CSE 443 Object Oriented Design, Fall 2023

Homework

Berru Lafcı

1901042681

1-) a)

Introduction:

Understanding of the problem:

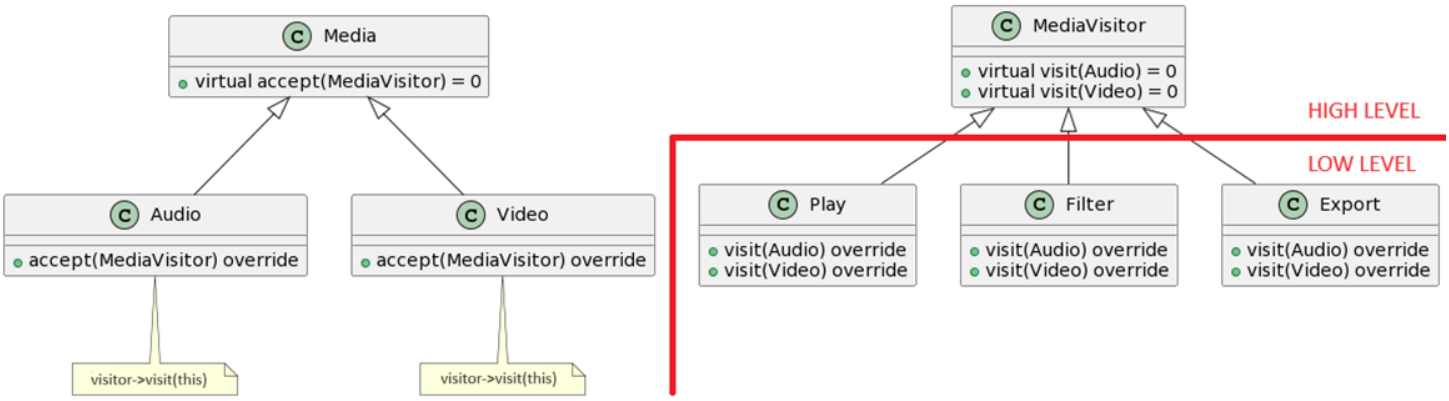
There is a base class which is media and there is a polymorphic behavior with the play() function which is pure virtual function. The goal is to add new operations (filter() and export()) to both Audio and Video classes.

Simply adding new functions directly to the existing Audio and Video classes violate the Open/Closed Principle. Because each addition would require modifications to all classes. This means it also modifies the base class which is very bad.

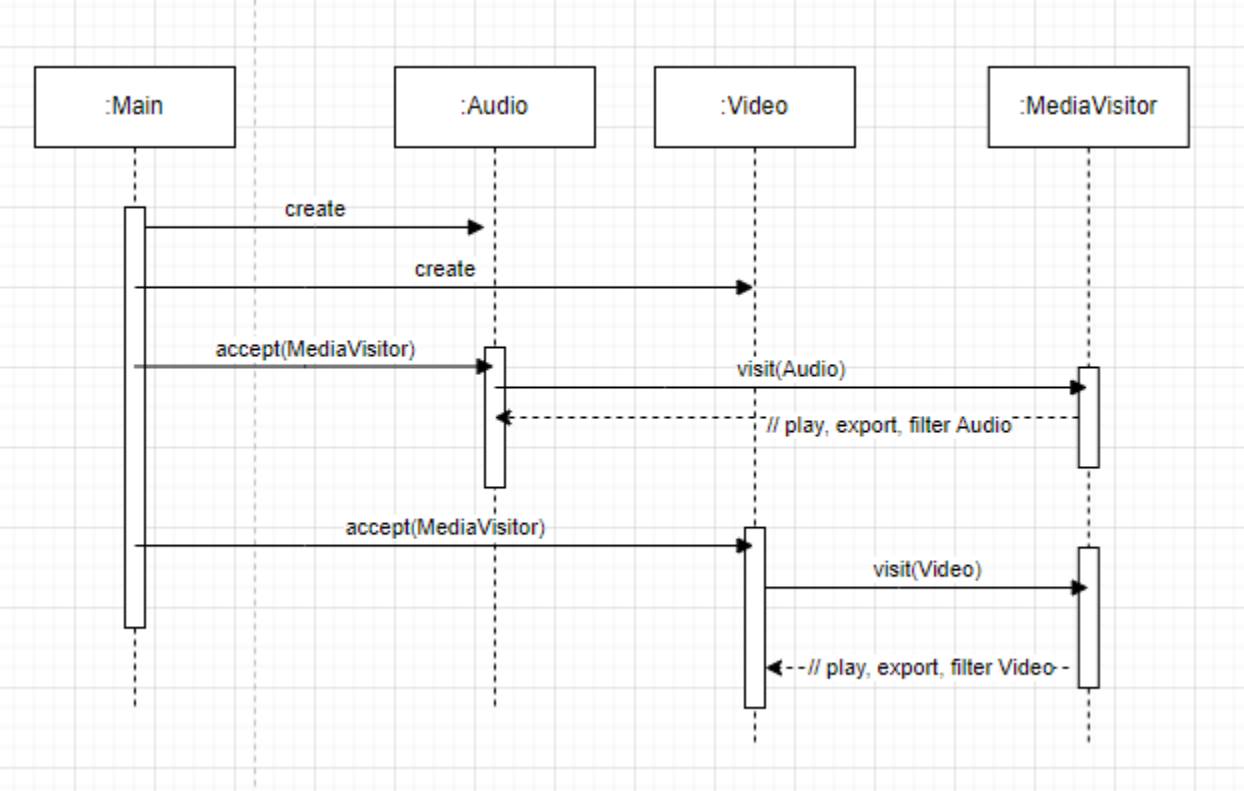
Required explanations:

