

Spike: Task 12.P

Title: Command Pattern

Author: Mitchell Wright, 100595153

Goals / deliverables:

To demonstrate an understanding of collection types, their strengths and weaknesses, and apply that knowledge in creating a functional inventory system.

Items created during task:

- Code, see: \12 - Spike – Command Pattern\zorkish

Technologies, Tools, and Resources used:

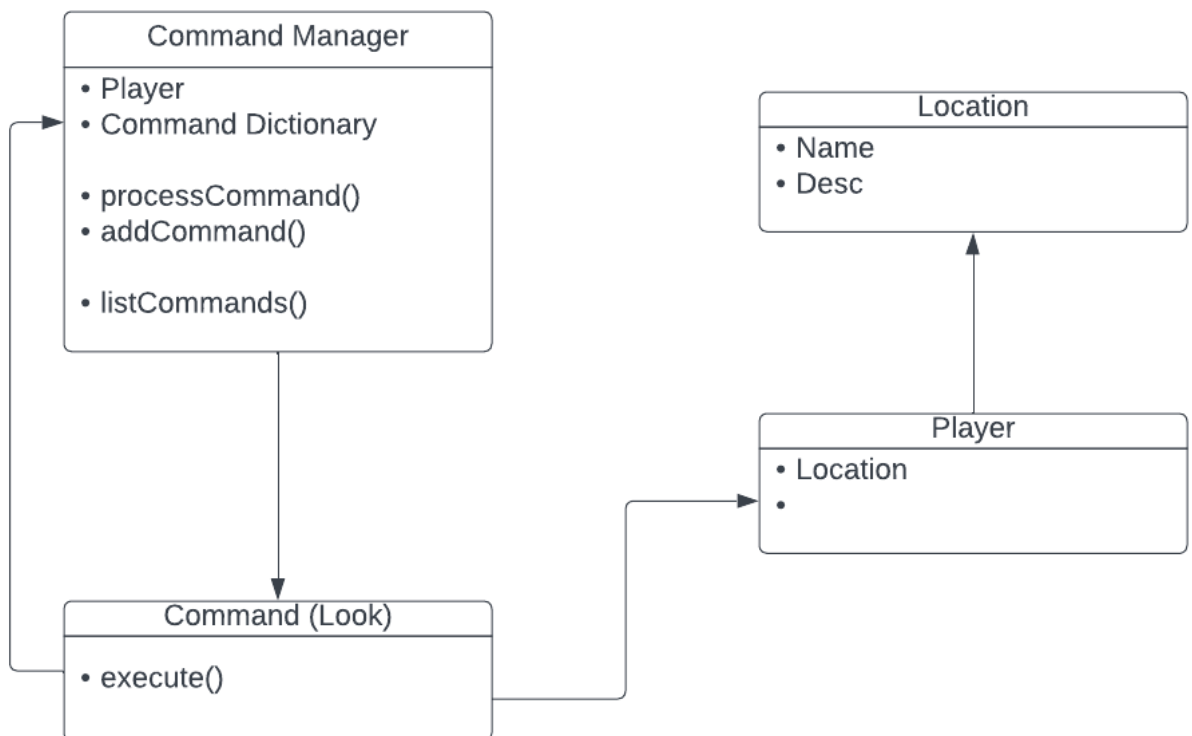
- Visual Studio 2022
- SourceTree
- GitHub
- Lecture 3.2 – Data Structures

Tasks undertaken:

- Design the Command Pattern
- Implement Command Manager
- Implement the Commands
- Commit to Git

What we found out:

1. Design the Command Pattern:



The above UML diagram shows roughly, the linkages between the Command Manager, Command subclasses (in this case, the Look command), and entities effected by the command.

As shown, the command manager has a list of commands and a way to look up those commands, in processCommand. The command itself has a reference to the player and a vector of strings. Said strings are then used to figure out which entities should be altered, then the objects are altered.

