# ASSIGNMENT # 1

## DESIGN DEFECTS & RESTRUCTURING (BCS-8A)

**HASSAN ALI**
**K20-1052**

**16TH FEB**

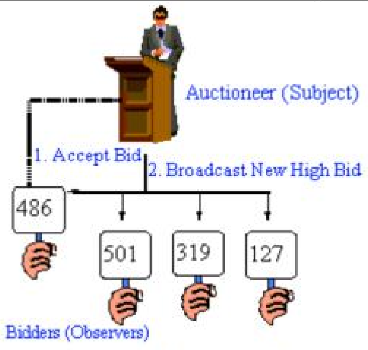
# Assignment #1

**DDR, Spring 2024**

**Question 1:** In this assignment, you need to describe your own example of any design pattern of your choice and give answers to all of the parts of the question 1. (Part A to Part D – see below) But your scenario / example must be a non-programming example explaining any real-life scenario, like the one given below.

**Observer Example**

The Observer defines a one to many relationships, so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price, which is broadcast to all of the bidders in the form of a new bid.



**Part (A) Give your own example (textual + graphic representation):**

Describe your own example of any designs pattern that we have covered in the class and give a substitute case of the design pattern being utilized in some other situation. (as shown in the above example) It very well may be a regular day to day example (e.g., how online goods are shipped, how to leave a place of employment and join another organization, ... whatever). In a manner similar to above example, briefly describe your example with a diagram and a brief textual description.

**Important Note:** The model you pick ought not be utilized in the GoF book (also minor varieties of the given examples are not allowed) or one that was utilized in the lecture slides or notes, neither it should be taken from any web article.

**Part (B) The problem, The Intent, The motivation and The Applicability**

Clarify how your case "coordinates" the example's essential thought, that is, the way your model is closely resembling the structure example's model. Typically, you should clarify how your model synchronizes with (1) the intent and (2) the problem being discuss. From the GoF patterns, it should coordinate the aim or intent, inspiration and applicability.

**Part (C) UML**

Draw a <u>class</u> diagram and <u>interaction</u> diagrams for the model you picked. What you sketch will rely upon the pattern you decided for your model. Utilize standard notations of UML.

**Part (D) Consequences**

Document a sensible situation that features one of the consequences (disadvantages) of the design pattern. That is, your case study must be feature an outcome that is a downside in the situation. Iyou're your situation should highlight how the solution failed to address your requirements. Else you should portray a situation wherein neglecting to utilize the pattern would bring about negative outcomes. For example, in strategy pattern there is a communication overhead between Strategy and its Context, you'll need tighter coupling between Strategy and Context.

**Part(E)** Write or Generate skeleton Code of class diagram in part (C).

**Question 2 – Pattern identification:**

Consider each of the scenarios below and identify the design pattern which is most directly addresses the problem described. Briefly explain your reasoning. (See what maps from GOF patterns and discuss and argue in favor of your thinking)

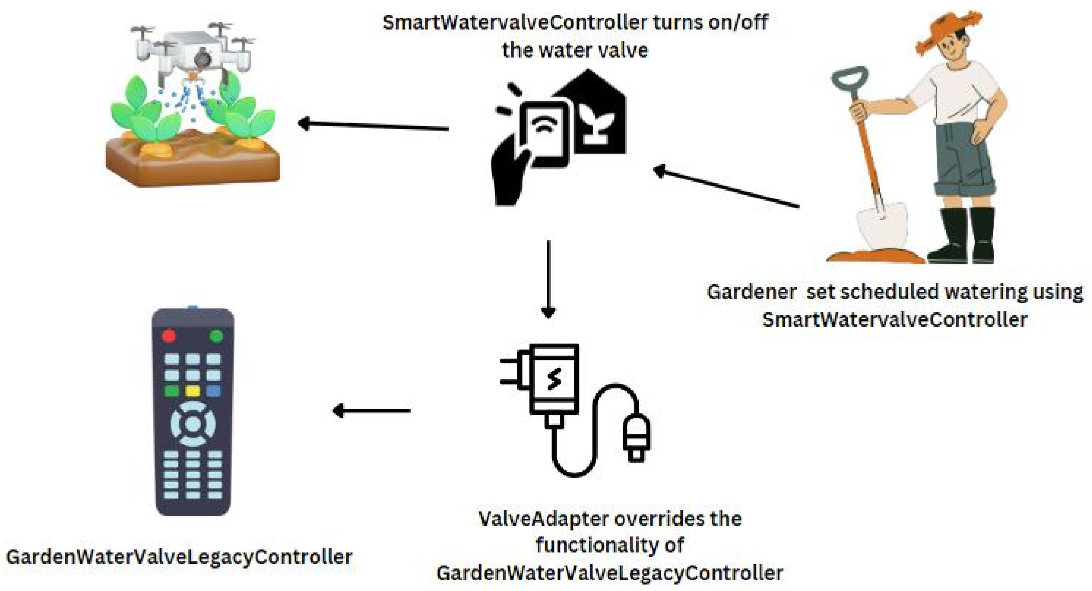
**Part (A):** You've developed a new implementation of the List interface and you'd like to test the behavior of your new data structure. You've written an algorithm to perform your speed tests, but the algorithm needs to make a great many instances of your list class. You want to test the performance of your list against the performance of ArrayList and LinkedList, but you don't want to have to write your algorithm three times in order for it to be able to create the right kind of list to test.

