# Compsci 308 - Software Design & Implementation

# VOOGASalad - Los Torres
# Tower Defense Design Document

## Game

**Describe your game genre and what qualities make it unique that your design will need to support.**

Our chosen game genre is Tower Defense. Perhaps one of the most well known examples of a Tower Defense game is Bloons Tower Defense. Tower Defense is not limited to a basic structure of stationary towers that shoot at Baddies moving along a path - it can be extended to include towers and Baddies with a wide range of movements and weapons, and can have much variation in its gameplay environments.

Here are several characteristics of Tower Defense that our design will support:

- "Game-authoring" as a part of gameplay
    - We need the normal game-authoring environment to define a path (or leave it as free movement), set attributes for actors, and determine other game settings
    - During gameplay, the user edits the environment to add towers and other defense mechanisms
    - Thus, we need 2 sorts of graphical game-authoring environments, with different functionality
- Hierarchy of Game Elements
    - We want to keep our design as flexible as possible, so we are using an extensive class hierarchy
    - At the highest level, we have Actors (Towers, Ammo (bullets, missiles, mines, etc.), Baddies) and Inanimates (Terrain, Enhancements)
    - Keeping our treatment of our Actors as general as possible in the Engine will allow for more complexity and variety in the games we design - at one extreme we can have a rigidly-defined path with simple stationary towers, and at another extreme we can have a sandbox environment that allows free-movement and serves as a sort of open battle arena between towers and Baddies
- Levels and Progress
    - The game will progress as a series of levels, comprised of waves. Each level will bring a stream of Baddies more difficult to handle than the Baddies from the previous level.
    - In between levels, users will have the ability to save progress and make upgrades
- In-game upgrades
    - Players have a certain amount of money available to purchase upgrades for towers and defenses
    - Money is earned through successful completion of levels
    - Possible upgrades are unlocked as the user progresses through the game
- AI Pathfinding
    - Finding the best move along a path of valid moves
    - In an open arena, supporting free movement throughout the game environment
    - Allowing for different AI implementations: seeking out towers, avoiding towers, circling towers
- Pseudo-3D (or as Duke calls it, 2.5D)
    - Actors (Towers and Baddies) can move along any XY Plane

    - Ability to occupy certain types of tiles depends on the type of actor and its implementation
    - Air, Ground, Water, Underwater, Underground tiles - some actors can move through, others can't
- Networking
    - Supporting cooperative multiplayer modes in an endless game type
    - Supporting head-to-head competitions

## Design Goals

---

**Describe the goals of your project's design (i.e., what you want to make flexible and what assumptions you may need to make), the structure of how games are represented, and how these work together to support a variety of games in your genre.**

The primary goal of our project's design is to create a flexible and extensible game platform by writing modular, shy, and consistent code, and to develop a powerful API that can be used on a variety of different game types within the Tower Defense game genre and possibly beyond. For example, while designing our API, we were very careful about not designing too closely around a "path-based" tower defense game in which Baddies are strictly confined to a linear path. Therefore, in order for Baddies to navigate a "sandbox" (also known as an "open arena"), the only code that needs to be added is a pathfinding AI to define the Baddie movements on the level. However, our larger game system and our API should still hold with minimal (or no) modification.

Another priority of our design is the explicit *decoupling* of the physical representation and backend "GameEngine" processing to the Visual representation. In doing this, we will be able to migrate off JavaFX, and move possibly even onto a web-based environment. In our game, we are planning on using JSon to represent various objects that can be created and modified in the Game Authoring Environment, and then instantiated in the GameEngine, and displayed in the View. This way, many different sources can access this JSon file, and select/load in only the data that is relevant to their environment. We also plan on using a system of tagging/serialization to help with communication between "frontend" and "backend" modules.

This decoupling also helped streamline our design of the internal API between the Game Player and the Game Engine, which is loosely based off of the Model-Controller-View pattern. By using a minimal but powerful API between the frontend Player and the backend Engine, we will also be able to use this same structure for a vast variety of different game types without much modification at all.

As was previously mentioned, a high priority in terms of the flexibility of our game Engine is the ability to allow for various movement and attack patterns for both the Baddies and the towers. To accomplish this goal, we plan on using composition over inheritance. That is to say we will have a small hierarchy of actors who hold behaviors, such as their movement, their attack, and their on-death behavior. These behaviors will extend in their own hierarchy, so that it is possible for a separate programmer to come along and code in a behavior that we did not consider. The use of composition will allow us to avoid some of the negatives of inheritance that Professor Duvall mentioned in class. For example, we will not have to construct the extended hierarchy to make a flying, shooting Baddie. Instead we will have an actor that holds an object with a flying behavior and another object with a shooting behavior.

