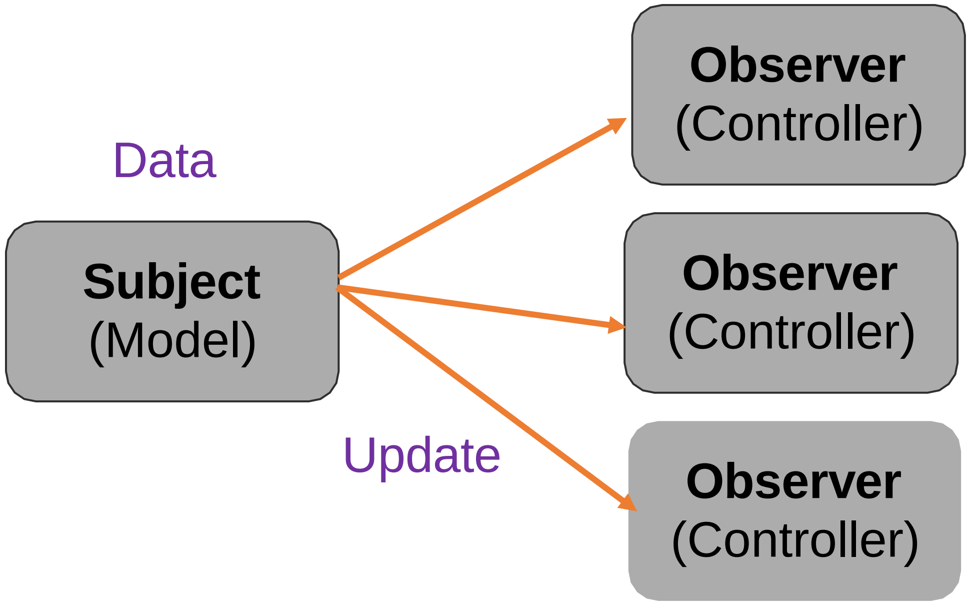
MVC is actually a combination of other patterns:

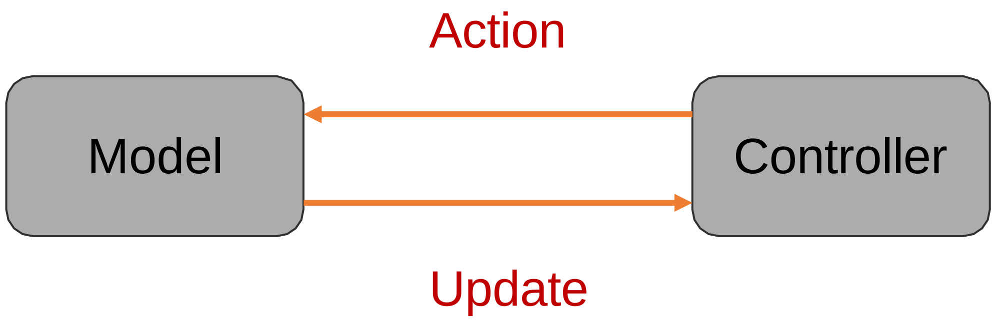
Model =	Observer	Behavioural pattern
View =	Composite pattern	Structural pattern
Controller =	Strategy pattern	Behavioural pattern

The model is the "Data Model" of the application

Our Model's data is accessed by the Controller to pass on to the View The

Observer pattern specialises in updating other objects with its data





Books on Design Patterns

Design Patterns: Element of Reusable Object-Oriented Software

- Original book for veteran coders

Head First Design Patterns

- Highly accessible to all levels

Both are free online in the Library!

Design Principles: SOLID

SOLID is a mnemonic acronym for five design principles intended to make software understandable, flexible, maintainable and extensible.

Using SOLID principles in Android development could be helpful and effective to follow clean code principles.

- SOLID principles are briefly as follows:
- Single Responsibility Principle (SRP): *A class should have one, and only one, reason to change: a single responsibility.*
- Open-Closed Principle (OCP): *A class should be open for extension, but closed for modification.*
- Liscov Substitution Principle (LSP): *Derived classes should be substitutable for their base classes.*
- Interface Segregation Principle (ISP): *Make fine grained interfaces that are client specific.*
- Dependency inversion Principle (DIP): *Depend on abstractions, not on concretions*

Single Responsibility Principle (SRP) Analysis

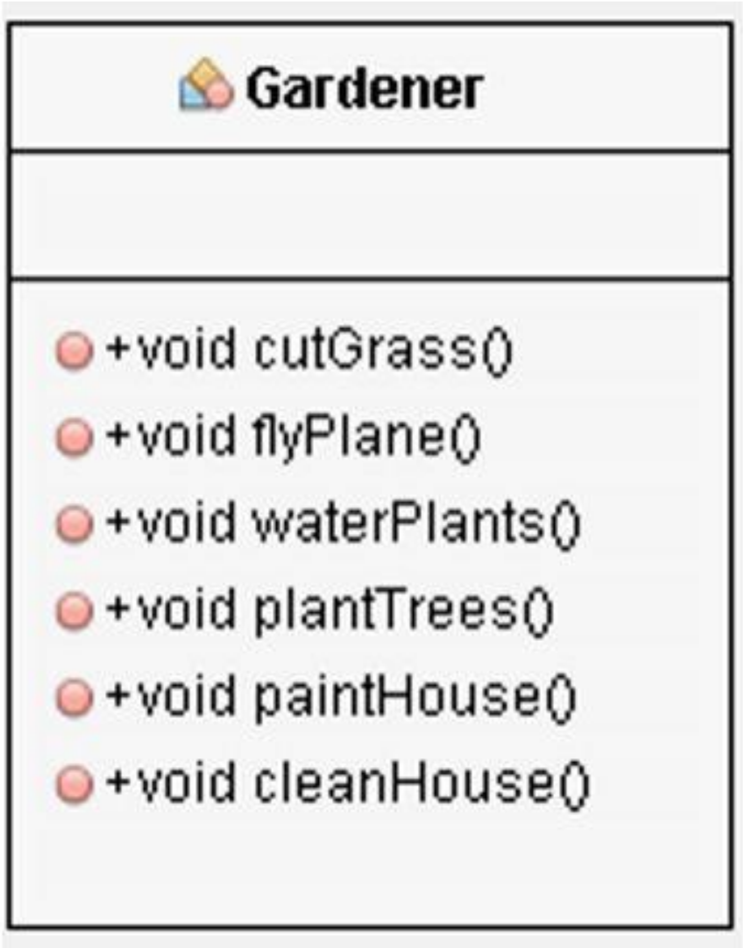
Check the statement: The______(class name)____(method name) itself.

