Spike: Task 14
Title: Command Pattern
Author: Sam Huffer, 101633177

Goals / deliverables:

- The game should be able to load adventures from text files, with locations and some game entities.
- Commands that will look at (but not move or change) entities. Commands will include:
  - HELP (list of commands and their syntax details)
  - INVENTORY (what the player has)
  - LOOK, LOOK AT (but not LOOK IN yet)
  - ALIAS (to remap commands)
  - DEBUG TREE (of the game graph world)
- A UML of your finished command pattern-related classes, included in your spike report.

Technologies, Tools, and Resources used:

- Visual Studio 2019
- Microsoft Word

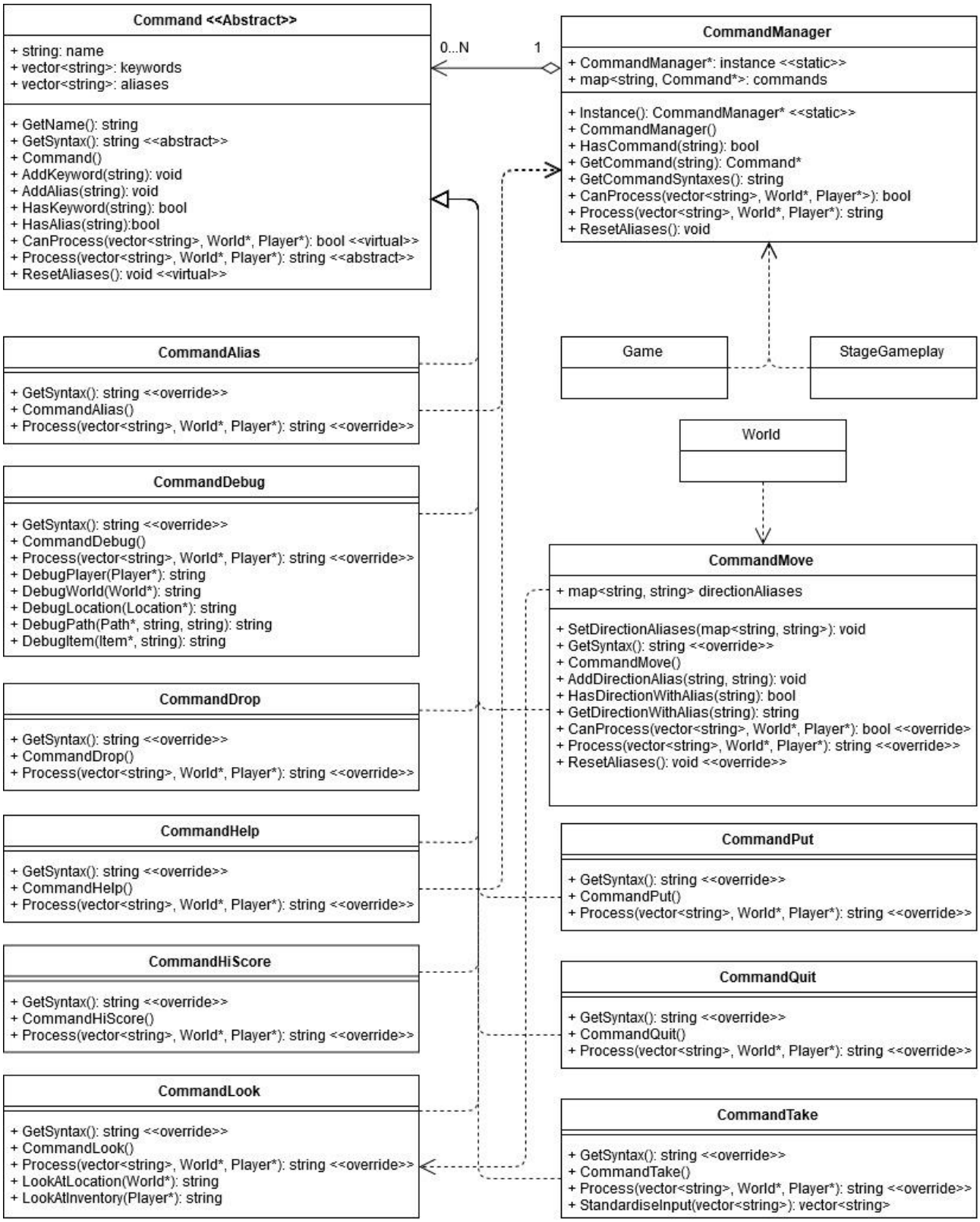
Tasks undertaken:

