Spike: Spike No.13
Title: Task 13 – Command Pattern

Author: Khang Trinh - 102118468

Goals / Deliverables:

The goal of this spike is to teach the developer how to implement the command system.

Technologies, Tools, and Resources used:

- Visual Studio 2017

Useful Links:

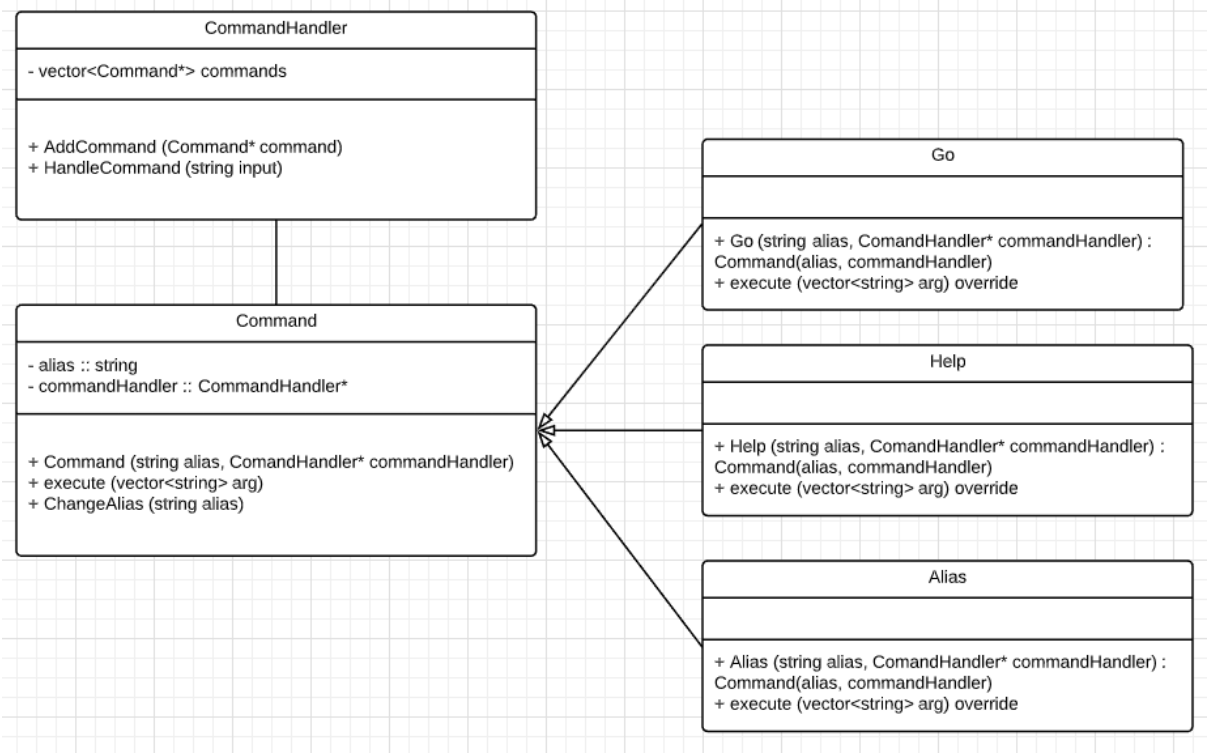
1. Explanation and example of implementation: https://gameprogrammingpatterns.com/command.html

Tasks undertaken:

Implementing the command pattern requires 3 components:

- An abstract class holding the structure of a command
- The implementations of the command (the commands themselves)
- A manager class that contains all the available commands and handles reading input and executing those commands

For this spike, we'll create basic command handling for a text-based adventure game. It'll have 3 commands: Go, Help, and Change.



If you've ever used a bot in any social media before (Discord bots, Twitch bots, etc.), you'll know that to make a bot execute a command, you need to provide it an alias (the name of the command), and any argument that it requires to execute said command.

