The command pattern

START

Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040

The
command pattern

START

Example:
Order at Restaurant

Definition:

The Command Pattern is a behavioral design pattern that allows us to encapsulate a request as an object. This object can then be used to parameterize and execute different requests, all while decoupling the request issuer from the request executor..

Structure of Command Pattern

The Command interface defines the basic operations that all ConcreteCommand classes must implement. These operations typically include an execute method that takes in a Receiver object as a parameter.

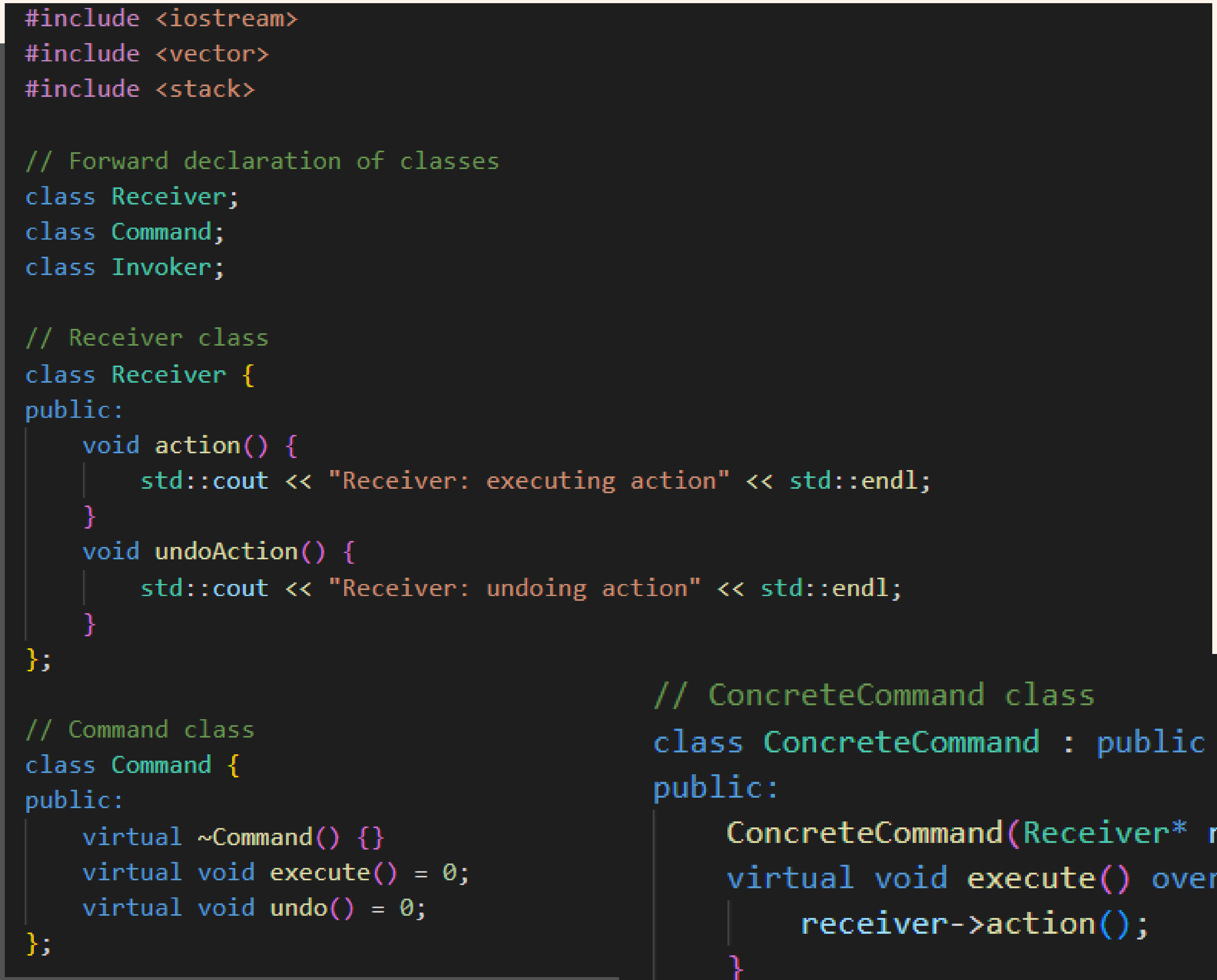
ConcreteCommand classes implement the Command interface and define the specific operations that will be executed when the command is executed. Each ConcreteCommand class is typically responsible for executing a single operation.

