Fatih Doğaç

1901042654

CSE443 HW Report

Question 1

1. Introduction

Initially, there were no problem with this code. The programmer wanted objects which could be iterated through a pointer. But then, some new operations needed to be added and this structure is not suitable for multiple operations. So we are asked to change the design pattern to something that can afford to have more operations and multiple objects.

If we add those functions to the code without modifying the code, we would have to write the same function over and over for each object.

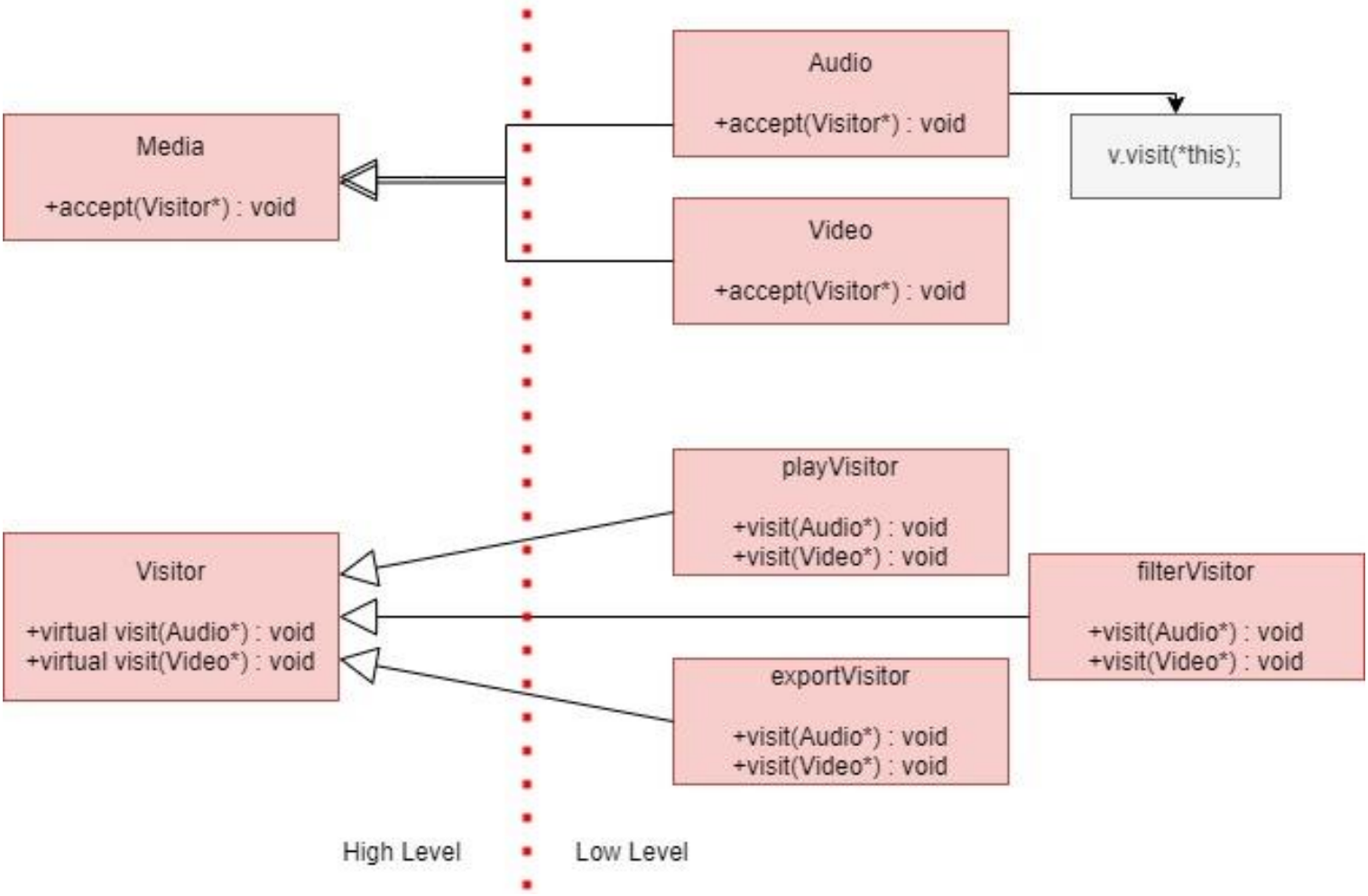
2. Solution

```
class Visitor{
    public:
        virtual void visit(Audio* ) = 0;
        virtual void visit(Video* ) = 0;
};
```

```
class Media{
    public:
        // Constructors
        virtual void accept(Visitor* ) = 0;

};

class Auido : public Media{
    public:
        void accept(Visitor* v){
            v.visit(*this);
        }
};

class Video : public Media{
    public:
        void accept(Visitor* v){
            v.visit(*this);
        }
};
```

