CSE 443 OBJECT ORIENTED ANALYSIS AND DESIGN

FALL 2023

HOMEWORK

MERT GÜRŞİMŞİR

1901042646

Contents

QUESTION 1-A

INTRODUCTION

Here, we have base class “Media” with a pure virtual function “play()” and then 2 derived classes “Audio” and “Video”. These classes inherit from “Media” and provide their own implementation of “play()” function. When we want to add new operations like “filter()” and “export()”, we violate the Single Responsibility and Open-Closed Principles of SOLID principles. The Open-Closed principle says that a class should be open to extension but closed to modification. In our current design, to add new operations, we would need to modify the base class “Media” and all the derived classes.

Modifying the base class means that all derived classes must adhere to base class and they also need to be changed. This could break existing code that relies on current design. New functionalities may call 3rd party applications and these may break the code.

Every time we want to add a new operation, we have to change the base class and derived classes. sThis becomes error-prone. Also the design becomes less flexible because it doesn’t easily accommodate new operations without requiring modifications to existing code.

Adding unrelated operations to the base class may also violate the single responsibility principle as I have mentioned. This principle says that a class should only have 1 task to do. The “Media” class should be responsible for defining the common behavior related to media playback, not for all possible operations.

Solution would be to use the Visitor Design Pattern. This pattern separates the algorithms or behaviors from the objects on which they operate. Each derived class can accept a visitor, and the visitor can perform the necessary operations.

Visitor Design Pattern lets us add new operations without modifying the existing structure. We can refactor the design to add the “filter()” and “export()” operations as follows:

CODE SNIPPET

CLASSES	VISITOR CLASSES

class Media {
public:
virtual void accept(MediaVisitor& visitor) = 0;
};
	class MediaVisitor {
	public:
	virtual void visit(Audio& audio) = 0;
	virtual void visit(Video& video) = 0;
	};

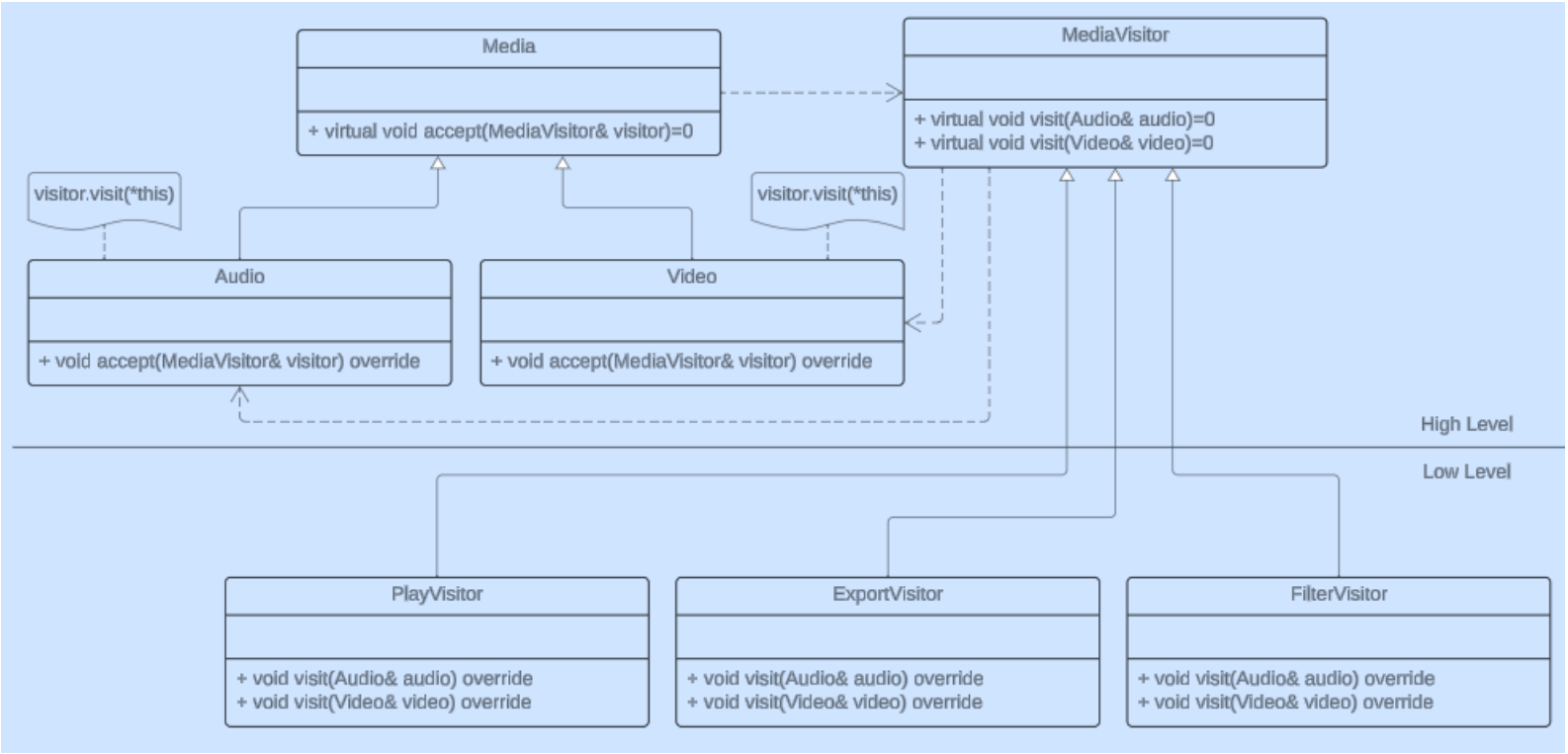
class Audio : public Media {
public:
void accept(MediaVisitor& visitor) override {
visitor.visit(*this);
}
};
	class PlayVisitor : public MediaVisitor {
	public:
	void visit(Audio& audio) override {
	//Audio play code
	}
	
	void visit(Video& video) override {
	//Video play code
	}
	};

class Video : public Media {
public:
void accept(MediaVisitor& visitor) override {
visitor.visit(*this);
}
};
	class FilterVisitor : public MediaVisitor {
	public:
	void visit(Audio& audio) override {
	//Filter audio
	}
	
	void visit(Video& video) override {
	//Filter video
	}
	};

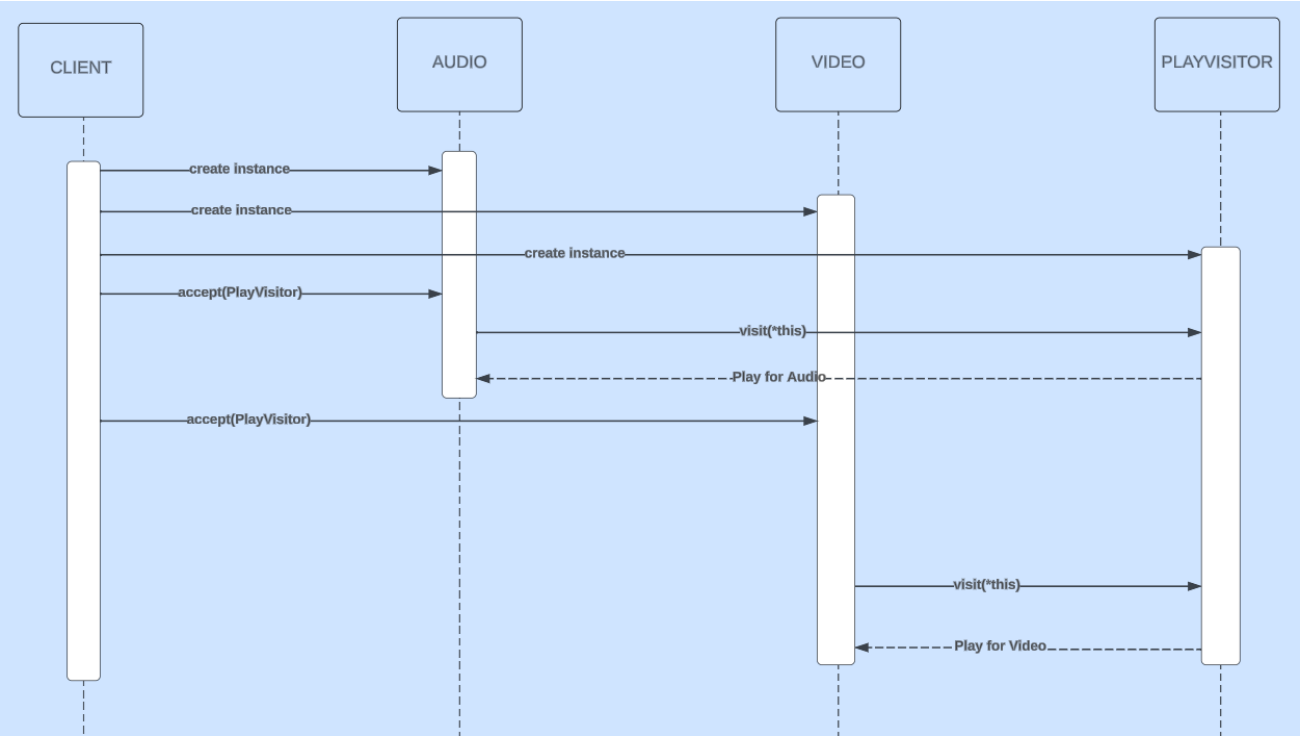
	class ExportVisitor : public MediaVisitor {
	public:
	void visit(Audio& audio) override {
	//Export audio
	}
	
	void visit(Video& video) override {
	//Export video
	}
	};

