1. **Name** - Ethan Parks
2. **Project description.**

   Title: Tower Defense

   Enemies enter the screen on one side and follow a set path through the map and off the screen. The player must use currency generated from defeating aliens to buy and place towers on the map which work together to defeat the aliens before they exit the map. The player may choose from upgrades to the towers to make them stronger or change their effects. The player receives points towards a high score upon defeating an enemy. When an enemy exits the screen, the player loses a life. The player runs out of lives, the game is over.

3. **List the features that were implemented (table with ID and title).**

Features implemented	
ID	Title
IF-01	Select an arbitrary map to play
IF-02	Users can create their own maps
IF-03	Place towers on the map
IF-04	Upgrade towers
IF-05	Pause game
IF-06	Tower placement undo/redo
IF-07	Tower upgrade undo/redo
IF-08	Multiple types of enemies
IF-09	Enemies spawn in waves
IF-10	Restart game
IF-11	Save game
IF-12	Load game
IF-13	Score tracking
IF-14	Score display
IF-15	Save highscores
IF-16	View highscores
IF-17	Currency system
IF-18	Add/remove currency as admin

IF-19	Spawn enemies as admin
IF-20	Command line interface

4. List the features were not implemented (table with ID and title).

Features not implemented		
ID	Title	Reason Not Implemented
UF-01	5 types of enemies	Only two types of enemies were needed to demonstrate the design patterns.
UF-02	5 types of towers	Tower projectiles have different types of upgrades instead of different types of towers.
UF-03	Internet accessible highscores database	Listed as stretch functionality.
UF-04	Save game replays	Listed as stretch functionality.

5. Show your final class diagram.

Figure 1 shows the way the packages interact within Tower Defense. In order to fit the class diagram on to the page, the diagram is split into Models in Figure 2 and Controllers and Views in Figure 3.