- Ask the question: **Does it make sense** (within the problem context)?
- If not separate the method from the class, and use a sensible class to modularise the behaviour.

Performing SRP analysis usually leads to lose coupling and high cohesion.

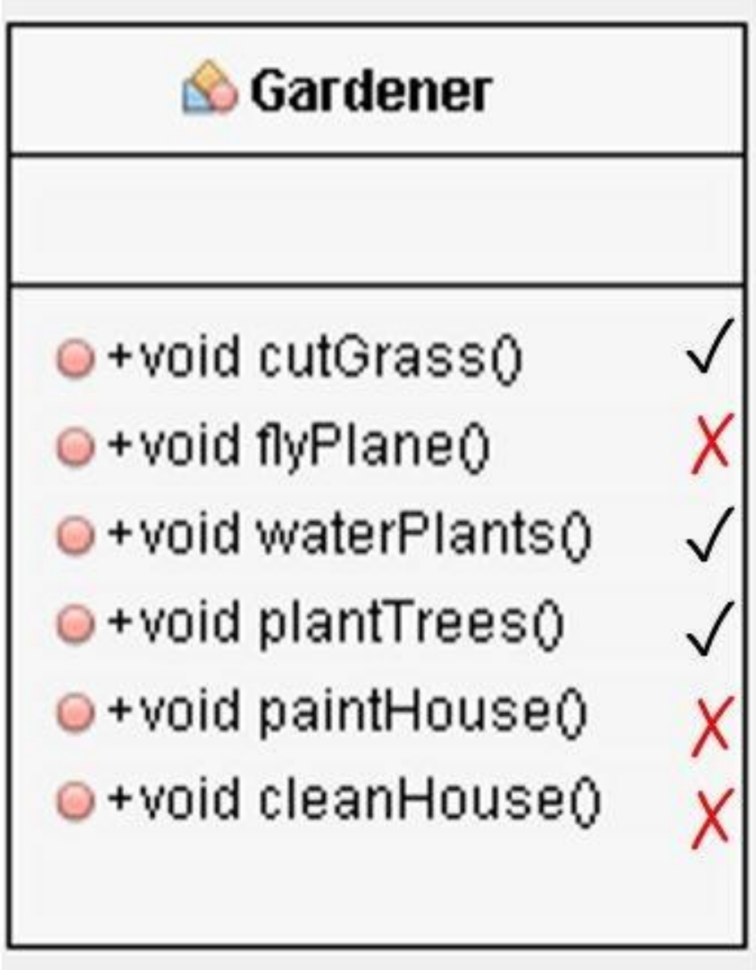
SRP Example

Perform SRP analysis on the following class.