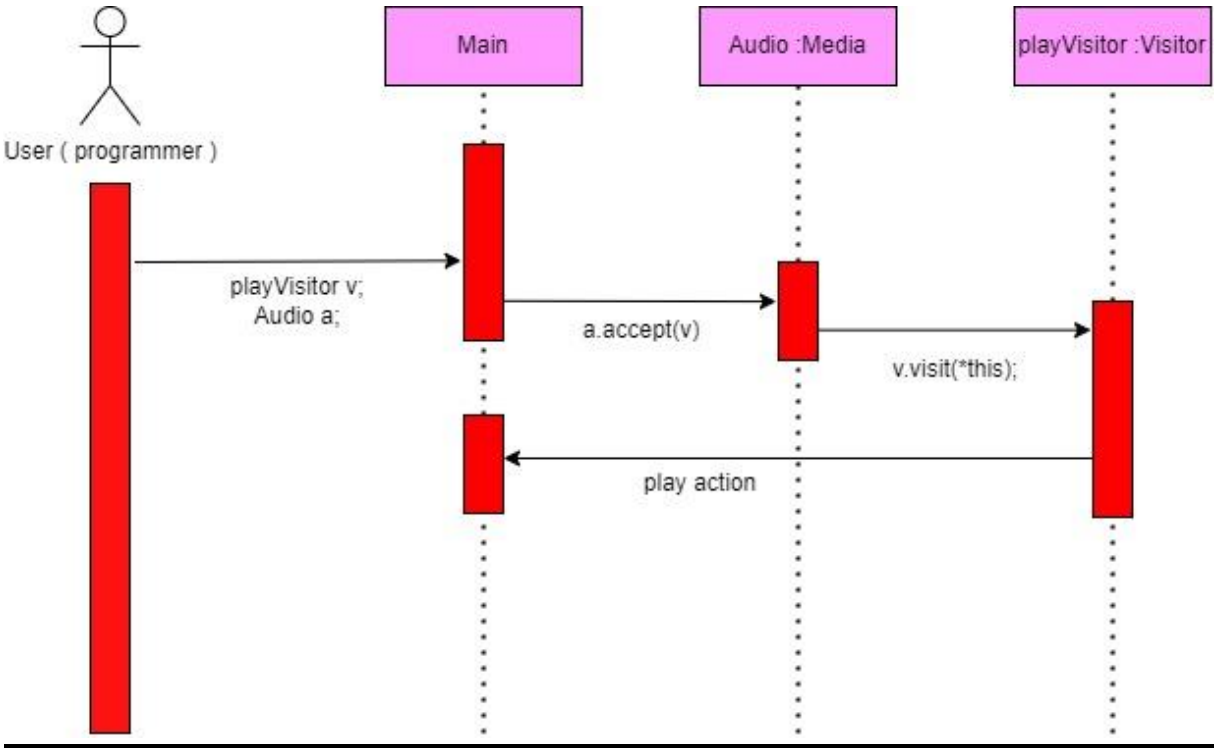
```cpp
class playVisitor : public Visitor{

    public:
        void visit(Audio* a){
            // play function for Audio
        }

        void visit(Video* a){
            // play function for Video
        }
};

class filterVisitor : public Visitor{

    public:
        void visit(Audio* a){
            // filter function for Audio
        }

        void visit(Video* a){
            // filter function for Video
        }
};

class exportVisitor : public Visitor{

    public:
        void visit(Audio* a){
            // export function for Audio
        }

        void visit(Video* a){
            // export function for Video
        }
};
```

