Implementing Design Patterns in HMI

Goal: Implement and understand the Singleton, Factory, Observer, and Strategy design patterns.

Steps:

Singleton:

Implement a singleton class HMISystem to manage the overall HMI state.

Factory:

Create a factory to instantiate different types of controls (Button, Slider, etc.).

Observer:

Implement an observer to notify widgets when the system switches modes (e.g., Day/Night mode).

Strategy:

Use the strategy pattern to define different rendering behaviors (e.g., 2D vs. 3D).

Implementation Example (Observer Pattern):

Plain Text

cppCopy codeclass ModeObserver {public: virtual void update(const std::string& mode) = 0;};class Button : public ModeObserver { void update(const std::string& mode) override { if (mode == "Night") { // Adjust visibility } }};

Submission Requirements:

Upload the source code for each task to a version control system (e.g., GitHub or GitLab).

Include comments explaining each step and its relevance to automotive HMI.

Prepare a report summarizing what you learned, challenges faced, and how you solved them.

This assignment ensures a practical understanding of C++ STL utilities and design patterns, aligning them with real-world HMI requirements in the automotive domain.

Program:

#include <iostream>

#include <memory>

#include <vector>

#include <string>

#include <mutex>

using namespace std;

class HMISystem

{

private:

static HMISystem\* instance;

string mode;

mutex mtx;

HMISystem() : mode("Day") {}

public:

static HMISystem\* getInstance()

{

if (instance == nullptr)

{

instance = new HMISystem();

}

return instance;

}

void setMode(const string& newMode)

{

lock\_guard<mutex> lock(mtx);

mode = newMode;

}

string getMode() const

{

return mode;

}

};

HMISystem\* HMISystem::instance = nullptr;

class Control

{

public:

virtual void render() = 0;

virtual ~Control() = default;

};

class Button : public Control

{

public:

void render() override

{

cout << "Rendering Button" << endl;

}

};

class Slider : public Control

{

public:

void render() override

{

cout << "Rendering Slider" << endl;

}

};

class ControlFactory

{

public:

enum class ControlType

{

Button,

Slider

};

static shared\_ptr<Control> createControl(ControlType type)

{

switch (type)

{

case ControlType::Button:

return make\_shared<Button>();

case ControlType::Slider:

return make\_shared<Slider>();

default:

throw invalid\_argument("Invalid Control Type");

}

}

};

class ModeObserver

{

public:

virtual void update(const string& mode) = 0;

};

class ButtonObserver : public ModeObserver

{

public:

void update(const string& mode) override

{

if (mode == "Night")

{

cout << "Button: Adjusting visibility for Night mode." << endl;

}

else

{

cout << "Button: Adjusting visibility for Day mode." << endl;

}

}

};

class SliderObserver : public ModeObserver

{

public:

void update(const string& mode) override

{

if (mode == "Night")

{

cout << "Slider: Dimmed for Night mode." << endl;

}

else

{

cout << "Slider: Brightened for Day mode." << endl;

}

}

};

class HMISystemWithObservers

{

private:

vector<ModeObserver\*> observers;

string mode;

public:

void addObserver(ModeObserver\* observer)

{

observers.push\_back(observer);

}

void setMode(const string& newMode)

{

mode = newMode;

notifyObservers();

}

void notifyObservers()

{

for (auto observer : observers)

{

observer->update(mode);

}

}

};

class RenderStrategy

{

public:

virtual void render() = 0;

virtual ~RenderStrategy() = default;

};

class Render2D : public RenderStrategy

{

public:

void render() override

{

cout << "Rendering in 2D" << endl;

}

};

class Render3D : public RenderStrategy

{

public:

void render() override

{

cout << "Rendering in 3D" << endl;

}

};

class HMISystemWithStrategy

{

private:

unique\_ptr<RenderStrategy> renderStrategy;

public:

void setRenderStrategy(unique\_ptr<RenderStrategy> strategy)

{

renderStrategy = move(strategy);

}

void render()

{

renderStrategy->render();

}

};

int main()

{

HMISystem\* system = HMISystem::getInstance();

system->setMode("Night");

cout << "HMISystem Mode (Singleton): " << system->getMode() << endl;

auto button = ControlFactory::createControl(ControlFactory::ControlType::Button);

button->render();

auto slider = ControlFactory::createControl(ControlFactory::ControlType::Slider);

slider->render();

HMISystemWithObservers hmiWithObservers;

ButtonObserver buttonObserver;

SliderObserver sliderObserver;

hmiWithObservers.addObserver(&buttonObserver);

hmiWithObservers.addObserver(&sliderObserver);

hmiWithObservers.setMode("Night");

hmiWithObservers.setMode("Day");

HMISystemWithStrategy hmiWithStrategy;

hmiWithStrategy.setRenderStrategy(unique\_ptr<RenderStrategy>(new Render2D()));

hmiWithStrategy.render();

hmiWithStrategy.setRenderStrategy(unique\_ptr<RenderStrategy>(new Render3D()));

hmiWithStrategy.render();

return 0;

}

Output:

HMISystem Mode (Singleton): Night

Rendering Button

Rendering Slider

Button: Adjusting visibility for Night mode.

Slider: Dimmed for Night mode.

Button: Adjusting visibility for Day mode.

Slider: Brightened for Day mode.

Rendering in 2D

Rendering in 3D