Final Documentation for 32 Pounds

CS 383: Software Engineering

32 Pounds

May 13, 2015

Contents

1 Project History **2**
- 1.0.1 Player Expectations . . . 3
- 1.0.2 Advanced Gameplay . . . 3
- 1.0.3 Multiplayer . . . 3

2 Implementation details **4**

3 Initial Project Design: Use Cases **5**
- 3.1 Player Use Cases . . . 5
  - 3.1.1 Speak with NPC . . . 5
  - 3.1.2 Picking Up/Placing Items in Inventory . . . 5
  - 3.1.3 Movement through Area . . . 6
  - 3.1.4 Inventory Item Use . . . 7
  - 3.1.5 Interact with a Coworker . . . 7
  - 3.1.6 World Entity Interaction . . . 8
- 3.2 Combat . . . 8
  - 3.2.1 Engage Combat . . . 8
  - 3.2.2 Combat Turn . . . 9
  - 3.2.3 Player Wins Battle . . . 10

4 Current Project Use Cases **11**
- 4.1 Map . . . 11
  - 4.1.1 Update Map . . . 11
  - 4.1.2 Draw Map . . . 11
- 4.2 Game State . . . 12
  - 4.2.1 Initialize Game . . . 12
  - 4.2.2 Add New Player . . . 13
  - 4.2.3 Add New Monster . . . 13
  - 4.2.4 Remove Player . . . 14
  - 4.2.5 Remove Monster . . . 14
- 4.3 Player . . . 14
  - 4.3.1 Player Movement . . . 14
  - 4.3.2 Level Transfer . . . 15
  - 4.3.3 Combat . . . 15

5 Initial Project Design: Class Diagrams **16**
5.1 Combat . . . 17
5.2 Bureaucracy - Positive and Negative Attention . . . 18
5.3 Map and Movement . . . 20
5.4 Player . . . 21

6 Current Project Design: Diagrams **22**
6.1 Game State Classes . . . 22
6.2 Input Classes . . . 23
6.3 Command Classes . . . 23
6.4 Networking Diagrams . . . 24
6.4.1 Class Diagram . . . 24
6.4.2 Sequence Diagrams . . . 25

7 Metrics **26**

8 jUnit Testing **27**
8.1 Comms . . . 27
8.1.1 GameID Test . . . 27
8.1.2 Parser Tester . . . 28
8.1.3 Command Memento . . . 31
8.2 Renderer Tests . . . 31
8.2.1 Sprite Storage Tests . . . 31

Chapter 1

Project History

Initially our project design was based primarily on working to fulfill homework requirements and trying to keep everything scrum based. For our group most of scrum work was new, really implementing project design was something most of us haven't done. Our typical work load was code, code, and more code. At the beginning this seemed to be a setback for our team, we had trouble focusing on setting up requirements, use cases, and class diagrams instead of just jumping straight into code. The workload wasn't necessary unbearable or hard, it was just a different process than most of us were used to. As a team we thought that it would have been easier to code and then set up these diagrams based on our code, but when setting up a business idea you must have an actual plan before you have the code, so this was essential.

Lets move to project design. We had high hopes for our game which was acceptable but later in the semester was not very feasible. This was ok because this class isn't necessarily based on the project outcome but really learning to work by the scrum techniques and to incorporate software design principles. We were about half way through the semester with very little coding done, design was in place but we were waiting on implementing anything due to the lack of knowledge of what the next homework was going to be. Basically half way through the semester we had nothing but design and a main menu. Previous students who took CS383 warned us of choosing a game because it seemed to be more difficult than expected. Most likely because expectations are high when it comes to games and experience in making them is somewhat low. For our group our experience varied, we have a couple very good programmers and others that aren't as efficient. Java wasn't necessarily a first language for most of us so having to learn java and work on a fairly difficult project was somewhat tough. As a team we diverted, some worked primarily on code while others worked on map, sprite art, documentation, and other necessary tasks.

Lack of time and experience seemed to hinder us from the initial game design we had developed. This was expected, we had to adapt and pick what we deemed to be most important from our backlog. This is where we chose to diverge from a very developed storyline and work more into networking. Networking was something our team wasn't very confident with so we decided it would be beneficial for us to work on that rather than playing with player and quests and such.

We decided to keep our initial documentation and also to make current documentation to see how things varied throughout the semester. Here are the designs that changed the

most throughout the semester.

1. Player design
2. Advanced gameplay i.e quests, factions, inventory
3. Multiplayer

1.0.1 Player Expectations

Initially we had a couple of different ideas drawn out for our main player character. We wanted a player inventory, player skills, and a faction or group that the player could join. We decided to that this would mean that we would need to incorporate a very defined story line that was single player heavy, which we did not want to do. The ideas for the player were scrapped do to the fact that we would rather create a better multiplayer environment than a single player environment.

1.0.2 Advanced Gameplay

Gameplay is something where the idea seems so easy but the implementation takes much more thought. So much needed to be implemented before we started gameplay that we never really got around to it. For instance, our game was structured for single player until we decided to reroute to multiplayer. The gameplay was different for a single player than it is for multiplayer in our game. Before we can completely work on the gameplay we wanted i.e quests, combat, factions, we needed inventory, player health, moving monsters, and also monsters that react to your movement which is more difficult than one would think. We ended up with making a gameplay in which you run around killing bugs that have invaded your office space, some bugs attack you, others dont.

1.0.3 Multiplayer

Much of the final sprint was dedicated to Multiplayer. It became a project fork around homework 6. The decision to focus on multiplayer was motivated by three key reasons: We had spent considerable effort keeping the code base compatible with a multiplayer mode, Many of us were interested in the experience of making a multiplayer game, and we though that we might be the only team to successfully acheive full multiplayer. We set up a meeting where the details of multiplayer were hashed out, like the structure of our packets. We chose to send the complete state of the game through a UDP packet at 40 packets per second so that the same packet would apply to all clients, and we don't have to remember which entities have moved.

Chapter 2

Implementation details

Our game can be run with “gradle run” from the “OSGame” directory within our project, or by running the jar file in the top level of our github repository. Both the client and the server are run from the same code, in fact every instance of the game is running a server and a client locally. The game has been tested to work freely within and between Windows, OS X, and Ubuntu and Arch Linux, although there are known issues when running in openJDK 7.

To run our game, simply launch the jar file. Network communications use UDP ports 5050 and 5051. To join a game, press ESC, then “join game”, and then enter the IP address of a computer running another instance of the game. The other instance only needs to be running, it is not necessary for them to press ”Host Game” or to even be playing in their local game at the time.

Gradle has been used to manage almost every aspect of the build process. Many team members have used its good, if limited, support for netbeans and eclipse to build from within their IDE.

The code is organized first between the “core” and the “desktop” folder. “core” contains the platform independent code, which is the vast majority of our code base. “desktop” contains a main class that deals with the specific considerations of launching our game on a desktop computing environment. If we wished to expand to other platforms, only the code within the desktop folder would need rewritten within a new folder.

Inside of “core/src” exists most of our game’s source code, split between the “com” with the application code and “test” with our unit tests. Within “com” and “test” are corresponding sets of .java files in a number of different subsystem directories.

Chapter 3

Initial Project Design: Use Cases

This chapter presents the initial use cases based off the major systems in the game. These categories include the player, monster, communications, game state and map.

3.1 Player Use Cases

3.1.1 Speak with NPC

Goal: Player wishes to speak with an NPC in order to gain information or recieve a quest.
Actors: Player, NPC
Preconditions:

1. The player is adjacent to an NPC.
2. The NPC has an available speak interact command

Summary: The player communicates with an NPC that is occupying the nearby space in the world. Communication will add new game information and/or start a new quest.
Related Use Cases: This use case extends the Interact with Coworker use case, and will be connected to use cases coorelating to interacting with NPCs through dialog.
Steps:

1. Player selects NPC to interact with
2. Player selects the speak function from the interact menu
3. Information is written to the screen
4. If necessary, input will be selected from options on the screen to allow player to respond to NPC.