Structure of Command Pattern

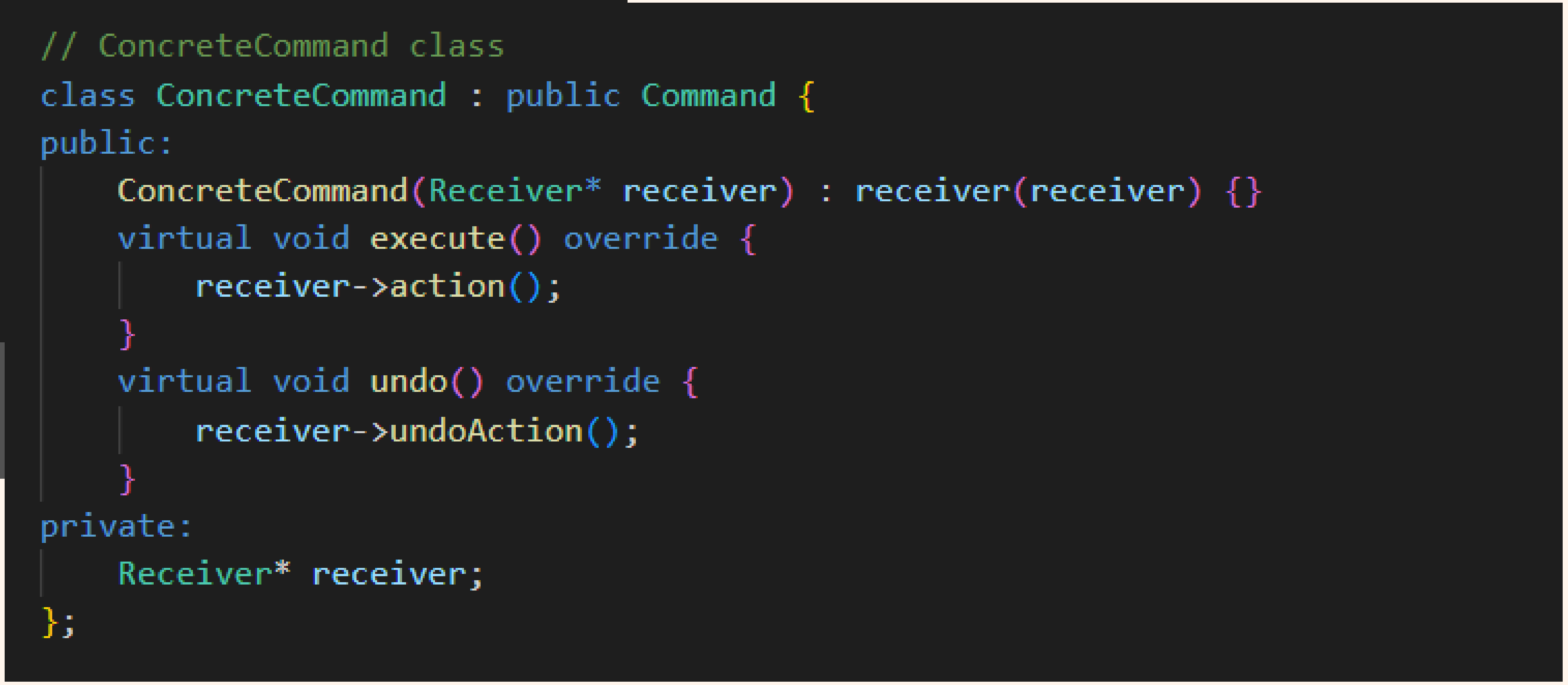
The Receiver class is responsible for defining the actions that will be taken when a ConcreteCommand object is executed. This class typically contains the business logic that will be executed in response to a command.

The Invoker class is responsible for executing commands. This class typically maintains a list of ConcreteCommand objects and provides methods for adding and removing commands from the list. When the Invoker class executes a command, it calls the execute method of the appropriate ConcreteCommand object.