**Part (B)** You've completed a compiler for a new language! There are many parts to your compilation process: parsing, transformation, assembly code generation, and so forth. You'd like to allow other programmers to use an interface to compile their code without resorting to system calls or other command-line invocations – your compiler can just run in their processes – but you don't want those users to have to know how to bring all of the steps of compilation together in order to use your compiler.

# Solution (Adapter Pattern)

## Q1

**A)** We have a Smart Garden Watering System in which there is a **WaterApp** use to automate garden water schedules. **Valves** that controls the water flow to different areas of garden. **GardenWaterValvesLegacyController** that conrols the old valve hardware and a **SmartWaterValveController** that controls the hardware using the Wifi. The WateringApp wants to integrate with new smart SmartWaterValveController to control Valves, however, the old legacy GardenWaterValveLegacyController is still installed to operate the valves manually. Therefore we need a **ValveAdapter** to adapt the new controller interfaces into the commands compatible with the old legacy controller. In this way we can integrate the new controller while reusing the old valve hardware.


SmartWatervalveController turns on/off the water valve
Gardener set scheduled watering using SmartWatervalveController
GardenWaterValveLegacyController
ValveAdapter overrides the functionality of GardenWaterValveLegacyController

**B) The Intent:**

The intent of Adapter pattern is to convert one interface into another so that classes can work together that couldn't otherwise because of incompatible interfaces.
My example showcases this by adapting between the new smart water valve controller interfaces and the legacy garden valve controller interfaces, allowing the new and old valve systems to work together from the WateringApp client perspective.

**The Problem:**

The Adapter pattern solves problems like:

- Reusing legacy components with incompatible interfaces
- Integrating with 3rd party systems having different interfaces
- Allowing substitution of components without changing clients

In my example, the problem is integrating the new WiFi valve control system while reusing the legacy valves. The ValveAdapter allows this integration and substitution of smart valve controller without changing the WateringApp.

**The Motivation:**

Typical motivations are:

- Client independence from concrete implementation
- Reuse existing functionality
- Introduce substitutes without changing clients

The motivation in my system is to enable client (WateringApp) independence from underlying valve control hardware. It can automatically control watering while allowing manual operation, substituting newer components over time.