3. Class Diagram

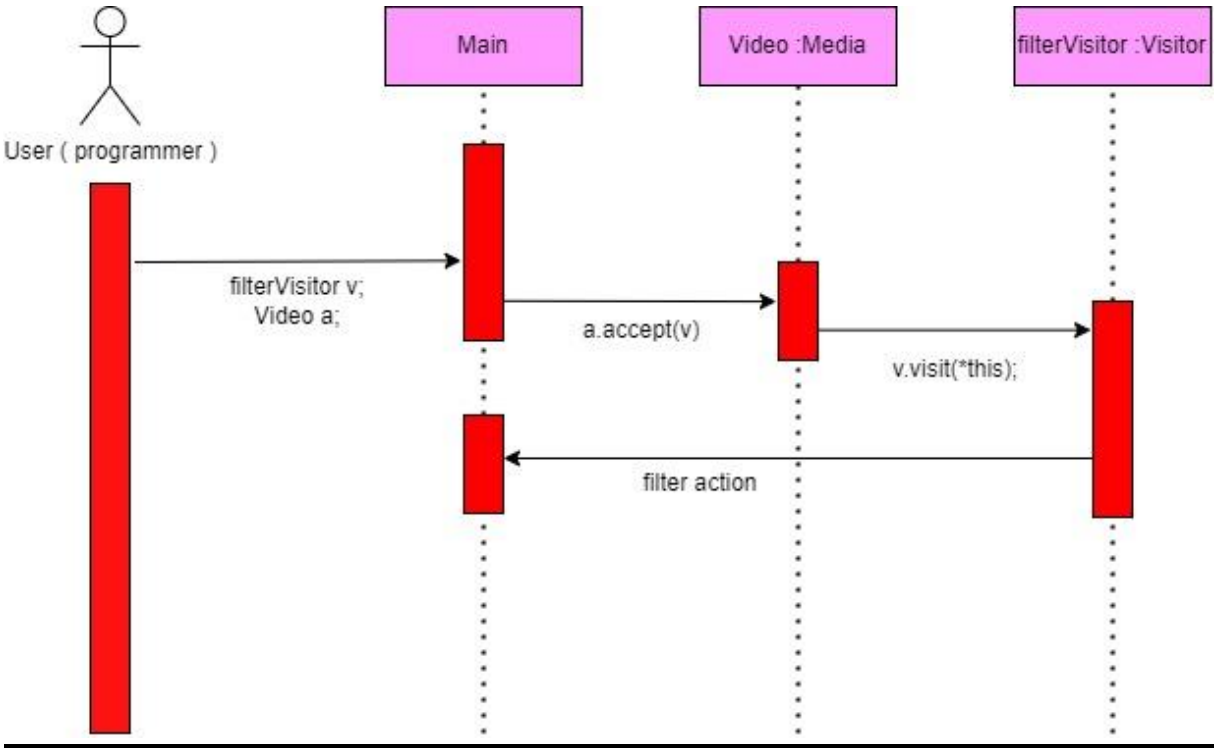


4. Sequence Diagrams

a. Play function sequence diagram



b. Filter function sequnece diagram



Question 2

1. Introduction

   In this question, we are asked to implement bridge design pattern. Our concern here is having multiple objects and not so many actions.