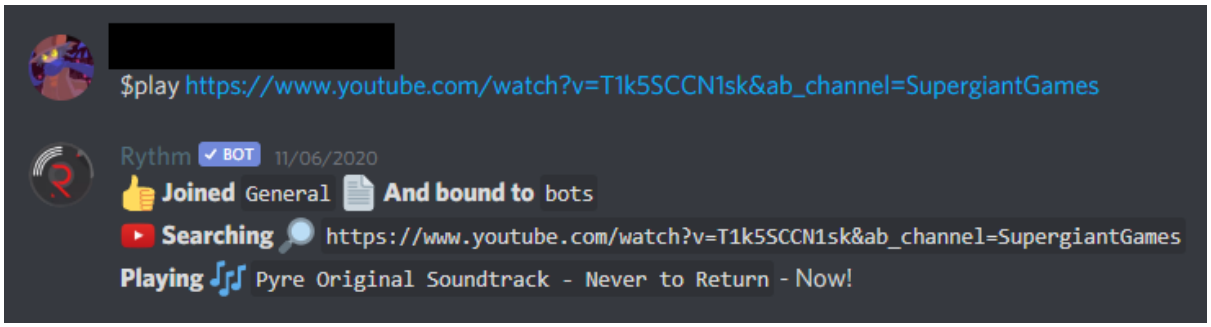


Fig 1. An example of a bot taking a command "play" to play a youtube link

Step 1: Command structure

All basic commands need to have an alias to identify itself, and a method to execute a command when called. Since different commands will do different things, we'll start by implementing a generic base class with the execution method being virtual.

```
class Command {
public:
    string alias;
    CommandHandler* commandHandler;

    Command(string alias, CommandHandler* commandHandler);
    virtual void execute(vector<string> input) = 0;
    void ChangeAlias(string alias) {
        this->alias = alias;
    }
};
```

It'll also need a reference to the CommandHandler class to add itself to the list of commands when initialized.

```
Command::Command(string alias, CommandHandler* commandHandler) {
    this->alias = alias;
    this->commandHandler = commandHandler;
    this->commandHandler->AddCommand(this);
}
```

Step 2: Command Handler

Then we need to create the Command Handler, adding a method to add commands into the list and a method to "read" and execute the input from the user. To "read" an input, we need to break the string down, splitting it by whitespace and putting them in a vector so we can iterate through later to get the arguments. For this example, we assume the first element of the input will always be the alias, so we'll compare that to the command's alias. If there's a match, we'll call execute() on that command, passing the entire vector into it.

```
class CommandHandler {
public:
    vector<Command*> commands;
    void AddCommand(Command* command) {
        commands.push_back(command);
    }

    void HandleCommand(string input) {
        istringstream buf(input);
        istream_iterator<string> beg(buf), end;

        vector<string> args(beg, end);

        for (Command* command : commands) {
            if (command->alias == args[0])
                command->execute(args);
        }
    }
};
```

Step 3: Implementing the *actual* commands

With the way we have the Command class setup, creating a new command is as simple as providing a constructor (inheriting it from the Command class), and an override method of the execute() function. You'll notice that each command may use one, or more, or no arguments.

```
class Go : public Command {
public:
    Go(string alias, CommandHandler* commandHandler) : Command(alias, commandHandler) {
    }

    void execute(vector<string> arg) override {
        cout << alias << " command received. Parameter: " << arg[1] << endl;
    }
};
```

```
class Alias : public Command {
public:
    Alias(string alias, CommandHandler* commandHandler) : Command(alias, commandHandler) {
    }

    void execute(vector<string> arg) override {
        for (Command* command : commandHandler->commands) {
            if (arg[1] == command->alias)
                command->ChangeAlias(arg[2]);
        }
        cout << alias << " command received. Command " << arg[1] << " changed to " << arg[2] << endl;
    }
};
```

```
class Help : public Command {
public:
    Help(string alias, CommandHandler* commandHandler) : Command(alias, commandHandler) {
    }

    void execute(vector<string> arg) override {
        cout << alias << " command received. Here are the available commands:" << endl;
        cout << commandHandler->commands[1]->alias << ": List of commands." << endl;
        cout << commandHandler->commands[0]->alias << " [location to change to]: Change location." << endl;
        cout << commandHandler->commands[2]->alias << " [command alias] [alias to change to]: Change command's alias." << endl;
    }
};
```

Step 4: Getting the user input

Note that getlin() is used, rather than cin >> input. This is to make sure everything the user types is recorded.

```
int main() {
    CommandHandler handler;
    Go* goCmd = new Go("go", &handler);
    Help* helpCmd = new Help("help", &handler);
    Alias* aliasCmd = new Alias("change", &handler);

    string input;

    while (true)
    {
        std::getline(std::cin, input);
        handler.HandleCommand(input);
    }
    return 0;
}
```

This is what the output should look like

```
D:\Uni\Sem 6\COS30031-Games Programming\Unit_Repo\13 - Spike - Command Pattern\Debug\Task 13.exe
go west
go command received. Parameter: west
change go move
change command received. Command go changed to move
move east
move command received. Parameter: east
```

What we found

- As demonstrated above, this pattern works best when you have inputs that may be changed in the future. This could be keyboard event or a command alias being different but still execute the same function.

Recommendation

- Generally a command will come in the form of “[alias] [arg 1] [arg 2] [arg n]” (see example of the music bot at the beginning), so we assumed that this is what the user will enter. Make sure to always communicate the convention you want them to use.
- You may want to implement an input sanitizer to keep the user from exploiting your code (especially if you’ll be dealing with handling a database).
- On that same note, you may also want to put a rule to how many arguments a command should take, so that the program will return an error to the user, rather than executing with missing/too many arguments.