

**SOEN 6441 : Advance Programming Practices (Winter-2016)**

**Tower Defense Game**   
Supervisor: **Dr.** **Joey Paquet**

**Submitted by: TEAM-2**

Lokesh Parappurath - 27299680  
Iftikhar Ahmed - 26854605  
Muhammad Umer - 40015021

Armaghan Sikandar - 27421001

**Introduction**

The Tower Defense Game application consist of a map in which the critters move across a path from start to end point and the player should try to destroy the critters by setting up tower defences before the critters reach the tower.

This document contains the details regarding the architecture design of the program code used in the second build. This build consist of creating a custom map and game play in which user selects a map and on starting the game, the critters move in the loaded map through the path. Towers can be placed near the path by the user to inflict damage to the critter eventually destroying them.

The game is developed as a desktop application using the MVC architecture design. The MVC architecture is implemented using Observer pattern design in our code. Further, we are using the Strategy, Singleton and Factory patterns for tower, game controller and critters for simplifying the code development and improving the understandability and reliability of the whole program structure.

The game has 2 views:

1. **MapEditor:** For creating and editing maps for the game.
2. **GameWindow:** For playing the game.

**Functional Requirements**

The functional requirements for the 2nd deliverable includes:

* + User driven interactive creation of a map as a grid of user-defined dimension.
  + User-driven allocation of grid elements such as scenery, path, entry point and exit point.
  + Saving a map to a file exactly as edited.
  + Loading a map from an existing file, then editing the map.
  + Verification of map correctness before saving.
  + Game starts by selecting a saved map, then loads the map
  + User-driven placing of towers on the map, following the game’s restrictions
  + Implementation of currency, cost to buy/sell a tower, and reward for killing critters.
  + Implementation of all towers’ level-dependent characteristics: cost to increase level, refund rate, range, power, rate of fire, special effects.
  + 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.
  + Implementation of three different kinds of towers that are characterized by having different special damage effects, e.g. splash, burning, freezing.
  + Towers can shoot at the critters, inflicting damage, and eventually killing them.
  + Towers can target the critters using different strategies, e.g. nearest to the tower, nearest to the end point, weakest, strongest, etc.
  + 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.
  + 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.
  + 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.

**Agile Methodology**

We have adopted agile scum development methodology for building the application and the whole project is going to be built in different stages. Further, the intermediate releases ensure that the project is being built the right direction.

This is the second stage of the development and here we demonstrate our accomplishment of how a map can be created successfully, how the demo game is implemented so far in which a wave of critters move through the path in the map once game starts and how damage is inflicted onto the critters by the user placed towers on the scenery.

In our team, the documentation, coding and testing has been done collectively with each of the work reviewed by one/more peer(s).

**Architectural Design – Observer Pattern**

The project is developed using the MVC architecture and is implemented using the object pattern design. This model has been selected because of its decoupling advantage and its allows development of the business logic and the view separately and independently. Further, it provides flexibility in feature addition and maintenance.

