**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**](#_xyo6ajmqttyr) **4**

[1.1 Purpose of the system](#_ns4mpv7iv5m7) 4

[1.2 Design goals](#_n1o2tmazjkk6) 4

[1.3 Trade-Offs](#_ls2stskjqpeh) 5

[**2. Software Architecture**](#_jdtzi2xkuuv) **5**

[2.1 Subsystem Decomposition & Architectural Style](#_ddh4errj1f6o) 5

[2.2 Hardware/Software Mapping](#_3k7zvlsg74c2) 8

[2.3 Persistent Data Management](#_5dp98wxl1i08) 8

[2.4 Access Control and Security](#_rkdrdueneozq) 8

[2.5 Boundary Conditions](#_k10c9cb8q76t) 8

[**3. Subsystem Services**](#_nexupu2rdexa) **9**

[3.1 Detailed Object Design](#_jg7ej3l6adgz) 9

[3.2 User Interface Management Subsystem](#_o6lpv5j1axu1) 9

[3.3 Game Logic Subsystem](#_f3zrhiurzrud) 10

[3.4 Game Entities Subsystem](#_5ix3jnqtdca4) 11

[Tower](#_9azrohe8yyej) 11

[Basic Tower](#_pycod8foez56) 11

[Ice Tower](#_qsul83yaikik) 11

[Explosive Tower](#_p50ktre5eqjw) 11

[Poison Tower](#_6auz1mob0ob2) 12

[Consumables](#_2sbchga36rmd) 12

[Stun Consumable](#_ch4nqkgajgh3) 12

[Glue Consumable](#_v58tae30p140) 12

[Bomb Consumable](#_yc9wq1wba4z2) 12

[Creep](#_1t21yvauwxk9) 12

[Viper](#_buy368l7om8k) 12

[IronBack](#_4ccrxtynoe9c) 12

[SpeedyGonzales](#_ds1rovxucwda) 12

[Boss](#_qs9wdtjq4962) 12

[3.5 File Management](#_p1vutmwgeap7) 13

[3.6 Input Management Subsystem](#_lo7mxstdgjg1) 13

[3.7 Game Map Management Subsystem](#_x5kk61o18q1i) 14

# **1. Introduction**

## **1.1 Purpose of the system**

Castle Defense is a tower defence 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 makes Castle Defence the best option for tower defence players.

## **1.2 Design goals**

Our design aim is offering different experience of tower defence 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 Defence will be shown. After the instructions, first waves is 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 Defence 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 defence game.

**Reliability:** Castle Defence 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 defence 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 Defence, 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 distracts users attention, low quality graphics also distracts users attention. Our design will balanced but in our design performance will outweigh memory, which means we will focus more performance rather than high quality graphics.