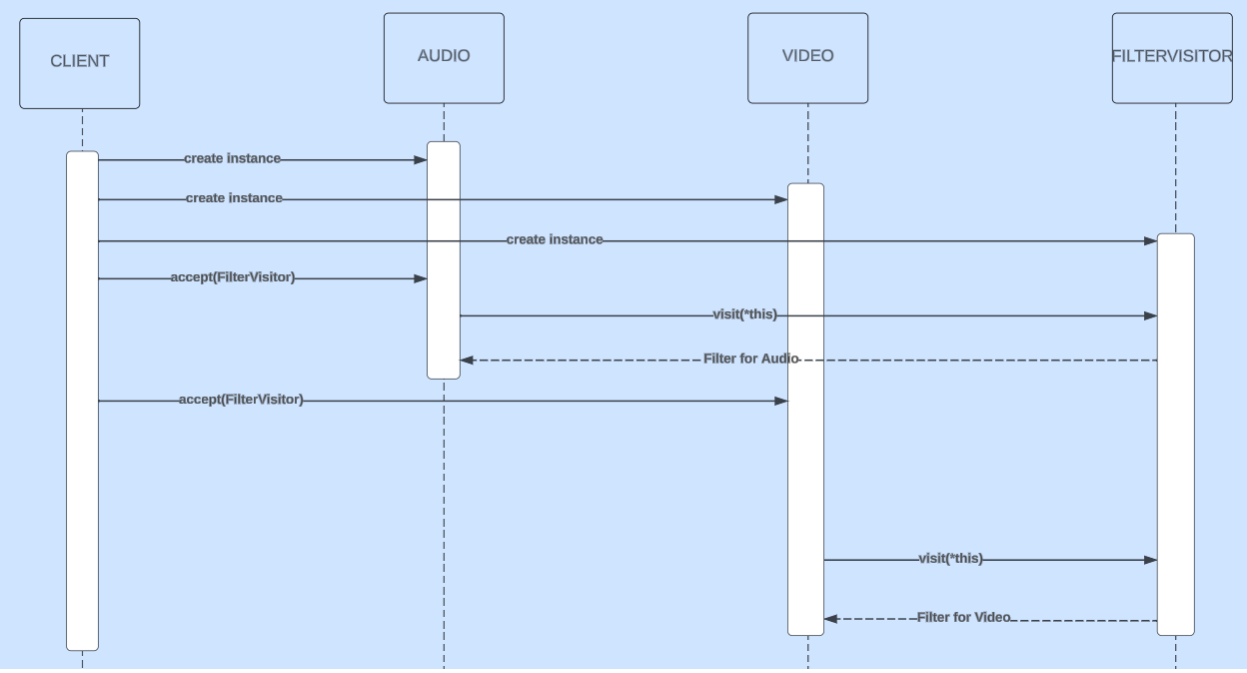
In the main function, instances of Audio, Video, PlayVisitor, FilterVisitor, ExportVisitor are created. Then, for example, to play audio media we need to call the accept method of the “Audio” with the PlayVisitor parameter.

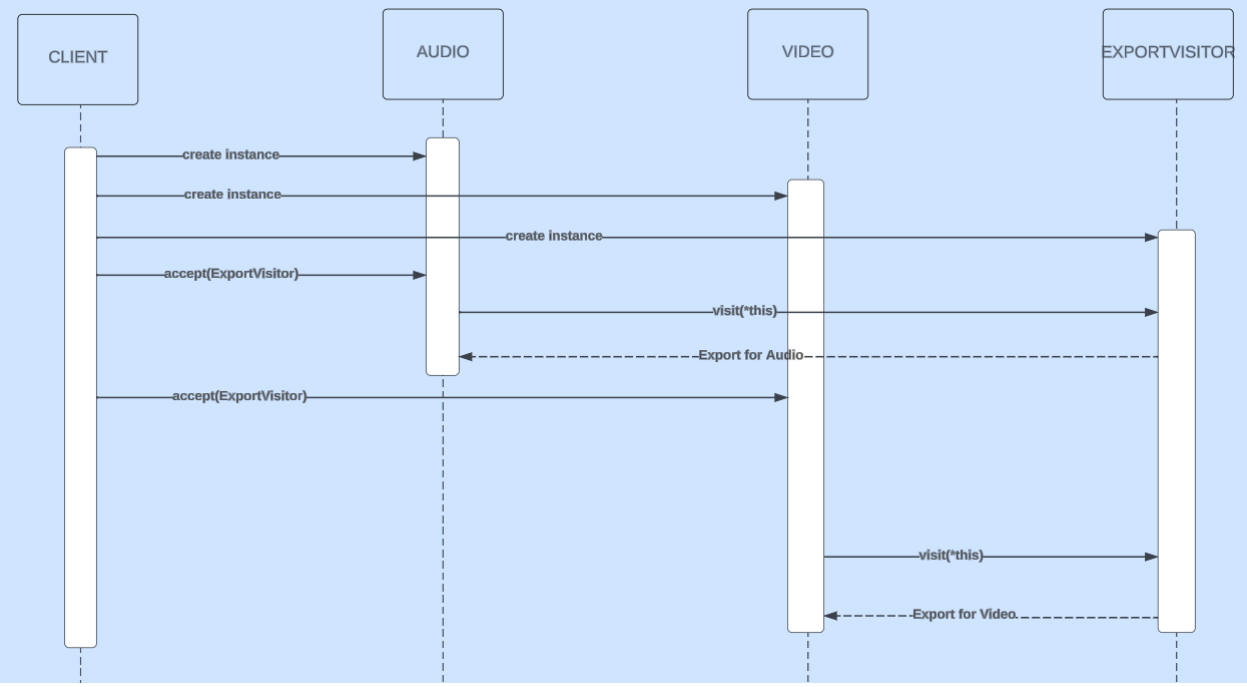
CLASS DIAGRAM



SEQUENCE DIAGRAMS




CLIENT
AUDIO
VIDEO
FILTERVISITOR
create instance
create instance
create instance
accept(FilterVisitor)
visit(*this)
Filter for Audio
accept(FilterVisitor)
visit(*this)
Filter for Video


CLIENT
AUDIO
VIDEO
EXPORTVISITOR
create instance
create instance
create instance
accept(ExportVisitor)
visit(*this)
Export for Audio
accept(ExportVisitor)
visit(*this)
Export for Video

QUESTION 1 – B

INTRODUCTION

In value-based approach, we can use a combination of variant types or polymorphism to represent different media types, and operations can be performed directly on these types. Here, “std::variant” is going to be used because if we do directly do on the virtual functions, overhead will be too much. This approach involves representing objects and their behavior through their inherent data types rather than relying on polymorphism or inheritance. The class hierarchy is simplified to focus on the specific data and behavior of each media type. There's no need for virtual functions or inheritance, which can lead to a more straightforward design.

CODE SNIPPET

CLASSES	**VISITOR CLASSES**

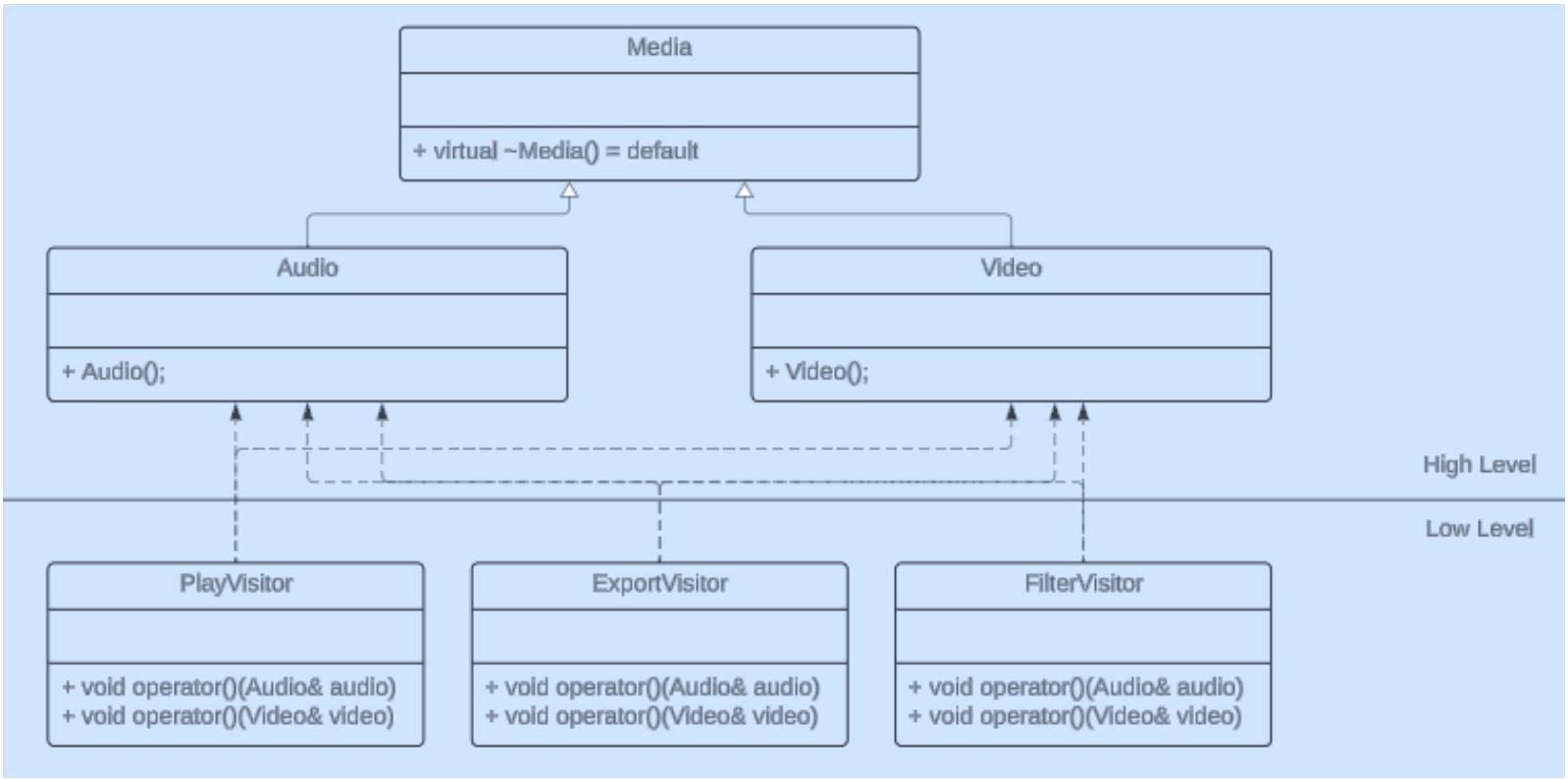
class Media {
public:
  //comman operations
  virtual ~Media() = default;
};
	class FilterVisitor {
	public:
	  void operator()(Audio& audio) const {
	  //Filter audio
	  }
	
	  void operator()(Video& video) const {
	  //Filter video
	  }
	};

class Audio : public Media {
public:
  //specific methods for Audio
  Audio(){}
  ~Audio() override = default;
};
	class ExportVisitor {
	public:
	  void operator()(Audio& audio) const {
	  //Export audio
	  }
	
	  void operator()(Video& video) const {
	  //Export video
	  }
	};

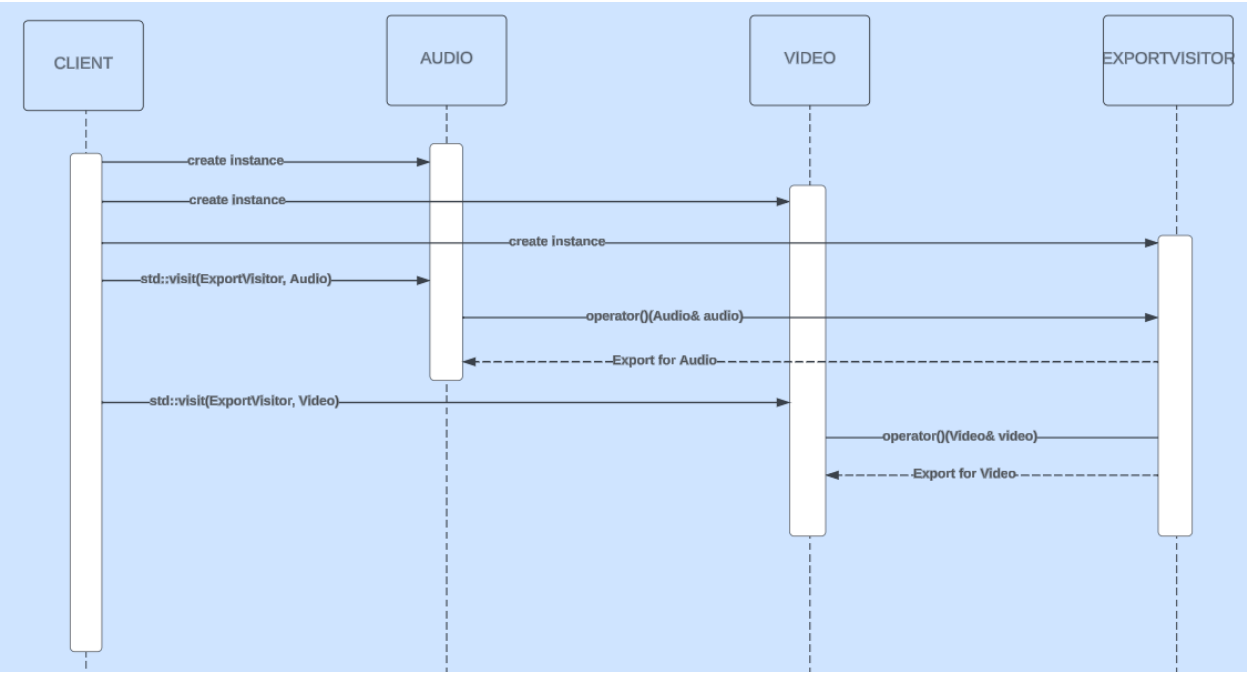
class Video : public Media {
public:
  Video(){}
  ~Video() override = default;
};	

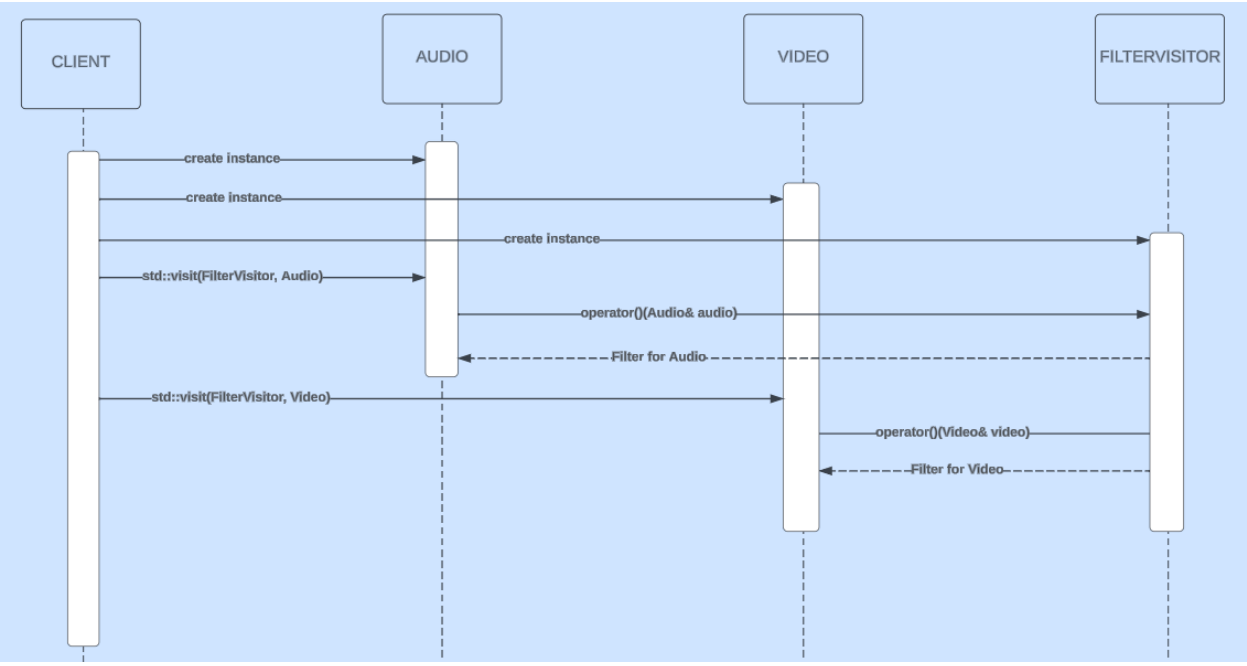
In main function, we should define the variant as vector type: “std::vector<std::variant<Audio, Video>> mediaVariant;” then emplace back what we want to this mediaVariant for example “mediaVariant.emplace_back(Audio{});” and “mediaVariant.emplace_back(Video{});”. Then we can call the filter operation on this audio and video within a for loop with visit function like “std::visit(FilterVisitor(),element);”. Visitor classes are function objects (functors) that define the operations directly in their “operator()” methods. Operations are applied using “std::visit” directly on the variant, dispatching the appropriate function object based on the type stored in the variant. This new design avoids need for virtual functions and polymorphism.

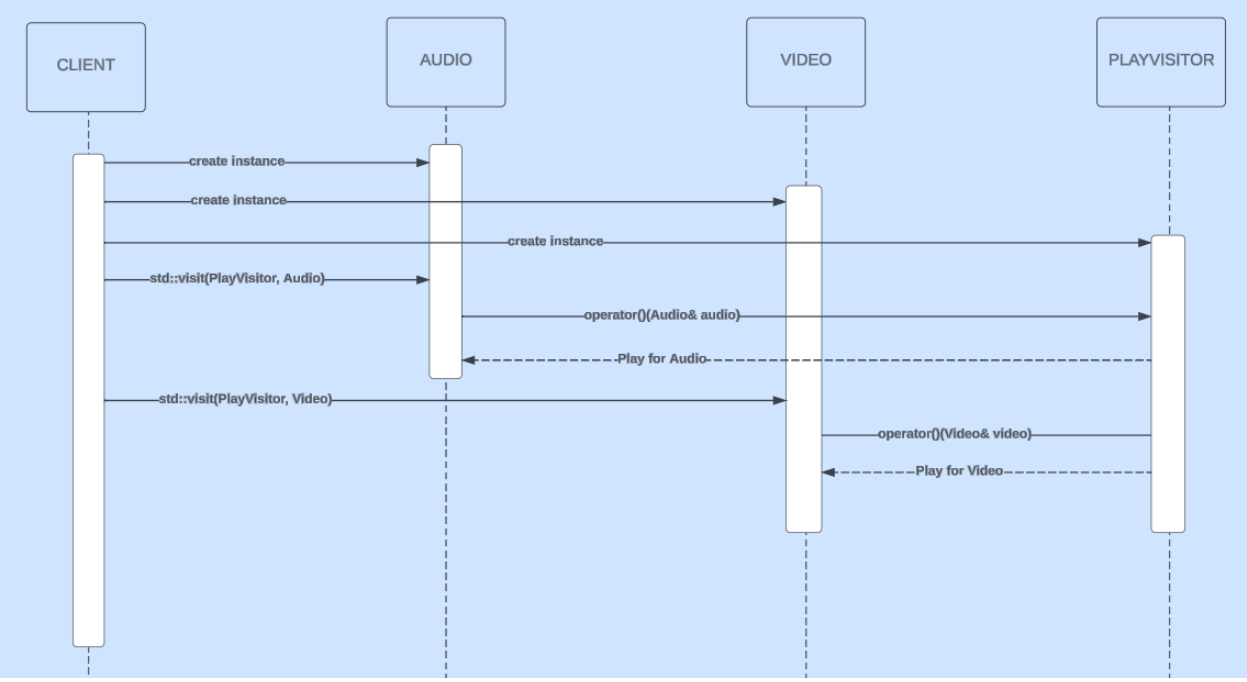
CLASS DIAGRAM



SEQUENCE DIAGRAMS






CLIENT
AUDIO
VIDEO
PLAYVISITOR
create instance
create instance
create instance
std::visit(PlayVisitor, Audio)
operator()(Audio& audio)
Play for Audio
std::visit(PlayVisitor, Video)
operator()(Video& video)
Play for Video

QUESTION 2

INTRODUCTION

Bridge design pattern splits a large class into two separate hierarchies which can be developed independently. Here, we are going to separate the abstraction (Media) from its implementation (output devices). With using bridge design pattern, the "Media" class is decoupled from the specific output devices like speakers and headphones. After that, we can easily add new output devices in the future without modifying the "Media" hierarchy. Without bridge, any changes or additions to output devices would likely require modifying the Media class. Without using the bridge pattern, we would have a direct coupling between the abstraction and its implementation.

CODE SNIPPET

OUTPUT DEVICE HIERARCHY	**MEDIA HIERARCHY**

class OutputDevice {
public:
  virtual void playOnDevice() const = 0;
};
	class Media {
	public:
	  Media(std::unique_ptr<OutputDevice>
	  outputDevice) :
	  m_outputDevice(std::move(outputDevice)) {}
	
	  virtual void play() = 0;
	
	protected:
	  std::unique_ptr<OutputDevice>
	  m_outputDevice;
	};

class Speakers : public OutputDevice {
public:
  void playOnDevice() override {
  //Implementation to play on Speakers
  }
};
	class Audio : public Media {
	public:
	  Audio(std::unique_ptr<OutputDevice>
	  outputDevice) :
	  Media(std::move(outputDevice)) {}
	
	  void play() override {
	  m_outputDevice->playOnDevice();
	  //Audio play code
	  }
	};

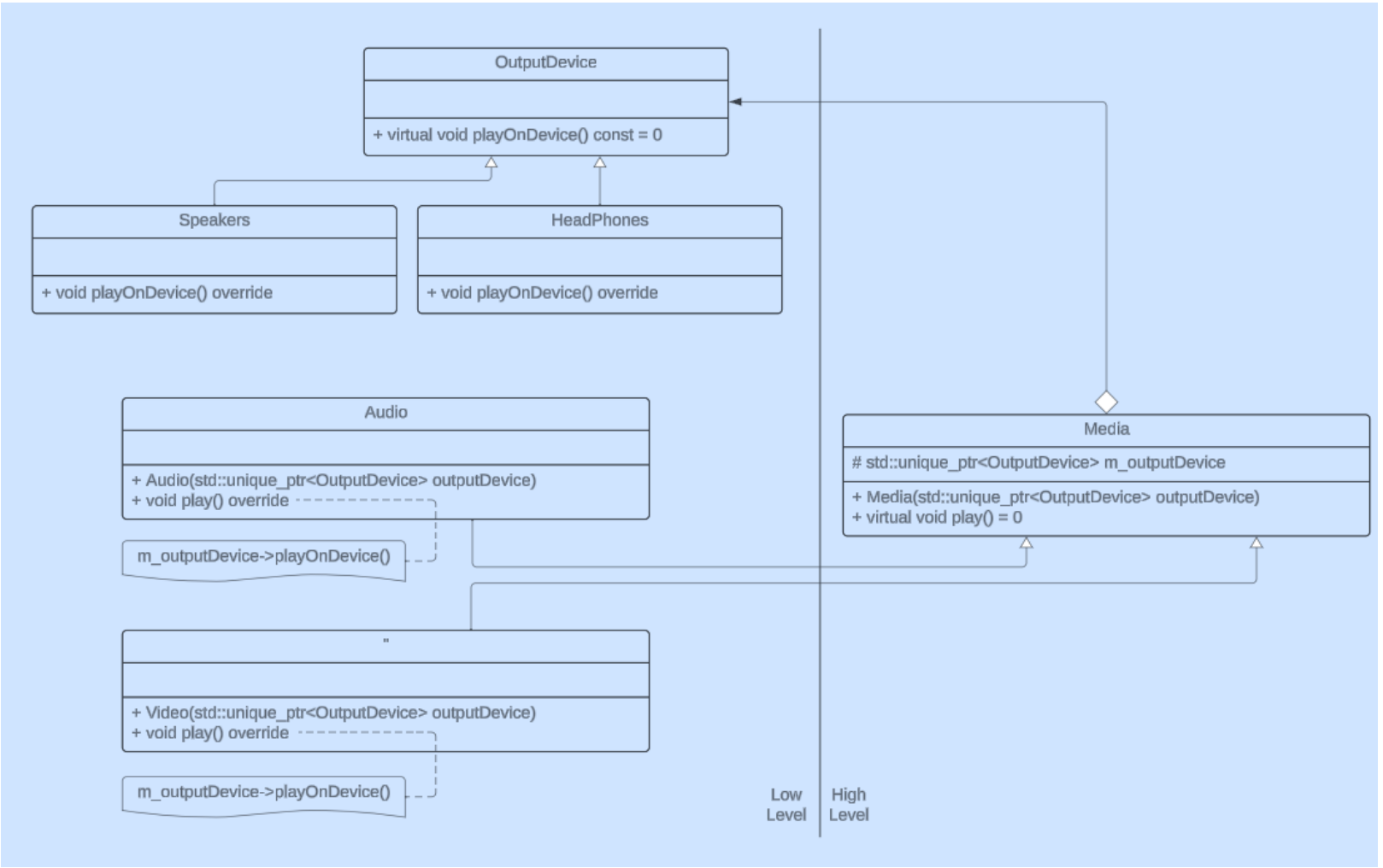
```
class Headphones : public OutputDevice {
public:
    void playOnDevice() override {
        //Implementation to play on Headphones
    }
};
```

```
class Video : public Media {
public:
    Video(std::unique_ptr<OutputDevice>
     outputDevice) :
     Media(std::move(outputDevice)) {}

    void play() override {
        m_outputDevice->playOnDevice();
        //Video play code
    }
};
```

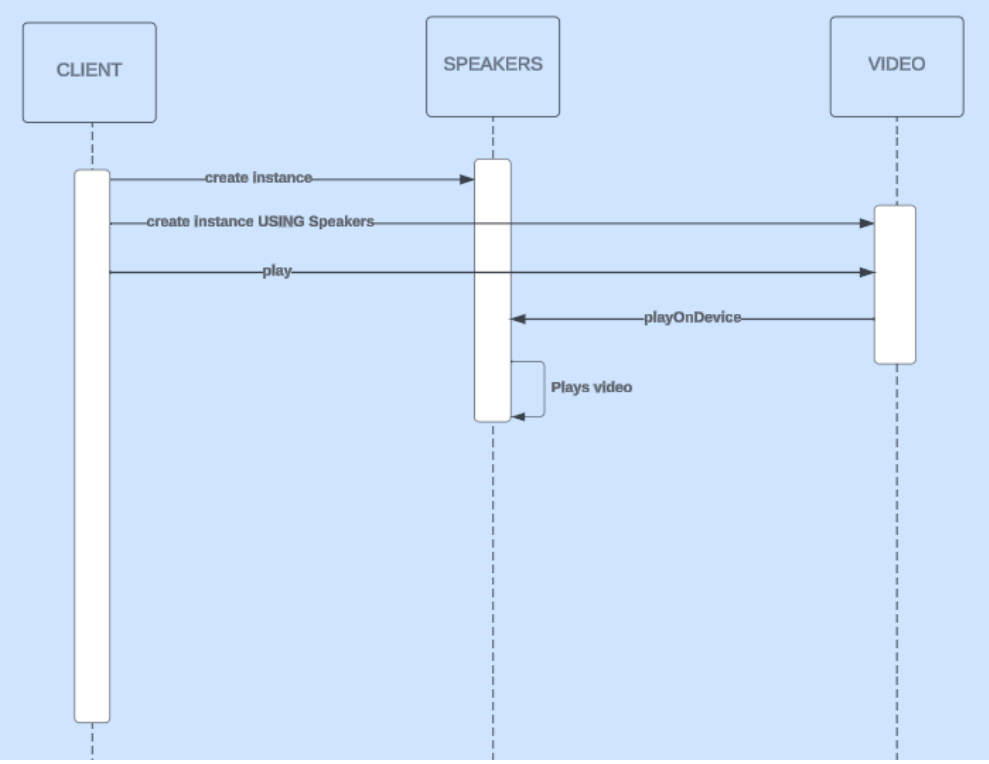
With bridge, the “Media” class only holds a reference to the “OutputDevice” interface. This decouples the abstraction (Media) from the details of its implementation (Speakers, Headphones, etc.). Otherwise, the “Media” class would directly depend on specific output device classes.

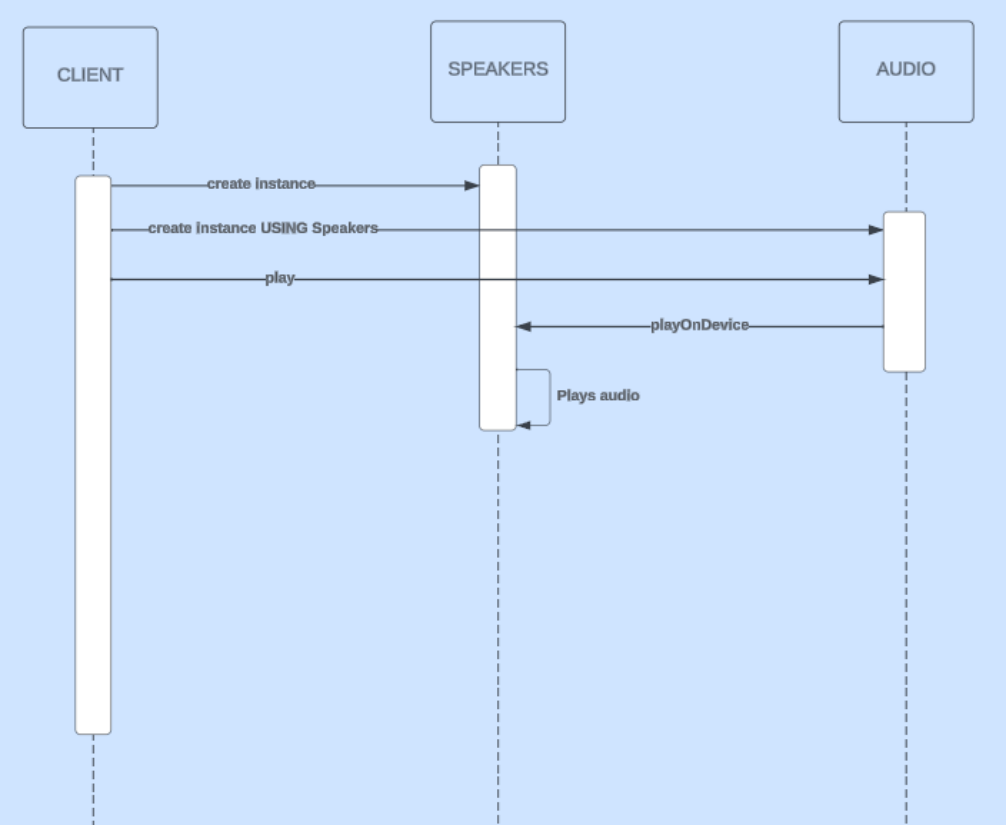
In main function, client should create the speakers and headphones as unique pointer of type OutputDevice. Then using this created pointers, user should create unique pointers of type Media for each audio and video media. For example, if user uses speakers to create Audio media, then playing operation on audio will use speaker output device.

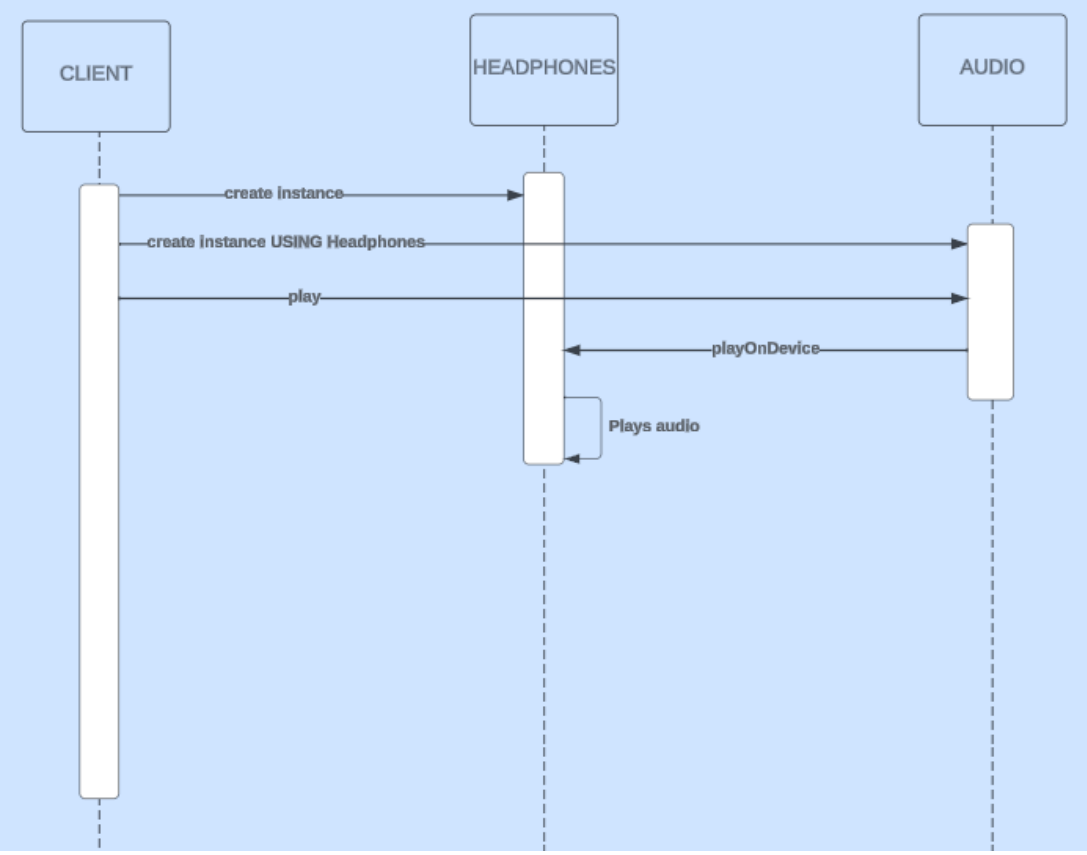
CLASS DIAGRAM

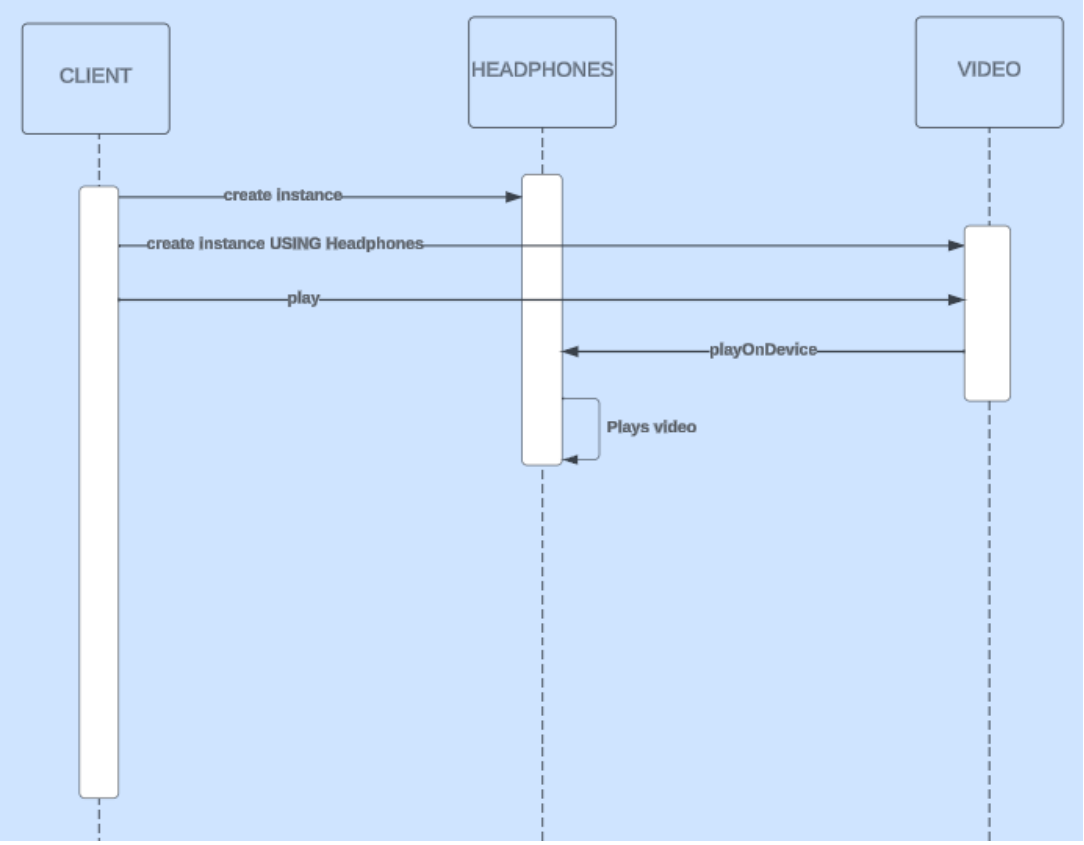


SEQUENCE DIAGRAMS






CLIENT
HEADPHONES
AUDIO
create instance
create instance USING Headphones
play
playOnDevice
Plays audio


CLIENT
HEADPHONES
VIDEO
create instance
create instance USING Headphones
play
playOnDevice
Plays video

QUESTION 3

INTRODUCTION

External polymorphism is used to separate the behavior of a class from its implementation. This pattern is employed when you want to define a family of algorithms, encapsulate each algorithm and make them interchangeable. With external polymorphism, we create separate classes for various behaviors and allowing a class to delegate its behavior to an instance of one of these strategy classes.

In the LibCircle and LibSquare classes, the logic for drawing and serializing is tghtly coupled with the shape classes themselves. So here I am going to introduce an abstract base class and child classes to separate the concerns. Each shape class is now focusing on its core functionality, while drawing and serializing logic is moved to separate classes.

CODE SNIPPET

LIBS	**MEDIA HIERARCHY**

class LibCircle {
public:
  int32_t getRadius(){
  return radius;
  }
  void setRadius(int32_t rValue){
  radius = r;
  }
private:
  int32_t radius;
};
	class ShapeConcept {
	public:
	  virtual void draw() = 0;
	  virtual string serialize() = 0;
	};

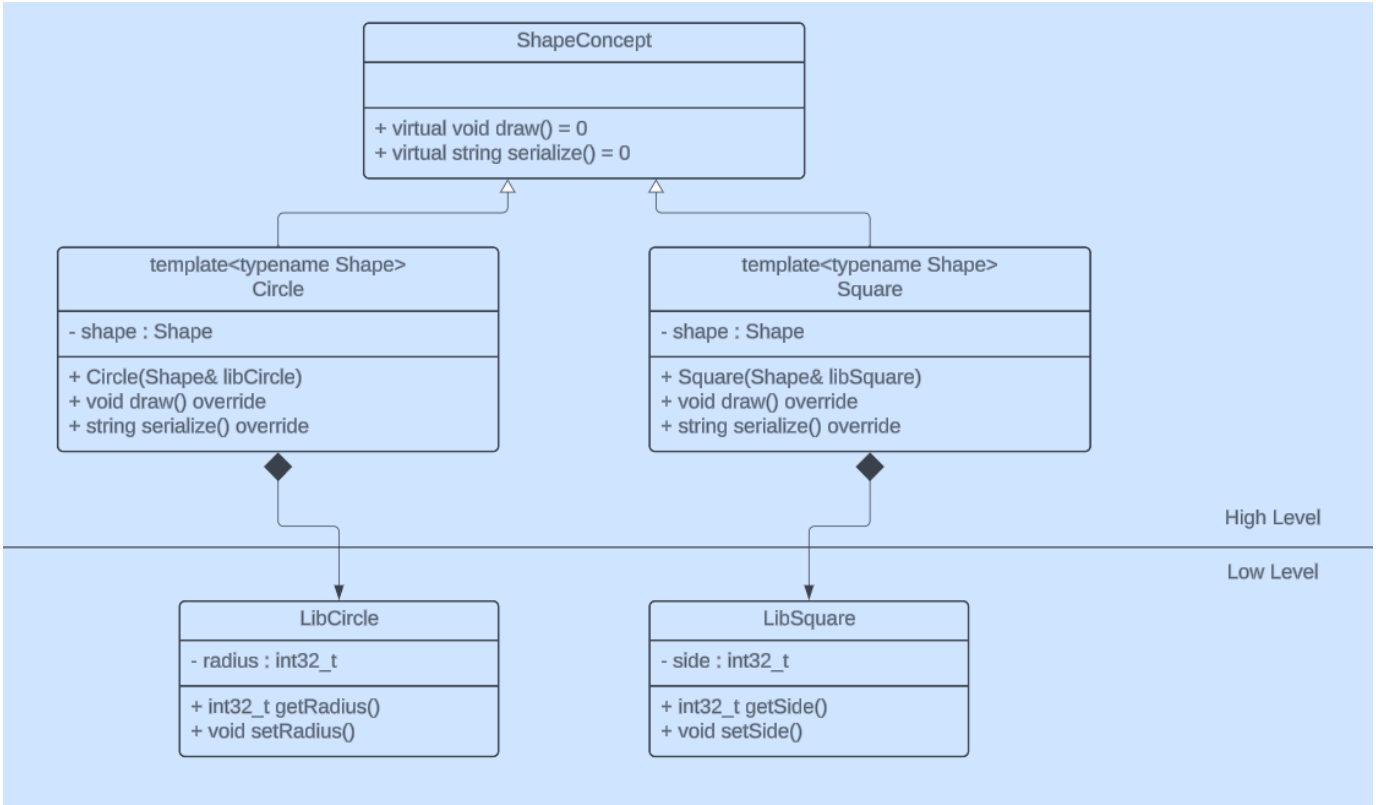
class LibSquare {
public:
  int32_t getSide(){
  return side;
  }
  void setRadius(int32_t sValue){
  side = sValue;
  }
private:
  int32_t side;
};
	template <typename Shape>
	class Circle : public ShapeConcept {
	public:
	  Circle(Shape& libCircle) : shape(libCircle){}
	
	  void draw() override{
	  //Draw operation
	  //shape.getRadius() is used
	  }
	  string serialize() override{
	  //Serialize operation
	  //shape.getRadius() is used

	private: Shape shape; };

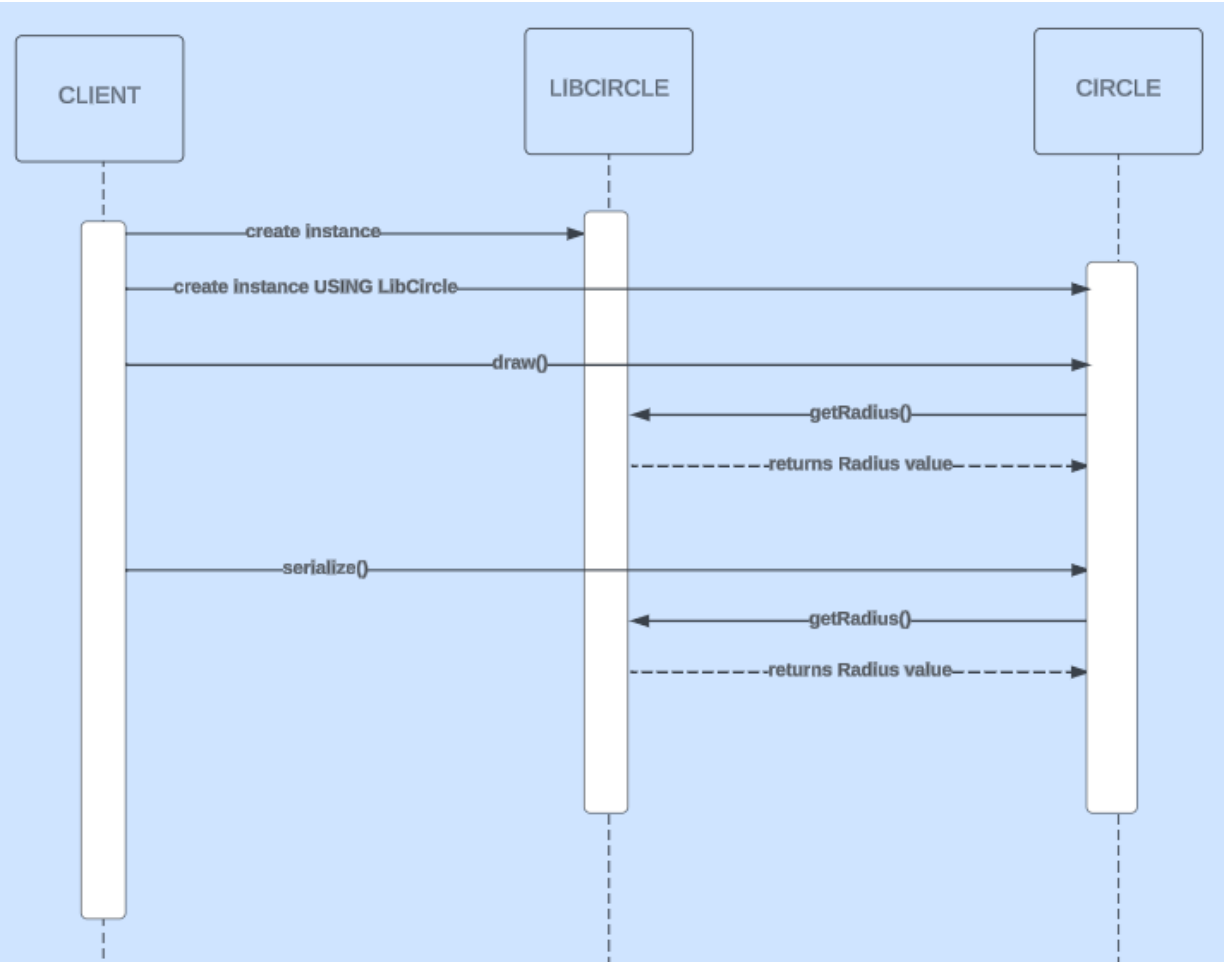
	template <typename Shape>
	class Square : public ShapeConcept {
	public:
	  Square(Shape& libSquare):shape(libSquare){}
	
	  void draw() override{
	  //Draw operation
	  //shape.getSide() is used
	  }
	  string serialize() override{
	  //Serialize operation
	  //shape.getSide() is used
	
	private:
	  Shape shape;
	};

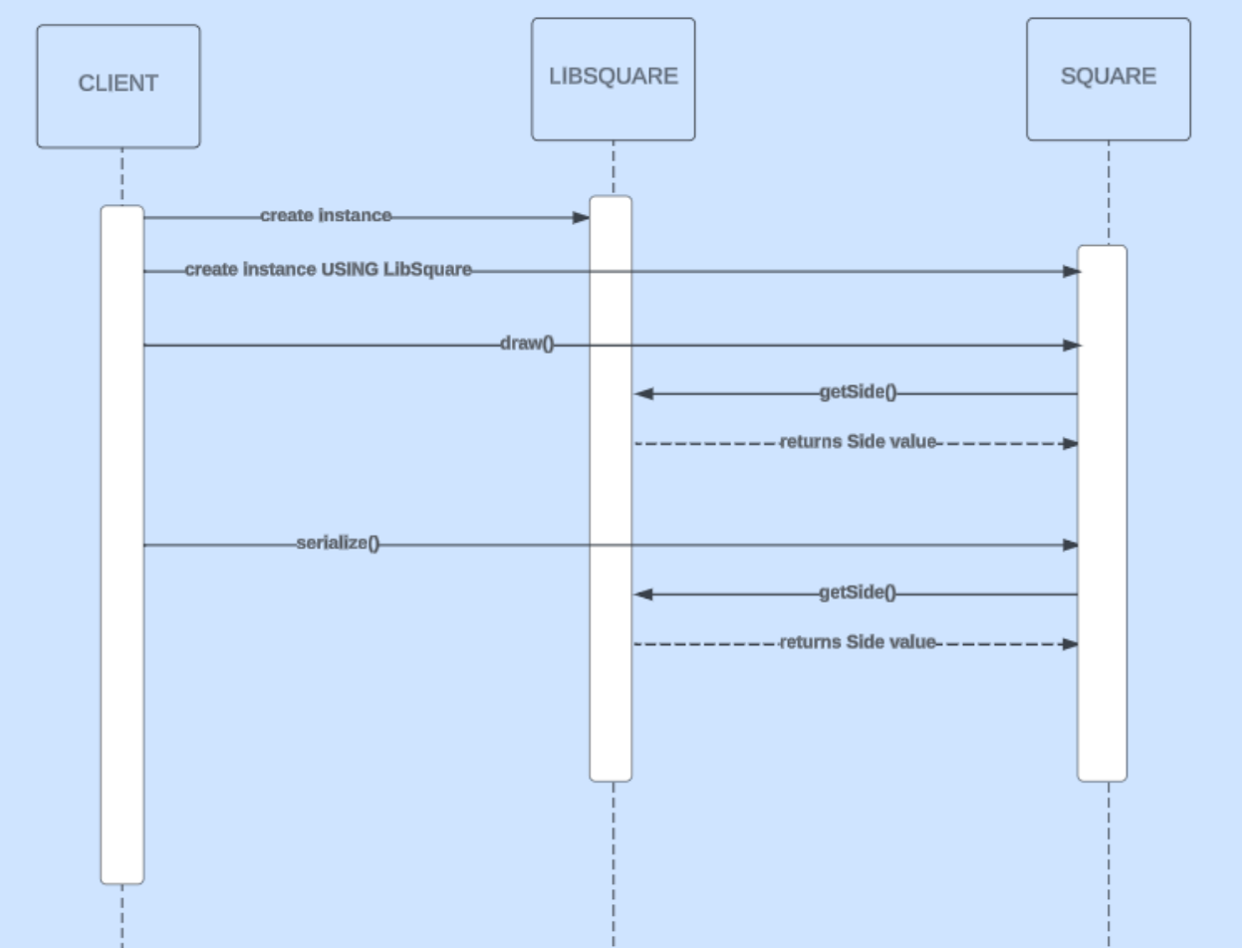
In main function, we should define LibCircle and set radius to it. Also LibSquare should be defined and set side to it. Then using them as template, we can create Circle and Square. After that we can use draw and serialize functions of the circle and the square.

CLASS DIAGRAM



SEQUENCE DIAGRAMS




CLIENT
LIBSQUARE
SQUARE
create instance
create instance USING LibSquare
draw()
getSide()
returns Side value
serialize()
getSide()
returns Side value

QUESTION 4

INTRODUCTION

In this question, we need to achieve that when the WMSystem is updated, corresponding stations are fed with the updated data. For this reason, to allow for future extensiblitity without modifying the "WMSystem" class when adding new stations, we can use the Observer Design Pattern from the GOF book. Observer Design Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

CODE SNIPPET

SYSTEM

```
class WMSystem {
public:
   enum StateChange{
      temperature,
      airPressure,
      wind
   }

   using WMObserver = Observer<WMSystem,
                               StateChange>;

   void addObserver(WMObserver* observer) {
      m_observers.insert(observer);
   }

   void removeObserver(WMObserver* observer) {
      m_observers.erase(observer);
   }

   void notifyObservers(StateChange state) {
      for (Observer* observer : m_observers) {
         observer->update(*this, state);
      }
   }
```

OBSERVERS

```
template<typename Subject, typename State>
class Observer {
public:
   virtual void update(Subject const& subject,
                       State state){}
};
```

```
class DisplayStation : public Observer<WMSystem,
                       WMSystem::StateChange> {
public:
   void update(WMSystem const& system,
               WMSystem::StateChange state) override{
      if (state == WMSystem::temperature){
         //Dısplay temperature
         system.getTemperature();
      }
      else if (state == WMSystem::airPressure){
         //Display air pressure
         system.getAirPressure();
      }
      else if (state == WMSystem::wind){
         //Display wind
         system.getWind();
      }
   }
};
```

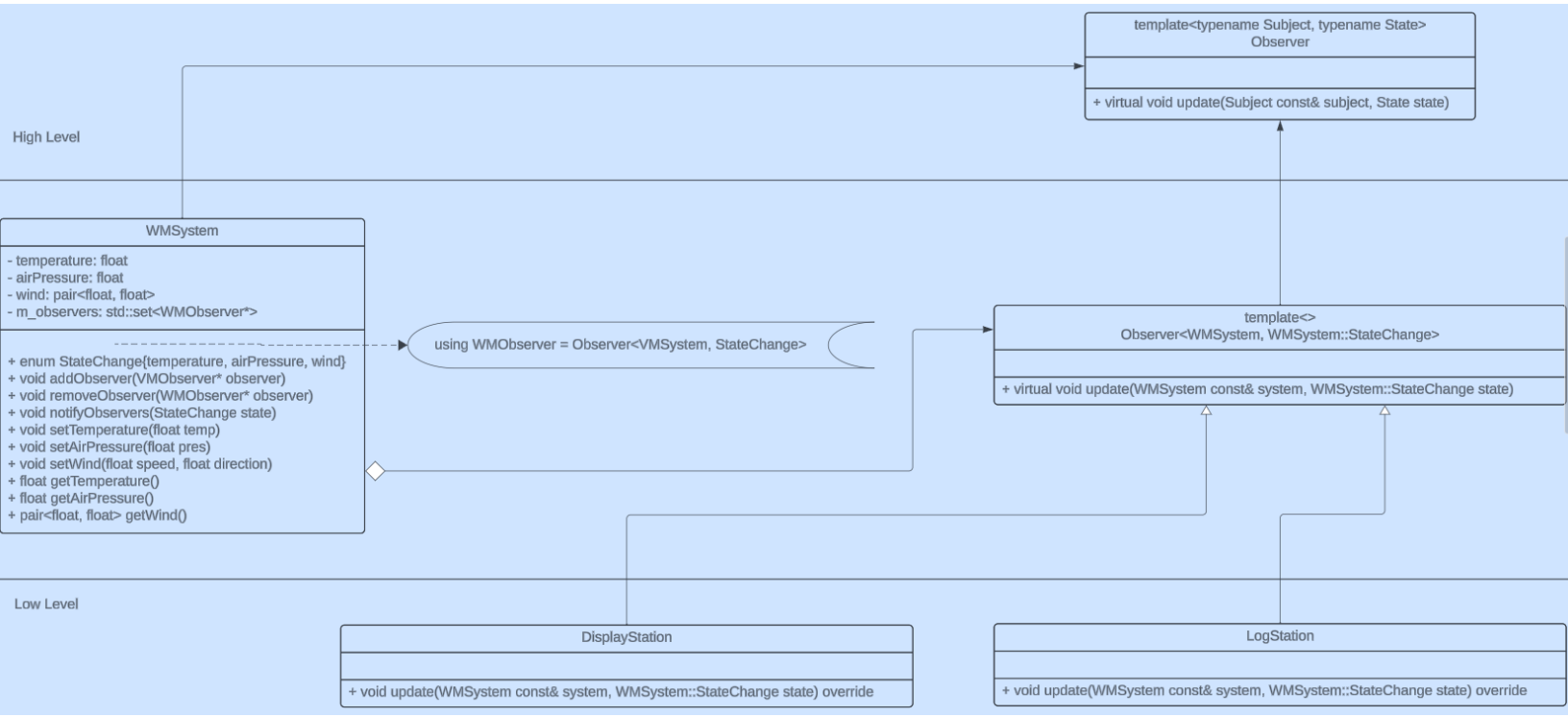
```
   void setTemperature(float temp) {
      temperature = temp;
      notifyObservers(temperature);
   }
   void setAirPressure(float pres) {
      airPressure = pres;
      notifyObservers(airPressure);
   }
   void setWind(float speed, float direction) {
      pair<float, float> newWind(speed, direction);
      wind = newWind;
      notifyObservers(wind);
   }




   float getTemperature(){
      return temperature;
   }
   float getAirPressure(){
      return airPressure;
   }
   pair<float, float> getWind(){
      return wind;
   }


private:
   float temperature;
   float airPressure;
   pair<float, float> wind;     //Wind speed and direction
   std::set<WMObserver*> m_observers;
};
```

```
class LogStation : public Observer {
public:
   void update(WMSystem const& system,
               WMSystem::StateChange state) override{
      if (state == WMSystem::temperature){
         //Log temperature
        system.getTemperature();
      }
      else if (state == WMSystem::airPressure){
         //Log air pressure
        system.getAirPressure();
      }
     else if (state == WMSystem::wind){
         //Log wind
        system.getWind();
      }
   }
};
```

In this design, “WMSystem” is our concrete class that maintains list of observers that it keeps in a set and notifies them when anything changes. “DisplayStation” and “LogStation” classes are concrete obserrvers that implement the “Observer” interface. When a parameter is updated in “WMSystem”, notifyObservers() method is called and this method calls the “update” method for each observer by pointers.

In main function, client should define “WMSystem”, “DisplayStation” and “LogStation” then add these observers to the weather system by addObserver() method. After that, for example when temperature is set, all the observers will be notified and stations will be updated accordingly. If a new class is needed to be added, we simply add an observer to our “WMSystem”. If we want to remove an observer, we should use removeObserver method.

CLASS DIAGRAM



SEQUENCE DIAGRAM