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 |
|  |  |  |  |

Table of Contents

**Introduction**

The purpose of this document is to outline in detail the software architecture and design of the “Tower Defence 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 that impact the system and will illustrate the system’s structure and functionalities through static and dynamic views of the system.

**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**   
  
While this document intends to give significant insight into the system, its scope is limited by the fact that it is a part of an educational project. Specifically, it is a requirement of this project that only three sequence diagrams are presented, depicting three architecturally significant Use Cases from the related Software Requirements Document (see Related Documents). Dynamic views of the system will be limited to these three diagrams

**Related Documents**

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

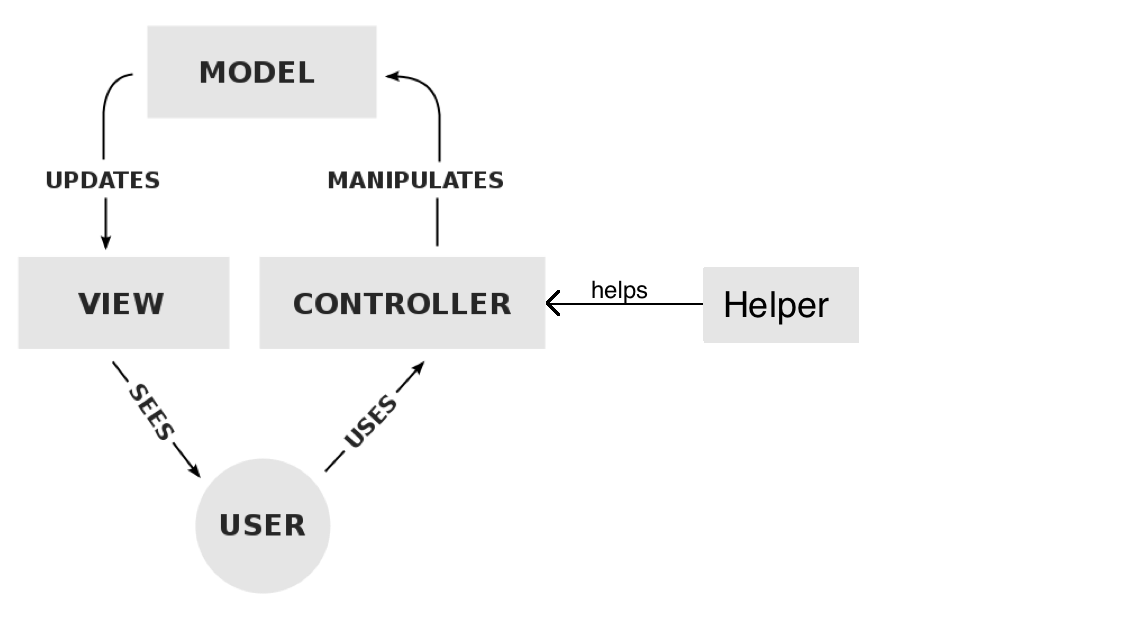
Documents that result from this document: “Yash’s documentation” Java doc.

