Bilkent University

Department of Computer Engineering

CS319 – Object Oriented Software Project

Project short-name: An Object Oriented Approach to Zork-Like (Text Based) Games

Final Report

Project Group 1

Fatih Karaoğlanoğlu
Erdoğan Ege Tokdemir
Furkan Emrehan Kılıç
İpek Lale

Final Report
Dec 25, 2014

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Table of Contents

1 Introduction

Text-based games are computer games that are displayed not with computer generated images, but with plain text. The user is, thus, free to use his/her imagination alongside a storyline expressed through text. Popular in the 1970s and 1980s, text -based games have now been largely replaced by video games with graphics; one can still program a text - based game to gain familiarity with programming due to their being easy to write[1]. This project was inspired by and aims to recreate the classical text -based game "Zork" (1977) using an object-oriented approach; the aforementioned being easier to write makes it possible to emphasize the object-oriented software engineering aspect of the project by rebuilding an old game using modern tools and paradigms.

Zork was released in 1977 for the DEC PDP-10 mainframe computer family, making it one of the earliest text-based games. It is an interactive fiction computer game that was written using the MDL Programming Language [2]. Even though text -based games have largely fallen out of the market since the advancements in computer graphics, some are still being written for current platforms. Zork is played in a command -line interface, where the user is supposed to type in commands to move his/her character. The game begins near a house in a forest inside what the developers call the "Great Underground Empire" from which the player must return with wealth and victory [1].

2. Case Description

The main goal is to implement a text-based game using an object-oriented programming language, unlike the original Zork, written in MDL. The proposed system will resemble Zork in terms of setting and game plot, but will differ largely in terms of software structure. Objects will be used to represent the character, the map and even the parser. Several other classes such as "User" will be implemented to further realize the game in an object - oriented fashion. The "User" class will also bring along a user management system; each user will have a username and a password, which when entered will maintain an individual account for saved game files and previous scores. This system may go as far as enabling one user to challenge other users online for a higher score. The parser will in fact have a class of its own and an instance of it will be initiated every time the player hits "new game". The entire system will reside in a class called 'GameSystem' which will manage instances of classes such as 'User', 'Parser' and 'Game'. Unlike the original Zork, this game will randomly generate the environment. Thus, the player will experience a unique gameplay each time he/she starts over.

Additionally, there will be features such as ignoring some typos in commands. For example, the original Zork parser would not recognize the command "taek", a mistyped version of the word "take". A sophisticated parser that will also recognize prepositions and conjunctions is ideal.

The system to be developed is a text-based game. As the name of the game's genre implies, the most important components of the game are user and game interactions since visual graphics do not exist. Written commands are the basic way to interact with the game. Unlike Zork, the game will provide flexibility for typos and case sensitivity.

The game plot will be based on survival. The game challenges a player to survive by displaying obstacles in his way such as hunger, heat and several hostile creatures. A player determines his/her score based on how well he/she can handle such situations. In addition, the player has a health rate which when drops to zero, ends the game. When the game ends, a high score table appears including the player's name, score and total number of moves.

Points and health will be calculated based on randomly generated values for some commands such as "eat" and "drink", thus enabling a chance factor in the game in addition to a skill factor.

3. Requirements Analysis

3.1 Functional Requirements

- In the game there must be an instructions screen which includes the description of the game and a list of commands that the parser understands.
- The game is controlled via mouse and keyboard.
- The game must have a score screen that shows the player's score.
- The game should tolerate typos and must not be case-sensitive.
- The game should include a "help" section.
- The game should recognize the following commands: look, search, go, take, attack, kill, use, examine.
- The user should be able to login to the game.
- The user should be able to save his/her progress in the game.
- The user should be able to load a previously saved progress.
- The user should be able to logout.
- The user should be able to restart at any point in the game.
- The user should be able to save only one instance of his playing.
- The user should be able to create a new account with a username and password.
- The game should display a high score table.
- The high score table should have dates, names and scores.
- The system should show the date in this format: dd / mm / yyyy.
- Multiple players should be able to play the game, although not simultaneously.

3.2 Non-functional Requirements

- The response time for commands should be less than one second.
- The game should be able to run on Windows, Mac and Linux systems.
- The content of the game should be easy to understand.
- There should be separate classes for each construction (locations, items, characters etc.) in the game to ease testability and increase flexibility of the system (working with separate classes make it easier to add features to the system)
- The level of expertise of the user shall be basic.
- The system should store a maximum of 10MB of data.
- The user should have JRE installed in his system in order to run the program.
- The user's system should have at least 128MB of installed memory.

3.3 Pseudo Requirements

1. The project should be completed within three months.
2. Java should be used as a programming language.
3. The system must be a desktop application.
4. The system must be distributable.

3.4 Scenarios

Overall usage

Lucy, a student at Bilkent University, has a lot of assignments and exams. She often gets bored and feels the need to play a computer game to relax; she runs the game, creates an account and starts playing. After a while, she gets bored, saves the game and quits. Later, she comes back and runs the game again. She loads her save file, and continues to play where she had left off. Eventually, she dies in the game. Then high score table appears and her score, name and time of end game appears along with other players. She then quits the game and continues with her studies.

Controls

During the game, she uses the "help" command to learn how to play the game. Then, using the commands that appear in the "help" section she moves the character and tries to survive the obstacles.

3.5 Use-Case Models

This diagram (Figure 1.1) shows the Use Case Model for logging in. The login operation can either return a success or a fail. The "load game" and "new game" operations follow "login".

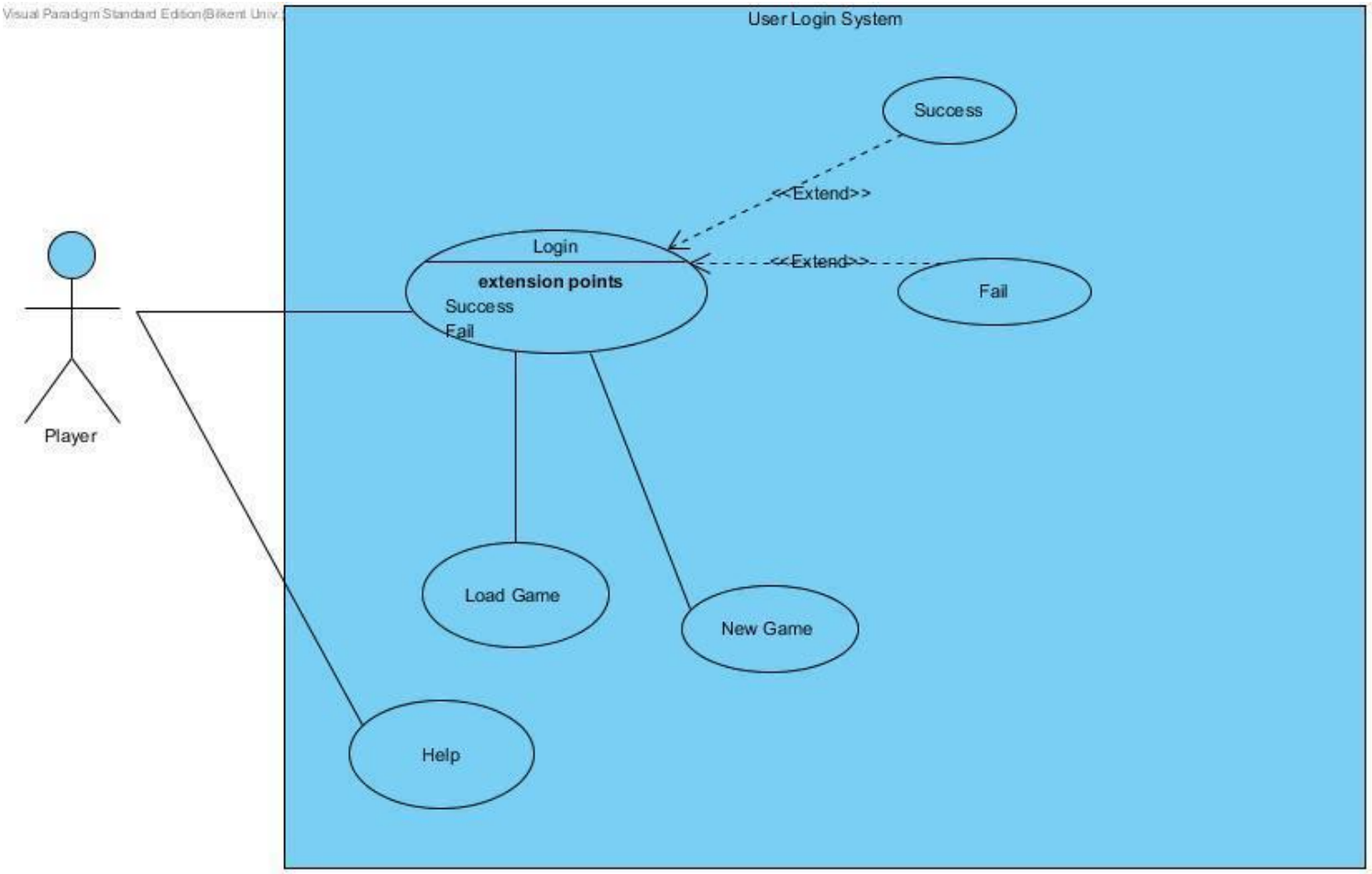


Figure 1.1

This diagram (Figure 1.2) shows the Use Case Model for a gameplay. The list of commands is as below. The entire gameplay will consist of typing commands into the command-line, some of which extend other commands.

