Week-04 tasks

Task-01:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

// Define the Control structure
struct Control {
    int id;             // Unique ID
    string type;        // "button" or "slider"
    string state;       // "visible", "invisible", "disabled"

    // Overload the equality operator
    bool operator==(const Control& other) const {
        return id == other.id && type == other.type && state == other.state;
    }
};

int main() {
    // Initialize the container with sample controls
```

```
vector<Control> controls = {
   {1, "button", "visible"},
   {2, "button", "invisible"},
   {3, "button", "disabled"},
   {4, "button", "visible"},
   {5, "button", "visible"},
   {6, "slider", "disabled"},
   {7, "slider", "visible"},
   {8, "slider", "invisible"},
   {9, "slider", "disabled"},
   {10, "slider", "visible"}
};

// std::for_each: Iterate through all controls and print their details
for_each(controls.begin(), controls.end(), [](const Control& ctrl) {
   cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << endl;
});

// std::find: Find a control with a specific ID
auto it = find_if(controls.begin(), controls.end(), [](const Control& ctrl) {
   return ctrl.id == 3;
});
if (it != controls.end()) {
   cout << "Control with ID 3 found: Type: " << it->type << ", State: " << it->state << endl;
} else {
   cout << "Control with ID 3 not found." << endl;
```

```
    }

    // std::find_if: Find the first invisible control
    auto invisibleControl = find_if(controls.begin(), controls.end(), [](const Control& ctrl) {
        return ctrl.state == "invisible";
    });
    if (invisibleControl != controls.end()) {
        cout << "First invisible control found: ID: " << invisibleControl->id << endl;
    } else {
        cout << "No invisible control found." << endl;
    }

    // std::adjacent_find: Check for consecutive controls with the same state
    auto adj = adjacent_find(controls.begin(), controls.end(), [](const Control& a, const
Control& b) {
        return a.state == b.state;
    });
    if (adj != controls.end()) {
        cout << "Consecutive controls with the same state found: " << adj->state << endl;
    } else {
        cout << "No consecutive controls with the same state found." << endl;
    }

    // std::count_if: Count the number of visible controls
    int visibleCount = count_if(controls.begin(), controls.end(), [](const Control& ctrl) {
        return ctrl.state == "visible";
```

```
    });
    cout << "Number of visible controls: " << visibleCount << endl;

    // std::count_if: Count sliders that are disabled
    int disabledSliders = count_if(controls.begin(), controls.end(), [](const Control& ctrl) {
        return ctrl.type == "slider" && ctrl.state == "disabled";
    });
    cout << "Number of disabled sliders: " << disabledSliders << endl;

    // std::equal: Compare two subranges of controls to check if they are identical
    bool areEqual = equal(controls.begin(), controls.begin() + 5, controls.begin() + 5);
    cout << "First 5 controls are " << (areEqual ? "identical" : "not identical") << " to the next 5 controls." << endl;

    return 0;
}
```

Output:

```
ID: 1, Type: button, State: visible
ID: 2, Type: button, State: invisible
ID: 3, Type: button, State: disabled
ID: 4, Type: button, State: visible
ID: 5, Type: button, State: visible
ID: 6, Type: slider, State: disabled
ID: 7, Type: slider, State: visible
ID: 8, Type: slider, State: invisible
ID: 9, Type: slider, State: disabled
```

ID: 10, Type: slider, State: visible

Control with ID 3 found: Type: button, State: disabled

First invisible control found: ID: 2

Consecutive controls with the same state found: visible

Number of visible controls: 5

Number of disabled sliders: 2

First 5 controls are not identical to the next 5 controls.

Task-02

```
#include <iostream>

#include <vector>

#include <set>

#include <algorithm>


int main() {

   // Initialize the dynamic widgets in a vector

   std::vector<std::string> dynamicWidgets = {"Speedometer", "Tachometer", "FuelGauge",
"Temperature", "Map"};