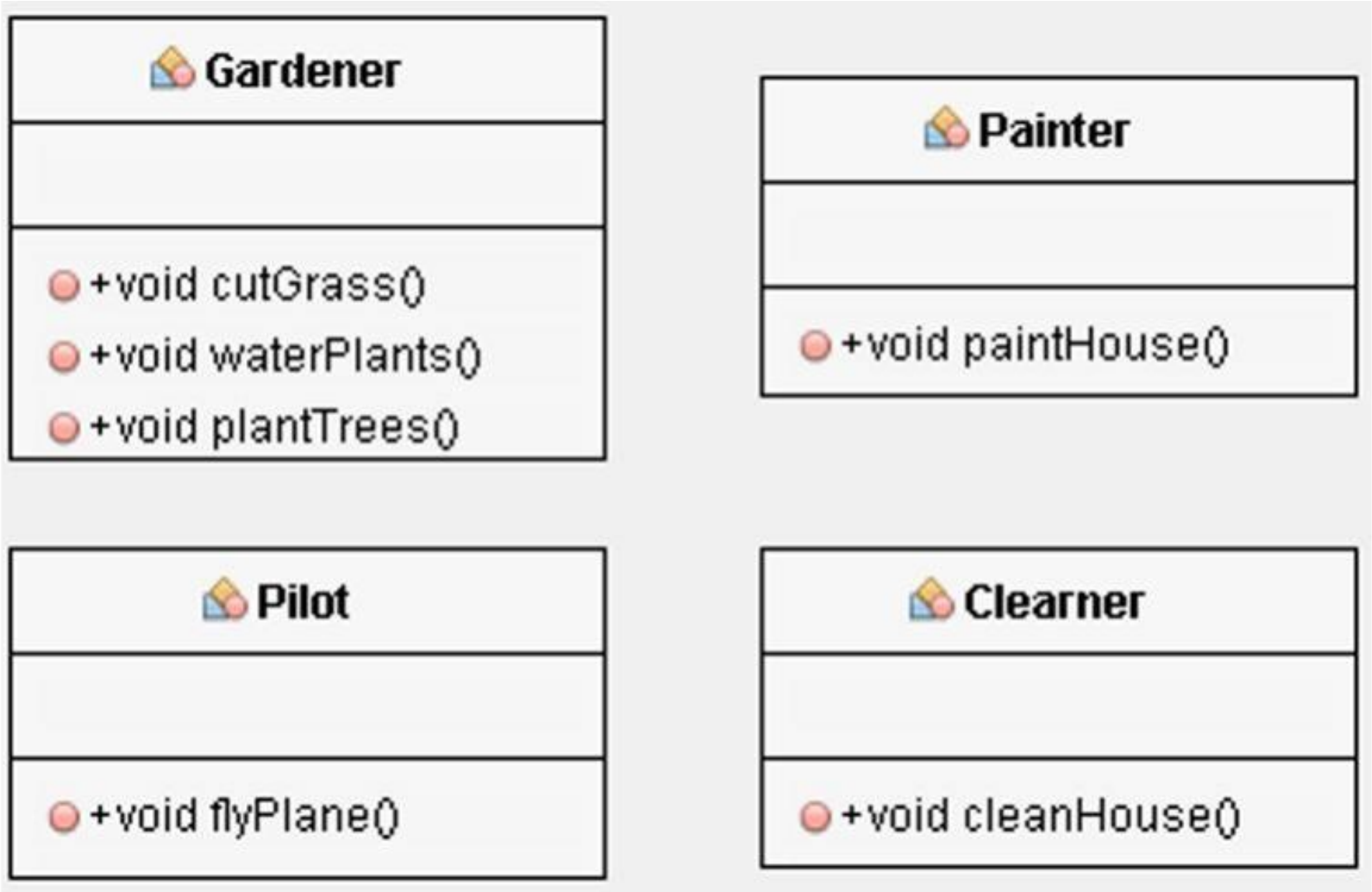


SRP Example

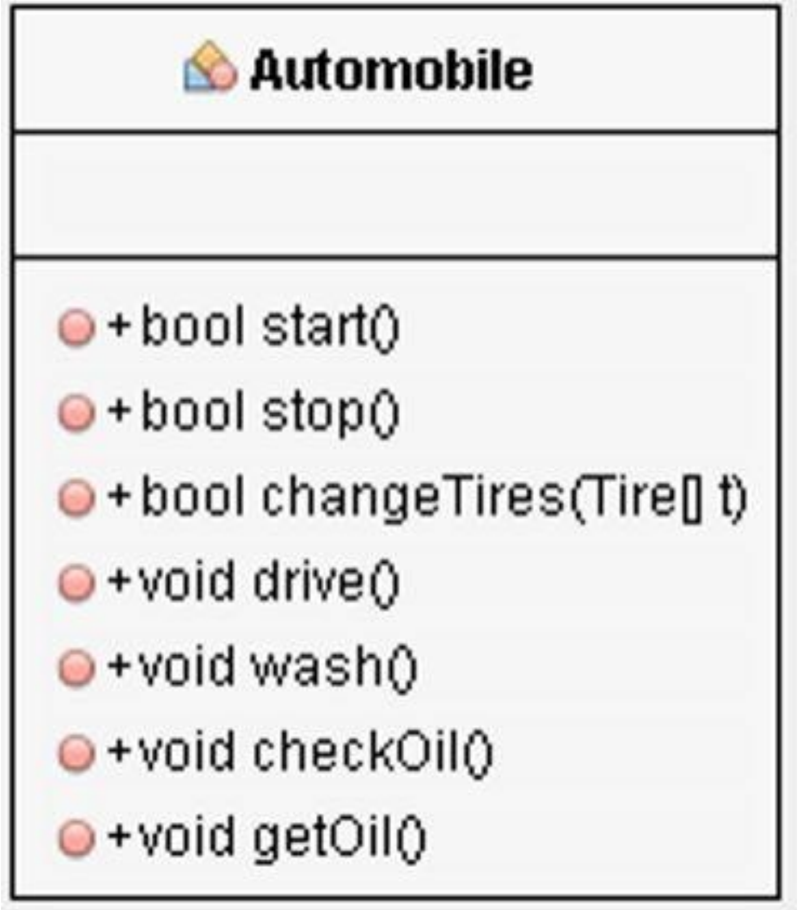
Perform SRP analysis on the following class.