To solve this, we should use Visitor design pattern. Visitor design pattern helps us add new operations to classes without changing their code directly. This keeps the original classes clean and simple, making it easier to understand, modify, and add new operations without causing a mess in the existing code.

Class Diagram: (with C++ codes and hierarchical boundaries)



Sequence Diagram:



b)

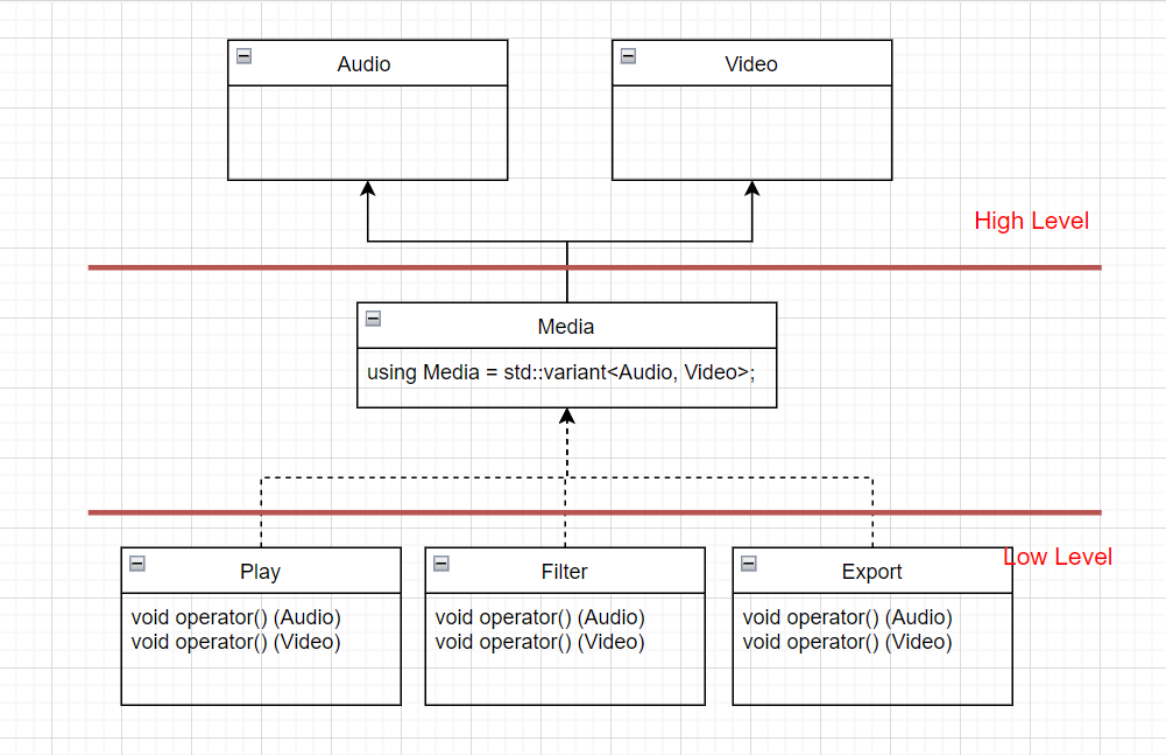
Understanding of the problem:

The problem with the original design is that it relies on polymorphism and virtual functions to implement the visitor pattern. While this approach allows adding new operations (like filter and export) without modifying the existing classes, it introduces runtime overhead due to virtual function calls.

Required explanations:

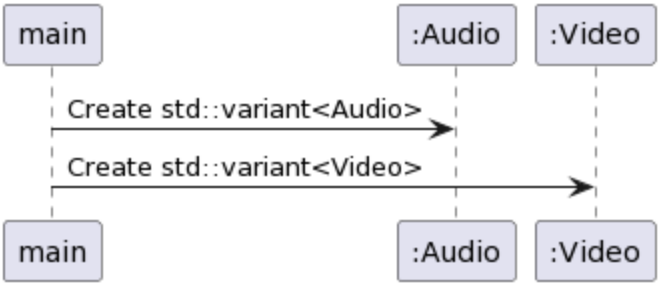
In value-based approach using **std::variant** provides a more efficient and extensible solution, solving the runtime overhead and reducing the need for a large number of visitor classes in comparison to the original polymorphic design.

