Lecture 16

Dr. Umair Rehman

Agenda

- Command Design Pattern
  - Direct Invocation Approach vs Command Design Pattern

Behavioral Design Patterns

- Focused on managing communication and interactions between objects.
- Promoting flexibility and efficient communication.
- By using behavioral patterns
  - Objects can work together more seamlessly
  - Maintaining clear responsibilities and reducing dependencies

Popular Behavioral Design Patterns

- Iterator: Sequential access to collection.
- Chain of Responsibility: Pass request along handlers.
- Command: Encapsulate actions as objects.
- Observer: Notify on state changes.
- State: Alter behavior by state.
- Strategy: Switchable algorithms at runtime.
- Template Method: Algorithm skeleton, modifiable steps.

Example: Universal Home Automation Remote

- Remote designed for controlling multiple home devices
- Supports current and future devices (e.g., lights, fans, TV, thermostat)

Example: Universal Home Automation Remote

- Perform Multiple Actions on Each Device
    - Basic control (on/off)
    - Advanced settings adjustments (e.g., fan speed, dimming lights)

Example: Universal Home Automation Remote

- Flexibility & Scalability
  - Seamless addition of new devices and actions
  - Minimal changes needed for future expansions

Direct Invocation Approach

- The Direct Invocation approach allows each button on the remote to directly invoke a specific command on a device.

- Buttons on the remote are directly mapped to device-specific commands (e.g., "Turn on Light," "Adjust Fan Speed").

Direct Invocation Approach

```cpp
#include <iostream>

// Device classes
class Light {
public:
    void turnOn() { std::cout << "The light is on." << std::endl; }
    void turnOff() { std::cout << "The light is off." << std::endl; }
};

```

Direct Invocation Approach

```

class Fan {
public:
    void turnOn() { std::cout << "The fan is on." << std::endl; }
    void turnOff() { std::cout << "The fan is off." << std::endl; }
};

```

Command Design Pattern

```cpp
// Invoker class (RemoteControl) - tightly coupled to specific devices
class RemoteControl {
private:
    Light* light;
    Fan* fan;

public:
    RemoteControl(Light* l, Fan* f) : light(l), fan(f) {}

    // Methods to control specific devices
    void turnOnLight() { light->turnOn(); }
    void turnOffLight() { light->turnOff(); }
    void turnOnFan() { fan->turnOn(); }
    void turnOffFan() { fan->turnOff(); }
};

```

Direct Invocation Approach

```

// Client code
int main() {
    Light light;
    Fan fan;
    RemoteControl remote(&light, &fan);

    // Directly controlling the light and fan through RemoteControl
    remote.turnOnLight();  // Output: "The light is on."
    remote.turnOffLight(); // Output: "The light is off."
    remote.turnOnFan();    // Output: "The fan is on."
    remote.turnOffFan();   // Output: "The fan is off."

    return 0;
}

```

Cons of Direct Invocation Approach

- Tight Coupling:
  - `RemoteControl` depends directly on Light and Fan.
- Dependency:
  - `RemoteControl` needs to know specifics of each device it controls.
- Impact of Changes:
  - Modifications in Light or Fan could require adjustments in `RemoteControl`.

Cons of Direct Invocation Approach

- Scalability Issue:
  - Adding new devices (e.g., TV) requires altering `RemoteControl` to accommodate device-specific methods like `turnOnTV`() and `turnOffTV()`.
- Maintenance Difficulty:
  - `RemoteControl` grows longer and harder to maintain with each addition.

Cons of Direct Invocation Approach

- No Reusability:
    - Actions (e.g., turning on/off) are hardcoded into `RemoteControl`.
- Duplication Issue:
    - Creating different remotes for various rooms requires replicating similar code.
- Reuse Constraints:
    - Code is not easily reusable across different remote types or future devices.

Cons of Direct Invocation Approach

- Single Responsibility Principal Violation:
  - `RemoteControl` handles multiple responsibilities.
  - Manages both action execution and device-specific details.
  - Changes in device behavior could disrupt `RemoteControl` functionality.

Refactor with Command Design Pattern

- The Command pattern is a behavioral design pattern

- Transforms a request or operation into a distinct object
    - Encapsulating all necessary details to execute the action

- Benefits:
    - Easier to add new commands.
    - Supports undo/redo functionality.
    - Enables action queuing.
    - Allows command history logging.

Command Design Pattern

```cpp
#include <iostream>

// Command interface
class Command {
public:
    virtual void execute() = 0;
    virtual ~Command() = default;
};
```

Command Design Pattern

```
// Receiver class for Light
class Light {
public:
    void turnOn() { std::cout << "The light is on." << std::endl; }
    void turnOff() { std::cout << "The light is off." << std::endl; }
};
```

Command Design Pattern

```cpp
// Receiver class for Fan
class Fan {
public:
    void turnOn() { std::cout << "The fan is on." << std::endl; }
    void turnOff() { std::cout << "The fan is off." << std::endl; }
};
```

Command Design Pattern

```cpp
// Concrete Command to turn on the light
class LightOnCommand : public Command {
private:
    Light* light;
public:
    LightOnCommand(Light* l) : light(l) {}
    void execute() override { light->turnOn(); }
};

