BILKENT UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

CS 319 – OBJECT ORIENTED SOFTWARE ENGINEERING

SECTION 03 – GROUP 3D

PROJECT TITLE: CASTLE DEFENSE

DESIGN REPORT
DRAFT

Team Members

Xhoana ALIU 21500429
Muhamed KETA 21503560
Onur KOCAHAN 21402013
Mehmet Can ALTUNTAS 21501883

1. Introduction

1.1 Purpose of the system

Castle Defense is a tower defense game, player tries to kill the creeps in waves with the tower s/he bought. The aim of the game is clearing all the waves with towers and consumable power-ups. Castle Defense has user friendly and complex, but easy to learn, game dynamics. Castle defense offers challenging gameplay after each wave cleared. Different type of towers, consumable power-ups and auto-generated map offer variety of strategy to players. Those features make Castle Defense the best option for tower defense players.

1.2 Design goals

Our design aim is offering different experience of tower defense game. Offering challenging gameplay after each wave cleared, while entertaining the user.

End User Criteria

Ease of Learning: At the beginning of the first game of the user, instructions about how to play Castle Defense will be shown. After the instructions, first waves are designed to be teach player how this game works. After each wave cleared, next wave will be harder little bit so that the user gets the dynamics of the game better.

Ease of use: Designing user-friendly game is crucial for game designers. Everything in the Castle Defense is designed for comfort of the user. The menu will be simple to understand and controls of the game will be simple. Mouse will be used in the game for whatever user wants.

Maintenance Criteria:

Extensibility: It is important to updating the game and changing the game, like adding new features, in order to attaining user's attention. One of our design goal is to designing maintainable tower defense game.

Reliability: Castle Defense offers users bug-free and crash-free gameplay experience. Crashing the game annoys most of the users, which leads to distract attention of the users.

Modifiability: Since the creeps and the towers will be implemented in object oriented base. In order to having balanced game, the system will be updated, which requires low-cost because of the object oriented based implementation, which results in attracting attention of users.

Performance Criteria:

Fluent Gameplay: In tower defense games having fluent gameplay is important for the users. Movement of the creeps and the projectile of the towers will be designed for offering users smooth gameplay. With each update of the game, the gameplay will be enhanced for better game experience.

Response Time: Since the game has fluent gameplay, there will be no lag during the game, which enhances gameplay and prevents user from distracting attention.

1.3 Trade-Offs

Usability vs. Functionality: The functionality makes the game complex and leads to distracting users' attention. On the other hand, having simple graphical interface and simple menus does not distract users' attention and make user easy to focus the game. Rather than complex menus, we will design Castle Defense, easy to understand for helping user to keep their focus.

Performance vs. Memory: In order to offering users fluent gameplay and better response time the graphics of the creeps and towers will not have high quality graphics. Whereas, lag and delay distract users' attention, low quality graphics also distracts users' attention. Our design will be balanced but in our design performance will outweigh memory, which means we will focus more performance rather than high quality graphics.

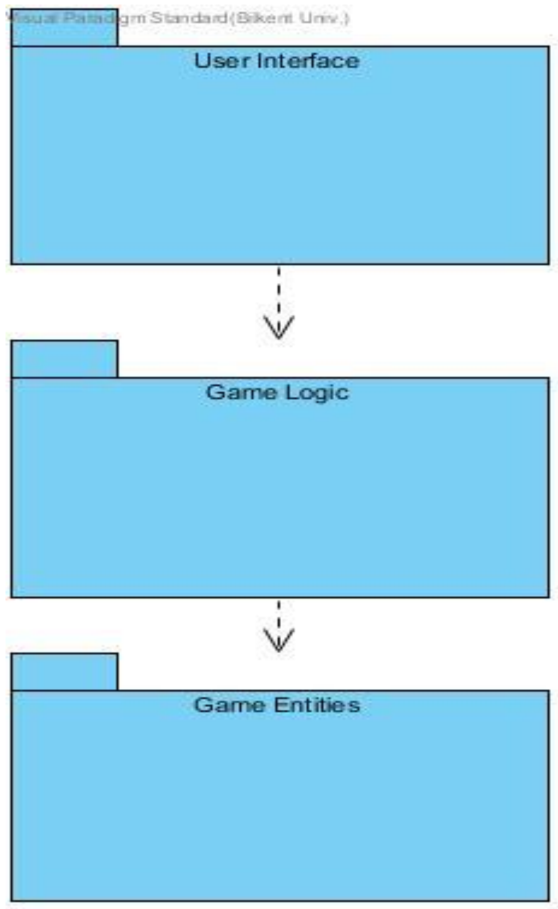
2. Software Architecture

2.1 Subsystem Decomposition & Architectural Style

We will decompose our system in such a way so it will apply Model View Controller architectural style. While we decompose our system into subsystems we try to reduce the coupling between the subsystems and also increase cohesion of the components of the subsystem.

Below the subsystem decomposition of our game system is shown. We have chosen the three – tire subsystem decomposition. We have decomposed our system into three subsystems; User Interface, Game Logic and Game Entities. The system is separated this way to favor the functionality and purpose of each subsystem.