- I copied the "Zorkish Adventure" project and the task 12 spike report into the task folder, stripping out the spike report's original content and replacing it with goals and resources pertaining to the task at hand.
- I had a look at the python Zorkish Adventure demo code for an idea of how to do the command pattern more formally than my own prior implementation of the look, move, take, put, and drop commands. I noticed that all commands were named "CommandSomething", which I thought was a good idea and realised I hadn't done that with the stage classes. So I went back and renamed the stage classes and files to be "StageSomething", and fixed up all their #includes statements to reference the new file names.
- I started transitioning StageGameplay.Move to a proper command pattern implementation in its own class inheriting from the Command class and being stored in the CommandManager class, but ran into errors with the CommandMove class showing a base class undefined error. I moved all #includes into "pch.h", for both files unique to this project and classes included from the standard library, and double checked that all header files included "pch.h" instead of including required header files manually.
- I moved the handling of direction aliases into the CommandMove class, such that it now holds the map of aliases to directions, which gets filled from world's constructor, and its CanProcess() and Process() methods check for aliased directions. I then commented out StageGameplay's movement-related code, and added a call to CommandMove via the CommandManager to StageGameplay.Update() to replace the removed movement code checks.
- I transitioned StageGameplay.Look to a new CommandLook class, adding it to the CommandManager. I then added to the CommandManager the methods CanProcess() and Process(), which run the passed inputs through each of its commands to see if anything can process the input. Having the CommandManager ask each Command directly rather than having StageGameplay call CommandManager.GetCommand() for each command, or adding a GetCommands() method, seemed more efficient and programmatically safer. I edited the calls to CommandMove's CanProcess() and Process() methods from StageGameplay via the CommandManager to instead call CommandManager's, and check that Process()'s returned string wasn't an error message before returning the output to the Game class.
- I remembered that I need to be able to reset information between different worlds, including the map of direction aliases, so I reorganised CommandMove and World's constructor such that World

builds a map of aliases to directions, and then sets CommandMove's directionAliases field to be that map, rather than adding the aliases one at a time.

- I noticed I had a number of signed unsigned mismatch warnings, so I went through each and cast each instance of a size_t causing the warning as an int.
- I removed the calls to StageGameplay.Drop() and StageGameplay.PutIn() and adapted the methods into their own Command classes – CommandDrop and CommandPut – and added them to the CommandManager.
- I removed the calls to StageGameplay.Take() and adapted the method into its own class, folding in its extra checks in StageGameplay.Update() into CommandTake.StandardiseInput() so that it would still be able to process "pick up" the same as "take".
- I went back to CommandLook and added to it its own StandardiseInput() method to convert "inventory" into "look at inventory", and removed the custom inventory check from CommandLook.Process().
- I took the "hiscore" and "quit" checks in StageGameplay.Update() and adapted them into their own Command classes.
- I added to Command an abstract GetSyntax() method, and took each command's syntax from StageHelp and added them as the result of their respective GetSyntax() methods, as well as the command's name and description. To each, I also added a check for any aliases for the command's main keyword, listing them with the rest of the syntax if there were any. For CommandMove, I also had it perform a similar check for direction aliases. I then added to CommandManager GetSyntaxes() to compile all syntaxes together for printing by StageHelp, as well as the new CommandHelp.
- I added to Command ResetAliases() to overwrite a command's aliases vector with a blank vector (CommandMove's overwrites directionAliases as well), gave CommandManager a method to trigger ResetAliases() for all commands, and added to Game.SetStage() a check for if the current stage is StageGameplay, calling CommandManager.ResetAliases() if it was StageGameplay, so that StageHelp wouldn't erroneously display command and direction aliases loaded from a world the player was no longer playing inside of.
- I modified CommandManager.GetCommand() to search for commands by keyword or alias rather than searching for them by their string index, and added HasCommand() to check if a command exists. I then added CommandAlias, configuring it to be able to add single-word aliases to commands provided another command doesn't already have it as a keyword or alias.
- Last, I looked at the python demo, specifically at the debug command to get an idea of what it did, and implemented a C++ version, CommandDebug, that would print all of the details of the current world, location (including paths), player, and any items in their location or inventory, with each variable labelled appropriately. For this, I added a few public properties to the classes having their details printed so that CommandDebug could access them properly. I didn't go as far as to print the details of all locations, however that doesn't seem like it would be too difficult to add, nor does allowing specifying a particular thing to debug.