**The Applicability:**

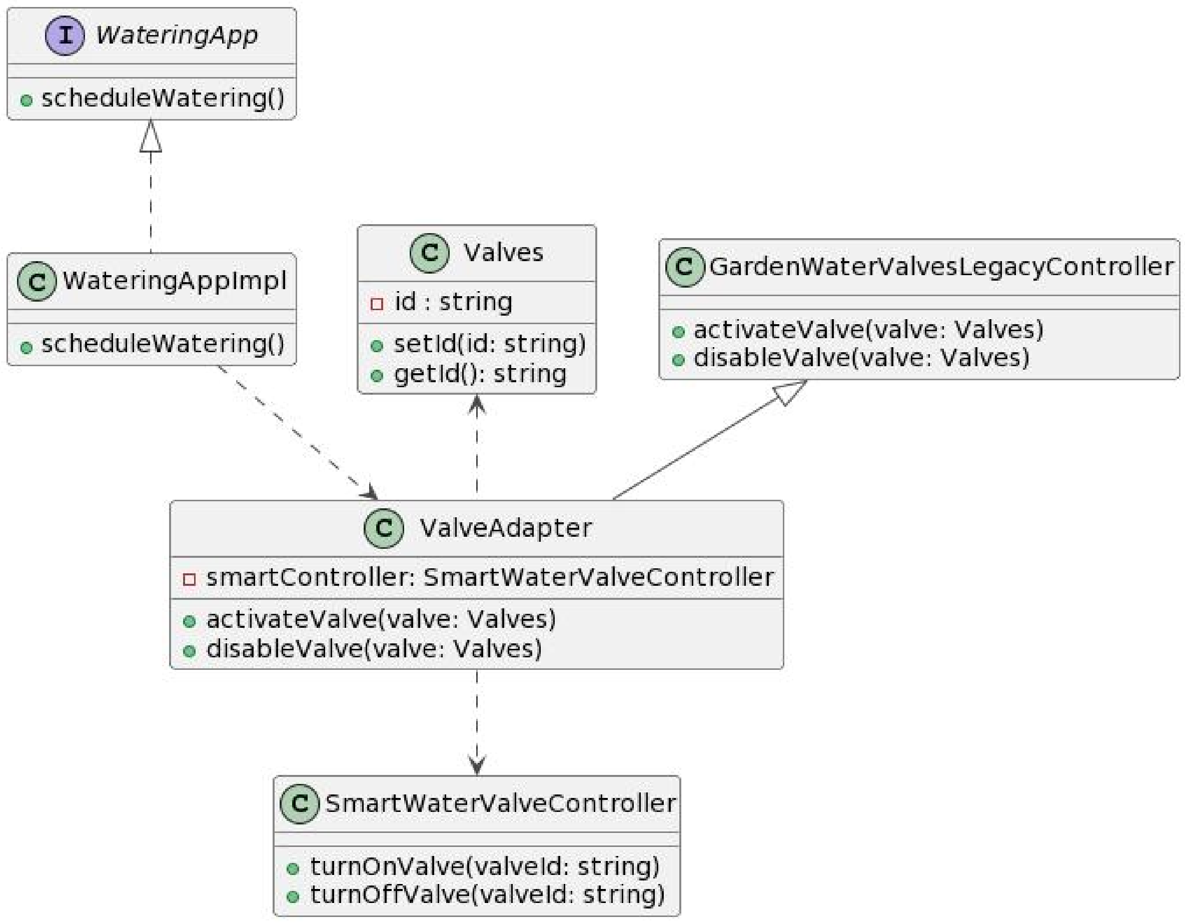
Adapter pattern applies when:

- A class's interface does not match what client requires
- Reusing classes even if interfaces don't match
- Features need to be used from incompatible libraries

In my case, it enables integrating the new SmartWaterValveController with incompatible legacy valve interfaces without replacing entire hardware. Allowing reuse of existing valves while adding new automated features.