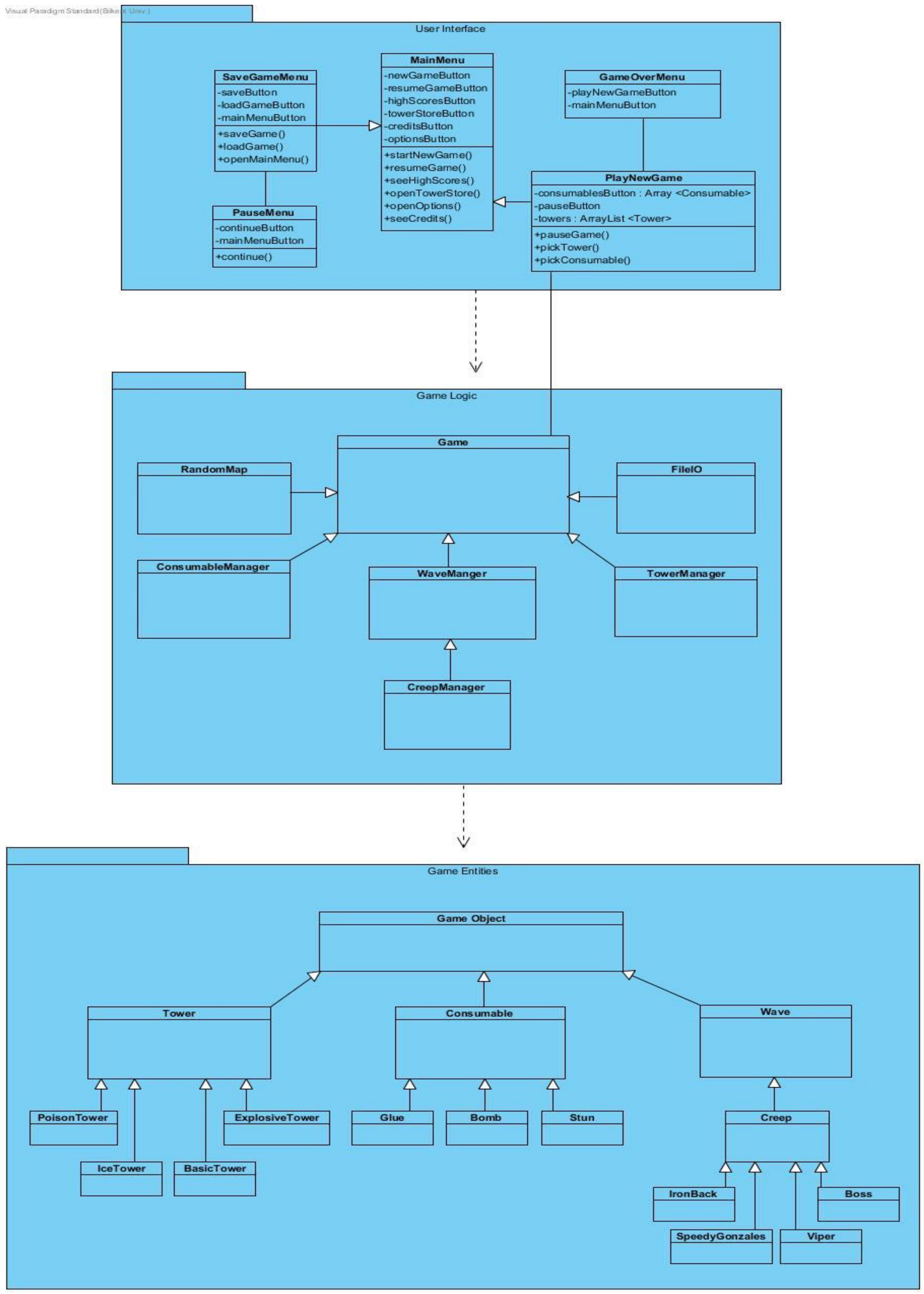
All three subsystems will be working together but they will not be affected that much by changes or errors on the other subsystems. As the system has three layers: layer 1 layer 2 and layer 3 each corresponding respectively to User Interface, Game Logic and Game Entities, layer 1 will be connected to layer 2 and layer 2 will be connected to layer 3. Any change in layer 1 can only affect layer 2 and any change in layer 2 can affect layer 3.



Subsystem Decomposition Diagram

Our subsystems are hierarchical because they have dependency on each other. Layer 1 is on the top since User Interface subsystem does not depend on any of the other subsystems. User Interface subsystem will be our View. Then it comes layer 2, which is dependent on User Interface. The Game Management subsystem will be our Control. Lastly, layer 3, Game Entities will be our Model. User Interface will communicate with Game Entities only via Game Management subsystem. Thus, the changes made on the User Interface will not affect Game Entities subsystem. In this way our system decomposition will be compatible with the Model View Control architectural style.

As it can be seen in the more detailed subsystem decomposition diagram, User Interface subsystem will contain the “Menu”, which will offer the user to choose between different options such as “Play New Game” and “Resume”. Based on the player’s choice, Game Management will handle the construction of the game using “Random Map” and “Game” classes. This package will be connected to Game Entities subsystem, which contains the objects such as towers, creeps and consumables. These elements will be manipulated by the Game Management subsystem.



More Detailed Subsystem Diagram

All in all, we designed our system such that it will provide loose coupling and cohesion in case any changes are done or any errors occur in any of the subsystems. Moreover, the MVC architectural style is applied.

2.2 Hardware/Software Mapping

The programming language that we will develop “Castle Defense” will be Java therefore it will require Java Runtime Environment. Regarding the hardware configuration for being able to play the game there will be needed just a mouse and the keyboard. The mouse will be used in order to put the towers on the desired place on the map of the game, selecting and using the game features (selling and buying towers, upgrading towers) and the keyboard will be used for inserting the name of the player in order to be used while displaying the high scores list. Therefore, the hardware requirements for our game system will be minimal. Regarding software systems, they will also be minimal since our game will only need a computer that has already a java compiler in order to be able for the game java file to be executed and run. Moreover, this will be an offline game and an internet connection is not required.

2.3 Persistent Data Management

“Castle Defense” will have randomly generated maps that will change every time the player starts a new game. However, since the player will be able to pause and resume the game whenever s/he wants, the map structure (meaning the parameters) of the game that will be left unfinished will be stored as text file in hard disk drive. Moreover, the high scores table of the game will also be saved into a text file. Since we are planning to add some sound effects and object animations, these too will be stored in hard disk drive with proper formats for images and sounds. All in all, all the game data will be stored in the hard disk drive and will be accessed whenever they are needed by the game system.

2.4 Access Control and Security

Since our game will not require network connection of any kind, there will not be any restrictions regarding the accessibility of the game. The game will be easily accessible for anyone who has the .java file of the game and will run it on her/his computer. Additionally, “Castle Defense” will not require a user profile and therefore there will not be any security issues regarding the game.

