# CSE 443 OBJECT ORIENTED ANALYSIS AND DESIGN HOMEWORK 1

1901042629

Çağrı ÇAYCI

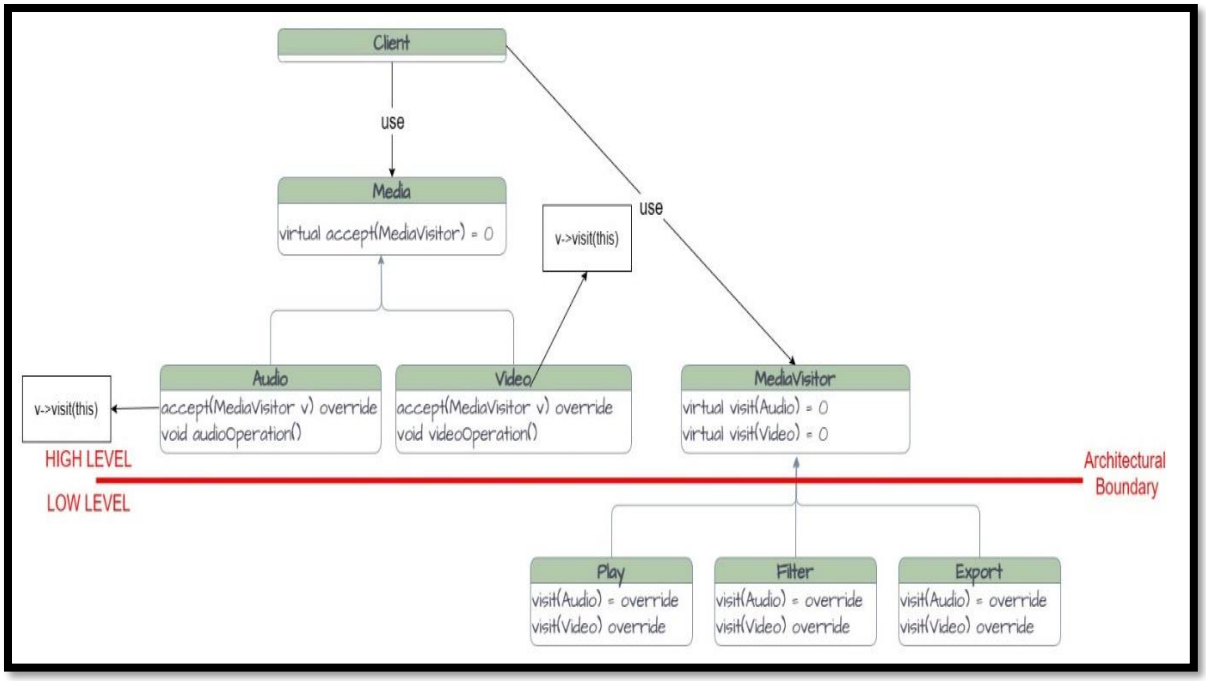
# CHAPTER 1

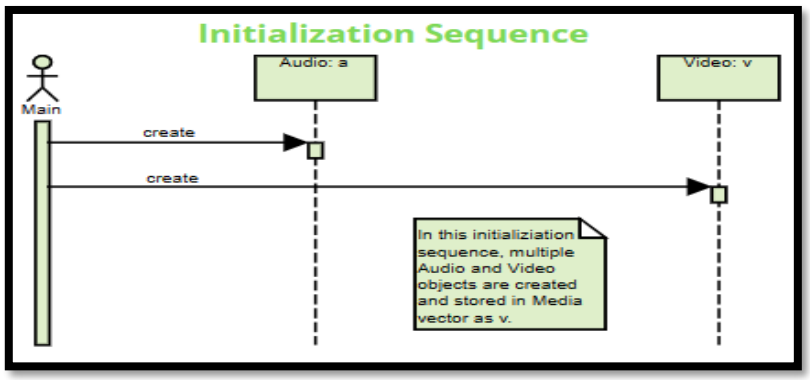
```
class Media {
public:
  // Constructors, destructor,
  virtual void play() = 0;
private:
  // Private members
};
```

```
class Audio : public Media {
public:
  // Constructors, destructor,
  void play() override {
    // Audio play code
  };
private:
  // Private members
};
class Video : public Media {
public:
  // Constructors, destructor,
  void play() override {
    // Video play code
  };
private:
  // Private members
};
```

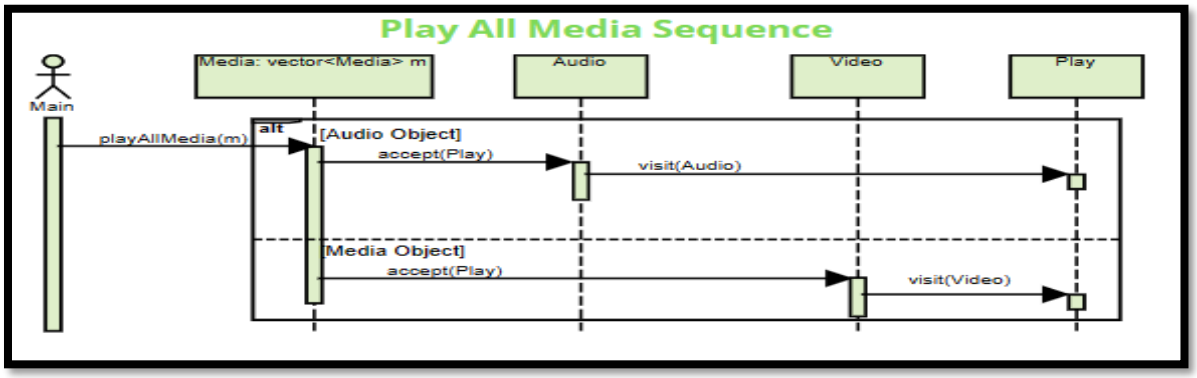
This is an object-oriented design for media types. Media class serves as a base class. Audio and Video classes serve as subclasses of Media class. This design has some advantages such as being cleaner and the ability to add new types. For example, a new type can be added by creating a class file for it without changing any existing code and class hierarchy. However, every advantage has its disadvantage. This object-oriented design works well for now, but it becomes challenging when we need to add new operations. The issue lies in the difficulty of incorporating new operations without making significant changes. For instance, if we want to introduce an export feature for our Media objects, we must modify the Media class by adding a pure virtual export function and then override it in the subclasses. This forces us to recompile and retest all the classes, which can be cumbersome.

To avoid backwards of object-oriented design, we use Visitor Design Pattern. In this design pattern, we create a visitor class (MediaVisitor) for our base class (Media) with pure virtual visit function and inherits it with our operations (Play, Filter, Export). Inherited visitor classes override visit function. We also need to add a pure virtual accept function to Media class and override the function inside its children. After we complete our design, our design is open to extension for operation. For example, adding a Serialize operation to Media is only creating a serialize class which is inherited from MediaVisitor.



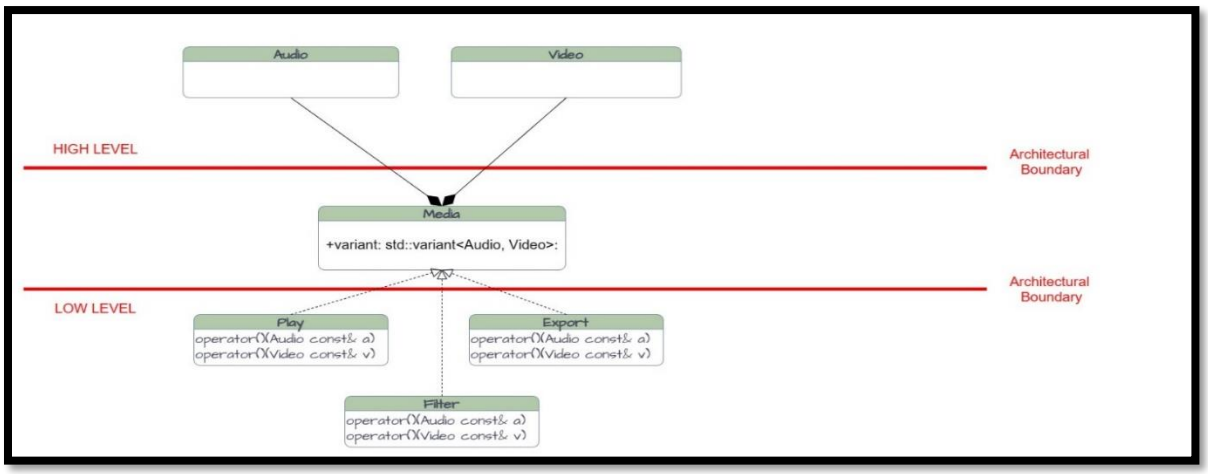




The design begins with the main function, where several Audio and Video objects are created. These objects are then stored in a Media Vector for future use.

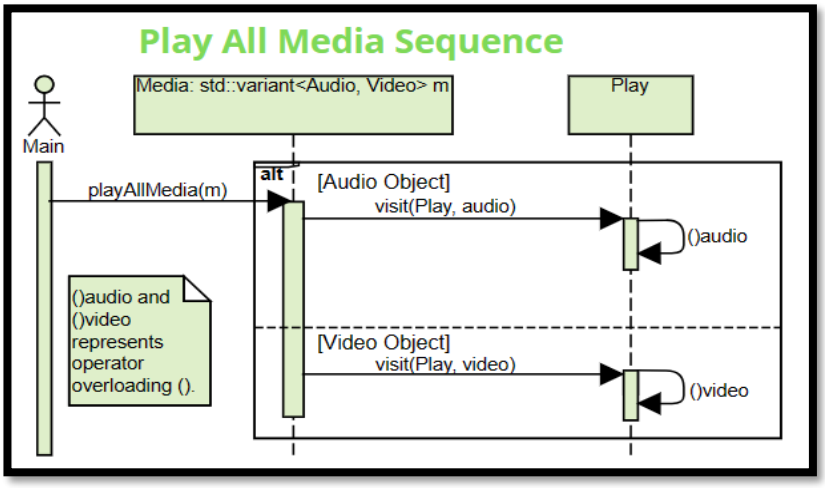



To play all the media objects, regardless of whether they are video or audio, the playAllMedia function is invoked with a vector containing Media objects. For each object in the vector, the accept function is triggered, passing the Play class as an argument. Both the Audio::accept and Media::accept functions then invoke the visit function of the Play class, providing the corresponding media type, such as Audio or Video. This approach allows us to play all the media objects seamlessly.

Another way of solving the extensibility problem of Object-oriented design is using C++17 features. After the C++17 was introduced, it turns out that std::variant can be used to solve this problem. A variant represents one of several alternative types. For example, std::variant<int, double, char> can contain one of these types: integer, double or char.




Like the Visitor Design Pattern, we can add new functionalities to our Media objects easily value based approach. Because, adding new functionalities is equal adding a new class without changing, recompiling, retesting any other classes. However, adding new types is also a difficult task in this design too. Because adding a new type results in changing Media, Play, Filter, Export classes.

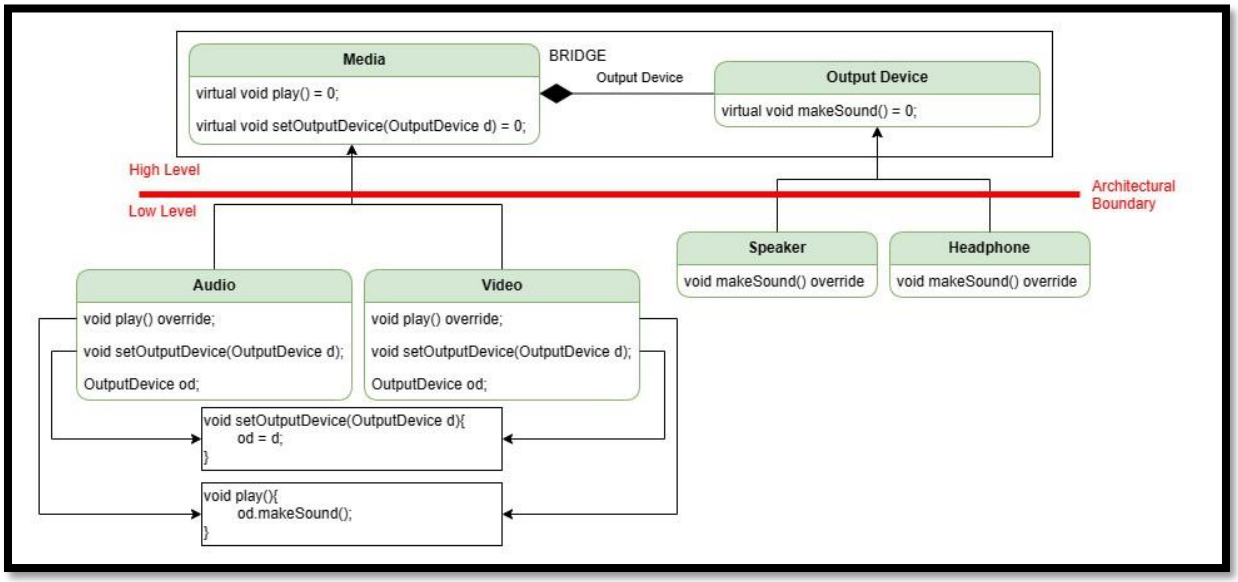



This is the sequence diagram of value-based approach to implement Visitor Design pattern. Contrary to Visitor Design Pattern, media objects is kept as std::variant<Audio, Video> in this approach.

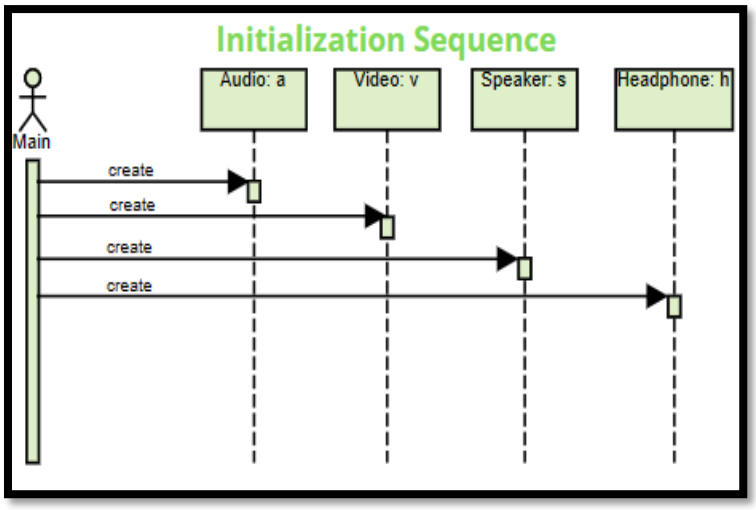
In the main function, there is a call to the playAllMedia function. Inside the playAllMedia function, the visit function is called, passing the Play class and the specific media object as arguments. Within the Play class, the operator() overload is triggered with the media object as an argument, allowing the media to be played.

# CHAPTER 2

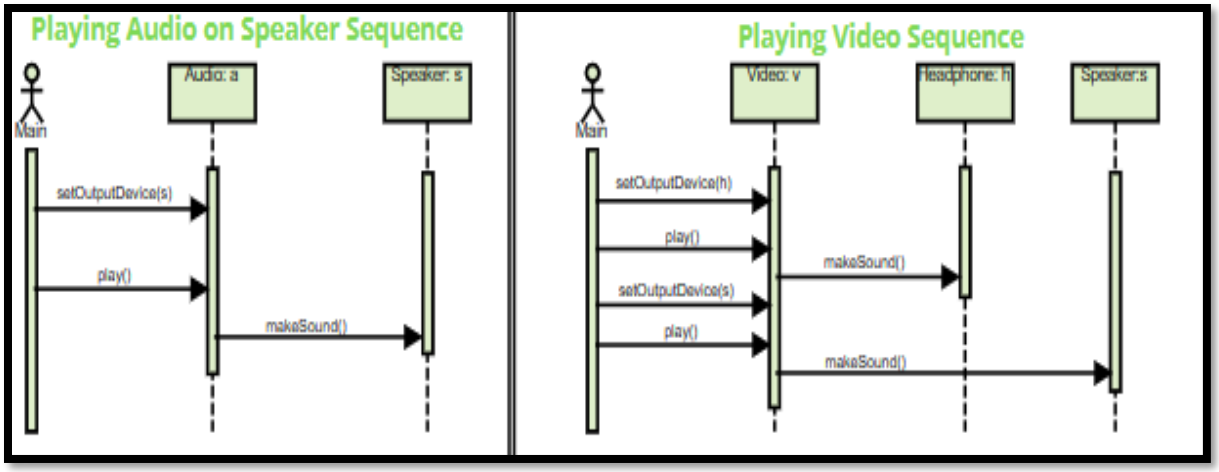
In this question, we want to use Bridge Design Pattern to isolate physical dependencies from implementation details. In this way, we don't have to expose our Media Files and Output Devices details. Also, using Bridge Design Pattern does not affect the simplicity of adding new derived types. For example, we can add new Media or OutputDevice classes without changing any existing code.




This is the class diagram of Bridge Design Pattern for this specific question. Media class has pure virtual play function, and it plays the media by making sound with Output Device. Media and Output Device classes are at a high level of Architectural Boundary whereas Audio, Video, Speaker, and Headphone are at low level. Because Media and Output Device classes do not have any implementation details.



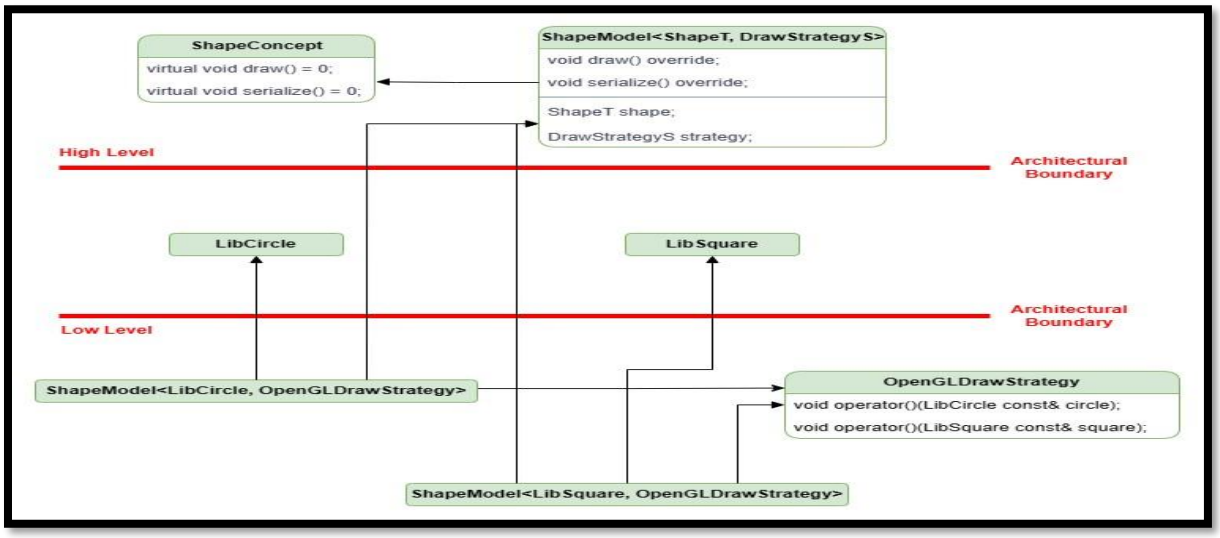

At the beginning, in the main function, we make Media objects - Audio and Video, and Output Device objects - Speaker and Headphone. This is the initialization sequence of this design.




To play an Audio object on the Speaker, the main function needs to configure the output device for the Audio object. Afterward, the main function invokes the play function of the Audio object. The Audio object, in turn, triggers the makeSound function of the Speaker. Additionally, this design offers another capability: we can switch the output device during the program execution.

# CHAPTER 3

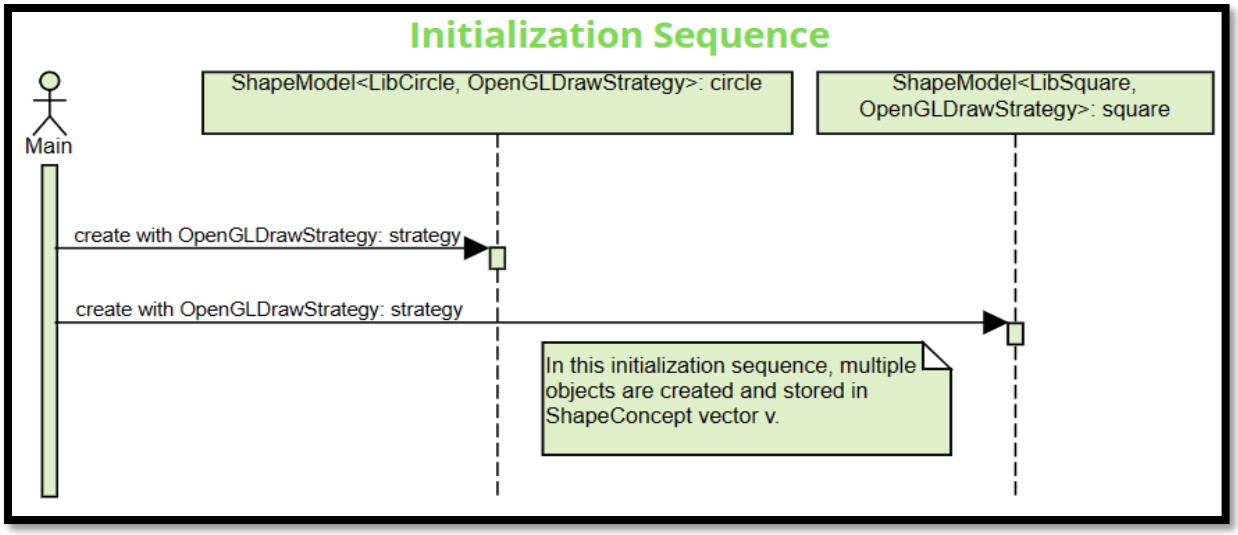
The use of external polymorphism design pattern in this scenario, involving the classes LibCircle and LibSquare, is valuable for its ability to enhance flexibility, extensibility, and maintainability of the codebase. By introducing an abstract base class ShapeConcept and creating specific shape classes (Circle and Square) derived from it, a clear separation of concerns is achieved. Each shape class encapsulates its drawing and serialization strategies, making it easier to add new shapes without altering existing code. This adheres to the open/closed principle, allowing for extension without modification. Furthermore, the pattern promotes code reusability, testability, and modularity, as each strategy can be independently tested and reused across different shapes. Overall, external polymorphism provides a robust framework for managing the behavior of diverse shapes through a common interface.



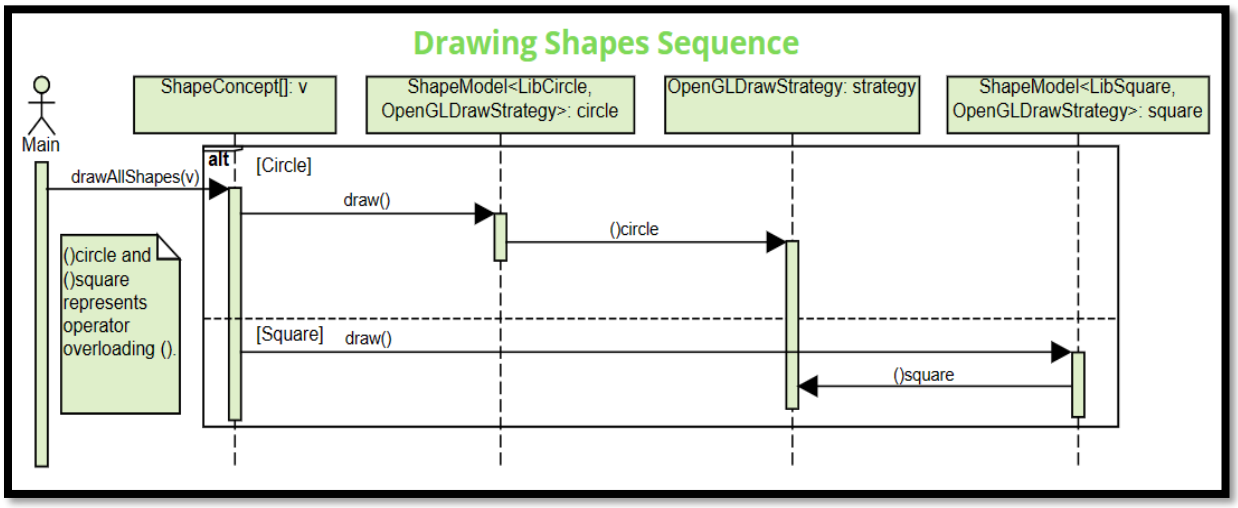

This design uses the External Polymorphism Design Pattern. At the top level, we have ShapeConcept and ShapeModel. ShapeConcept defines pure virtual functions for drawing and serializing, providing a high-level abstraction. ShapeModel is a template class that takes Shape and Strategy as parameters.

Moving to the middle level, we find LibCircle and LibSquare. These classes represent geometric entities and completely non-polymorphic.

At the lowest level of architecture, there are strategy classes like OpenGLDrawStrategy, handling specific drawing or serializing implementations. This hierarchical structure enables flexibility and external polymorphism within the design pattern.




This sequence diagram outlines the initialization process. We start by creating objects using the template ShapeModel class. During the creation of ShapeModel objects, we specify the shape and strategy. These objects are then stored in a vector of ShapeConcept.




To draw all shapes, the main function initiates the drawing process by calling the drawAllShapes function with the ShapeConcept vector as an argument. Inside drawAllShapes, each object within the ShapeConcept vector is iterated over, and the draw function is invoked for each object. The class diagram indicates that the ShapeModel class retains information about the shape and strategy. Additionally, the strategy classes have an overloaded function call operator "()". Consequently, the draw function within ShapeModel calls the strategy with the shape: “void ShapeModel::draw(){ strategy(shape); }”. This mechanism ensures that the draw function effectively invokes the appropriate strategy for each shape, utilizing the overloaded function call operator.

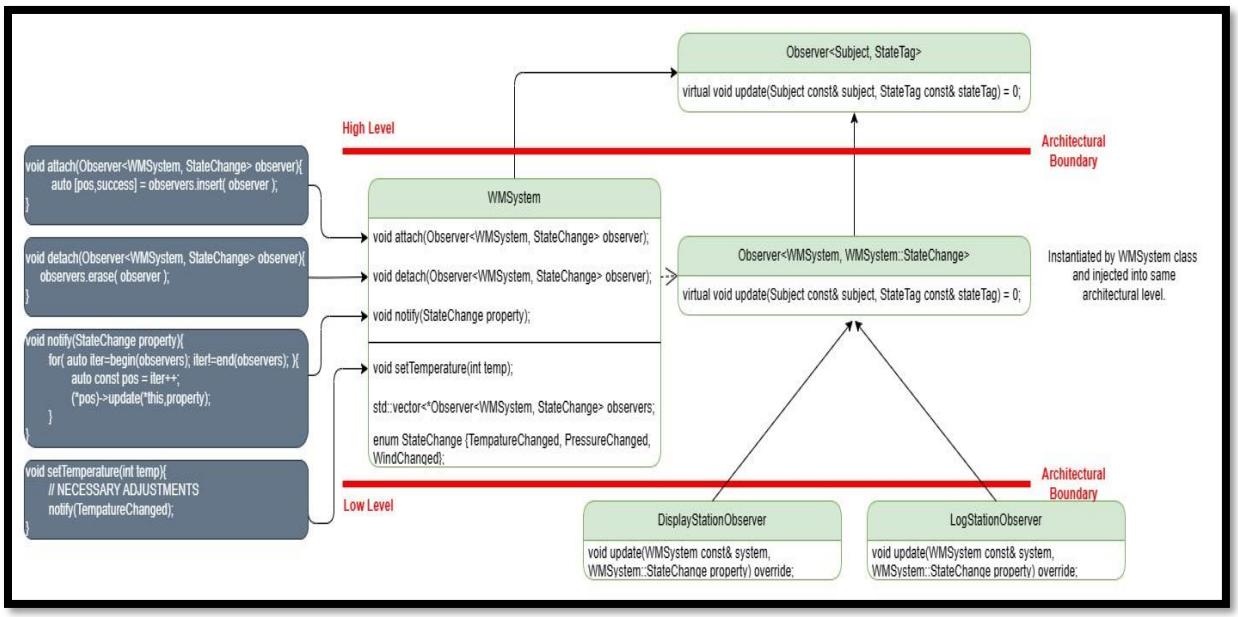
# CHAPTER 4

In this question, if we persist in adopting the current system design approach, the necessity arises to update the WMSystem class for every newly added station.

```
void WMSystem::setTemperature(float temp){
  temperature = temp;
  displayStation.displayTemperature(temperature)
}
```

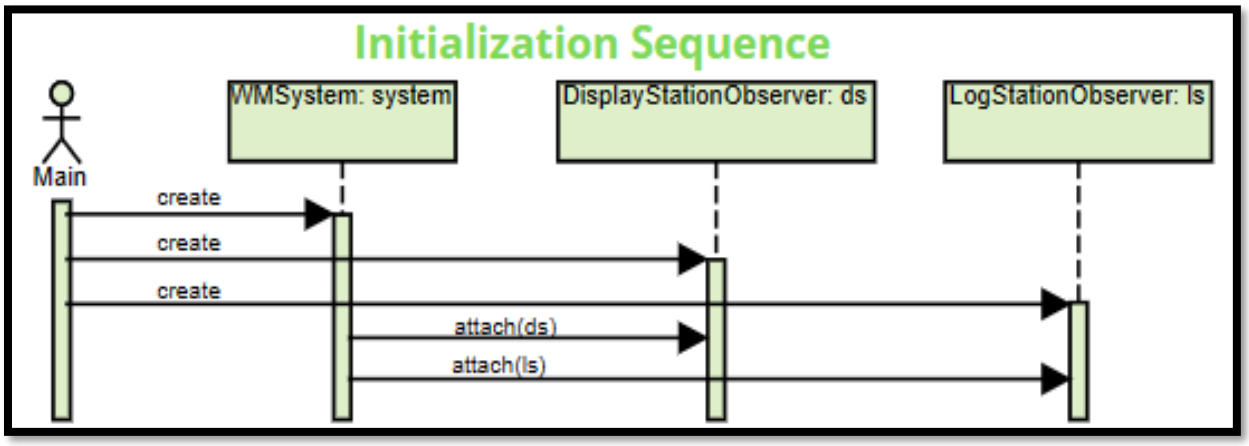
```
void WMSystem::setTemperature(float temp){
  temperature = temp;
  displayStation.displayTemperature(temperature)
  logStation.logTemperature(temperature)
  ...
}
```

The images above clearly demonstrate that whenever a new station is introduced to the system, it is necessary to adjust the WMSystem functions to keep the stations updated. As we know, updating a class results in recompiling and retesting the class. In order to overcome the drawbacks of this design approach, the Observer Design Pattern should be employed. This behavioral design pattern establishes a one-to-many relationship between objects, ensuring that when one object undergoes a state change, all its dependents receive automatic notifications and updates. Consequently, employing this pattern allows for the seamless integration of new stations without necessitating modifications to the WMSystem class.

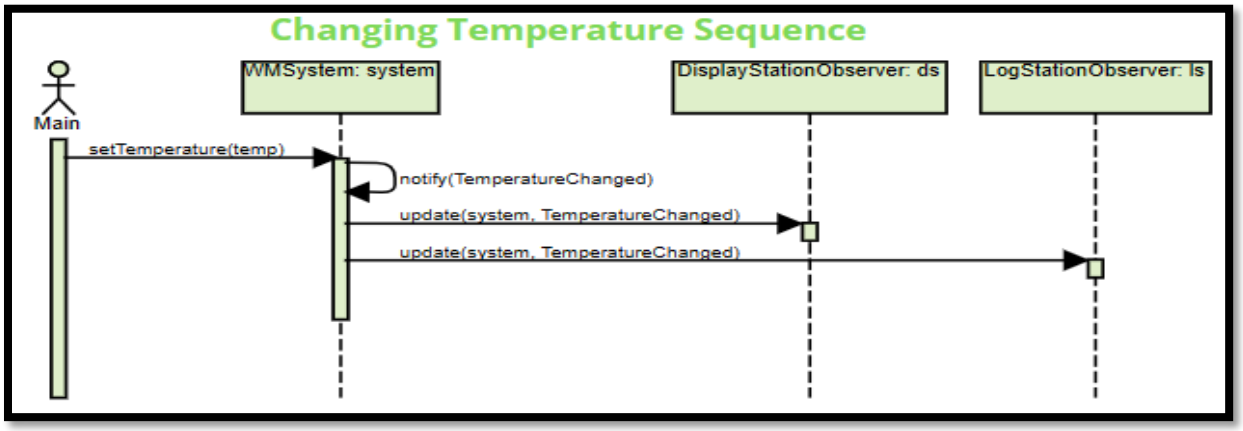



The Observer Design Pattern class diagram presented here addresses a specific problem. By employing this design, modifications, or recompilation of WMSystem are unnecessary. Clients can seamlessly incorporate new Observers into the system using the WMSystem::attach function.

Within WMSystem, a collection of observers is managed through a vector, offering essential functionalities like adding a new observer (attach function), removing an existing observer (detach function), and notifying observers (notify function). This enables the addition, removal, or notification of observers, such as PressureObserver, TemperatureObserver, and WindObserver, all of which inherit from the Observer base class.




This is how the design starts: In the main function, we create WMSystem and observers. Then, WMSystem attaches these observers. Note that WMSystem stores the observers as a vector in its data field.




The process of changing the temperature is described in a sequence diagram. The main function sets the temperature using an object called WMSystem. When the WMSystem::setTemperature function is called, it notifies observers with the tag "TemperatureChanged" using the WMSystem::notify function. The notify function calls the update function for the attached observers, which include DisplayStationObserver and LogStationObserver.

The sequence diagram of other processes such as changing wind or air pressure follows the same path. So, their sequence diagrams are not added.