2023

# Object Oriented Analysis and Design Homework

1801042617

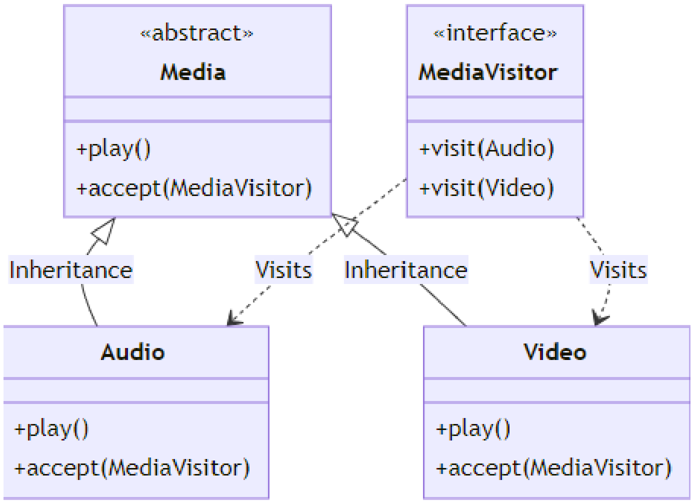
ELIFNUR KABALCI

1)(a) The problem with adding new operations like filter() and export() to this design is that it violates the Open-Closed Principle (OCP), which states that software entities should be open for extension, but closed for modification. In this case, every time a new operation needs to be added, you would have to modify the base Media class and all its derived classes (Audio, Video, etc.). This can lead to code fragility and it makes the system harder to maintain as it grows.

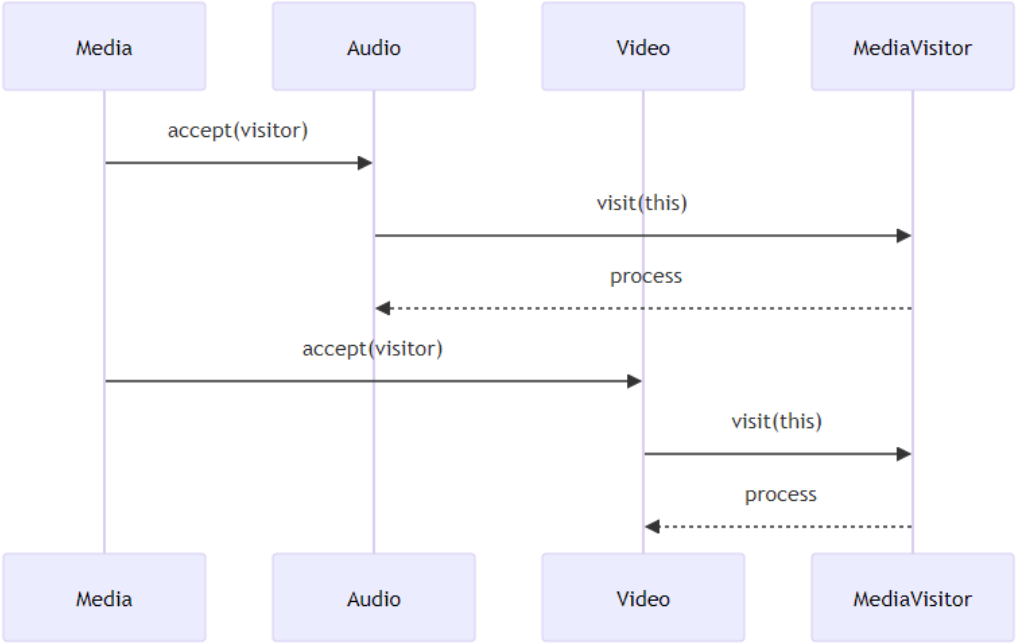
The solution to this problem is to use the Visitor design pattern. This pattern allows you to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.

```
class MediaVisitor;

class Media {
public:
   virtual void play() = 0;
   virtual void accept(MediaVisitor& visitor) = 0; // accept a visitor
};

class Audio : public Media {
public:
   void play() override {
      // Audio play code
   }
   void accept(MediaVisitor& visitor) override;
};

class Video : public Media {
public:
   void play() override {
      // Video play code
   }
   void accept(MediaVisitor& visitor) override;
};

class MediaVisitor {
public:
   virtual void visit(Audio& audio) = 0; // visit an Audio
   virtual void visit(Video& video) = 0; // visit a Video
};
```