Figure 1: Package Diagram

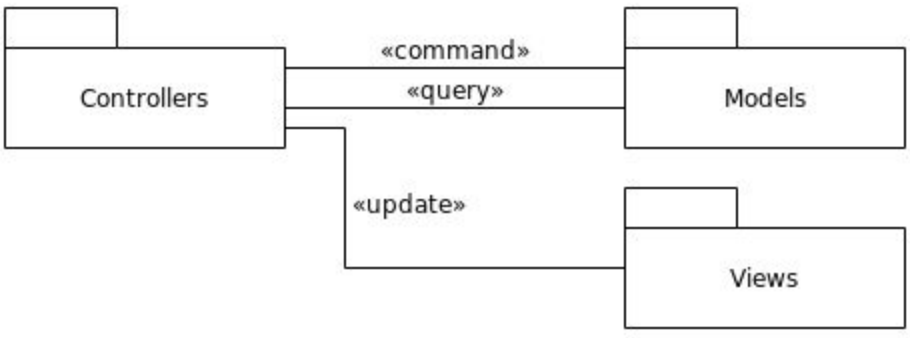


Figure 2: Models

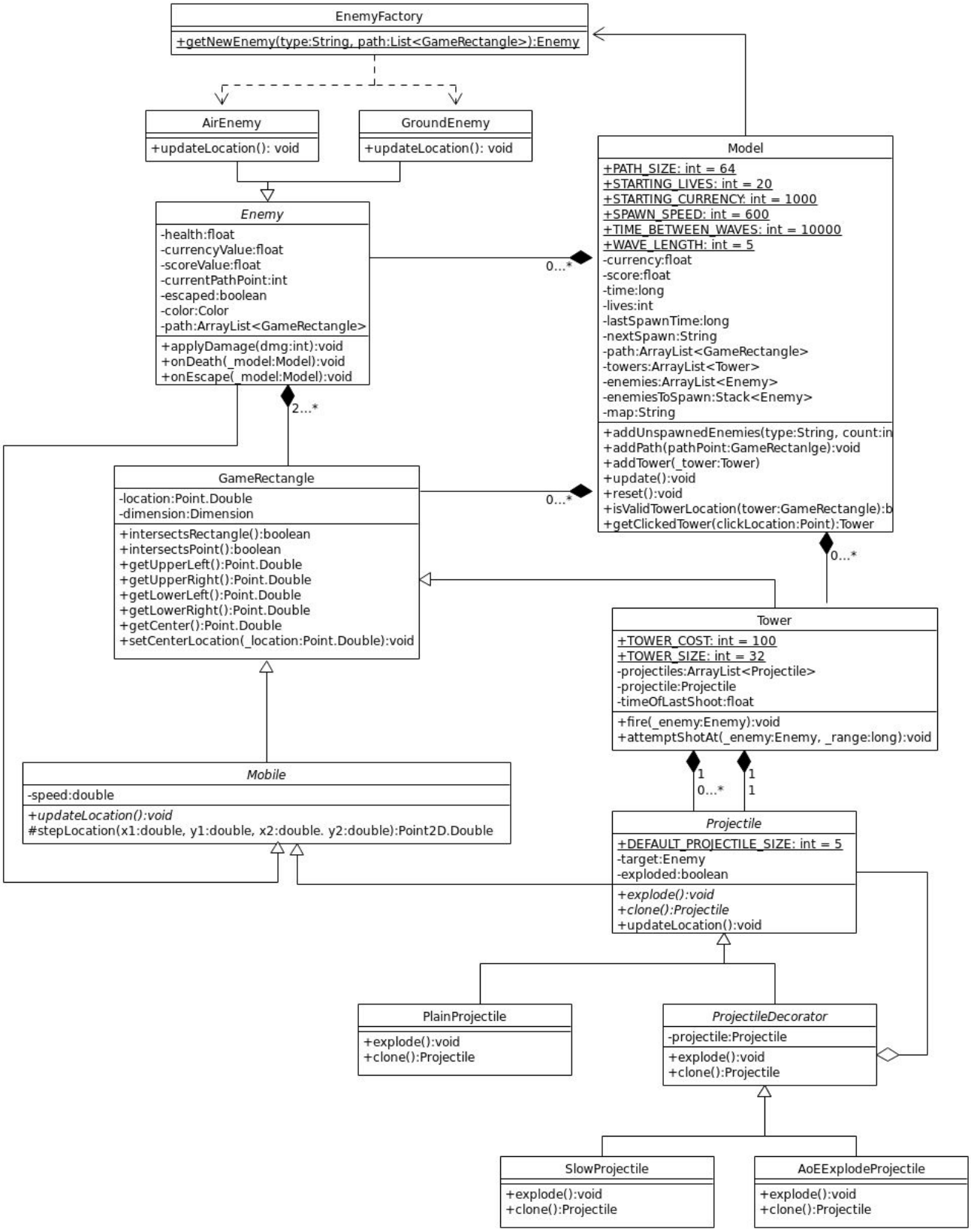
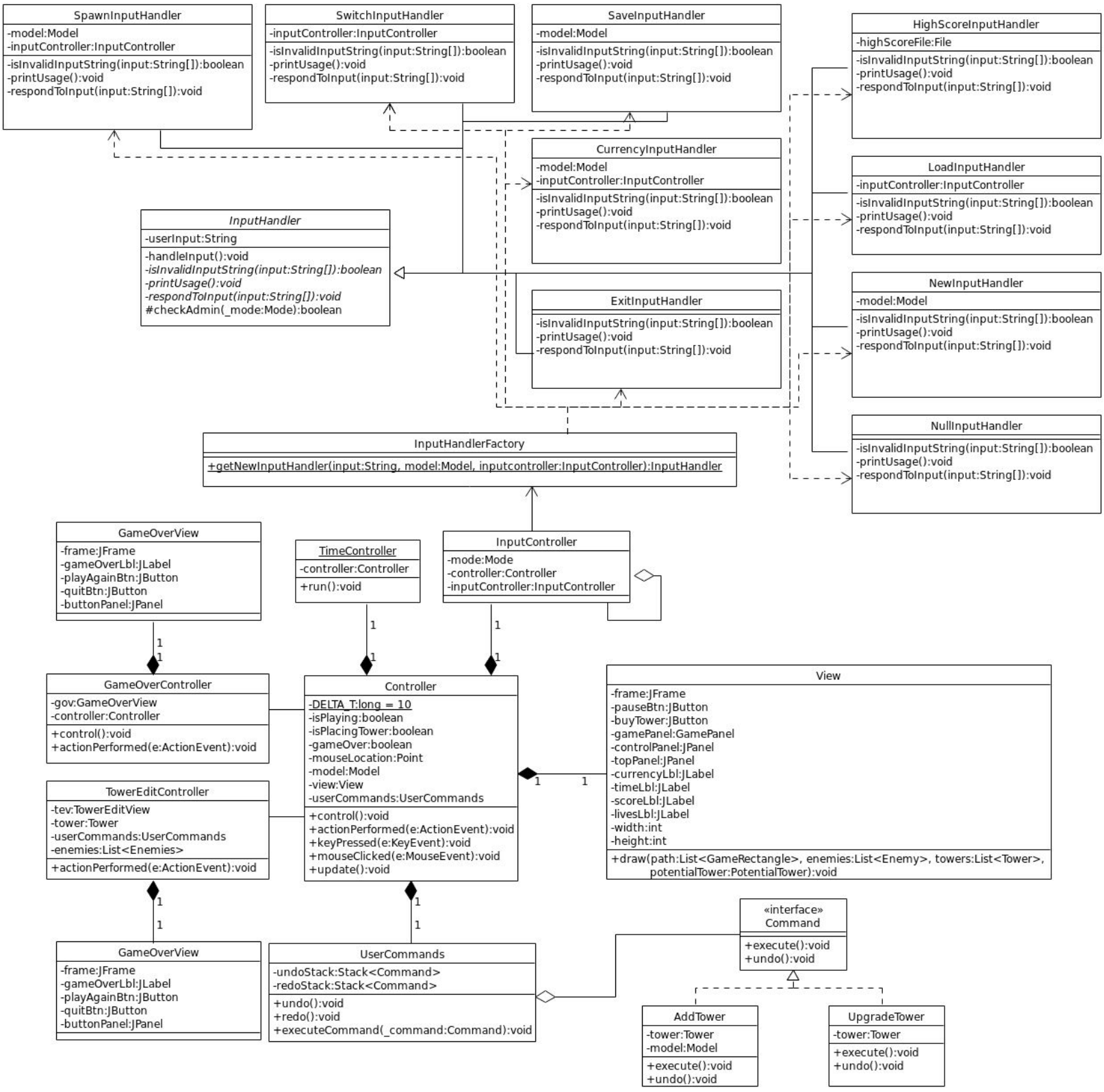


Figure 3: Controllers and Views



What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

The class diagram I previously made helped map out the direction of development. There were many updates to the class diagram, but the vast majority were additions to the original class diagram.

In the models, I made the enemy class abstract and extended by the different types of enemies. This allowed me to encapsulate the variation between enemies within their own classes. Now that there were different classes for each enemy type I needed a factory class to simplify the creation process. I also realized that each entity in the game (e.g. towers, enemies, projectiles…) needed to have a bounding box to track location and size and determine collisions. To do this I added the GameRectangle class which is extended by each entity in the game. During development it became apparent that the moving entities in the game (enemies and projectiles) needed much of the same code. To consolidate this shared code for movement I created the abstract Mobile class. The mobile class extends the GameRectangle in addition to adding on the method to move the entity at a given speed toward a target.

