Command Pattern

Name: Shiva Gupta
Student ID: 102262514
Unit Code: COS30031
Task Number: 12
Date: 05/11/2021
Repo link: https://bitbucket.org/ShivaGupta_/cos30031-102262514/src/master/12%20-%20Spike%20-%20Command%20Pattern/

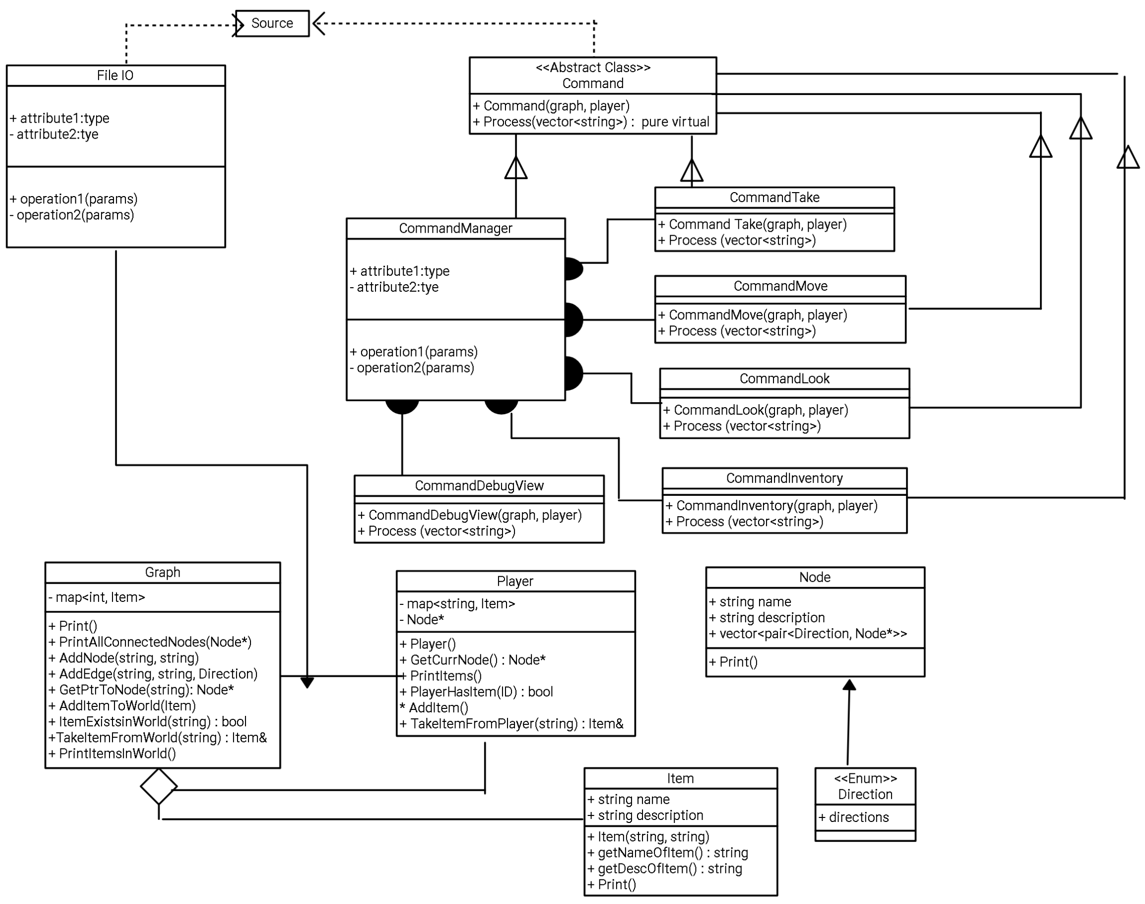
Goals/Deliverables

[SPIKE REPORT] + [CODE] + [DESIGN DOCUMENT]

Knowledge Gap

Understanding the command pattern, implementing a command parser for Zorkish game using that pattern. The parser should be robust and should include ways to specify, manage and extend commands.