SRP Example

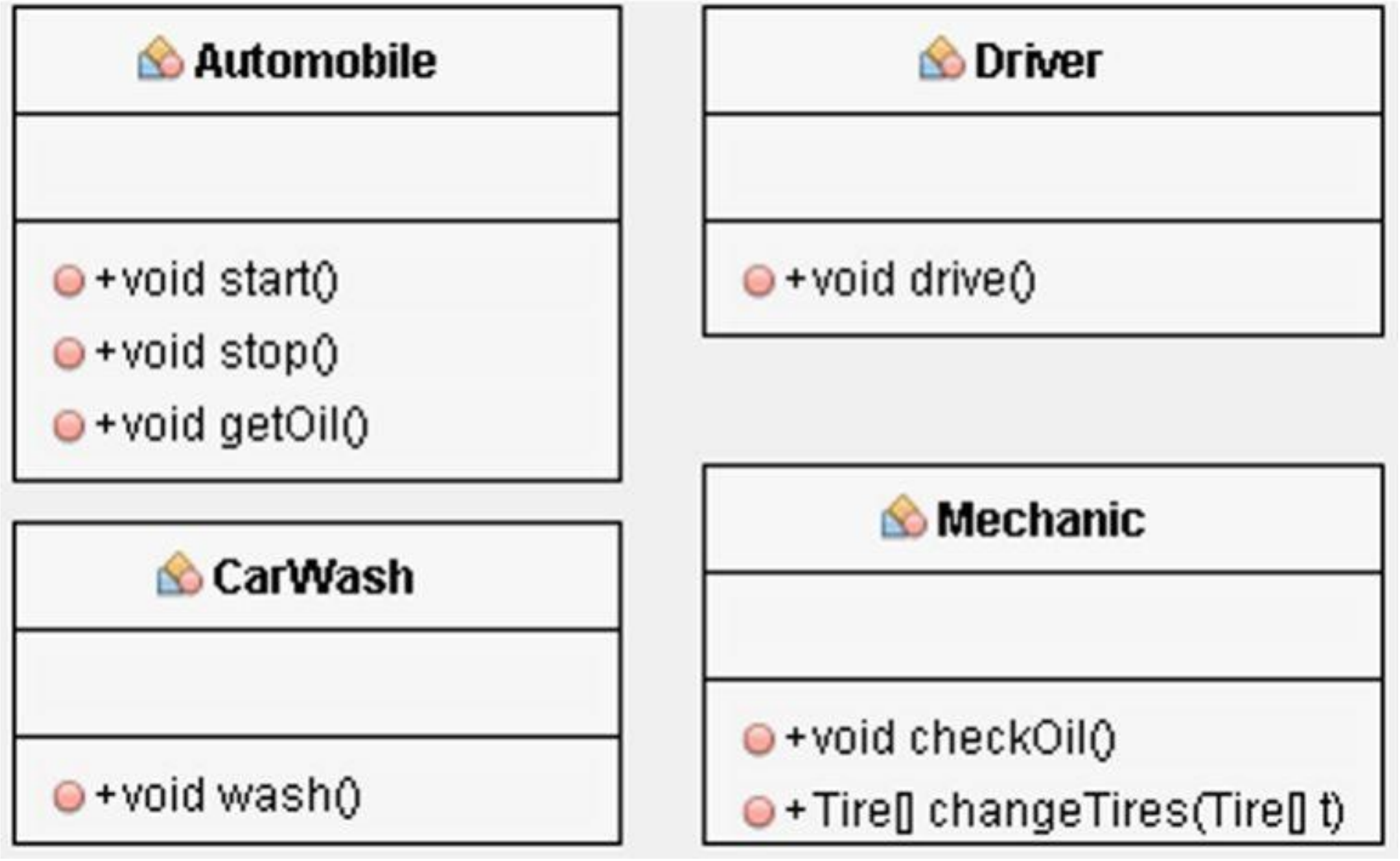

Gardener
+void cutGrass()
+void waterPlants()
+void plantTrees()
Painter
+void paintHouse()
Pilot
+void flyPlane()
Clearner
+void cleanHouse()

Perform SRP analysis on the following class.



SRP Example

An automobile certainly starts, stops and gets oil by itself, but it does not drive, wash, checks oil or changes tire by itself! We need sensible classes to capture these actions.



- Check-in code:
- **HR-HP-NW**

Open-Closed Principle (OCP)

Classes should be **open for extension** and **closed for modification**.

This is all about allowing change: but doing so without modifying existing code.

OCP Example

How would you add a drawer for another shape, e.g. a hexagon?

Modify the drawShape method: it would require changing existing code!

Editing the class violates OCP!

```
public class ShapeDrawer
{
    public void drawShape(Shape newShape)
    {
        if (newShape instanceof Circle)
        {
            drawCircle(newShape);
        }
        else
        {
            drawRectangle(newShape);
        }
    }
}
```

OCP Example

With **abstract** class and **inheritance**, we no longer need to change the **ShapeDrawer** class.

```
public abstract class Shape {
    abstract void draw();
}
public class Circle extends Shape {
    public void draw() {
        // draw circle
    }
}
public class ShapeDrawer {
    public void drawShape(Shape newShape) {
        newShape.draw();
    }
}
```

@Override annotation here!

Liskov Substitution Principle (LSP)

A subclass must be **substitutable** for its superclass.

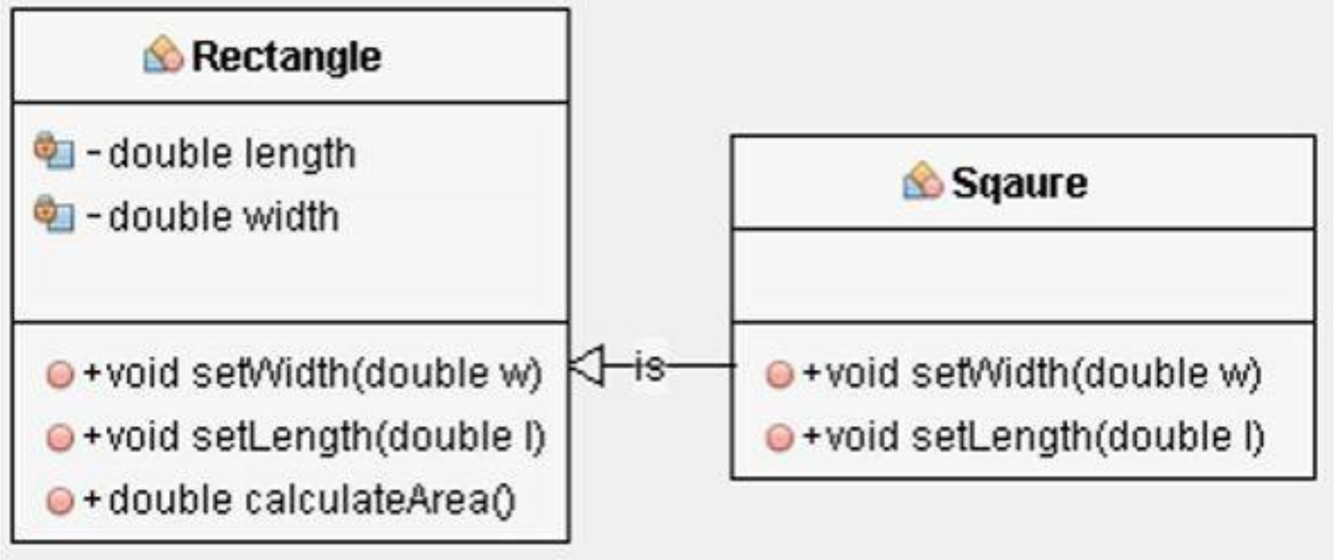
- LSP is all about **well-designed inheritance**.
- Whenever a superclass is used, it must be possible to **replace it with one of its subclasses** without breaking anything.
- Analyse inheritance: If I call a method from the subclass, would it perform as expected?

