SOEN 6441- Fall 2014 “Tower Defense”

Deliverable 2

Team 8

Dr. Paquet Joey

Name	ID
Wang, Guan	7225229
Sokke Nagaraj, Pavan	7608144
Karnwal, Arjun	7592043
Balekuttira, Kariappa Thimmaiah	7420269
Baiazid, Reem	5782465
Singh, Kanwaldeep	6756581

1. Introduction

Design document outlines the architectural and component design of the Tower Defense Game; it combines textual descriptions and UML class diagrams. The architectural design used is MVC architecture, internal design of each of its subsystem model, view and controller and the reason for this design. In addition Factory design pattern and tower targeting strategy

2. Functional Requirements

The functional requirements for deliverable 1 of the game include:

1. Saving a map to a file.
2. Loading a map from an existing file, then edit the map.
3. User driven interactive creation of a map
4. User-driven allocation of grid elements such as scenery, path, entry point and exit point.
5. Verification of map correctness before saving
6. Game starts by selecting a saved map, then loads the map
7. User-driven placing of towers on the map, following the game's restrictions
8. Implementation of currency and cost to buy or sell a tower
9. Implementation of towers' level-dependent characteristics such as level, cost to increase level, refund rate, range, power, rate of fire, special effects, etc.
10. Tower inspection window that shows its current characteristics.
11. Tower inspection window allows to sell the tower.
12. Tower inspection window allows to increase the level of a tower, changing its characteristics.
13. Implementation of the currency, cost to buy/sell a tower, and reward for killing critters
14. Tower dependent characteristics, cost to increase level, refund rate, range, power, rate of fire, special effects
15. Implementation of three different kinds of towers that are characterized by having different special damage effects.
16. Allows the user to start a wave, upon which a wave of critters starts moving from the starting point to the ending point along path cells

17. The towers can shoot at the critters, inflicting damage, and eventually killing them.
18. The towers can target the critters using different strategies, e.g. nearest to the tower, nearest to the end point, weakest, strongest, etc.
19. Wave-based play, i.e. when all critters in a wave have been killed, the player can place new towers, upgrade towers, and start a new wave.
20. End of game, e.g. when a certain number of critters reach the exit point of the map, or the critters steal all the player's coins.
21. Tower inspection window that shows its current characteristics, allows to sell the tower, increase the level of the tower, and select the tower's targeting strategy.

3. Architectural Design

The architecture of our system adopts the widely used **Model-View-Controller (MVC)** architecture, this model is chosen for our project since its architectural pattern separates the representation of information from the user's interaction with it that is the source code has no reference to the Controller or the view. As a result, a clear separation of concern between the logic, data and presentation is achieved. There are many advantages of highly cohesive software architecture such as maintainable system where modifying one component doesn't necessarily require changing the other components. In addition, MVC architecture is useful for developing in a team environment because it allows the three components to be developed simultaneously allowing for a very flexible and rapid development.

A **model** notifies its associated views and controllers of state changes. This notification allows the views to produce updated interface, and the controllers to change the available set of commands.

A **view** registers the controller to receive user interface events and requests from the model the information that it needs to generate an output representation.

A **controller** handles the user`s input and can send commands to its associated view to change the view's presentation of the model. When the change is purely cosmetic we update the view.

This pattern was used in implementing the map elements

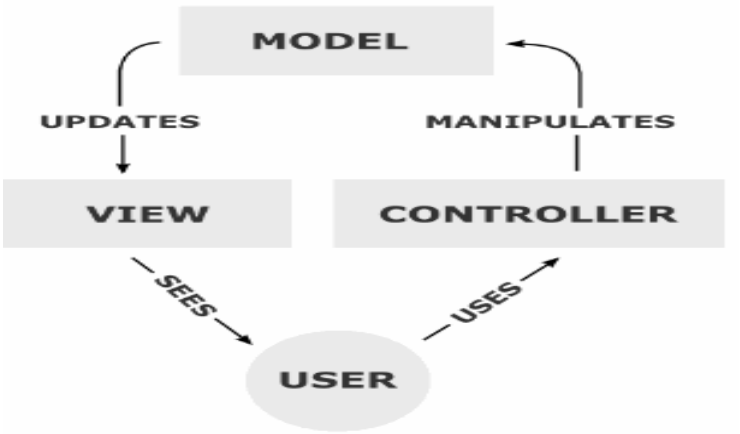
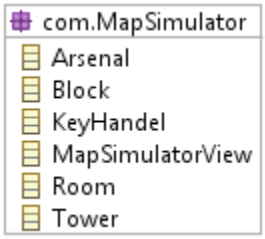