UML Design


Source
File IO
+ attribute1:type
- attribute2:tye
+ operation1(params)
- operation2(params)
<<Abstract Class>> Command
+ Command(graph, player)
+ Process(vector<string>) : pure virtual
CommandTake
+ Command Take(graph, player)
+ Process (vector<string>)
CommandManager
+ attribute1:type
- attribute2:tye
+ operation1(params)
- operation2(params)
CommandMove
+ CommandMove(graph, player)
+ Process (vector<string>)
CommandLook
+ CommandLook(graph, player)
+ Process (vector<string>)
CommandDebugView
+ CommandDebugView(graph, player)
+ Process (vector<string>)
CommandInventory
+ CommandInventory(graph, player)
+ Process (vector<string>)
Graph
- map<int, Item>
+ Print()
+ PrintAllConnectedNodes(Node*)
+ AddNode(string, string)
+ AddEdge(string, string, Direction)
+ GetPtrToNode(string): Node*
+ AddItemToWorld(Item)
+ ItemExistsinWorld(string) : bool
+TakeItemFromWorld(string) : Item&
+ PrintItemsInWorld()
Player
- map<string, Item>
- Node*
+ Player()
+ GetCurrNode() : Node*
+ PrintItems()
+ PlayerHasItem(ID) : bool
* AddItem()
+ TakeItemFromPlayer(string) : Item&
Node
+ string name
+ string description
+ vector<pair<Direction, Node*>>
+ Print()
Item
+ string name
+ string description
+ Item(string, string)
+ getNameOfItem() : string
+ getDescOfItem() : string
+ Print()
<<Enum>> Direction
+ directions

Tasks Undertaken

The tasks undertaken for this task were to implement the logic for parsing different commands and handling user input with appropriate case scenarios.

Loading Items From Adventure File

1. To load the items from adventure file we can use the File IO class from the previous task. The format of the adventure file is as follows.

```
<Number of Items>
<Item Name>
<Item Description>
<Item Name>
<Item Description>
    ...
```

2. We can loop through the file and add item to the world.

Add item To World

3. We can use the insert function for map container.

```
    _worldItems.insert(pair<string, Item>(itm.getNameOfItem(), itm));
```

Creating Items

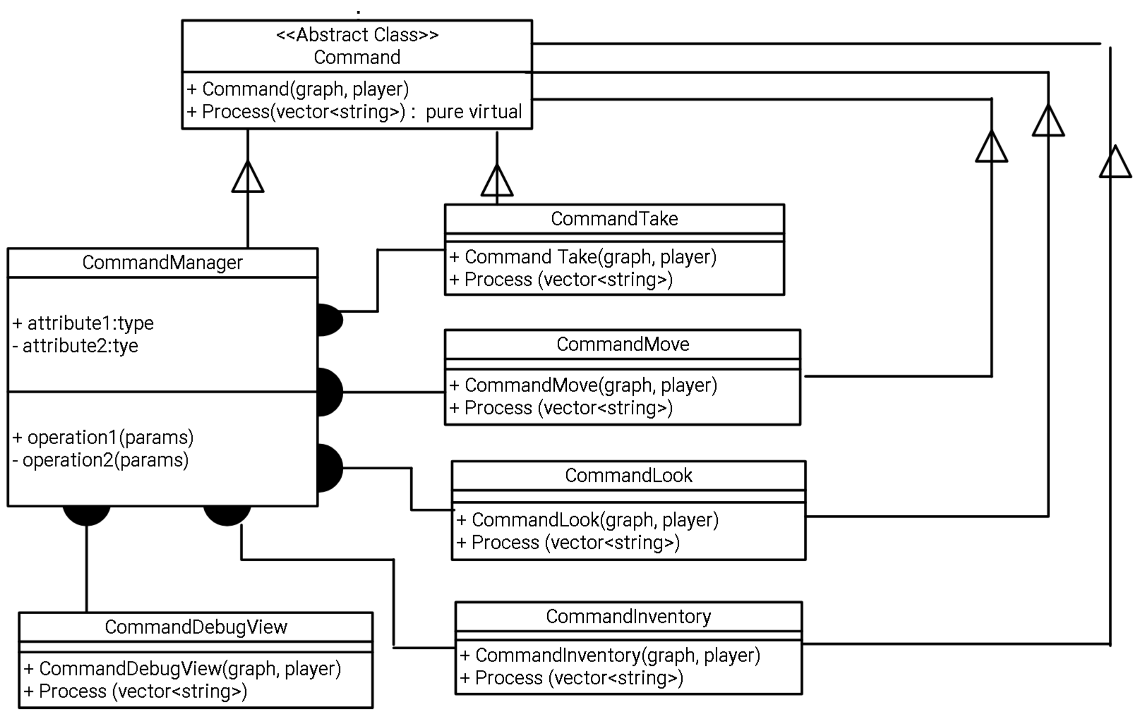
Items are entities that can be found in the world. The player can interact with them using commands.