Prof. Barbara Liskov
PC: K. C. Zirkel

LSP Example

Perform LSP analysis on the following design.

```
Rectangle r = new Square();
r.setLength(3);
r.setWidth(4);
System.out.println(r.calculateArea());
```



Overridden methods setWidth and setLength both set length L and width W of the Square.

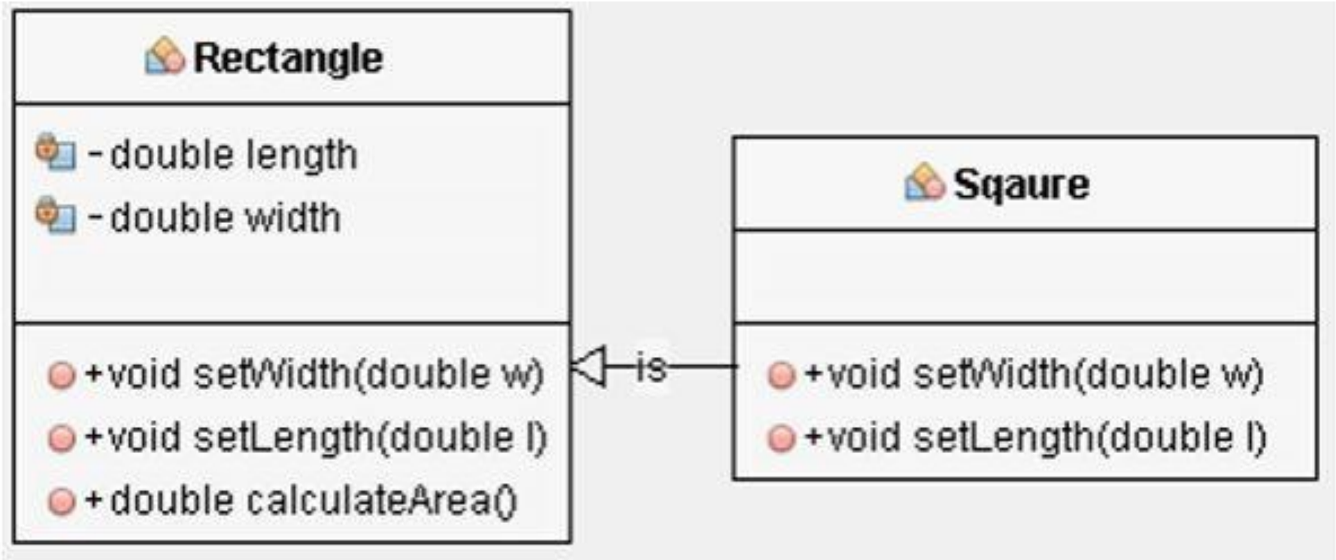
What result will I expect if I call calculateArea from a Square (in place of a Rectangle) after setting L = 3 and W = 4?

LSP Example

Perform LSP analysis on the following design.

```
Rectangle r = new Square();
r.setLength(3);
r.setWidth(4);
System.out.println(r.calculateArea());
```

Overridden methods setWidth and setLength both set length L and width W of the Square.



What result will I expect if I call calculateArea from a Square (in place of a Rectangle) after setting L = 3 and W = 4?

We get 16, rather than 12.
Makes sense in Square, but not so much in a Rectangle.

Reminder

- In addition to thinking about **IS-A relationship for inheritance**, be mindful about their **extrinsic behaviour**:

- A Square may be a special case of a Rectangle mathematically, but **their behaviours are not interchangeable**.

Interface Segregation Principle (ISP)

A class should not be forced to implement methods that it does not use.

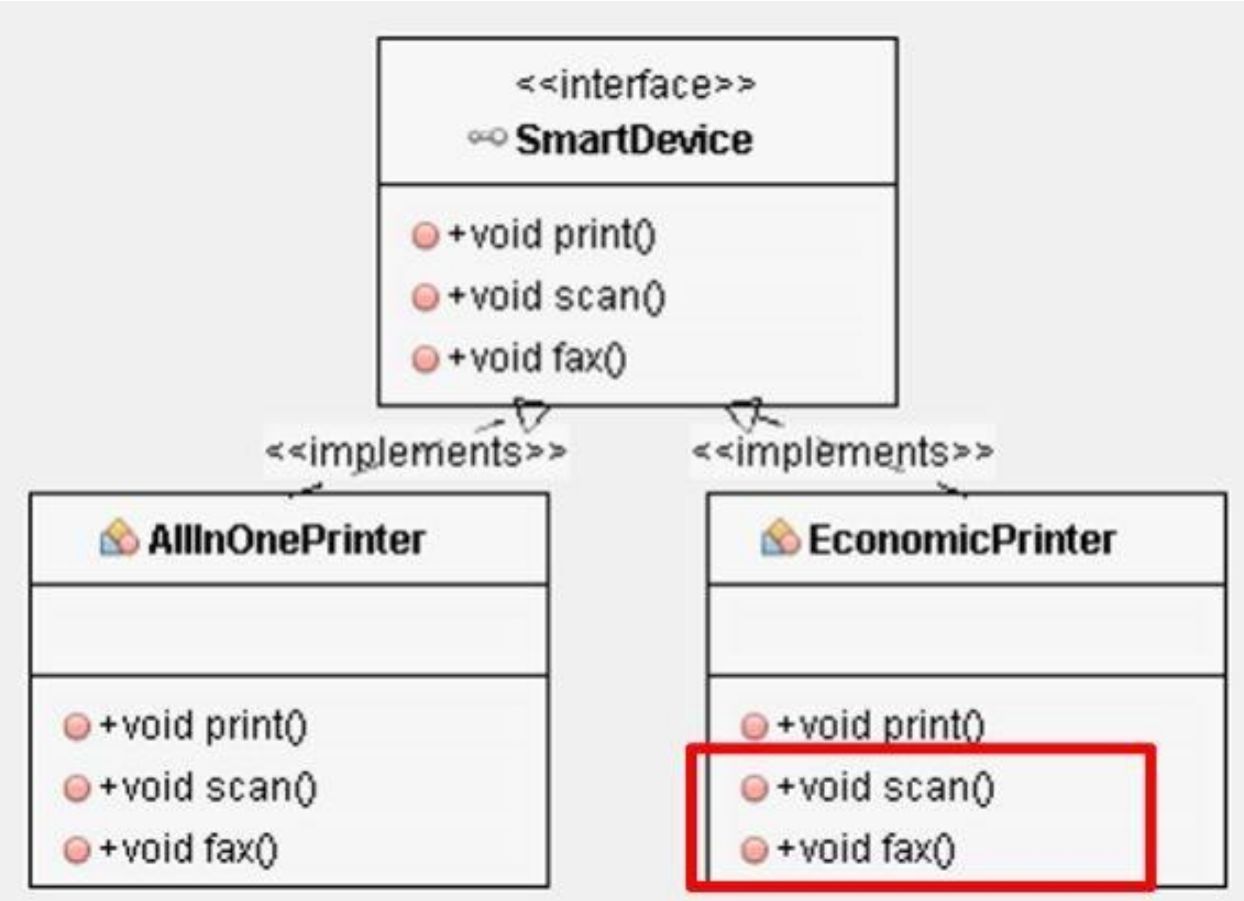
- Favour **many small interfaces**, rather than one mega-interface (a.k.a. ”fat” interface) consisting of all behaviour that you may ever need.