UML Diagram


Command <<Abstract>>
+ string: name
+ vector<string>: keywords
+ vector<string>: aliases
+ GetName(): string
+ GetSyntax(): string <<abstract>>
+ Command()
+ AddKeyword(string): void
+ AddAlias(string): void
+ HasKeyword(string): bool
+ HasAlias(string):bool
+ CanProcess(vector<string>, World*, Player*): bool <<virtual>>
+ Process(vector<string>, World*, Player*): string <<abstract>>
+ ResetAliases(): void <<virtual>>
0...N
1
CommandManager
+ CommandManager*: instance <<static>>
+ map<string, Command*>: commands
+ Instance(): CommandManager* <<static>>
+ CommandManager()
+ HasCommand(string): bool
+ GetCommand(string): Command*
+ GetCommandSyntaxes(): string
+ CanProcess(vector<string>, World*, Player*>): bool
+ Process(vector<string>, World*, Player*): string
+ ResetAliases(): void
CommandAlias
+ GetSyntax(): string <<override>>
+ CommandAlias()
+ Process(vector<string>, World*, Player*): string <<override>>
Game
StageGameplay
World
CommandDebug
+ GetSyntax(): string <<override>>
+ CommandDebug()
+ Process(vector<string>, World*, Player*): string <<override>>
+ DebugPlayer(Player*): string
+ DebugWorld(World*): string
+ DebugLocation(Location*): string
+ DebugPath(Path*, string, string): string
+ DebugItem(Item*, string): string
CommandMove
+ map<string, string> directionAliases
+ SetDirectionAliases(map<string, string>): void
+ GetSyntax(): string <<override>>
+ CommandMove()
+ AddDirectionAlias(string, string): void
+ HasDirectionWithAlias(string): bool
+ GetDirectionWithAlias(string): string
+ CanProcess(vector<string>, World*, Player*): bool <<override>>
+ Process(vector<string>, World*, Player*): string <<override>>
+ ResetAliases(): void <<override>>
CommandDrop
+ GetSyntax(): string <<override>>
+ CommandDrop()
+ Process(vector<string>, World*, Player*): string <<override>>
CommandHelp
+ GetSyntax(): string <<override>>
+ CommandHelp()
+ Process(vector<string>, World*, Player*): string <<override>>
CommandPut
+ GetSyntax(): string <<override>>
+ CommandPut()
+ Process(vector<string>, World*, Player*): string <<override>>
CommandHiScore
+ GetSyntax(): string <<override>>
+ CommandHiScore()
+ Process(vector<string>, World*, Player*): string <<override>>
CommandQuit
+ GetSyntax(): string <<override>>
+ CommandQuit()
+ Process(vector<string>, World*, Player*): string <<override>>
CommandLook
+ GetSyntax(): string <<override>>
+ CommandLook()
+ Process(vector<string>, World*, Player*): string <<override>>
+ LookAtLocation(World*): string
+ LookAtInventory(Player*): string
CommandTake
+ GetSyntax(): string <<override>>
+ CommandTake()
+ Process(vector<string>, World*, Player*): string <<override>>
+ StandardiseInput(vector<string>): vector<string>