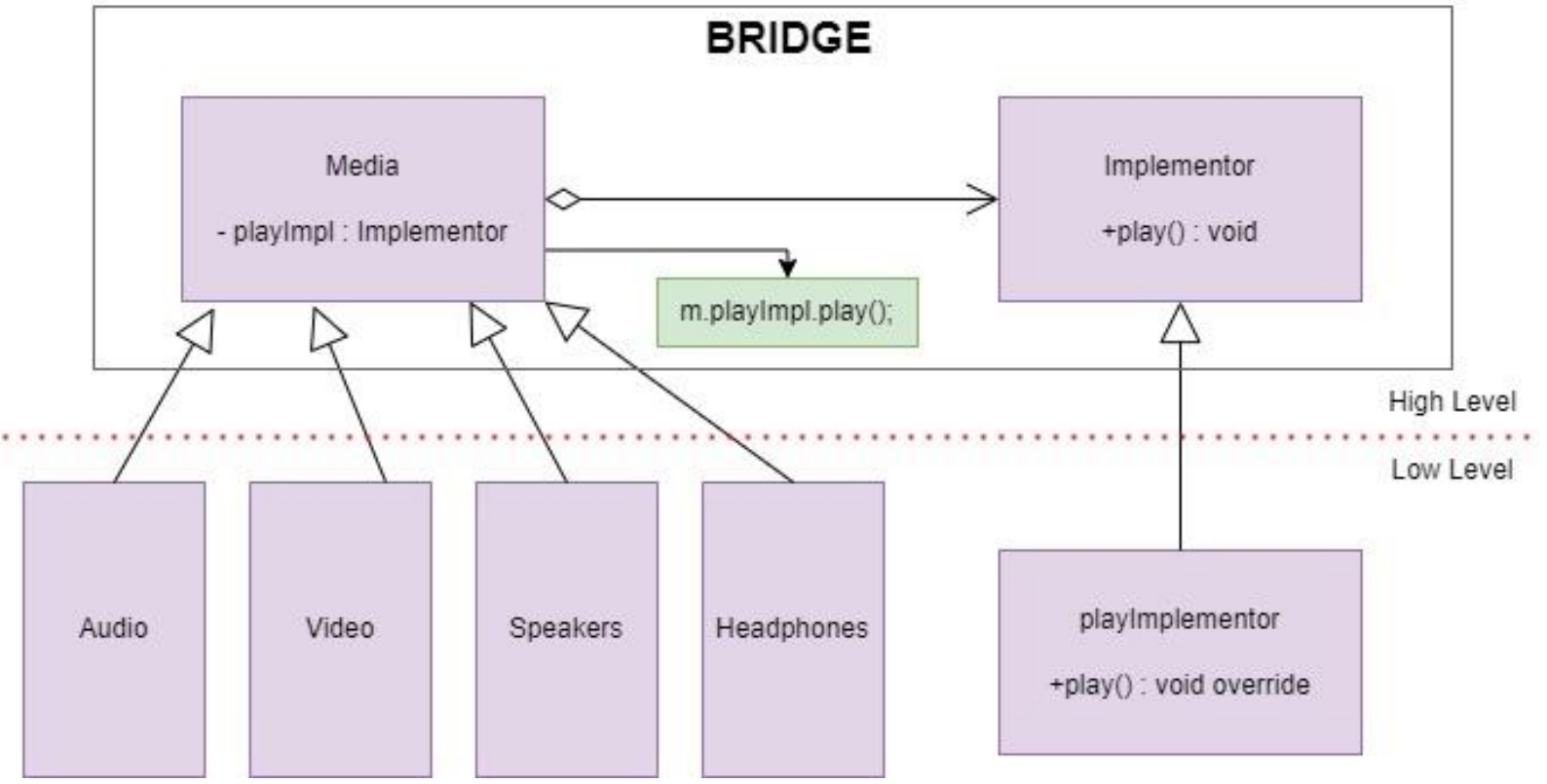
2. Solution

```
class Media{

    Implementor playImpl;
};

class Auido : public Media{
};

class Video : public Media{
};

class Speakers : public Media{
};

class Headphones : public Media{
};
```

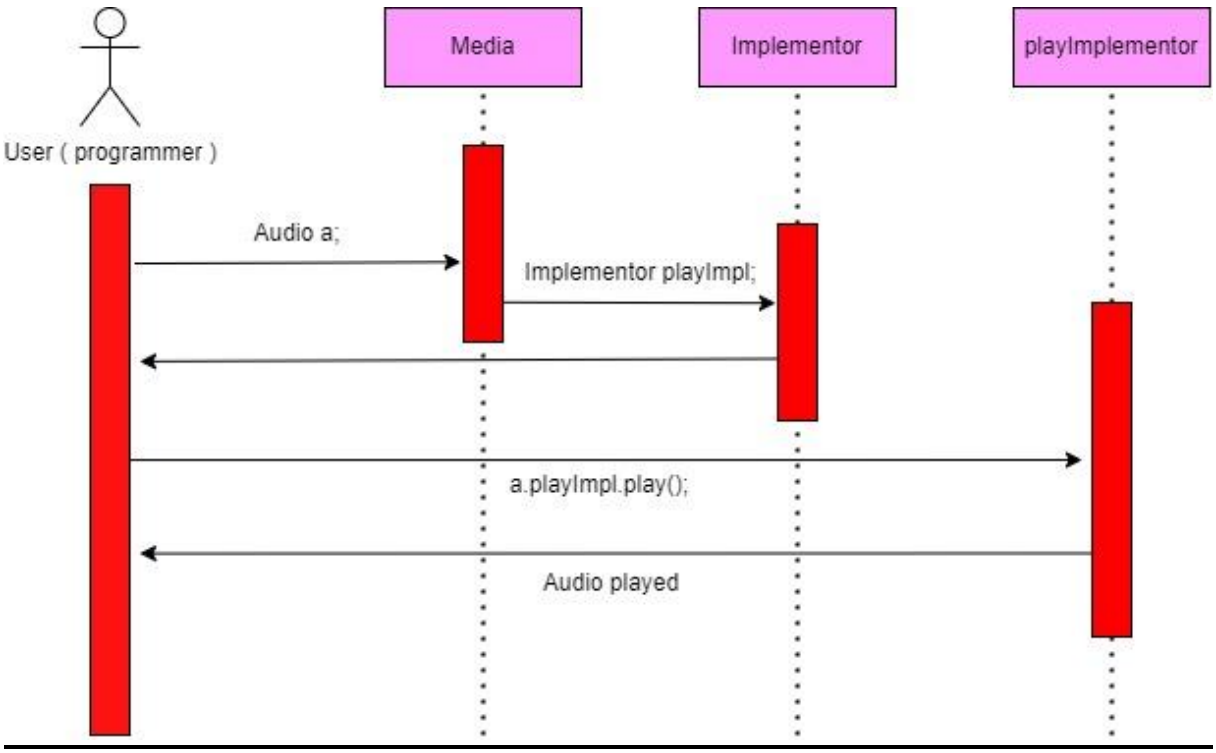
```
class Implementor{
    public:
        virtual void play() = 0;
}

class playImplementor: public Implementor{
    public:
        void play(){
            // play code
        }
}
```

```
int main(){
    Audio a;
    a.playImpl.play();
}
```