4. For now, the item will have a name and description with some helper methods to retrieve details for an item.

```
class Item
{
private:
    string _name;
    string _desc;
public:
    Item(const string& name, const string& desc) : _name(name), _desc(desc)
{}

    const string& getNameOfItem()
    {
        return _name;
    }

    const string& getDescOfItem()
    {
        return _desc;
    }

    void Print()
    {
        cout << "Item name: " << _name << endl;
        cout << "Item Description: " << _desc << endl;
    }
}
```

Command Parser

5. To implement the a command processor, we will use the command pattern. The main features of this pattern is that the command is responsible for executing itself. To execute itself, it would need the object on which the command needs to be executed.
6. We will make a `Command` abstract class so that we can compose different commands in any container.
7. For this task, the `CommandManager` approach was implemented which at it's core is a map with command names and pointers to command instances. Since, the manager has pointers, it can clean up the command instances.



<<Abstract Class>>
Command
+ Command(graph, player)
+ Process(vector<string>) : pure virtual
CommandManager
+ attribute1:type
- attribute2:tye
+ operation1(params)
- operation2(params)
CommandTake
+ Command Take(graph, player)
+ Process (vector<string>)
CommandMove
+ CommandMove(graph, player)
+ Process (vector<string>)
CommandLook
+ CommandLook(graph, player)
+ Process (vector<string>)
CommandInventory
+ CommandInventory(graph, player)
+ Process (vector<string>)
CommandDebugView
+ CommandDebugView(graph, player)
+ Process (vector<string>)

8. Make a Command abstract class.

```
class Command
{
protected:
    Graph& _graph;
    Player& _player;
public:
    Command(Graph& graph, Player& player) : _graph(graph), _player(player)
{}
    virtual void Process(vector<string>& args) = 0;
};
```

9. Now, make the Command manager which will be instantiated in main. The manager itself is inherited from the `Command` class so that the command it points to can be called with the manager's process methods which provides a nice wrap up.

```
private:
    Player& _p;
    Graph& _g;
    map<string, Command*> _cmds;
public:
    CommandManager(Graph& g, Player& p) : _g(g), _p(p)
    {
        //Move
        _cmds.insert(pair<string, Command*>("move", new CommandMove(_g,
_p)));
        _cmds["go"] = _cmds["move"];
                ....
    }

    void Process(vector<string>& args)
    {
        map<string, Command*>::iterator it;
        it = _cmds.find(args[0]);
        //if no command is entered
        if (args.size() == 0)
            cout << "Enter a command to interact with surroundings." <<
endl;
        //if only one word is entered in the command
        else if (args.size() == 1)
        {
            if (it != _cmds.end())
            {
                (it->second)->Process(args);
            }
            else if (args[0] == "help")
            {
                PrintAllCommands();
            }
            //there is no command as the user entered
            //must be a direction
            else
            {
                args[0] = "move";
                (it->second)->Process(args);
            }
                    ....
        }
```