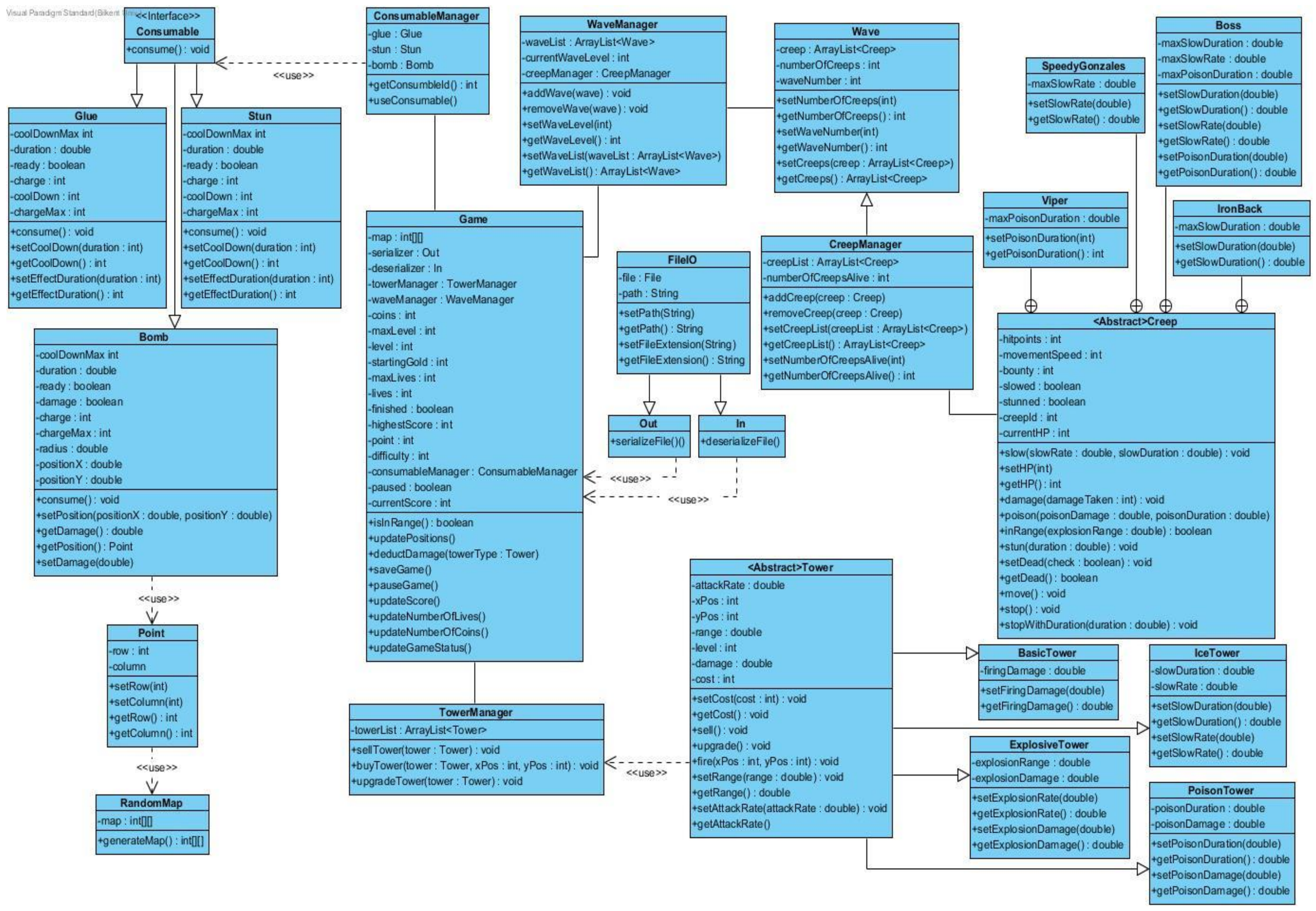
2.5 Boundary Conditions

“Castle Defense” can be initialized by running the executable jar file of the game. When the player finishes the game (wins or loses), s/he will see a popup message on the screen depicting the points, coins and whether s/he set a new record. After the popup message the user can choose to play a new game or s/he can return to the menu and exit the program. The player can terminate the program also by clicking on the Exit button (X) at the upper right side of the window. If the player exits before the game has ended the game is automatically saved and the next time the player opens the game s/he can resume it and continue where s/he left. Moreover, if the game did not respond or crashed because of any performance issues, the player may lose all of her/his current data. Additionally, if the user tries to open the game while it is already running, the program will crash.

3. Subsystem Services

3.1 Detailed Object Design

The class diagram supplied below contains detailed information about the mechanism of the game and provides a better understanding of the game's logic. The Game is a state of the current game and will be serialized with the Out class if user decides to save the game. Manager classes ensure the functionality of the game. Creeps and waves are handled by the game in the background, meanwhile other managers are there to ensure the user can use the game mechanics properly. Besides this class diagram that captures both Game Logic and Game Entities there are also Menu classes of User Interface subsystem. The classes will be described in a more detailed way below.