2. Implementing the Command Manager:

The command manager basically needs to encompass the commands and be able to invoke them upon receiving a command.

```
CommandManager::CommandManager(Player* player) : _player(player)
{
    _commands.emplace("go", new GoCommand());
    _commands.emplace("look", new LookCommand());
    _commands.emplace("help", new HelpCommand(this));
    _commands.emplace("inventory", new InventoryCommand());
    _commands.emplace("alias", new AliasCommand(this));
    _commands.emplace("debug", new DebugCommand());
    // _commands.emplace("quit", new QuitCommand());
}
```

Within the constructor, a set list of commands is created. Some commands require the addition of a reference to the command manager, thus they have been given a pointer.

```
void CommandManager::processCommand(vector<string> command)
{
    if (_commands.find(command.at(0)) != _commands.end())
    {
        _commands.at(command.at(0))->execute(command, _player);
    }
    else
    {
        cout << "Invalid command." << endl;
    }
}
```

Processing commands requires the inputting of a vector of strings. The first string is checked against the command dictionary, then the command's execute function is called, with the command and the player being fed through. Otherwise, the player is informed that they have not provided an incorrect command.

The manager is stored in the gameworld, and processCommand is called once per update.

```
void GameWorld::update()
{
    _commandManager->processCommand(processInput());
    movePlayer();
}
```

3. Implementing Commands:

Commands are pretty easy, but there are a lot of them. So let's get to explaining these commands, one by one. Valus Ta'aurc. From what I can gather he commands the Siege Dancers from an Imperial Land Tank outside of Rubicon. He's well protected, but with the right team, we can punch through those defences, take this beast out, and break their grip on Freehold. Surely this won't get me into any trouble with plagiarism detection.

The base command class is very simple.

```
class Command
{
protected:
public:
    virtual void execute(vector<string>, Player*) = 0;
};
```

It's a virtual function. Wow!

Look: Gets the player's current location and outputs the description.

```
void LookCommand::execute(vector<string> command, Player* player)
{
    if (command.size() == 1)
    {
        look(player->getLocation());
    }
    else if (command.at(1) == "at" && command.size() == 3)
    {
        LookAt(player->getLocation(), command.at(2));
    }
    else
        cout << "Invalid look command." << endl;
}

void LookCommand::look(Location* location)
{
    cout << location->getDesc() << endl;
}
```

Look At: Gets an item within the player's current location and outputs the description.

```
void LookCommand::lookAt(Location* location, string itemName)
{
    cout << location->getItem(itemName)->getDesc() << endl;
}
```

Help: This is one of the commands that requires the use of the command manager. This command calls a function on the command manager that outputs all the commands available in the list.

```
void HelpCommand::execute(vector<string>, Player*)
{
    _commandManager->showCommands();
}
```

Alias: Takes 2 inputs, then if the command exists, copies the command to a new entry in the list of commands.

```
void AliasCommand::execute(vector<string> command, Player* player)
{
    if (command.size() == 3)
        _commandManager->addCommand(command.at(1), command.at(2));
    else
        cout << "Invalid Alias Command." << endl;
}
```

Debug: Outputs as much information as is available and possible for me to access easily.

```
void DebugCommand::execute(vector<string> command, Player* player)
{
    cout
        << "===DEBUG INFORMATION===" << endl
        << "---PLAYER INFO---" << endl
        << "Player Health: " << player->getHealth() << endl
        << "Items Held: " << endl;
    player->getInventory()->view();
    cout
        << "---LOCATION INFORMATION---" << endl
        << "Location Name: " << player->getLocation()->getName() << endl
        << "Location Desc: " << player->getLocation()->getDesc() << endl
        << "Location Connections: " << endl;
    player->getLocation()->showConnections();
}
```

Go: Checks if the direction has been entered, if it has, then it checks if there's anything in that direction. Once checked, it moves the player to that location via a function previously used in other tasks.


```
void GoCommand::execute(vector<string> command, Player* player)
{
    if (command.size() == 1)
    {
        cout << "No direction specified. Please try again and provide a direction." << endl;
    }
    else if (command.size() == 2)
    {
        Location* pLoc = player->getLocation();
        const string pCon = pLoc->findConnection(command.at(1));

        if (pCon == "")
        {
            cout << "Invalid direction. Please choose another direction." << endl;
        }
        else
        {
            player->setLocName(pCon);
        }
    }
    else
    {
        cout << "Unable to process command. Please input command and single direction." << endl;
    }
}
```

Inventory: Get's the player's inventory and outputs it to console.

```
void InventoryCommand::execute(vector<string> command, Player* player)
{
    player->getInventory()->view();
}
```

4. Commit to Git:



- Complete code for T12, Started work on T15
- T12 progress
- Finished code for T11, started T12