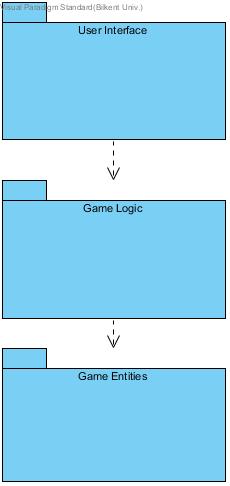
# **2. Software Architecture**

## 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.

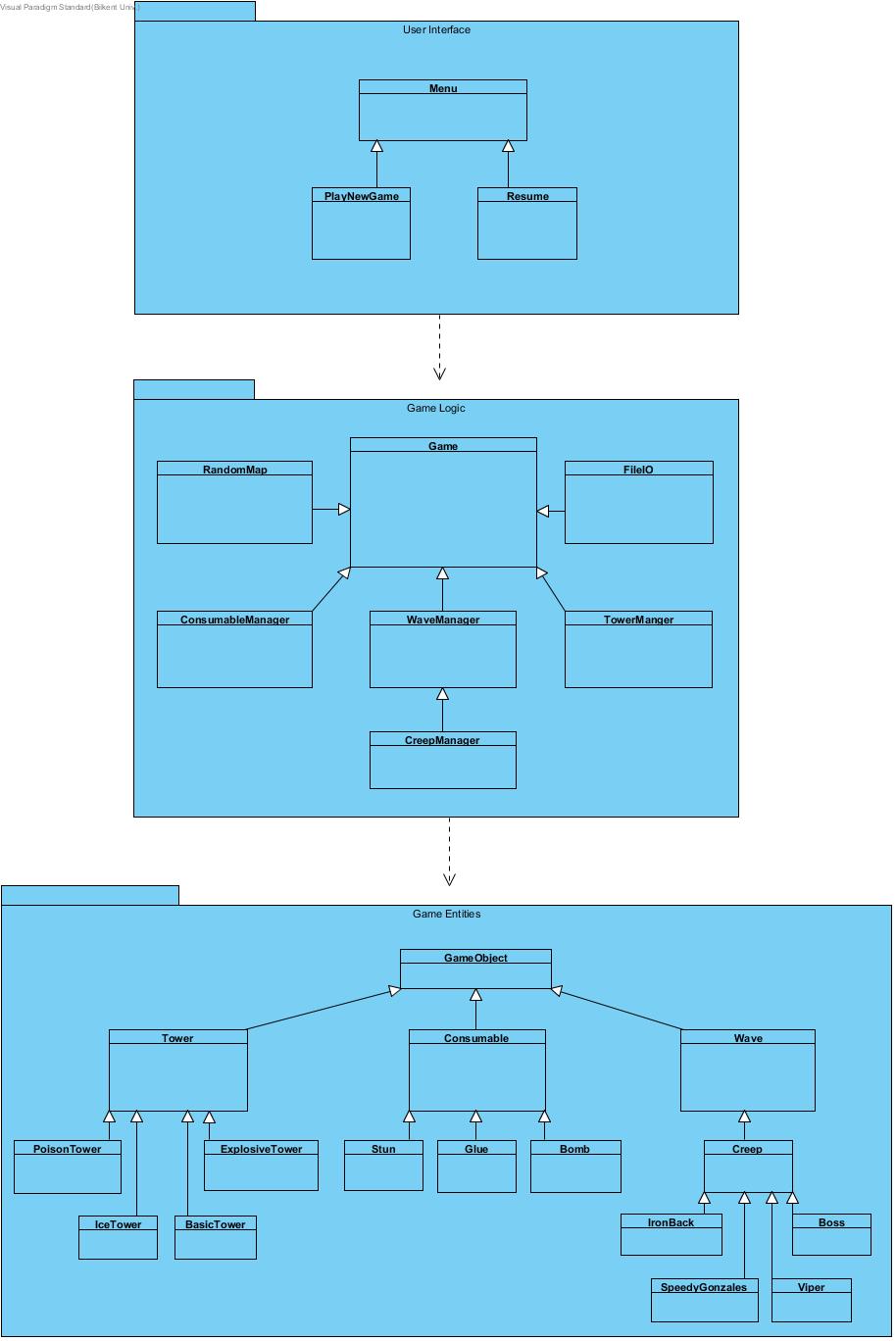
All three subsystems will be working together but they will not be affected that much by changes or errors on the other subsystems. As the system has three layers: layer 1 layer 2 and layer 3 each corresponding respectively to User Interface, Game Logic and Game Entities, layer 1 will be connected to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