10. Once we have the abstract class and the manager, we just need to implement the logic for different commands.

Command Move Logic

11. The idea is to set the current position pointer to the passed in node provided location passed in is correct.

```
        if (args.size() != 2)
            cout << "Can't move like that! Syntax: \" move (direction)\" "
<< endl;
        else
            for (auto n : _player.GetCurrNode()->nodes)
                //if the direction entered by user is same as that of
connected node
                if (args[1] == Node::DirToString(n.first))
                    _player.SetCurrNode(n.second);
                    cout << _player.GetCurrNode()->name << endl;
                    _graph.PrintAllConnectedNodes(_player.GetCurrNode());
                    break;
                else
                    cout << "Can't move " << args[1] << endl;
                    break;
```

Command Look Logic

12. The logic behind the look command is to print the details of the current location and the connected edges.

```
        if (args.size() == 1)
            cout << "Current Node: " << _player.GetCurrNode()->name << endl;
            _graph.PrintAllConnectedNodes(_player.GetCurrNode());
```

13. If the command is `look at` then we can print the details of the items present in the world.

```
        else if (args.size() == 2)
        {
            _graph.PrintItemsInWorld();
        }
```

Command Inventory Logic

14. The `inventory` command is similar to the debug command because we just need to print the details of the items that the player has in their inventory.

```
        cout << "You currently have: " << endl;

        _player.PrintItems();
```

Command Debug View Logic

15. This command is not as much for the player as it is for the developer. We print all the information currently available to us. This can also be thought of as the combination of logic of the commands mentioned above.

```
        if (args.size() == 1)
        {
            cout << "Current Node: " << _player.GetCurrNode()->name << endl;
            _graph.PrintAllConnectedNodes(_player.GetCurrNode());
            _graph.PrintItemsInWorld();
            _player.PrintItems();
        }
```

Command Alias and Help Logic

16. The main purpose of the alias command is to map the commands to passed in words. In the command manager we are mapping the commands with keywords. So, it is best to abstract the logic to the Command Manager class otherwise we have to take extra steps to find the items and refer to them by name.
17. We check if the command already exists. It it does not then we map the command manager's map appropriately.

```
        //alias <new_word> <existing_word>
        else if (args.size() == 3)
        {
            map<string, Command*>::iterator itr;
            map<string, Command*>::iterator iter;
            itr = _cmds.find(args[1]);
            iter = _cmds.find(args[2]);

            if (itr != _cmds.end())
                cout << "Alias Failed. New command word " + args[1] +
"already exists." << endl;
            else if (iter == _cmds.end())
                cout << "Alias Failed. Command Word " + args[2] + "clready
exists" << endl;
            //Create the alias
            else
            {
                _cmds[args[1]] = _cmds[args[2]];
                cout << "Alias created. Command " + args[2] + " can now be
access via " + args[1] <<                    endl;
            }
        }
```