Class Diagram:



Sequential Diagram:

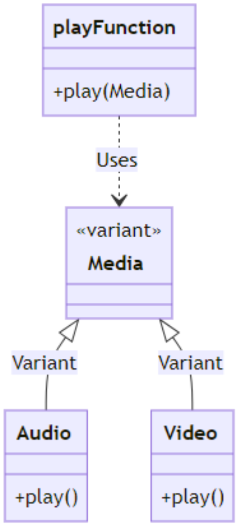


1)(b) In a value-based approach, instead of using inheritance and polymorphism, you would use a variant type to hold either an Audio or Video value, and then use std::visit to apply operations to the variant. This approach is more flexible and allows you to add new operations without changing the Media classes. However, it requires more boilerplate code and is less efficient due to the need to check the type of the variant at runtime.

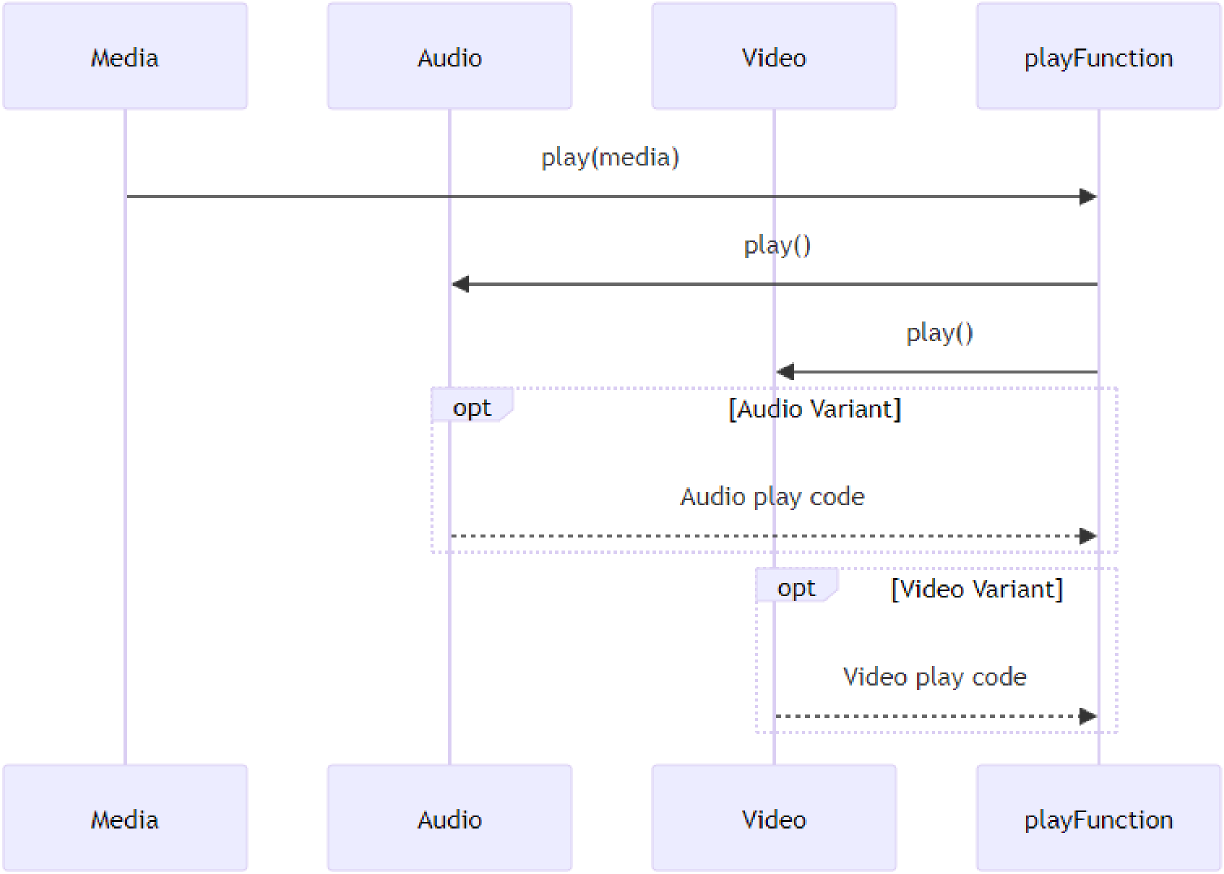
```
#include <variant>

class Audio {
public:
   void play() {
      // Audio play code
   }
};

class Video {
public:
   void play() {
      // Video play code
   }
};

using Media = std::variant<Audio, Video>;

void play(Media& media) {
   std::visit( { m.play(); }, media);
}
```