This design choice is not only beneficial in terms of less code, but it also is easily extensible. It will allow us to create many varieties of Tower Defense games, as we will be able to greatly vary the movement patterns and attacking patterns of the Towers, Baddies, and Ammo. Furthermore, other programmers can, without modifying any of the existing code, easily create new types of behaviors for the actors that we may not have thought of. In fact, one could use these behavior classes to create actors that are traditionally not associated with Tower Defense games and even create a MOBA (multiplayer online battle arena).

## Primary Modules and Extension Points

**Describe the program's core architecture (focus on behavior not state), including pictures of UML diagrams and "screen shots" of your intended user interface. Each sub-project should have its own API for others in the overall team. Also describe the purpose and format of different data files you will use to save each game's state.**

## Game Authoring Environment

The game authoring environment allows the user to build new games.

*Path Construction*

Every tower defense game has a path that the Baddies can travel down. Sometimes this path is very well defined with lines, forks, and curves. Conversely, sometimes the path is simply the entire screen except for the areas occupied by towers. Therefore, the first step in path construction will be choosing between two types of paths (specific vs. screen-wide). Regardless of whether the path is well-defined or simply the entire screen, there must be at least one start location (where the Baddies spawn/enter the screen) and at least one end location (where the Baddies exit the screen/damage the player). Thus, the user will then select the start point(s) and the end point(s). In the case of the screen-wide path system, no further work is required in this section.

In the case of the specific path scenario, the user will then have the ability to drag and drop lines and curves onto the screen to construct the path. These linear and curved components will be created using a factory design as they will subclass an abstract PathComponent class. A frontend Path class will simply be an abstraction around a linked list that holds the connected components. As components are connected together (this phenomenon can be determined using JavaFX) these components will be added to their correct location in the linked list. In order for the user to complete this stage, they must have connected components from the starting location to the ending location. The frontend controller will then call a JSONWriter to write the linked list of components into a representation that will later be interpreted by the GameEngine in order to construct the path on the backend.

The simplest solution for the backend path in the game Engine that supports both the specific path and screen-wide is to create a grid of cells. Cell itself will be an abstract class with subclasses PathCell and NonPathCell. A PathCell is a cell that is on the path and can therefore hold Baddies. A NonPathCell is a cell that is not on the path of the Baddies and can hold a tower. Once the user has created a valid path and it has been saved to JSON, the path construction phase is over.

*Baddie Construction*

The user will next enter the Baddie construction phase. In this phase, the user will be able to choose to add previously created Baddies (with previously created healths and damages) to the game. The user will also be able to create their own Baddies by adding images and specifying the speed, health, damage, etc., attributes. In terms of design there will be a tilepane view that allows the user to check previously created Baddies and a “create new Baddie tab” to make a new Baddie. Each of these frontend

Baddies will be represented by an BaddieData object that will hold a map of attributes and an image path for each Baddie. An BaddieDataHolder will hold all of the BaddieData instances. Once the user has selected and created all his/her desired Baddies, this stage is complete.

*Tower Construction*

Tower construction will be very similar to Baddie construction in that the user will be able to select previously created towers and create towers of their own. The additional complexity will be that the user will also be able to specify which Baddies a certain tower can shoot as well as the image and speed at which the ammo that the tower shoots travel. Additional attributes that will be set for the tower include its health, firing speed, and range. A newly created tower's attributes will be stored in a TowerData which will have a AmmoData instance. The user will be able to set all of these attributes (those for the tower and those for its ammo) using the GUI. A TowerDataHolder List abstraction will hold and manage all the towers. Once the user is content with the selected towers, they enter the level building phase.

*Level Construction*

Level construction is the authoring of levels for the game. The levels will be represented in a ListView. To begin the user will add a level by specifying the types of Baddies that will be present in the level as well as the number of each Baddie that will appear. The user will also be able to specify how much money the player earns by completing the level. The data of each level will be stored in a LevelData object that is stored in a LevelDataHolder list abstraction. The contents of a LevelData object will likely be a map of an Baddie to the number of that particular Baddie in that level as well as a length attribute that will specify how long the level will last. In the game Engine, if the level length is short, all of the Baddies will be made in a short amount of time.