Class Diagram: (with C++ codes and hierarchical boundaries)

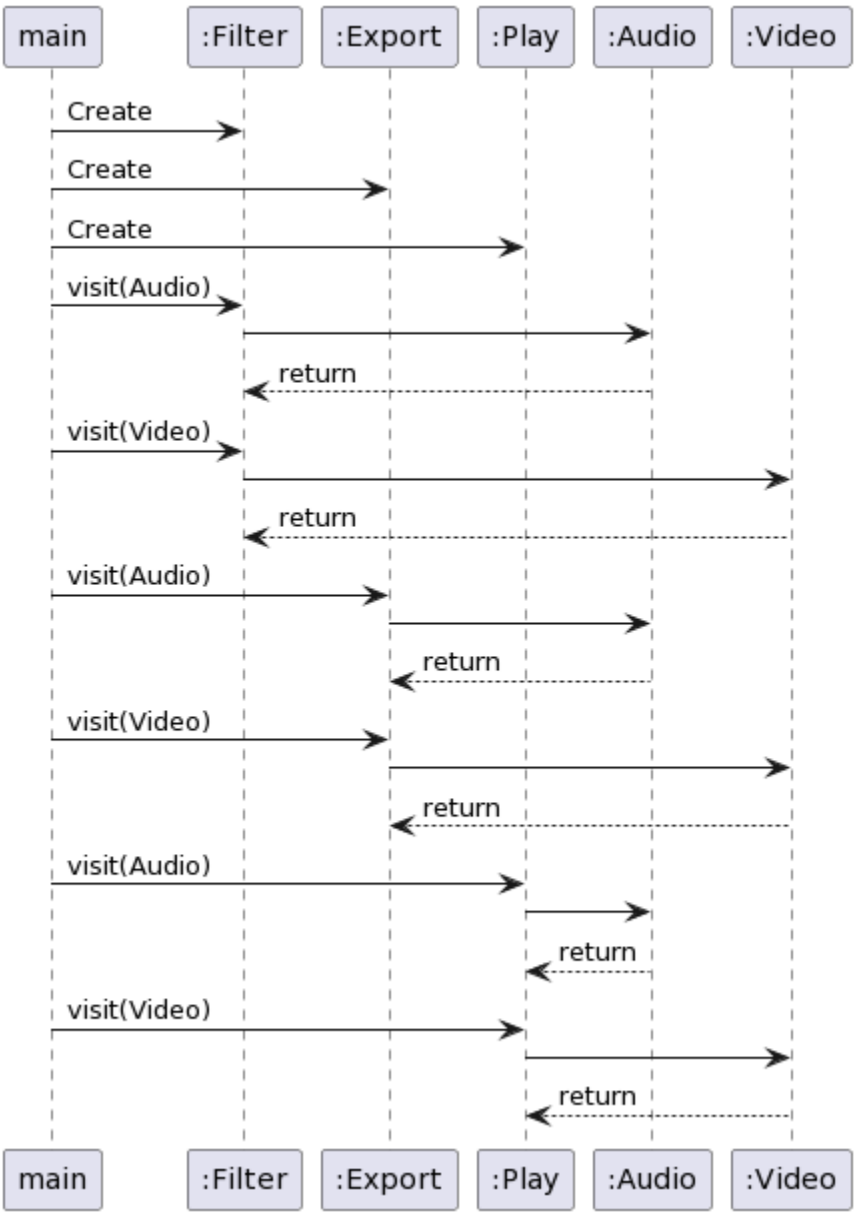


Sequence Diagram:

Initialization sequence:



Action sequence:



2-)

Introduction:

Understanding of the problem:

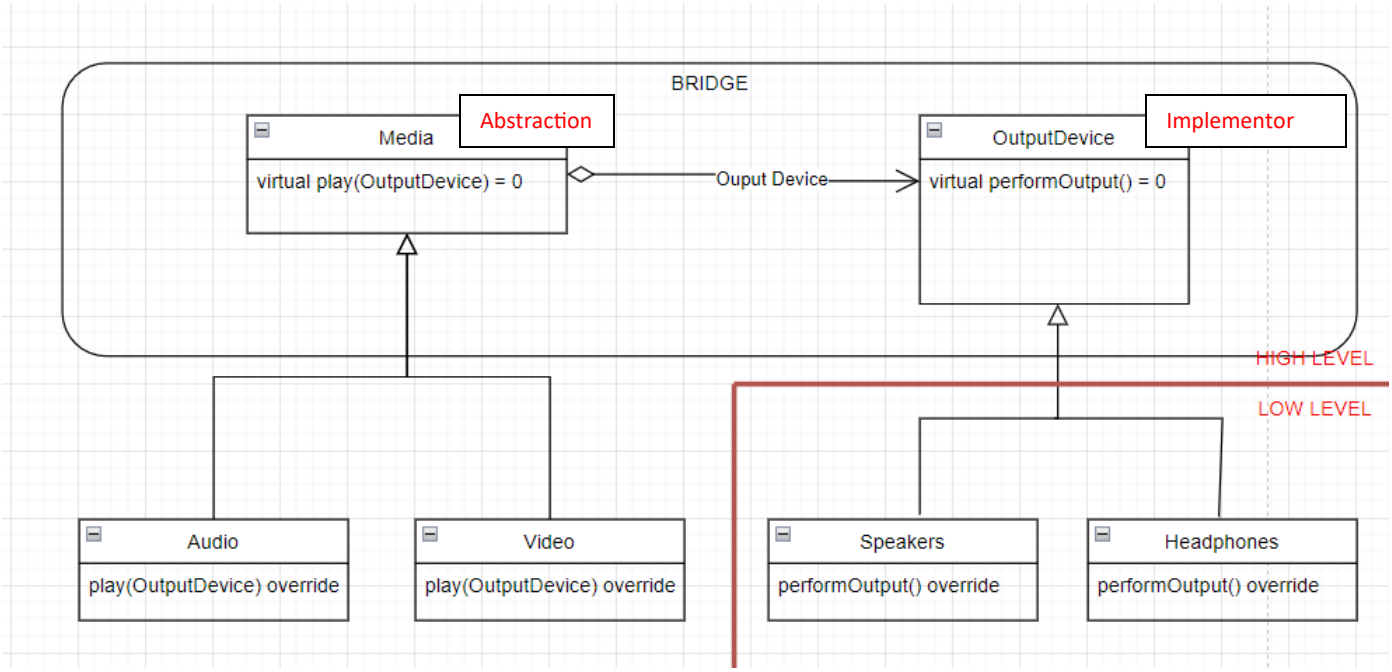
There is a base class which is media and there is a polymorphic behavior with the play() function which is pure virtual function. The goal is to add different output devices, specifically "Speakers" and "Headphones." To solve this, Bridge design pattern is recommended.

Required explanations:

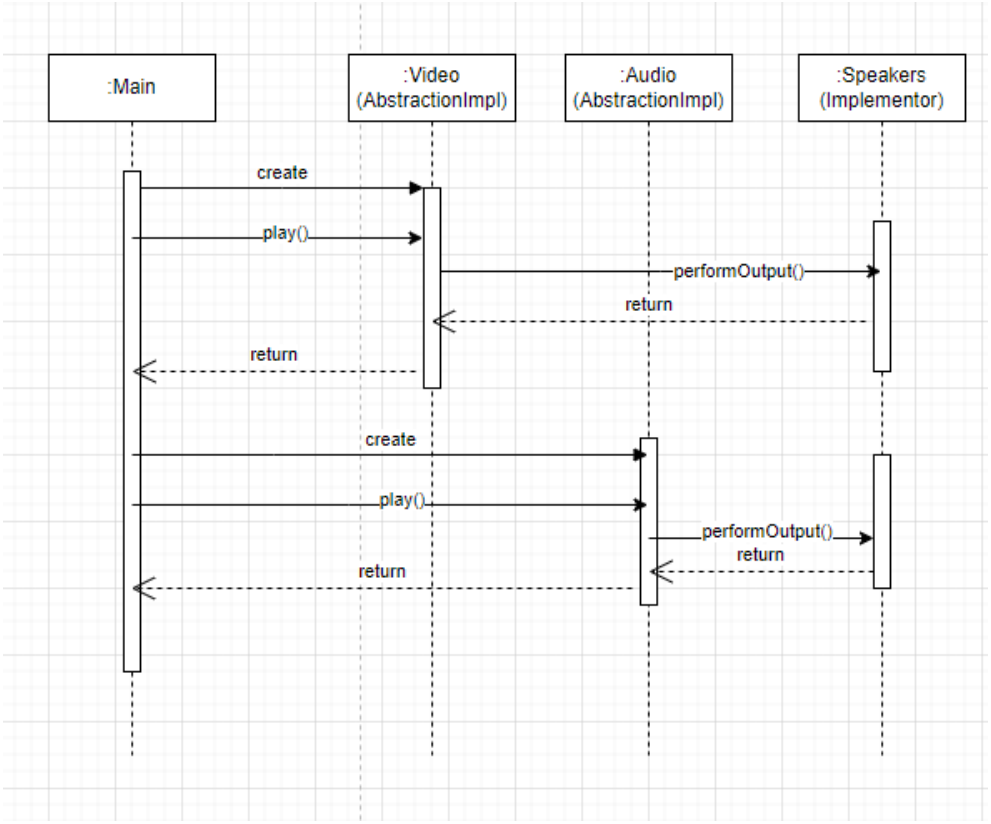
This solution uses the Bridge pattern to separate the abstraction (media files) from the implementation (output devices). The Media hierarchy focused on media-specific functionality, while the Output Device hierarchy handles output-specific operations.