```

Command Design Pattern

```cpp
// Concrete Command to turn off the light
class LightOffCommand : public Command {
private:
    Light* light;
public:
    LightOffCommand(Light* l) : light(l) {}
    void execute() override { light->turnOff(); }
};

```

Command Design Pattern

```cpp
// Concrete Command to turn on the fan
class FanOnCommand : public Command {
private:
    Fan* fan;
public:
    FanOnCommand(Fan* f) : fan(f) {}
    void execute() override { fan->turnOn(); }
};
```

Command Design Pattern

```cpp
// Concrete Command to turn off the fan
class FanOffCommand : public Command {
private:
    Fan* fan;
public:
    FanOffCommand(Fan* f) : fan(f) {}
    void execute() override { fan->turnOff(); }
};

```

Command Design Pattern

```cpp
// Invoker class (RemoteControl)
class RemoteControl {
private:
    Command* command;
public:
    void setCommand(Command* cmd) { command = cmd; }
    void pressButton() {
        if(command) {
            command->execute();
        }
        else {
            std::cout << "No command set." << std::endl;
        }
    }
};

```

Command Design Pattern

```
// Client code
int main() {
    // Create receiver objects
    Light light;
    Fan fan;

    // Create command objects
    LightOnCommand lightOn(&light);
    LightOffCommand lightOff(&light);
    FanOnCommand fanOn(&fan);
    FanOffCommand fanOff(&fan);

    // Create invoker
    RemoteControl remote;
```

Command Design Pattern

```
    // Turn on the light
    remote.setCommand(&lightOn);
    remote.pressButton();  // Output: "The light is on."

    // Turn off the light
    remote.setCommand(&lightOff);
    remote.pressButton();  // Output: "The light is off."

    // Turn on the fan
    remote.setCommand(&fanOn);
    remote.pressButton();  // Output: "The fan is on."

    // Turn off the fan
    remote.setCommand(&fanOff);
    remote.pressButton();  // Output: "The fan is off."

    return 0;
}

```

1. Client Creates the Receiver (Light)

```
Light light;

```

- The Light object (light) is created.
- This object has methods like `turnOn()` and `turnOff()` to perform specific actions.
- Light object is the receiver because
  - Ultimately carry out the requested action

2. Client Creates the Command (LightOnCommand)

```
LightOnCommand lightOn(&light);

```

- The client creates a `LightOnCommand` object (`lightOn`) and provides it with a reference to the light object (the receiver).
- Dependency Injection occurs here:
  - the light object (dependency) is injected into the `LightOnCommand` through the constructor.
- This means `lightOn` now "knows" which `Light` instance to control.

3. Client Creates the Invoker (RemoteControl)

```
RemoteControl remote;

```

- The client creates the `RemoteControl` object (remote), which will act as the invoker.
- The `RemoteControl` doesn't yet know what command it will execute
  - it will simply be given a command later.

4. Client Sets the Command in the Invoker

```
remote.setCommand(&lightOn);

```

- The client tells the remote to store the lightOn command by calling `remote.setCommand(&lightOn);`.
- This means the remote is now set up with a command to execute
  - But doesn't yet know what the command does

5. Invoker (RemoteControl) Executes the Command

```
void pressButton() {
    if (command) {
        command->execute(); // Executes the stored command
    }
}

```

- The `RemoteControl’s` `pressButton()` method is called.
- Inside this method, remote calls `execute()` on the stored command (`lightOn`).
- The `RemoteControl` doesn’t know what `execute()` does or which device it controls.
- It just knows that it’s supposed to call `execute()` on the command when the button is pressed.

6. Receiver (Light) Performs the Action:

```cpp
void execute() override {
    light->turnOn(); // Calls the turnOn method on the Light receiver
}

```

- When `execute()` is called on `lightOn`, it performs its predefined action.
- In this case, `LightOnCommand` calls the `turnOn()` method on the Light object (the receiver) it controls.
- This step enables the command to tell the receiver to perform the specific action it's designed for.

7. Command Executes the Action on the Receiver:

```
void turnOn() {
    std::cout << "The light is on." << std::endl;
}

```

- The Light object (receiver) receives the call to `turnOn()` and performs the actual action of turning on the light.

- This step completes the action requested by the client, and we see the output "The light is on."

Advantages of Command Pattern

- Loose Coupling:
    - The `RemoteControl` doesn't know about the specific devices'
    - It only knows about commands.
    - This makes it easier to add new devices without changing `RemoteControl`.

Advantages of Command Pattern

- Scalability:
  - New commands (for new devices or actions) can be added independently without modifying the `RemoteControl` class.

Advantages of Command Pattern

- Reusability:
  - Each command class is independent and can be reused across different contexts or with different invokers (e.g., other types of remotes).

Advantages of Command Pattern

- Flexibility:
  - Allows us to implement additional features like undo/redo or action logging by adding methods to commands or wrapping them in higher-level structures, without modifying existing code.

Conclusion

- Command Design Pattern
  - Direct Invocation Approach vs Command Design Pattern