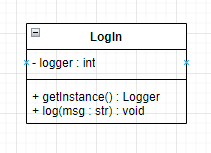
# TASK 1: Singelton Pattern:

**Scenario:**

A logging system that needs to ensure only one instance of the logger exists throughout the application.

**Class Diagram:**



**Implementation:**

#include <iostream>

using namespace std;

class LogIn {

private:

static LogIn\* instance;

LogIn() {}

public:

static LogIn\* getInstance() {

if (instance == nullptr) {

instance = new LogIn();

}

return instance;

}

void log(const std::string& message) {

cout << message << endl;

}

};

// Initialize the static member

LogIn\* LogIn::instance = nullptr;

**Drawbacks:**

Global state, which can make the code less modular.

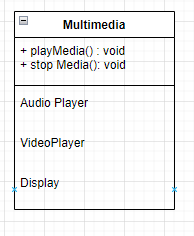
Difficult to unit test due to the static nature.

# TASK 2: Façade Pattern:

**Scenario:**

A multimedia system that has subsystems for audio, video, and display, and you want to provide a simple interface to control them all.

**Class Diagram:**



**Implementation:**

#include <iostream>

// Subsystems

class AudioPlayer {

public:

void play() { /\* implementation \*/ }

void stop() { /\* implementation \*/ }

};

class VideoPlayer {

public:

void play() { /\* implementation \*/ }

void stop() { /\* implementation \*/ }

};

class Display {

public:

void show() { /\* implementation \*/ }

void hide() { /\* implementation \*/ }

};

// Facade

class MultimediaFacade {

private:

AudioPlayer audioPlayer;

VideoPlayer videoPlayer;

Display display;

public:

void playMedia() {

audioPlayer.play();

videoPlayer.play();

display.show();

}

void stopMedia() {

audioPlayer.stop();

videoPlayer.stop();

display.hide();

}

};

**Drawbacks:**

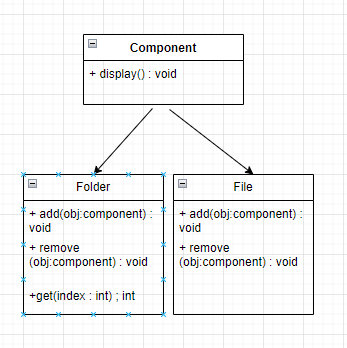
May hide the complexity, but if the facade becomes too complex, it can be a maintenance burden.

# TASK 3: Composite Pattern:

**Scenario:**

Representing a file system structure with directories and files.

**Class Diagram:**



**Implementation:**

#include <iostream>

#include <vector>

using namespace std;

// Component interface

class Component {

public:

virtual void display() = 0;

};

// Composite class

class Folder : public Component {

private:

vector<Component\*> children;

public:

void add(Component\* c) {

children.push\_back(c);

}

void remove(Component\* c) {

// Remove logic

}

Component\* getChild(int index) {

return children[index];

}

void display() {

cout << "Folder" <<endl;

for (Component\* c : children) {

c->display();

}

}

};

// Leaf class

class File : public Component {

public:

void display() {

cout << "File" << endl;

}

};

**Drawbacks:**

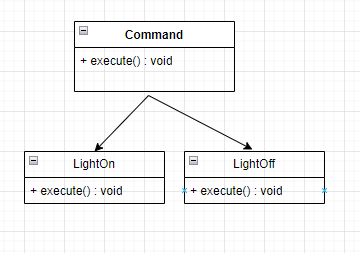
Can make the system overly general and complex for simpler use cases.

# TASK 4: Command Pattern:

**Scenario:**

Implementing a remote control for various electronic devices.

**Class Diagram:**



**Implementation:**  
#include <iostream>

// Command interface

class Command {

public:

virtual void execute() = 0;

};

// Receiver classes

class Light {

public:

void turnOn() { /\* implementation \*/ }

};

class TV {

public:

void turnOff() { /\* implementation \*/ }

};

// Concrete Command classes

class LightOnCommand : public Command {

private:

Light\* light;

public:

LightOnCommand(Light\* l) : light(l) {}

void execute() {

light->turnOn();

}

};

class TVOffCommand : public Command {

private:

TV\* tv;

public:

TVOffCommand(TV\* t) : tv(t) {}

void execute() {

tv->turnOff();

}

};

**Drawbacks:**

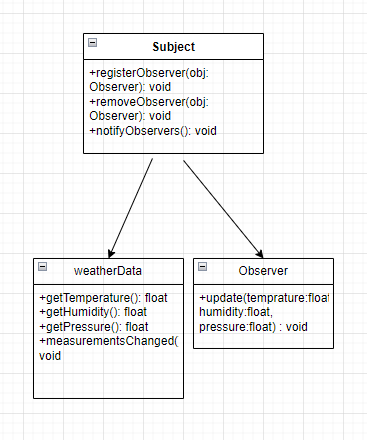
The code can become more complex, especially when dealing with a large number of commands.

# TASK5: Observer Pattern:

**Scenario:**

Creating a weather monitoring system where multiple displays need to be updated whenever the weather changes.

**Class Diagram:**



**Implementation:**

#include <iostream>

#include <vector>

// Observer interface

class Observer {

public:

virtual void update(float temperature, float humidity, float pressure) = 0;

};

// Subject interface

class Subject {

public:

virtual void registerObserver(Observer\* o) = 0;

virtual void removeObserver(Observer\* o) = 0;

virtual void notifyObservers() = 0;

};

// Concrete Observer class

class Display : public Observer {

public:

void update(float temperature, float humidity, float pressure) {

// Code to update display based on new weather data

}

};

// Concrete Subject class

class WeatherData : public Subject {

private:

std::vector<Observer\*> observers;

float temperature;

float humidity;

float pressure;

public:

void registerObserver(Observer\* obj) {

observers.push\_back(o);

}

void removeObserver(Observer\* obj) {

// Remove logic

}

void notifyObservers() {

for (Observer\* o : observers) {

o->update(temperature, humidity, pressure);

}

}

void measurementsChanged() {

// Code to get new measurements

notifyObservers();

}

};

**Drawbacks:**

Inefficient if there are many observers and frequent updates.