Figure 1 – Model View Controller Diagram

(i) Design Pattern: Model View Controller

Intent: Apply a "Separation of Concerns" principle in a way that allows developers to build and test user interfaces.

UML Class diagram for MVC Architecture
package: com.IDG.mapSimulator



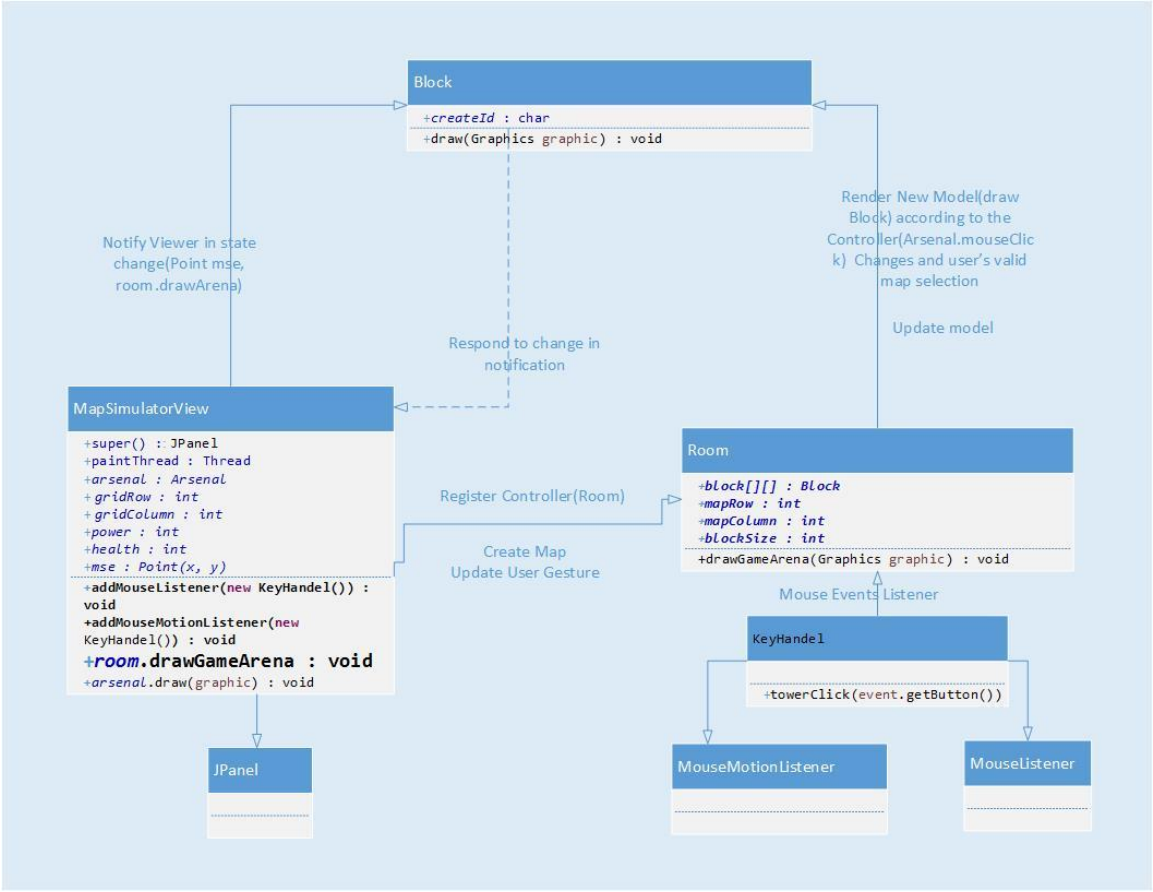


Figure 2 – UML Class diagram for MVC Pattern
Model – Block.java
View – MapSimulatorView.java
Controller – Room.java