18. For the **HELP** command we can simply iterate through the command manager's map and print all the keywords.

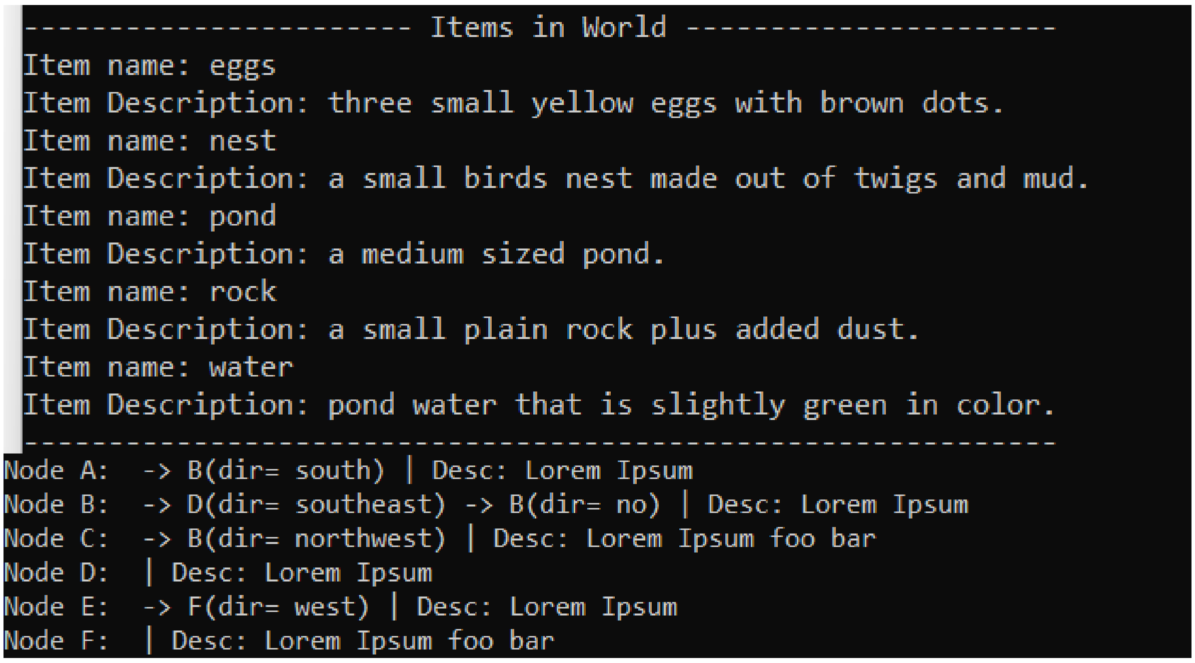
```
    void PrintAllCommands()
    {
        map<string, Command*>::iterator it;
        cout << "---------------------- Available Commands -------------
----------------" << endl;
        for (it = _cmds.begin(); it != _cmds.end(); it++)
            cout << it->first << endl;
```

What we found out

After the completion of this task, we have successfully developed a command parser for the Zorkish game. The command parser is based on the command pattern and implements features like extensive error handling and such. The main idea to take away is to that the commands are passed in the objects that need to be worked on and the command is responsible for the execution. It produces low-coupled code because the commands can act as black boxes which know how to execute.

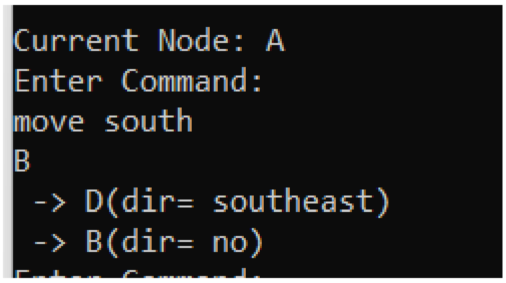
OUTPUT

Items and Locations



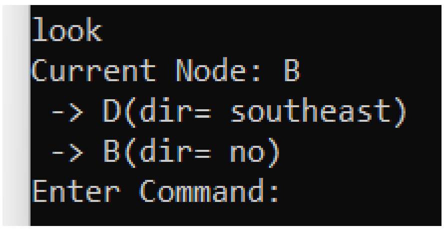
```
----------------------- Items in World ----------------------
Item name: eggs
Item Description: three small yellow eggs with brown dots.
Item name: nest
Item Description: a small birds nest made out of twigs and mud.
Item name: pond
Item Description: a medium sized pond.
Item name: rock
Item Description: a small plain rock plus added dust.
Item name: water
Item Description: pond water that is slightly green in color.
-------------------------------------------------------------
Node A:  -> B(dir= south) | Desc: Lorem Ipsum
Node B:  -> D(dir= southeast) -> B(dir= no) | Desc: Lorem Ipsum
Node C:  -> B(dir= northwest) | Desc: Lorem Ipsum foo bar
Node D:  | Desc: Lorem Ipsum
Node E:  -> F(dir= west) | Desc: Lorem Ipsum
Node F:  | Desc: Lorem Ipsum foo bar
```