   // Initialize the static widgets in a set

   std::set<std::string> staticWidgets = {"Logo", "WarningLights", "BatteryStatus", "Time"};


   // Step 2: Use Iterators to print all dynamic widgets

   std::cout << "Dynamic Widgets: " << std::endl;

   for (auto it = dynamicWidgets.begin(); it != dynamicWidgets.end(); ++it) {

      std::cout << *it << std::endl;
```

```
    }

    // Step 3: Check if "WarningLights" is in the static widgets set using std::set::find
    auto findStatic = staticWidgets.find("WarningLights");
    if (findStatic != staticWidgets.end()) {
        std::cout << "\n'WarningLights' is in the static widgets." << std::endl;
    } else {
        std::cout << "\n'WarningLights' is NOT in the static widgets." << std::endl;
    }

    // Step 4: Advanced Iteration - Combine both containers into a vector using std::copy
    std::vector<std::string> combinedWidgets;

    // Copy dynamic widgets to the combined vector
    std::copy(dynamicWidgets.begin(), dynamicWidgets.end(),
std::back_inserter(combinedWidgets));

    // Copy static widgets to the combined vector
    std::copy(staticWidgets.begin(), staticWidgets.end(),
std::back_inserter(combinedWidgets));

    // Step 5: Use std::find to locate a specific widget in the combined container (e.g., "Map")
    auto findCombined = std::find(combinedWidgets.begin(), combinedWidgets.end(),
"Map");

    if (findCombined != combinedWidgets.end()) {
        std::cout << "\n'Map' found in the combined widget list!" << std::endl;
```

```
    } else {
        std::cout << "\n'Map' not found in the combined widget list." << std::endl;
    }

    // Step 6: Print the combined widgets list
    std::cout << "\nCombined Widget List: " << std::endl;
    for (const auto& widget : combinedWidgets) {
        std::cout << widget << std::endl;
    }

    return 0;
}
```

Output:

```
Dynamic Widgets:
Speedometer
Tachometer
FuelGauge
Temperature
Map

'WarningLights' is in the static widgets.

'Map' found in the combined widget list!

Combined Widget List:
Speedometer
```

Tachometer

FuelGauge

Temperature

Map

BatteryStatus

Logo

Time

WarningLights

=== Code Execution Successful ===

Task-03

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <random>

struct Control {
   int id;
   std::string type;  // "button" or "slider"
   std::string state; // "visible", "invisible", "disabled"
};
```

```
int main() {
    // Initializing controls with a mix of button and slider states
    std::vector<Control> controls = {
        {1, "button", "visible"},
        {2, "slider", "disabled"},
        {3, "button", "invisible"},
        {4, "slider", "visible"},
        {5, "button", "disabled"},
        {6, "slider", "visible"},
        {7, "button", "invisible"},
        {8, "slider", "invisible"},
        {9, "button", "visible"},
        {10, "slider", "disabled"}
    };

    // Step 2: Manipulate Control States
    // 1. Use std::copy to create a backup of the control list
    std::vector<Control> backupControls = controls;

    // Print backup
    std::cout << "Backup of Controls:" << std::endl;
    for (const auto& ctrl : backupControls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state <<
std::endl;
    }
```

```
    // 2. Use std::fill to set all states to "disabled" temporarily
    std::fill(controls.begin(), controls.end(), Control{0, "", "disabled"});
    std::cout << "\nAll controls temporarily set to 'disabled':" << std::endl;
    for (const auto& ctrl : controls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state <<
std::endl;
    }

    // 3. Use std::generate to generate random states ("visible", "invisible", "disabled")
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, 2);

    auto randomState = [&]() {
        switch (dis(gen)) {
            case 0: return "visible";
            case 1: return "invisible";
            case 2: return "disabled";
            default: return "visible";  // Default to visible
        }
    };

    std::generate(controls.begin(), controls.end(), [&]() {
        static int id = 1;
        return Control{id++, "slider", randomState()};
    });
```

```
    std::cout << "\nControls with random states:" << std::endl;
    for (const auto& ctrl : controls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // Step 3: Apply Transformations
    // 4. Use std::transform to change the state of all sliders to "invisible"
    std::transform(controls.begin(), controls.end(), controls.begin(), [](Control& ctrl) {
        if (ctrl.type == "slider") {
            ctrl.state = "invisible";
        }
        return ctrl;
    });

    std::cout << "\nAfter transforming all sliders to 'invisible':" << std::endl;
    for (const auto& ctrl : controls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // 5. Use std::replace to replace "disabled" with "enabled" for testing
    std::replace_if(controls.begin(), controls.end(), [](const Control& ctrl) {
        return ctrl.state == "disabled";
    }, Control{0, "", "enabled"});
```

```
    std::cout << "\nAfter replacing 'disabled' with 'enabled':" << std::endl;
    for (const auto& ctrl : controls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // 6. Use std::remove_if to filter out invisible controls from the list
    controls.erase(std::remove_if(controls.begin(), controls.end(), [](const Control& ctrl) {
        return ctrl.state == "invisible";
    }), controls.end());

    std::cout << "\nAfter removing 'invisible' controls:" << std::endl;
    for (const auto& ctrl : controls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // Step 4: Other Operations
    // 7. Use std::reverse to reverse the control order (for a debug layout)
    std::reverse(controls.begin(), controls.end());
    std::cout << "\nAfter reversing the control order:" << std::endl;
    for (const auto& ctrl : controls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }
```

```
    // 8. Use std::partition to group visible controls together
    auto partitionPoint = std::partition(controls.begin(), controls.end(), [](const Control& ctrl)
{
        return ctrl.state == "visible";
    });


    std::cout << "\nAfter partitioning to group visible controls together:" << std::endl;
    for (const auto& ctrl : controls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state <<
std::endl;
    }


    return 0;
}
```