3.1.2 Picking Up/Placing Items in Inventory

Goal: Player wishes to move item from world space into their inventory.
Actors: Player
Preconditions

1. Player occupies space with object

2. Player has an empty slot in their inventory

Summary: The player comes into contact with an object within the world space, moves it from the world space to their inventory.

Related Use Cases: This use case extends the World Entity Interaction use case, and the use case upgrading items extends from it.

Steps:

1. Player moves to the same space as a visible object occupying world space.

2. Player selects the place object in inventory selection from interaction menu.

3. Object appears in single slot of players inventory.

4. Object is removed from the world space.

Alternatives: Player does not have an empty slot in their inventory, no action is taken (see *Preconditions*).

3.1.3 Movement through Area

Goal: To cross the current area and enter the next one.

Actors: Player

Preconditions:

1. The tile the player is moving to must not have an obstacle.

2. If there is an item required to move on to the next area, the player must possess it.

Summary: The player will traverse across an area until an exit tile is reached. If he/she encounters a wall or solid object, they player will not move in the direction of the obstacle. When the player stands on an exit tile, the will move to an new area.

Related Use Cases: All overworld interactions are extended by this use case. Talking to npcs, combat, and picking up items are examples.

Steps:

1. The player chooses a direction designated by the movement keys.

2. The input handler checks to see if the move is legal.

3. The player continues to move around the area until an exit tile is reached.

4. The player moves onto the exit tile to take him/her to the next areal.

3.1.4 Inventory Item Use

Actors: Player, Inventory, Useable Item
Preconditions:

1. Inventory belongs to Player.
2. Item is in Inventory.
3. Item is useable from the Inventory
4. Player inputs the use command, selecting Item for use.

Summary: The Player uses the Item in their Inventory. The Item produces its use effect.
Related Use Cases All Useable Item Uses extend this use case. Examples would be quaffing a potion or reading a note.
Steps:

1. Player selects Item from Inventory.
2. System enables the input of the use command.
3. Player inputs the use command.
4. System evaluates the effect of the use command on the Useable Item and updates the Item's state accordingly.

3.1.5 Interact with a Coworker

Goal: The Player wishes to perform an interaction with an NPC.
Actors: Player, NPC
Preconditions:

1. The Player is adjacent to the NPC.

Summary: The Player interacts with an NPC occupying space in the World. The interaction may or may not cause a state change for either the Player or the NPC relationship.
Related Use Cases: All specific NPC interactions extend this use case. Examples include modifying a relationship or managing quests.
Steps:

1. The Player inputs the interact command, selecting the NPC.
2. The System performs the interaction, gathing additional input from the Player as necessary.

3.1.6 World Entity Interaction

Actors: Player, Interactable Entity
Preconditions:

1. Player is adjacent to the Interactable Entity.
2. Player inputs the interact command, selecting the Interactable Entity.

Summary: The Player interacts with the Entity occupying space in the World. The Entity produces its interactive effect.
Related Use Cases All World Entity Interactions extend this use case. Examples would be opening a door or operating a switch.
Steps:

1. Player moves adjacent to an Interactable Entity.
2. System enables the input of the interact command.
3. Player inputs the interact command.
4. System evaluates the effect of the interact command on the Interactable Entity and updates the Entity's state accordingly.
5. Player moves away from the Interactable Entity.
6. System disables the input of the interact command.

3.2 Combat

3.2.1 Engage Combat

Goal: The Player encounters an enemy and starts combat.
Actors: Player(s), Enemy Entity(-ies)
Preconditions:

1. The Player is adjacent to the Enemy Entity.
2. The Player has the Combat flag enabled.
3. The Enemy Entity has the Combat flag enabled.

Summary: The Player engages in combat with an Enemy Entity occupying space in the World. Combat is carried out in Turns, with each combatant selecting a Move each Turn, which are then resolved in order from the entity with the highest Speed value to the entity with the lowest Speed value. When every member of one side has their Health reduced to 0, then the other side is victorious. If the Player(s) won, then go to use case *Player Victory*. If the Player(s) lost, then go to use case *Player Death*.

Related Use Cases: This use case is a parent to *Combat Turn*. The use cases *Player Wins Battle* or *Player Death* immediately follow. This use case can be optionally extended to allow for alternate forms of combat.

Steps:

1. The Player inputs the Begin Combat command, *or* the Enemy Entity begins combat with the Player.
2. The System disables the Player's Combat flag.
3. The use case Cmbat Turn is now performed.

3.2.2 Combat Turn

Goal: The Player seeks to reduce the Enemy Entity's Health to 0 and keep their own Health above 0.

Actors: Player(s), Enemy Entity(-ies)

Preconditions:

1. The Player is currently in the *Engage Combat* use case.
2. The Enemy Entity is currently in the *Engage Combat* use case.

Summary: The Player and the Enemy Entity both select a Move. Moves are then sequentially executed, with the Move selected by the Entity with the largest Speed value being evaluated first, and the Move selected by the Entity with the smallest Speed value being evaluated last.

Related Use Cases: This use case is a child of the *Engage Combat* use case, and can be optionally extended to provide alternative combat mechanics.

Steps:

1. The System prompts the Player with a list of legal Moves.
2. The Player selects a Move.
3. The Enemy entity selects a Move.
4. The System evaluates all Moves, in order of Speed.
5. The System updates the state of each Entity as appropriate.
6. If the Enemy Entity has a Health of 0, go to use case *Player Victory*.
7. If the Player has a Health of 0, go to use case *Player Death*.

Steps:

1. Perform use case *Combat Turn*.
2. If neither side has been reduced to 0 Health, go to step 1.
3. If the Enemy Entity has a Health of 0, go to use case *Player Victory*.
4. If the Player has a Health of 0, go to use case *Player Death*.

3.2.3 Player Wins Battle

Goal: The Player seeks to list and distribute the benefits of victory.

Actors: Player(s), Enemy Entity(-ies)

Preconditions:

1. The Player is currently in the *Combat Turn* use case.
2. The Player has a Health which is greater than 0.
3. The Enemy Entity is currently in the *Combat Turn* use case.
4. The Enemy Entity has a Health of 0.
5. The *Combat* use case is requesting for the *Player Victory* use case to be run.

Summary: The Player is informed of any Items or Reputation gained. If multiple Players are present, then they reach an agreement on how the Items are to be distributed.

Related Use Cases: This use case can only be begun when transitioning from the *Combat Turn* use case.

Steps:

1. The System calculates the Items and/or Reputation gained by the Player based upon the Enemy Entity.
2. The System informs the Player of the Items and/or Reputation gained.
3. If there is only one player, go to step 7.
4. The System allows the Players to distribute the Items gained.
5. The Players either agree or do not agree to the distribution.
6. If the Players do not agree, go to step 4
7. Go to use case *Place Items Inventory*.

Chapter 4

Current Project Use Cases

This chapter presents the current use cases based off the major systems in the game. These categories include the player, monster, communications, game state and map.

4.1 Map

4.1.1 Update Map

Goal: To update the map after player or monster movement.

Actors: Renderer, Game State

Preconditions: In order for the map to be updated, the game window must be open and the original map render must be completed.

Summary: When the player moves, the map updates to show the player on the new position of the map

Related Use Cases: Draw Map.

Steps:

1. Player or monster changes position on the map.
2. Game state checks validity of movement.
3. Game state is updated to reflect new position
4. Renderer is called to redraw the map.
5. Map is redrawn to the screen with updated positions.