Move Command



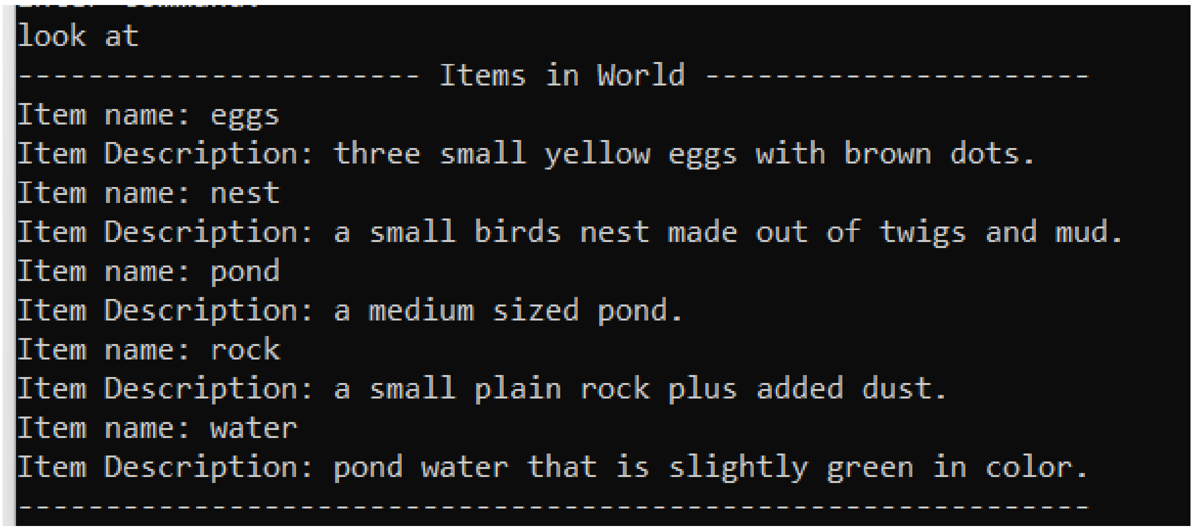
```
Current Node: A
Enter Command:
move south
B
 -> D(dir= southeast)
 -> B(dir= no)
```

Look Command



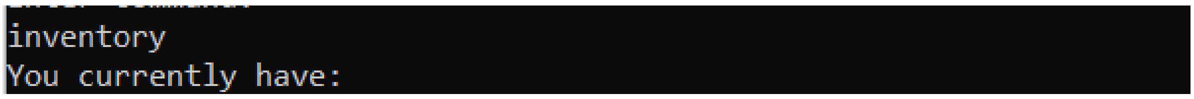
```
look
Current Node: B
 -> D(dir= southeast)
 -> B(dir= no)
Enter Command:
```

Look at command

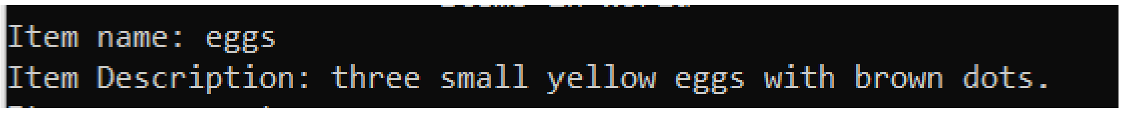


```
look at
----------------------- Items in World ----------------------
Item name: eggs
Item Description: three small yellow eggs with brown dots.
Item name: nest
Item Description: a small birds nest made out of twigs and mud.
Item name: pond
Item Description: a medium sized pond.
Item name: rock
Item Description: a small plain rock plus added dust.
Item name: water
Item Description: pond water that is slightly green in color.
-------------------------------------------------------------
```