3. Class Diagram



4. Sequence Diagram



Question 3

1. Introduction

In this question, we are asked to implement external polymorphism to LibSquare and LibCircle. They have no common base class. That's why we are adding a base class "externally".

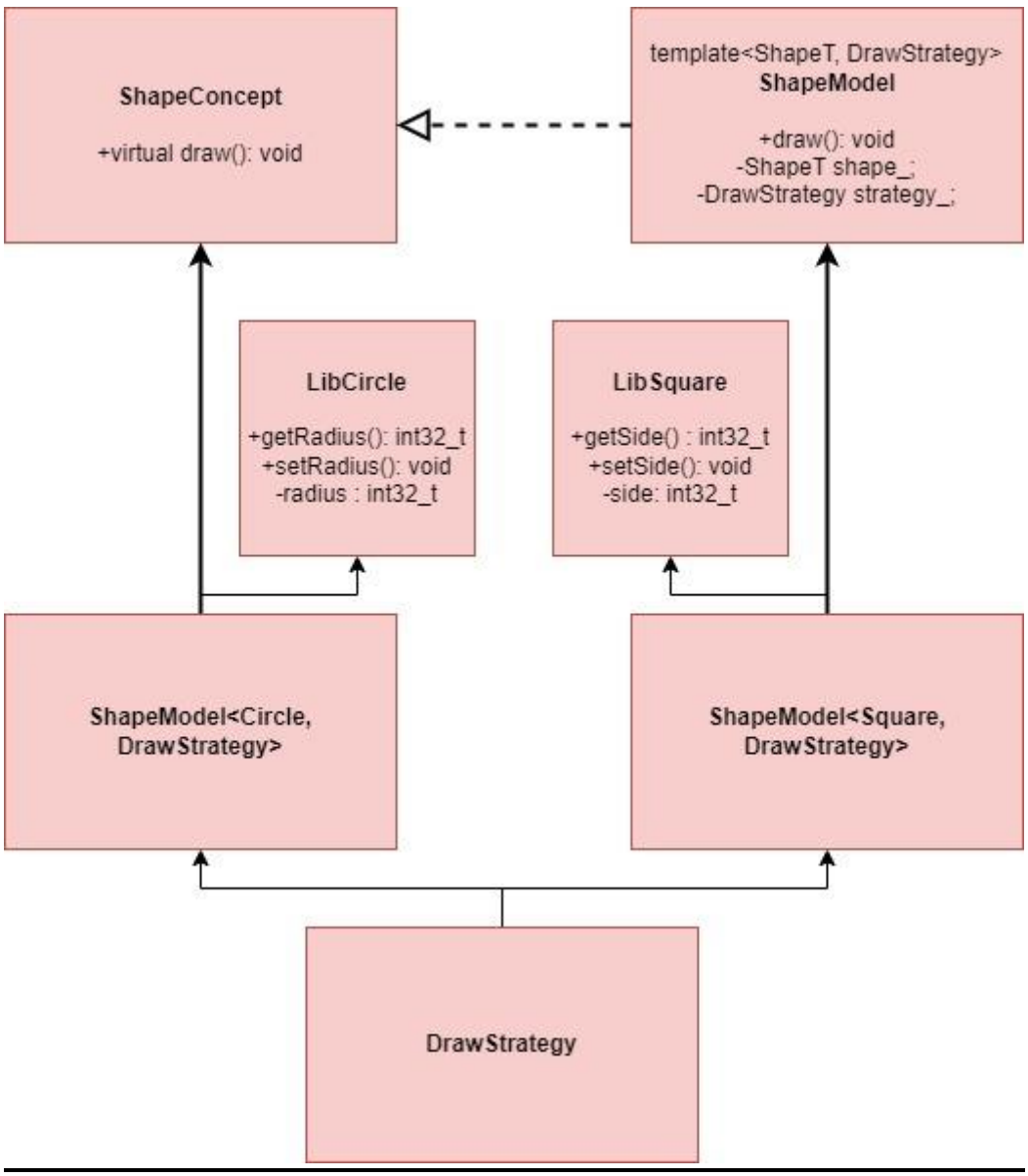
ShapeModel is kind of a templated version of the initial shape classes. So we can add as many different shapes as we like. We can also specify the draw function for each shape. After those operations, say we will have a Circle object with Radius of 2.3 cm. But we will refer to it as CircleModel which is the ShapeModel template that we built but with the original Circle class.