Visual Paradigm Standard(Bikent
<<Interface>>
Consumable
+consume() : void
<<use>>
ConsumableManager
-glue : Glue
-stun : Stun
-bomb : Bomb
+getConsumbleId() : int
+useConsumable()
Glue
-coolDownMax int
-duration : double
-ready : boolean
-charge : int
-coolDown : int
-chargeMax : int
+consume() : void
+setCoolDown(duration : int)
+getCoolDown() : int
+setEffectDuration(duration : int)
+getEffectDuration() : int
Stun
-coolDownMax int
-duration : double
-ready : boolean
-charge : int
-coolDown : int
-chargeMax : int
+consume() : void
+setCoolDown(duration : int)
+getCoolDown() : int
+setEffectDuration(duration : int)
+getEffectDuration() : int
Bomb
-coolDownMax int
-duration : double
-ready : boolean
-damage : boolean
-charge : int
-chargeMax : int
-radius : double
-positionX : double
-positionY : double
+consume() : void
+setPosition(positionX : double, positionY : double)
+getDamage() : double
+getPosition() : Point
+setDamage(double)
<<use>>
Point
-row : int
-column
+setRow(int)
+setColumn(int)
+getRow() : int
+getColumn() : int
<<use>>
RandomMap
-map : int[][]
+generateMap() : int[][]
WaveManager
-waveList : ArrayList<Wave>
-currentWaveLevel : int
-creepManager : CreepManager
+addWave(wave) : void
+removeWave(wave) : void
+setWaveLevel(int)
+getWaveLevel() : int
+setWaveList(waveList : ArrayList<Wave>)
+getWaveList() : ArrayList<Wave>
Wave
-creep : ArrayList<Creep>
-numberOfCreeps : int
-waveNumber : int
+setNumberOfCreeps(int)
+getNumberOfCreeps() : int
+setWaveNumber(int)
+getWaveNumber() : int
+setCreeps(creep : ArrayList<Creep>)
+getCreeps() : ArrayList<Creep>
SpeedyGonzales
-maxSlowRate : double
+setSlowRate(double)
+getSlowRate() : double
Boss
-maxSlowDuration : double
-maxSlowRate : double
-maxPoisonDuration : double
+setSlowDuration(double)
+getSlowDuration() : double
+setSlowRate(double)
+getSlowRate() : double
+setPoisonDuration(double)
+getPoisonDuration() : double
Game
-map : int[][]
-serializer : Out
-deserializer : In
-towerManager : TowerManager
-waveManager : WaveManager
-coins : int
-maxLevel : int
-level : int
-startingGold : int
-maxLives : int
-lives : int
-finished : boolean
-highestScore : int
-point : int
-difficulty : int
-consumableManager : ConsumableManager
-paused : boolean
-currentScore : int
+isInRange() : boolean
+updatePositions()
+deductDamage(towerType : Tower)
+saveGame()
+pauseGame()
+updateScore()
+updateNumberOfLives()
+updateNumberOfCoins()
+updateGameStatus()
FileIO
-file : File
-path : String
+setPath(String)
+getPath() : String
+setFileExtension(String)
+getFileExtension() : String
Out
+serializeFile()()
In
+deserializeFile()
<<use>>
<<use>>
CreepManager
-creepList : ArrayList<Creep>
-numberOfCreepsAlive : int
+addCreep(creep : Creep)
+removeCreep(creep : Creep)
+setCreepList(creepList : ArrayList<Creep>)
+getCreepList() : ArrayList<Creep>
+setNumberOfCreepsAlive(int)
+getNumberOfCreepsAlive() : int
Viper
-maxPoisonDuration : double
+setPoisonDuration(int)
+getPoisonDuration() : int
IronBack
-maxSlowDuration : double
+setSlowDuration(double)
+getSlowDuration() : double
<Abstract>Creep
-hitpoints : int
-movementSpeed : int
-bounty : int
-slowed : boolean
-stunned : boolean
-creepId : int
-currentHP : int
+slow(slowRate : double, slowDuration : double) : void
+setHP(int)
+getHP() : int
+damage(damageTaken : int) : void
+poison(poisonDamage : double, poisonDuration : double)
+inRange(explosionRange : double) : boolean
+stun(duration : double) : void
+setDead(check : boolean) : void
+getDead() : boolean
+move() : void
+stop() : void
+stopWithDuration(duration : double) : void
<Abstract>Tower
-attackRate : double
-xPos : int
-yPos : int
-range : double
-level : int
-damage : double
-cost : int
+setCost(cost : int) : void
+getCost() : void
+sell() : void
+upgrade() : void
+fire(xPos : int, yPos : int) : void
+setRange(range : double) : void
+getRange() : double
+setAttackRate(attackRate : double) : void
+getAttackRate()
TowerManager
-towerList : ArrayList<Tower>
+sellTower(tower : Tower) : void
+buyTower(tower : Tower, xPos : int, yPos : int) : void
+upgradeTower(tower : Tower) : void
<<use>>
BasicTower
-firingDamage : double
+setFiringDamage(double)
+getFiringDamage() : double
IceTower
-slowDuration : double
-slowRate : double
+setSlowDuration(double)
+getSlowDuration() : double
+setSlowRate(double)
+getSlowRate() : double
ExplosiveTower
-explosionRange : double
-explosionDamage : double
+setExplosionRate(double)
+getExplosionRate() : double
+setExplosionDamage(double)
+getExplosionDamage() : double
PoisonTower
-poisonDuration : double
-poisonDamage : double
+setPoisonDuration(double)
+getPoisonDuration() : double
+setPoisonDamage(double)
+getPoisonDamage() : double

3.2 User Interface Management Subsystem

User Interface Management Subsystem will be composed of “MainMenu”, which is a view that will enable the player choose between different options. The player will be offered the following options: Play New Game, Resume Game, High Scores, Options, Tower Store and Credits. The most important classes will be PlayNewGame and SaveGameMenu classes which will take care of making possible to interact with the Controller of the system or Game Logic subsystem.

SaveGameMenu

This class is concerned with saving the game and also loading a previously saved game when the player chooses to resume a certain game. This view class will offer the user the following 3 options: Go to Main Menu, Save Game, Load Game.

PauseMenu

This class is a view class which is concerned with the pause action by the user. If the user chooses to pause the game, then the user will be offered with 2 options: Continue the Game or Go to Main Menu.