Output:

Backup of Controls:

ID: 1, Type: button, State: visible

ID: 2, Type: slider, State: disabled

ID: 3, Type: button, State: invisible

ID: 4, Type: slider, State: visible

ID: 5, Type: button, State: disabled

ID: 6, Type: slider, State: visible

ID: 7, Type: button, State: invisible

ID: 8, Type: slider, State: invisible

ID: 9, Type: button, State: visible

ID: 10, Type: slider, State: disabled

All controls temporarily set to 'disabled':

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

ID: 0, Type: , State: disabled

Controls with random states:

ID: 1, Type: slider, State: disabled

ID: 2, Type: slider, State: disabled

ID: 3, Type: slider, State: invisible

ID: 4, Type: slider, State: disabled

ID: 5, Type: slider, State: disabled

ID: 6, Type: slider, State: invisible

ID: 7, Type: slider, State: disabled

ID: 8, Type: slider, State: visible

ID: 9, Type: slider, State: visible

ID: 10, Type: slider, State: disabled

After transforming all sliders to 'invisible':

ID: 1, Type: slider, State: invisible

ID: 2, Type: slider, State: invisible

ID: 3, Type: slider, State: invisible

ID: 4, Type: slider, State: invisible

ID: 5, Type: slider, State: invisible

ID: 6, Type: slider, State: invisible

ID: 7, Type: slider, State: invisible

ID: 8, Type: slider, State: invisible

ID: 9, Type: slider, State: invisible

ID: 10, Type: slider, State: invisible

After replacing 'disabled' with 'enabled':

ID: 1, Type: slider, State: invisible

ID: 2, Type: slider, State: invisible

ID: 3, Type: slider, State: invisible

ID: 4, Type: slider, State: invisible

ID: 5, Type: slider, State: invisible

ID: 6, Type: slider, State: invisible

ID: 7, Type: slider, State: invisible

ID: 8, Type: slider, State: invisible

ID: 9, Type: slider, State: invisible

ID: 10, Type: slider, State: invisible

After removing 'invisible' controls:

After reversing the control order:

After partitioning to group visible controls together:

=== Code Execution Successful ===

Task-04

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>

struct Control {
    int id;
    std::string type;  // "button" or "slider"
    std::string state; // "visible", "invisible", "disabled"

    // Define less-than operator to allow sorting by ID
    bool operator<(const Control& other) const {
        return id < other.id;
    }
};

int main() {
    // Initialize two lists of controls with different IDs
    std::vector<Control> controls1 = {
```

```
    {1, "button", "visible"},
    {3, "slider", "disabled"},
    {5, "button", "invisible"},
    {7, "slider", "visible"}
  };

  std::vector<Control> controls2 = {
    {2, "slider", "disabled"},
    {4, "button", "visible"},
    {6, "slider", "visible"},
    {8, "button", "disabled"}
  };

  // Step 2: Sorting the controls
  // Sort controls by ID using std::sort
  std::sort(controls1.begin(), controls1.end());
  std::sort(controls2.begin(), controls2.end());

  // Print sorted controls1
  std::cout << "Sorted controls1 by ID:" << std::endl;
  for (const auto& ctrl : controls1) {
    std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state <<
std::endl;
  }

  // Step 3: Use std::stable_sort to maintain relative order for controls with equal IDs
```

```
    // For the sake of this example, we manually introduce a control with an equal ID in both lists
    controls1.push_back({3, "button", "disabled"});
    std::stable_sort(controls1.begin(), controls1.end());

    // Print sorted controls1 with stable_sort
    std::cout << "\nControls after stable_sort (ID 3 should maintain relative order):" << std::endl;
    for (const auto& ctrl : controls1) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // Step 4: Binary Search
    // Use std::lower_bound to find a control by ID
    auto it = std::lower_bound(controls1.begin(), controls1.end(), Control{3, "", ""});
    if (it != controls1.end() && it->id == 3) {
        std::cout << "\nControl with ID 3 found (lower_bound): ID: " << it->id << ", Type: " << it->type << ", State: " << it->state << std::endl;
    }

    // Use std::upper_bound to find the next control after ID 3
    auto it_upper = std::upper_bound(controls1.begin(), controls1.end(), Control{3, "", ""});
    if (it_upper != controls1.end()) {
        std::cout << "Control after ID 3 (upper_bound): ID: " << it_upper->id << ", Type: " << it_upper->type << ", State: " << it_upper->state << std::endl;
    }
```

```
    // Step 5: Merging two sorted lists of controls
    std::vector<Control> mergedControls;
    std::merge(controls1.begin(), controls1.end(), controls2.begin(), controls2.end(), std::back_inserter(mergedControls));

    std::cout << "\nMerged controls:" << std::endl;
    for (const auto& ctrl : mergedControls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // Step 6: Use std::inplace_merge to merge two sorted segments within the same list
    std::vector<Control> controls3 = {
        {1, "button", "visible"},
        {3, "slider", "disabled"},
        {5, "button", "invisible"},
        {7, "slider", "visible"}
    };

    std::vector<Control> controls4 = {
        {2, "slider", "disabled"},
        {4, "button", "visible"},
        {6, "slider", "visible"},
        {8, "button", "disabled"}
    };
```

```
    // Merge controls3 and controls4 into one vector using inplace_merge
    controls3.insert(controls3.end(), controls4.begin(), controls4.end());
    std::inplace_merge(controls3.begin(), controls3.begin() + 4, controls3.end());

    std::cout << "\nControls after inplace_merge:" << std::endl;
    for (const auto& ctrl : controls3) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // Step 7: Set Operations
    // Use std::set_union to find all unique controls in the merged list
    std::set<Control> set1(controls1.begin(), controls1.end());
    std::set<Control> set2(controls2.begin(), controls2.end());
    std::vector<Control> unionControls;

    std::set_union(set1.begin(), set1.end(), set2.begin(), set2.end(), std::back_inserter(unionControls));

    std::cout << "\nUnion of controls (unique IDs only):" << std::endl;
    for (const auto& ctrl : unionControls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state << std::endl;
    }