Screenshots

```
std::string StageGameplay::Update(std::string input)
{
    if (!setup)
    {
        return Setup();
    }

    std::vector<std::string> inputStrings = StringManager::Instance()->StringToVector(input, ' ');

    if (CommandManager::Instance()->CanProcess(inputStrings, world, player))
    {
        std::string result = CommandManager::Instance()->Process(inputStrings, world, player);

        if (result != "Error")
        {
            return result + "\n:> ";
        }
        else
        {
            return "I'm sorry, that is not valid input.\n:> ";
        }
    }

    return "I'm sorry, that is not valid input.\n:> ";
}
```

Figure 1: StageGameplay.Update() passing the input to the appropriate command, or saying "Nope, can't do it."

```
bool CommandManager::CanProcess(std::vector<std::string> input, World* world, Player* player)
{
    for (std::pair<std::string, Command*> pair : commands)
    {
        if (pair.second->CanProcess(input, world, player))
        {
            return true;
        }
    }

    return false;
}

std::string CommandManager::Process(std::vector<std::string> input, World* world, Player* player)
{
    for (std::pair<std::string, Command*> pair : commands)
    {
        if (pair.second->CanProcess(input, world, player))
        {
            return pair.second->Process(input, world, player);
        }
    }

    return "Error";
}
```

Figure 2: CommandManager checking with its Commands for who can handle the input, and passing it to the appropriate command.

```
bool Command::HasKeyword(std::string string)
{
    for (std::string keyword : keywords)
    {
        if (string == keyword)
        {
            return true;
        }
    }

    return false;
}

bool Command::HasAlias(std::string string)
{
    for (std::string alias : aliases)
    {
        if (string == alias)
        {
            return true;
        }
    }

    return false;
}

bool Command::CanProcess(std::vector<std::string> input, World* world, Player* player)
{
    return HasKeyword(input[0]) || HasAlias(input[0]);
}
```

Figure 3: The default Command.CanProcess() method. CommandMove's version also checks for directions or direction aliases, in case the player just input those and didn't type "move" at the front.

```
std::string CommandTake::GetSyntax()
{
    std::string result;

    result += "TAKE\n";
    result += "----------------------------\n";
    result += "Function:\n";
    result += "\t- Take an item from your location or a container and put it in your inventory.\n";
    result += "Syntax:\n";
    result += "\t- \"take [item]\"\n";
    result += "\t- \"take [item] from [container]\"\n";
    result += "\t- \"pick up [item]\"\n";
    result += "\t- \"pick up [item] from [container]\"\n";

    if (aliases.size() > 0)
    {
        result += "Aliases for \"take\":\n";

        for (std::string alias : aliases)
        {
            result += "\t- \"" + alias + "\"\n";
        }
    }

    return result;
}
```

Figure 4: CommandTake's GetSyntax() method. Each follows pretty much this pattern, and CommandMove's checks for direction aliases.

```
std::string CommandTake::Process(std::vector<std::string> input, World* world, Player* player)
{
    input = StandardiseInput(input);

    if (input.size() >= 2)
    {
        input.erase(input.begin());

        if (!StringManager::Instance()->VectorContainsString(input, "from"))
        {
            //Find item in location
            if (!world->GetCurrentLocation()->HasItem(input))
            {
                return "You cannot find item '" + StringManager::Instance()->VectorToString(input, ' ') + "' at your current location";
            }
            else
            {
                //Add item to inventory
                Item* item = world->GetCurrentLocation()->GetItem(input);
                world->GetCurrentLocation()->RemoveItem(input);
                player->AddItem(item);
                return "You added " + item->GetName() + " to your inventory.";
            }
        }
        else
        {
            std::vector<std::string> itemName = std::vector<std::string>();
            std::vector<std::string> containerName = std::vector<std::string>();

            //Taking item from container in inventory or location
            //Find cut-off of item and container names
            for (int i = (int)input.size() - 1; i >= 0; i--)
            {
                if (input[i] == "from")
                {
                    //Get item and container names
                    for (int j = 0; j < (int)input.size(); j++)
                    {
                        if (j < i)
                        {
                            itemName.push_back(input[j]);
                        }
                        else if (j > i)
                        {
                            containerName.push_back(input[j]);
                        }
                    }

                    break;
                }
            }

            if (containerName.size() == 0)
            {
                return "What do you want to take '" + StringManager::Instance()->VectorToString(itemName, ' ') + "' from?";
            }
            else if (StringManager::Instance()->VectorToString(containerName, ' ') == "inventory")
            {
                if (player->HasItem(itemName))
                {
                    return "You remove " + StringManager::Instance()->VectorToString(itemName, ' ') + " from your inventory, and add it back to your inventory";
                }
                else
                {
                    return "'" + StringManager::Instance()->VectorToString(itemName, ' ') + "' is not in your inventory. Even if it was, you would merely be put
                }
            }
```