For the controllers and views, I realized that in addition to the main window I also needed a window for upgrading towers and a window to show when the game was over. To do this, I made a controller and view class for each window. This kept the main view and controller classes from becoming too large.

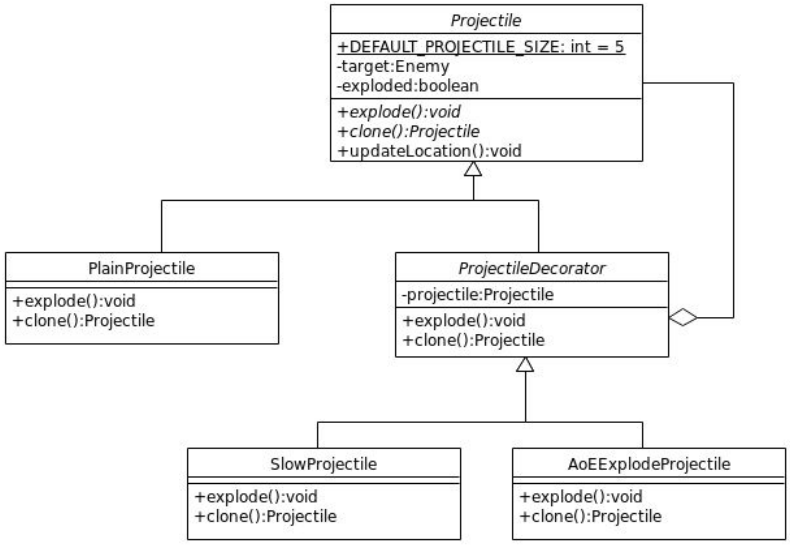
After making the original class diagram, I decided to use the command design pattern to implement undo and redo for tower creation and upgrades. To do this, I created the UserCommands class and the abstract Command class. The UserCommands class executes the commands and handles undo and redo requests from the user. The abstract Command class is extended by each type of command.

To keep the main Controller class from getting too large, I created the InputController class. The InputController class receives the command line input from the user and uses a InputHandler to handle the input. I added the abstract InputHandler class to use the template design pattern to respond to the command line input. The InputHandler class is extended by each class needed to respond to the different command line inputs. A InputHandlerFactory class was created to create the appropriate InputHandler based on the command line input.

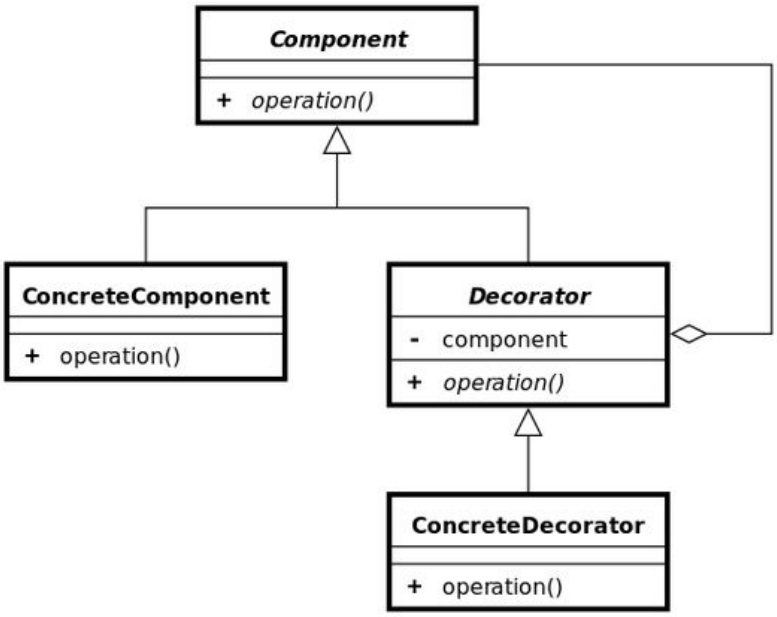
6. **For each design pattern implemented,**
    - **Show the classes from your class diagram that implement each design pattern.**
    - **Show the class diagram for the design pattern.**
    - **Explain how you implemented the design pattern, why you selected that design pattern.**

I. Decorator - Projectiles

My implementation:



Design Pattern:



I implemented the decorator design pattern to handle the different projectile effects. Each decorator applied to the PlainProjectile adds a new effect when the projectile explodes.