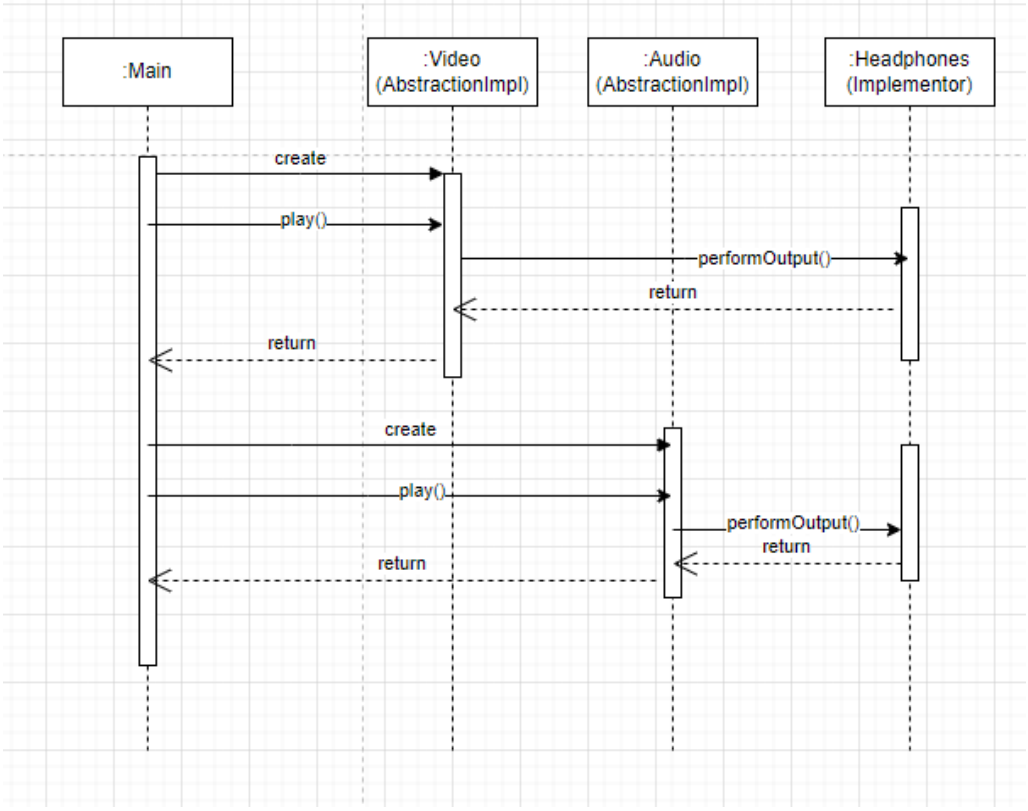
With this solution, new output devices can be added without modifying existing media classes, which is good for OCP. Also with this way, the Media class will focus solely on defining the behavior of media files without being concerned about the details of output devices which is good for SRP.

Class Diagram: (with C++ codes and hierarchical boundaries)



Sequence Diagram:





3-)

Introduction:

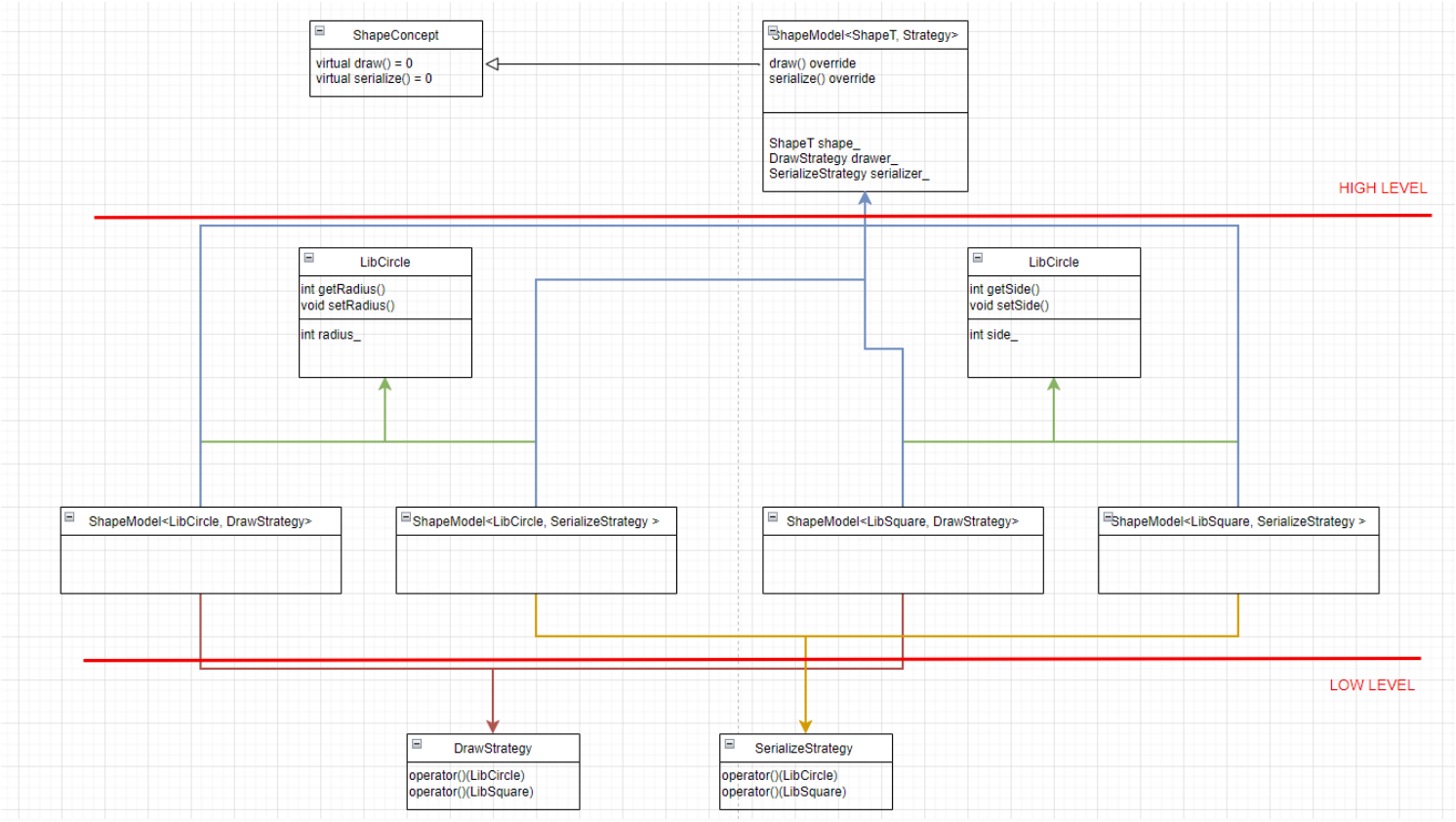
Understanding of the problem:

The problem has two shape classes, namely "LibCircle" and "LibSquare." Our goal is to implement operations on these shapes using an external polymorphism design pattern. This design pattern allows us to define a common interface (base class) and have multiple derived classes (specific shapes).

Required explanations:

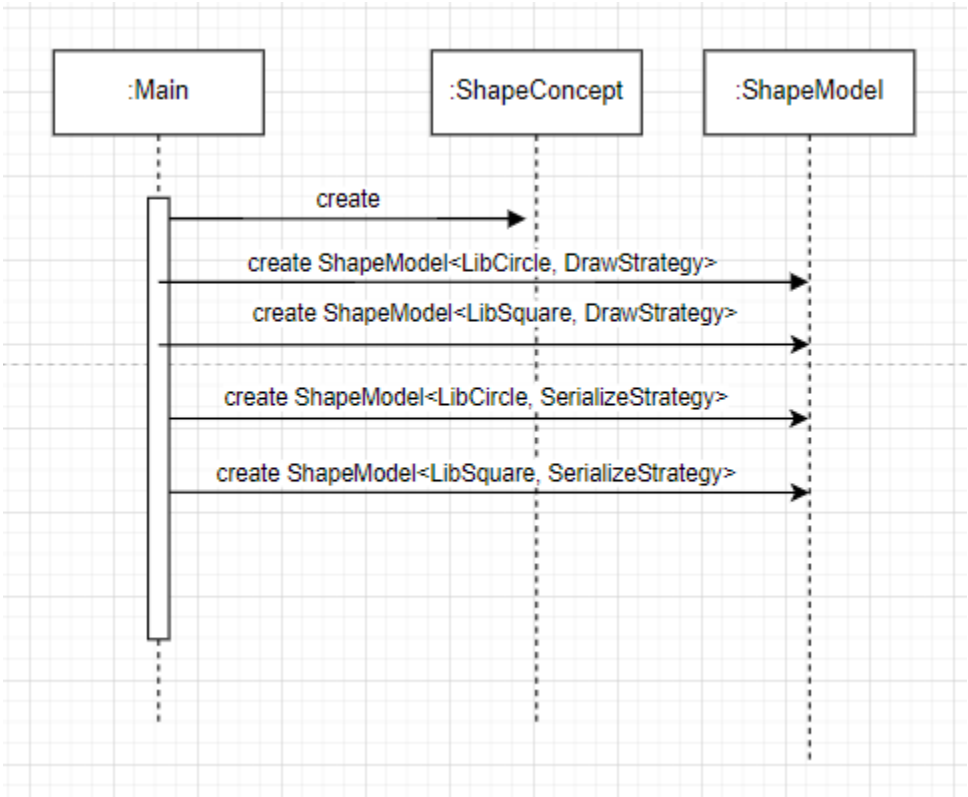
To solve this problem, we are required to create our own shape hierarchy with an abstract base class named "Shape" and class templates ("MyCircle" and "MySquare") derived from it. These classes should handle polymorphic operations such as drawing and serializing shapes.

