McGill University

Architecture and Design Specification

Team 6: Amlekar R, Aydede E, Gupta Y, Wright A

ECSE 321:Introduction to Software Engineering

Dr. Sinnig

2015/03/29

Revision History:

Date	Changes to Document	Version	Changes made by
2015/03/29	Creation of Document	1.0	Team 6
2015/04/05	Heavy editing and addition of UML diagrams	1.1	Team 6

Table of Contents

List of Illustrations

Introduction

The purpose of this document is to outline in detail the software architecture and design of the "Tower Defense Game" - a basic game for the project component of course ECSE 321 "Introduction to Software Engineering". The document will capture the significant architectural and design decisions used in the software.

Audience

The intended audience of this document is described in the table below:

Readers	**Use of Document**
Dr. Daniel Sinnig	To understand implementation of project and gauge aptness of architecture used
Testers	To plan testing of the game
Developers maintaining/extending the application	To understand the game for maintenance or extension purposes

Scope

The Tower Defense Game can be complex but given the project timeframe and for the sake of simplicity and elegance, certain features are omitted. They include:

Multiplayer feature:

One might expect a survival mode where players can compete against each other in real time, to outlast the Critters longer than their opponent. Although this would be a great feature, it would require a longer timeframe to implement.

3D graphics:

Since we are developing a Tower Defense Game, one might expect it to not only have a state of the art game engine, but also have immersive graphics. We will only be implementing the game engine, as time spent working on the engine will produce higher value than better graphics. The graphics will be implemented from a library.

Related Documents

Prerequisite document: "Tower Defense - Software Requirements Specifications" Software Requirements Specification (SRS), V. 1.0

Standard Java documentation: Javadoc/index.html

System Architecture

The Tower Defense game implements a layered, multitier, architectural strategy. This architecture is based off of the typical client-server three-tier architecture, which has a presentation layer, a domain logic layer, and a data storage layer. Since the game does not store or use much data, the third layer has been removed. The architecture is thus a two-layered architecture, comprised of a presentation layer, and a domain logic layer. Advantages of any n-tier architecture include being able to make changes to the program easily. The developer can change aspects of the presentation layer with minimally affecting the domain model layer.

The architectural set-up that the game implements is the Model-View-Controller set-up. Although this was only briefly mentioned in class, the teaching assistant Shabbir demonstrated its usefulness, and taught the tutorial attendees how to implement it (through a simple snake game). The MVC layout is beneficial as it fits nicely into the two-tier architecture being used. The view component of the MVC fits into the presentation layer, whereas the models and controllers fit into the domain logic layer. Having one controller allows for high-cohesion and low coupling, as it is not necessary for all of the "models" to be coupled to one-another. Instead, the controller connects all of the models together, and displays it through the view. In our case, we have one Model-View-Controller for the map editor, and one for the game (although they reuse many of the same classes). We implement a slight alteration, which is that we add a "Helper" component to the controller, so it is better represented by the phrase Model-View-Controller-Helper.

One can see this representation below, with the modified diagram from Shabbir's tutorial.

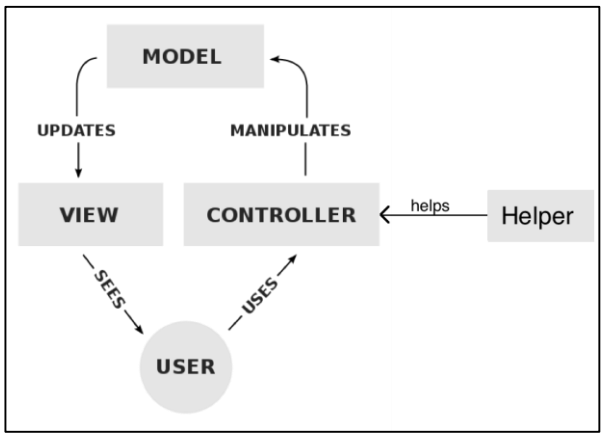

Figure 1 - Model View Controller Helper diagram (Shabbir)

Complete Architecture Diagram

Please note that the first diagram shown is the full architectural diagram (but with most methods and attributes omitted). Then there are more focused diagrams of the presentation and business logic layers.

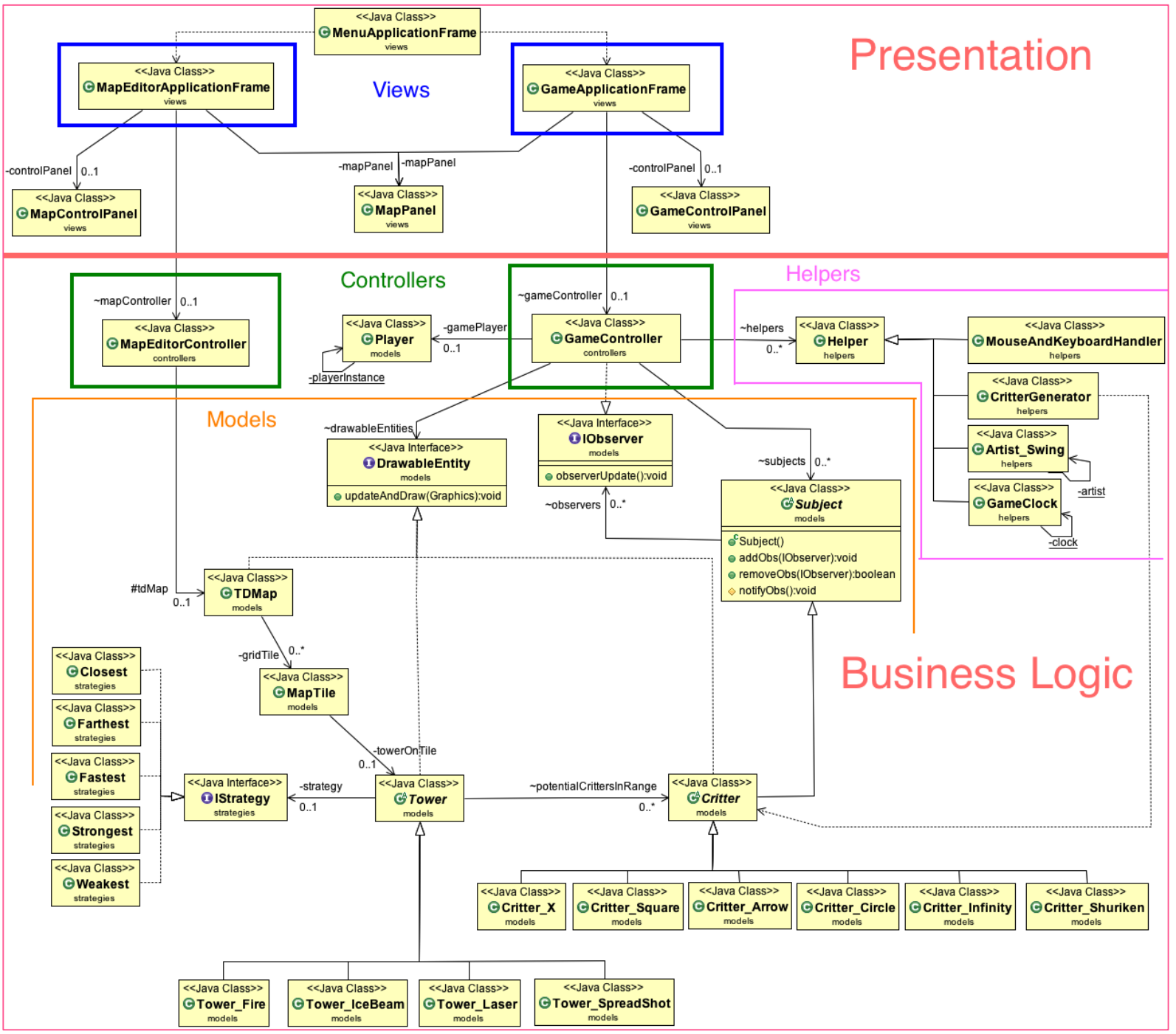

Figure 2 - Complete Architecture Diagram

Although this first diagram may look complex, it actually demonstrates the low coupling and relative clean setup between the classes. Without the MVCH layout, there may have been seemingly unnecessary, messy, connections. With the MVCH, all of the connections can be easily understood. For example, there is a connection between the GameController and DrawableEntity. This interface is implemented by the Map, Tower, and Critter. Critters and Towers have their own subclasses. These classes inherit from the superclasses. This is

much cleaner than having all of these subclasses connect directly to the GameController (and is a direct result of the indirection principle).

Presentation Layer:

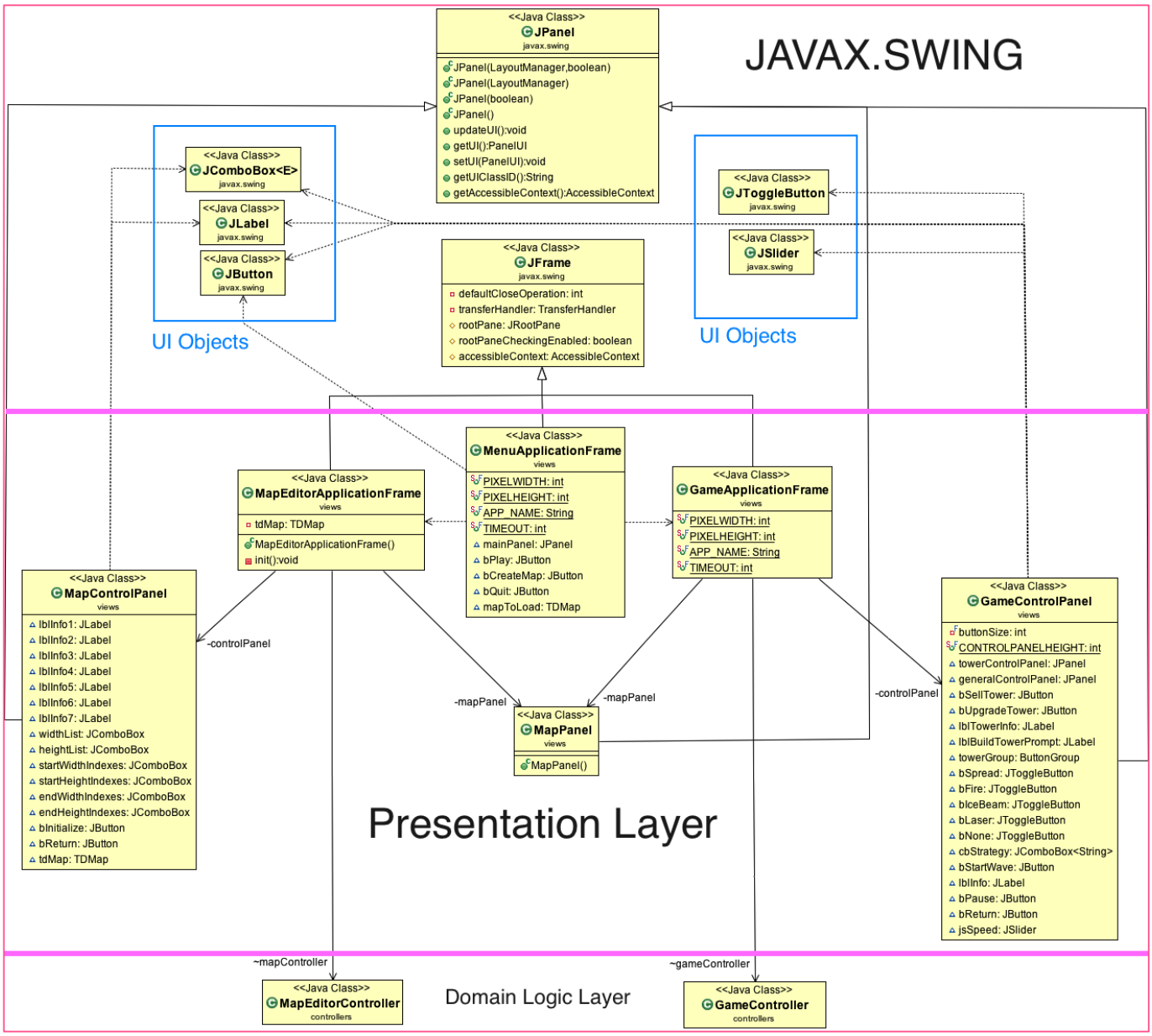

Figure 3 - Presentation Layer Diagram

Above is the presentation layer with all of the methods and attributes shown. Java Swing was used to create the GUI that actually displayed all of the models. As such, the presentation layer (or the view in model-view-controller) extends heavily from Javax.Swing. The control panels for the game and for the map editor are both JPanels, the three application frames are JFrames, and all of the buttons or labels or comboboxes are from Swing as well.

Domain Logic Layer

Drawable Entities

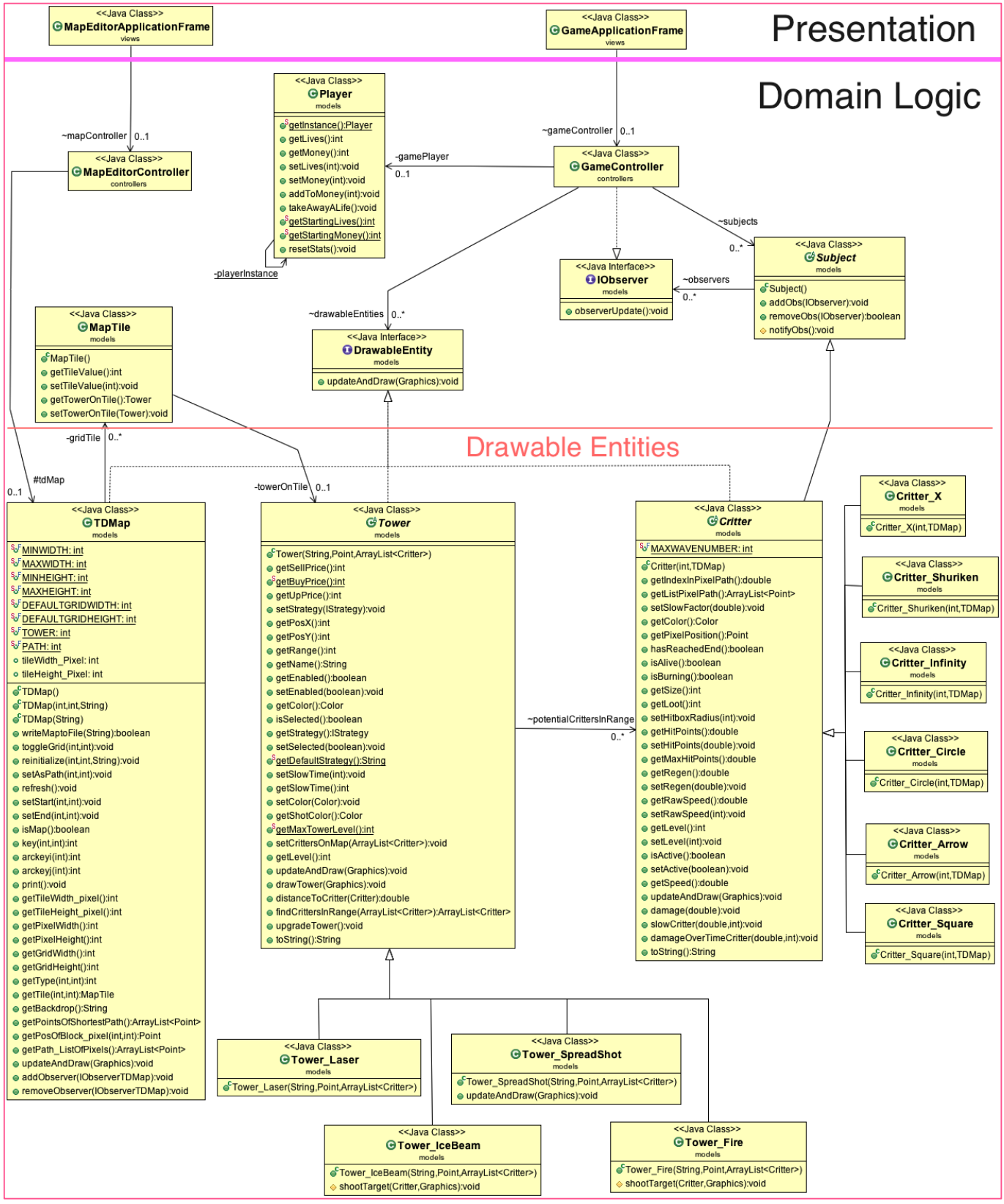

Figure 4 - Drawable Entities

Note that the attributes of the Map, Tower, and Critter were not shown, as it created messy diagrams. In addition, it is redundant, considering that there are getter and setter methods that are shown.

Game Controllers and Helpers

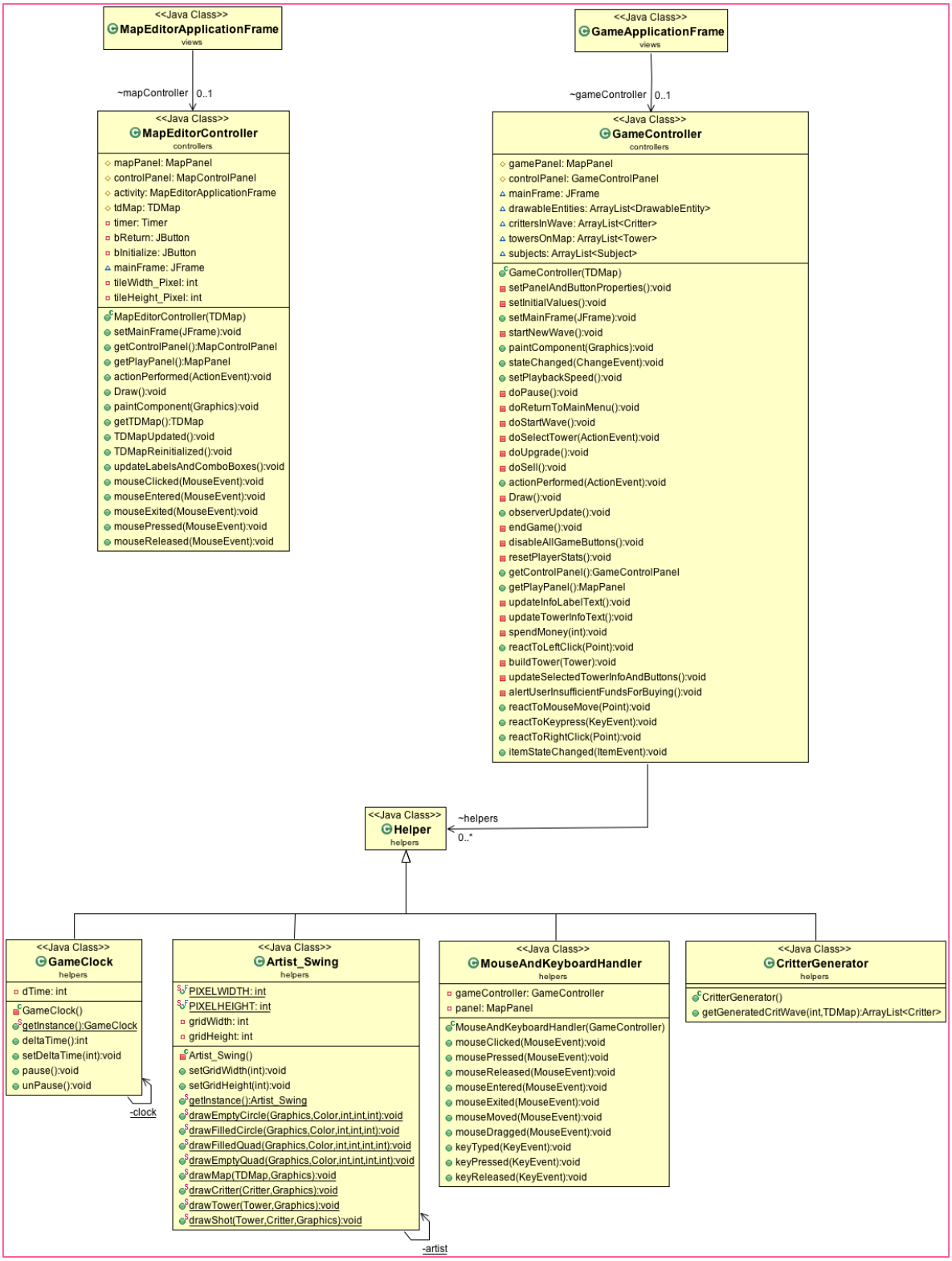

Figure 5 - Game Controllers and Helpers

Strategies

This layout in the business logic section is also a design pattern that will be discussed below in the principles section.

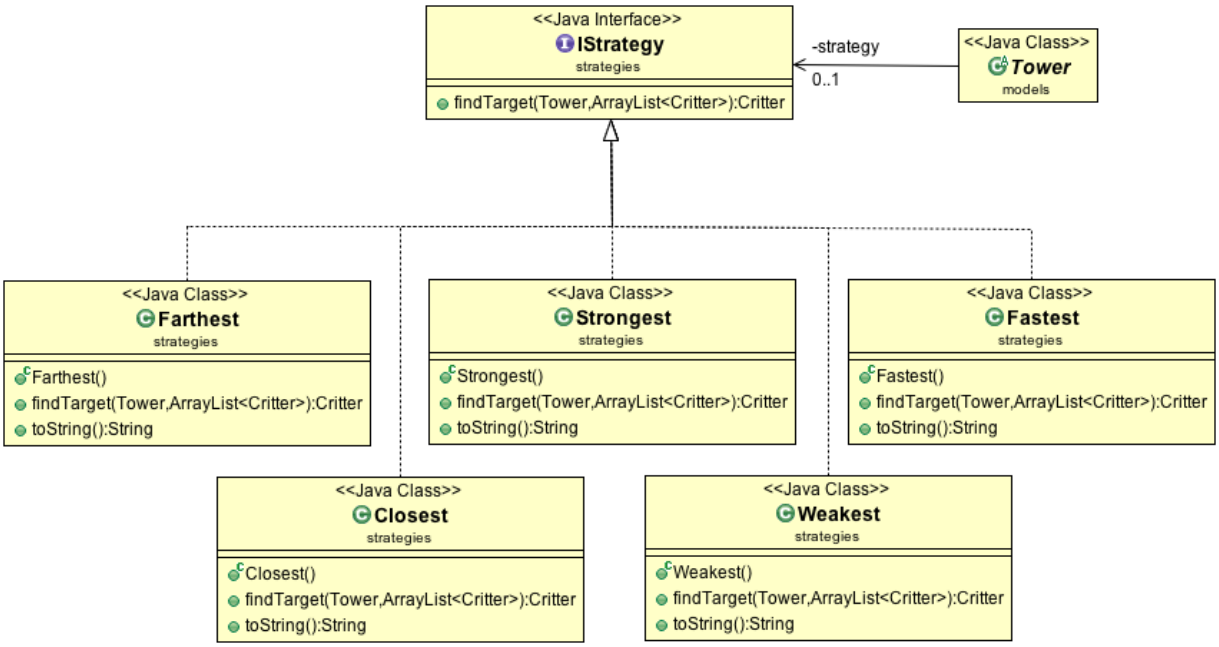

Figure 6 – Strategies Diagram

GRASP and Design Patterns:

This section is dedicated to any programming problems or inefficiencies that were solved using the object oriented GRASP principles or design patterns learned in class or tutorials. Throughout the development of the Tower Defense game our team was able to identify and resolve general coding problems using the various GRASP principles. Some of the most prevalent principles used include:

- Indirection
- Information Expert
- High Cohesion
- Polymorphism

These principles allowed the code of the Tower Defense game to be considerably easier to understand and altered, if needed. In addition to the GRASP principles, our team implemented many design patterns as well. Among these design principles are:

- Subject Observer
- Strategy
- Singleton

Without these fundamental principles and patterns, the process of interpreting or editing existing code would be an unpleasant experience.

One result of using these principles and patterns is that it is relatively simple to add on to code, or to alter it. For example, our group initially decided to use the Light Weight Java Game Library (LWJGL) to display the game. Due to information expert and in a goal to create high cohesion, we had all of the methods for drawing in one class called Artist_LWJGL. When we decided to switch to Swing, we changed the class to Artist_Swing, and implemented the drawing with Swing. As there was very little coupling between the Drawable Entities (critters, towers, maps) and the artist, this change was painless.

Principles

1) Principle Name: Indirection Principle

Applying the indirection principle results in low coupling. It involves creating a middle class that acts between two other classes. We have applied the indirection principle throughout our code, and the helpers package contains classes that all are partially the result of the indirection principle. One example of the indirection principle is how we handle Critters being observed by the game controller.

Sample Problem: In the tower defense game, should the Critter class be directly responsible for adding removing or notifying observers?

Solution: Instead of directly implementing the add remove and notify observer methods in Critter, we create a middle class called Subject that Critter extends. This subject includes the list of IObservers and the methods for adding, removing, and notifying observers. This removes the coupling between observers and critters and allows the potential for other classes to be observed by having them extend the Subject class. It also means that if we wanted to change a property of a Subject, we simply change the Subject class.

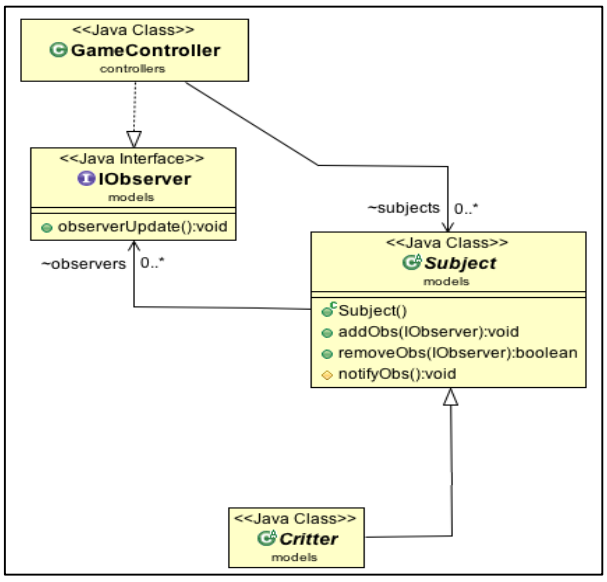

Figure 7 - Indirection of Subject

2) Principle Name: Information Expert

Information Expert pertains to what classes should know about other classes, and what they should be able to do. For instance, the Map does not need to know anything about the critters or the towers that are placed on it, but the Critters need to know about the Map's path.

This principle is used on multiple occasions throughout our code to ensure that we do not have any upwards dependencies, and that classes know as little as they need to in order to still function correctly.

Sample Problem: In the Tower Defense Game which class should be responsible for knowing the information of how a Tower shoots a Critter?

Solution: Towers and critters are the classes that are responsible for knowing their respective list of activities. The tower and critter instances are their own respective information experts. This means that when a Tower shoots a Critter, it does not directly affect the Critter's attributes. This is because the Tower does not know about the Critter. Instead, the Tower shoots a critter, which results in triggering a damage method in the Critter. The Critter then handles this damage accordingly.

3) Principle Name: High Cohesion

A class is said to be highly cohesive if most of the methods in its class are connected or similar.

Problem: How do we ensure that our classes are highly cohesive?

Solution: Most of the classes used in this game are highly cohesive, and this is especially true for the helper classes. By creating these helper classes, we can ensure that they only handle one subject (like the clock or the drawing). This results in highly cohesive classes.

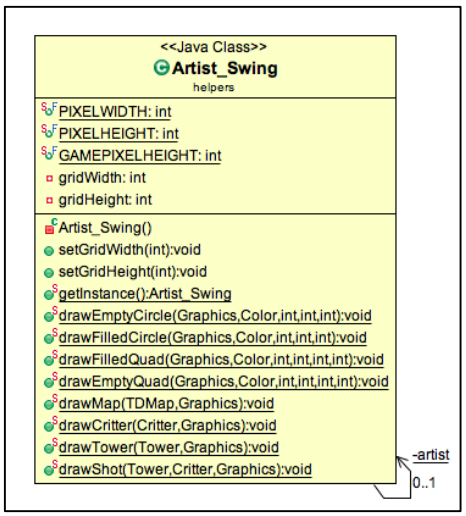

Figure 8 - Artist High Cohesion

One can see that all of the methods in Artist_Swing (apart from the setters) are related to drawing. This is a very highly cohesive class.

4) Principle Name: Polymorphism

Polymorphism is when classes may have variations in their behavior based on their type, even if they all extend the same superclass.

Sample Problem: In the Tower Defense game how are Towers of nearly identical attributes yet varying attribute values and slightly different methods handled?

Solution: The Tower Defense game requires the creation or use of different types of Towers. These towers will have almost identical variables and methods, but they need to have slight differences. For instance, a fire tower must act like all of the other towers, but it must also be able to set Critters on fire, and damage them over time. This is when the principle of polymorphism is applied. With the implementation of a Tower superclass, every Tower type does not need to be referenced individually and the creation of additional Tower types is simplified since all Towers inherit attributes from the superclass. Additionally, towers can be created that have slightly different methods (like the setting on fire).

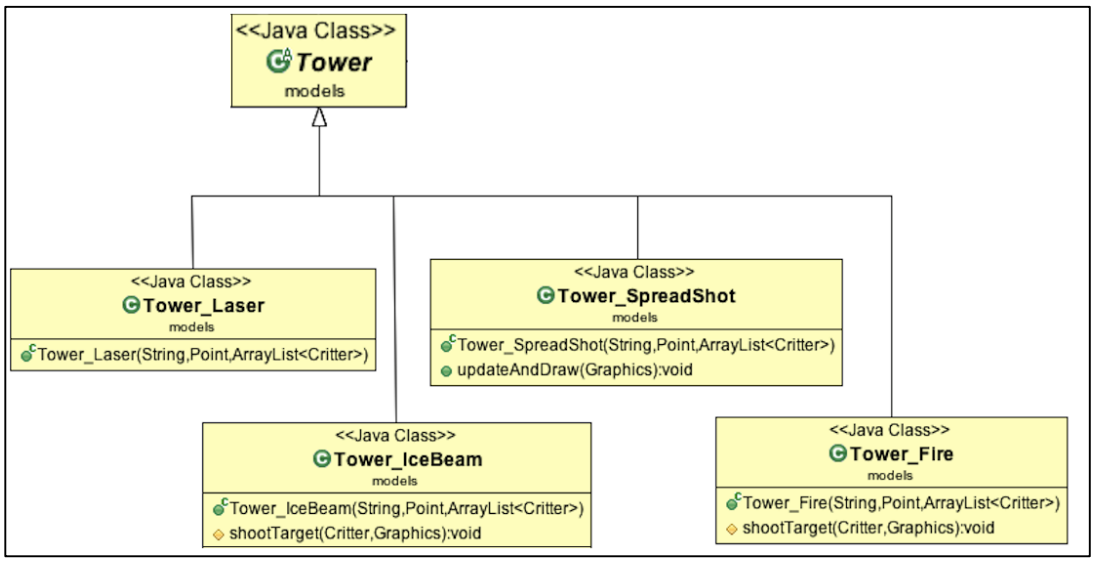

Figure 9 - Tower Polymorphism

One can see that the spread-shot tower, the fire tower, and the ice beam tower all have "extra" methods that override some of the standard Tower methods. This allows them to shoot more than one critter, apply damage over time, and slow critters, respectively.

5) Principle Name: Creator

The Creator principle decides which classes are responsible for creating other classes. For instance, it would not be appropriate for the Tower class to be responsible for the creation of the Map, or even vice versa.

Sample Problem: In the tower defense game, who should be responsible for creating instances of the critter class?

Solution: There is a specific class for creating critters, called the Critter Generator. This class is the only class that is allowed to instantiate critters at runtime. The game controller is the class that is responsible for creating the generator (and is its Creator), and the generator creates the Critters. One reason why this is helpful is so that the creation of classes happens in a minimum amount of places. For instance, the creation of critters only happens in the generator, so if we want to change what critters show up, we just change the generator class. Note that having a generator class rather than generating them in the game controller directly is another instance of the indirection principle being applied.

6) Principle Name: Controller

Problem: In the tower defense game, what class should be responsible for mouse and keyboard input events?

Solution: There is a dedicated class for handling all mouse related input events named MouseAndKeyboardHandler. This class is used by the GameController, in the business logic layer, and thus handles input to the GameApplicationFrame and GameControlPanel in the presentation layer. Allowing all input related events to be handled by a controller class has allowed the implementation of input related events in the presentation layer without any downward dependencies. This is also an example of the **indirection principle** (having a separate class to handle these inputs over using the GameController directly).

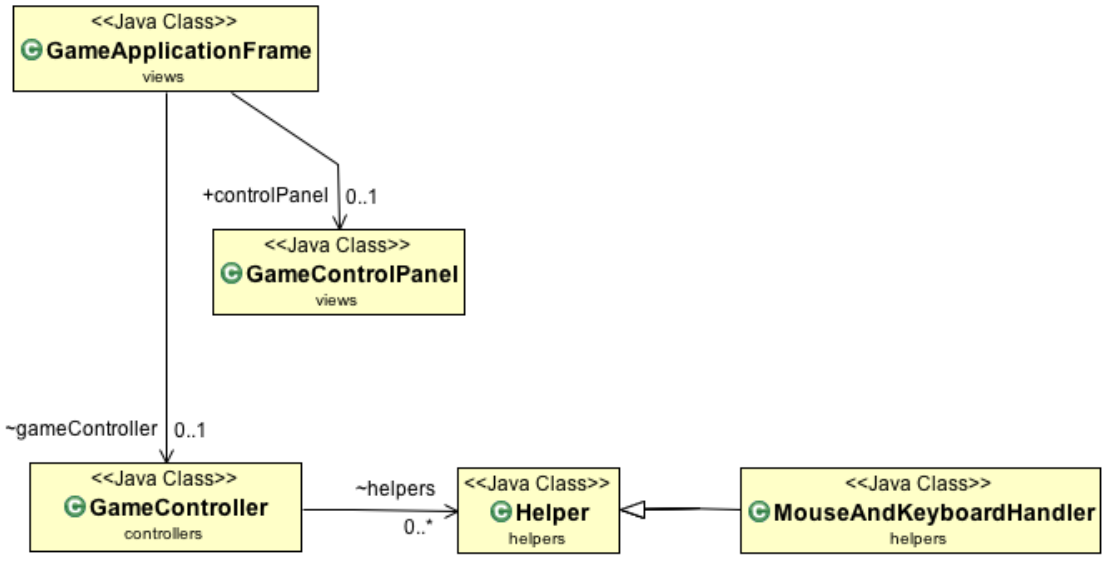

Figure 10 - Controller

Design Patterns

1) Pattern name: Singleton

Sample Problem: In the tower defense game how do we ensure that we create one (and only one) instance of the Player, Artist_Swing, and GameClock class while still letting these classes communicate and interact with other classes?

Solution: We apply the Singleton pattern, which ensures that there is only ever one instance of these classes. This instance is created when the program starts (via a private constructor), and can only be accesses through the public getInstance() method.

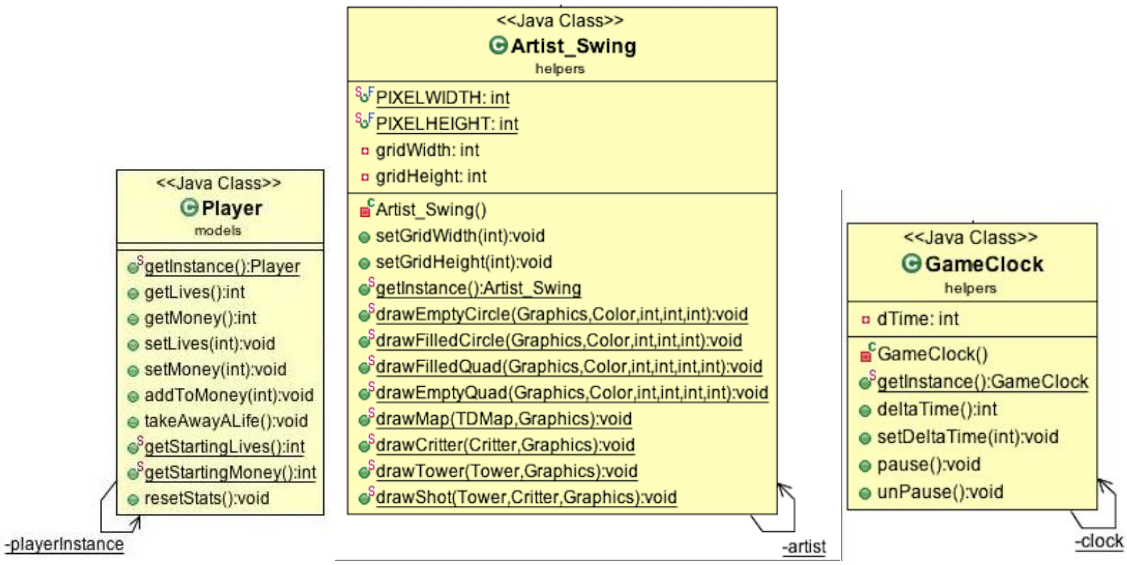

Figure 11 - Singleton Classes

i) Artist_Swing:

It is crucial that the Artist_Swing has only one instance of itself because we need to access it from any class that implements DrawableEntity (TDMap, Critter, and Tower). We do not want to create a different artist for each one, and have to pass in information about the map each time. Instead, we get the instance in our GameController, and pass in the information there. Following this, we can easily get that instance from any other class (TDMap, Critter, or Tower) and use its methods. This demonstrates the power of this pattern.

ii) Player:

It is important that the Singleton pattern is applied to the Player class since there should only be one player who can be created. This cannot be a static class because it needs to have variables (lives and money) that can be altered.

iii) Game Clock:

The Game Clock is another class that relies heavily on the Singleton pattern. It is similar to the Artist_Swing class, as it too needs to be called from any object that is time-dependent (Critter and Tower), but also from the controller. Without the Singleton pattern, there could be many instances of the clock, and one would have to alter each clock if he/she needed the time to be synced. With the Singleton pattern, when the rate of ticking is altered in the game controller, the Critters and Towers are immediately affected. This allows for cool features like the pause button (calling the Game Clock instance's pause()) method, and speeding up the wave (calling the Game Clock instance's setDeltaTime()) method.

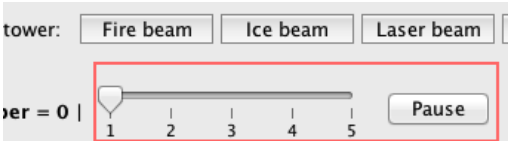

Figure 12 - The UI for changing the speed of the Wave

2) Pattern name: Observer

The subject observer design principle is used to notify classes that are higher in the class hierarchy without creating an upwards dependency.

Sample Problem: In the tower defense game, how do we ensure that the Game Controller is updated every time there is an important change to the Critter, like reaching the end or dying?

Solution: The implementation of the IObserver interface by the game controller, and the inheritance of the Subject class by the Critter solve this problem. First, the gameController is made to be an observer of the Critter. Then, when the Critter changes, it notifies all of its observers. The method UpdateObservers is called in the game controller, and it is able to check if something changed with the critter. If the Critter has reached the end or died, the game controller can update the Player's lives or money, respectively.

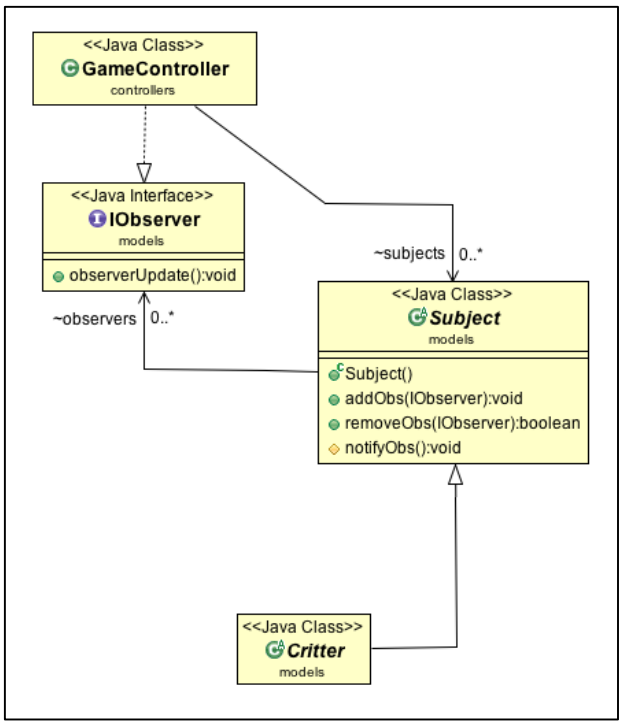

Figure 13 - Observer Pattern

3) Pattern name: Strategy

The strategy pattern is a design pattern where something needs to be done, but it can be done in many ways.

Sample Problem: In the tower defense game how do we implement more than one strategy for a Tower selecting a target Critter?

Solution: Rather than hard-coding in all of these ways, one interface (IStrategy) is created, and all of the different ways of doing it (strategies) implement this interface. The Tower class then has an IStrategy object, which could be any of the strategies. Each of these has the method that selects the target critter, and the Tower calls this method to select the critter it wants to target, without knowing explicitly how to choose a target.

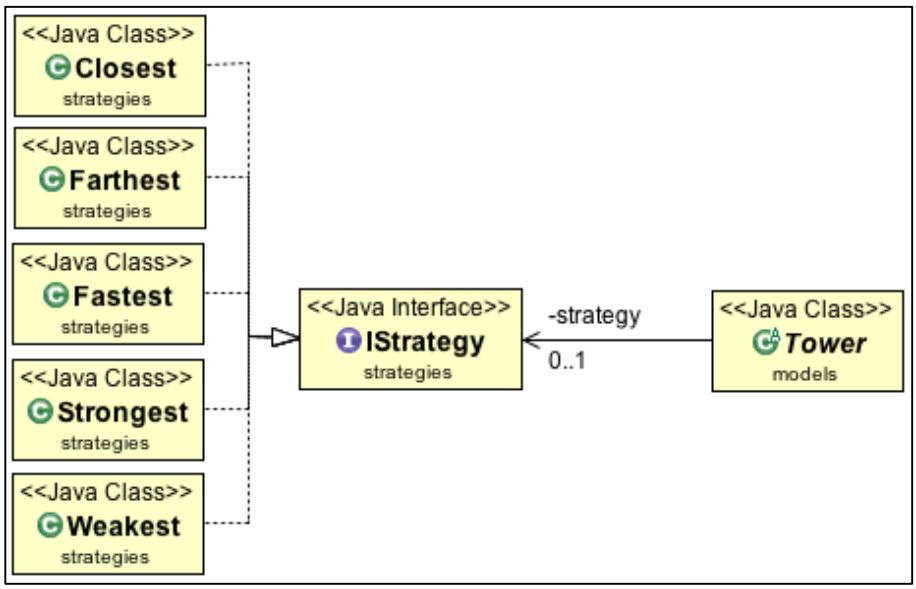

Figure 14 - Strategy Pattern

Sequence Diagrams

Sequence diagrams have been created for the four architecturally significant use cases:

1. Playing the Game, which would encompass all the other use cases, as the ulterior motive for the user.
2. Creating a new Game Map,
3. Preparing for a Wave,
4. Playing a Wave.