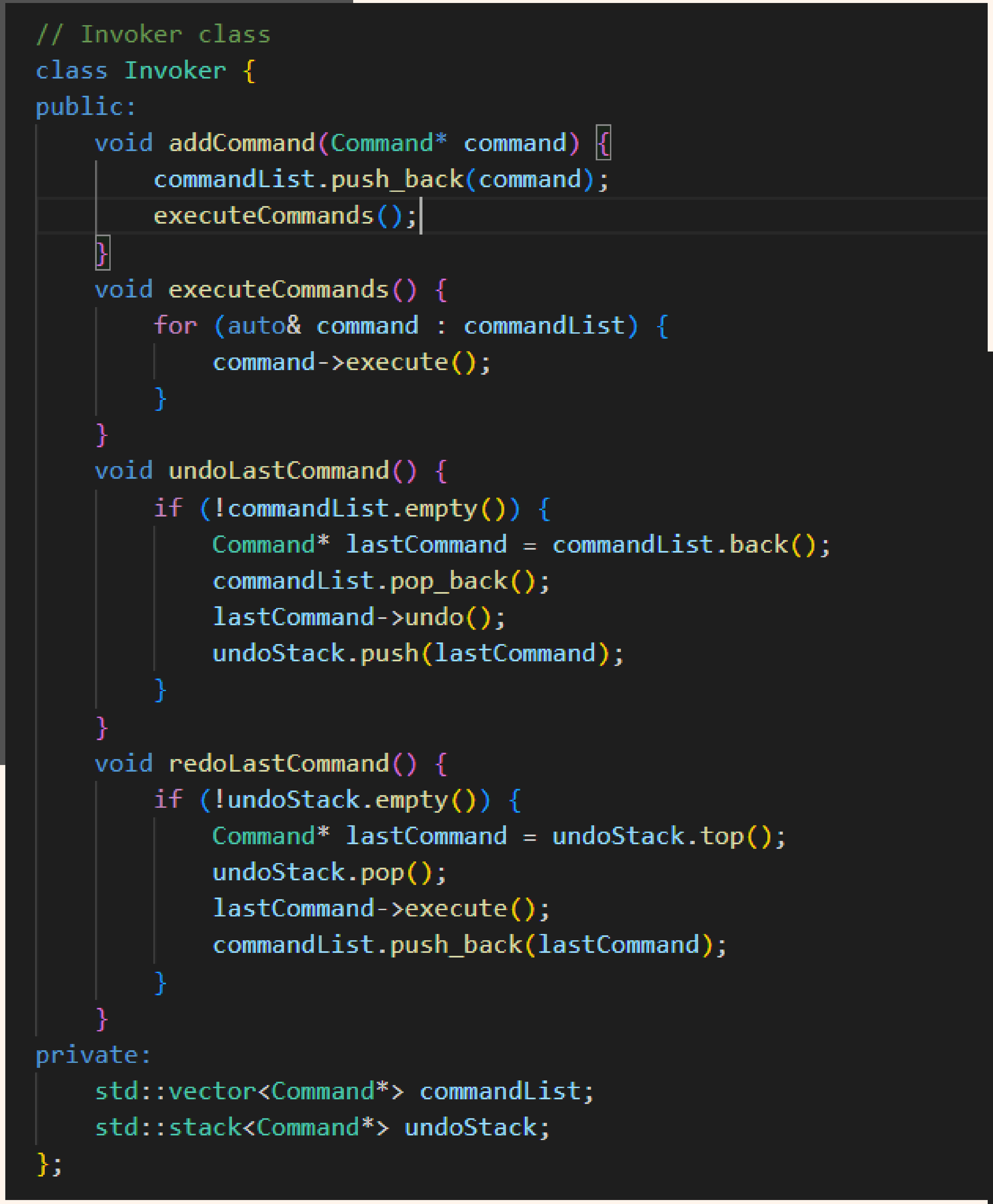
Implemetation



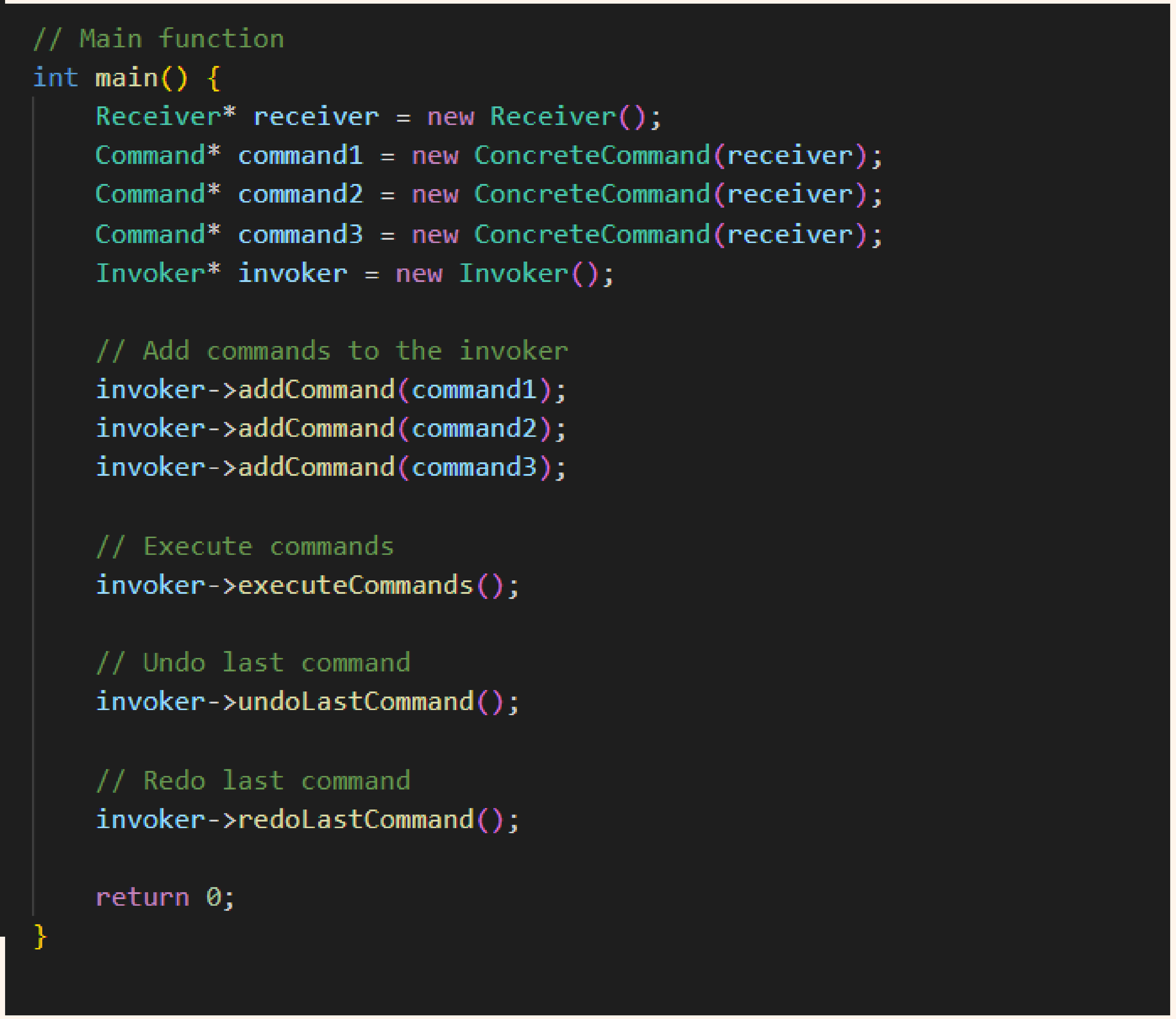
```cpp
#include <iostream>
#include <vector>
#include <stack>

// Forward declaration of classes
class Receiver;
class Command;
class Invoker;

// Receiver class
class Receiver {
public:
    void action() {
        std::cout << "Receiver: executing action" << std::endl;
    }
    void undoAction() {
        std::cout << "Receiver: undoing action" << std::endl;
    }
};

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



```cpp
// ConcreteCommand class
class ConcreteCommand : public Command {
public:
    ConcreteCommand(Receiver* receiver) : receiver(receiver) {}
    virtual void execute() override {
        receiver->action();
    }
    virtual void undo() override {
        receiver->undoAction();
    }
private:
    Receiver* receiver;
};
```



```cpp
// Invoker class
class Invoker {
public:
    void addCommand(Command* command) {
        commandList.push_back(command);
        executeCommands();
    }
    void executeCommands() {
        for (auto& command : commandList) {
            command->execute();
        }
    }
    void undoLastCommand() {
        if (!commandList.empty()) {
            Command* lastCommand = commandList.back();
