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 basic idea of Tower Defense Game is to destroy moving critters in a specific path using towers. This is the complete final build of the game and has all the features mentioned in the below functional requirements.
This document contains the details regarding the architecture design of the application. This final build consist of a window for creating a custom map and a game play window in which user selects a map and starts playing the game. When the game is started, 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 and 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 this deliverable includes:

- User-driven interactive creation of a map as a grid of user-defined dimension with grid elements such as scenery, path, entry point and exit point.
- Game starts by user selection of a previously user-saved map, then loads the map.
- First the player can place new towers, upgrade towers, sell towers, and signify that critters are allowed in on the map, when all critters in a wave have been killed or reached the end point, a new wave starts.
- When a certain number of critters reach the exit point of the map, or the critters steal all the player's coins, or the player succeeds in killing a certain number of waves.
- Implementation of currency, cost to buy/sell a tower, and reward for killing critters.
- Critter waves are created with a level of difficulty increasing at every wave.
- Implementation of at least three different kinds of towers that are characterized by special damage effects.
- The towers can target the critters using the following mandatory strategies: nearest to the tower, nearest to the end point, weakest critter, strongest critter. It must be possible to set a different targeting strategies for individual towers.

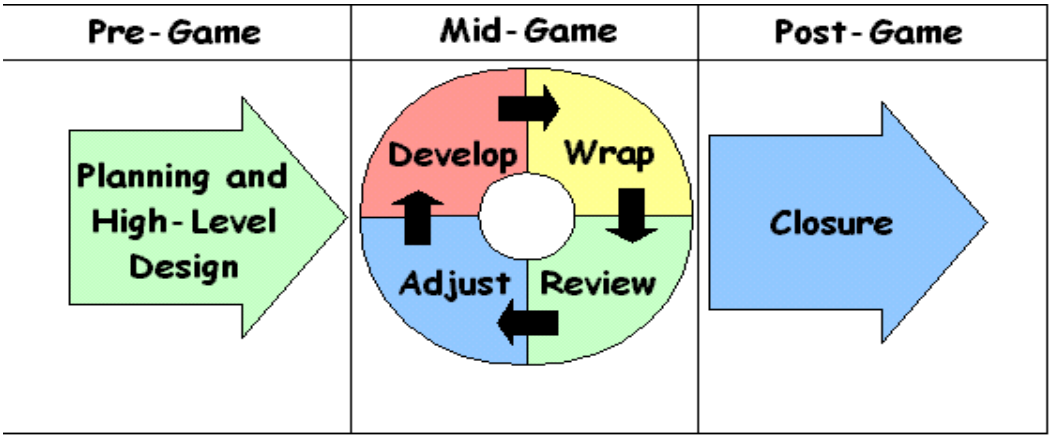
- Tower inspection window that dynamically shows its current characteristics, allows to sell the tower, increase the level of the tower, select the tower's targeting strategy and view the individual tower's log.
- Critter observer that allows to dynamically observe the current hit points of any critter on the map.
- Game log that records all events happening in the game, including placement/ upgrade/ selling of towers, critter wave creation, etc.
- Map log that records in the map file the time of original creation of the map, when it was edited, when it was played and what was the result of the game every time it was played.
- As a game is being played, allow the user to save the game in progress to a file, and allow the user to load the game in exactly the same state as saved.

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 final stage of the development and here we demonstrate our accomplishment of how a map can be created successfully, how the demo game is implemented in which different waves 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).

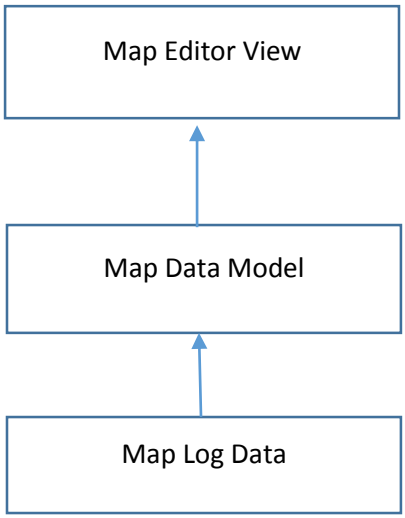


[1]