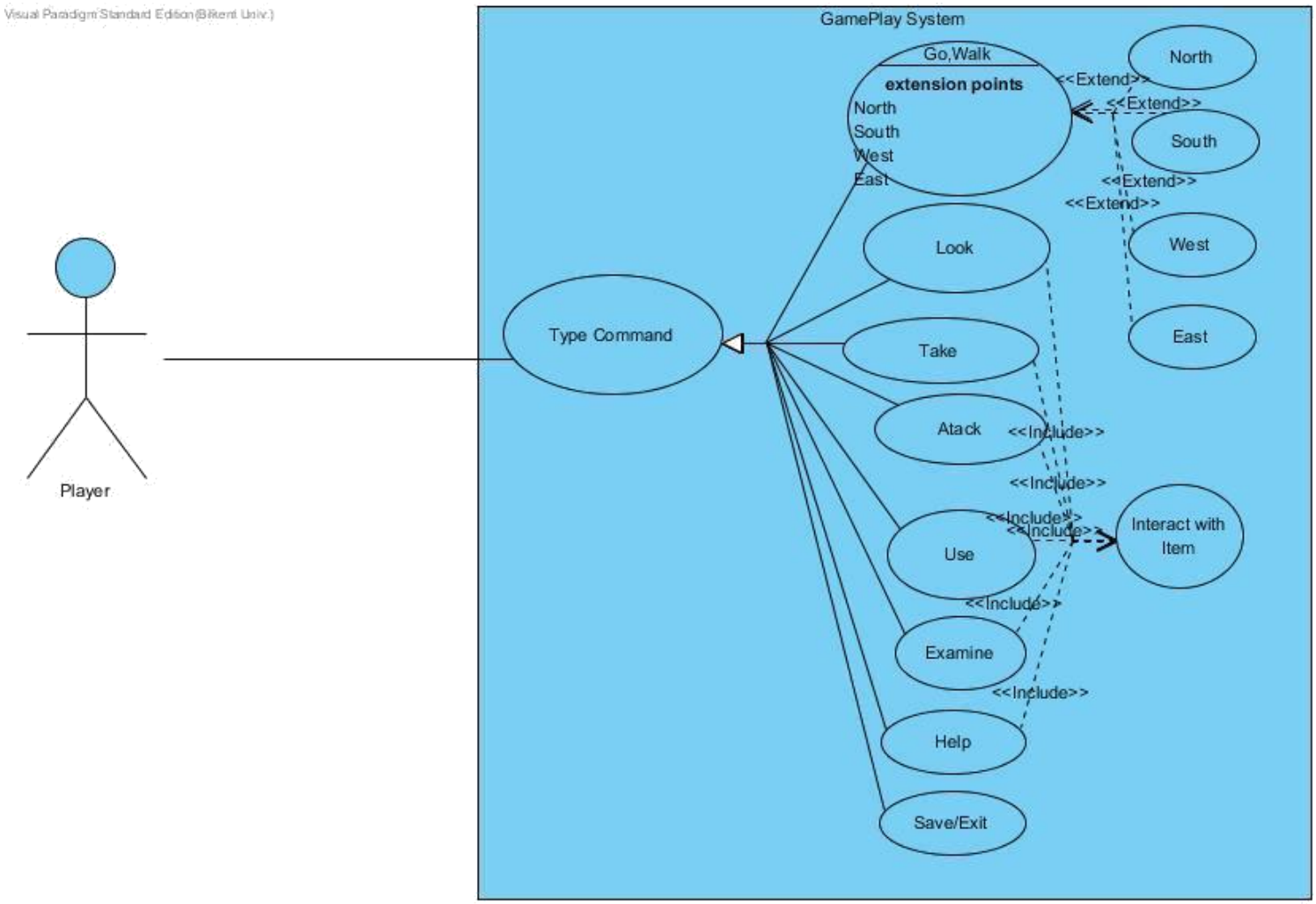


Figure 1.2

3.6 User Interface/Screen mockups

Below are screenshots of the system's interface:

```
You are in a vast field of green. A rather soft breeze blows at your direction,
 carrying with it a wicked scent of sulfur. It must be coming from close by.
go West
You are in a swamp. If you go any further in this direction you are sure to be sucked into an abyss of gooey nightmare.
A mighty troll blocks your way!
```

```
A mighty troll blocks your way!
attack troll elvish_sword
You swing your Elvish Sword with all your might!
Necati inflicted 24 damage to Troll
Troll fought back
Troll inflicted 18 damage to Necati!
```

```
take lantern
Taken
take elvish_sword
Cannot take item because your inventory is full.
```

4 Analysis Models

4.1 Object Model

4.1.1Domain Lexicon

User: A person who plays the game
Account: Entity that is created by user with a password and user name and enables user to login the game.
Username: Entity that is created by user in order to login to the game.
Password: Entity that is created by user to encrypt his/her account.
Player: Entity that User controls in the game.
Hostile Character: Entity that is hostile against the player in the game.
Location: Locations that user can wander through in the game.
Map: Collection of Locations.
Furniture: non-movable entities in the game that user can interact.
Direction: self-explanatory

4.1.2 Class Diagrams

This diagram (Figure 2.1) shows the entire structure of classes and their relations. The GameSystem class, as mentioned earlier, is the class that manages all operations related to individual users. It holds an instance of "Parser" which in turn receives a Game instance as parameter. When a new game is to be started, a new instance of Game is generated inside GameSystem.

The Parser class can manage one Game object at a time. Every command entered into the terminal is delivered to the Parser object as a string, which is then interpreted in the form of a modification to the Game object. The Game object is the next class in the hierarchy and it can hold instances of several other classes such as Map and Character. Character is further divided into the Player and Non-Player Characters, which are dungeon trolls, ogres and other similar creatures the player can combat. As stated earlier, the map of the game will be randomly generated according to a seed. The map will consist of Location instances that can be thought of as coordinates linked to one another. An additional detail here is that these Location instances will represent a two dimensional plane, while some Location instances will be linked to another Location instance that resides on a completely different plane of Locations which can be accessed by "climbing up" or "climbing down" a certain structure, or Thing, inside that particular Location instance, which ultimately imitates a three dimensional environment.

Lastly, the aforementioned Thing class represents all objects available in the game, i.e items that can be acquired by the player or other stationary structures such as walls, doors, tables etc. The classes that inherit the Thing class are appropriately titled Item, which is divided further into edibles and inedibles (weapons, treasure, pamphlets etc.) via class attributes; and Furniture, both respectively modeling the previously mentioned objects in the environment.

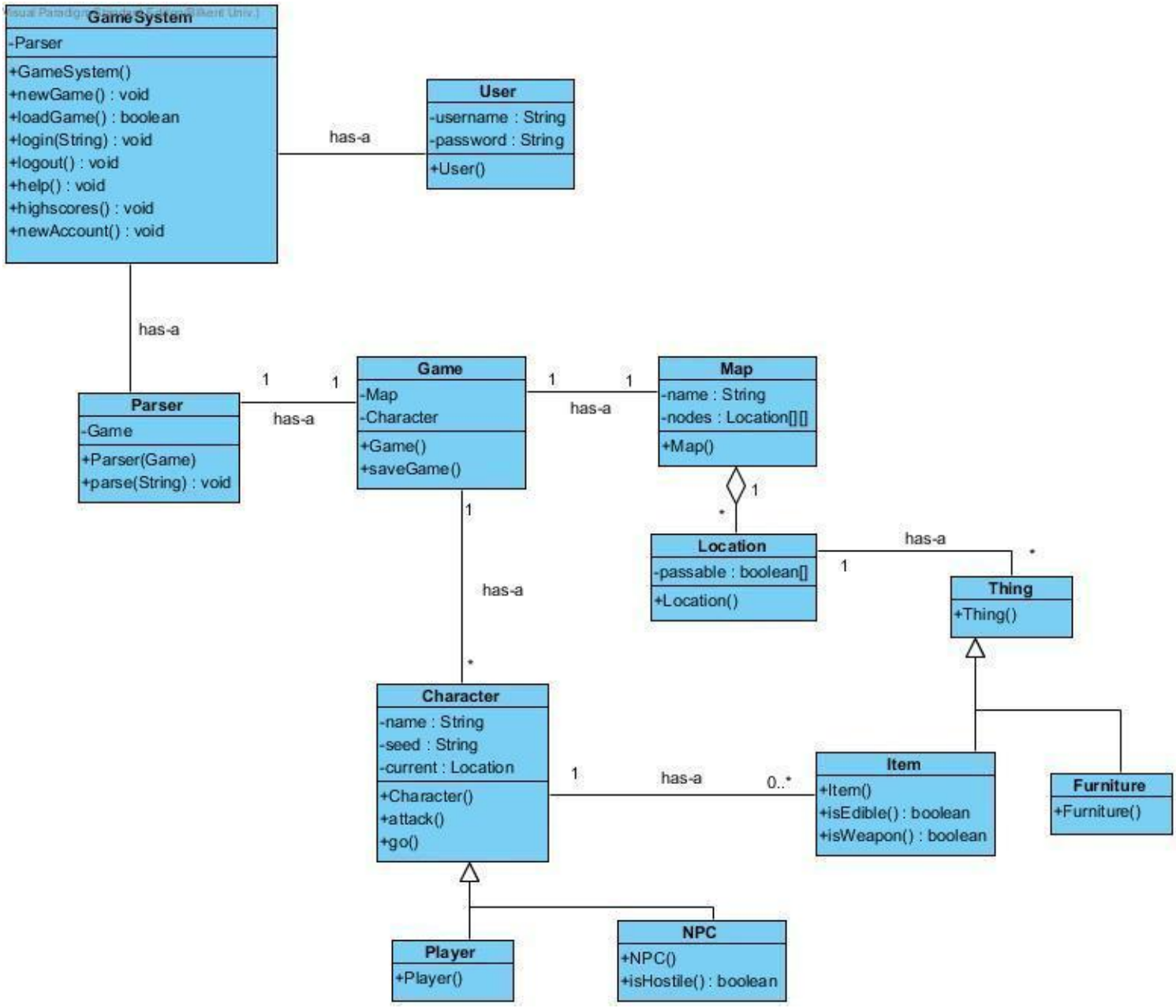


Figure 2.1

4.2 Dynamic Models

4.2.1 State Chart

This diagram (Figure 3.1) shows a State Machine Diagram that realizes the entire system. The "Parse" state models actual gameplay and the rest model the non-gameplay system.