            commandList.pop_back();
            lastCommand->undo();
            undoStack.push(lastCommand);
        }
    }
    void redoLastCommand() {
        if (!undoStack.empty()) {
            Command* lastCommand = undoStack.top();
            undoStack.pop();
            lastCommand->execute();
            commandList.push_back(lastCommand);
        }
    }
private:
    std::vector<Command*> commandList;
    std::stack<Command*> undoStack;
};
```



```cpp
// Main function
int main() {
    Receiver* receiver = new Receiver();
    Command* command1 = new ConcreteCommand(receiver);
    Command* command2 = new ConcreteCommand(receiver);
    Command* command3 = new ConcreteCommand(receiver);
    Invoker* invoker = new Invoker();

    // Add commands to the invoker
    invoker->addCommand(command1);
    invoker->addCommand(command2);
    invoker->addCommand(command3);

    // Execute commands
    invoker->executeCommands();

    // Undo last command
    invoker->undoLastCommand();

    // Redo last command
    invoker->redoLastCommand();

    return 0;
}
```

Avantages

1 Flexibility

2 Decoupling

3 Undo/Redo operations

4 Logging and Auditing

Disavantages

1 Complexity

2 Performance overhead

3 Increased memory usage

4 Difficulty in implementing some command

THANK YOU!