In this code, Media is a variant that can hold either an Audio or Video. The play function uses std::visit to apply the play operation to the Media, regardless of whether it’s an Audio or Video. You can add new operations in a similar way, without needing to modify the Media classes. However, each new operation requires a new function, which can lead to code duplication if many operations are needed. This approach also doesn’t support operations that need to behave differently depending on the combination of types involved. For that, you would need to use a more complex approach, such as multiple dispatch.

Class Diagram:



Sequential Diagram:



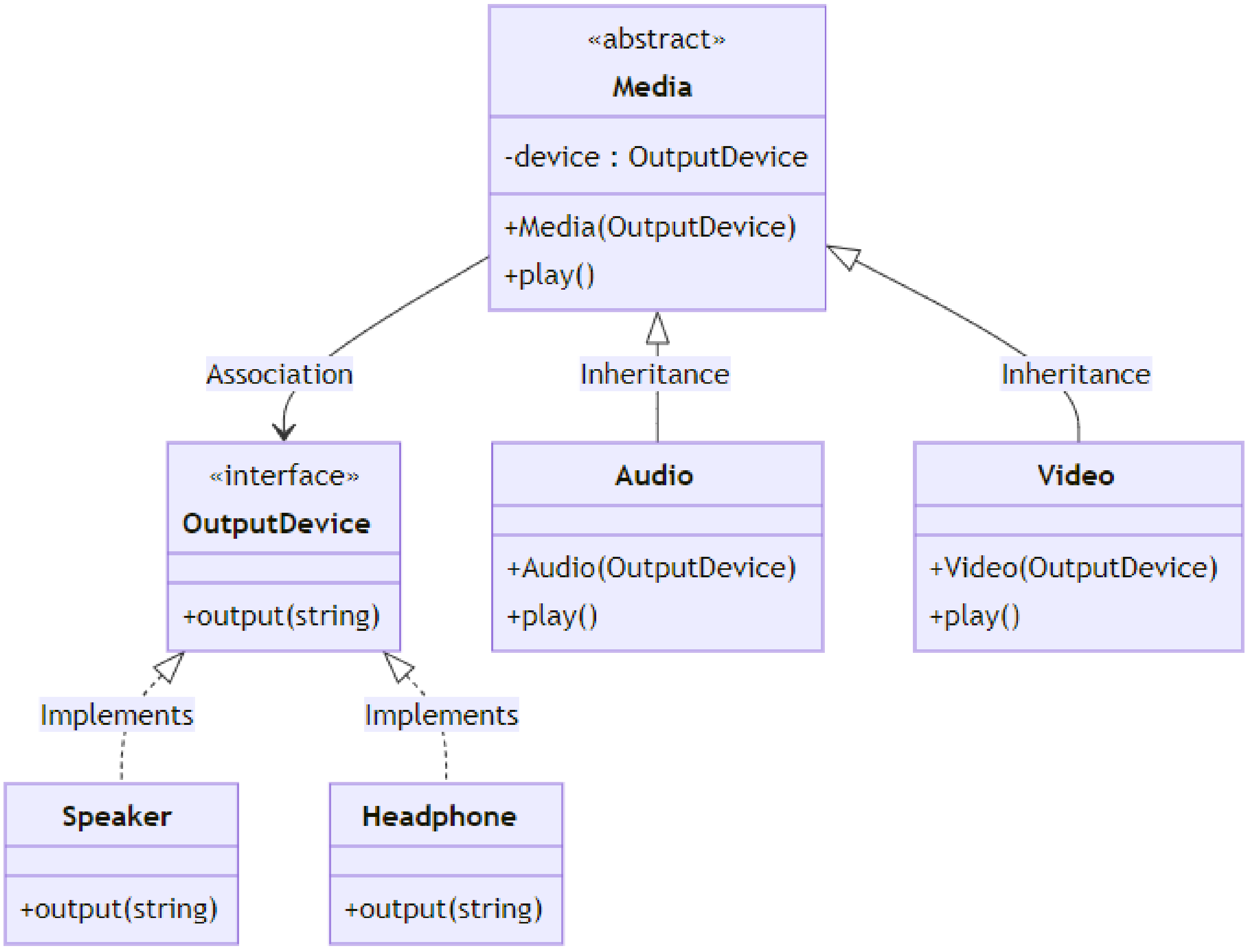
2)The Bridge design pattern is a structural pattern that decouples an abstraction from its implementation so that the two can vary independently. This pattern involves an interface which acts as a bridge and enhances the functionality of independent interfaces. In this case, the Media class hierarchy and the output device class hierarchy can be decoupled using the Bridge pattern.

```
// OutputDevice is the Implementor in the Bridge pattern
class OutputDevice {
public:
    virtual ~OutputDevice() {}
    virtual void output(const std::string& soundData) = 0;
};

class Speaker : public OutputDevice {
public:
    void output(const std::string& soundData) override {
        // output soundData to speakers
    }
};

class Headphone : public OutputDevice {
public:
    void output(const std::string& soundData) override {
        // output soundData to headphones
    }
};

// Media is the Abstraction in the Bridge pattern
class Media {
public:
    Media(OutputDevice* device) : device(device) {}
    virtual ~Media() {}
    virtual void play() = 0;

protected:
    OutputDevice* device;  // the Bridge
};

class Audio : public Media {
public:
    Audio(OutputDevice* device) : Media(device) {}
    void play() override {
        std::string soundData = /* get sound data */;
        device->output(soundData);  // delegate to the Implementor
    }
};

class Video : public Media {
```