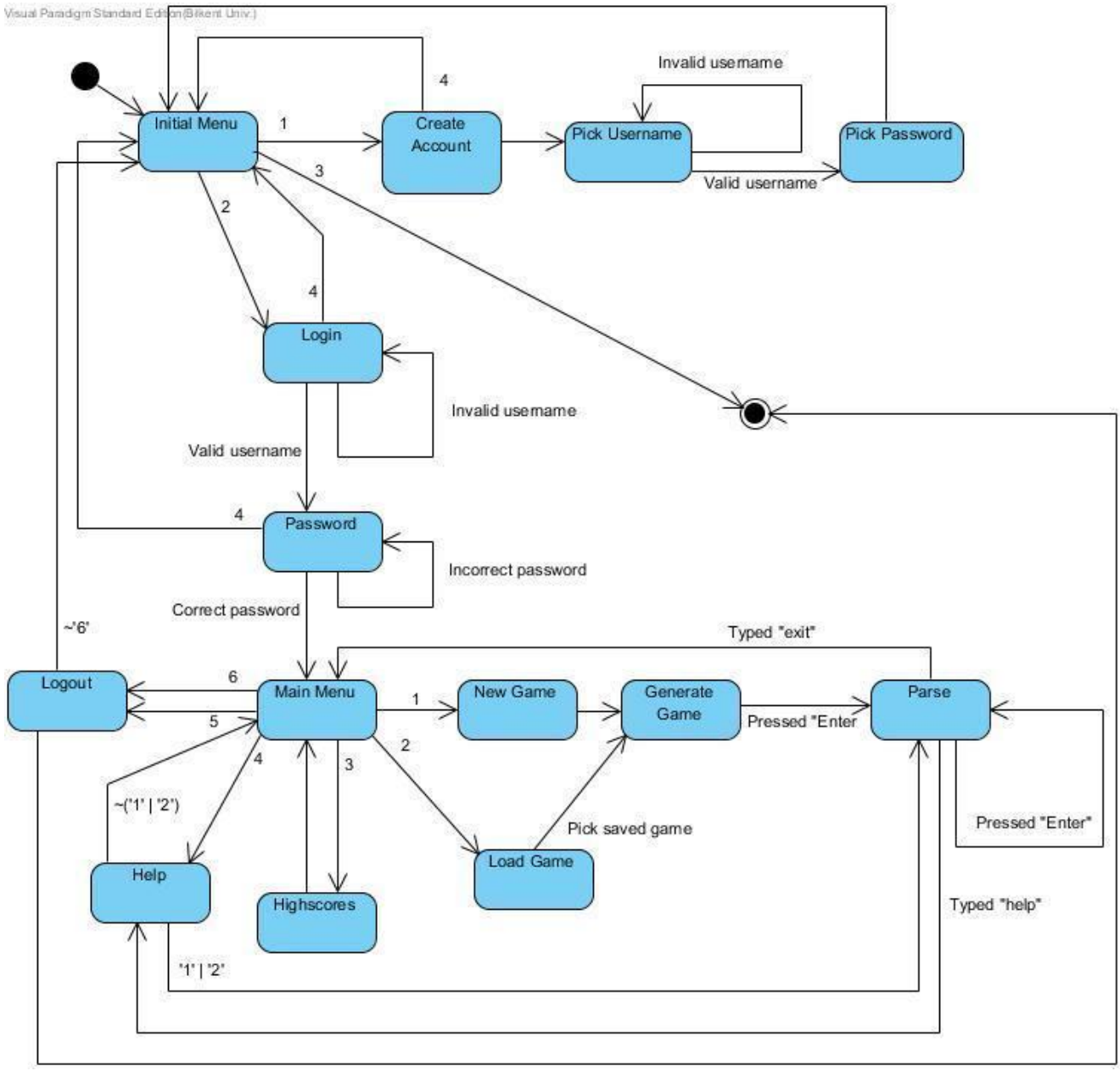


Figure 3.1

This is an additional table (Table 1) that describes the meanings of the numbers in Figure 3.1.

Table 1

Initial Menu	Main Menu
1. Create account	1. New game
2. Login	2. Load game
3. Quit	3. View high scores
4. Back (only used after 1 or 2)	4. Help
	5. Logout
	6. Quit

4.2.2 Sequence Diagram

This diagram (Figure 3.2) shows a Sequence Diagram for the saveGame() function.

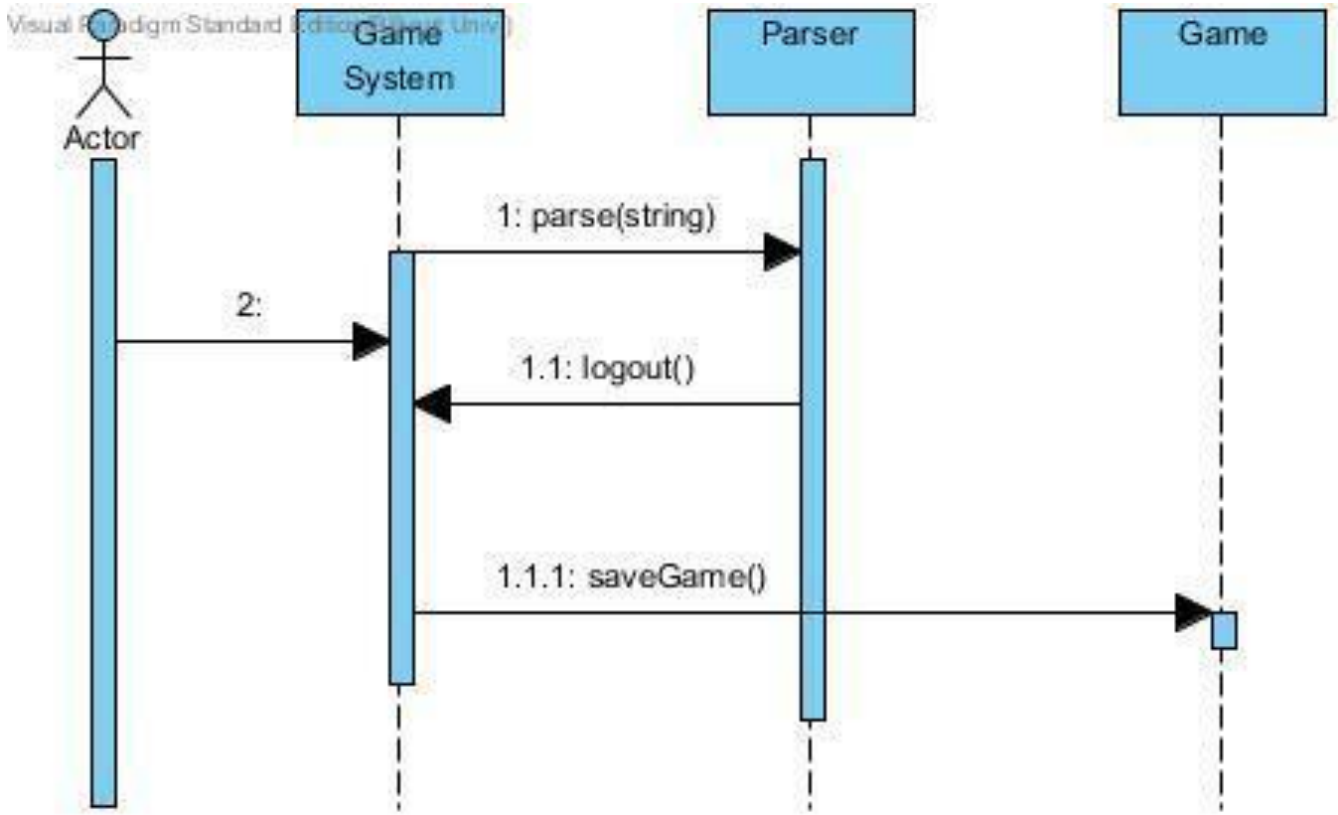


Figure 3.2

This diagram (Figure 3.3) shows a Sequence Diagram for the function startGame() which initiates a new game.

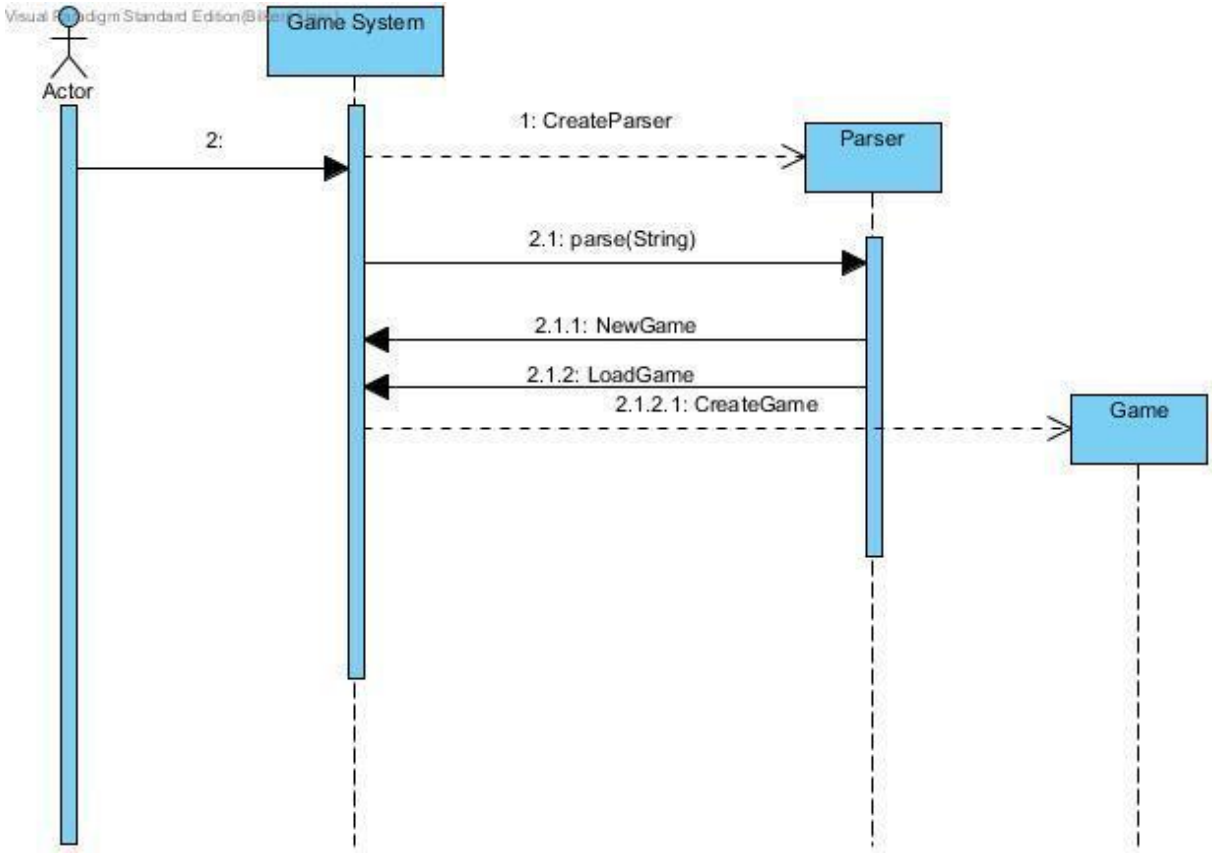


Figure 3.3

5. System Design

5.1 Design Goals

・ Allow user to complete their task without being distracted by software or losing train of thought

・ After user types commands system responses in less than one second

・ Allow user to run the system on all types of operation systems

・ Give users access to information they need to complete their task (i.e. information on other pages, etc.).