    // Use std::set_intersection to find common controls between the two sets
```

```
    std::vector<Control> intersectionControls;
    std::set_intersection(set1.begin(), set1.end(), set2.begin(), set2.end(),
std::back_inserter(intersectionControls));

    std::cout << "\nIntersection of controls (common IDs):" << std::endl;
    for (const auto& ctrl : intersectionControls) {
        std::cout << "ID: " << ctrl.id << ", Type: " << ctrl.type << ", State: " << ctrl.state <<
std::endl;
    }

    return 0;
}
```

Output:

Sorted controls1 by ID:

ID: 1, Type: button, State: visible

ID: 3, Type: slider, State: disabled

ID: 5, Type: button, State: invisible

ID: 7, Type: slider, State: visible

Controls after stable_sort (ID 3 should maintain relative order):

ID: 1, Type: button, State: visible

ID: 3, Type: slider, State: disabled

ID: 3, Type: button, State: disabled

ID: 5, Type: button, State: invisible

ID: 7, Type: slider, State: visible

Control with ID 3 found (lower_bound): ID: 3, Type: slider, State: disabled

Control after ID 3 (upper_bound): ID: 5, Type: button, State: invisible

Merged controls:

ID: 1, Type: button, State: visible

ID: 2, Type: slider, State: disabled

ID: 3, Type: slider, State: disabled

ID: 3, Type: button, State: disabled

ID: 4, Type: button, State: visible

ID: 5, Type: button, State: invisible

ID: 6, Type: slider, State: visible

ID: 7, Type: slider, State: visible

ID: 8, Type: button, State: disabled

Controls after inplace_merge:

ID: 1, Type: button, State: visible

ID: 2, Type: slider, State: disabled

ID: 3, Type: slider, State: disabled

ID: 4, Type: button, State: visible

ID: 5, Type: button, State: invisible

ID: 6, Type: slider, State: visible

ID: 7, Type: slider, State: visible

ID: 8, Type: button, State: disabled

Union of controls (unique IDs only):

ID: 1, Type: button, State: visible

ID: 2, Type: slider, State: disabled

ID: 3, Type: slider, State: disabled

ID: 4, Type: button, State: visible

ID: 5, Type: button, State: invisible

ID: 6, Type: slider, State: visible

ID: 7, Type: slider, State: visible

ID: 8, Type: button, State: disabled

Intersection of controls (common IDs):

=== Code Execution Successful ===

Task-05

```
#include <iostream>
#include <vector>
#include <memory>
#include <string>
#include <algorithm>

// ====================== Singleton Pattern ======================
```

```
class HMISystem {
private:
   static HMISystem* instance;

   // Private constructor to prevent instantiation
   HMISystem() {}

public:
   static HMISystem* getInstance() {
      if (instance == nullptr) {
         instance = new HMISystem();
      }
      return instance;
   }

   void displayMode(const std::string& mode) {
      std::cout << "HMI is now in " << mode << " mode." << std::endl;
   }
};

// Initialize the static member
HMISystem* HMISystem::instance = nullptr;



// ==================== Factory Pattern ====================
```

```
// Abstract Control class
class Control {
public:
    virtual void render() = 0;
};

// Concrete Control classes
class Button : public Control {
public:
    void render() override {
        std::cout << "Rendering Button" << std::endl;
    }
};

class Slider : public Control {
public:
    void render() override {
        std::cout << "Rendering Slider" << std::endl;
    }
};

// Factory class
class ControlFactory {
public:
    std::shared_ptr<Control> createControl(const std::string& type) {
        if (type == "Button") {
```

```
        return std::make_shared<Button>();
    } else if (type == "Slider") {
        return std::make_shared<Slider>();
    } else {
        return nullptr;
    }
  }
};



// ===================== Observer Pattern =====================

// Observer interface
class ModeObserver {
public:
  virtual void update(const std::string& mode) = 0;
};