2. Solution

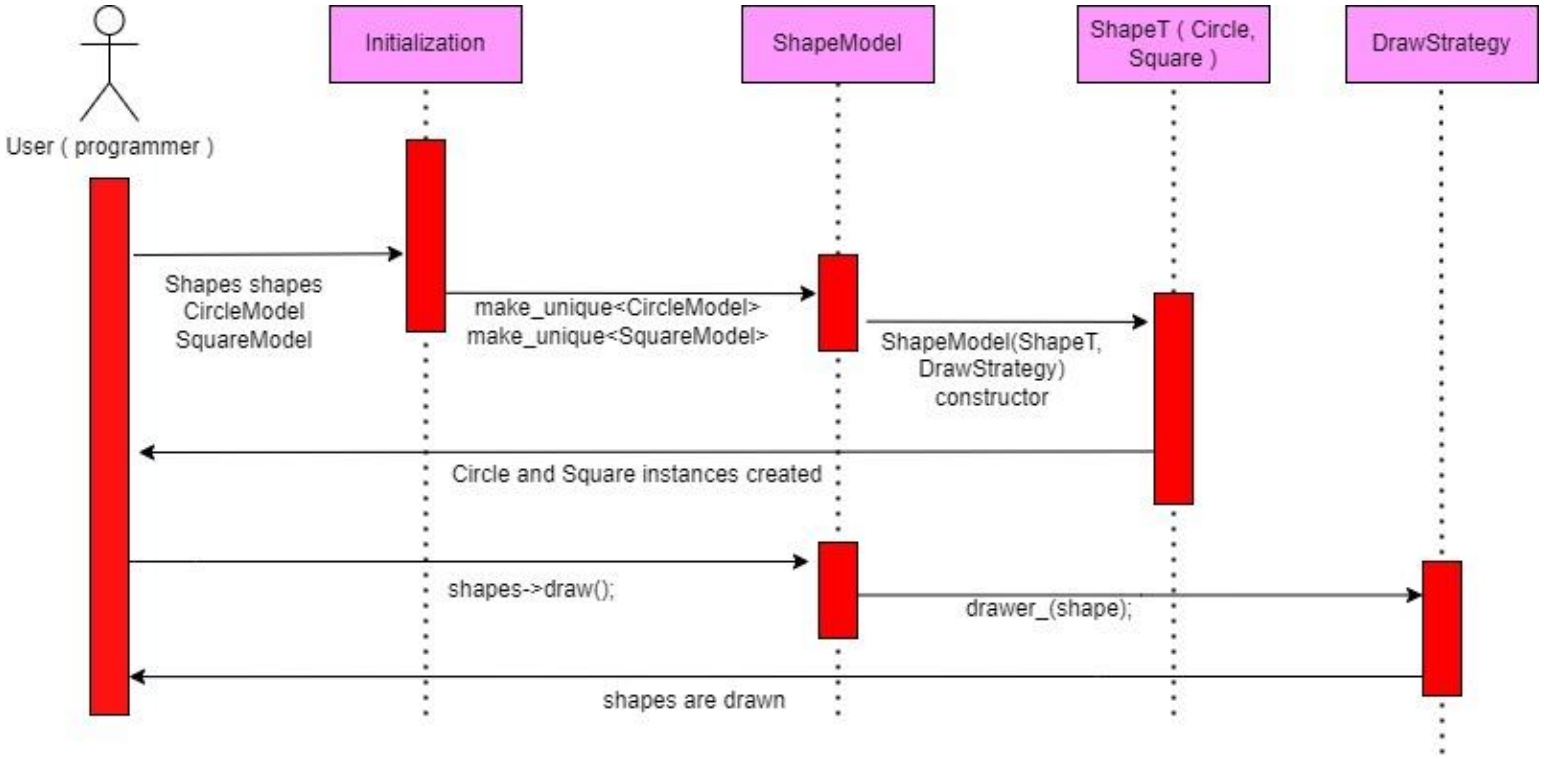
```
class ShapeConcept
{
public:
    virtual ~ShapeConcept() = default;
    virtual void draw() const = 0
}

template< typename ShapeT
    , typename DrawStrategy >
class ShapeModel : public ShapeConcept
{
public:
    explicit ShapeModel( ShapeT shape, DrawStrategy drawer )
    : shape_{ std::move(shape) }
    , drawer_{ std::move(drawer) }
    {}

    void draw() const override { drawer_(shape_); }

private:
    ShapeT shape_;
    DrawStrategy drawer_;
};
```

```
int main(){

    using Shapes = std::vector<std::unique_ptr<ShapeConcept>>;
    using CircleModel = ShapeModel<Circle,DrawStrategy>;
    using SquareModel = ShapeModel<Square,DrawStrategy>;

    Shapes shapes{};

    shapes.emplace_back(
        std::make_unique<CircleModel>(Circle{2.3}, DrawStrategy(/*...red...*/) ) );
    shapes.emplace_back(
        std::make_unique<SquareModel>(Square{1.2}, DrawStrategy(/*...green...*/) ) );
    shapes.emplace_back(
        std::make_unique<CircleModel>(Circle{4.1}, DrawStrategy(/*...blue...*/) ) );

    // Drawing all shapes
    for( auto const& shape : shapes )
    {
        shape->draw();
    }

    return EXIT_SUCCESS;
}
```