Alternative: Player/monster movement is determined invalid in step two, map is not updated.

4.1.2 Draw Map

Goal: Print the map to the game window.

Actors: Renderer

Preconditions: The JSON file containing the tile specifics and the text file with the ASCII map are located in the assets file. Map has been loaded into the array grid.

Summary: The renderer is called and parses through the map text file, prints corresponding PNG images to the game window to create the visual map.

Related Use Cases: Update Map

Steps:

1. For each character in the grid array it selects the position of the grid
2. The character is searched for the corresponding tile texture
3. If multiple textures are used for a certain tile(i.e. a chair), print the bottom texture first.
4. Print the texture to the corresponding space on the screen
5. Repeat steps 2-4 for all characters in the array.

Alternatives: If texture PNG file has not been specified in the switch statement or in the JSON file, the parser will not be able to print it to the screen which can either create a program error or print a black space to the screen.

Find Instance of Char

Goal: Find the first instance of an object on the map. Used for finding the player/other players on the map.

Actors: Parser

Preconditions: The .map file has been loaded into the grid array.

Summary: Parses the array for the first instance of a specified character.

Steps:

1. Move to the first space on the map.
2. Check the character of that space against the one searching for.
3. If the two are the same, return the x,y position,if not move to the next location.
4. If no match is found in map, return the position 0,0.

4.2 Game State

4.2.1 Initialize Game

Goal: To set the system variables for a new instance of the game.

Actors: Player, system

Summary: A new instance of the game is started. The program sets the starting system variables.

Steps:

1. User runs the program, starting new instance of the game.

2. The system creates a new instance of the game state.

3. System runs through initialization sequence, setting all system variables to their starting values.

4. Game is launched.

4.2.2 Add New Player

Goal: Create new player entity with client-server communications.
Actors: Player, System
Preconditions: There is an available GameID to be assigned to the new player entity.
Related Use Cases: Add New Monster
Summary: A new player entity is created, and assigned a GameID for use in the client-server communications.
Steps:

1. New game is started by the player, player instance is created within initialization.

2. A game ID is requested for the new player entity.

3. Player is placed on the map.

4. The player is assigned the new ID.

5. The player is added to the list of drawable items on the map.

6. The drawable list is updated and sorted.

4.2.3 Add New Monster

Goal: To add a new monster instance to the game.
Actors: System
Preconditions: There is an available GameID to be assigned to the new monster entity.
Related Use Cases: Add New Player
Summary: New instance of a monster entity is added to the game.
Steps:

1. System requests for new instance of a monster entity.

2. A game ID is requested for the new monster entity.

3. Monster is placed on the map.

4. The monster is assigned the new ID.

5. The monster is added to the list of drawable items on the map.

6. The drawable list is updated and sorted.

4.2.4 Remove Player

Goal: To remove an instance of a player entity from the game.

Actors: System

Preconditions: The player entity requested to be removed is recognized by the system.

Related Use Cases: Remove Monster

Summary: Player dies in the game, which in turn needs to delete the player instance from the game. Can also be invoked by a player leaving a networked multiplayer game.

Steps:

1. Player exits the game, or dies.
2. System removes the player instance from the list of drawables.
3. System removes the player from the lists of players.

4.2.5 Remove Monster

Goal: To remove an instance of a monster entity from the game.

Actors: System

Preconditions: The monster entity requested to be removed is recognized by the system.

Related Use Cases: Remove Player

Summary: Monster is killed by a player in the game. This creates the need for the specific instance for that monster to be removed from the game.

Steps:

1. Monster is killed by a player in the game.
2. System removes the monster instance from the list of drawables.
3. System removes the monster from the lists of players.

4.3 Player

4.3.1 Player Movement

Goal: Simple movement across the map.

Actors: Player

Preconditions:

1. The tile the player is moving to must be walkable.
2. The tile must not be a portal.

Summary: The player can traverse the level until he/she reaches an obstacle or wall.

Related Use Cases: All overworld interactions are extended by this use case. Combat and Level Transfer are examples.

Steps:

1. The player chooses a direction designated by the movement keys.
2. The input handler checks to see if the move is legal.
3. The player continues to move around the area until an obstacle or wall is reached.

4.3.2 Level Transfer

Goal: Transportation from level 1 to level 2. **Actors**: Player
Preconditions:

1. The tile the player is moving from must be walkable.

Summary: The player has found the portal tile and would like to tranfer to or from each level
Related Use Cases: Player Movement
Steps:

1. The player chooses a direction designated by the movement keys.
2. The input handler checks to see if the move is legal.
3. The player moves to a transport tile.
4. The input handler checks to see if the transport tile is vacant.
5. The player is moved to the next/previous level.

4.3.3 Combat

Goal: The Player encounters an enemy and starts combat.
Actors: Player(s), Enemy Entity(-ies)
Preconditions:

1. The Player and entity have tried to access the same tile.

Summary: Either the player or entity have collided, depending on the entity, the player either lives or dies.
Steps:

1. If the player is stronger than the entity, the entity will die.
2. Players score will be incremented by 1 point.

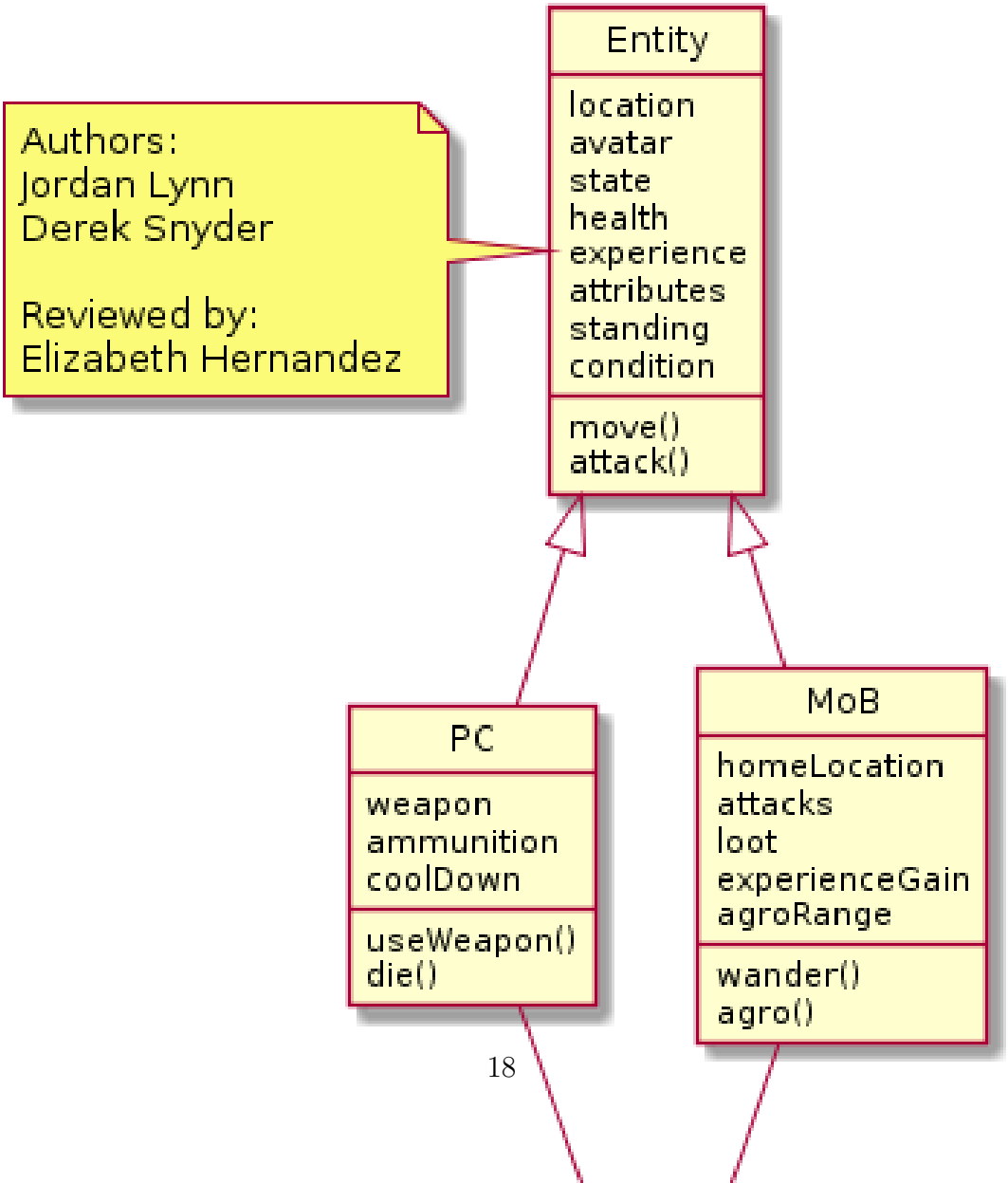
Alternatives:

1. If the entity is stronger than the player, the player will die.
2. Players score will be recorded.
3. Player will return to Main Menu Screen.

Chapter 5

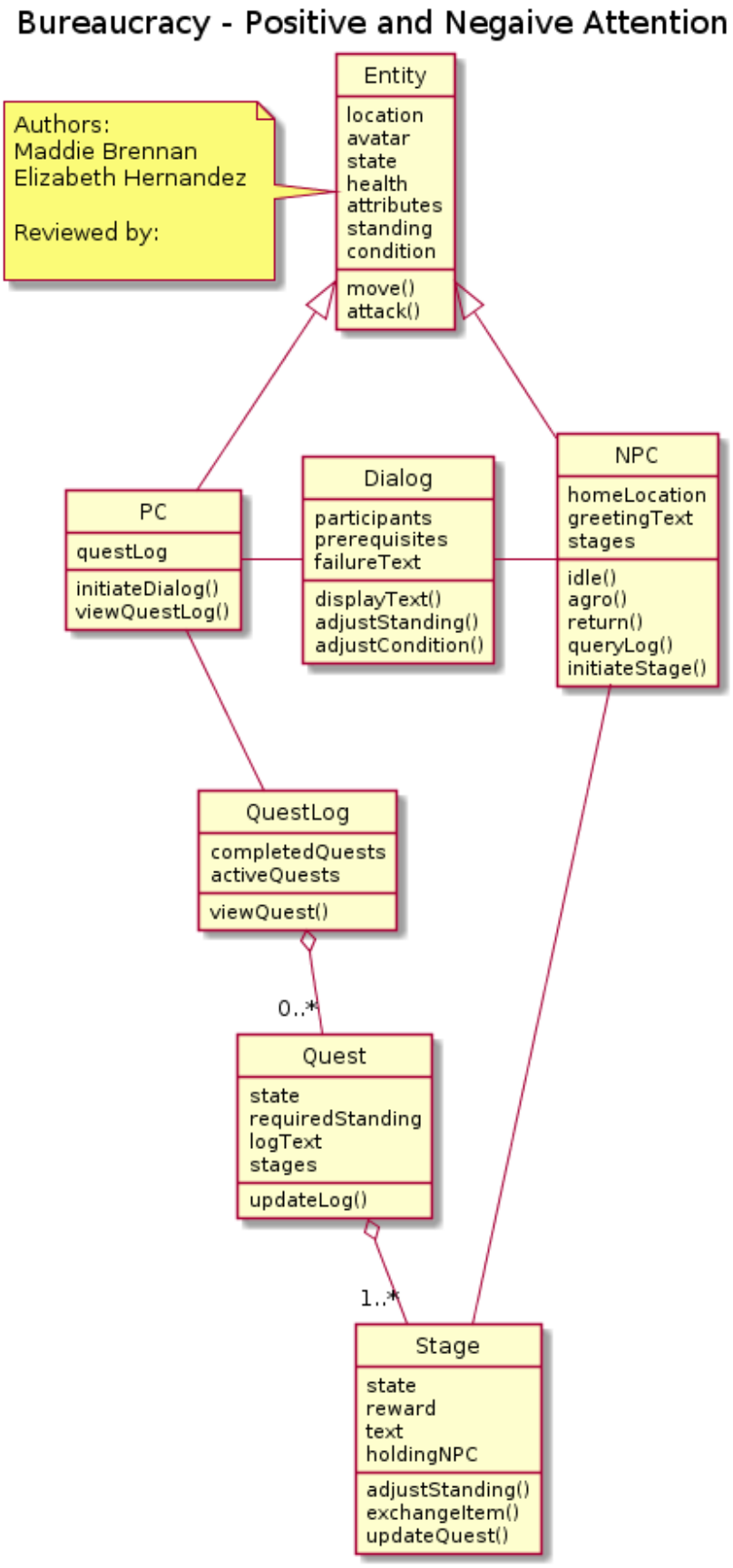
Initial Project Design: Class Diagrams

5.1 Combat



5.2 Bureaucracy - Positive and Negative Attention

Madeleine Brennan, Elizabeth Hernandez


Bureaucracy - Positive and Negaive Attention
Authors:
Maddie Brennan
Elizabeth Hernandez
Reviewed by:
Entity
location
avatar
state
health
attributes
standing
condition
move()
attack()
PC
questLog
initiateDialog()
viewQuestLog()
Dialog
participants
prerequisites
failureText
displayText()
adjustStanding()
adjustCondition()
NPC
homeLocation
greetingText
stages
idle()
agro()
return()
queryLog()
initiateStage()
QuestLog
completedQuests
activeQuests
viewQuest()
0..*
Quest
state
requiredStanding
logText
stages
updateLog()
1..*
Stage
state
reward
text
holdingNPC
adjustStanding()
exchangeItem()
updateQuest()