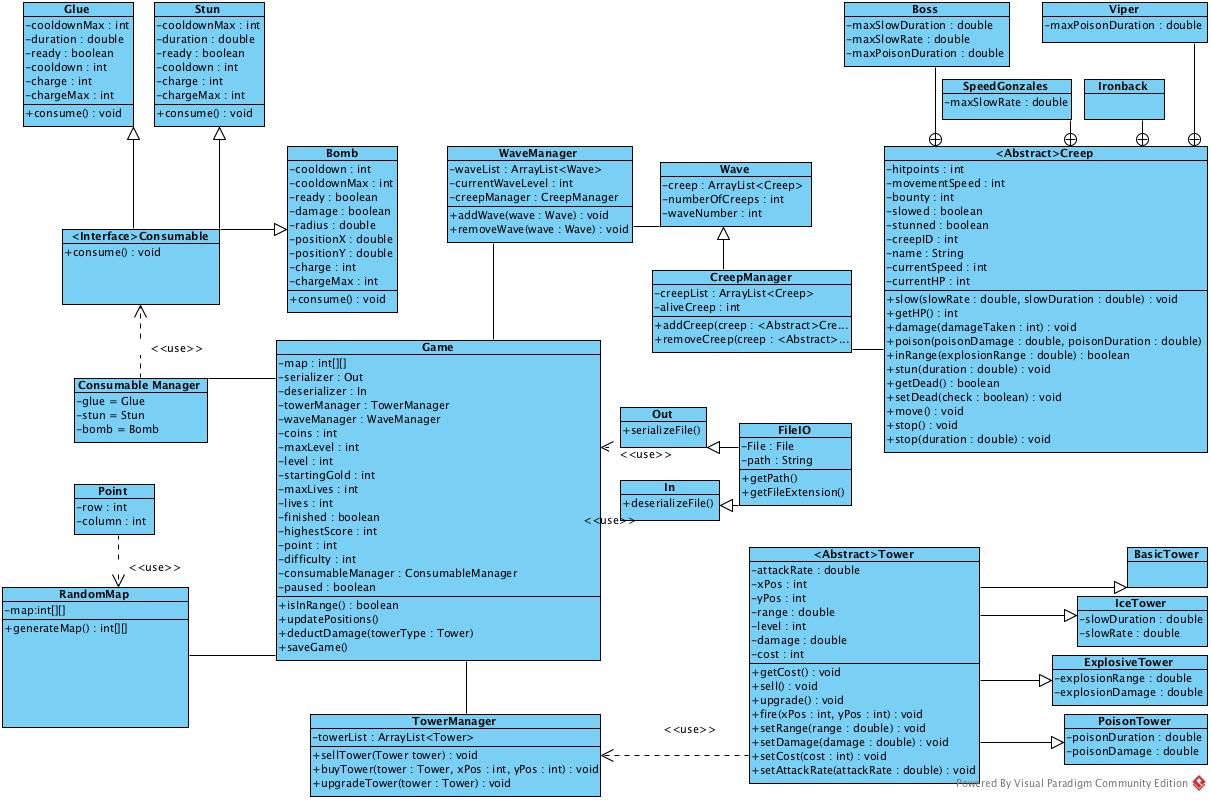
## 2.5 Boundary Conditions

“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.



## 3.2 User Interface Management Subsystem

User Interface Management Subsystem will be composed of “Menu”, 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, High Scores, Options, Tower Store and Credits. Although not shown in the more detailed subsystem diagram, all these option will be other View classes and will interact actively with the Menu Class. The most important classes will be PlayNewGame and ResumeGame classes which will take care of making possible to interact with the Controller of the system or Game Logic subsystem.

## 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 much coins user has and which level he/she is in. Therefore Game class is the main component behind the games logic.

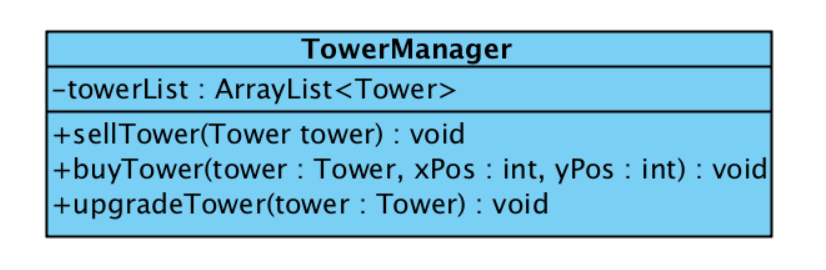
There is a Game instance running inside the game’s main loop. It tracks position of the creeps in and tries to find if one of them is in range in real time. It also checks for damage deducted by creeps, damage taken by creeps and whether a creep is dead or not. At the end of every cycle, it tries to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

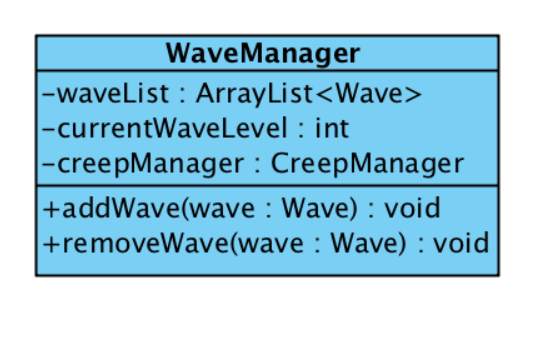
**TowerManager Class**

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.



**WaveManager Class**

Wave Manager is responsible for generating new creep waves. Waves will consist creeps with different kinds and abilities. Difficulty selected by the user will affect the size of the wave and the types of the creeps.



**ConsumableManager Class**

Consumable Manager class handles predefined consumables and their usage.

## 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.

### 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.

### Glue Consumable

Consumable that slows the creeps down.

### Bomb Consumable

Consumable that damages all creeps in a certain area.