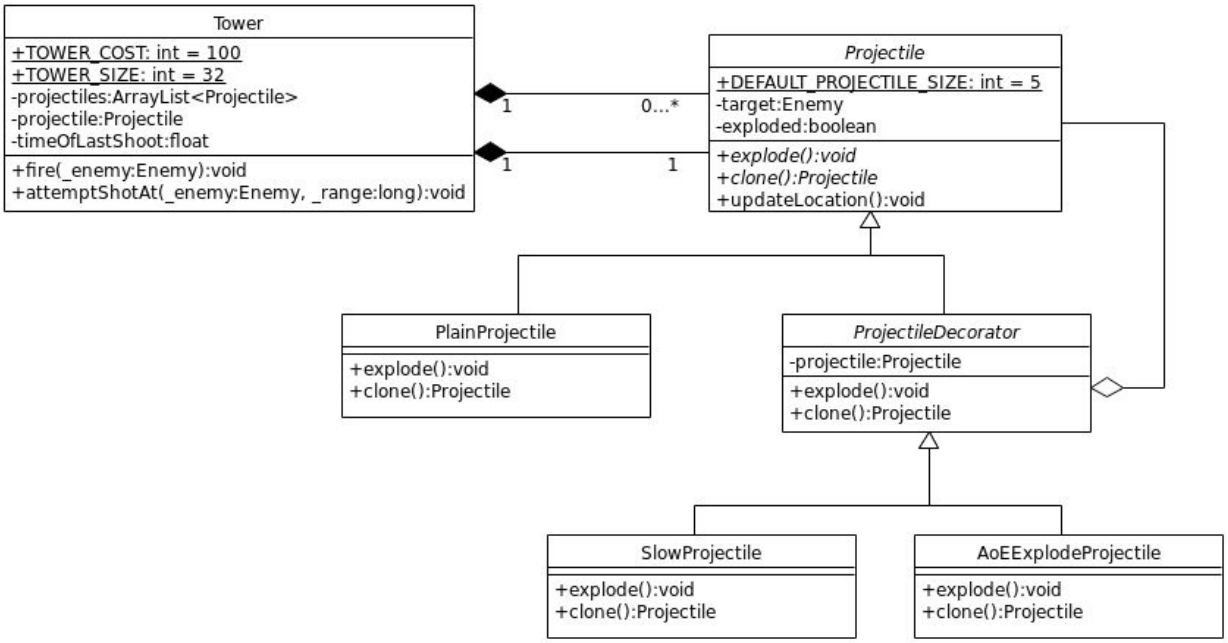
The Projectile class acts as the abstract component, the PlainProjectile class acts as the ConcreteComponent, the ProjectileDecorator class acts as the abstract Decorator class, and the SlowProjectile and AoEExplosionProjectile classes act as the ConcreteDecorator classes.

I chose the decorator design pattern because combining projectile effects intuitively mapped to the different stacking effects provided by the design pattern. It also allowed me to encapsulate the variation within explosion effects into separate classes. Without the decorator design pattern, all possible functionalities would have to be stored within a single Projectile class and a list would have to be maintained of which functionalities to apply. Using the decorator design pattern also made it easy to undo upgrades to a tower's projectile. To undo an upgrade to the projectile, simply remove the outermost decorator.

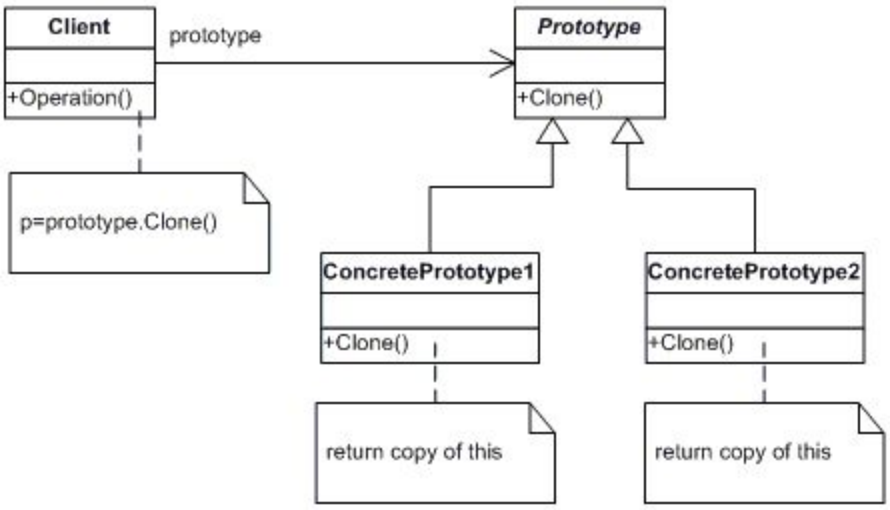
In addition to the explode method, I added a clone method to the projectiles so that they could be used in the prototype design pattern discussed below. To clone the decorators, the projectiles return a clone of themselves and pass the projectile they are decorating into the constructor.

II. Prototype - Projectiles

My implementation:



Design pattern:



I used the prototype design pattern to keep track of what type of projectile the tower should create when shooting. The tower clones the projectile and adds the clone its list of active projectiles.

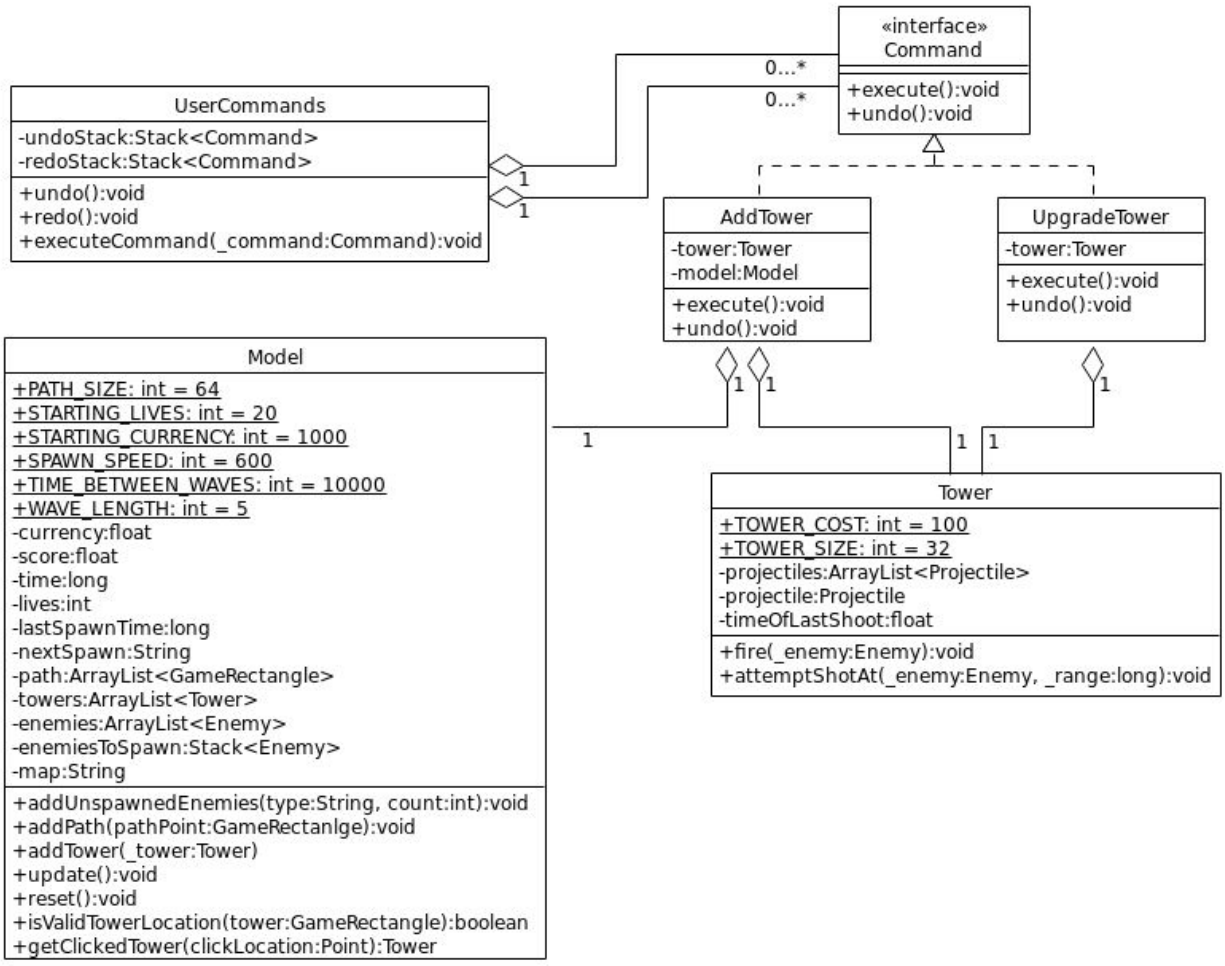
The tower acts as the client, the abstract Projectile class acts as the prototype and the SlowProjectile and AoEExplosionProjectile classes act as the ConcretePrototypes.

I chose to use the prototype design pattern for this case because the tower's projectile effect was changing over the course of the game. If I had not used the prototype design pattern I would have had to store flags within the tower for each possible type of decorator to apply to the projectile when creating a new projectile. As the system grew this process would not scale because of the large number of flags required.

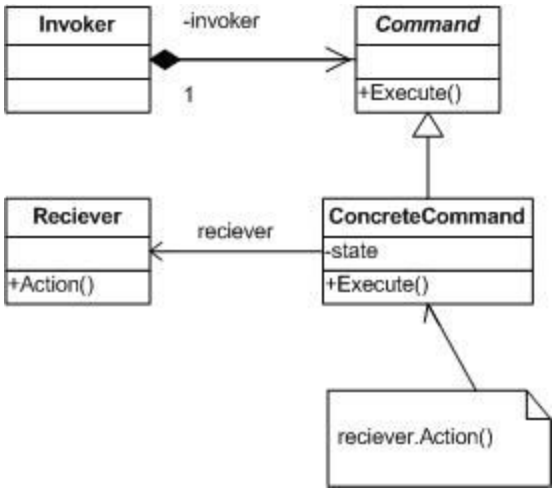
I use composition rather than association as seen in the design pattern because the tower has a projectile strictly for itself and the projectile cannot exist without its tower.

III. Command

My implementation:



Design pattern:



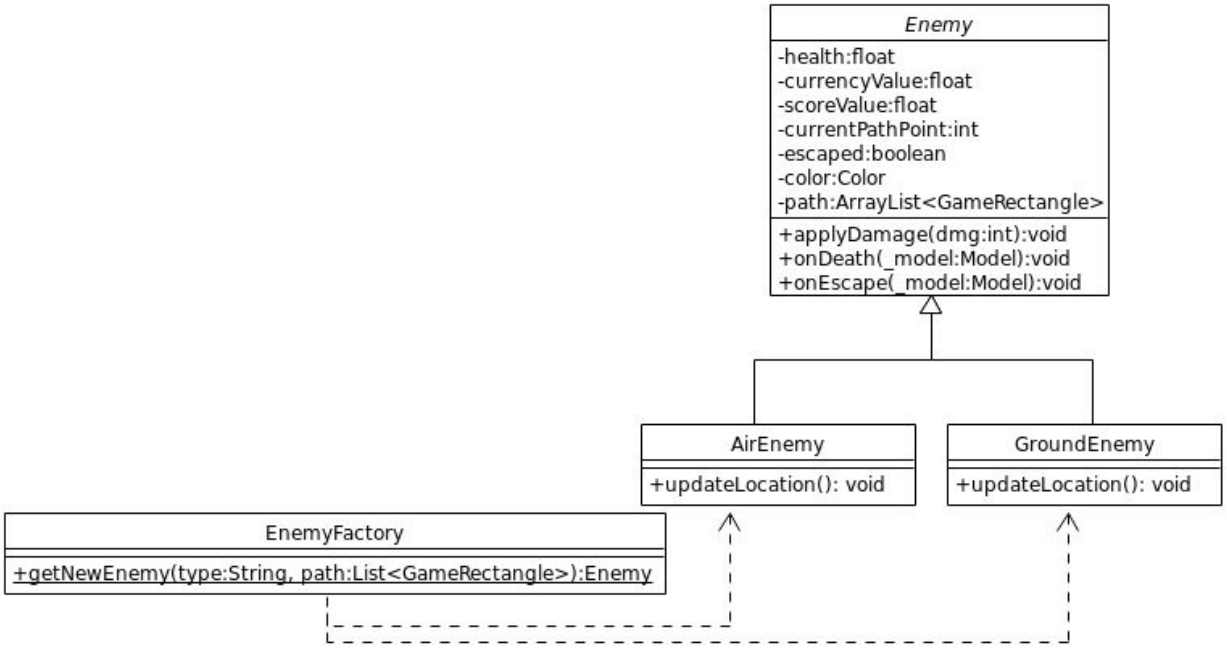
I implemented the Command design pattern to facilitate the undo and redo functionality.

The UserCommands class acts as the invoker, the Command interface acts as the class used by the invoker, and the AddTower and UpgradeTower classes act as the ConcreteCommands. For the AddTower command, the receiver is the Model class because the list of towers within the model is updated. For the UpgradeTower command, the Tower class is the receiver because the projectile within the tower is updated.

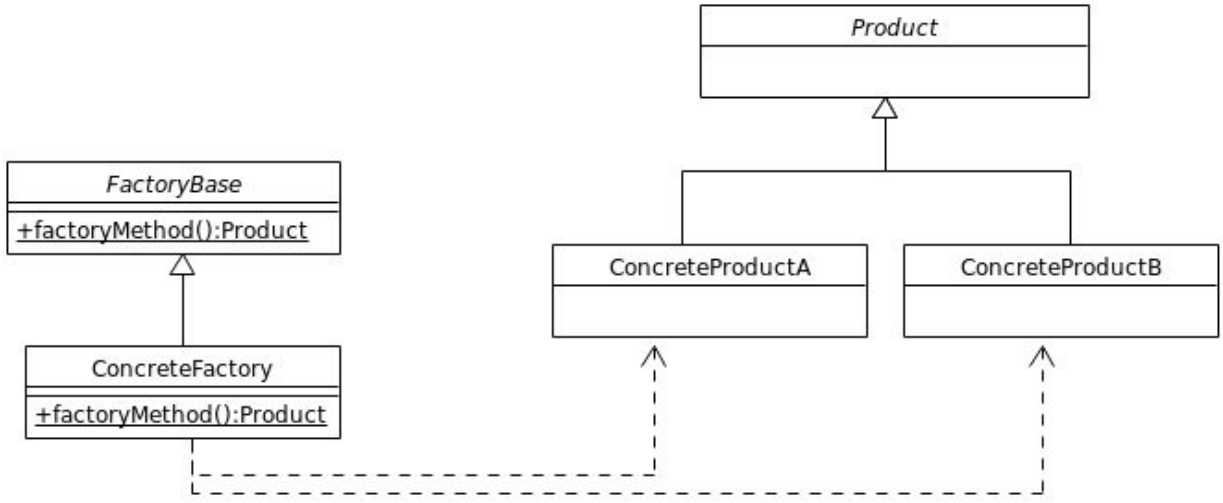
I chose to use the Command design pattern because encapsulating each command as an object allowed for easy tracking of the commands. The client code can easily undo the last command by popping it off the stack in the UserCommands class and calling the undo method within the command object. Having each command in their own object allows the encapsulation of the differences in commands.

IV. Factory - Enemies

My implementation:



Design Pattern:



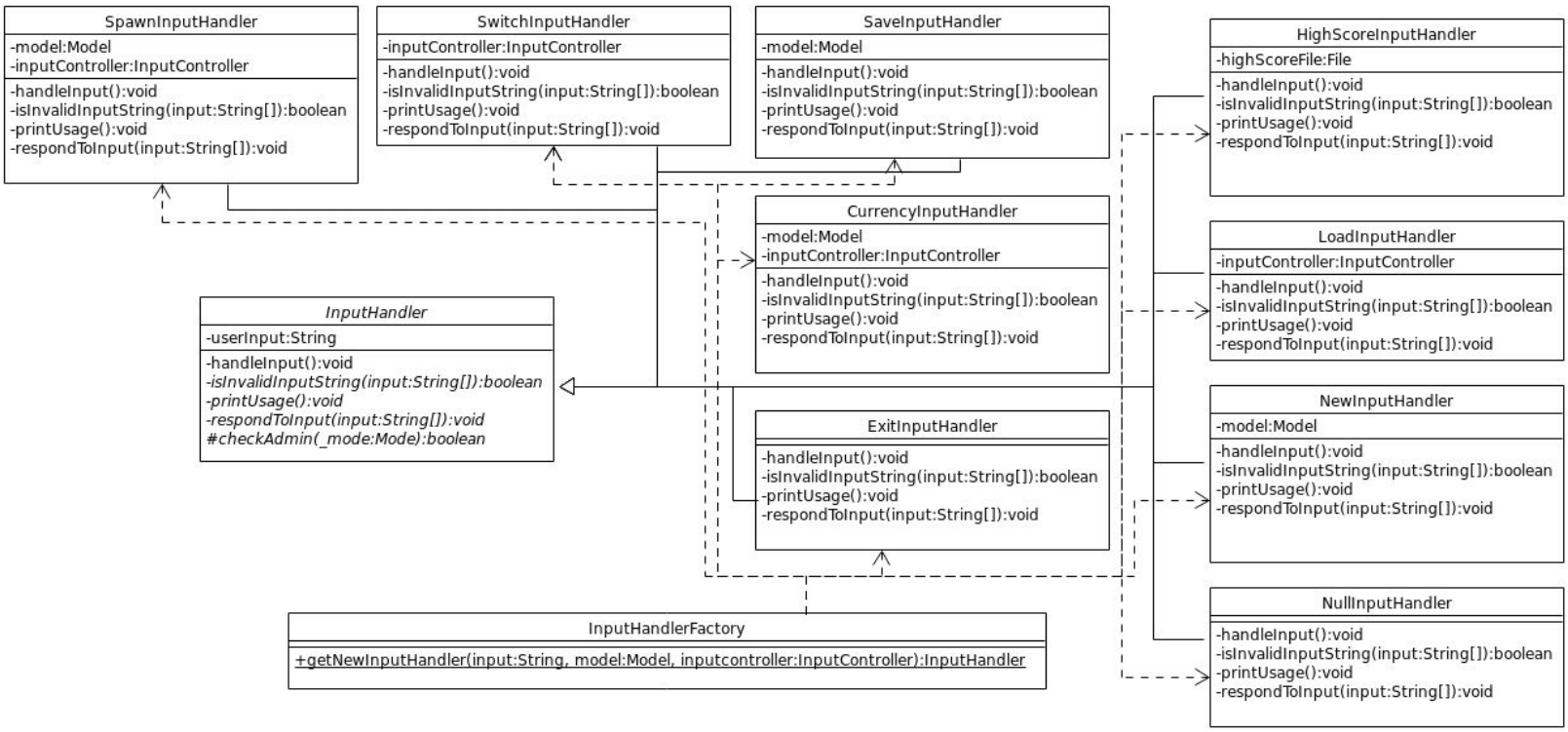
I chose to use the factory design pattern to create the different types of enemies.

I implemented a simple factory without the abstract factory because I only needed one type of factory to create enemies. The EnemyFactory class acts as the ConcreteFactory, the abstract Enemy class acts as the abstract Product class, and the AirEnemy and GroundEnemy classes act as the ConcreteProducts.

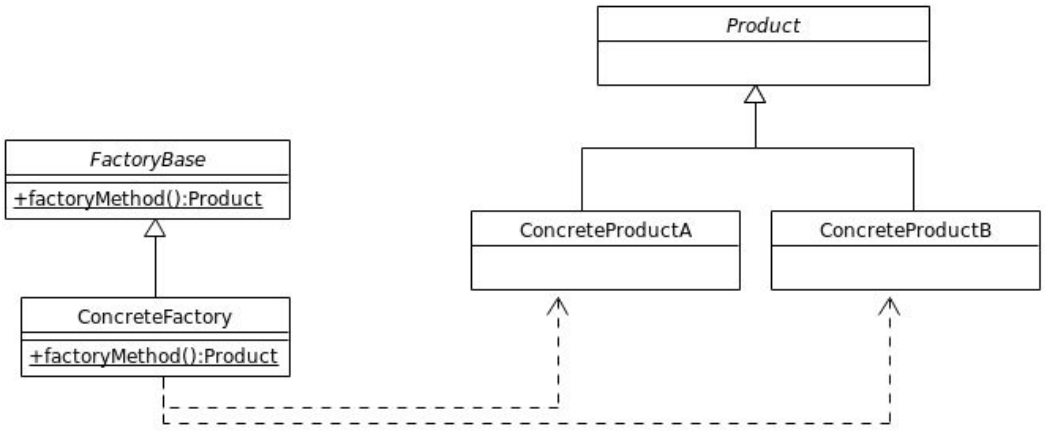
I chose to use the Factory design pattern to create enemies because I had to use if/else statements to determine which type of enemy to create. Using the factory design pattern allowed me to encapsulate the if/else statements into one class. This encapsulation is important because it allows the code to be reused and only one set of if/else statements has to exist within the codebase for enemy creation. Without the factory design pattern there would be several sets of if/else statements throughout the codebase which would have to be maintained each time a new enemy type was added. Because of this, the factory design pattern also adds scalability to the codebase.

V. Factory - InputHandler

My implementation:



Design pattern:



I chose to use the factory design pattern to create the different types of InputHandlers. I implemented a simple factory without the abstract factory because I only needed one type of factory to create InputHandlers.

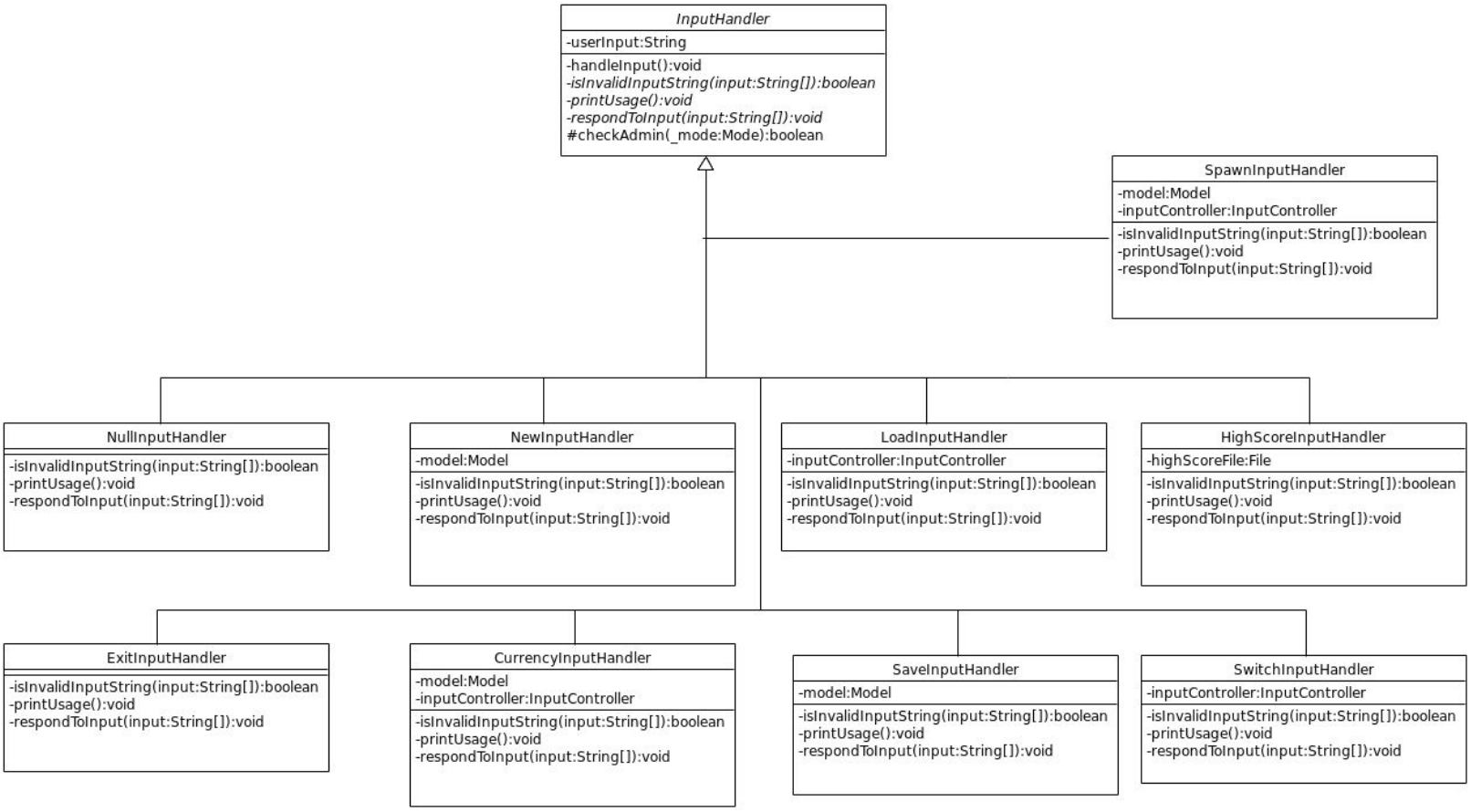
The InputHandlerFactory acts as the ConcreteFactory, the abstract InputHandler class acts as the abstract Product, and the rest of the classes act as the ConcreteProcducts.

I chose to use the factory design pattern in this instance because I had an InputHandler class for each type of command line input. The factory design pattern allowed for simple creation of the correct

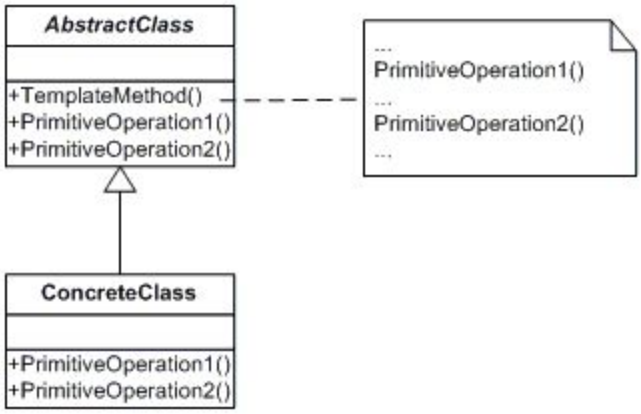
InputHandler based on the string received from the command line. The InputHandlerFactory class also encapsulates the if/else block required to parse the command line input and create an InputHandler. Adding new command line functionality is also made easier by using the factory design pattern because all that needs to be done is creation of a new InputHandler and modification of the factory method.

VI. Template - InputHandler

My implementation:



Design pattern:



I chose to use the template design pattern for processing the command line input.

The abstract InputHandler class acts as the AbstractClass in the design pattern above and the rest of the classes act as the ConcreteClasses. The template method reference in the design pattern above is

handleInput in the abstract class InputHandler. The primitive operations are isInvalidInputString which checks to make sure the command arguments were valid, printUsage which prints the proper command usage if the command arguments were invalid, and respondToInput which responds to the command assuming valid command arguments.

The handleInput method in the abstract InputHandler class first uses the isInvalidInputString method to validate the input. If the input was invalid, it calls printUsage and exits. If the input was valid, it calls respondToInput.

I chose to use the template design pattern because it streamlined the process for creating classes to respond to command line inputs. By guaranteeing that each command line input was handled with the same process, it also became easier to debug. Adding new command line functionality will be easier in the future because the algorithm is already defined.

7. **What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?**

I have learned the value in thinking through a design before coding. Using the class diagram made implementing many parts of the project trivial. It was easy to get confused about how the code should work, but looking at the class diagram often cleared the confusion. The hardest parts of implementing the code were the parts which had not been completely thought through in the initial design. One example of this was the handling of command line input. When initially designing the system, I had not realized how messy it was going to be using a single class to handle all of the different possible inputs. Additionally, functionality such as moving an enemy along a path proved to be harder than expected. However, this functionality was much easier to implement after breaking it down into smaller pieces such as the GameRectangle and Mobile classes which each took on one portion of the functionality.