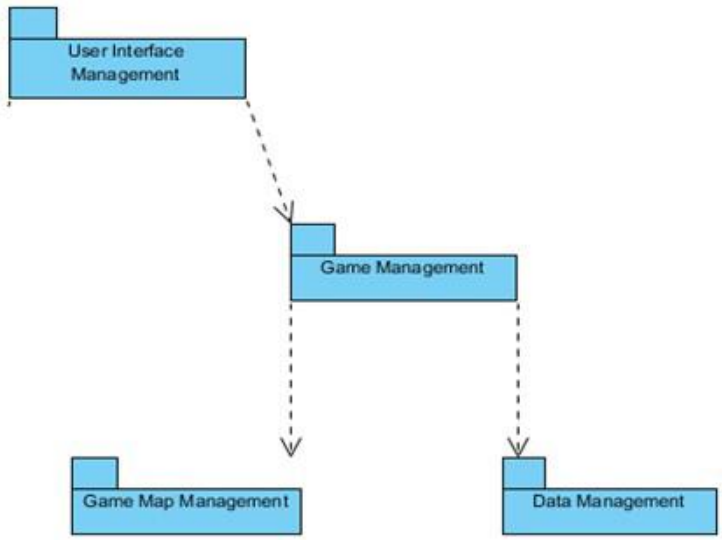
・ Add clear tutorial about game to the help menu to make system easily understandable

・ System will not interrupt the game play because of the small typos of the player

・ System can be modified without changing entire structure of it

5.2 Sub-System Decomposition

We decided to handle our games in four subsystems which connect themselves
with the interactions shown in the figure. Details about subsystems and their classes are given in the subsystem services part.



User Interface Management Subsystem

User Interface Management Subsystem includes a ConsoleView Class

ConsoleView Class

ConsoleView class is used to view all menus that can be reached by the main menu of the game. This class is the first class that is initialized when user opens
the game. The UI that is created by this class includes a text based interface with
following options “Play Game”, “Instructions”, “High Scores”, “Settings” and “Exit
Game”. When “Play Game” option is choosed, further commands are asked from the
used by providing a set of choices in a text based format. If “Instructions”, “High
Scores” or “Settings” option is choosed, new text based interface is created by the
class in order to show the information that is wanted by the user. ConsoleView class
gets required information from GameSystem class. Last choice on the menu, “Exit
Game”, if any progress made it saves it and finally exits the game.

Game Management Subsystem

Game Management Subsystem includes following classes.

GameSystem Class

GameSystem manages all aspects of the game. GameSystem supplies information to UI-related classes with the information it gets from the classes of the
Attribute Management Subsystem and Data Management Subsystem, and controls

the progress of the game and updates the game during game-play. Decision of
whether the game is over or not is also made by this class.

Game Class

After user starts to play the game, game class becomes the messanger of the system. It helps to the objects to reach t othe classes in other subystems. For example object of the Reader class can call methods of the player object with the
instance of Game object which is a property of Reader
class(game.getPlayer().go(Direction.d) will be called from Reader according to Input)

Character Class

Character class is an abstract class which is expanded by Player class and NonPlayerCharacter class

NonPlayerCharacter class

NonPlayerCharacter class is an expanded by HostileCharacter class. It exist to make
it possible to make friendly NPC's later on

HostileCharacter Class

Instances of HostileCharacter class tends to attack to player

Player Class

Player class is where all player stats and action methods exist.

Game Map Management Subsystem

Game Map Management Subsystem includes 3 classes which are Map, Location and
LocationFactory class.

Map Class

Map class keeps a 2 dimensional Location array and if player hits the bounds it will
extend it from every side. It has a LocationFactory which will generate locations the
player moves.

Location Class

Location class keeps a list of Things as a property, which will be randomly generated
by LocationFactory.

LocationFactory Class

LocationFactory class keeps a queue of Locations, and it will feed the map with those
when player moves to an unvisited location. It will also generate new maps when idle
to keep the queue non-empty.

User Management Subsystem

User Management Subsystem includes following classes.

UserManager Class

It handles User Login and data save operations

User Class

Object of this class is initialized when user login, it will carry information about user
and this data will be saved before exitting

Map Objects Subsystem

Map Objects Subsystem includes following classes.

Direction Enumeration

It carries direction information. Used in many direction based methods across the
system.

Thing Interface

An Interface that implemented by Item and Furniture Classes

Furniture Class

Furnitures that exists in locations, player cant take those but may interact with it

Item Class

Items that exist in locations, and inventories of characters. Can be taken and used
accordingly by player. This includes weaponry and foods.

IO Subsystem

Reader Class

Reads and analyzes the typed user input, and calls player methods accordingly.

5.3 Architectural Patterns

We have applied the three-tier and the pipe-and-filter architectural patterns.

5.3.1 Three-tier Pattern

This was seen as an appropriate pattern because the system contains three independent modules: the user interface, the functional process logic and the data storage. The functional process logic module contains functionalities such as parsing the input given by the user, maintained at large by the Reader class. This module carries out the detailed task of analyzing user input.

The user interface module contains classes responsible for displaying the outputs of the system. This is the topmost module in this architecture, and it communicates with the user as well as the other two modules. The data storage module is responsible for providing data to the save/load functions (logic module), as well as providing data for the generation of Location objects inside the Map object. This module uses XML files to store relevant information at move it up the architectural ladder when required.

One important advantage of this pattern is that it provides independency among the modules, meaning that each of these modules may be upgraded separately, when necessary. For example, in case one decides to upgrade this project to have GUI, the user interface module is the only module needed to be modified.

5.3.2 Pipe and Filter Pattern

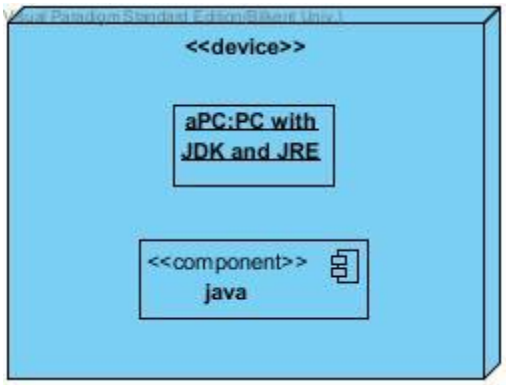
The pipe and filter pattern may have been appropriate for this system, since each subsystem sends and receives information from and to other subsystems. The user interface management subsystem, which manages the parsing of user input, sends information to the game management subsystem, which is only acknowledged about the input that it receives, regardless of where it came from. The pipes become the information being transferred, and the filters the subsystems. The game management subsystem reads the input and in turn sends information to the game map management subsystem.

5.4 Hardware/Software Mapping

In our project Java will be used as a programming language. Users who have enough tools to execute Java projects will be able to execute our program. Only software requirement for our system is that.

When issue comes to hardware requirements, a regular keyboard is enough. Keyboard will be used to initialize the game, enter login information and commands to direct the game.

As a result, our project is not required an extra tool than that already exist in all computers.



5.5 Addressing Key Concerns

5.5.1 Persistent Data Management

User information and save files will be saved as .DAT files to the hard drive. Interior of these files will be decided during implementation.

5.5.2 Access Control and Security

Upon creation of an account, login credentials of that account will be stored to the hard drive. When a user wants to play the game players must type in their login ID's and passwords. If login is successful their game data will be loaded and game will start.

Otherwise they will get an error message explaining the situation. Users can access other players via a proxy and they only will be able to see their game condition and scores.

5.5.3 Global Software Control

The control flow that is used in this project is the event-driven control because it is the one that is most appropriate for an object-oriented programming language, such as Java. The main loop of the program is going to take an external event as input, in this case, the commands that the user enters. This is a fairly simple control flow, but our design does in fact require threads for maintaining Location instances.

Threads will become necessary for the generation of Location instances. A queue will maintain Location instances that are generated at the beginning of the program, and at several other points in the game. In order for the user not to experience a slow gameplay, a thread will be used for the generation of Location instances, such that Location instances will be generated whenever the number of Location instances drops below a threshold. The main loop and Location generation will run on different threads.

5.5.4 Boundary Conditions

Initializaton

Since Zoork does not have regular .exe or such extension, it will not require any installation. Game will be initialized via a .jar file.

Termination

Because of the necessity of saving, game must be terminated with exit command otherwise system will not be able to save and store game data and will lose the process achieved since last save.

Error

In the case of the corruption of the data system will not be able to recover the data and all processes will be lost.

5.5.5 Object Design Trade-Offs

This section defines the different tradeoffs in object design and the need to make decisions about them. In the making of a text-based computer game, aspects such as memory usage, availability, durability and maintainability are important. Compromises need to be made from these in order to come up with the optimal design.

- Memory space vs. response time

Text-based games often need a good share of memory even though no graphics are involved.
As more memory is required to store all the user's data, response time may deteriorate; this, however, should not happen. The management classes involved in sending and receiving data, in turn, take care of this problem by limiting data transfer to only when it is required.

- Buy vs. build

Usually, a build policy may be sought because of the simple design of most objects. The parser component, however, may be supplied from elsewhere, which will in fact save time. The compromises needed to be made here is between cost and time, and one component may be bought if the cost is not too high, and if it will save time.

- Platform dependence vs. flexibility

The game needs an optimal platform, such as the operating system and even architecture. This optimal platform will allow perfect programmability for the game, but it also needs to be
flexible, such that only subtle changes to the design will grant it new platform dependency properties. Most games in the market are optimized for one platform, however since this is a text-based game, the switch between platforms is as simple as an integrated pattern.

6. Object-Design

In the object design part, we explained the design patterns that we applied and class interfaces which describe classes.

6.1. Design Patterns

We applied four Design Patterns, which are Singleton, State, Command and Façade patterns.

6.1.1. Singleton Pattern

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. We used this pattern in GameSystem, GameReader, LoginReader,UserManagement and LocationFactory

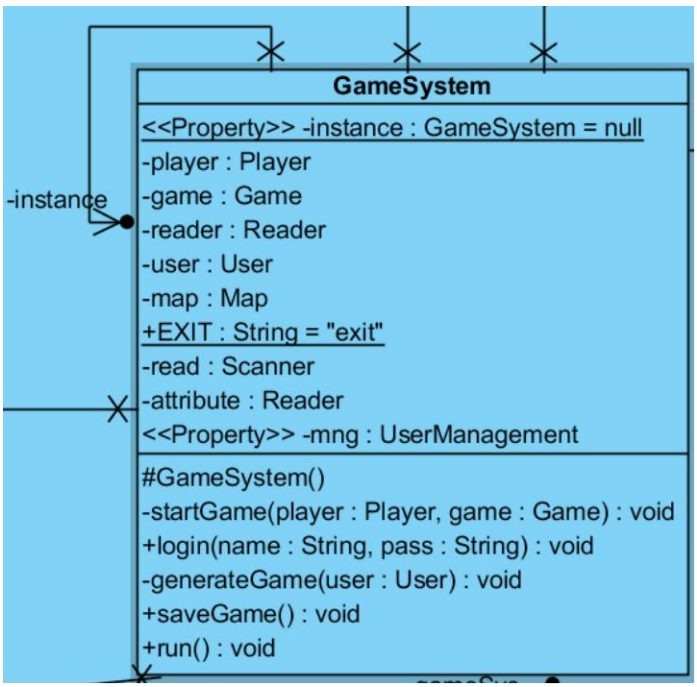
GameSystem

GameSystem class is the main controller of the game. It keeps current game data, and the main game loop inside.