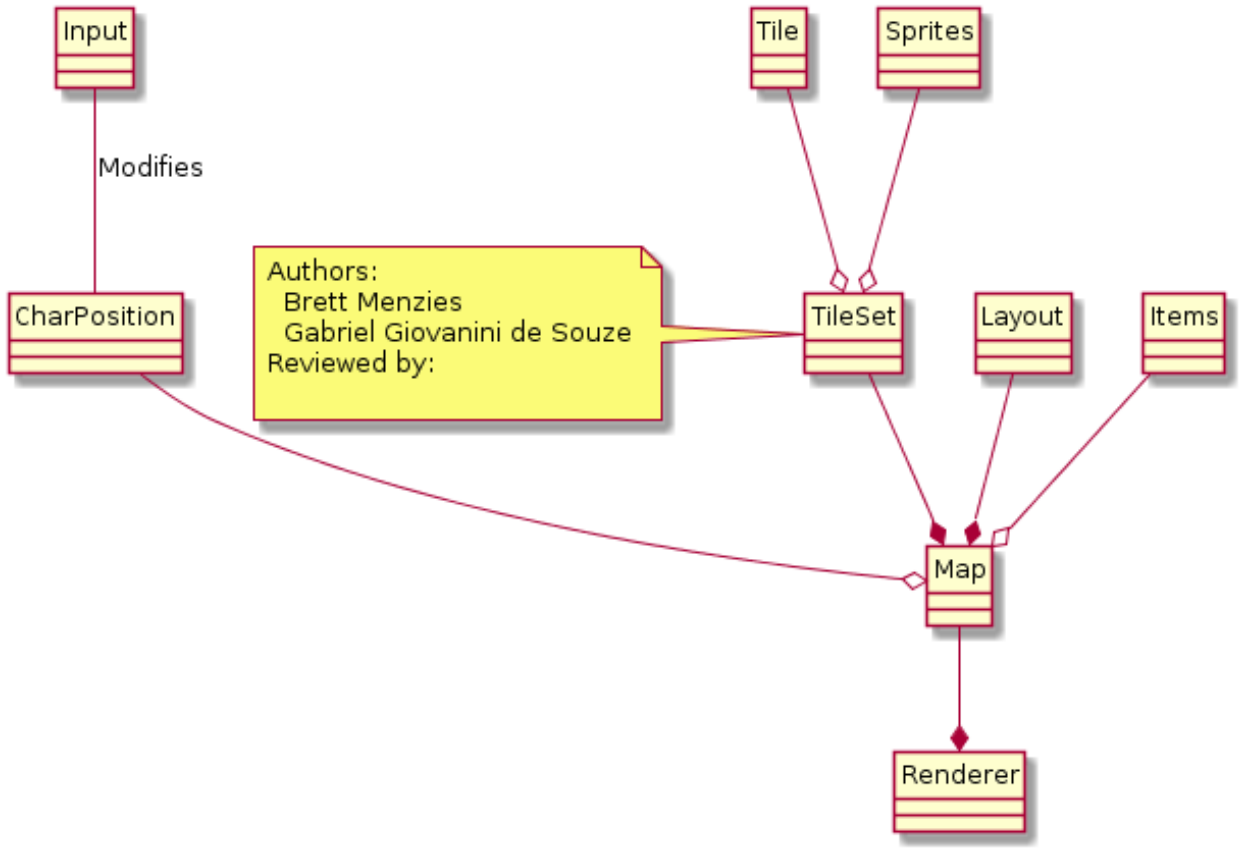
Description:

Bureaucracy deals with interactions between entities in the game, with the biggest interaction within this category being quests. The diagram above visually shows the classes that are related to receiving and completing quests. This relates to bureaucracy because completing quests will help achieve higher approval ratings with the faction that hosts the quest.

The root class is titled "Entity" and defines basic features of all characters in the game. The classes "NPC" and "PC", which stand for "non-player character" and "player character" respectively, are extensions of the Entity class. Each PC will track the state of their quests in a Quest Log. The Quest Log will consist of 0 to many quests. Each Quest consists of 1 to many quest Stages. Stages hold requirements and rewards for each step of progression in its parent Quest. Each Stage is held by an NPC. A PC must engage in Dialog with an NPC in order to gain and advance through Stages of their Quests.

5.3 Map and Movement

Brett Menzies, Gabriel Giovanini de Souze

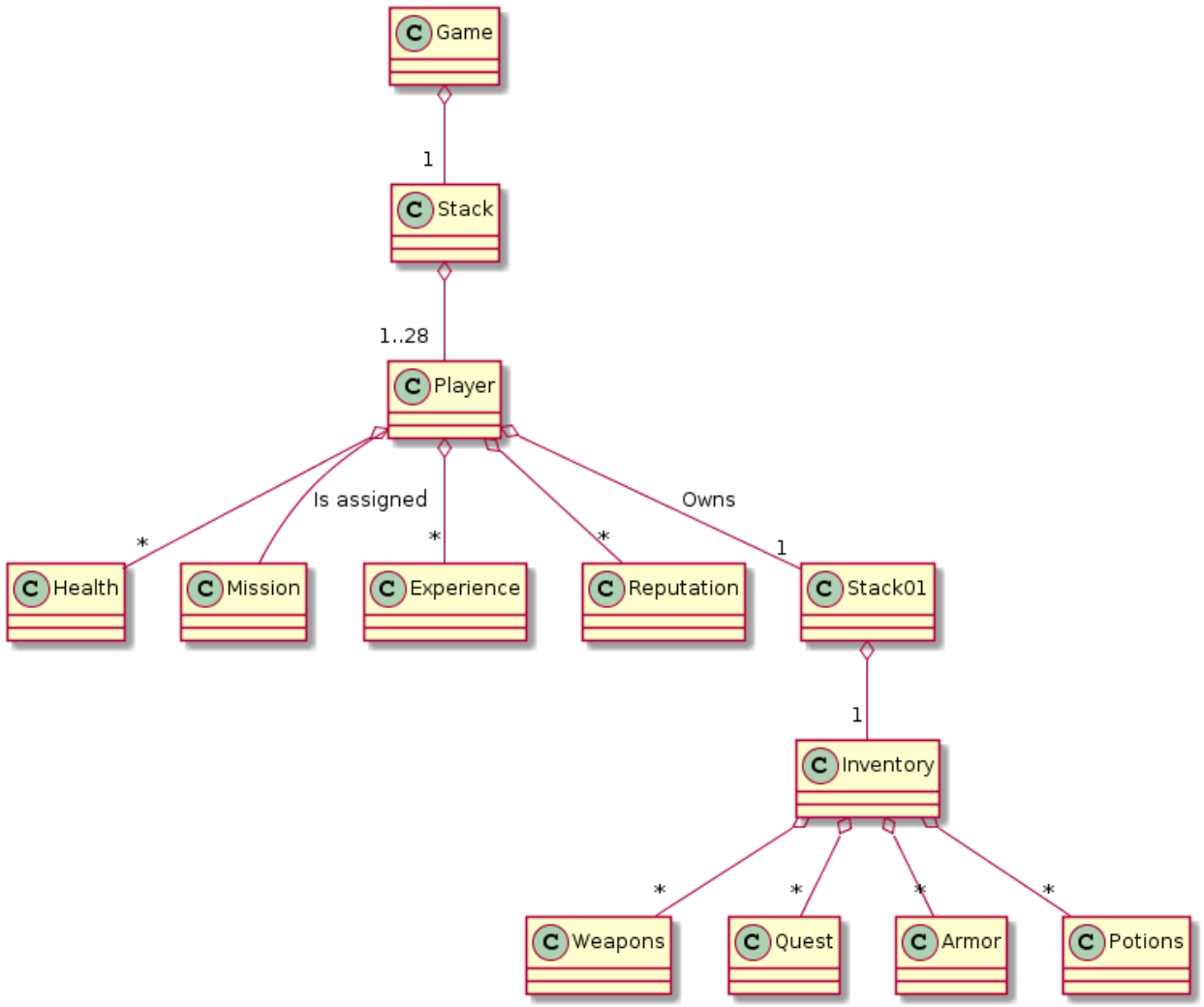


Description: The map subsystem revolves around the`Map" class, which is composed of a`Layout" and a`Tile set", and aggregates sets of "Item Position" and "Player Position" classes. A`Renderer" class requires an instance of a`Map"; Its other connections are outside the scope of this diagram. A`Tile set" aggregates "Tiles" (fixed size map tiles) and "Sprites"

(Other images drawn on a map, like items and characters) for use by the map and rendering code. Each “Item position” object represents the position and sprite id of items that have been dropped on the map. “Character position” acts similarly, except it can be modified by`input”, which represents networked, npc, and local player inputs for the purposes of this diagram.