GameOverMenu

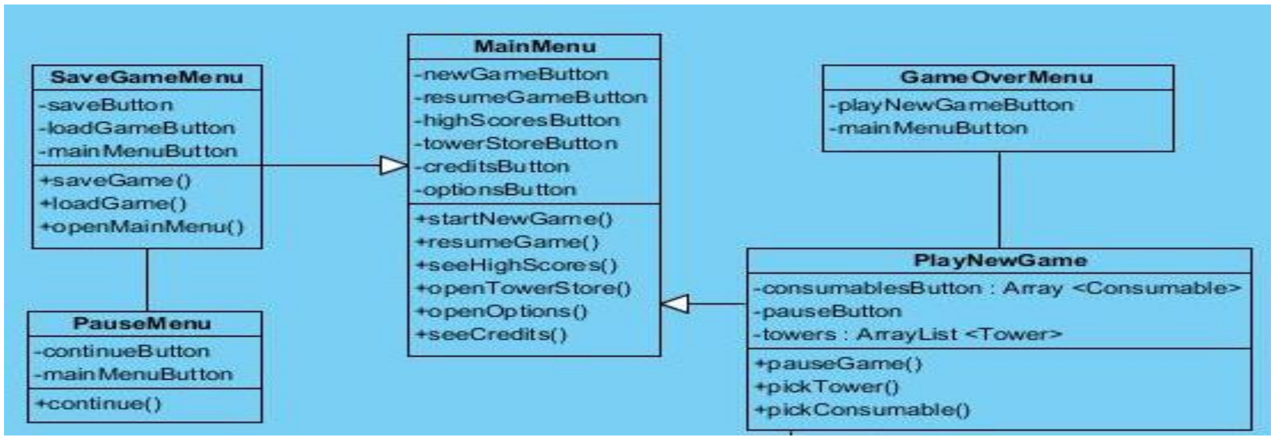
This class is concerned with the final state of the game. It will display the result of the game and will offer the player to choose between playing a new game or return to main menu.

PlayNewGame

This class will serve as the point of connection of User Interface subsystem with Game Logic. This class makes the interaction between these two subsystems. It will have pauseGame() method which will pause the game. pickTower() method will allow the user choose between different types of towers while pickConsumable() will allow the user to choose between different types of Consumables in order to use them during the game.

MainMenu

This class will contain 6 buttons offering the player different options in order to interact with the game. It will consist of the first view of the game screen.



3.3 Game Logic Subsystem

Classes that handle game’s logic are responsible for managing the user interactions such as adding a tower, saving/reloading a previous game and using consumables. It also handles incoming enemy creep waves and decides whether they are hit or not.

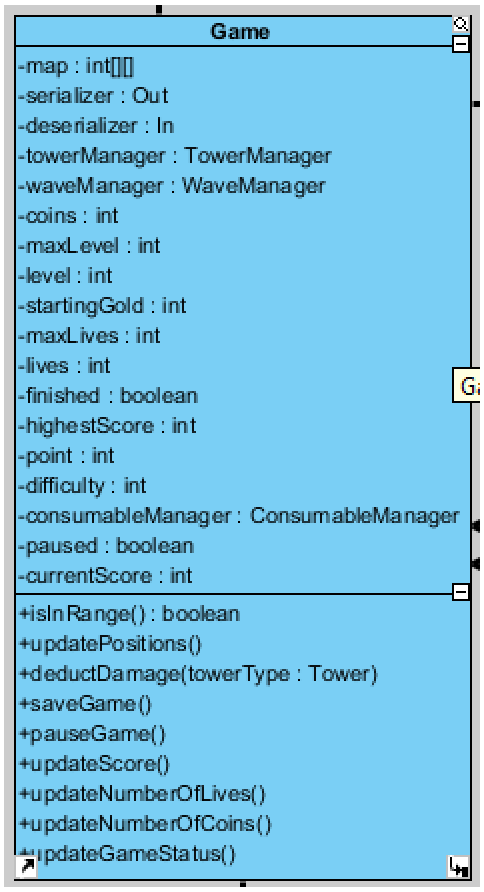
Game Class

Game class stores the state of the game; it stores every tower on the map, it handles every consumable user wants to use and generates creep waves. It also stores data such as how many coins user has and which level he/she is in. Therefore, Game class is the main component behind the games logic.

There is a Game instance running inside the game’s main loop. It tracks position of the creeps in and tries to find if one of them is in range in real time. It also checks for damage deducted by creeps, damage taken by creeps and whether a creep is dead or not. At the end of every cycle, it tries to determine if the game is over or not.

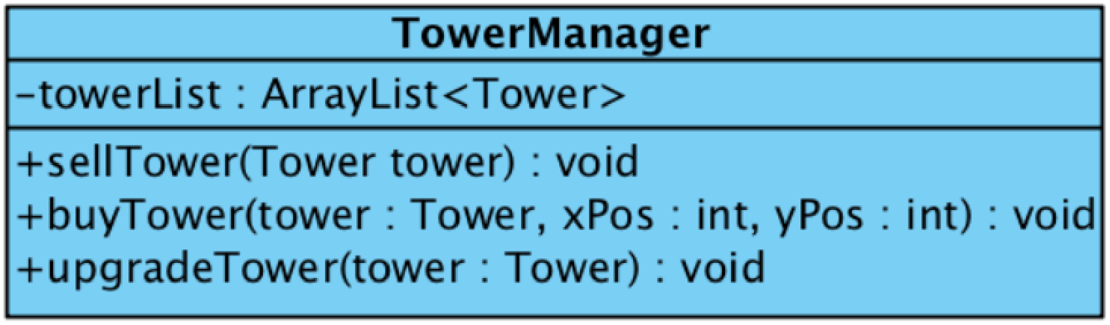
Since the Game class is a Java object, it is serializable; and if the user wants to save the game, Game class is serialized and can be deserialized for future use.

Notice that Game class acts as a buffer between control and model classes.