Code of the pattern as follows

```
protected GameSystem(){
   mng = UserManagement.getInstance();
   read = new Scanner(System.in);
}
public static GameSystem getInstance(){
   if( instance == null){
      instance = new GameSystem();
   }
   return instance;
}
```

Constructer is protected so it can only be instantiated with getInstance() method.



Readers

Reader classes are responsible with analysing the user input. While LoginReader handles login, register inputs, GameReader handles the commands user types in.

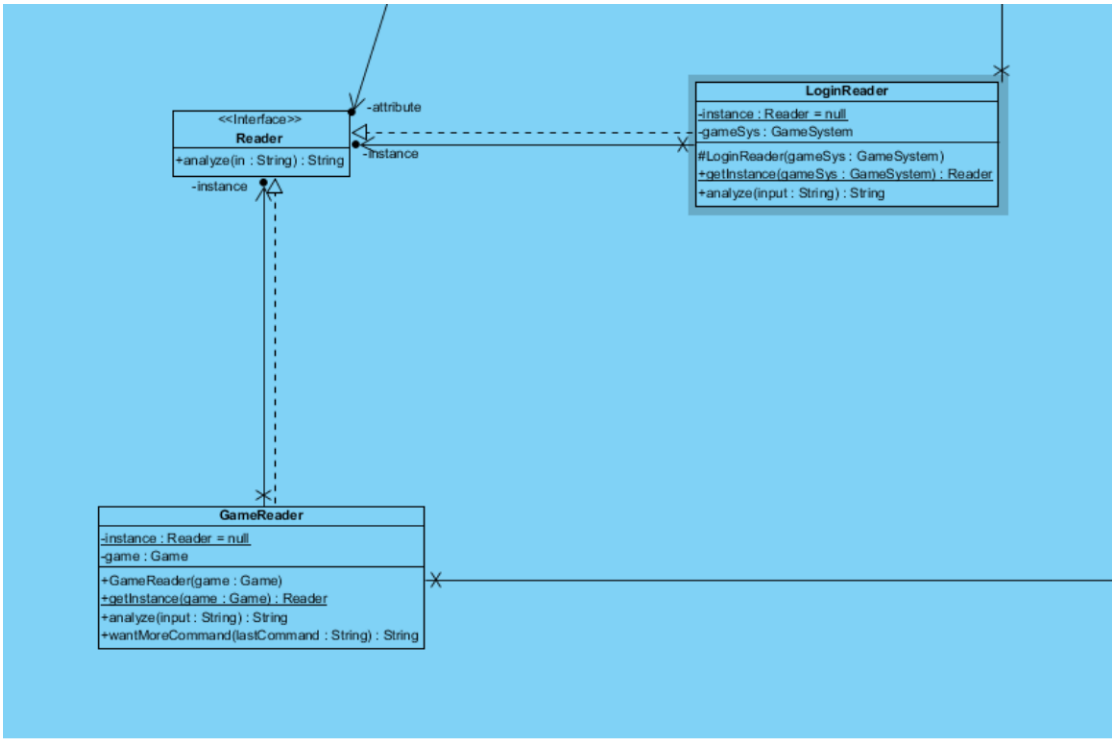
Codes of the pattern as follows

```
private static Reader instance = null;

protected LoginReader(GameSystem gameSys)

{

    this.gameSys = gameSys;
```

```
}
public static Reader getInstance(GameSystem gameSys)
{
    if(instance == null)
    {
        instance = new LoginReader(gameSys);
    }
    return instance;
}
```

and

```
private static Reader instance = null;
public GameReader(Game game)
{
    this.game = game;
}
public static Reader getInstance(Game game)
{
    if(instance == null)
    {
        instance = new GameReader(game);
    }
    return instance;
}
```



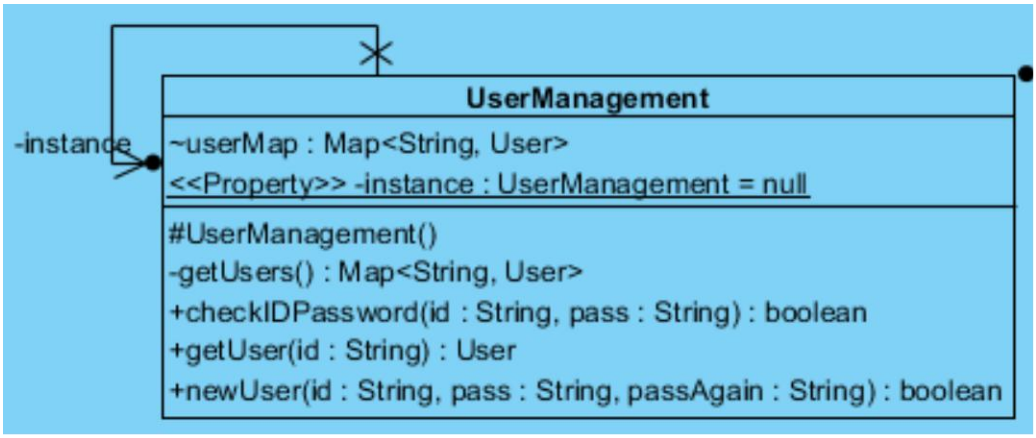
UserManagement

UserManagement class handles User database actions and it has login-register methods.

Codes of the pattern as follows

```
private static UserManagement instance = null;

protected UserManagement(){
        try{
            userMap = getUsers();
        }catch(FileNotFoundException e){
            e.printStackTrace();
        }
    }
public static UserManagement getInstance() {
    if(instance == null) {
        instance = new UserManagement();
```

```
}
   return instance;
}
```



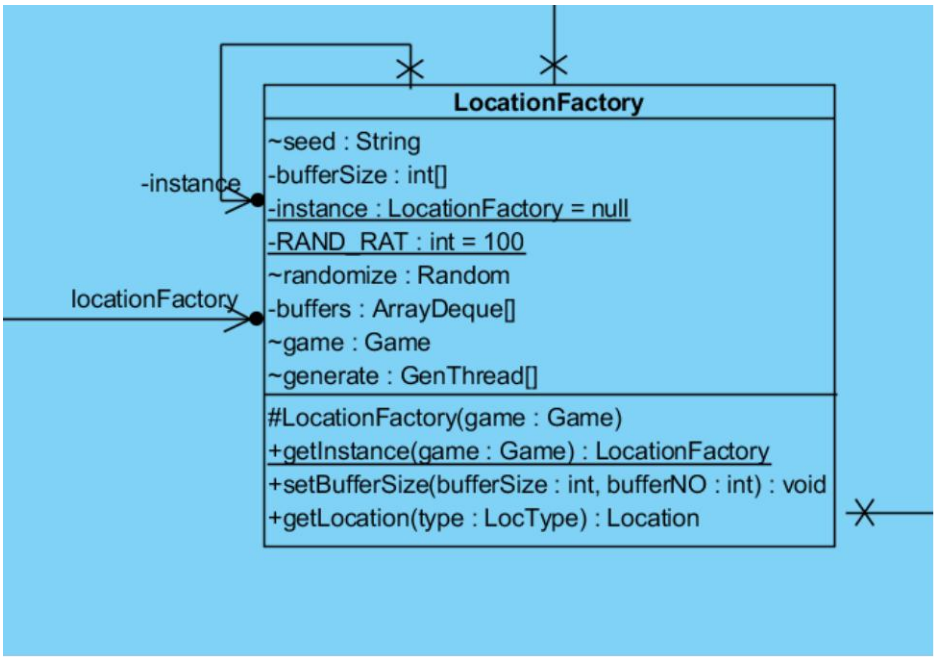
LocationFactory

LocationFactory class generates new locations in parallel threads and keeps a buffer of locations for the user wander.

Code for the pattern as follows

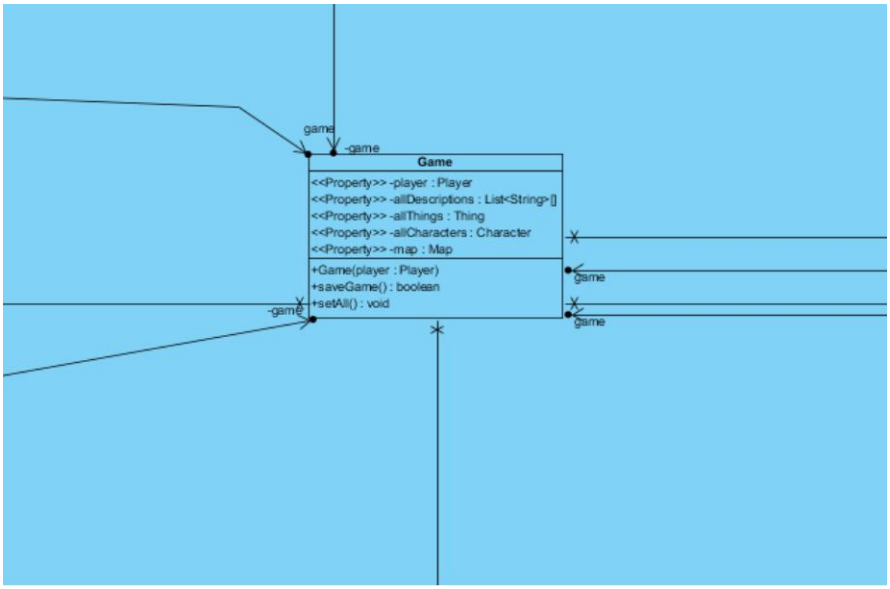
```
private static LocationFactory instance = null;

protected LocationFactory(Game game){
        this.game = game;
        this.seed = game.getPlayer().getSeed();
        bufferSize = new int[]{15,15,15,15};
        buffers = new ArrayDeque[]{
                new ArrayDeque<Location>(bufferSize[0]),
                new ArrayDeque<Location>(bufferSize[1]),
                new ArrayDeque<Location>(bufferSize[2]),
                new ArrayDeque<Location>(bufferSize[3])};
        randomize = new Random(Integer.parseInt(seed));
        generate = new GenThread[4];
```

```
    }
public static LocationFactory getInstance(Game game) {
    if(instance == null) {
        instance = new LocationFactory(game);
    }
    return instance;
}
```