3. Class Diagram



4. Sequence Diagram



Question 4

1. Introduction

   First of all, in this quesiton we would like to “observe” the sensors and respond from another classes. So we will use observer design pattern.

2. Solution

   First we implement an Observer interface.

```cpp
template< typename Subject, typename StateTag >
class Observer
{
public:
    virtual ~Observer() = default;
    virtual void update( Subject const& subject, StateTag property ) = 0;
};
```

Then our system comes:

```cpp
class WMSystem
{
public:
    enum StateChange
    {
    TemperatureChanged,
    AirPressureChanged,
    WindChanged
    };

    using SystemObserver = Observer<WMSystem,StateChange>;

    explicit WMSystem( std::string temp, std::string pres , std::string dir , std::string speed )
    : temp_{ std::move(temp) }
    , pres_{ std::move(pres) }
    , dir_{ std::move(dir) }
    , speed_{ std::move(speed) }
    {}
```

```
    bool attach( SystemObserver* observer );
    bool detach( SystemObserver* observer );

    void notify( StateChange property );

    void setTemperature(float temp);
    void setAirPressure(float pres);
    void setWind(float dir, float speed);

private:
    std::string temp_;
    std::string pres_;
    std::string dir_;
    std::string speed_;
    std::set<SystemObserver*> observers_;
};
```

So, we have attributes like temp and pres. When the readings from the sensors change, they set the values of these attributes. So in setter methods, we should notify the observers which using our system to work.

```
// ----------------------------------------

void WMSystem::setTemperature(float temp){
    temp_ = std::move(temp);
    notify(TemperatureChanged);
}

// same goes for airpressure and wind changes.
```

Notify method should warn all the observers. So we have to keep a list or set to keep all of the observers.

That's why we have observers_ attribute. And in notify() method, we just traverse the observers and notify them one by one.

```
void WMSystem::notify( StateChange property )
{
    for( auto iter=begin(observers_); iter!=end(observers_); )
    {
        auto const pos = iter++;
        (*pos)->update(*this,property);
    }
}
```

We have also an enumaration type which is called StateChange. We are using it just to specify the observers which attribute has changed.

So, if we come to the observers, stations that use the sensors data.

```
class DisplayStation : public Observer<WMSystem>{
    public:
    void displayTemperature(float temp);
    void displayPressure(float pres);
    void displayWind(float dir, float speed);

    void update(WMSystem system, WMSystem::StateChange property);
}
```

Display observer (station) has its own works (displaying attributes) and an update method which is mandatory for observers.

What update method does is restarting the system with the new data as the new data come you might say.

Same goes for the LogStation.