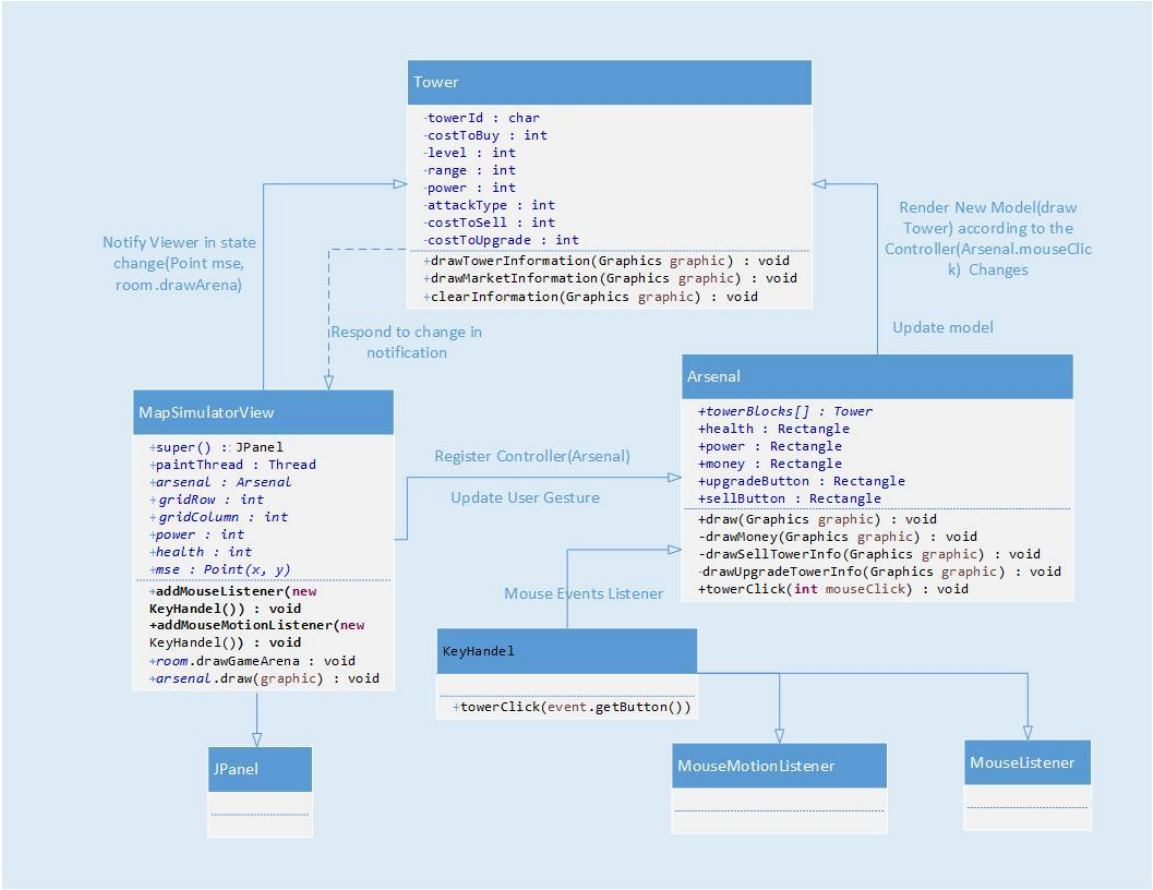


Figure 3 – UML Class diagram for MVC Pattern

Model – Tower.java

View – MapSimulatorView.java

Controller – Room.java

(ii) Design Pattern: Factory Method

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

The second architecture of our system adopts the **Factory design pattern** which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created. This is done by creating objects via calling a factory method which is specified in an interface and implemented by a child rather than by calling a constructor.

It's used in the critter wave creation, Strong enemy and Weak enemy which are subclasses of Enemy type.