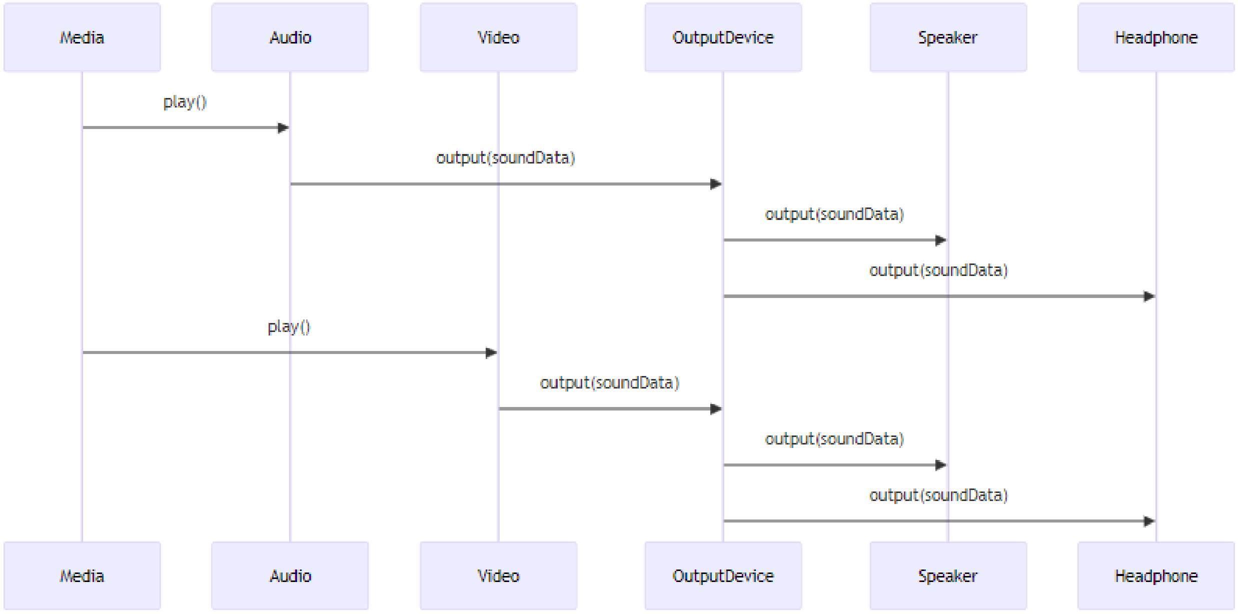
```
public:
    Video(OutputDevice* device) : Media(device) {}
    void play() override {
        std::string soundData = /* get sound data */;
        device->output(soundData);  // delegate to the Implementor
    }
};
```

In this code, Media is the “Abstraction” in the Bridge pattern, and OutputDevice is the “Implementor”. The play method in Media (and its subclasses) delegates to the output method of the OutputDevice, allowing the output device to vary independently of the media type. This design allows you to add new types of media and new types of output devices without modifying existing classes. You can simply create a new subclass of Media or OutputDevice as needed. This is in line with the Open-Closed Principle (OCP), which states that software entities should be open for extension, but closed for modification. The Bridge pattern helps achieve this goal by decoupling the Media and OutputDevice hierarchies.

Class Diagram:



Sequential Diagram:



3)The external polymorphism design pattern, also known as the Adapter pattern, allows you to use polymorphism with classes that weren't designed with this in mind. In this case, you can create an abstract Shape class with draw and serialize methods, and then create adapter classes for LibCircle and LibSquare that implement these methods.

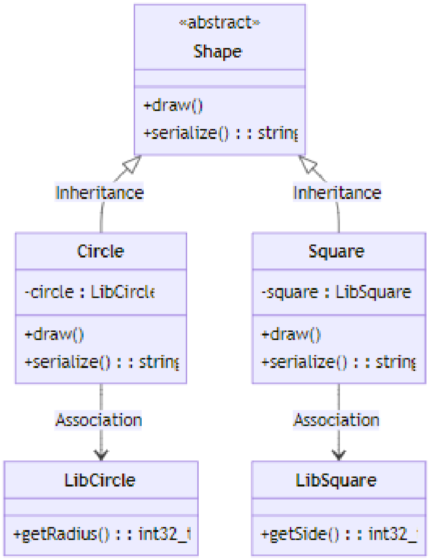
```
#include <iostream>
#include <string>

// Abstract base class
class Shape {
public:
   virtual ~Shape() {}
   virtual void draw() = 0;
   virtual std::string serialize() = 0;
};

// Adapter for LibCircle
class Circle : public Shape {
public:
   Circle(LibCircle* circle) : circle(circle) {}
   void draw() override {
      // Draw the circle using its radius
      int32_t radius = circle->getRadius();
```

```
        std::cout << "Drawing a circle with radius " << radius << std::endl;
    }
    std::string serialize() override {
        // Serialize the circle's data
        int32_t radius = circle->getRadius();
        return "Circle with radius " + std::to_string(radius);
    }
private:
    LibCircle* circle;
};

// Adapter for LibSquare
class Square : public Shape {
public:
    Square(LibSquare* square) : square(square) {}
    void draw() override {
        // Draw the square using its side length
        int32_t side = square->getSide();
        std::cout << "Drawing a square with side length " << side << std::endl;
    }
    std::string serialize() override {
        // Serialize the square's data
        int32_t side = square->getSide();
        return "Square with side length " + std::to_string(side);
    }
private:
    LibSquare* square;
};
```