Inventory Command

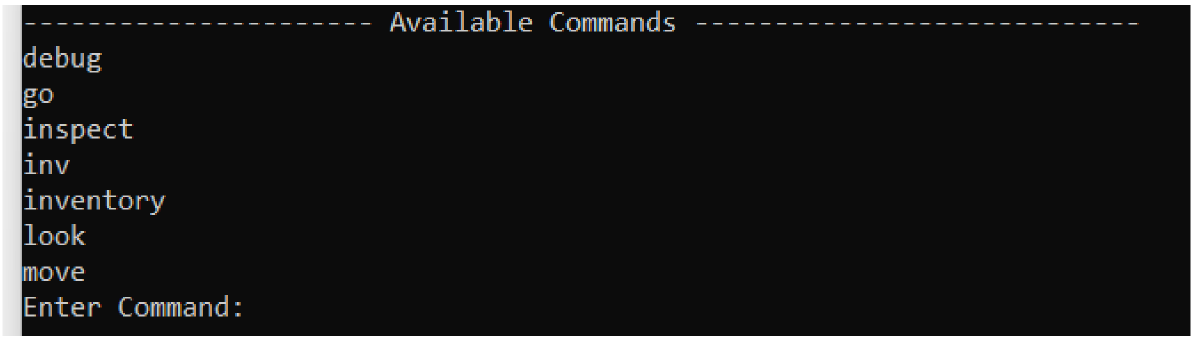


```
inventory
You currently have:
```



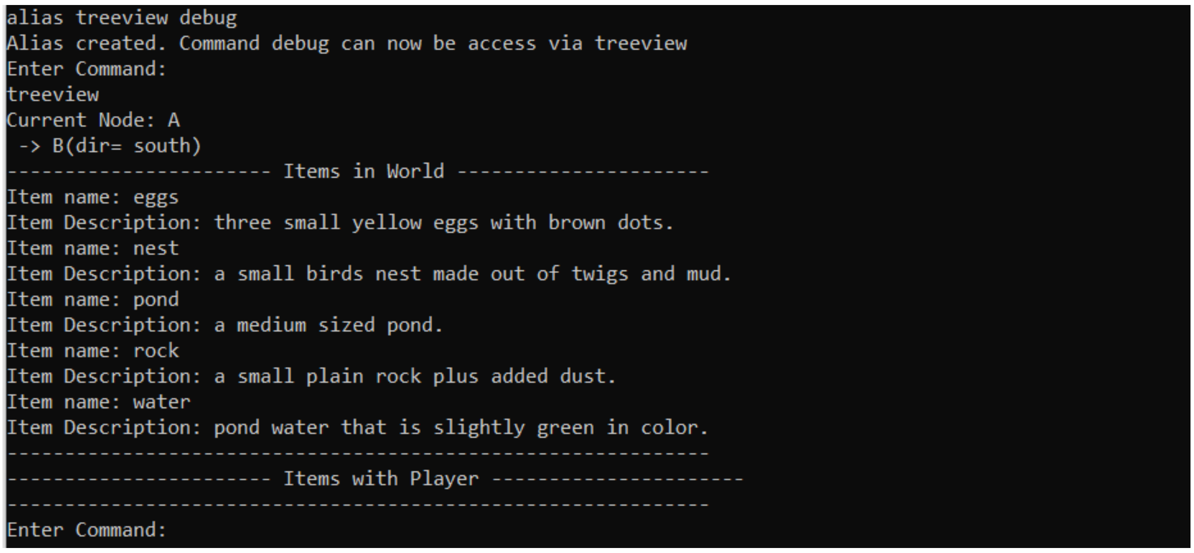
```
Item name: eggs
Item Description: three small yellow eggs with brown dots.
```