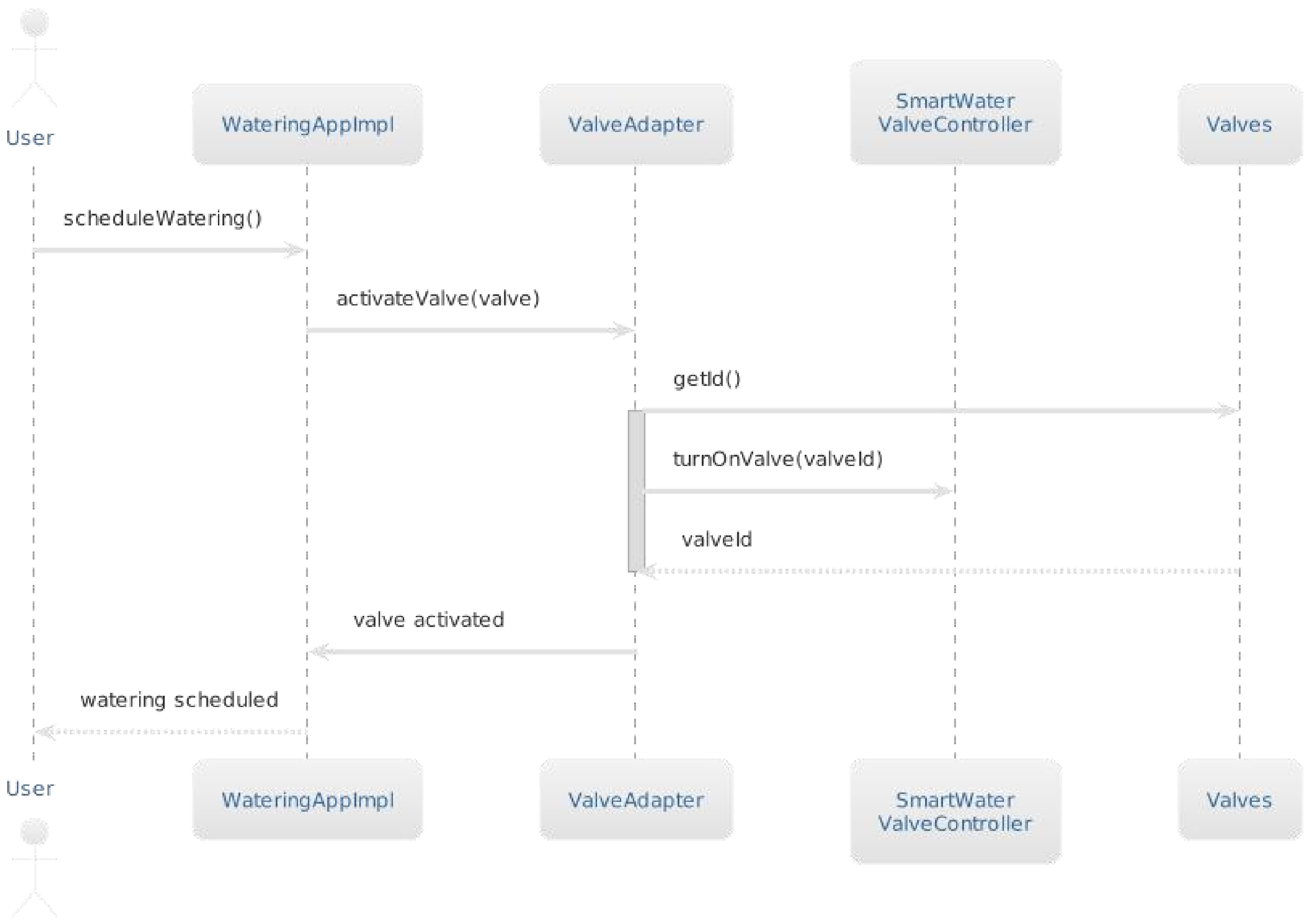
**C)** I have used PlantUML for creating UML diagrams

**Class Diagram**



WateringApp
scheduleWatering()
WateringAppImpl
scheduleWatering()
Valves
id : string
setId(id: string)
getId(): string
GardenWaterValvesLegacyController
activateValve(valve: Valves)
disableValve(valve: Valves)
ValveAdapter
smartController: SmartWaterValveController
activateValve(valve: Valves)
disableValve(valve: Valves)
SmartWaterValveController
turnOnValve(valveId: string)
turnOffValve(valveId: string)

**Interaction Diagram**



**D)** The adaption between the legacy garden valves controllers and the new WiFi-enabled controllers has an added performance overhead. Each time a watering schedule request comes from the WateringApp, the ValveAdapter needs to:

- Identify the appropriate valve based on valve ID
- Map the valve ID to corresponding controller
- Translate the control APIs for that controller
- Route requests & responses between new/legacy controllers

This runtime mapping, translation and routing introduces latency in the valve activation process.
During peak summer watering schedules, when there are lots of parallel requests to open valves across zones, this added latency results in slight delays in valves opening. Over time, the delays add up and impact the overall water distribution.

**E)**

```java
// WateringApp interface
interface WateringApp {
  public void scheduleWatering();
}

// Concrete implementation of WateringApp
class WateringAppImpl implements WateringApp {

  WateringAppImpl() {
    // Constructor logic
  }

  @Override
  public void scheduleWatering() {
    // Schedule watering logic
  }

}

// Valve class
class Valve {

  private String id;

  public void setId(String id) {
    this.id = id;
  }

  public String getId() {
    return this.id;
  }

}

// Legacy water valve controller
class GardenWaterValveLegacyController {

  public void activateValve(Valve valve) {
    // Logic to activate valve
  }

  public void disableValve(Valve valve) {
    // Logic to disable valve
  }

}
```

```java
// Modern smart valve controller
class SmartWaterValveController {

  public void turnOnValve(String valveId) {
    System.out.println("Turning on valve " + valveId);
  }

  public void turnOffValve(String valveId) {
    System.out.println("Turning off valve " + valveId);
  }

}

// Valve adapter
class ValveAdapter extends GardenWaterValveLegacyController {

  private SmartWaterValveController smartController;

  public ValveAdapter(SmartWaterValveController smartController2)
{
    this.smartController = smartController2;
  }

  @Override
  public void activateValve(Valve valve) {
    // Adapt to smart controller API
    smartController.turnOnValve(valve.getId());
  }

  @Override
  public void disableValve(Valve valve) {
    // Adapt to smart controller API
    smartController.turnOffValve(valve.getId());
  }

}

class GardenWatering {

  public static void main(String[] args) {

    // Create watering app
    WateringApp wateringApp = new WateringAppImpl();

    // Create legacy valves
    Valve gardenValve1 = new Valve();
    gardenValve1.setId("valve1");

    Valve gardenValve2 = new Valve();
    gardenValve2.setId("valve2");
    // Create smart valve controller
    SmartWaterValveController smartController = new
SmartWaterValveController();
```

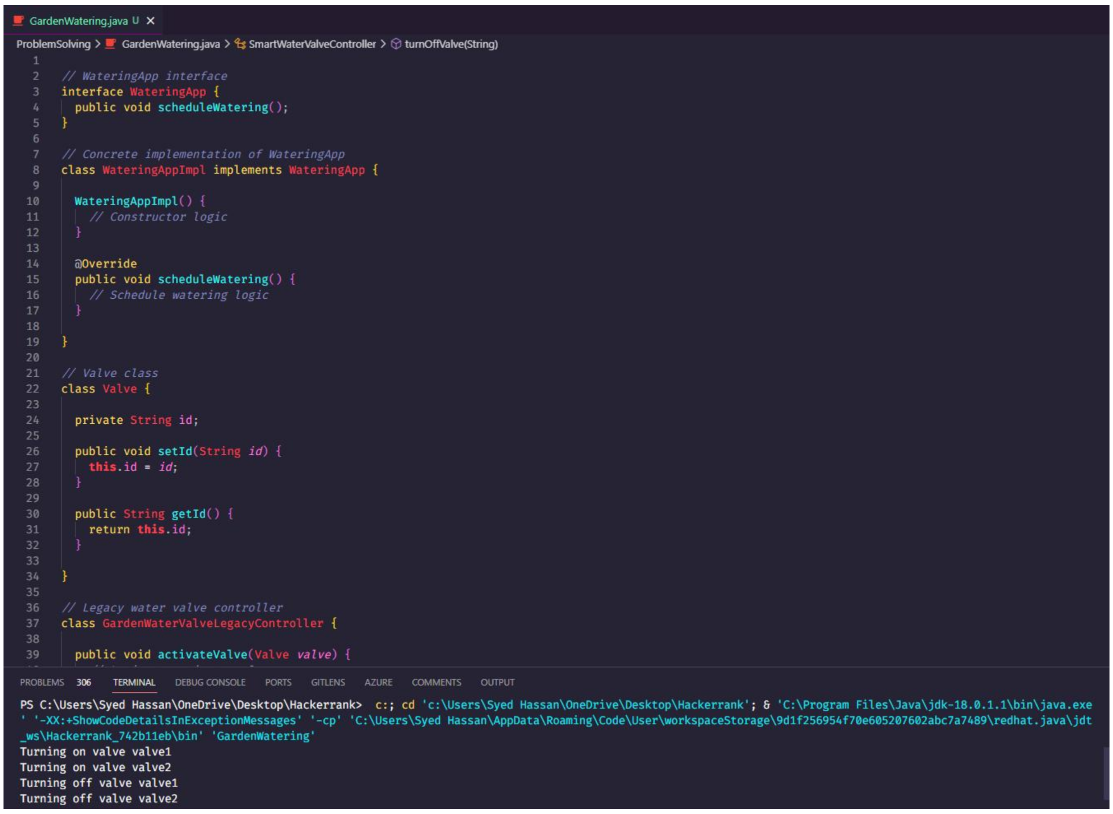
```
    // Create valve adapter
    ValveAdapter adapter = new ValveAdapter(smartController);
```

```
    // Schedule watering using adapter for both valve types
    wateringApp.scheduleWatering();
    adapter.activateValve(gardenValve1);
```

```
    wateringApp.scheduleWatering();
    adapter.activateValve(gardenValve2);
```

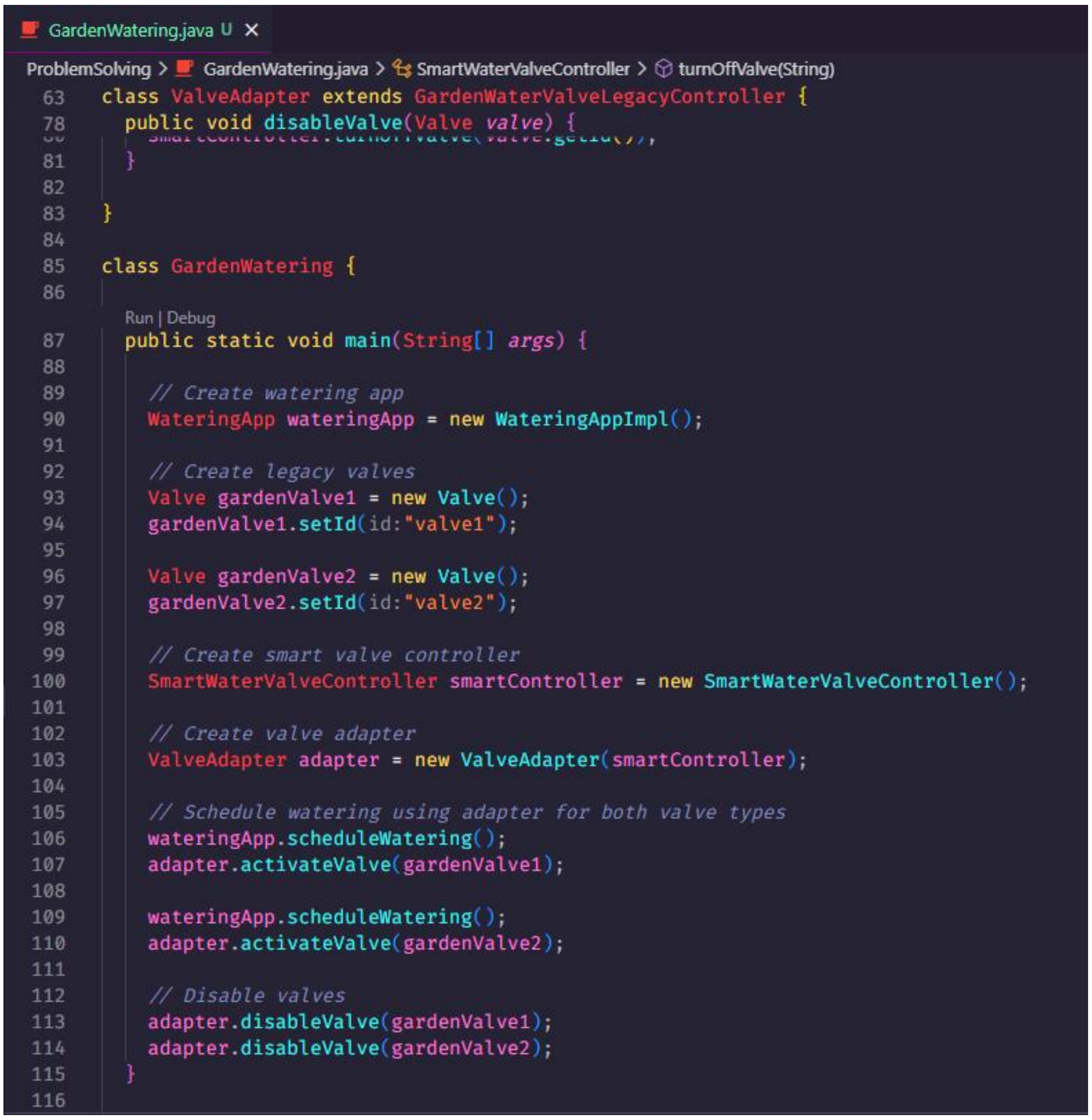
```
    // Disable valves
    adapter.disableValve(gardenValve1);
    adapter.disableValve(gardenValve2);
  }
```

```
}
```