6.1.2 Facade Pattern

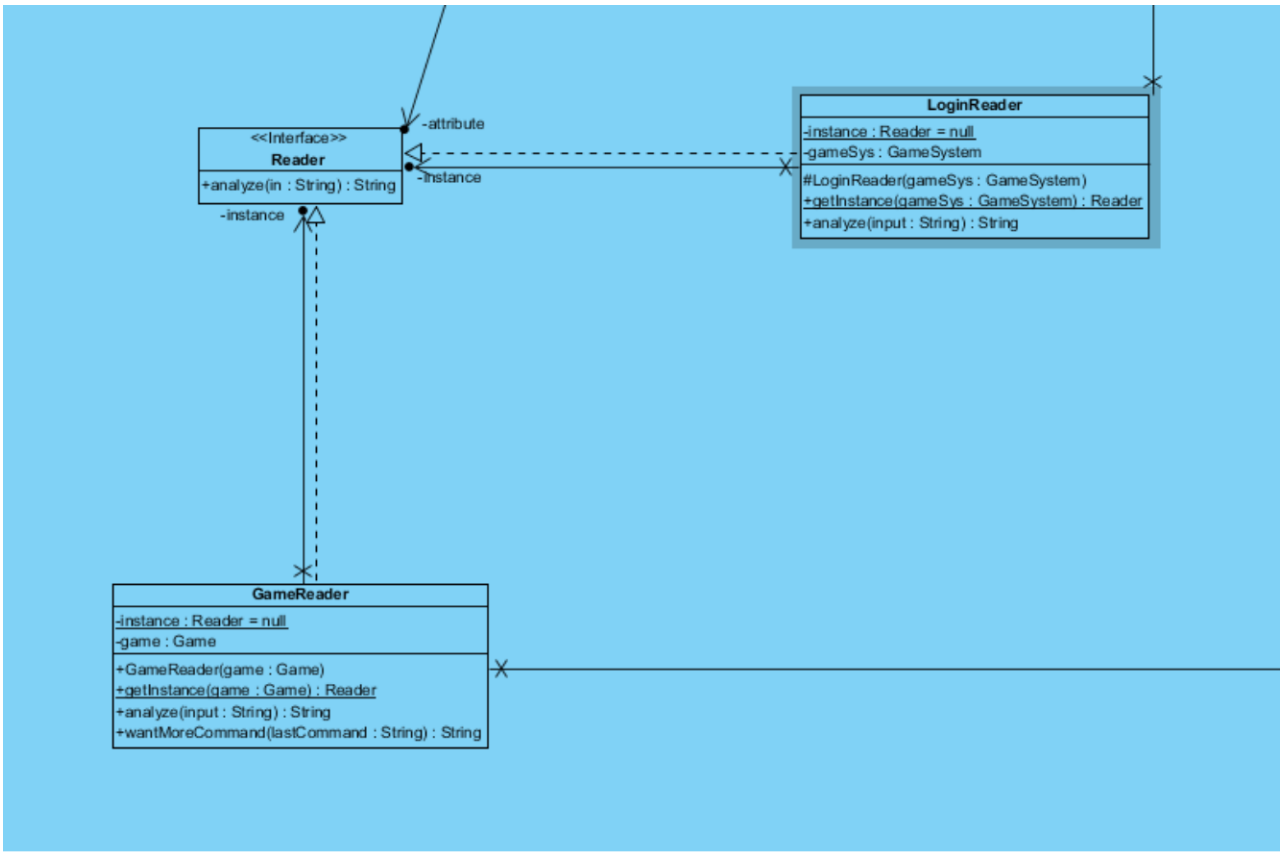
In our project there are lots of cross-class interactions. This is why we used facade pattern. In our project many classes use Game class to reach and use objects of other classes



6.1.3 Command Pattern

In our project we get two types of input from user. First for login-register and authentication then gameplay inputs. This is why we chose command pattern.

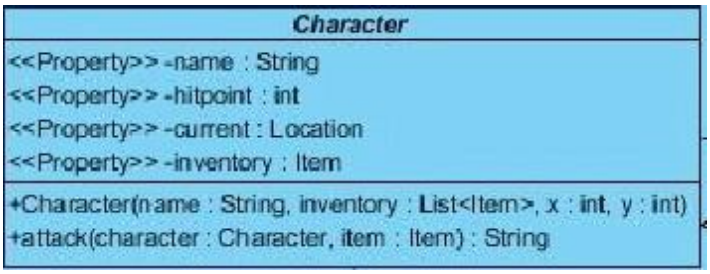
After user finishes login process GameSystem changes its Reader instance from LoginReader to GameReader and user can type commands to play.



6.2 Class Interfaces

a)game Package

1.**public** abstract class Character



Properties

private String name:
This is name of the Character

This property is **private** because it is only intended to access by setter and getter Methods provided

private Location current:
This is the current location of the Character

This property is **private** because it is only intended to access by setter and getter Methods provided

private List<Item> items:
This is the list of items that character has

This property is **private** because it is only intended to access by setter and getter Methods provided

private int hitpoint:
Remaining hit-points of the Character

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public Character(String name,List<Item> inventory,int x,int y); **@param** name String name to assign to character**@param** inventory List of Items Character has**@param** x Location of the Character
@param y Location of the Character

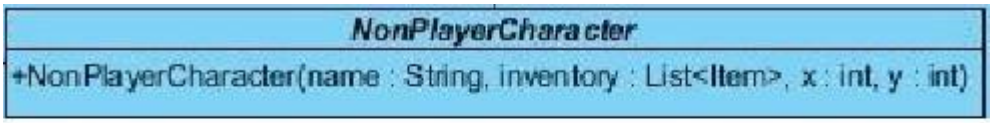
name and inventory will be initialized. current Location of the Character will be set according to x and y

Methods

public String attack(Character target,Item item);

Player attacks given target with the given item if it is possible, result given with an appropriate **String** message Damage will be calculated according to users stats, and equipment This method designed to be called from Player object inside Game object inside Parser object game.player.attack(character,item);

@param target to Attack



@param Item to use while attacking

public String getName();

Getter method for the property name

@return name as a String

public Location getLocation();

Getter method for the property Location

@return current Location of the Character

public List<Item> getItems()

Getter method for the property items

@return items as a List<Item> object

public int getHitPoints()

Getter method for the property hitpoint

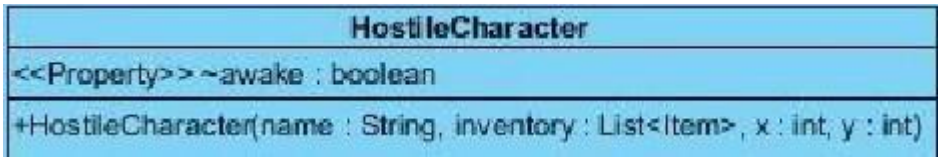
@return hitpoint Integer

2.**public** abstract class NonPlayerCharacter extends Character

Constructors

public NonPlayerCharacter(String name,List<Item> inventory,int x,int y); **@param** name String name to assign to character
@param inventory List of Items Character has**@param** x Location of the Character **@param**y Location of the Character

This will call the super Constructor with the given parameters
3.**public** class HostileCharacter extends NonPlayerCharacter



Properties

private boolean awake:

this boolean shows that hostile character is aware of the presence of the player, and it will attack to player with the each action player does.

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public NonPlayerCharacter(String name,List<Item> inventory,int x,int y); **@param** name String name to assign to character
@param inventory List of Items Character has**@param** x Location of the Character **@param**y Location of the Character

This will call the super Constructor with the given parametersMethods

Methods

public boolean getAwake()

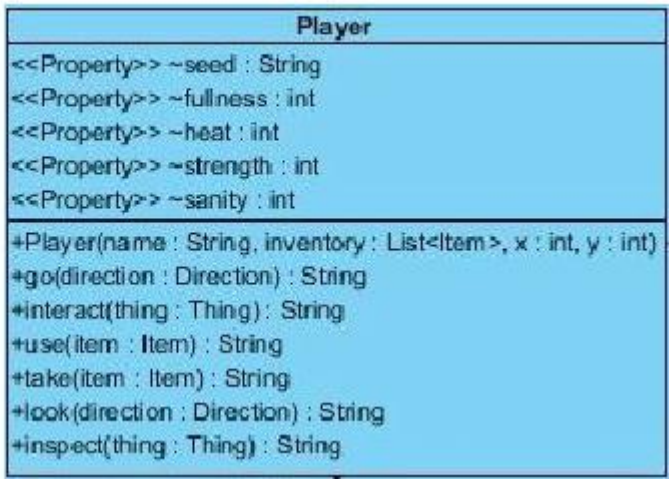
Getter method for the property awake

@return awake as a boolean

public void setAwake(boolean newAwake)

Setter method for the property awake

sets the awake boolean accordingly



4.**public** class Player extends Character

Properties

private String seed:

This is the seed of the Player, all randomness will be according to this value, this way players will be able to challenge other players on same generation, also they will be able to play to play on the same world again

This property is **private** because it is only intended to access by setter and getter Methods provided

private int fullness:

This is the hunger level of the Player, it is designed as fullness to make it parallel with other stats, this number will decrease with every action player does, when it hits zero player will starve

This property is **private** because it is only intended to access by setter and getter Methods provided

private int heat:

This is the heat level of the player, this will change according to surrounding temperature. Player is forced to find some heat source in order to survive nights

This property is **private** because it is only intended to access by setter and getter Methods provided

private int strength:

This is the strength stat of the player, this number will act as a multiplier when calculating the damage output of his/her atacks. Player's strength may increase according to tasks done, or decrease with hunger, etc.