See also

System Architecture

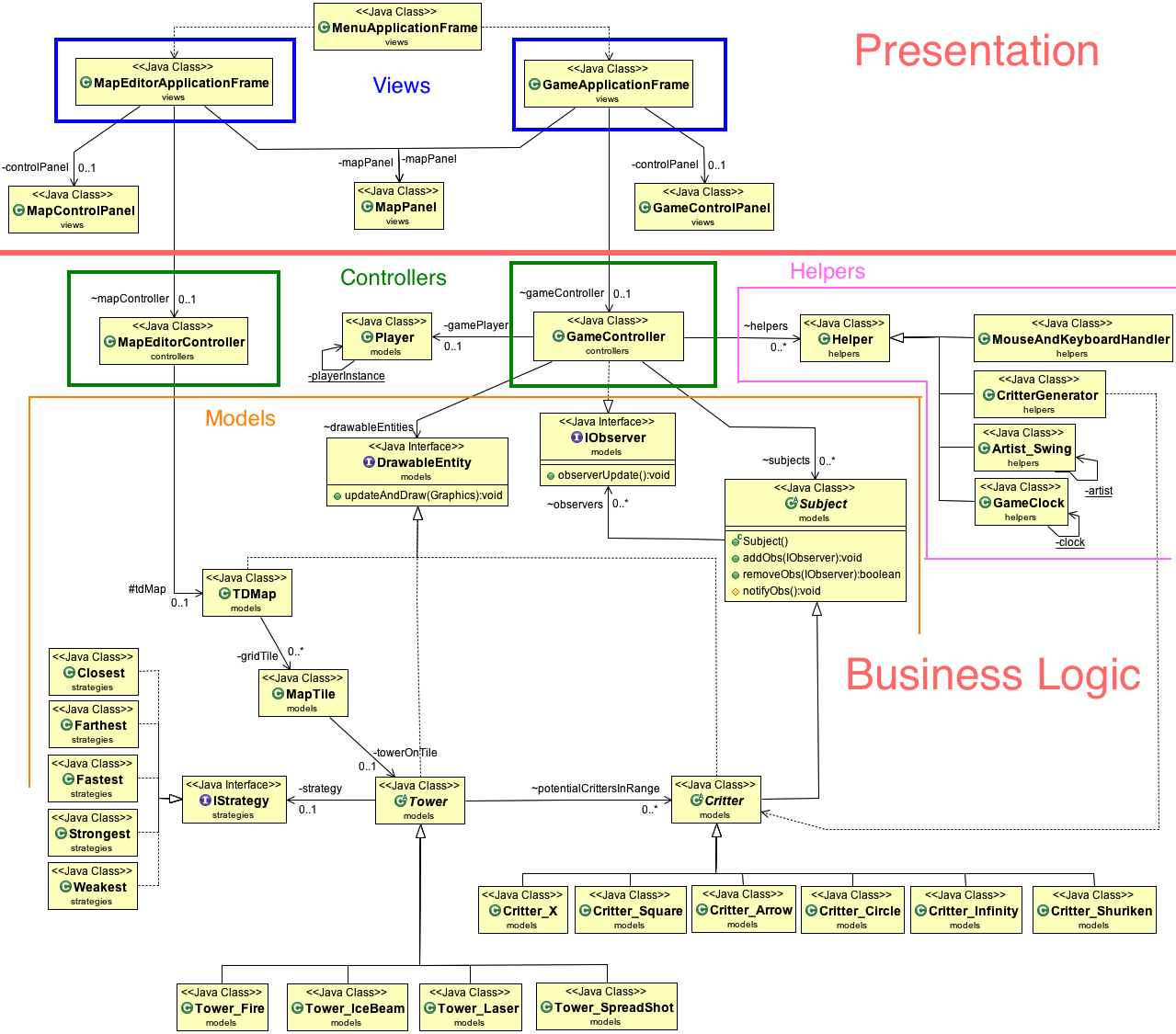
The Tower Defense game implements a layered, multitier, architectural strategy. This architecture is based off of the typical client-server three-tier architecture, which has a presentation layer, a domain logic layer, and a data storage layer. Since the game does not store or use much data, the third 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 showed 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.