- Primary aim:
  - **Avoid unintended coupling** between classes implementing the interface.
- Analyse your interfaces:
  - Can I split the methods in **smaller logical groups**?

ISP Example

SmartDevice interface has lots of similar abstract methods: **print**, **scan** and **fax**.

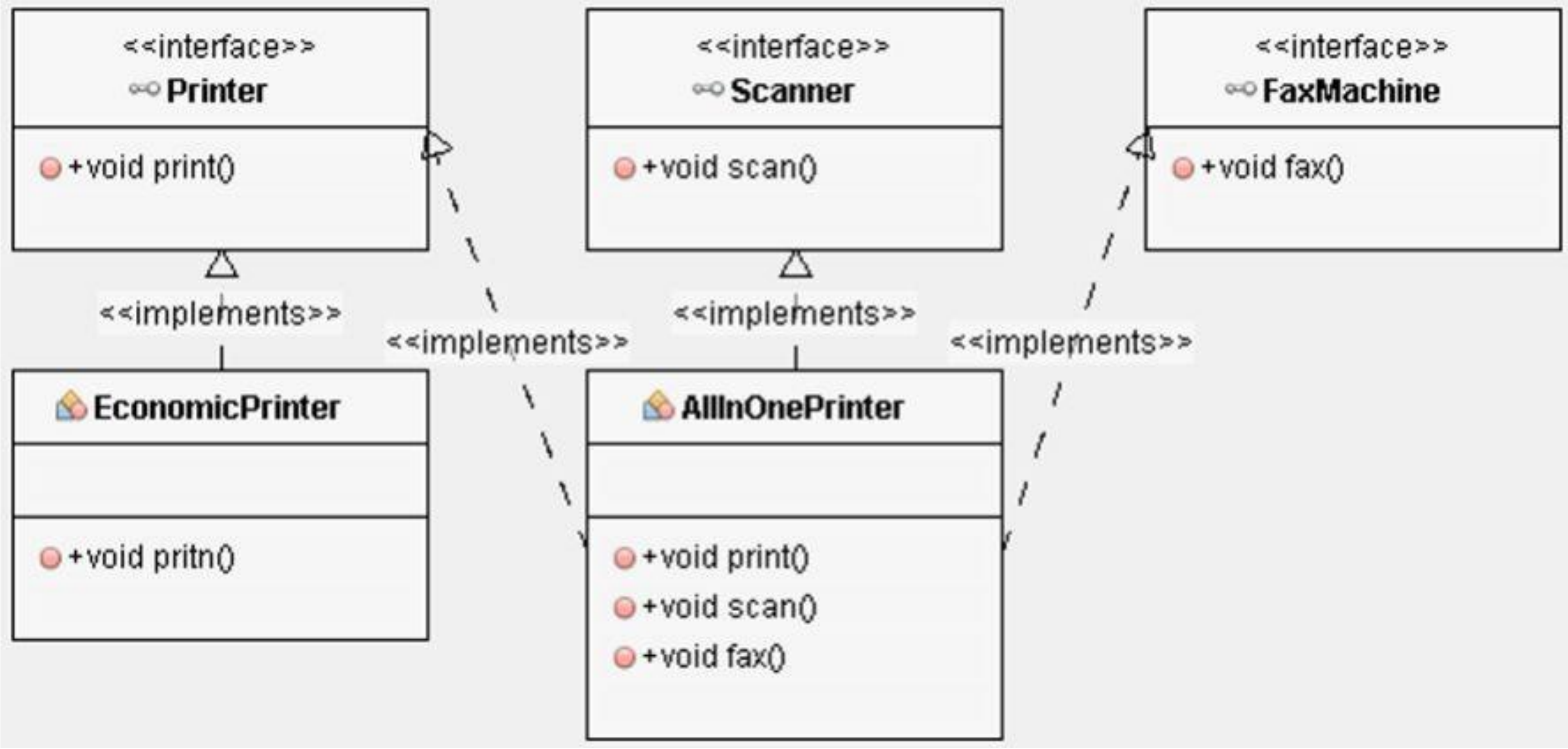
EconomicPrinter had to implement **scan** and **fax** methods, but it is never going to use these.



ISP Example

SmartDevice is split into three interfaces: **Printer**, **Scanner** and **FaxMachine**.

Both **EconomicPrinter** and **AllInOnePrinter** classes implement relevant methods.



Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details, details should depend on abstractions.