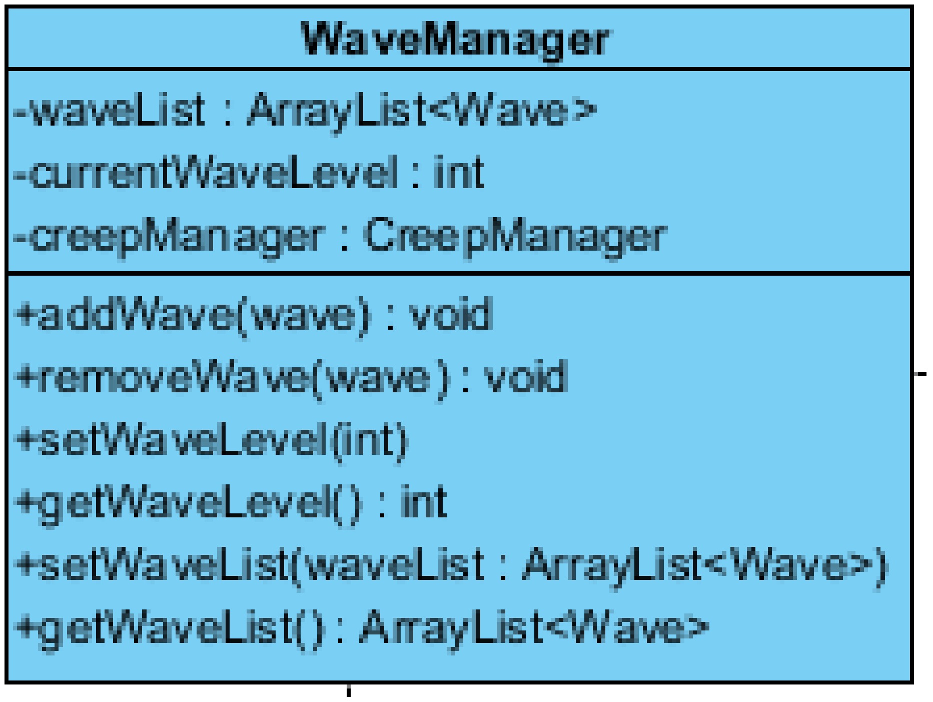
TowerManager Class

Tower Manager class is responsible for adding new towers or managing the existing ones. The user is going to be able to buy new towers, sell the existing ones and upgrade them. Towers on the map will be tracked with an ArrayList consisting of Towers, which stores the position and the type of the tower. It also manages the process of getting new towers, or selling them so that the coins gained by them can be used for a new strategy. The TowerManager class also consists of upgradeTower method; towers can be upgraded thrice and that method is used for that functionality.


TowerManager
−towerList : ArrayList<Tower>
+sellTower(Tower tower) : void
+buyTower(tower : Tower, xPos : int, yPos : int) : void
+upgradeTower(tower : Tower) : void

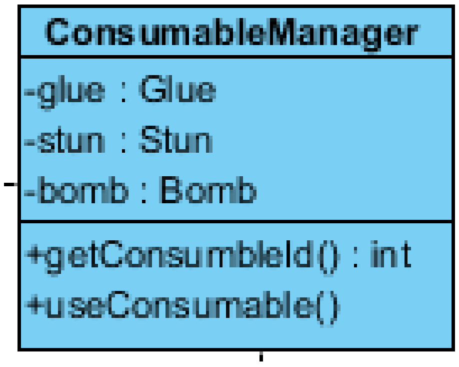
WaveManager Class

Wave Manager is responsible for generating new creep waves. Waves will consist creeps with different kinds and abilities. The internal CreepMenager class handles creeps that would be in the next wave. Difficulty selected by the user will affect the size of the wave and the types of the creeps. Waves are being recorded in a list. Current level of the wave will be recorded and will be used to determine how difficult the next level would be.


WaveManager
-waveList : ArrayList<Wave>
-currentWaveLevel : int
-creepManager : CreepManager
+addWave(wave) : void
+removeWave(wave) : void
+setWaveLevel(int)
+getWaveLevel() : int
+setWaveList(waveList : ArrayList<Wave>)
+getWaveList() : ArrayList<Wave>

ConsumableManager Class

Consumable Manager class handles predefined consumables and their usage. It tracks consumables that are being used by the user.



3.4 Game Entities Subsystem

Tower

This is the base class for towers. This manages the stats of each tower, such as the damage they deal, the speed they attack and the type of damage they do.
From this we extend the other types of towers.

Basic Tower

The tower with the balanced stats and normal single-creep-hitting projectiles. It has a fixed fireDamage which will be the damage it will do to the creeps.