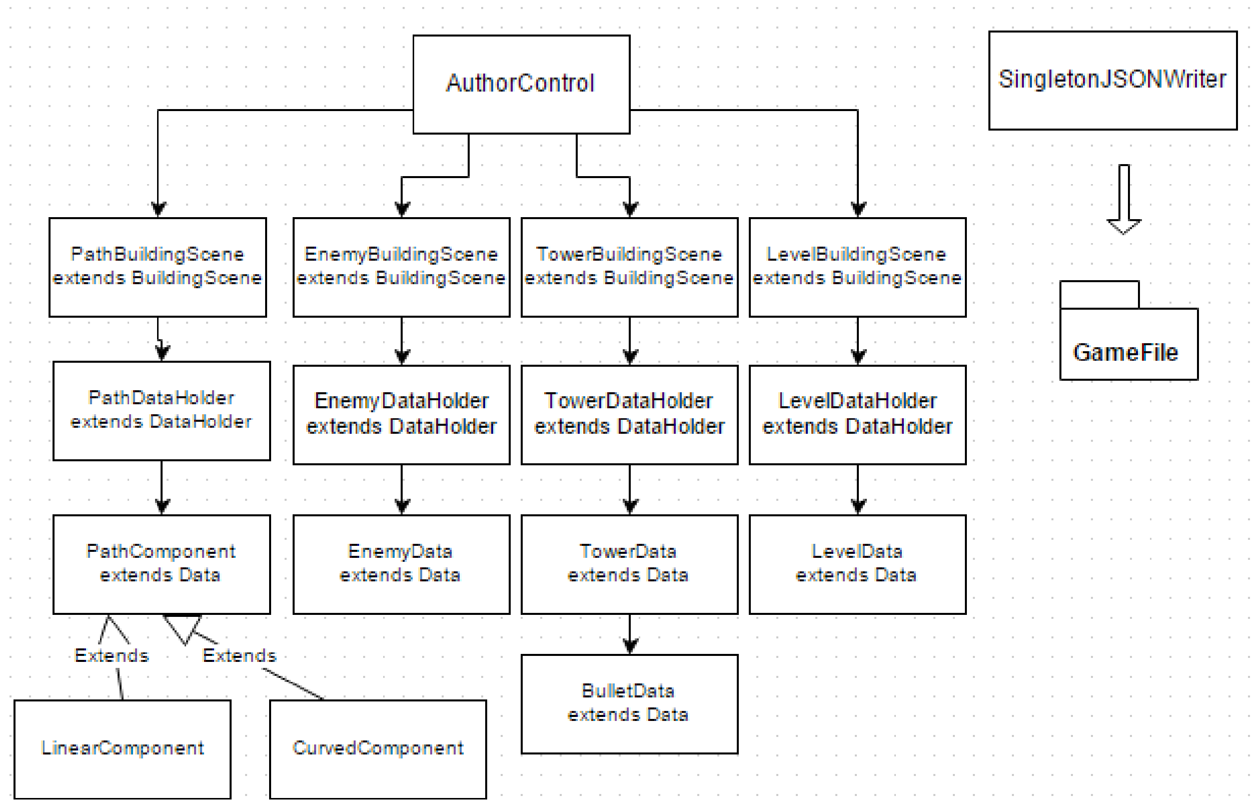

AuthorControl
SingletonJSONWriter
GameFile
PathBuildingScene extends BuildingScene
EnemyBuildingScene extends BuildingScene
TowerBuildingScene extends BuildingScene
LevelBuildingScene extends BuildingScene
PathDataHolder extends DataHolder
EnemyDataHolder extends DataHolder
TowerDataHolder extends DataHolder
LevelDataHolder extends DataHolder
PathComponent extends Data
EnemyData extends Data
TowerData extends Data
LevelData extends Data
Extends
Extends
BulletData extends Data
LinearComponent
CurvedComponent