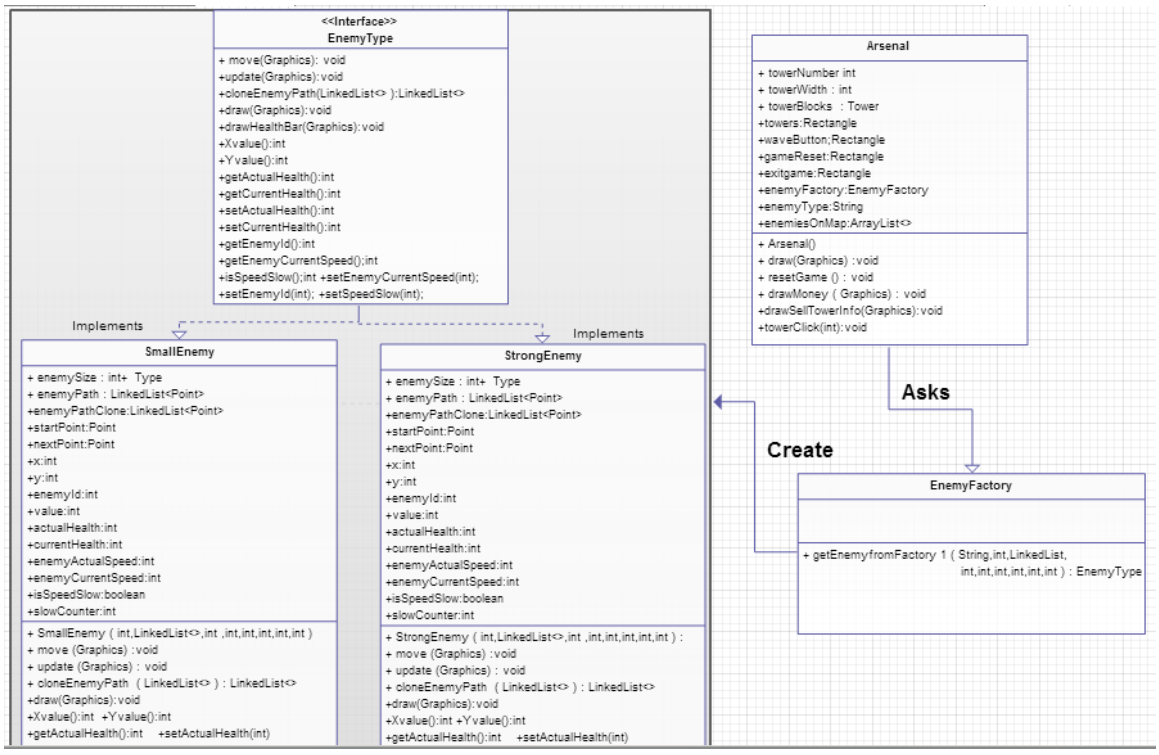


Figure 4 – UML Class diagram for Factory Pattern

(iii) Design Pattern: Singleton

Intent: Ensure a class only has one instance, and provide a global point of access to it. This is useful when exactly one object is needed to coordinate actions across the system.