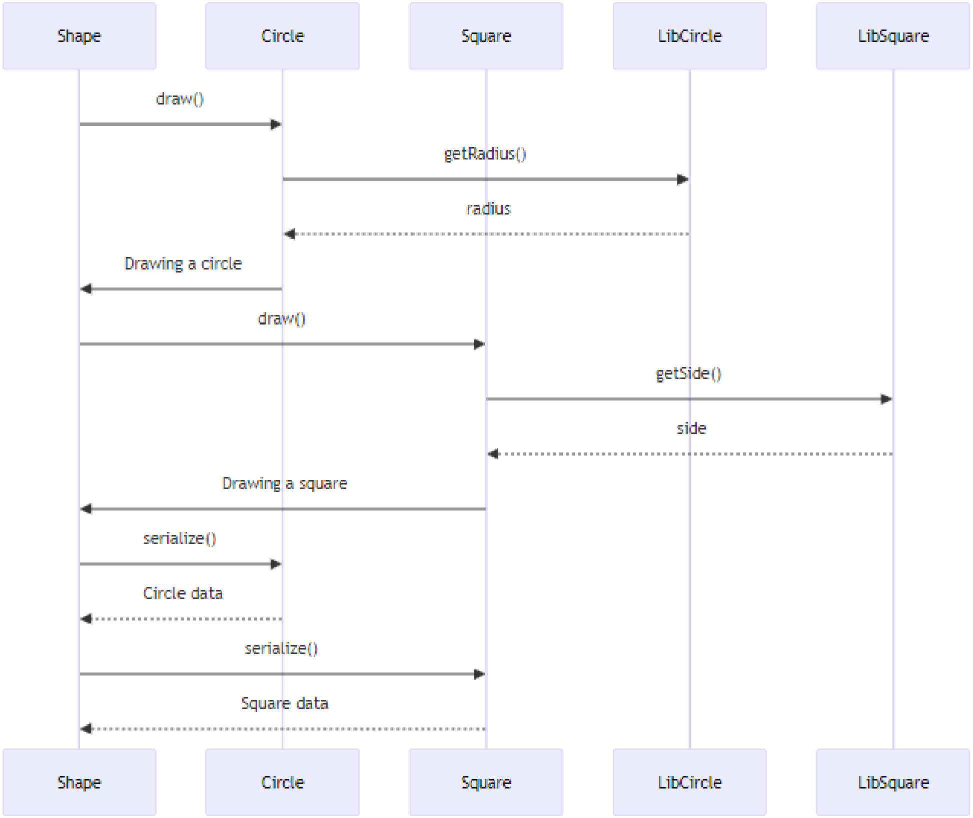
In this code, Shape is an abstract base class that declares the draw and serialize methods. Circle and Square are adapter classes that implement these methods for LibCircle and LibSquare, respectively. The draw method outputs a message to the console, and the serialize method returns a string representing the shape’s data. You can replace these implementations with your own drawing and serialization code.

This design allows you to treat LibCircle and LibSquare objects polymorphically as Shape objects, and to add new operations (like draw and serialize) without modifying the LibCircle and LibSquare classes. If you need to add more shapes in the future, you can simply create a new adapter class for each one. This is in line with the Open-Closed Principle (OCP), which states that software entities should be open for extension, but closed for modification. The Adapter pattern helps achieve this goal by providing a flexible interface for adding new operations to existing classes.