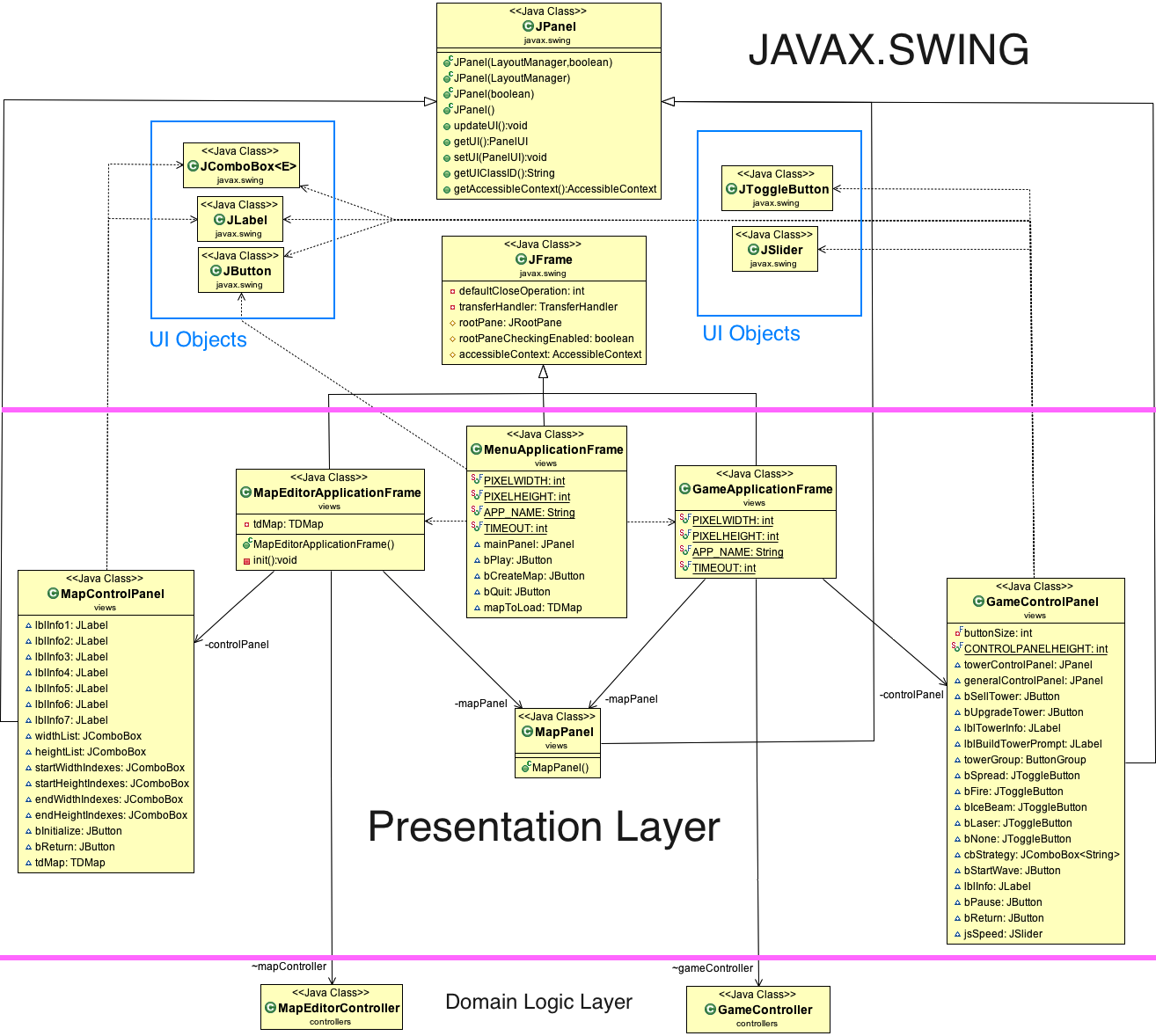
Architecture Diagram

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



Although this first diagram may look complex, it actually demonstrates very low coupling, and good organization. Whereas without the MVCH layout, there may have been seemingly unnecessary connections, here all of the connections are 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. This is much cleaner than having all of these subclasses connect directly to the GameController.