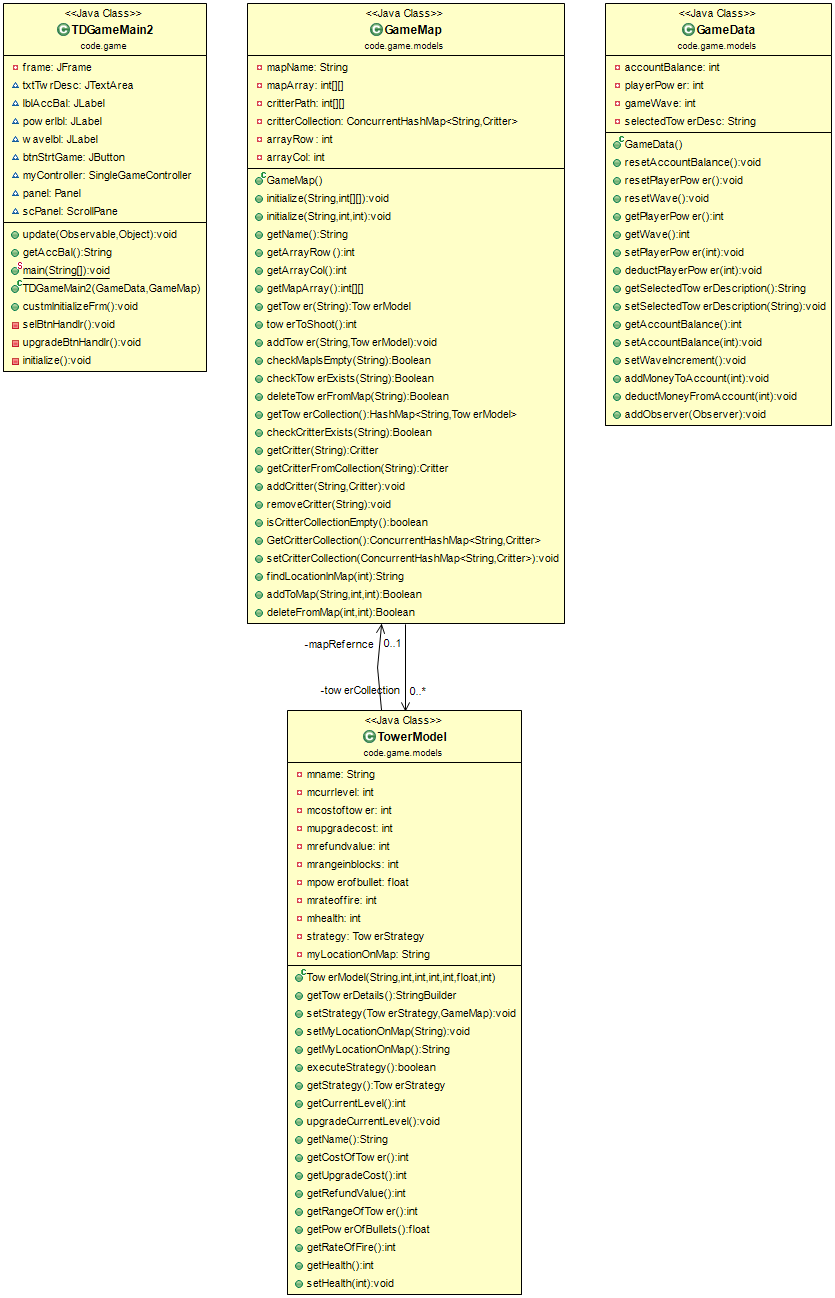
The project is developed such that it has a view which extends the observable class and has a model which implements the observer class. The user interactions with the view are transferred to a separate controller class which handles all the user events and accordingly update the model objects.

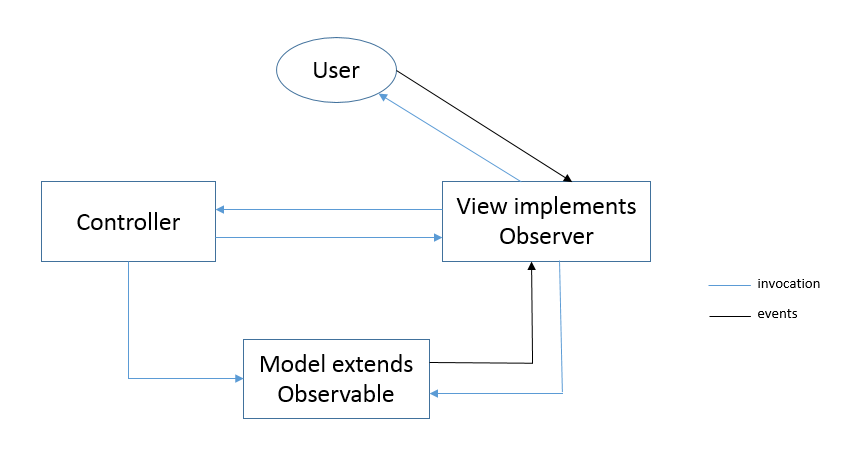
The **view** is updated by the model by triggering an event every time its state gets updated. The view invokes methods in the controller class for handling user interactions.

The **controller** is responsible for the implementing the application behavior towards user interactions. It updates the model classes based on the user interactions.