// Concrete Observer classes for ModeObserver
class ButtonObserver : public ModeObserver {
public:
  void update(const std::string& mode) override {
    if (mode == "Night") {
      std::cout << "Button visibility set to low (Night mode)" << std::endl;
    } else {
      std::cout << "Button visibility set to normal (Day mode)" << std::endl;
```

```
        }
    }
};

class SliderObserver : public ModeObserver {
public:
    void update(const std::string& mode) override {
        if (mode == "Night") {
            std::cout << "Slider visibility set to low (Night mode)" << std::endl;
        } else {
            std::cout << "Slider visibility set to normal (Day mode)" << std::endl;
        }
    }
};

// Subject class (ModeNotifier)
class ModeNotifier {
private:
    std::vector<ModeObserver*> observers;
    std::string mode;

public:
    void addObserver(ModeObserver* observer) {
        observers.push_back(observer);
    }
```

```
    void removeObserver(ModeObserver* observer) {
        observers.erase(std::remove(observers.begin(), observers.end(), observer),
observers.end());
    }

    void setMode(const std::string& newMode) {
        mode = newMode;
        notifyObservers();
    }

    void notifyObservers() {
        for (auto observer : observers) {
            observer->update(mode);
        }
    }
};



// ====================== Strategy Pattern ======================

// Abstract Strategy interface
class RenderStrategy {
public:
    virtual void render() = 0;
};
```

```
// Concrete Strategy for 2D Rendering
class Render2D : public RenderStrategy {
public:
    void render() override {
        std::cout << "Rendering in 2D" << std::endl;
    }
};

// Concrete Strategy for 3D Rendering
class Render3D : public RenderStrategy {
public:
    void render() override {
        std::cout << "Rendering in 3D" << std::endl;
    }
};

// Context class (HMISystem) that uses a RenderStrategy
class HMIContext {
private:
    std::shared_ptr<RenderStrategy> strategy;

public:
    void setRenderStrategy(std::shared_ptr<RenderStrategy> newStrategy) {
        strategy = newStrategy;
    }
```

```
    void render() {
        strategy->render();
    }
};

// ====================== Main Function ====================

int main() {
    // ====================== Singleton Usage ====================
    std::cout << "Singleton Example:" << std::endl;
    HMISystem* hmi = HMISystem::getInstance();
    hmi->displayMode("Day");

    // ====================== Factory Usage ====================
    std::cout << "\nFactory Example:" << std::endl;
    ControlFactory factory;
    auto button = factory.createControl("Button");
    button->render();  // Output: Rendering Button

    auto slider = factory.createControl("Slider");
    slider->render();  // Output: Rendering Slider

    // ====================== Observer Usage ====================
    std::cout << "\nObserver Example:" << std::endl;
    ModeNotifier modeNotifier;
```

```
    ButtonObserver buttonObserver;
    SliderObserver sliderObserver;

    modeNotifier.addObserver(&buttonObserver);
    modeNotifier.addObserver(&sliderObserver);

    // Change to Night mode
    modeNotifier.setMode("Night");

    // Change to Day mode
    modeNotifier.setMode("Day");

    // ====================== Strategy Usage ======================
    std::cout << "\nStrategy Example:" << std::endl;
    HMIContext context;

    // Set 2D rendering strategy
    context.setRenderStrategy(std::make_shared<Render2D>());
    context.render();  // Output: Rendering in 2D

    // Switch to 3D rendering strategy
    context.setRenderStrategy(std::make_shared<Render3D>());
    context.render();  // Output: Rendering in 3D

    return 0;
}
```

Output:

Singleton Example:

HMI is now in Day mode.

Factory Example:

Rendering Button

Rendering Slider

Observer Example:

Button visibility set to low (Night mode)

Slider visibility set to low (Night mode)

Button visibility set to normal (Day mode)

Slider visibility set to normal (Day mode)

Strategy Example:

Rendering in 2D

Rendering in 3D

=== Code Execution Successful ===