Figure: Authoring Environment Class UML Diagram

The class structure is fairly simple. There are four major building scenes that are responsible for creating the JavaFX components that will allow the user to build and select paths, Baddies, towers, and levels. Each of these Scenes will store data in a list abstraction called a DataHolder. The data stored for each type of object (Level, Tower, Baddie, and Path) is unique and is defined in the Data Classes. As the user is building objects in a phase of the authoring process, objects are created and stored in the data holder of the scene being shown in the phase. Once a phase is completed, the scene will be able to call the singleton instance of the JSONWriter and pass the data holder object (that contains all the data). This writer will then write the information to a JSON file that can be used by the game Engine to load the authored settings. Below is a screenshot of what our GUI will look like for one of the phases of the authoring process: Path building. In the shot, the user has selected a start point and an end point and can now click on the lines and curve tiles on the right to drag and drop components on the screen. Each time a component is dropped or connected, a PathComponent is created and the PathDataHolder object that holds the linked list of components is updated to reflect the user's path building progress!

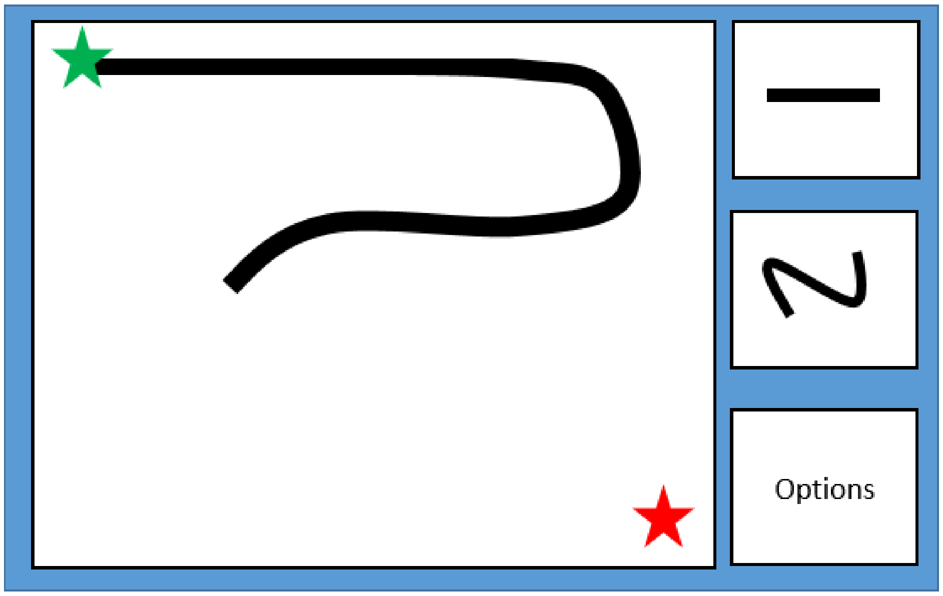
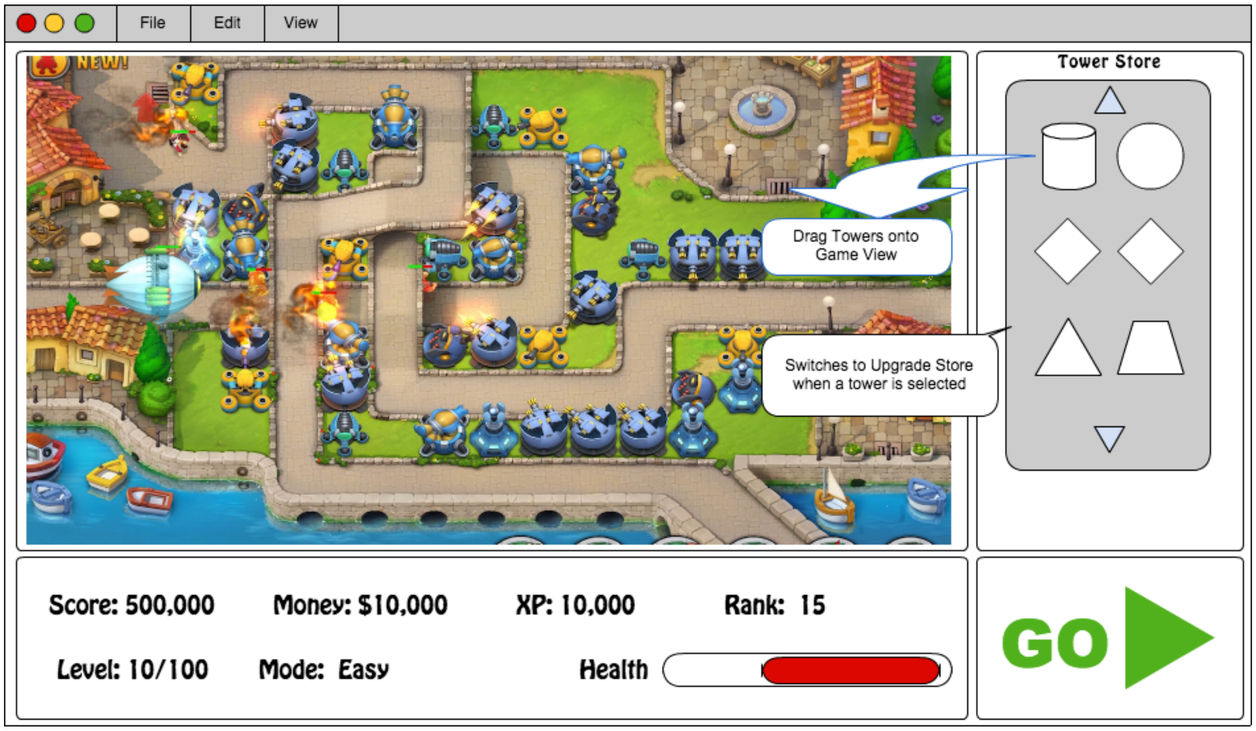