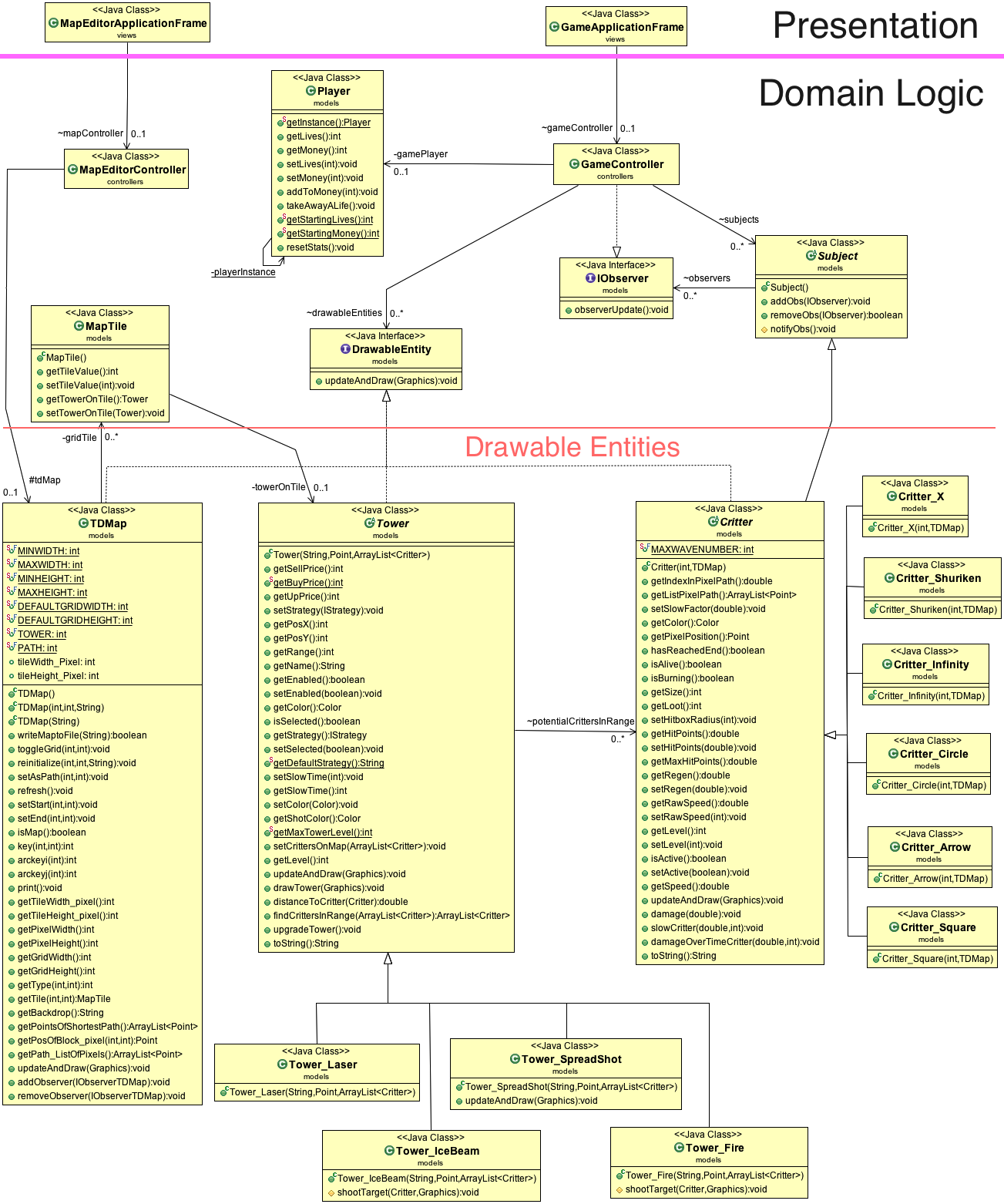
Below is the presentation layer with all of the methods and attributes shown.



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.