Ice Tower

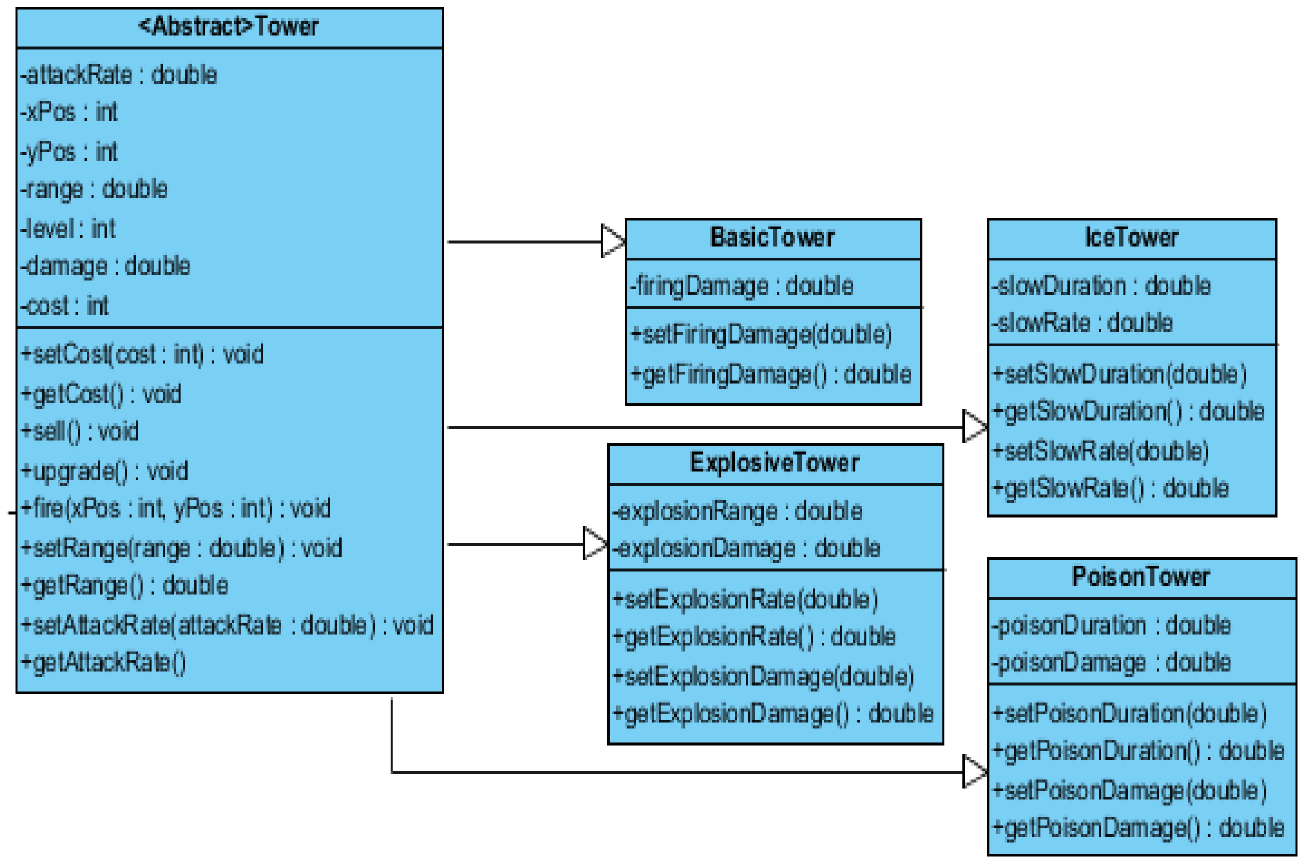
The tower that hits single creeps and slows them down. This kind of tower has different effects on different enemies. It can be an advantage when paired up with other towers that have higher damage but lower rate of fire.

Explosive Tower

The tower has a lower attack statistic, but hits multiple enemies. This can be an advantage when there are a lot of lower HP enemies.

Poison Tower

The tower that hits creeps and gives them a poison effect. This kind of tower gives different effects on different creeps. The poison effect damages creeps over time, with possible side effects.



Consumables

These are one time use buyable items, buyable at any point of the level.

Stun Consumable

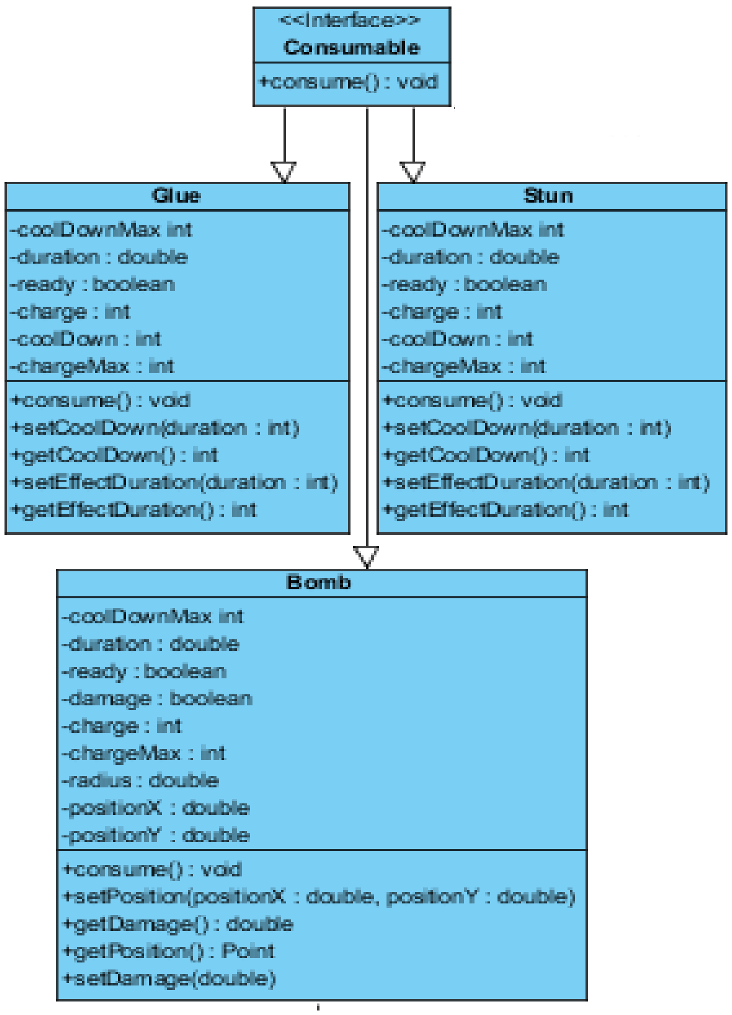
Consumable that stops the movement of creeps for a certain time period. It will have a certain duration of its effect.

Glue Consumable

Consumable that will slow the creeps down. It will have a certain duration of its effect.

Bomb Consumable

Consumable that damages all creeps in a certain area. Depending on the coordinates that the user chooses to consume the Bomb Consumable, it will calculate the damage it will do te the creeps in its range.



Creep

This is the main class that is used to extend the type of creeps. It holds the amount of HP they have and the type of stats they can differentiate with each-other.

Viper

This is the balanced type of creep, with an adequate hit point number that goes away with the speed they have.