The **model** classes implement the business logic and handles the data associated with the application. It is an extension of the observable class.

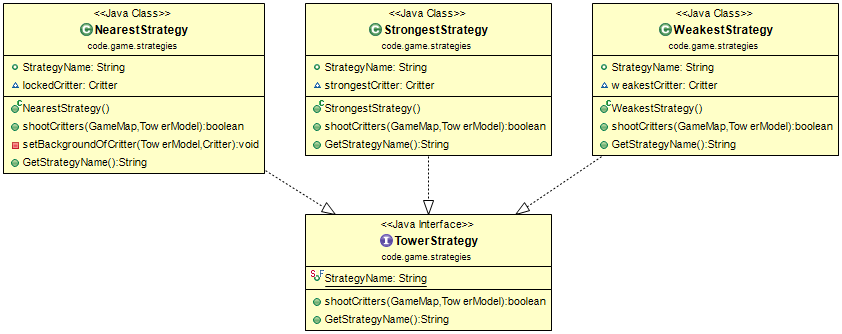
The following is the top level view of the Observer pattern we have implemented:





**Strategy Pattern**

While implementing the towers we came across the scenario when the shooting logic of the towers has to be determined by the user during the game play. Now, since this strategy needs to be determined at runtime, we have implemented the tower shooting logic using the strategy pattern since the tower can have different ways of targeting the critters. We are using this pattern since its design aligns in purpose with the selection of tower shooting strategy. The following is the UML diagram of our strategy pattern implementation:



We have defined a base interface class with few methods which is implemented using different algorithms for different approaches such as nearest critter 1st, strongest critter 1st and weakest critter 1st in 3 different strategy classes.

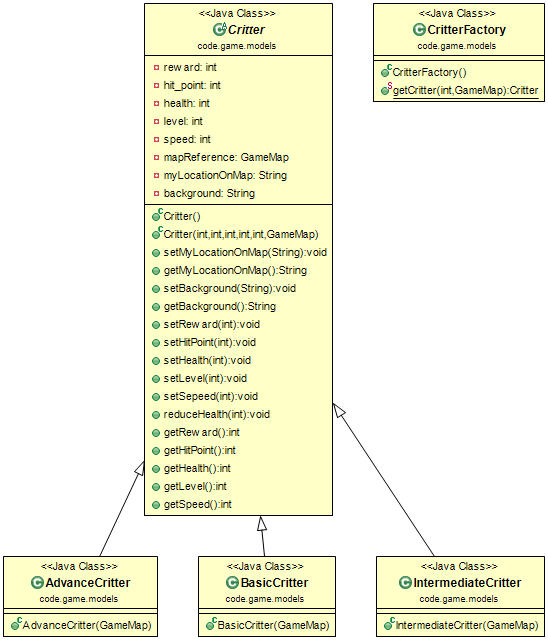
**Singleton Pattern**

During application development, there occurs cases where a single instance of a class is required. For such cases we use the singleton pattern. The singleton pattern ensures that only one instance of a class is created and has a global point of access to it.

In our game application, we have created the game controller as a singleton controller. This is because for the whole game play, only one game controller class instance is required. Ensuring that only a single instance of the class is created keeps the code design simple.

**Factory Pattern**

In the game application, we are using different types of critters all of which are having the same properties. Since the purpose of the factory design pattern is to handle polymorphism and in situations when a class uses hierarchy of classes to specify which object it creates, we decided to implement critters using the Factory design pattern.



In the application, we have a base critter class with the basic properties and methods to handle the critters. There are 3 different critter classes which derive from the base class. The instantiation of the critter is decided by the critter factory class which returns a critter type decided at runtime to the base critter reference object.

**Testing**

Our testing had 2 phases:

Unit testing: Unit testing is done using the Junit Framework provided by Java. Since it is time consuming to test all the methods in the system in the short period, we have restricted ourself to 26 specific important test cases. The test cases have been described in short in the test case document.

Integration testing: All the units of the code have been integrated after unit testing to form this project build. Further the code has been verified to work properly after integration.

The build is then checked for acceptance testing where all the requirements are checked for one on one functionality acceptance check.