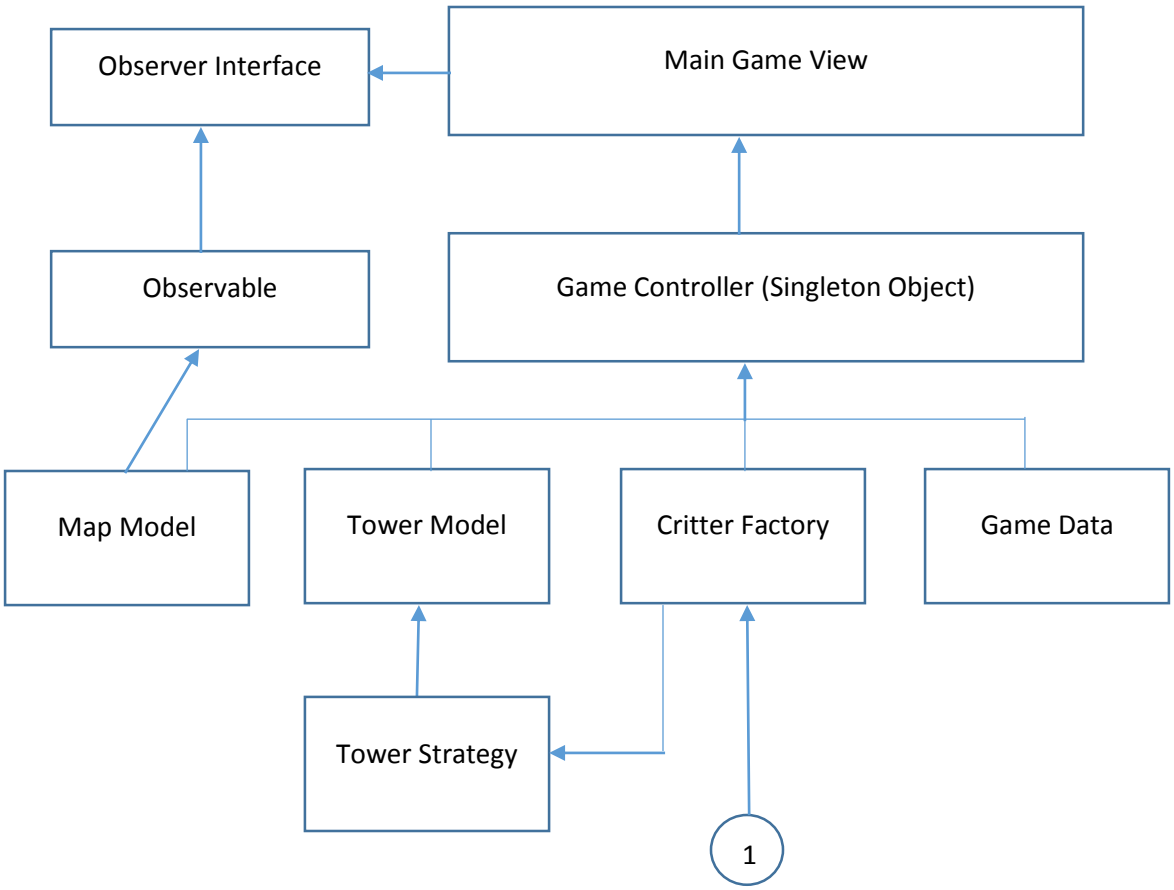
Architectural Design

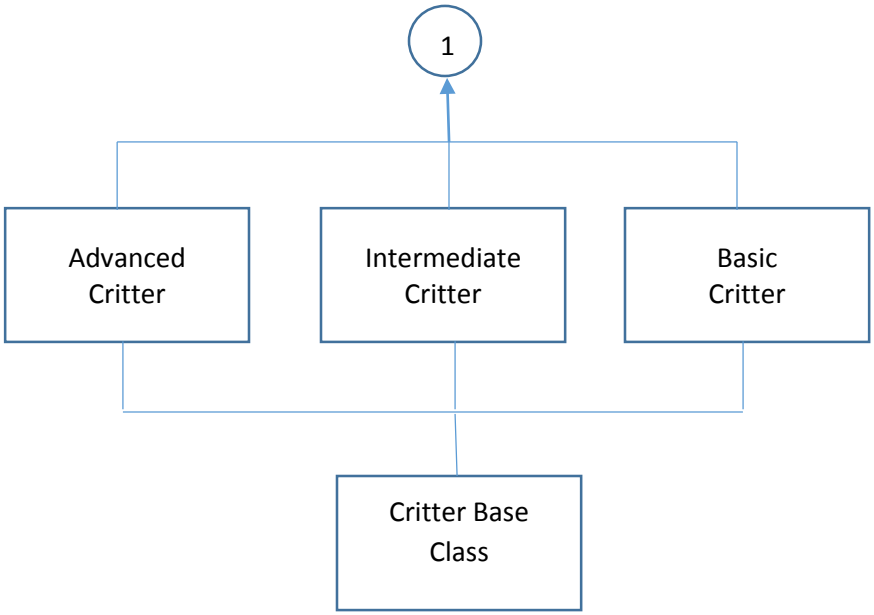
The game application, as mentioned earlier, has 2 parts:

Map Editor



Game Window

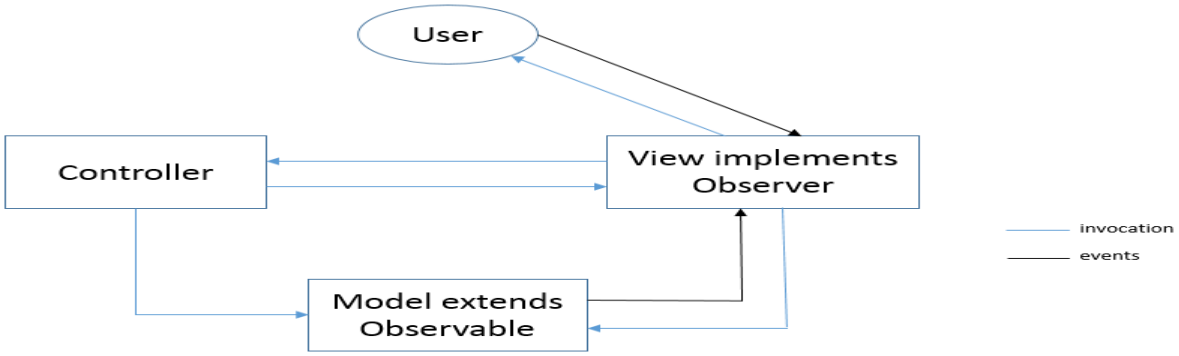




Observer Pattern

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 it allows development of the business logic and the view separately and independently. The project is developed such that it has a view which extends the observable class and has a model which implements the observer interface. 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.



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.

We have defined a base interface class with few methods which is implemented using different algorithms, for different approaches such as nearest critter to tower 1st, nearest critter to end point 1st, strongest critter 1st and weakest critter 1st, in 4 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 and prevents multiple instantiation of the controller class.

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.

Logs

In any application, the development process is not perfect. In order to assist the developer with debugging and the user with analyzing errors or unpredicted behaviors, we use logs. Logs help in determining the errors or unexpected behaviors faster. In our game application, we are also using the log files to determine what is happening inside the application when various functionalities are being executed.

We have created separate log files for maps, towers, critters and for the whole game. This has assisted us in debugging the vast application more easily.

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.

Software Versioning Repository

Repositories are a great way to manage the code when the development team consist of multiple developers and project mates and development is performed in stages. This helps in backtracking any changes which turns out to cause unpredicted results and in roll back to previous stable versions.

One of the popularly used repositories is the github which provides a web interface as well as eclipse plugins to manage the code online. Due to the reliability, ease of use and multiple platform support provided by github, we are using git for versioning our game application, with the project stored online in github for easy multiple remote access.

Reference

[1] http://www.codeproject.com/Articles/4798/What-is-SCRUM

[2] http://users.encs.concordia.ca/~paquet/wiki/images/8/82/SOEN6441.patterns.ppt