1. Playing the game

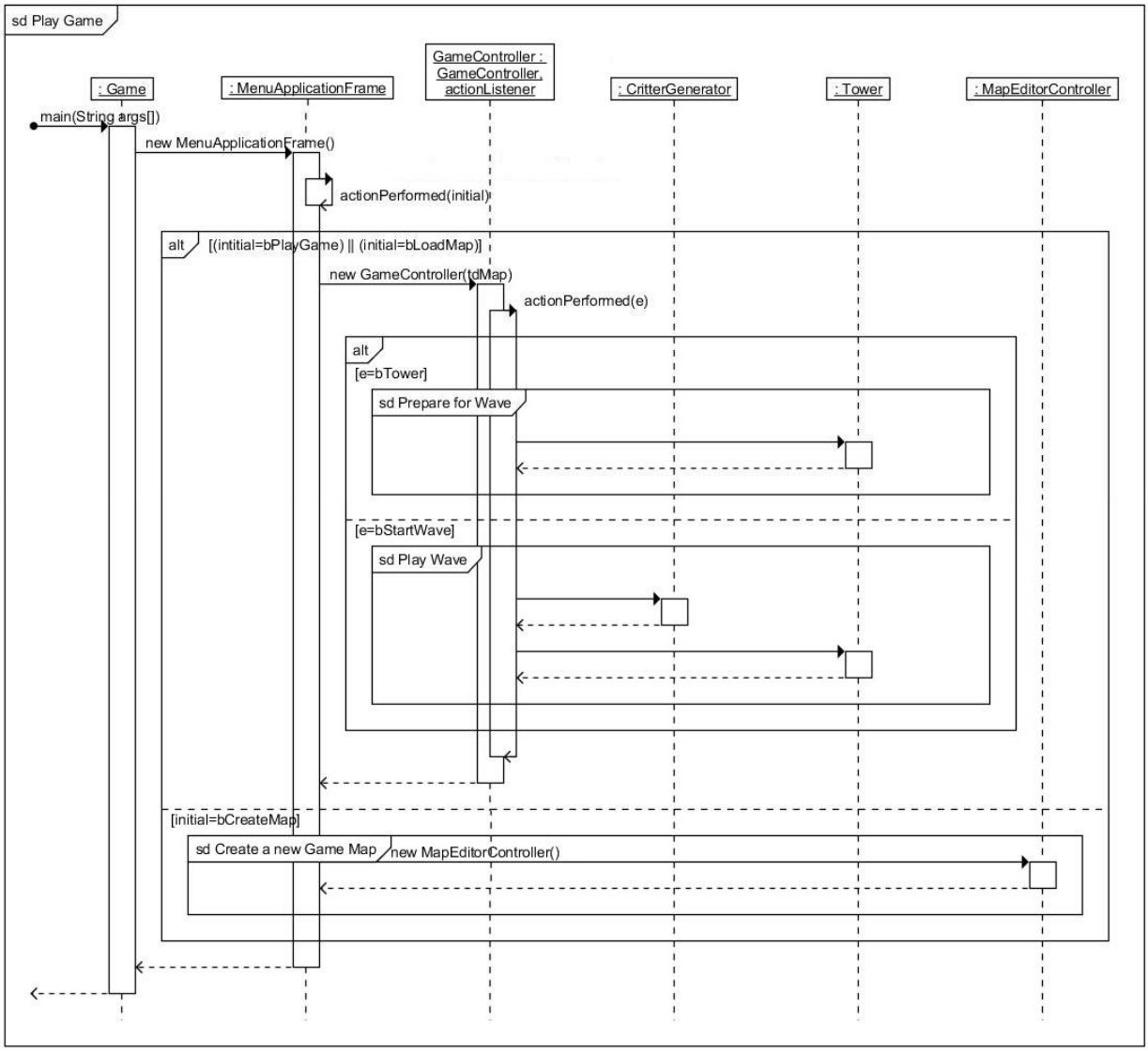

Figure 15 - Sequence Diagram Play Game

This use case displays the architectural implementation of how the different layers of the system interact with each other and the user to achieve user's goals.

2. Creating a new game map

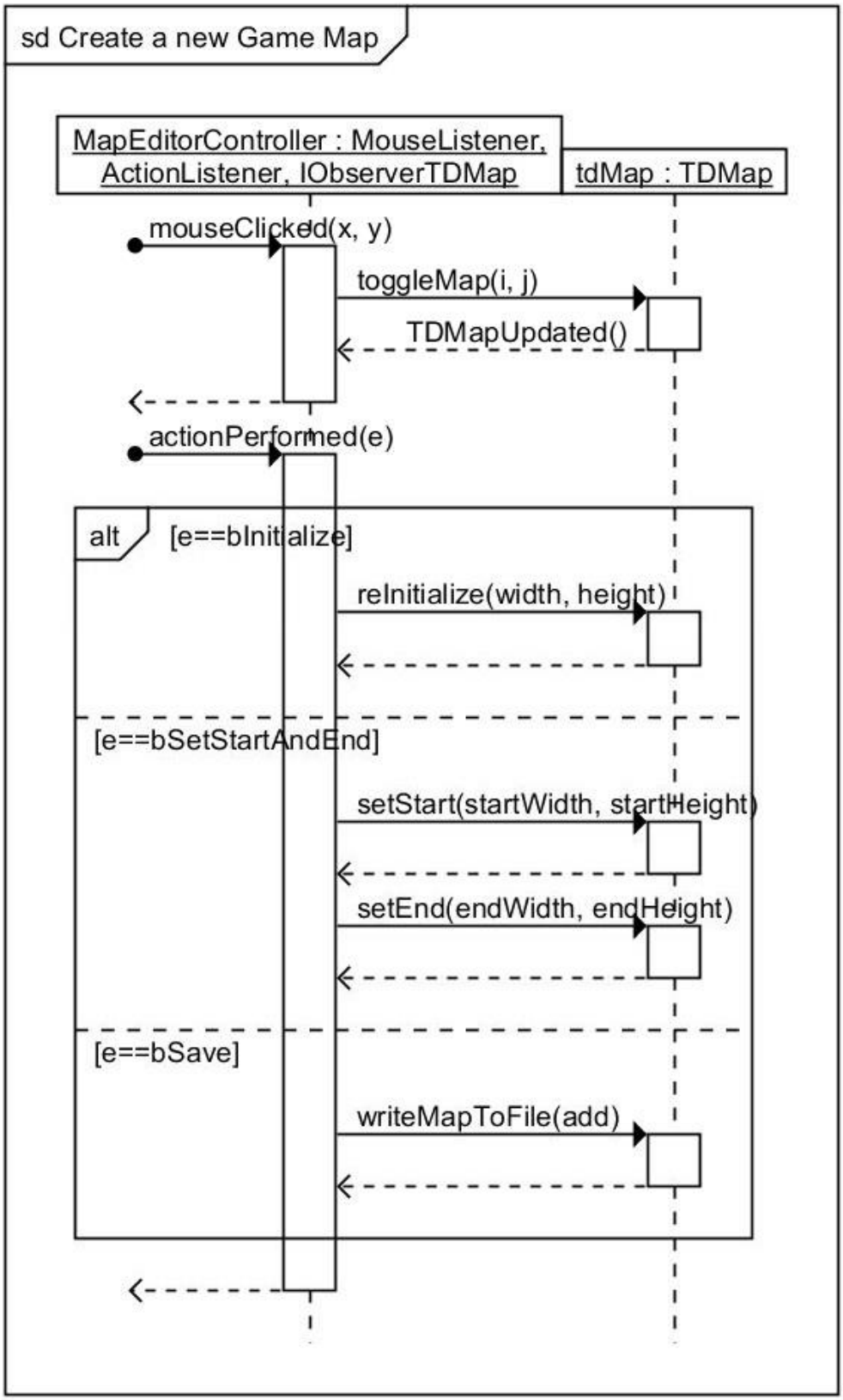

Figure 16 - Sequence Diagram New Map

3. Starting a wave

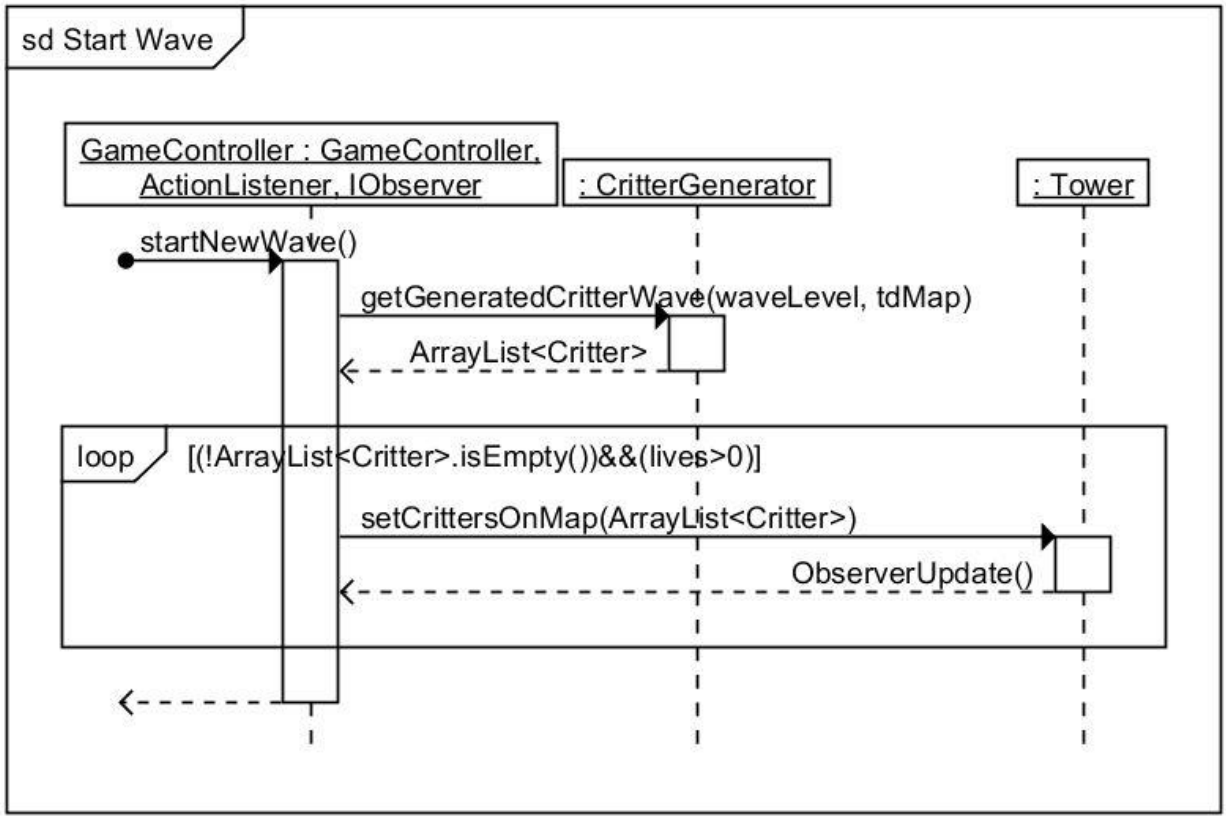

Figure 17 - Sequence Diagram Start Wave

The architectural sequence diagrams, Figures 16, 17 and 18, show in greater detail, the user specific use cases that make up the ulterior use case of playing the game. The choice of choosing the use case of making a new Game Map is that the user would like to make his own maps that would moderate the difficulty of the game, according to the user's preferences. The user-level use-cases of Preparing for a Wave and Starting a Wave are fundamental to the Tower Defense Game in particular.

4. Preparing for the wave

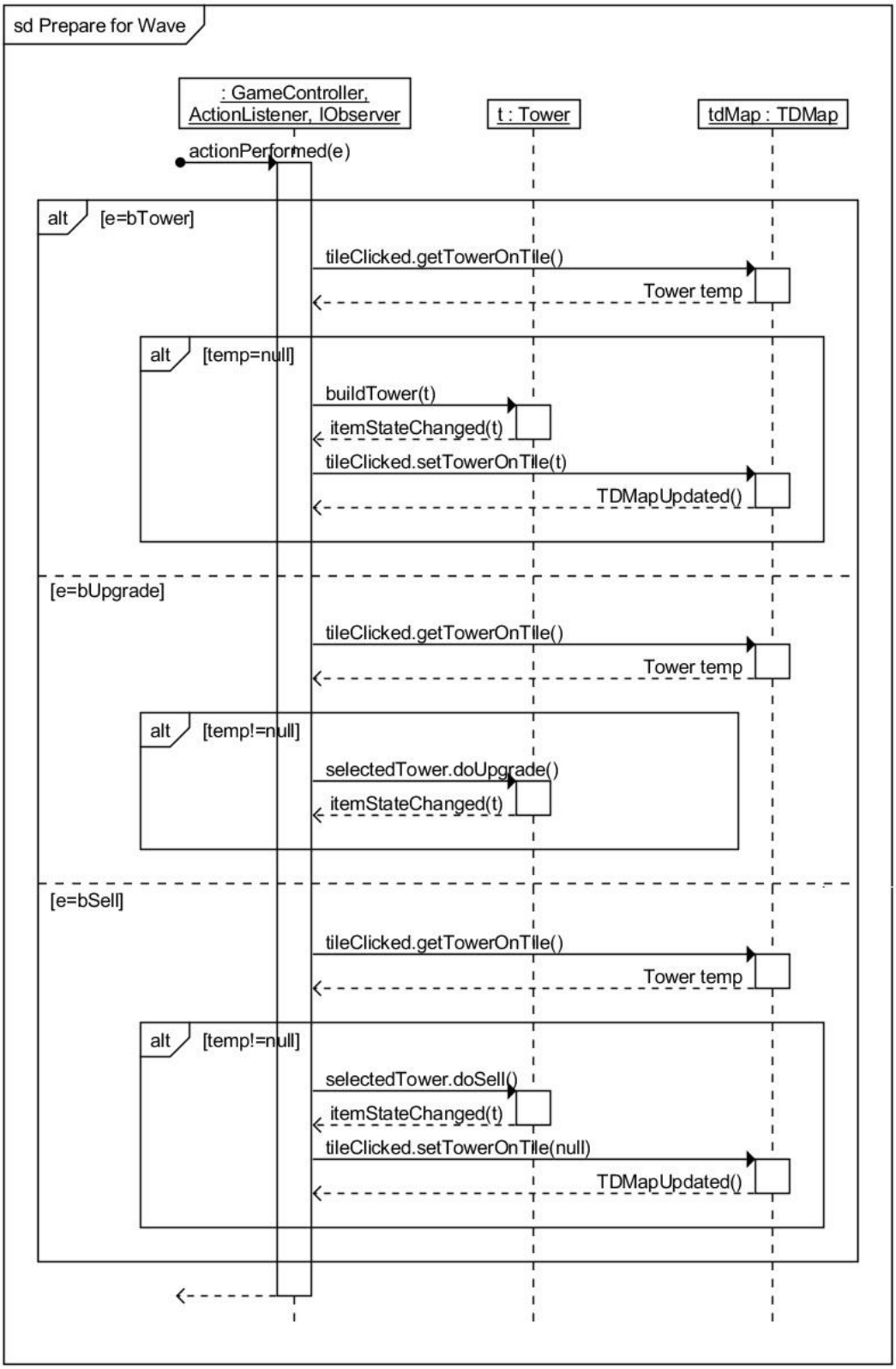

Figure 18 - Sequence Diagram Prepare