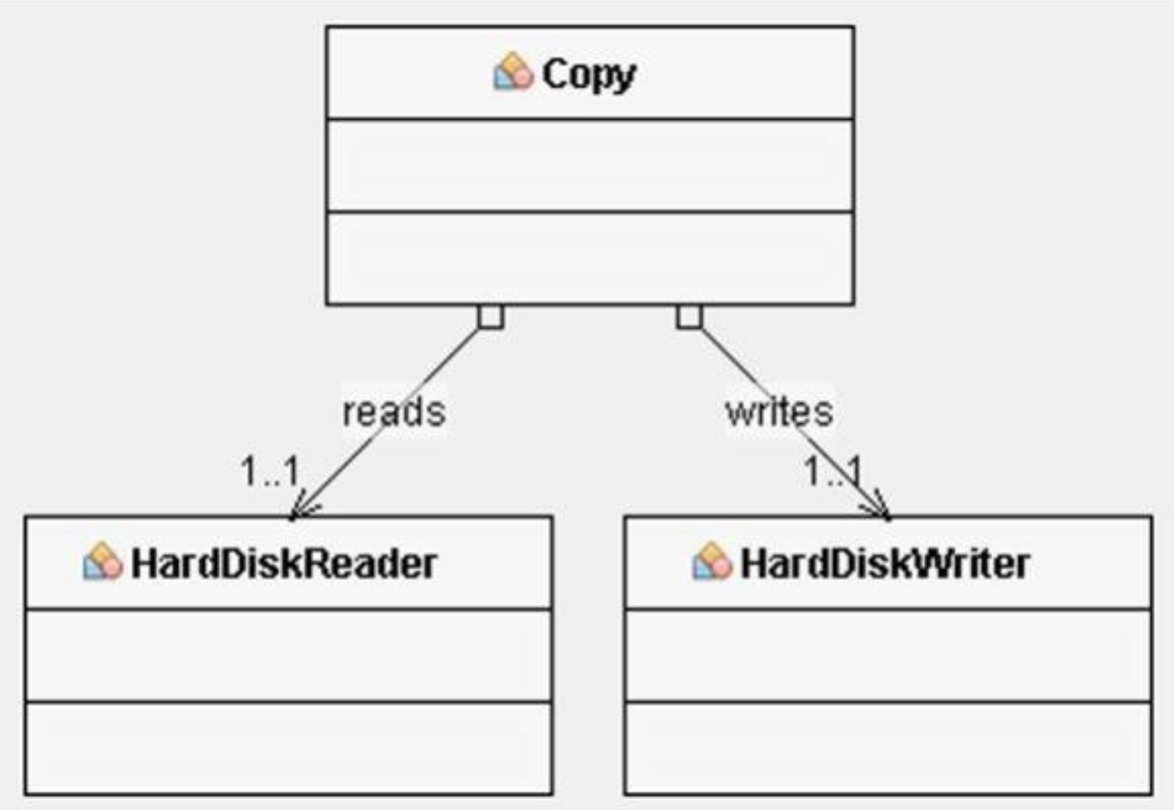
To avoid interdependency between parts of the code. Interdependency leads to:

- Rigidity - Code is hard to change because every change affects (too) many other parts of the system.
- Fragility - Unexpected parts break as a result of a change.
- Immobility - Parts of the code are hard to reuse in another application because they can’t be disentangled from the current system.