Class Diagram: (with C++ codes and hierarchical boundaries

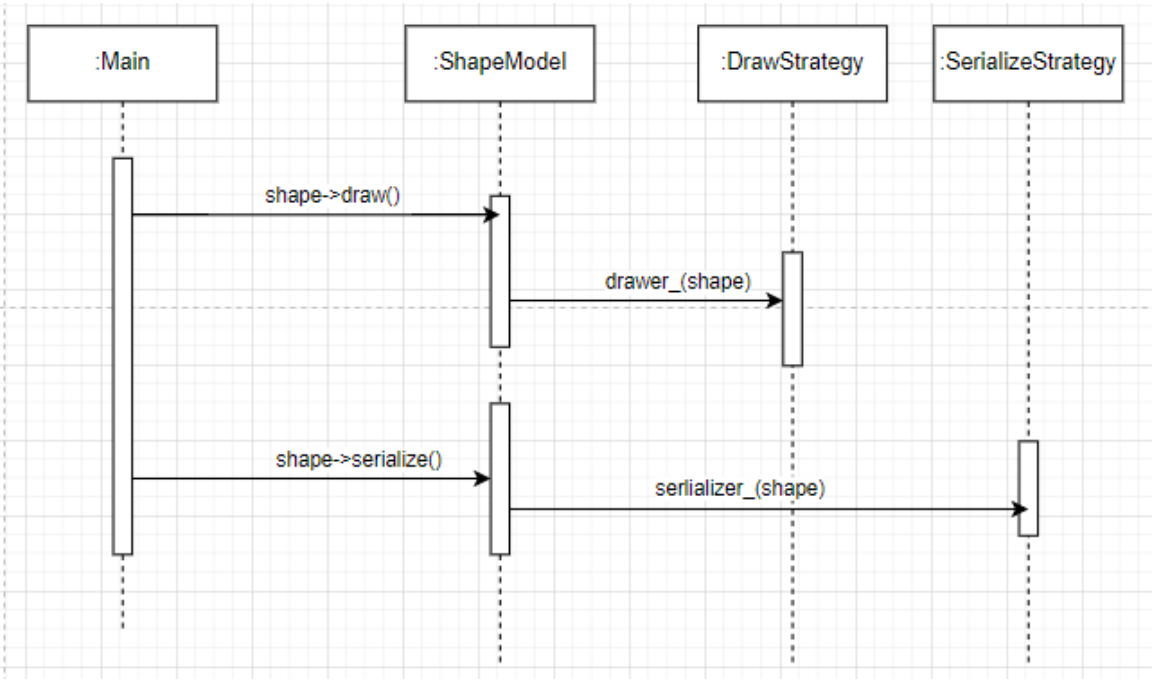


Sequence Diagram:

Initialization sequence:



Action sequence:



4-)

Introduction:

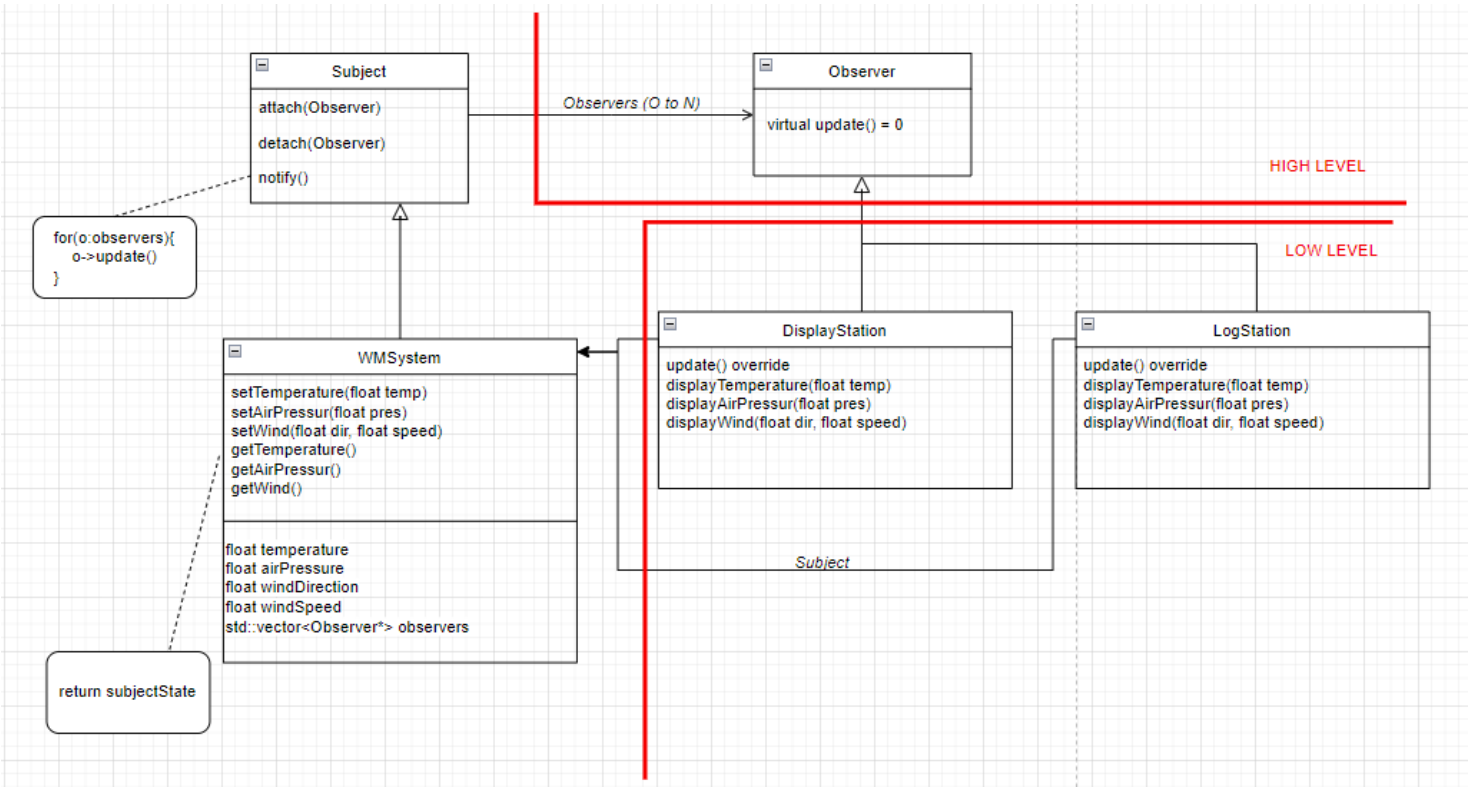
Understanding of the problem:

The problem is to implement a system that allows various stations, such as the existing DisplayStation and a new LogStation, to receive updates whenever the weather sensor readings change. The current design lacks this capability, and we will solve it with Observer pattern.

Required explanations:

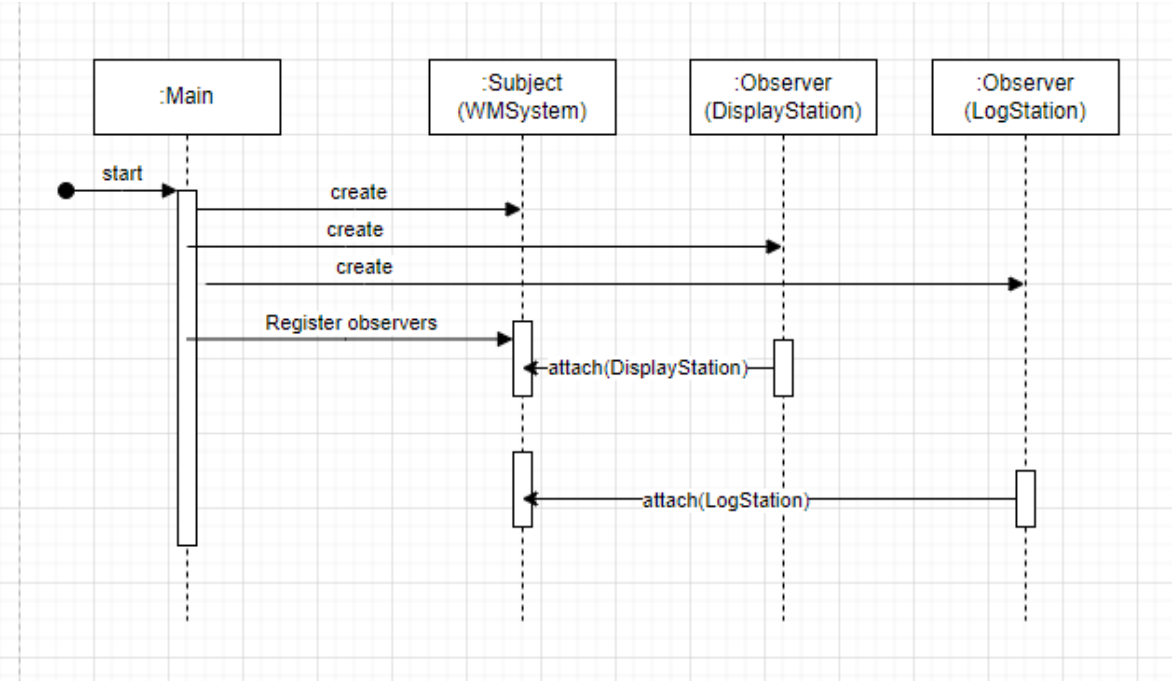
The Observer pattern decouples the subject (WMSystem) from its observers (DisplayStation, LogStation), ensuring that adding new stations in the future won't require changes to the WMSystem class. The WMSystem class will open for extension but closed for modification, meaning that new observer stations can be added without modifying the existing WMSystem class.

Class Diagram: (with C++ codes and hierarchical boundaries



Sequence Diagram:

Initialization sequence:



Action sequence: