

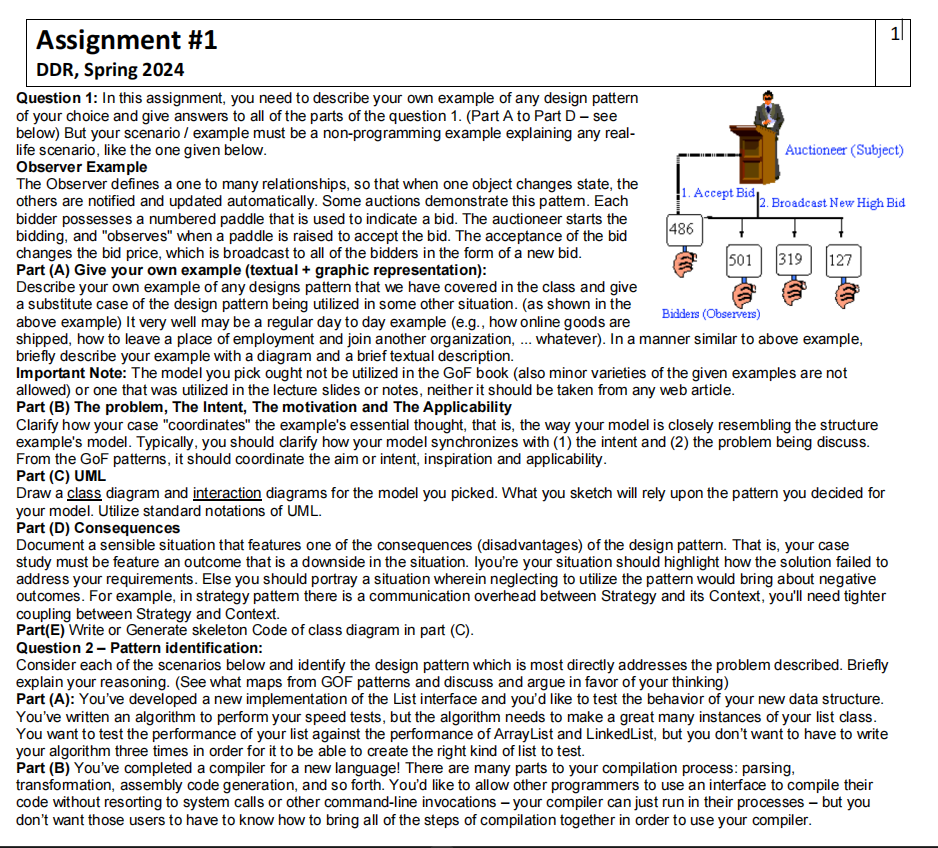
**ASSIGNMENT # 1**

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

**HASSAN ALI**

**K20-1052**

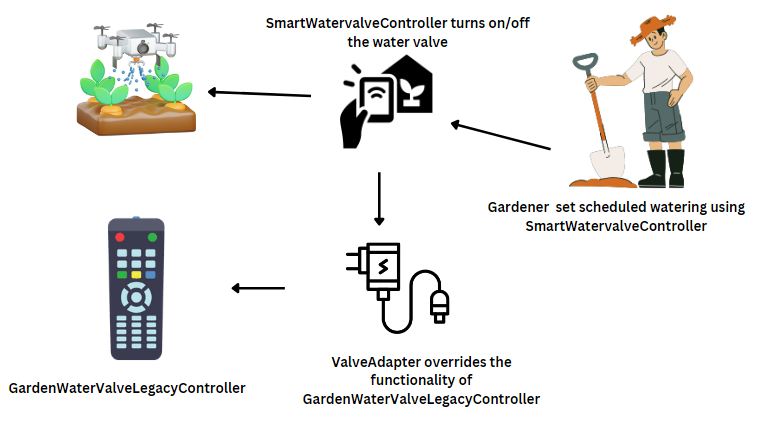
**16TH FEB**



**Solution (Adapter Pattern)**

**Q1**

1. 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.



1. **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:**

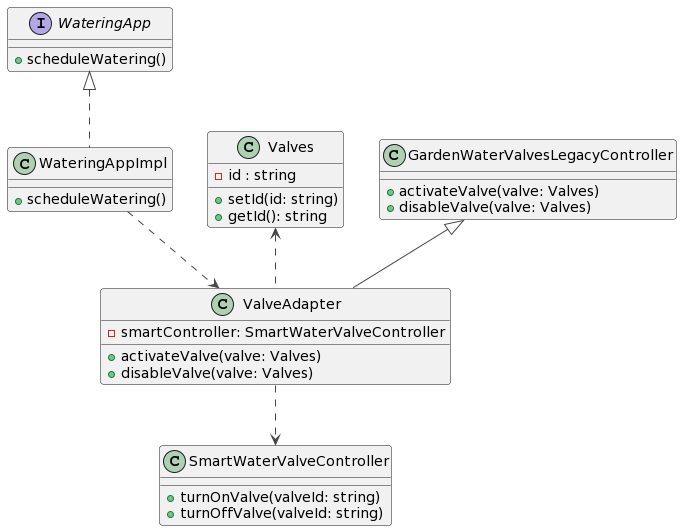
Adapter pattern applies when:

* 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

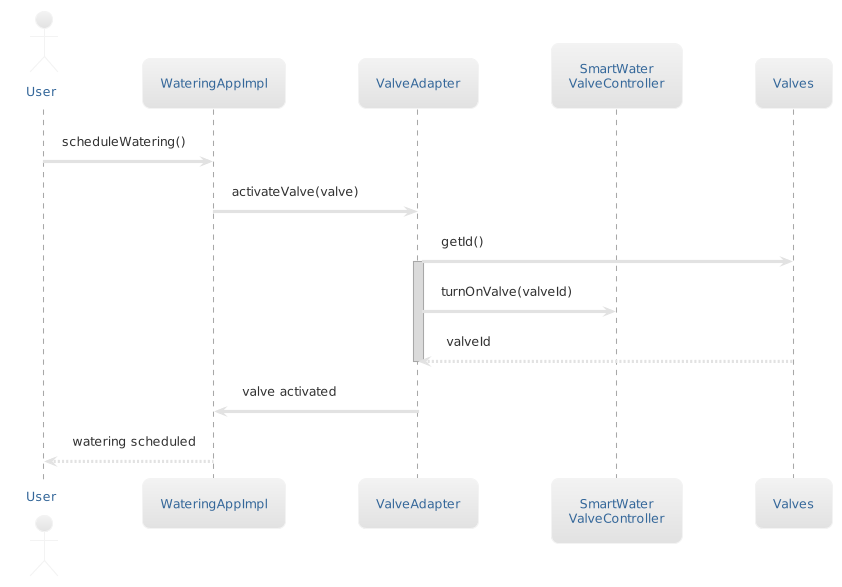
In my case, it enables integrating the new SmartWaterValveController with incompatible legacy valve interfaces without replacing entire hardware. Allowing reuse of existing valves while adding new automated features.

1. I have used PlantUML for creating UML diagrams

**Class Diagram**



**Interaction Diagram**

1.  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)**

*// 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*

  }

}

*// 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);

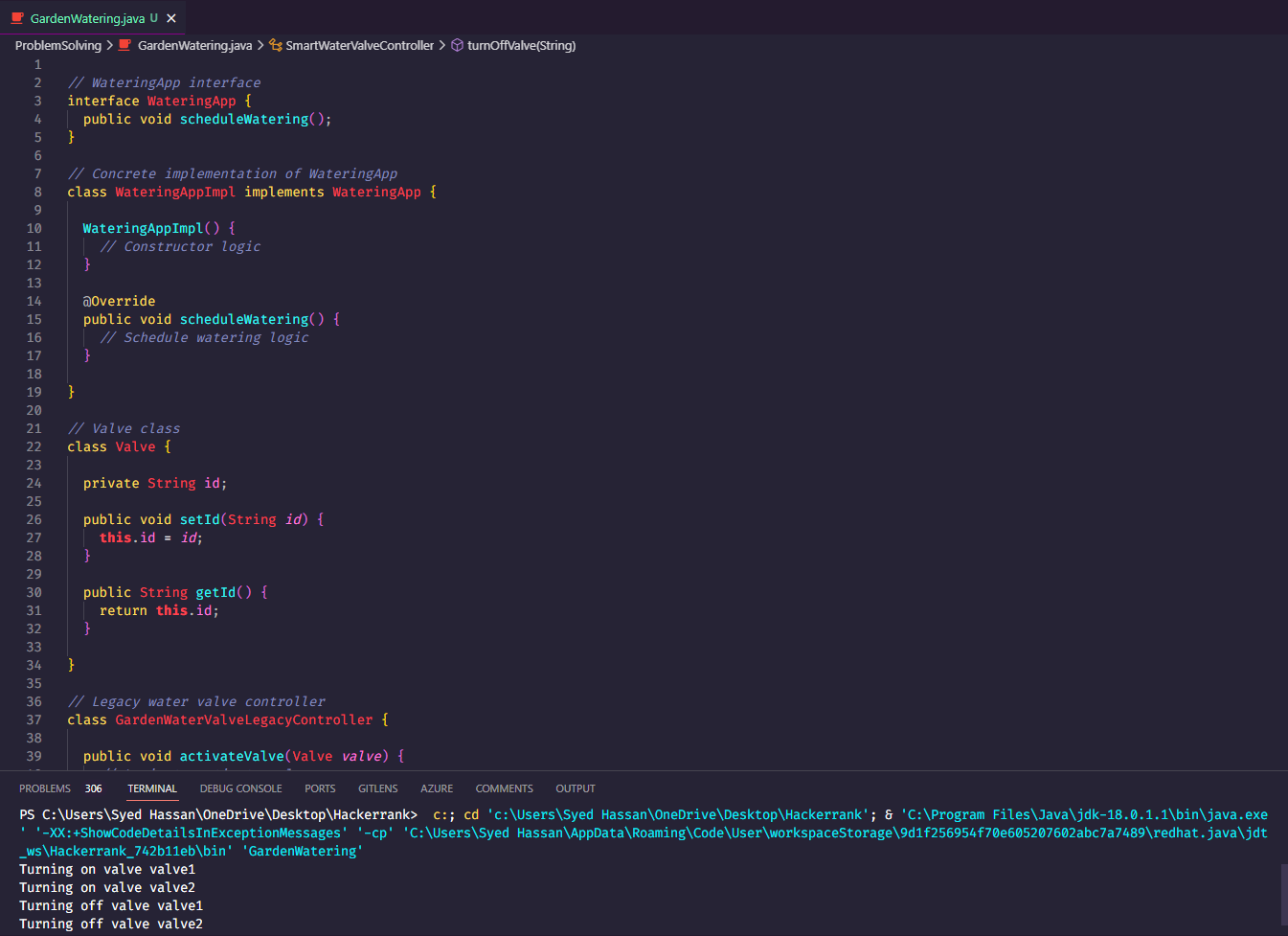
*// Disable valves*

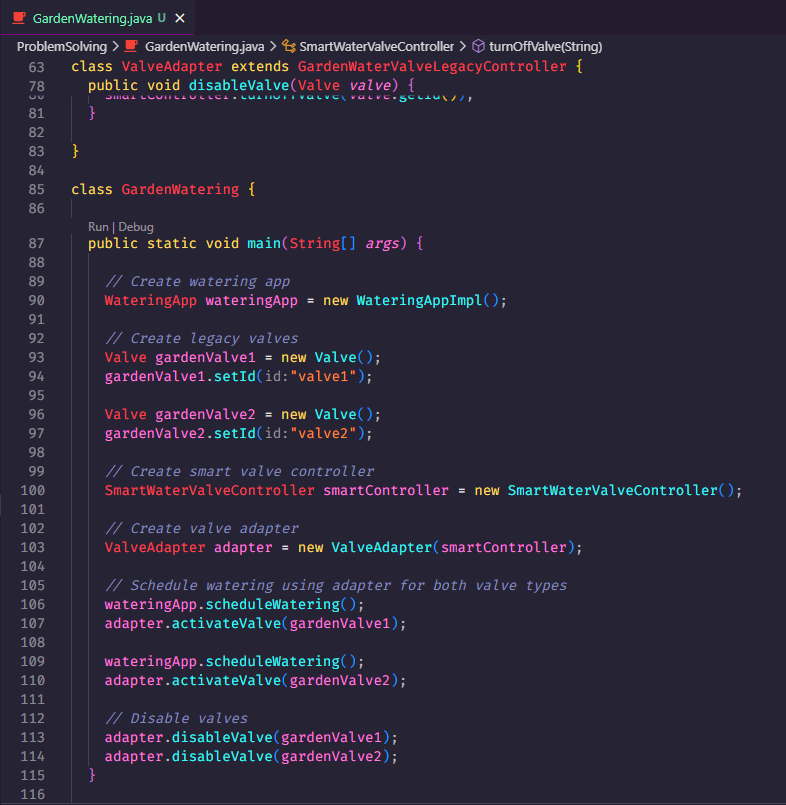
    adapter.disableValve(gardenValve1);

    adapter.disableValve(gardenValve2);

  }

}





**Q2**

1. 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.

1. 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.