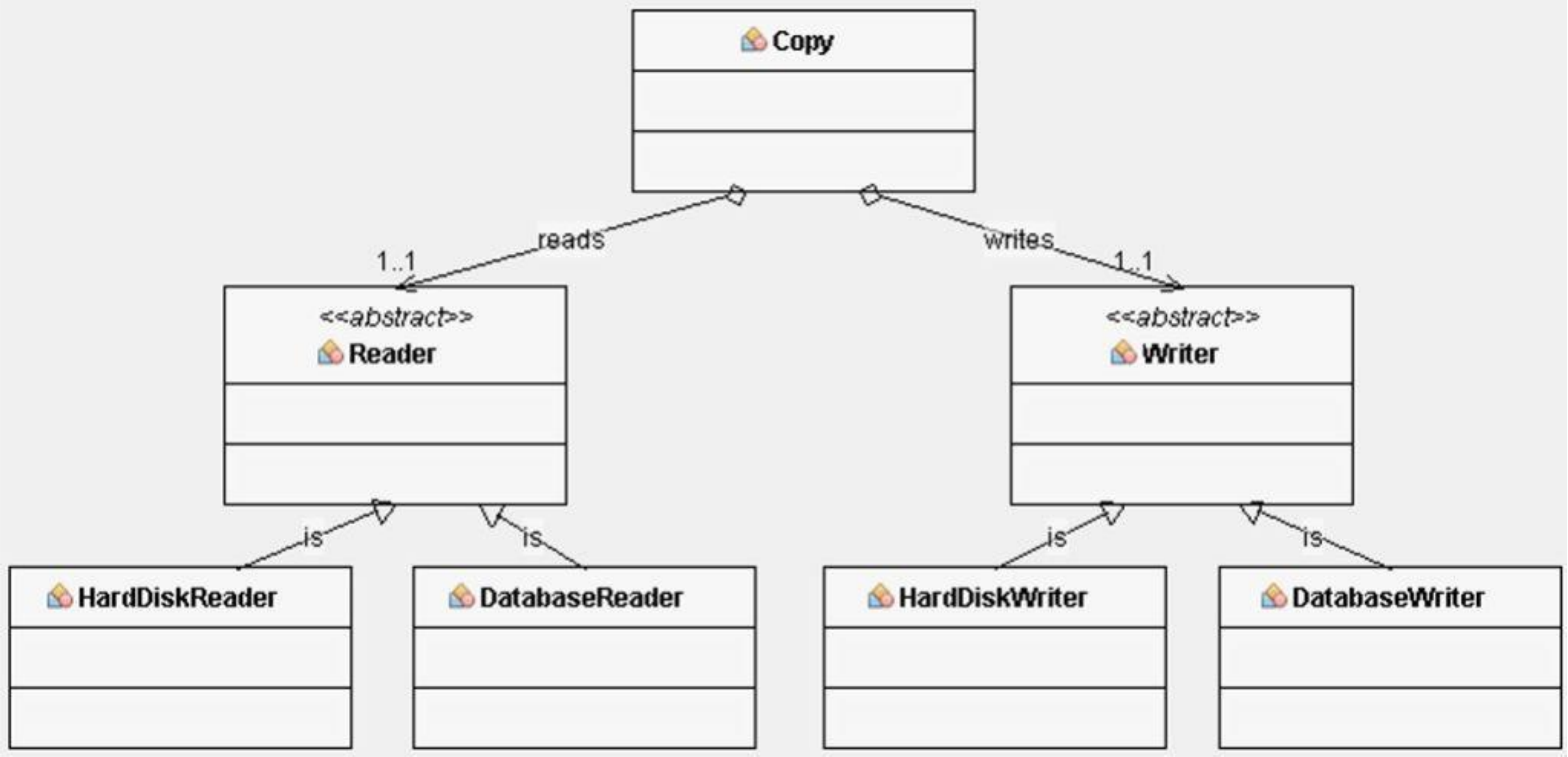
DIP Example

We have a class for copying from a source to a destination.

What happens if we decide to write to a Database instead?



DIP Example



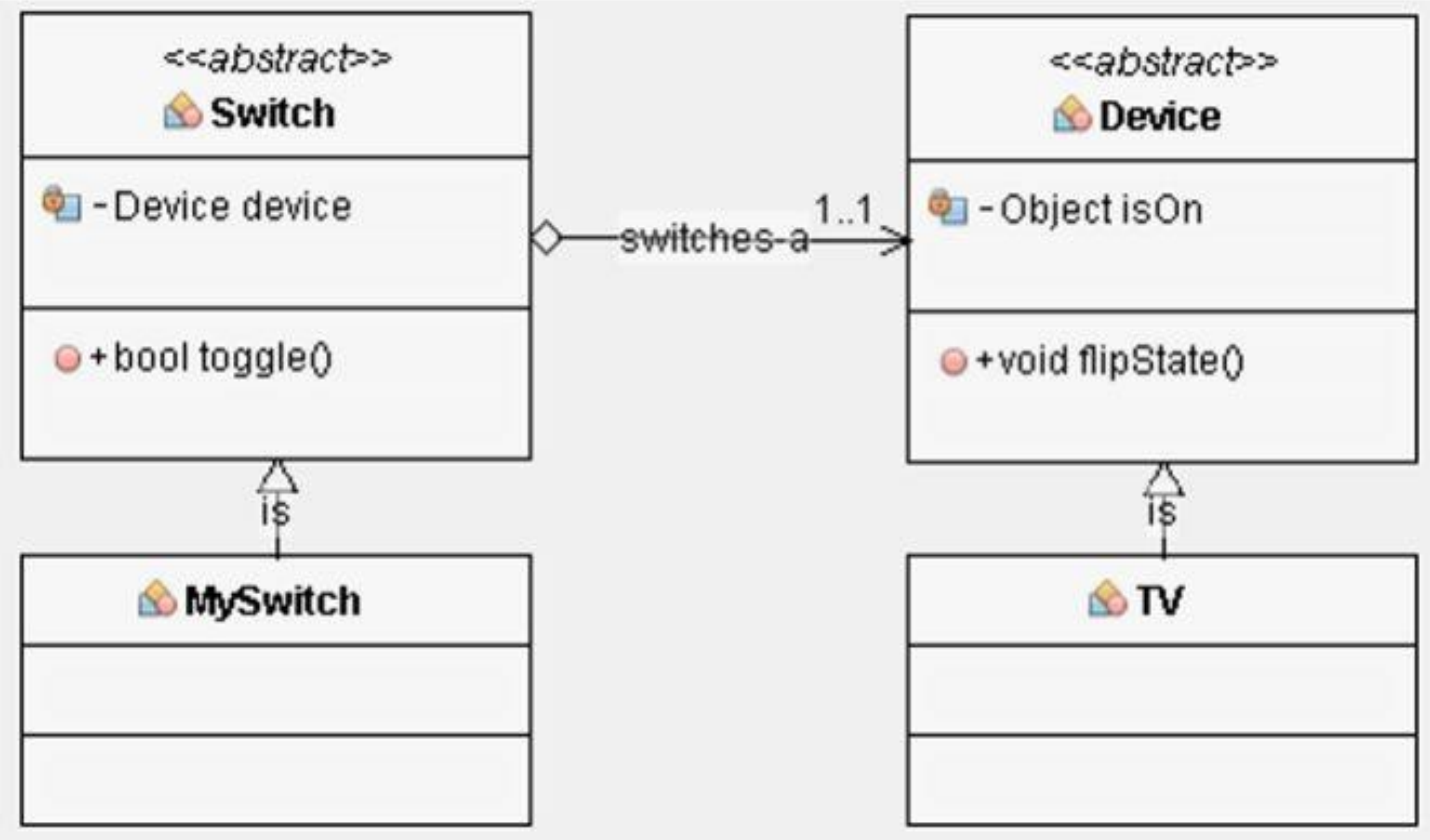
Copy
reads
1..1
writes
1..1
<<abstract>>
Reader
<<abstract>>
Writer
is
is
is
is
HardDiskReader
DatabaseReader
HardDiskWriter
DatabaseWriter

DIP Example

Consider the following scenario:

- We would like an application that can turn a TV on or off.
- The application should have at least two classes: Switch and TV.
- Design a program that adheres to DIP.

DIP Example



<<abstract>>
Switch
-Device device
+bool toggle()
switches-a
1..1
<<abstract>>
Device
-Object isOn
+void flipState()
is
is
MySwitch
TV

Thank you