5.4 Player

Michael Mueller, Alexia Doramus

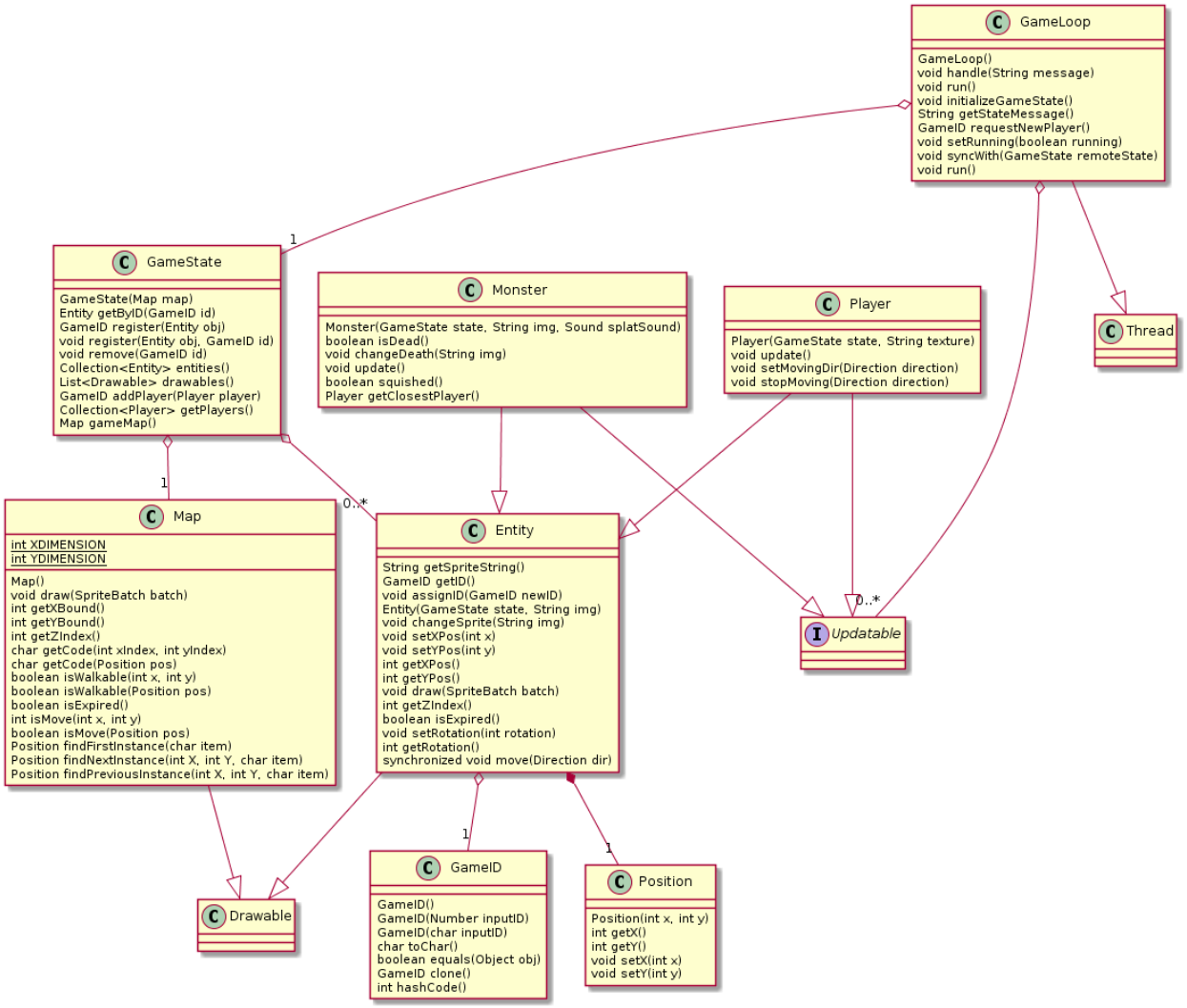


Description:
The player class is an aggregate of multiple parts. It consists of “health”, “experience”, “reputation”, and it owns an “inventory”. The player class can also be assigned a`mission”. The`inventory” is part of a stack that keeps track of the multiple items, including: “weapons”, “quest items”, “armor”, and “potions”. The`player” is part of a stack which can consist of up to 28 players. The stack that is an aggregation of the players is part of the game.