Figure: Wireframe of Path Building

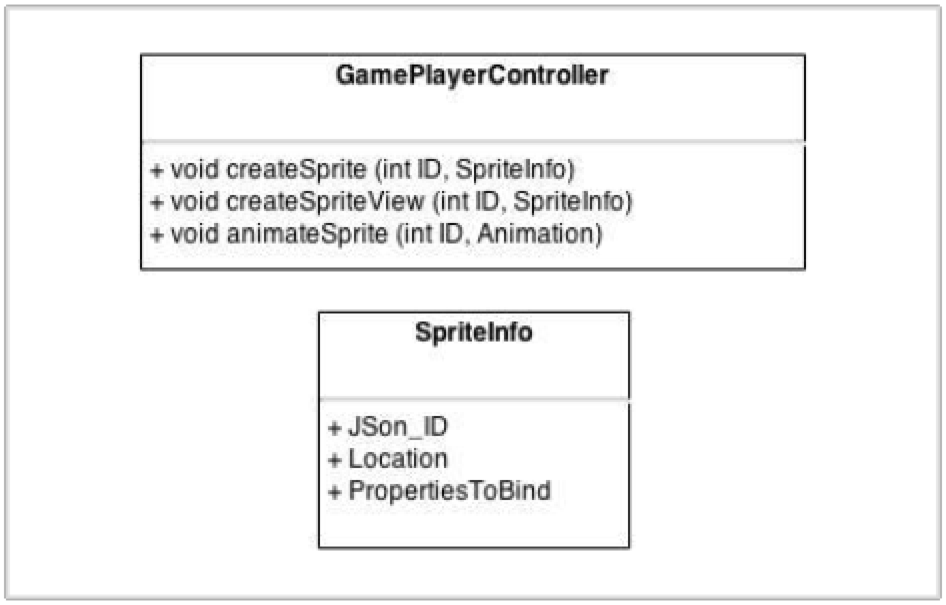
**Note on API**

The game authoring environment will not offer an API. It will also not require a GameEngine API for the system as a whole (the process of making a game) to function. We have elected to write the data created by this environment to a JSON file that can be read by the GameEngine upon game-load. The reason this is better than connecting the authoring environment to the GameEngine via an API is that you would not be able to save a game configuration for future use (you need a data file to do this). Furthermore, the GameEngine and authoring environment do not have to be connected. In our case, a game can be authored in the environment and saved to a File. This file could then be transferred to another computer. On this computer the GameEngine could read the file and the GamePlayer could start the game. Coupling the authoring environment to the GameEngine would not allow this scenario as it requires both pieces of software are running simultaneously (or are in the same project).

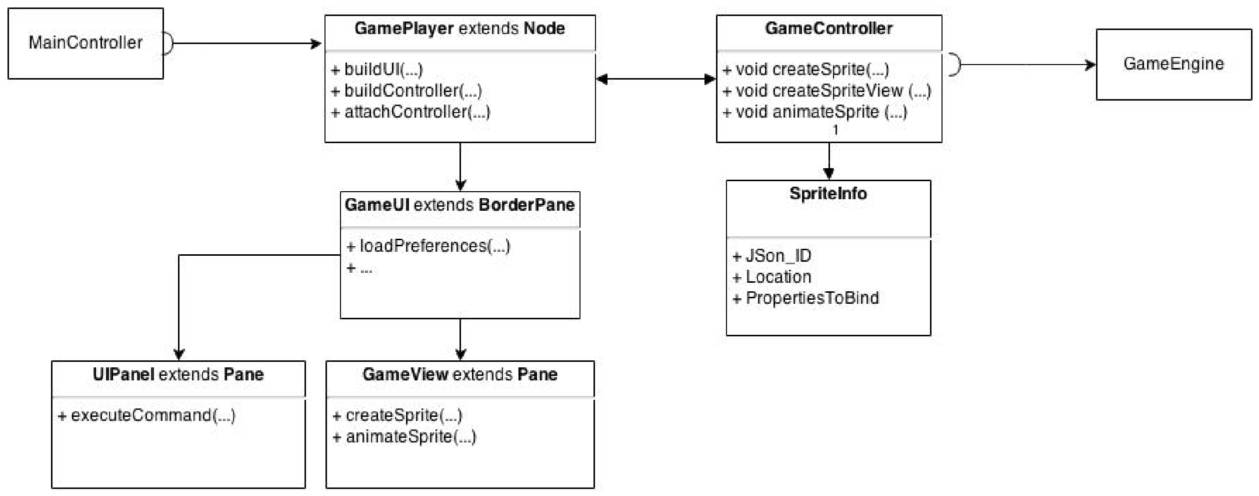
## Game Player (UI)



**Game UI WireFrame Sketch**


GamePlayerController
+ void createSprite (int ID, SpriteInfo)
+ void createSpriteView (int ID, SpriteInfo)
+ void animateSprite (int ID, Animation)
SpriteInfo
+ JSon_ID
+ Location
+ PropertiesToBind

**Game Player ⟵⟶ Game Engine API**



**Game Player Major Classes**

When the human player introduces new elements into the game scene such as a new tower, the createActor method is invoked bearing a ActorInfo object. The Game Engine creates the appropriate element using the JSon_ID and then invokes the createActorView method with a corresponding ActorInfo object. The game player then creates an appropriate image to the game scene. The PropertiesToBind information will be used to introduce binding between data that needs to remain similar between a Actor and its corresponding ActorView (such as x,y coordinates). The ID will be used to match an Engine element to its corresponding image when the Engine invokes the animateActor method. The animageActor method takes an Animation object from an Animation hierarchy that defines the animations to be applied on the image such as imploding, exploding, shattering, etc.

## Game Engine

The key features of the Game Engine will be the game manager, which simply controls the game loop, and the actors who are performing actions within the loop. The game loop will loop through all actors and call on them to update their positions. The IMove interface will allow the actor to call on pathfinding AI for more intelligent choices, and the IAttack interface will provide the ability to fire their weapons. This could mean shooting bullets, laying mines, or throwing punches, depending on the actor's abilities. If they are dead, they will instead execute their onDeath methods, which is related to ITerminate and could provide parting shots as a form of revenge. Other than the game loop, the game manager will also have the ability to load the game state of a file specified by name or to save the current state of the game in the specified file. Of course, it will also be able to initialize the game environment of a specified file. As for gameplay, the game manager has the ability to play, pause, and stop the game, and these commands will correspond to buttons on the view.

Another major component of the Game Engine would be the factories. They will be responsible for creating actors, such as the Towers, Baddies, and Ammo, and inanimates, such as Terrain and Enhancements. This is important for interaction between the Engine and Player. The Engine will create Baddies to spawn during the game, thus giving it an "automatic" feel, and the Player will be able to call on the factories to create a Tower when the user chooses to. Reflection will hide implementation details.

The actors we mentioned before will fall into a simple hierarchy that can distinguish between Towers, Baddies, and Ammo. Beyond that, we will use composition over inheritance to define the various behaviors of the actors. This will allow us to avoid creating a massive hierarchy by simply assigning behavior interfaces to specific objects that the actors will hold. These behaviors will be located in a separate hierarchy that defines, as expected, the behavior of the actor. Each behavior will be an interface that can have many possible implementations. For example, the movement behavior will have many implementations, such as aerial, aquatic, and subterraneal. Creating classes in this way allows our code to be easily extended by third party programmers.

As an example, it would be very simple for another programmer to create a new Baddie with a new weapon that moves in a way we have already defined. All they would need to do would be to write their new implementation of IAttack. Then they can create a new instance of Baddie with their new weapon object and our preexisting movement object, and then they are done. This pattern is highly extensible, as it allows you to create many varieties of Baddies simply by creating new behaviors, and without modifying an existing hierarchy.

As for Engine/Player interaction, a few methods will exist in the controller while relying on the Engine. The addActorImage() method is designed to fill in information on the actor specified by the ID, but the animateActor() method will instead pass in an animation associated with the ID. Those two are methods in which the backend sends information to the frontend. The frontend can use createModelActor() can call on the Engine's factories to build an actor as specified with assigned IDs.

The controller is able to help streamline synchronization between the frontend and backend, and per loop, information will be passed from the backend to the frontend and vice-versa, although the former will more primarily send packets from the backend to the front. Also, because calculations in the frontend are orders of magnitude faster than rendering visuals on the frontend, we will not loop by frame in the backend like the frontend would. This means that they are on two separate threads, which allows us to have the front operate on frames per second instead of running per loop like the backend would. For now, any exceptions that come up will be handled in the backend, which will specify how the frontend should react. This communication can be handled in the controller.

- Actor (These objects store behavior objects listed below)
    - Tower
    - Baddie
    - Ammo
- Inanimate
    - Terrain
    - Enhancement

- Behaviors
    - ISpawn
        - BaseBirth
    - IAttack
        - BaseWeapon
    - IMove
        - BaseVehicle
    - IDefend
        - BaseArmor
    - ITerminate
        - BaseDeath

# Example Code

**Rather than providing specific Java code, unit tests, or actual data files, describe three example games from your genre *in detail* that differ significantly. Clearly identify how the functional differences in these games is supported by your design and enabled by your authoring environment. Use these examples to help make concrete the abstractions you have identified in your design.**

- **Game 1 - Gauntlet**
  - Baddies can only move down a strict, defined path. This path is designed in the authoring environment and saved as data in a JSON file to be loaded by the Engine. For the Engine's purposes, the environment exists as a grid of cells that are either part of a path or not part of a path (PathCell vs. NonPathCell). The user must rely on the quantitative and qualitative strength of his towers to hold off the progress of Baddies down the path. The user will have access to a wide range of towers with varying abilities as the game progresses and the Baddies become more difficult to handle. The towers available are defined during game authoring, and as are the behavior and difficulty of Baddies. This is a design in the style of games like Bloons Tower Defense.

- **Game 2 - Red Light, Green Light**
  - Baddies can move down various paths, which are again created in the authoring environment. The generality of a grid using PathCells and NonPathCells allows for any feasible path to be created (although certain paths are unreasonable for gameplay). In this game, the user has access to a limited range of towers, with the primary focus on traffic light towers. These traffic towers slow down Baddies as they travel down the various paths, so that limited projectile towers are able to take out the Baddies successfully. To modify the difficulty of the game, restrictions on towers are imposed based on number (i.e. no more than 5 projectile towers at once) or type (i.e. only a few basic types of towers are allowed). This could even be extended to a sort of puzzle or survival mode, where there is no functionality to attack Baddies at all, and the user can only strategically place a limited number of traffic towers to delay the Baddies for as long as possible (with score based on time).

- **Game 3 - Sandbox Open Arena**
  - In this game, paths are irrelevant, as the whole environment provides free-movement. This is possible by implementing the environment as a grid of entirely PathCells, which will be a valid option in the game authoring environment. This provides for a whole world of possibilities in terms of gameplay. For a version of "sharks and minnows", Baddies can spawn at any point along one edge of the environment ("infinite spawn" points) and can take any path they want to the other side ("infinite" exit points). These points can be defined as part of the authoring environment - spawn points as well as goal/exit points. The classes defining the Baddies themselves will also specify how the Baddies move across the environment. The user will be able to place towers across the screen to stop the Baddies from crossing (with a high score for the least number of Baddies allowed to cross). This game will rely more heavily on AI-pathfinding as part of its gameplay, as "smart" Baddies will seek to avoid towers to make it across the environment "alive".

- **Beyond "Tower Defense"**
  - We see our game Engine as being extensible enough to possibly support games in the style of MOBA (multiplayer online battle arena), like DOTA and League of Legends. By keeping our design flexible in terms of updating movement and attacks at the most basic level (while leaving specific

implementation details to a class hierarchy), we could feasibly create a viable version of player combat using a modified “Tower” as a player character and modified Baddies for engaging in individual battle. The free-movement capabilities of our game environment and our AI ambitions make this a reasonable goal to test the limits of our design.

# Alternate Designs

---

**Each sub-project should describe at least one alternative to your design that the team discussed and explain why the team choose the one it did.**

- **Overall Design**: We had much debate about how to best handle the interaction between actors in the game Engine and their visual counterparts in the frontend. We considered tethering our actors and their data directly to JavaFX features (i.e. Actor extends ImageView), which would make updating very easy as the backend could directly modify the visual fields that change frontend appearance. However, we decided that this design did not allow for much extendibility. The backend would be entirely shackled to JavaFX, and there would be little separation between Model and View. We want our code to be extensible so that our Engine could be used for a different frontend implementation (for example, once JavaFX becomes obsolete), with as little modification as possible to non-visual components.

- **Game Player**: Once we made the previous decision about separating the frontend and backend representation of actors, we had to decide on our ideal way to keep the two sides updated with each other. We considered using Observer/Observable, but we felt that the necessity of passing objects back and forth made things pretty messy. Instead, we opted for a hybrid system that uses both property binding and identification numbers. Because movement occurs very frequently, we felt that looping through each individual frontend actors to update would not be the best design. Instead, we decided to bind the frontend and backend actors’ properties for their x and y coordinates so that movement in the backend is immediately reflected in the frontend. We use the ID numbers for less frequent interactions, like collisions. Each collision detected in the backend can use the ID numbers of the actors involved in the collision to update the correct frontend elements (with corresponding ID numbers).

- **Game Authoring**: One of our original ideas was to focus on creating TDs with user defined paths only. This design choice, however, would have greatly limited the extensibility of our project as we would not have been able to support a completely free “sandbox” TD, where Baddies use AI to decide where to go next. Changing to a design that allows for this possibility is clearly a better choice, as it allows for the more general types of TDs to be created in addition to the more constrained ones that we initially considered.

- **Game Engine:** We added pseudo-3D so towers and baddies can fly over, walk under, dig under, swim over or under, etc. We want to allow for occupying different XY planes of the same patch/tile. We can still support a simple basic top-down 2D tower defense, but this adds dimensions to our design extensibility. We will also include pathfinding AI to allow for more intelligent gameplay and Tower/Baddie interactions. As a result, we will have a system of tile-ranking to help determine where the actors should move. Also, our original idea of observables has also been eliminated. Paralleling the backend model with the frontend view will let the backend update and drive the frontend. All communication will therefore go through a controller. Property binding and IDs will allow us to interact with the Game Player without compromising the SHYness of our implementation.

# Team Roles

---

**List of each team member's role in the project and a breakdown of what each person is expected to work on.**

**By individual:**

- Brian Bolze
  - Game Player, Controller, Game Engine, general GUI design
- Duke Kim
  - Game Engine, Game Authoring
- Allan Kiplagat
  - GamePlayer, Controller, general GUI design
- Austin Kyker
  - Game Authoring, Pathfinding AI (Game Engine)
- Greg Lyons
  - GamePlayer, Controller, general GUI design
- Chase Malik
  - Game Authoring, Game Engine
- Timesh Patel
  - Game Engine
- Scotty Shaw
  - Game Engine, Game Authoring, Game Player, overall team communication
- David Zhang
  - Game Authoring

**By team:**

- Game Authoring
  - Primary: Austin, David, Chase
  - Secondary: Scotty, Duke, Allan
  - **Extensions**: Open Arena
- Game Engine
  - Primary: Scotty, Duke, Timesh
  - Secondary: Chase, Austin, Brian, Allan, Greg
  - **Extensions**: Pathfinding AI
    - Scotty, Duke, Austin
- Game Player
  - Primary: Greg, Allan, Brian
  - Secondary: Scotty
  - **Extensions**: Scrollable Environments
  - **More:** UI/UX Embellishments, Animations, Networking
- Game Data
  - Will be investigated and utilized by all groups
  - Game Authoring must be able to write JSON files to set game settings/data
  - Game Player must be able to write JSON files to save game state (score, progress, etc.)
  - Game Engine must be able to read JSON files to load game data