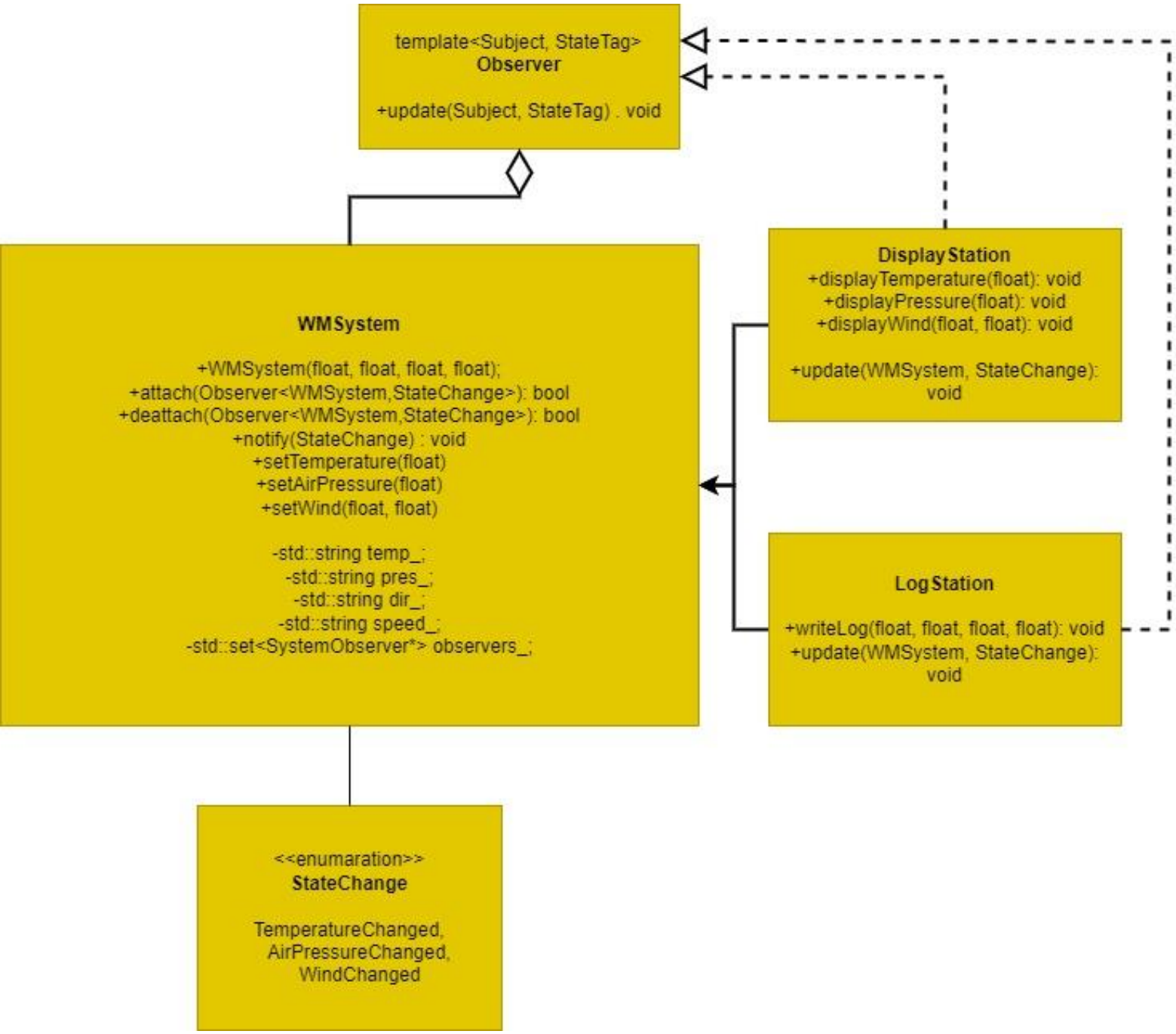
```
class LogStation : public Observer<WMSystem>{
    public:
    void writeLog(float temp, float pres, float dir, float speed);

    void update(WMSystem system, WMSystem::StateChange property){
        writeLog();
    }
}
```

Notice all stations (observers) implements the Observer template class.

In the main section, we are just attaching the observer and then the system and observers handle themselfs just good.

```cpp
int main()
{
    using SystemObserver = Observer<WMSystem,WMSystem::StateChange>;

    SystemObserver Observer( StateChange );

    WMSystem sys();

    // Attaching observers
    sys.attach( &SystemObserver );

    return EXIT_SUCCESS;
}
```

3. Class Diagram



4. Sequence Diagram