### **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

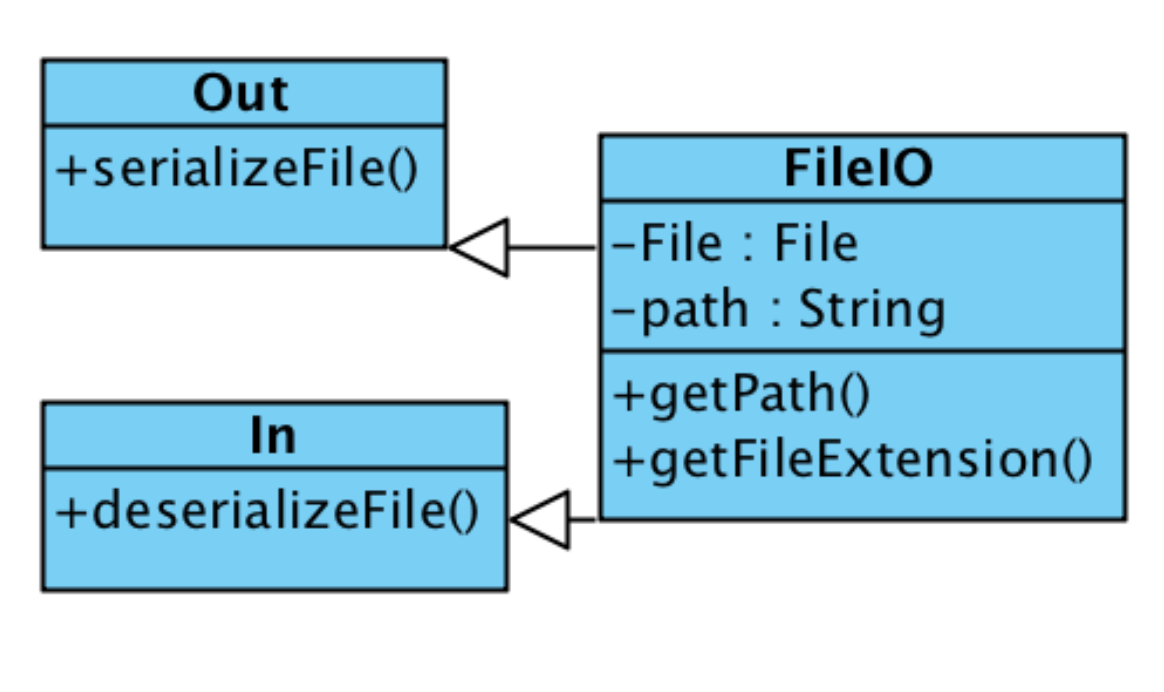
This is the type of creep, signified from the size of the creep. It has more hitpoints and has a slight lesser speed from the Viper type. It is only in the last creep of the wave.

## 3.5 File Management

**File Management**

File management subsystem consists on 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.

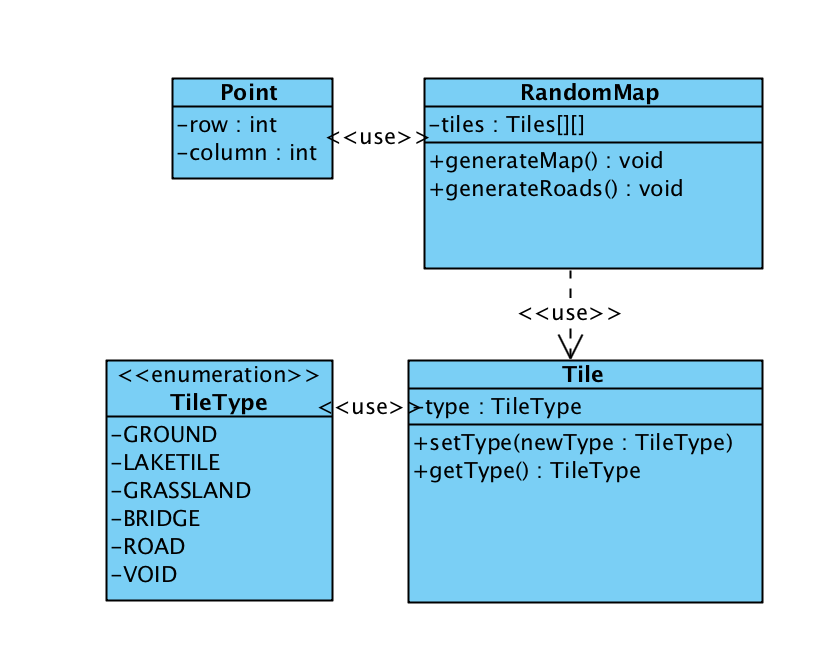


## 3.6 Input Management Subsystem

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 Defence 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 TileType 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.