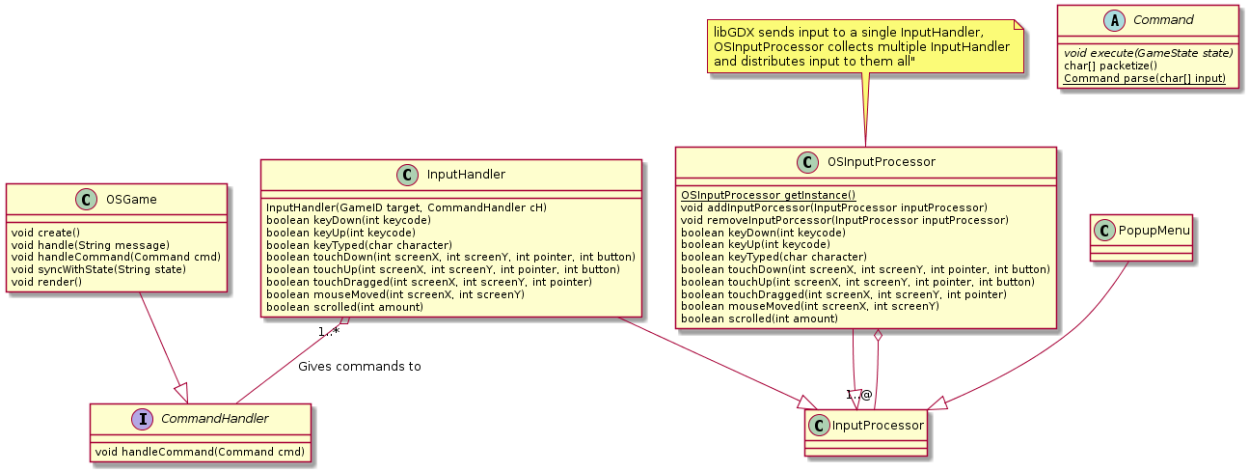
Chapter 6

Current Project Design: Diagrams

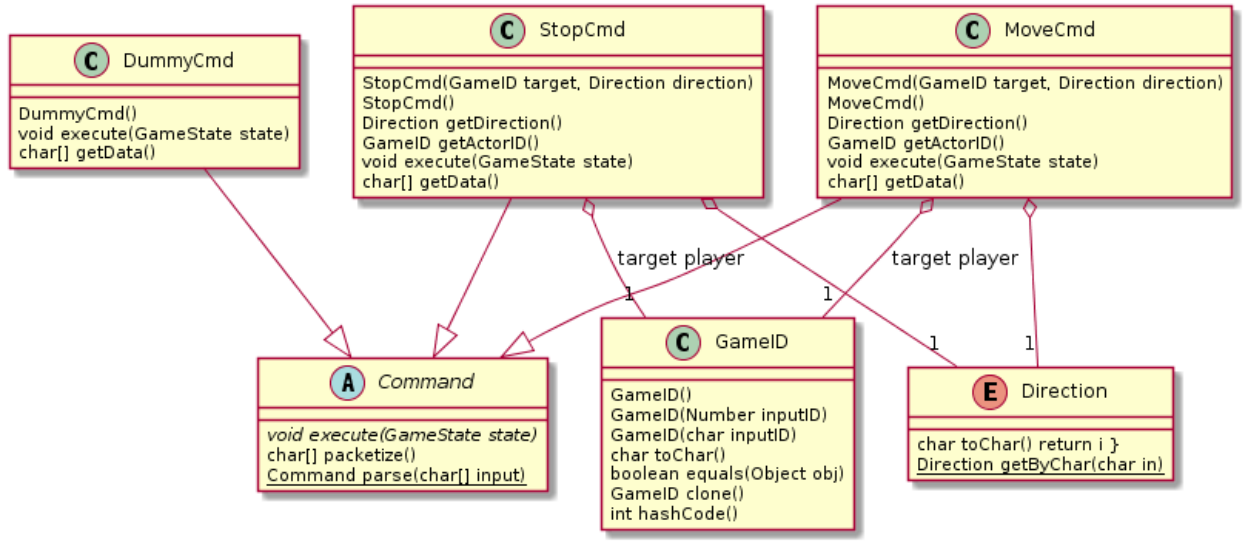
6.1 Game State Classes



6.2 Input Classes

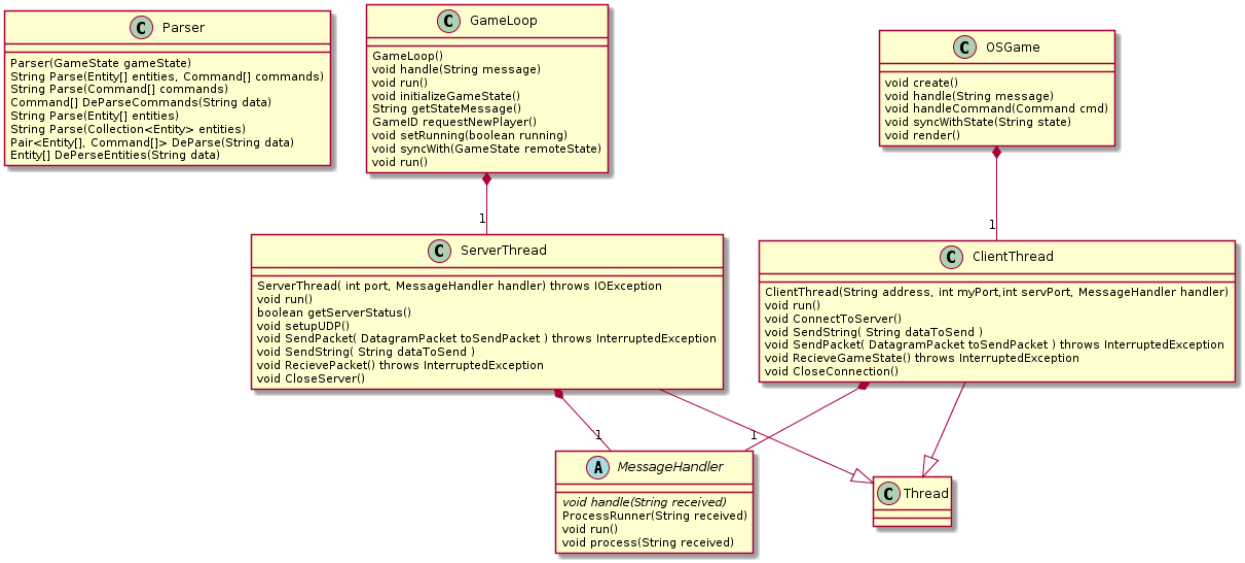


6.3 Command Classes

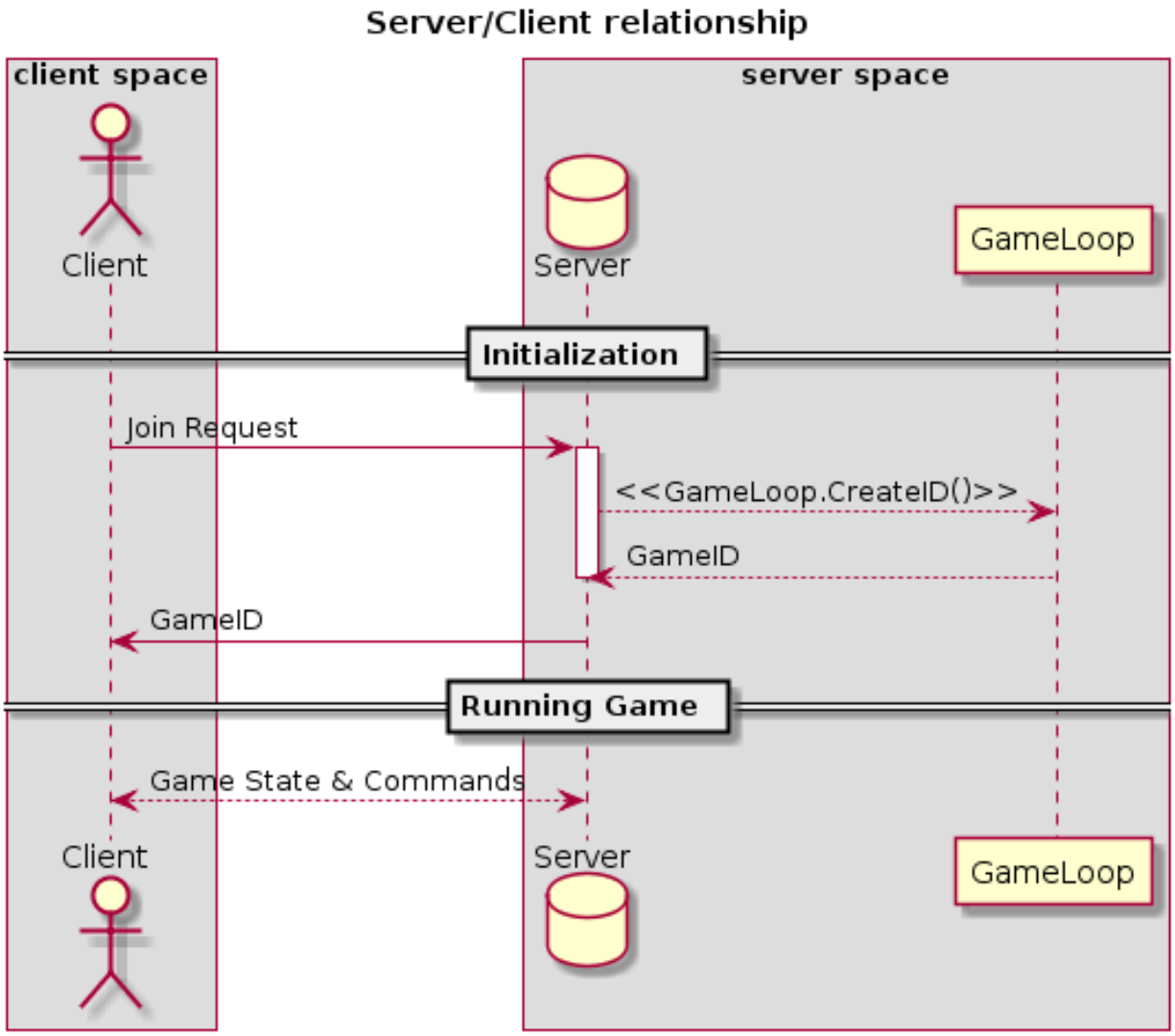


6.4 Networking Diagrams

6.4.1 Class Diagram



6.4.2 Sequence Diagrams



Chapter 7

Metrics

We chose to use PMD for our software metrics. Specifically, we invoked the Basic, Code Size, Coupling, and Design rulesets. A detailed output is availible as HTML in the "PMD" folder of our Documentation git repository. In short, PMD's biggest complaints were the high Cyclomatic Complexity of our monster AI (max of 18), a large number of Law of Demeter violations, and some variables that it suggests should be made "final".

Chapter 8

jUnit Testing

Our unit testing was run though gradle's test task, which uses JUnit. Our test code is located within the OSGame/core/src/test directory of our project, and run with "gradle test". A "Passed" or "Failed" message will be printed to the terminal for each test run. Standard error and standard output are also piped to the terminal. Gradle automatically makes a parallelized test runner, so there is no "main" to run our tests from.