Figure 5: The first part of CommandTake.Process(), as an example of how that method is implemented by each Command. This is one of the pre-existing Commands that were pretty much just copied, pasted, and cleaned up a little bit into its Command class, rather than written from scratch; this one would have been implemented when implementing the inventory system to be able to demonstrate that.

```
        else if (StringManager::Instance()->VectorToString(containerName, ' ') == "location")
        {
            if (world->GetCurrentLocation()->HasItem(itemName))
            {
                //Add item to inventory
                Item* item = world->GetCurrentLocation()->GetItem(itemName);
                world->GetCurrentLocation()->RemoveItem(itemName);
                player->AddItem(item);
                return "You added " + item->GetName() + " to your inventory.";
            }
            else
            {
                return "'" + StringManager::Instance()->VectorToString(itemName, ' ') + "' is not at your current location.";
            }
        }
        else
        {
            //Find container in inventory or location
            Item* containerItemAsItem = nullptr;

            if (world->GetCurrentLocation()->HasItem(containerName))
            {
                containerItemAsItem = world->GetCurrentLocation()->GetItem(containerName);
            }
            else if (player->HasItem(containerName))
            {
                containerItemAsItem = player->GetItem(containerName);
            }

            if (containerItemAsItem == nullptr)
            {
                return "You can't find '" + StringManager::Instance()->VectorToString(containerName, ' ') + "' at your current location or in your inventory.";
            }
            else if (!containerItemAsItem->GetIsContainer())
            {
                return "You cannot take '" + StringManager::Instance()->VectorToString(itemName, ' ') + "' from '" + StringManager::Instance()->VectorToString(co
                    StringManager::Instance()->VectorToString(containerName, ' ') + "' is not a container.";
            }
            else
            {
                //Find item in container
                ContainerItem* containerItem = (ContainerItem*)containerItemAsItem;

                if (!containerItem->HasItem(itemName))
                {
                    return "You cannot find '" + StringManager::Instance()->VectorToString(itemName, ' ') + "' inside '" + StringManager::Instance()->VectorToStr
                }
                else
                {
                    //Take item from container
                    Item* item = containerItem->GetItem(itemName);
                    containerItem->RemoveItem(itemName);
                    player->AddItem(item);
                    return "You added " + item->GetName() + " to your inventory";
                }
            }
        }
    }

    return "You can't take stuff like that.";
}
```

Figure 6: Part 2 of CommandTake.Process()

```
std::string CommandDrop::Process(std::vector<std::string> input, World* world, Player* player)
{
    input.erase(input.begin());

    if (!player->HasItem(input))
    {
        return "Item '" + StringManager::Instance()->VectorToString(input, ' ') + "' is not in your inventory";
    }

    Item* item = player->GetItem(input);
    player->RemoveItem(input);
    world->GetCurrentLocation()->AddItem(item);
    return "You dropped " + item->GetName() + ".";
}
```

Figure 7: CommandDrop.Process(), a much shorter example of Command.Process() being implemented.

What we found out:

- Not a bad idea with a state pattern or command pattern to name state or command classes “CommandSomething” rather than just “Something”, for ease of recognition and organising.
- It’s a good idea to have the CommandManager check with each Command if it can process input, and then pass said input to the Command via the CommandManager. It’s pretty extensible, only requiring that the CommandManager have each new Command in its list of Commands rather than requiring new if statement clauses for each new Command; and it’s pretty safe, as only CommandManager is checking with each Command, rather than having other classes fetch each or all Commands from CommandManager and then calling CanProcess() and Process() directly.
- The CommandManager works nicely with the singleton pattern; derived commands arguably could, but that wouldn’t work as cleanly if they in turn had further derived commands.