GardenWatering.java U ×
ProblemSolving > GardenWatering.java > SmartWaterValveController > turnOffValve(String)
PROBLEMS 306
TERMINAL
DEBUG CONSOLE
PORTS
GITLENS
AZURE
COMMENTS
OUTPUT

```java

// WateringApp interface
interface WateringApp {
  public void scheduleWatering();
}

// Concrete implementation of WateringApp
class WateringAppImpl implements WateringApp {

  WateringAppImpl() {
    // Constructor logic
  }

  @Override
  public void scheduleWatering() {
    // Schedule watering logic
  }

}

// Valve class
class Valve {

  private String id;

  public void setId(String id) {
    this.id = id;
  }

  public String getId() {
    return this.id;
  }

}

// Legacy water valve controller
class GardenWaterValveLegacyController {

  public void activateValve(Valve valve) {
```

```
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>  c:; cd 'c:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank'; & 'C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe
' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Syed Hassan\AppData\Roaming\Code\User\workspaceStorage\9d1f256954f70e605207602abc7a7489\redhat.java\jdt
_ws\Hackerrank_742b11eb\bin' 'GardenWatering'
Turning on valve valve1
Turning on valve valve2
Turning off valve valve1
Turning off valve valve2
```



GardenWatering.java U ×

ProblemSolving > GardenWatering.java > SmartWaterValveController > turnOffValve(String)

```java
class ValveAdapter extends GardenWaterValveLegacyController {
  public void disableValve(Valve valve) {
    smartController.turnOffValve(valve.getId());
  }

}

class GardenWatering {

  public static void main(String[] args) {

    // Create watering app
    WateringApp wateringApp = new WateringAppImpl();

    // Create legacy valves
    Valve gardenValve1 = new Valve();
    gardenValve1.setId(id:"valve1");

    Valve gardenValve2 = new Valve();
    gardenValve2.setId(id:"valve2");

    // Create smart valve controller
    SmartWaterValveController smartController = new SmartWaterValveController();

    // Create valve adapter
    ValveAdapter adapter = new ValveAdapter(smartController);

    // Schedule watering using adapter for both valve types
    wateringApp.scheduleWatering();
    adapter.activateValve(gardenValve1);

    wateringApp.scheduleWatering();
    adapter.activateValve(gardenValve2);

    // Disable valves
    adapter.disableValve(gardenValve1);
    adapter.disableValve(gardenValve2);
  }

```

## Q2

**A)** This scenario dealing with creating different types of lists to test an algorithm aligns very well with the Factory pattern.
The key motivation is to instantiate related but differing classes based on type without specifying concrete classes explicitly. Factory handles creating objects so client code can focus on using them.
So you could have a ListFactory that can produce ArrayList, LinkedList, or MyCustomList based on input parameter. The test algorithm code just calls factory to get required type of list, remaining isolated from actual instantiation.

**B)** The scenario of allowing client programs to invoke a compiler without knowing compilation steps matches very closely with Facade pattern.
Key motivations of Facade are:

- Provide a simple, unified interface hiding subsystem complexity
- Decouple client from intricate internal implementations
- Improve readability, maintainability of client code

So you could have a CompilerFacade that exposes clean compile() interfaces to client code, while internally it coordinates between parser, generators, optimizers etc. to perform full compilation without exposing client to that complexity.