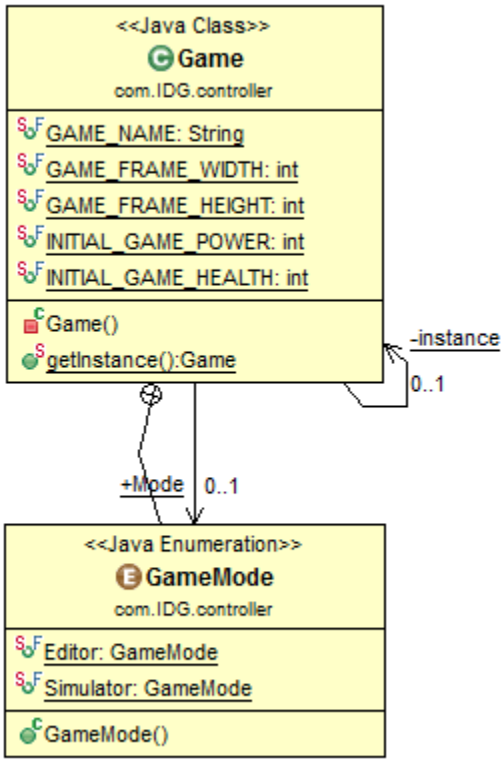


Figure 4 – UML Class diagram for Singleton Pattern

```
/**
 *
 * If the Game instance was not previously created, create the instanc
 *
 * @author Pavan Sokke Nagaraj <pavansn8@gmail.com>
 * @version Build 2
 * @since Build 2
 * @return return the created/existing Game instance
 */
public static Game getInstance() {
    if (instance == null)
        instance = new Game();
    return instance;
}
```

Code Snippet of singleton pattern in Game.java

(iv) Design Pattern: Observer Pattern

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

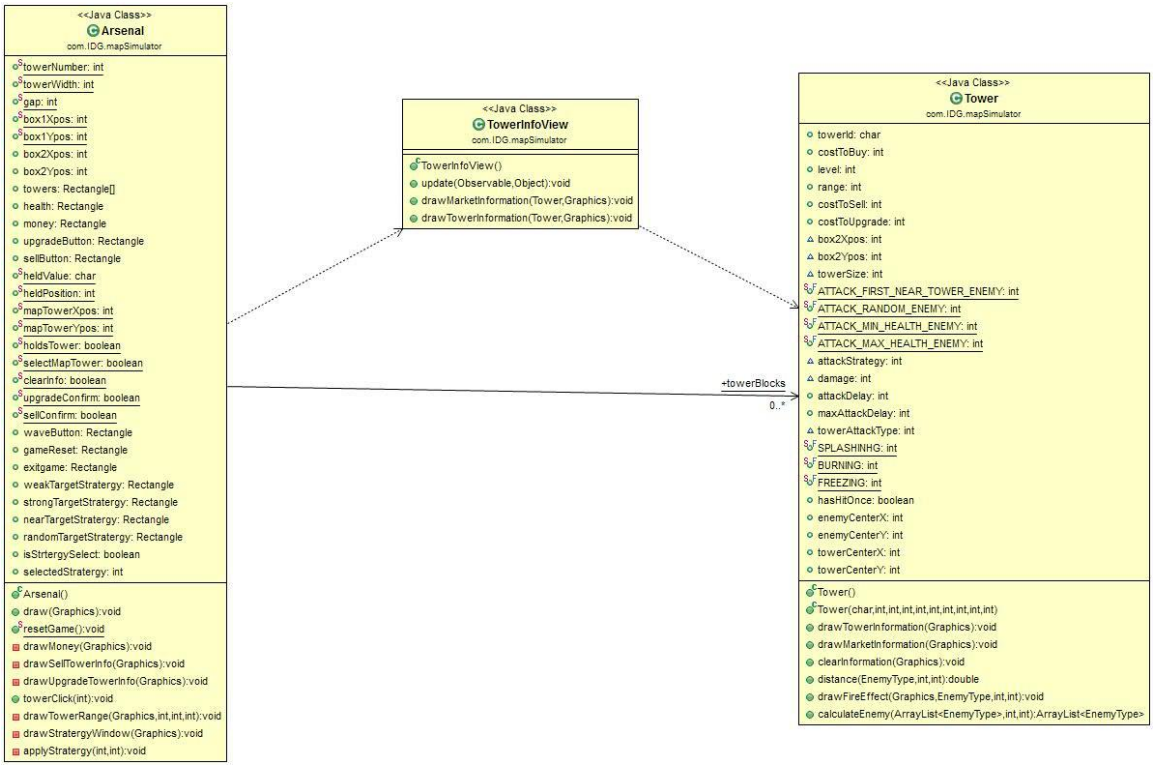


Figure 5 – UML Class diagram for Observer Pattern

Model Class: - Tower.java

Viewer Class: - TowerInfoView.java

4. Testing

Unit testing applied to our system is the JUnit Framework for the Java Programming Language. Since its time consuming to test all the methods in the system as the large number of methods in our system is large, only selected sets of methods were tested and the method chosen test the most important aspects of the code.