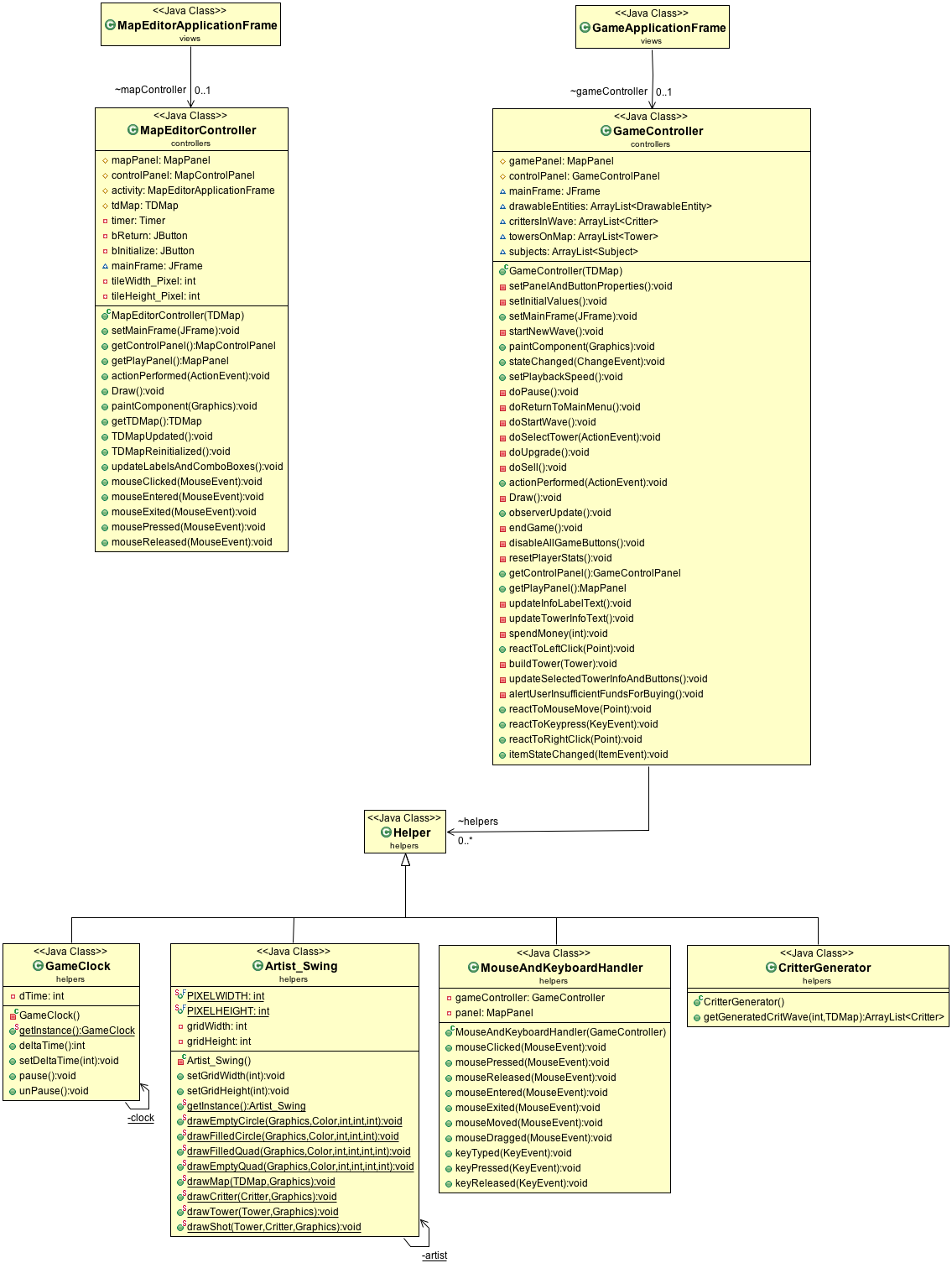
Domain Logic Layer

1. Drawable Entities

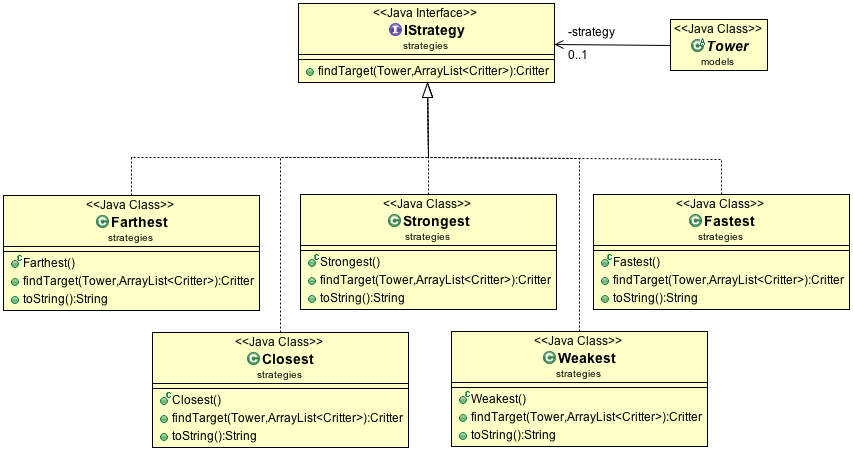


Note that the attributes of the Map, Tower, and Critter were not shown, as it created messy diagrams. In addition, it is redundant, considering that there are getter and setter methods that are shown.

1. Game Controllers and Helpers



1. Strategies



This design principle will be discussed below in the principles section.

**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.

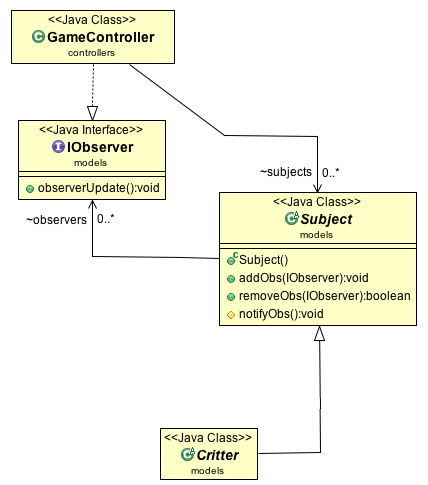
One result of using these principles and patterns is that it is relatively simple to add on to code, or to alter it. For example, our group initially decided to use the Light Weight Java Game Library (LWJGL) to display the game. Due to information expert and in a goal to create high cohesion, we had 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

**Problem:** In the tower defense game, who should be responsible for adding removing or notifying observers?

**Solution:** Instead of directly implementing the add remove and notify observer methods in critter, a critter extends a separate subject class that includes the list of IObservers and all associated methods. This lowers the coupling between observers and critters and allows the potential for other classes to be observed by having them extend the Subject class.



**2) Principle Name:** Information Expert

**Notes:** This principle is one that is used on multiple occasions throughout the coding process ho

**Problem:** In the Tower Defense Game which class should be responsible for knowing the activity information of a tower or critter instance?

**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 results in low coupling creating potential for modularity.

(Optional Diagram)

**3) Principle Name:** High Cohesion

**Problem:**

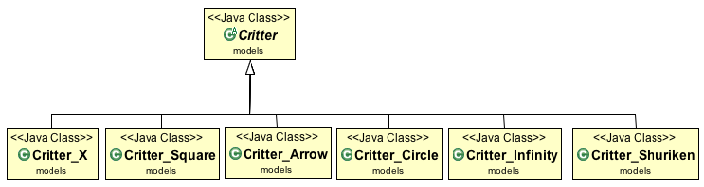
**Solution:**

(Optional Diagram)

**4) Principle Name:** Polymorphism

**Problem:** In the Tower Defense game how are critters of identical attributes yet varying attribute values handled?

**Solution:** The Tower Defense game will require the creation/use of multiple critter “types” and as such, it would impractical to reference critter “types” individually when the demand to reference every critter is required. In this situation the Polymorphism principle is extremely applicable and recommended. With the implementation of a critter “superclass”, every critter type does not need to be referenced individually and furthermore the creation of additional critter types becomes significantly simplified has critter all critter types inherit attributes from the critter superclass. A beneficial side effect of applying this principle is the reduced coupling it produces within the critter hierarchy as the individual critter types are very rarely if ever referenced instead of the critter superclass.



**5) Principle Name:** Creator

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

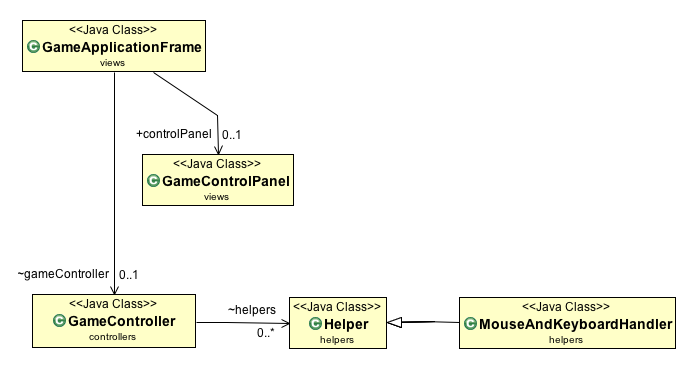
**Solution:** A creator class, in the form of gameController, is responsible for the instantiation of all towers and critters. The gameController contains towers and critters and as such has the authority to instantiate them demonstrating an upwards dependency.

(Optional Diagram)

**6) Principle Name:** Controller

**Problem:** In the tower defense game, who 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 gameController, in the business logic layer, and consequently by GameApplicationFrame and GameContolPanel, 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.



**Patterns**

**1) Pattern name:** Singleton

**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.

**i) Artist\_Swing:**

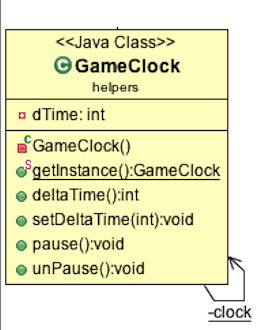
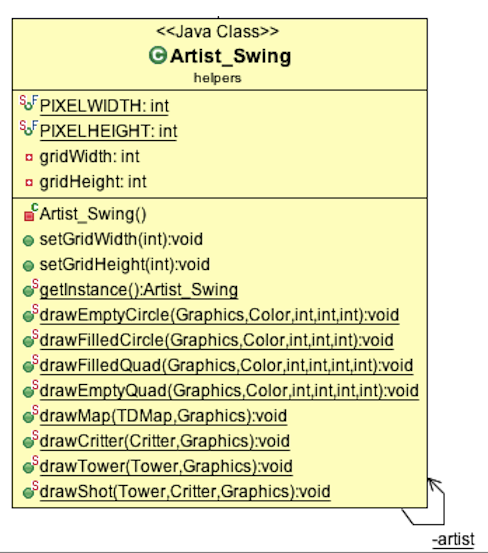
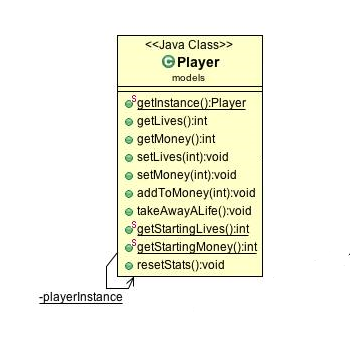
It is crucial that the Artist\_Swing has only one instance of itself because we need to access it from any class that implements DrawableEntity (TDMap, Critter, and Tower). We do not want to create a different artist for each one, and have to pass in information about the map each time. Instead, we get 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.

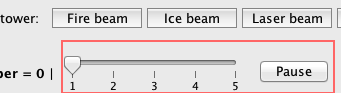
**ii) Player:**

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.

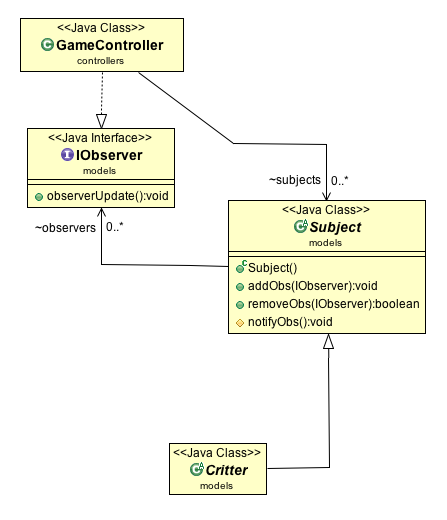




**2) Pattern name:** Observer

**Problem:** In the tower defense game who is responsible for allowing the gameController to be updated every time a change occurs in the critter class?

**Solution:** The implementation of the IObserver interface and the subject class allow the gameController to be informed every time a critter moves or becomes damaged. The critter class extends the subject class and the gameController implements IObserever thus acting as an observer for critters.



**3) Pattern name:** Strategy

**Problem:** In the tower defense game who is responsible for the various strategies that can be applied to the tower class resulting in a change in targeting patterns?

**Solution:** The interface IStrategy is responsible for all strategies used by the tower. The strategy classes themselves, each with different targeting patterns, implement the strategy interface.