Help



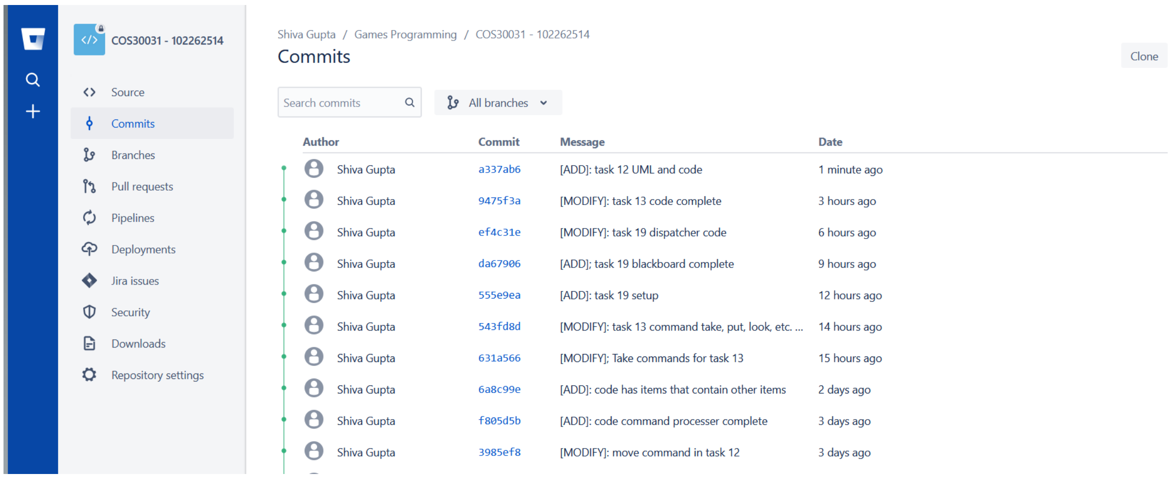
```
--------------------- Available Commands ---------------------------
debug
go
inspect
inv
inventory
look
move
Enter Command:
```

Alias



```
alias treeview debug
Alias created. Command debug can now be access via treeview
Enter Command:
treeview
Current Node: A
 -> B(dir= south)
---------------------- Items in World ---------------------
Item name: eggs
Item Description: three small yellow eggs with brown dots.
Item name: nest
Item Description: a small birds nest made out of twigs and mud.
Item name: pond
Item Description: a medium sized pond.
Item name: rock
Item Description: a small plain rock plus added dust.
Item name: water
Item Description: pond water that is slightly green in color.
-----------------------------------------------------------
---------------------- Items with Player ---------------------
-----------------------------------------------------------
Enter Command:
```

Git Log



Shiva Gupta / Games Programming / COS30031 - 102262514

Commits

Author	Commit	Message	Date
Shiva Gupta	a337ab6	[ADD]: task 12 UML and code	1 minute ago
Shiva Gupta	9475f3a	[MODIFY]: task 13 code complete	3 hours ago
Shiva Gupta	ef4c31e	[MODIFY]: task 19 dispatcher code	6 hours ago
Shiva Gupta	da67906	[ADD]; task 19 blackboard complete	9 hours ago
Shiva Gupta	555e9ea	[ADD]: task 19 setup	12 hours ago
Shiva Gupta	543fd8d	[MODIFY]: task 13 command take, put, look, etc. ...	14 hours ago
Shiva Gupta	631a566	[MODIFY]; Take commands for task 13	15 hours ago
Shiva Gupta	6a8c99e	[ADD]: code has items that contain other items	2 days ago
Shiva Gupta	f805d5b	[ADD]: code command processer complete	3 days ago
Shiva Gupta	3985ef8	[MODIFY]: move command in task 12	3 days ago