This property is **private** because it is only intended to access by setter and getter Methods provided

private int sanity:

This is the sanity level of the player, It will decrease in dark (night,dungeons,etc.) and will increase if player has no hunger, heat problem during daytime. If this number goes too low, game will be over

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public Player(String name, List<Item> inventory, int x, int y); **@param** name String name to assign to Player**@param** inventory List of Items Player has
@param x Location of the Player**@param** y Location of the Player

This will call the super Constructor with the given parameters

Methods

public String go(Direction direction);**@param** direction to move

@return resulting String message

Player moves according to given direction if it is possible

public String interact(Thing thing);

@param thing to interact

@return resulting String message

Player interacts with given Thing if it is possible

public String use(Item item);

@param item to use

@return resulting String message

Player uses given Item if it is possible

public String take(Item item);

@param item to take

@return resulting String message

Player takes given Item if it is possible, if successful Item will be added to players inventory.

public String look(Direction direction);

@param direction to look

@return resulting String message

Player looks to given direction if it is possible

public String inspect(Thing thing);

@param thing to inspect

@return resulting String message

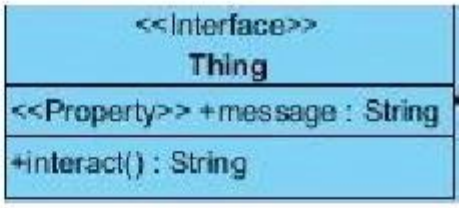
Player inspects given thing if it is possible,Reader will understand look Thing, as inspect Thing

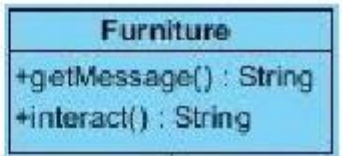
public String getSeed();**@return** seed as String

Getter method for the property seed **public** void setFullness(int fullness);
@param fullness to set
Setter method for the property fullness **public** int getFullness();
@return fullness as int

Getter method for the property fullness

public void setHeat(int heat);**@param** heat to set
Setter method for the property heat**public** int getHeat();
@return heat as int
Getter method for the property heat





public void setStrength(int strength);**@param** strength to set

Setter method for the property strength **public** int getStrength();

@return strength as int

Getter method for the property strength

public void setSanity(int sanity);**@param** sanity to set

Setter method for the property sanity**public** int getSanity();

@return sanity as int

Getter method for the property sanity

5.**public** interface Thing

Methods

public String getMessage();

@return String message to be displayed

This Methods returns the message to be send when interacted with the object itself

public String interact();

@return String message to be displayed

A method to interact with the Thing
6.**public** class Furniture implements Thing

Methods

public String getMessage();

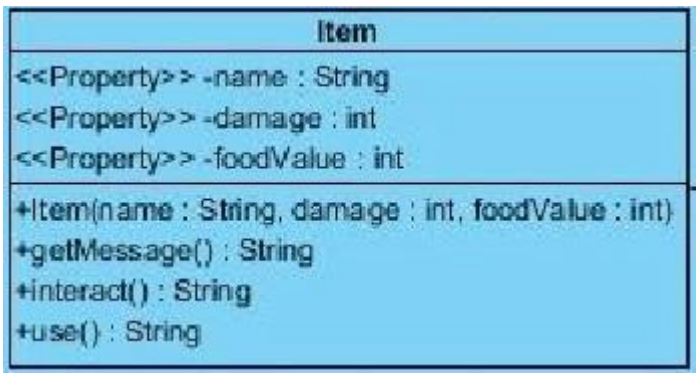
@return String message to be displayed

This Methods returns the message to be send when interacted with the object itself

Methods

public String interact();

@return String message to be displayed A method to interact with the Furniture



7.**public** class Item implements Thing

Properties

private String name:
This is name of the Item

This property is **private** because it is only intended to access by setter and getter Methods provided

private int damage:

If Item is a weapon this will get a positive value and it will be used to calculate output damage of the Character attacks. If it is zero this means it is not a weapon

This property is **private** because it is only intended to access by setter and getter Methods provided

private int foodValue:

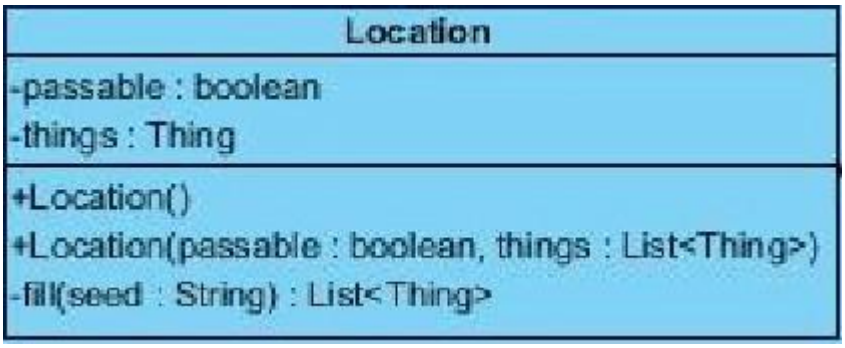
If Item is a food this will get a positive value and it will be edible and this food will increase the fullness of the Player by the value of it

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public Item(String name,int damage,int foodValue);

Methods



public String getMessage();

@return String message to be displayed

This Methods returns the message to be send when interacted with the object itself

Methods

public String interact();

@return String message to be displayed

A method to interact with the Furniture

public String use()

@return String message to be displayed

This method is called when Item is used

public String getName();**@return** name as String
Getter method for the property name

public int getDamage();**@return** damage as int
Getter method for the property damage

public int getFoodValue();**@return** foodValue as int
Getter method for the property foodValue

7.**public** class Location
Parameters

private boolean passable;

Indicates if Location is passable, can be changed as Boolean[3] passable,

to indicate directions from this location depending of the implementation

This property is **private** because it is only intended to access by setter and getterMethods provided

private List<Thing> things;

List of Thing's inside this location, will be randomly generated if location is new, while loading it will be generated according to given List

This property is **private** because it is only intended to access by setter and getterMethods provided

Constructors

public Location();

This is the default Constructor which is used to generate new locations randomly

public Location(boolean passable,List<Thing> things);

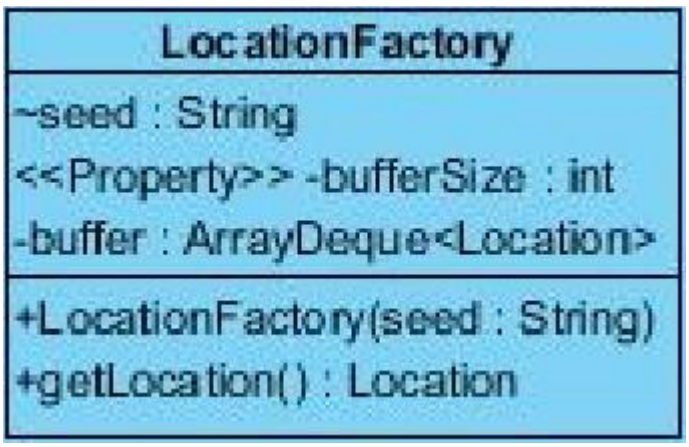
@param passable boolean that designates if location is passable

@param things is a list of Thing objects which exist on that location object
ThisConstructor is used when loading a saved game from a save file

Methods

private List<Thing> fill(String seed);
@param seed a string that manages the randomness**@return** list of Thing's that exists on that location

This method randomly fills a location according to a seed.
This private method called when object is constructed with default Constructor.
8.public class LocationFactory



private String seed;

A copy of the seed of the player to manage randomness

This property is **private** because it is only intended to access by setter and getter Methods provided

private int bufferSize;

A designated buffer size of buffer to manage the number of Locations to keep in the ArrayDeque

This property is **private** because it is only intended to access by setter and getter Methods provided

private ArrayDeque<Location> buffer;

An ArrayDeque that keeps generated Locations inside. It is used as a queue in this program

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public LocationFactory(String seed);

@param seed as a String to manage randomness

This will initialize the property seed with parameter seed

public void setBufferSize(int bufferSize);

@param bufferSize as a String to change buffer size according to needs

public Location getLocation();

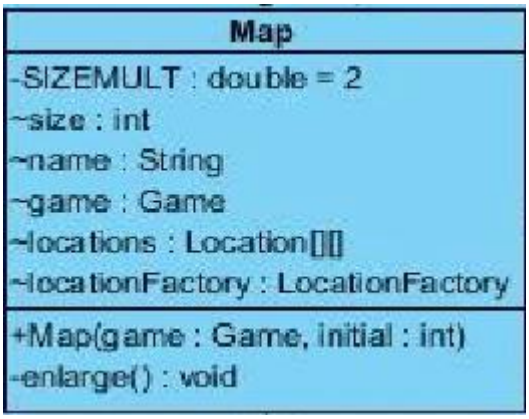
@return newLocation

This method poll a Location from arraydeque and return it.

public void addLocation();

This method will be called when system is idle and it will fill the arraydeque with newly generated Locations

8.**public** class Map



Parameters

private final double SIZEMULT = 2;

SIZEMULT is a **private** constant double that designates the size increase of the map when border is reached

private int size;

size is an **int** that shows the current size of the square map **private** String name;

This property is **private** because it is only intended to access by setter and getter Methods provided

Game game;
An instance of the game object to reach other objects in the program

private Location[][] locations;
A 2 dimensional array of Locations to represent the Map.

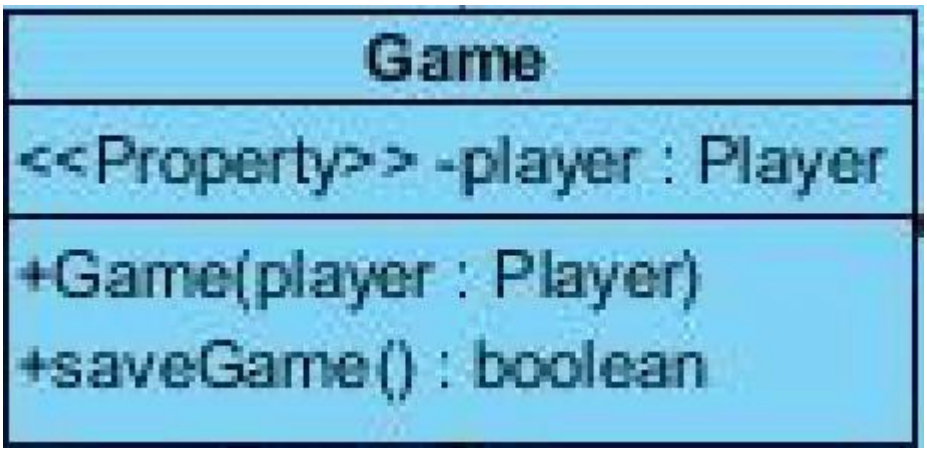
This property is **private** because it is only intended to access by setter and getter Methods provided
private LocationFactory locationFactory;

A LocationFactory to generate Locations when player move

Constructors

public Map(Game game,int initial);

@param game, game object passed from the caller to make it possible to reach other objects In the program



@param initial an integer value that shows the initial size of the square 2 dimensional Location array

Constructor sets the Properties with the parameters and initializes locations and LocationFactory

Methods

private void enlarge();

This method is called when player hits the borders of the existing Map, it will create a new Location[][] with the size = (size *SIZEMULT) and pass the existing locations to the middle of the new Location[][]

9.**public** enum Direction

Direction is enumeration that shows directions with the following constants in it

North,

East,

South,

West,

Upwards,

Downwards

10.**public** class Game

Parameters

private Player player;

An Instance of the player objects to be called from other classes that have the instance of the game object

for example game.getPlayer().go(Direction.North)

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public Game(Player player);