## Class Diagram:



## Sequential Diagram:



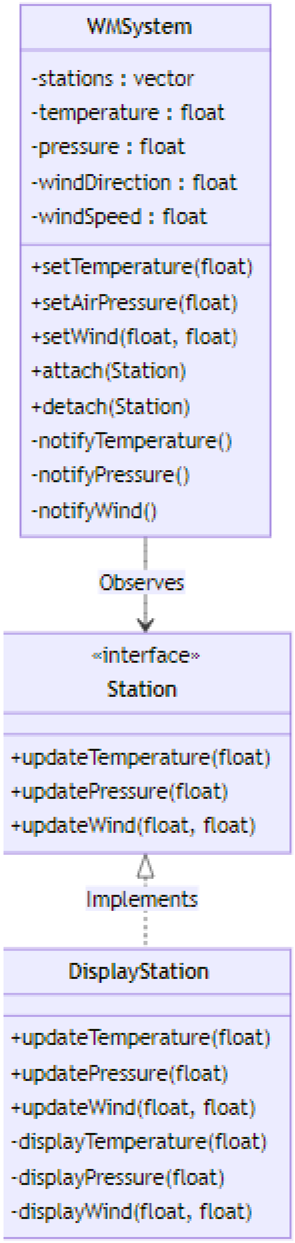
4)The Observer design pattern is a perfect fit for this scenario. It allows an object (the “subject”) to notify other objects (the “observers”) when its state changes, without knowing anything about these observers. This makes it easy to add new observers without modifying the subject.

```
#include <vector>

class Station {
public:
    virtual ~Station() {}
    virtual void updateTemperature(float temp) = 0;
    virtual void updatePressure(float pres) = 0;
    virtual void updateWind(float dir, float speed) = 0;
};

class WMSystem {
public:
    void setTemperature(float temp) {
        temperature = temp;
        notifyTemperature();
    }
    void setAirPressure(float pres) {
        pressure = pres;
        notifyPressure();
    }
    void setWind(float dir, float speed) {
        windDirection = dir;
        windSpeed = speed;
        notifyWind();
    }
    void attach(Station* station) {
        stations.push_back(station);
    }
    void detach(Station* station) {
        stations.erase(std::remove(stations.begin(), stations.end(), station), stations.end());
    }

private:
    std::vector<Station*> stations;
    float temperature;
    float pressure;
    float windDirection;
    float windSpeed;

    void notifyTemperature() {
        for (Station* station : stations) {
            station->updateTemperature(temperature);
        }
    }
    void notifyPressure() {
        for (Station* station : stations) {
            station->updatePressure(pressure);
        }
    }
    void notifyWind() {
        for (Station* station : stations) {
            station->updateWind(windDirection, windSpeed);
        }
    }
};

class DisplayStation : public Station {
public:
    void updateTemperature(float temp) override {
        displayTemperature(temp);
    }
    void updatePressure(float pres) override {
        displayPressure(pres);
    }
    void updateWind(float dir, float speed) override {
        displayWind(dir, speed);
    }

private:
    void displayTemperature(float temp) {
        // Display temperature
    }
    void displayPressure(float pres) {
        // Display pressure
    }
    void displayWind(float dir, float speed) {
        // Display wind
    }
};

// Similarly, you can define LogStation and other stations
```