8.1 Comms

8.1.1 GameID Test

```
public class GameIDTest
{
  LwjglApplication lwjglApplication;
  @Before
  public void before() throws Exception {
  }

@After
public void after() throws Exception{
}

@Test
public void testGameIDGeneration() throws Exception{
    ArrayList<GameID> seen = new ArrayList<GameID>();
    for (int i=0; i<200; i++) {
        GameID n = new GameID();
        assertTrue(``check previous values'', !seen.contains(n));
        seen.add(n);
    }
}
```

```
    @Test
    public void testCharTranslation() throws Exception{
        GameID origin = new GameID();
        char  idChar = origin.toChar();
        GameID result = new GameID(idChar);
        assertTrue(``Test GameID translation'', result.equals(origin));

        origin = new GameID((Number)42);
        idChar = origin.toChar();
        result = new GameID(idChar);
        assertTrue(``Test GameID translation'', result.equals(origin));
    }


}
```

8.1.2 Parser Tester

```
public class ParserTest
{
 LwjglApplication lwjglApplication;

 public void before() throws Exception
 {
     if (lwjglApplication == null)
     {
         LwjglApplicationConfiguration config = new
             LwjglApplicationConfiguration();
         config.title = ``Test'';
         config.width = 960;
         config.height = 576;
         lwjglApplication = new LwjglApplication(new ApplicationAdapter()
         {
             @Override
             public void create()
             {
                 super.create();
             }
         }, config);
     }
 }

/**                                                                    \
 * Method: Parse(Command[] commands)
 */
 @Test
```

```
public void testParseCommands() throws Exception
{
    Command[] commands = new Command[3];
    commands[0] = new DummyCmd();
    commands[1] = new DummyCmd();
    commands[2] = new DummyCmd();

    String result = new Parser(new GameState(new Map())).Parse(commands);

    assertEquals(``3,com.comms.DummyCmd:test,com.comms.DummyCmd:test,com.comms.DummyCmd:tes
        result);
}

/**
 * Method: DeParseCommands(String data)
 */
@Test
public void testDeParseCommands() throws Exception
{
    String test =
        ``3,com.comms.DummyCmd:test,com.comms.DummyCmd:test,com.comms.DummyCmd:test,'';

    Command[] result = new Parser(new GameState(new
        Map())).DeParseCommands(test);

    assertEquals(``com.comms.DummyCmd:test'', new
        String(result[0].getData()));
    assertEquals(``com.comms.DummyCmd:test'', new
        String(result[1].getData()));
    assertEquals(``com.comms.DummyCmd:test'', new
        String(result[2].getData()));
}

/**
 * Method: Parse(Entity[] entities)
 */
@Test
public void testParseEntities() throws Exception
{
    String resultString = ``0, ,0,2,2,@,0,2,3,\%,0,2,#,&,0,2,'';

    Entity[] entities = new Entity[4];

    Entity entity = new Entity(new GameState(new Map()), ``0'');
    entity.assignID(new GameID(' '));
    entity.setYPos(2);
    entity.setXPos(0);
```

```
        entities[0] = entity;

        entity = new Entity(new GameState(new Map()), ‘‘2’’);
        entity.assignID(new GameID(’@’));
        entity.setYPos(2);
        entity.setXPos(0);
        entities[1] = entity;

        entity = new Entity(new GameState(new Map()), ‘‘3’’);
        entity.assignID(new GameID(’\%’));
        entity.setYPos(2);
        entity.setXPos(0);
        entities[2] = entity;

        entity = new Entity(new GameState(new Map()), ‘‘#’’);
        entity.assignID(new GameID(’&’));
        entity.setYPos(2);
        entity.setXPos(0);
        entities[3] = entity;

        String result = new Parser(new GameState(new Map())).Parse(entities);

        org.junit.Assert.assertEquals(resultString, result);
    }

    /**
     * Method: DeParse(String data)                                          \
     */
        @Test
        public void testDeParse() throws Exception
        {
        Pair<Entity[],Command[]> pair = new Parser(new GameState(new
            Map())).DeParse(‘‘4,0,
            ,0,22,2,@,0,2,3,\%,0,2,#,&,0,2,3,com.comms.DummyCmd:test,com.comms.DummyCmd:test,com
        Entity[] entities = pair.getKey();
        Command[] result = pair.getValue();

        assertEquals(entities[0].getImageCode(), ‘‘0’’);
        assertEquals(entities[0].getID().toChar(), ’ ’);
        assertEquals(entities[0].getYPos(), 22);
        assertEquals(entities[0].getXPos(), 0);

        assertEquals(entities[1].getImageCode(), ‘‘2’’);
        assertEquals(entities[1].getID().toChar(), ’@’);
        assertEquals(entities[1].getYPos(), 2);
    assertEquals(entities[1].getXPos(), 0);
```

```
    assertEquals(entities[2].getImageCode(), ‘‘3’’);
    assertEquals(entities[2].getID().toChar(), ’\%’);
    assertEquals(entities[2].getYPos(), 2);
    assertEquals(entities[2].getXPos(), 0);

    assertEquals(entities[3].getImageCode(), ‘‘#’’);
    assertEquals(entities[3].getID().toChar(), ’&’);
    assertEquals(entities[3].getYPos(), 2);
    assertEquals(entities[3].getXPos(), 0);

    assertEquals(‘‘com.comms.DummyCmd:test’’, new
        String(result[0].getData()));
    assertEquals(‘‘com.comms.DummyCmd:test’’, new
        String(result[1].getData()));
    assertEquals(‘‘com.comms.DummyCmd:test’’, new
        String(result[2].getData()));

}
```

8.1.3 Command Memento

```
@Test
public void testPacketize() throws Exception {
GameID testID = new GameID();
Direction testDir = Direction.NORTH;
Command orig = new MoveCmd(testID, testDir);

char[] data = orig.packetize();

Command result = Command.parse(data);

assertTrue(‘‘Packetization class test’’, result.getClass()==orig.getClass());
MoveCmd mres = (MoveCmd) result;
assertTrue(‘‘Direction test’’, mres.getDirection().equals(testDir));
assertTrue(‘‘Actor ID test’’ , mres.getActorID().equals(testID));
}
```

8.2 Renderer Tests

8.2.1 Sprite Storage Tests

```
public class SpriteStorageTest
{
```

```
LwjglApplication lwjglApplication;

@Before
public void before() throws Exception
{
    if(lwjglApplication == null)
    {
        LwjglApplicationConfiguration config = new
            LwjglApplicationConfiguration();
        config.title = ``Test'';
        config.width = 960;
        config.height = 576;
        lwjglApplication = new LwjglApplication(new ApplicationAdapter()
        {
            @Override
            public void create()
            {
                super.create();
            }
        }, config);
    }
}

@After
public void after() throws Exception
{
    lwjglApplication.exit();
}

/**                                                                    \
 * Method: getInstance()                                              \
 */
@Test
public void testGetInstance() throws Exception
{
    assertNotNull(SpriteStorage.getInstance());
}

/**                                                                    \
 * Method: loadAssets()                                               \
 */
@Test
public void testLoadAssets() throws Exception
{
    try
    {
```

```
        SpriteStorage.getInstance().loadAssets();
        assertTrue(``SpriteStorage loaded assets without error'', true);
    } catch (Exception e)
    {
        assertTrue(e.getMessage(), false);
    }
}

/**                                                                    \
 * Method: getTexture(String code)                                     \
 */
@Test
public void testGetTexture() throws Exception
{
    assertNotNull(SpriteStorage.getInstance().getTexture(`` ``));
    assertNotNull(SpriteStorage.getInstance().getTexture(``X''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``^''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``v''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``@''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``Thomas''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``L''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``D''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``R''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``C''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``M''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``S''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``.''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``|''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``l''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``d''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``r''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``c''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``s''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``T''));
    assertNotNull(SpriteStorage.getInstance().getTexture(``/''));
 }
}
```