@param player is an instance of player from caller object

Constructor sets property player with parameter player

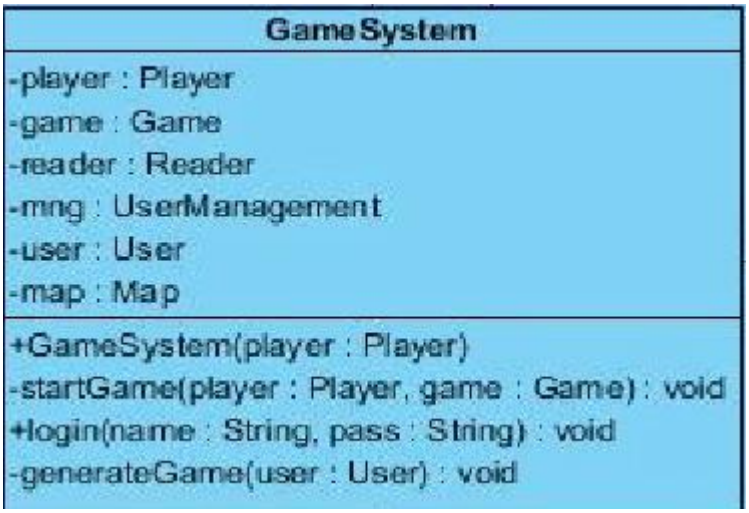
Methods

public Player getPlayer();

@return Player

returns the player object

11.**public** class GameSystem



Methods of

Properties

private Player player;

A player object that will be generated according to User logged in

This property is **private** because it is only intended to access by setter and getter Methods provided

private Game game;

A Game object that will be generated according to User logged in

This property is **private** because it is only intended to access by setter and getter Methods provided

private Reader reader;

A Reader object which will analyze the inputs of the User and call the the player object

This property is **private** because it is only intended to access by setter and getter Methods provided

private UserManagement mng;

An Object that manages the login authentications

This property is **private** because it is only intended to access by setter and getter Methods provided

private User user;

An object that represents the user logged in and keeps its data at runtime and saves it during logout

This property is **private** because it is only intended to access by setter and getter Methods provided

private Map map;

A Map object which player will be wandering during playtime

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public GameSystem();

A Constructor with no parameters will initialize the UserManagement object

Methods

private void startGame(Player player,Game game);

@param player a Player object that generated according to user data

@param game, a Game object that generated according to user data

A **private** method that will be called when user logs in and starts the game

public void login(String name, String pass);

@param name is a String that shows the login name of the user

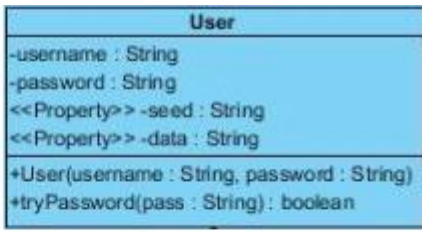
@param pass is a String that shows the login password of the user

This method will take username and password and call the authentication method of the UserManagement object. If authentication is successful it will call the generateGame method.

private void generateGame(User user);

@param user the user object that authenticated

This method will load the game according to User data, if data is empty it will randomly assign a seed and generate according to that value. After loading it will start the game.



b)datamanage Package

1.**public** class User

Properties

private String username;

A **String** that indicates the login username of the User

This property is **private** because it is only intended to access by setter and getter Methods provided

private String password;

A **String** that indicates the login password of the User. This can be encrypted in order to increase security

This property is **private** because it is only intended to access by setter and getter Methods provided

private String seed;

A String, seed value that will guide randomness

This property is **private** because it is only intended to access by setter and getter Methods provided

private String data;

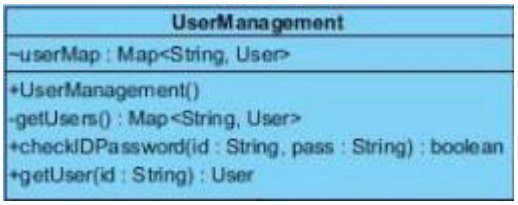
A String, data which will be loaded at the startup, and will be saved during exit

This property is **private** because it is only intended to access by setter and getter Methods provided

Constructors

public User(String username, String password);

@param username,that indicates the login username of the User



@param password,that indicates the login password of the User

This Constructor sets the parameter values to property values

Methods

public String getSeed();

@return seed as a String

This is the getter Method for Property seed

public String getData();
@ return data as a String
This is getter Method for Property data **public** boolean tryPassword(String pass)
@param pass a String that will be compared to password of the user **@return** boolean success of the trial
This method checks if password user entered is correct

2.**public** class UserManagement

private Map<String,User> userMap;

public UserManagement();

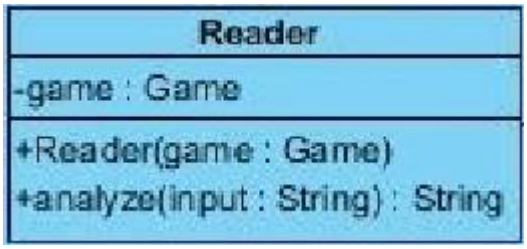
This Constructor takes no parameters and it fills userMap with the method getUsers()

private Map<String,User> getUsers();

@return Map<String,User> as a map of Users with names as their keys

This method reads a certain data file and fills the userMap accordingly

public boolean checkIDPassword(String id,String pass);



@param id a String that shows the id that will be tried

@param pass a String that shows the password that will be tried

@return boolean success

This method finds the user with given id and checks if pass is correct, and returns a boolean that shows the success of the authentication

public User getUser(String id);

@param id a String as the id of the User

@return User that found,null if User does not exists

This method looks for given id in the userMap and returns that user if it exists

c)IOManage package

1.**public** class Reader

Properties

private Game game;

A game object that will help Reader to reach other objects in the program

Constructors

public Reader(Game game)

@param game a simple Constructor to set property game as parameter game

Methods

public String analyze(String input)

@param input a string input that typed by the user

@return resulting String message to print on the screen

This Method takes a **String** input with format action receiver tool or action direction, etc. and execute the regarding Methods. After execution returns the resulting **String** to the caller

6.3 Specifying Contracts using OCL

context Player inv:

items->size <= MAX_NUM_ITEMS

//the maximum number of items must be less than or equal to the limit(MAX_NUM_ITEMS)

context Player::take(item) pre:

not item = NULL

//player can take item only if it is not null

context Player::take(item) pre:

player.inventory.size() < MAX_NUM_ITEMS

//player cannot have more items than MAX_NUM_ITEMS

context Player::take(item) post:

player.inventory.contains(item)

//after item is taken it should be included in player's inventory

context Player::take(item) post:

not player.getCurrent().things.contains(item)

//item should be disappear in its location after item is taken by player

context Player::drop(item) pre:

player.inventory.size() > 0

//player can drop its own items only if its inventory size is positive

context Map::enlarge() post:

size = SIZEMULT * @pre.size

//after map is enlarged new size of the map should be equal to the size multiplication factor times the current size

context Location :: setXY(x,y) post:

self.x = x

//once set method is called current x is set as x that comes from parameter of method

context Location :: setXY(x,y) post:

self.y = y

//once set method is called current y is set as y that comes from parameter of method

context Location :: getAdjacent(d) pre:

d->passable = true

//To get adjecent location in a direction this direction should be passable

context Location :: getAdjacent(d) post:

```
        x = @pre.x + 1
```

//getting adjacent location means enlarging map by adding 1 to x coordinate and,

```
context Location :: getAdjacent(d) post:
        y = @pre.y + 1
```

//adding 1 to y coordinate

```
context Item inv:
        getDamage() >= 0
```

// Damage of an item should be either positive or zero

```
context Item inv:
        getFoodValue() >= 0
```

// food value should either be positive or zero

```
context Item inv:
        not getName() = NULL
```

//names of items should be initialized so they cannot be null

```
context Item inv:
        not getMessage() = NULL
```

//meassages of items should be initialized so they cannot be null

```
context Character :: attack(c, i) pre:
        c-> hitPoint >= c->strength * i->damage
```

//to be attacked, the hitPoint of the character should be more than its strength times the damage of that item

context Character :: attack(c,i) post:

c-> hitPoint = @pre.(c->hitPoint - c->strength*i->damage)

//once a character is attacked its hitPoint decreases with the amount of its strength times damage of the item

context Character :: setStrength(s) pre:

s >= 0

// strength of a character should be a positive value or zero

context Character :: setStrength(s) post:

self.s = s

//once set methods is called for strength current strength is set as strength that comes from parameter of method

context Character :: setHitpoint(h) pre:

h >= 0

//hitPoint of a character must be positive value or zero

context Character :: setHitpoint(h) post:

self.h = h

//once set method is called for hitpoint current hitpoint is set as hitpoint that comes from parameter of the method

context Furniture inv:

not getName() = NULL

//name of a furniture should be initialized; it cannot be null

context Furniture inv:

not getMessage() = NULL

//name of a furniture should be initialized it cannot be null

context Game::setAll() pre:

allThings.isEmpty() = true

//to set things, list of things should be empty

context Game::setAll() pre:

allCharacters.isEmpty() = true

//to set characters, list of characters should be empty

context Game::setAll() pre:

allDescriptions.isEmpty() = true

//to set descriptions, list of descriptions should be empty

context Reader::analyze(input) pre:

not input = NULL

// to analyze an input, input should not be empty

context HostileCharacter :: setAwake(a) post:

self.a = a

//once set methods is called for awake (boolean), the return value of awake should be set as parameter

context GenThread :: getLocs() pre:

not listofLocks.isEmpty()

//getLocs method can be called only if listofLocks is not empty

context Usermanagement :: checkIDPassword(id, pass) pre:

not id=NULL

//to check id it should not be null

context Usermanagement :: checkIDPassword(id, pass) pre:

not pass=NULL

//to password it should not be null

7 Conclusion

This project aimed to realize an object-oriented approach towards text-based computer games, using several architectural and object design patterns, with UML as the primary language to visualize this approach. To make a true analysis of the system so that it is easily maintained and functions correctly and consistently, several models containing use cases were developed. Then, the system was decomposed to a number of subsystems, where each subsystem would maintain a unique architectural style. After the stage of object design, finally the portion of implementing the project using Java was completed.

There were many challenges faced when developing this project. Object design patterns were difficult to apply; finding the right pattern for each subsystem required a good knowledge of object design and a detailed visualization of the software to be developed. Working as a team and putting time and effort into research, we were able to manage each subsystem and apply the most appropriate patterns. Similarly, architectural patterns were not easy to implement, as it required outside-the-box thinking to fully comprehend the implications of each pattern. Again, a good deal of research made it possible to select and apply only one pattern that we deemed fit. Other obstacles such as time constraint and technical issues were always present, as is in all software development projects, and only pushed us even further to accomplish our goals.

The true triumph of building object-oriented software is to have accomplished a goal that applies to a computer, using real world solutions to solve real world problems. We believe that with the knowledge that we have gained out of this project, we are ready to explore further challenges to help solve real world problems through a fundamental understanding of the core concepts of object-oriented software engineering. As far as future work goes, it is almost a certainty that the principles that we have acquired in this project will be of guidance in many of our personal or corporate projects.

References

[1] http://en.wikipedia.org/wiki/Zork.

[2] http://en.wikipedia.org/wiki/Text-based_game

Appendix