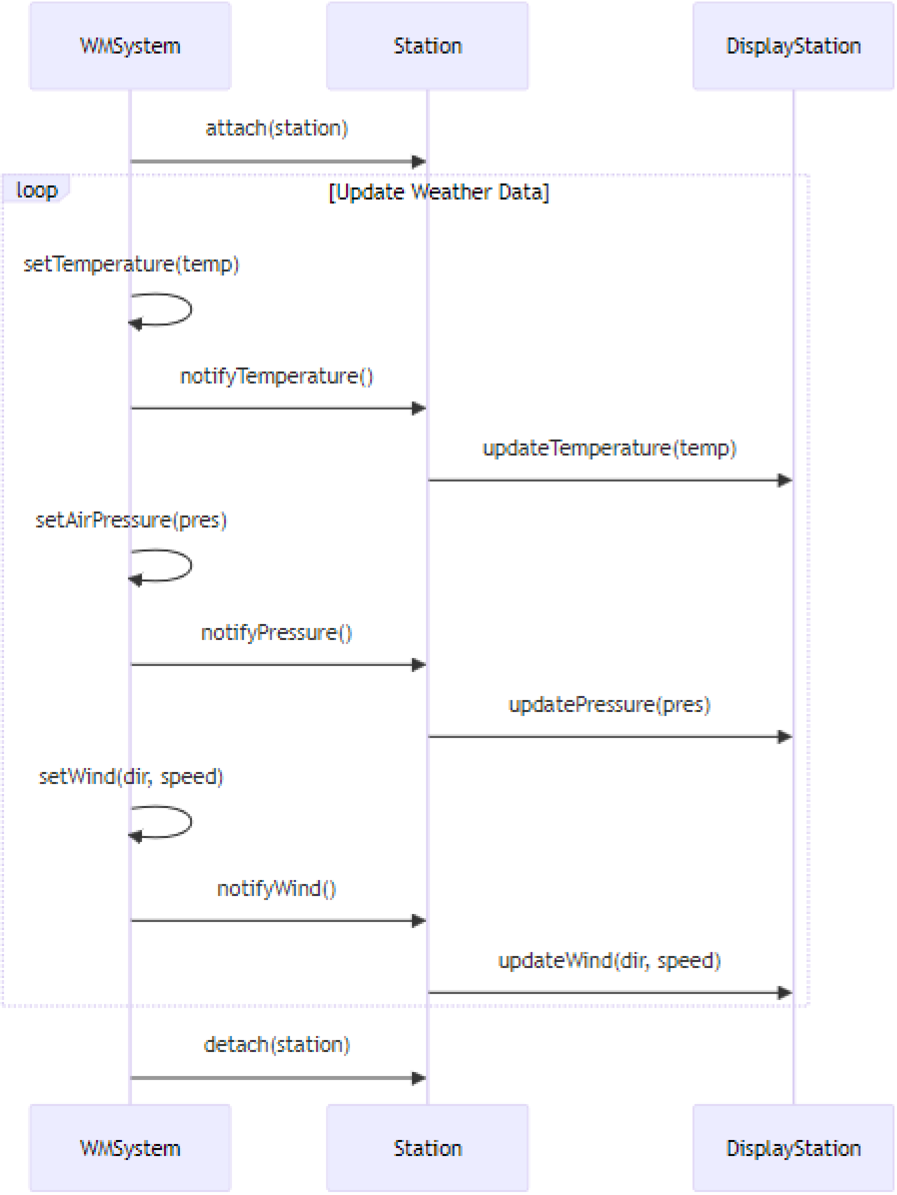
In this code, WMSystem is the “subject” and Station is the “observer”. When the state of WMSystem changes (via the setTemperature, setAirPressure, and setWind methods), it notifies all attached stations by calling their update methods. DisplayStation is a concrete observer that implements these methods to display the updated data.

This design allows you to add new types of stations (like LogStation) simply by creating a new subclass of Station and attaching instances of it to the WMSystem. The WMSystem doesn't need to know anything about these stations, other than that they implement the Station interface. This is in line with the Open-Closed Principle (OCP), which states that software entities should be open for extension, but closed for modification. The Observer pattern helps achieve this goal by decoupling the WMSystem from the stations that depend on its data.

Class Diagram:



Sequential Diagram:


WMSystem
Station
DisplayStation
attach(station)
loop
[Update Weather Data]
setTemperature(temp)
notifyTemperature()
updateTemperature(temp)
setAirPressure(pres)
notifyPressure()
updatePressure(pres)
setWind(dir, speed)
notifyWind()
updateWind(dir, speed)
detach(station)
WMSystem
Station
DisplayStation