IronBack

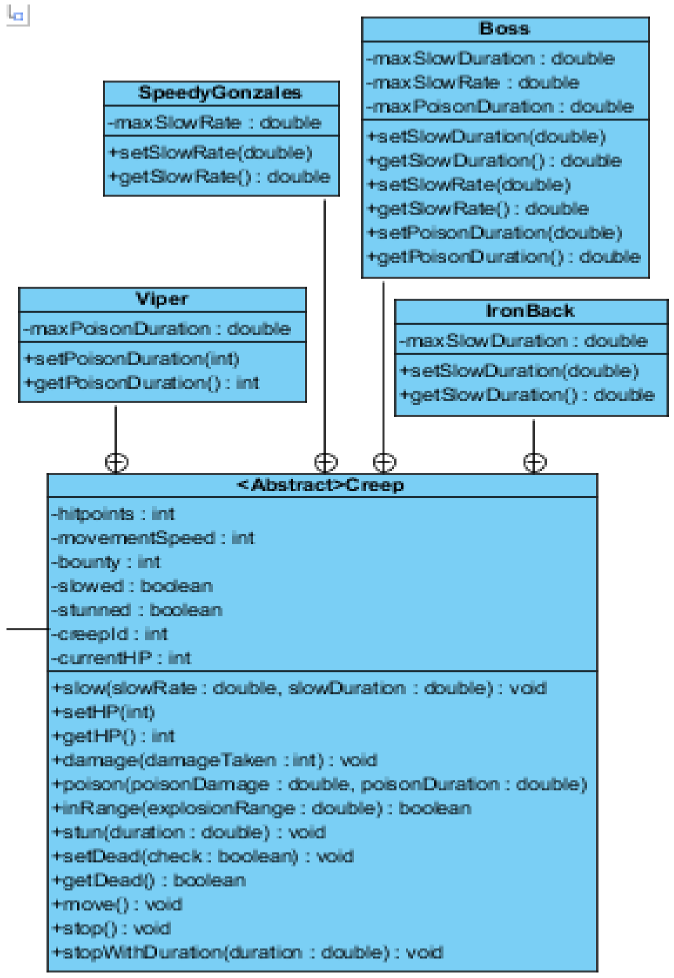
This is the slower type of creep who has a greater number of hitpoint from the Viper type.

SpeedyGonzales

This is the special type of creep who has a greater speed, but has slightly less hitpoint from the Viper type.

Boss

This is the type of creep, signified from the size of the creep. It has more hitpoints and has a slight lesser speed from the Viper type. It is only in the last creep of the wave.

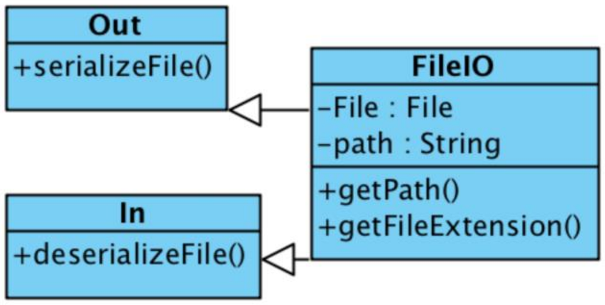

Boss
-maxSlowDuration : double
-maxSlowRate : double
-maxPoisonDuration : double
+setSlowDuration(double)
+getSlowDuration() : double
+setSlowRate(double)
+getSlowRate() : double
+setPoisonDuration(double)
+getPoisonDuration() : double
SpeedyGonzales
-maxSlowRate : double
+setSlowRate(double)
+getSlowRate() : double
Viper
-maxPoisonDuration : double
+setPoisonDuration(int)
+getPoisonDuration() : int
IronBack
-maxSlowDuration : double
+setSlowDuration(double)
+getSlowDuration() : double
<Abstract>Creep
-hitpoints : int
-movementSpeed : int
-bounty : int
-slowed : boolean
-stunned : boolean
-creepId : int
-currentHP : int
+slow(slowRate : double, slowDuration : double) : void
+setHP(int)
+getHP() : int
+damage(damageTaken : int) : void
+poison(poisonDamage : double, poisonDuration : double)
+inRange(explosionRange : double) : boolean
+stun(duration : double) : void
+setDead(check : boolean) : void
+getDead() : boolean
+move() : void
+stop() : void
+stopWithDuration(duration : double) : void

3.5 File Management

File Management

File management subsystem consists of 3 parts:

- **FileIO:** an interface for basic file handling operations; such as opening a file, getting the path, getting the extension of the found file.
- **In:** a class that handles deserialization.
- **Out:** a class that handles serialization.

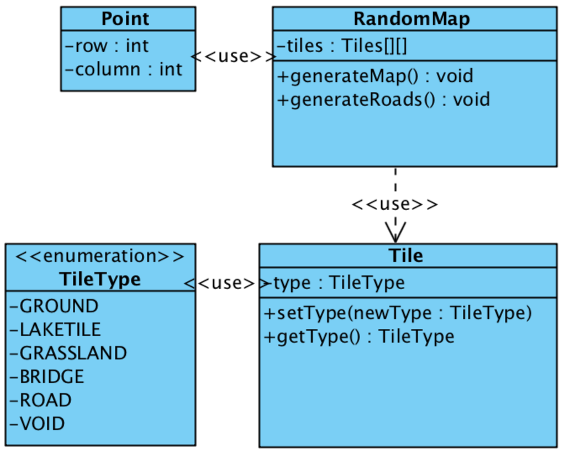


3.6 Input Management Subsystem

Input Handling System handles the mouse events created by the user. Since the user is going to be able to enter a username for himself/herself, and since the game will keep a record about who has the highest score, input handling subsystem should be able to handle mouse interactions as well.

The module will use KeyListener and MouseListener supplied by Java itself, for handling keyboard and mouse outputs, respectively.

3.7 Game Map Management Subsystem



Castle Defense is a game with procedurally generated maps. Game Map Management Subsystem is responsible for creating a random map represented with a 2-dimensional array.

RandomMap Class

RandomMap Class generates the entire map represents with two-dimensional array consisted of Tile objects.

Tile Class and Tile Type Enumeration

Tile class stores what kind of value the tile holds. This value can be either ground, lake tile, grassland, bridge or a road. These values are defined (along with the void, which is the default value of all tiles if not specified otherwise) in the TileType enumeration.

Point Class

Point class maps the row and column value of the given Tile. It also supplies algorithms such as distance calculation between two points and establishing a grid based road system.