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 Enemies moving along a path - it can be extended to include towers and Enemies 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, Projectiles, Enemies) 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 Enemies
- Levels and Progress
    - The game will progress as a series of levels, comprised of waves. Each level will bring a stream of Enemies more difficult to handle than the Enemies 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 Enemies) 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 Enemies are strictly confined to a linear path. Therefore, in order for Enemies 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 Enemy movements on the level. However, our larger game system and our API should still hold with minimal (or no) modification.

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 Enemies 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 and their attack. 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 Enemy. 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 is also 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, Enemies, and Projectiles. 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.

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.

<u>Game Authoring Environment</u>

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

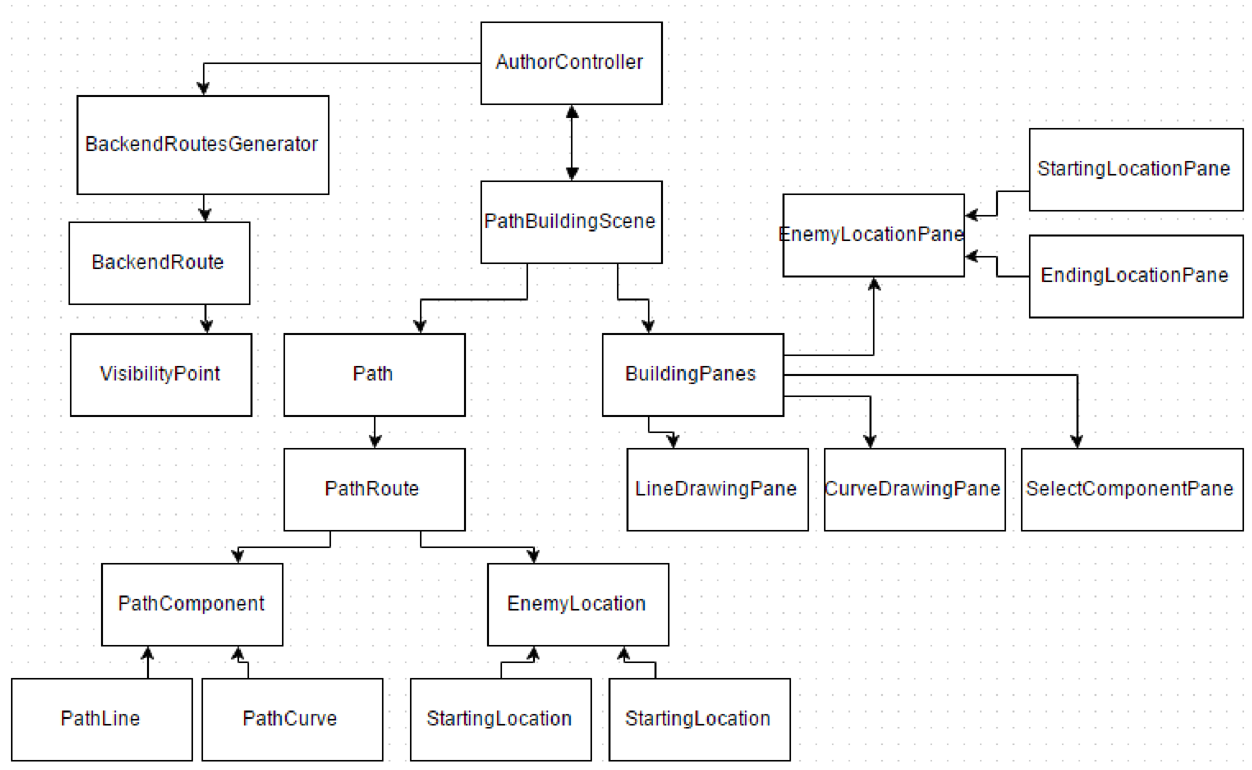
Path Construction

Every tower defense game has a path that the Enemies 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 Enemies spawn/enter the screen) and at least one end location (where the Enemies 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. The user will also be able to upload a background for their game using a Drag and Drop screen.

Now, in the case of the specific path implementation, the user can build their path by connecting linear and curved components. These JavaFX components will extend a PathComponent class. A PathRoute is defined as a list of PathComponents connecting a start point to an end point. A Path class manages the different PathRoute classes as they are created and has the algorithms necessary for merging routes when they are dragged together and deleting routes when the user chooses to do this. Thus, the Path holds a list of possible routes that enemies can take from the designated starting points to the end points. When the path is built, the program ensures that all of the PathRoutes are continuous from the start location to end location. Once this check is done, the PathRoute objects are translated into BackendRoute objects which represent these paths in a form that doesn't require holding JavaFX components. A BackendRoute is simply a set of Visibility Points where a visibility point represents a point at the start or end of one of the PathComponents. Therefore, the set of all these points represents a route. The set of all these routes represent all the ways an enemy can get from start to finish.

The visibility point also has a visibility field that represents whether the enemy will be visible or invisible (underground or in a tunnel) while traveling from this point to the next point. These backend routes are written to a JSON file using the google's GSON API. In the game engine, upon enemy creation, the enemy is randomly assigned one of these possible routes and the movement behavior of the enemy will examine the route's points as well as the current location of the enemy to determine where the enemy will move.

In summary, the user links JavaFX PathComponents together in the authoring environment to build a path. These components are translated into a set of points known as a BackendRoute. These BackendRoute objects are written to GSON and can be read and employed by the game engine in enemy creation and movement. Below is a UML diagram representing the class structure for path building in the authoring environment:



In terms of design, an observable pattern exists between the author controller and the PathBuildingScene. When the user is done building the path and clicks the finished button, the PathBuildingScene notifies the AuthorController and delivers the finished and verified Path. The AuthorController can then make use of a BackendRoutesGenerator to translate the PathRoute (with JavaFX components) into a BackendRoute (of Visibility Points). Now, lets look at the PathBuildingScene itself. It has a Path which has a list of PathRoute objects composed of PathComponents (either PathLine or PathCurve) and a Starting and Ending Location. More interesting is the BuildingPane hierarchy. Instead of having one pane where the user can draw starting locations, lines, curves, select components etc where the on-click event handler operates differently on each pane, a hierarchy of panes was created. These panes can be switched in and out interchangeably and each pane implements its own on-click action. This design was far superior because it removed convoluted code from PathBuildingScene and meant that we could remove a massive switch case to check which mode was the scene was currently in (line, curve, selection, etc.). In several locations I also implemented the Command Principle in which I passed a function rather than data to a lower-level classes.

Enemy Construction

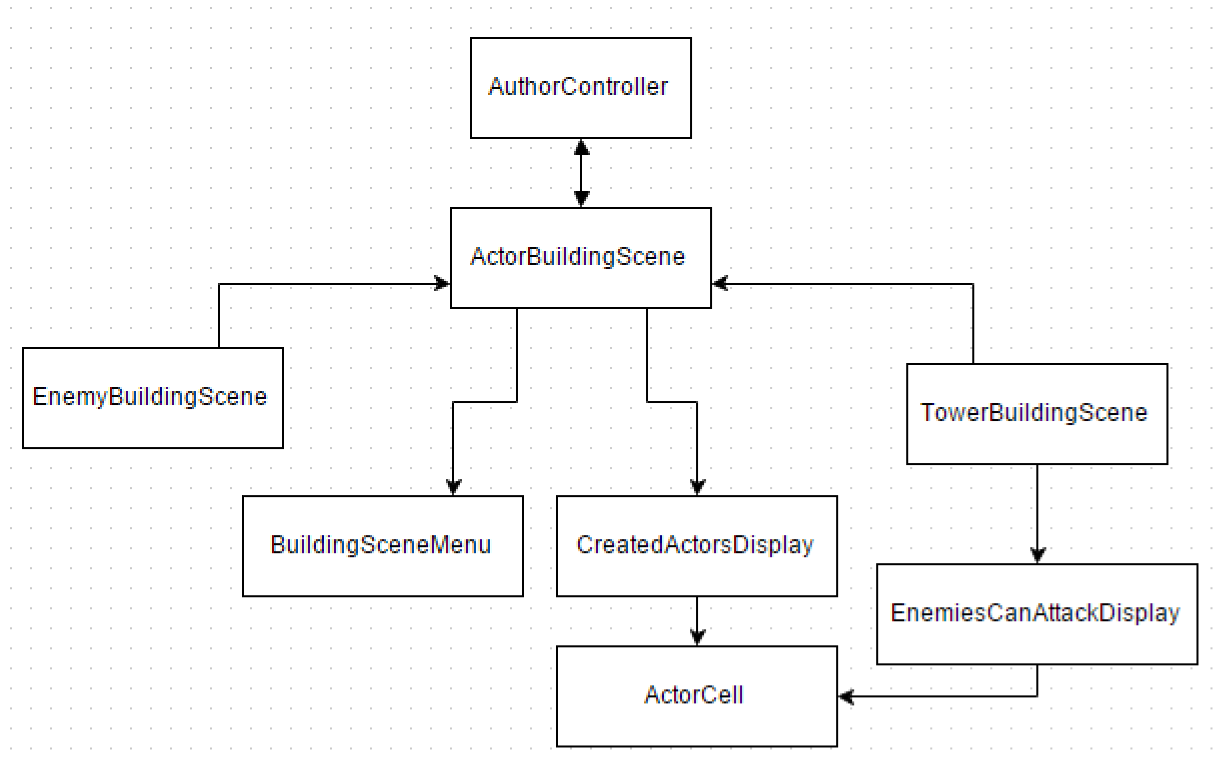
After path creation, the user will next enter the Enemy construction phase. In this phase, the user will be able to choose to add previously created Enemies (with previously created healths and damages) to the game. These enemies will be read from a JSON file using the GSON API.

The user will also be able to create their own Enemies by adding images and specifying the different behaviors of the enemies (see composition section in GameEngine). In terms of the GUI, there will be a ListView on the left of the screen to display the enemies that have been created/loaded. The fields to create a new enemy will be in the center of the screen, and a drag and drop file pane will exist on the right side of the screen to easily allow the user to select an image file for the enemy. Once an enemy has been created and it appears in the ListView, the enemy can be reselected and its behaviors can be edited or it can be deleted.

Tower Construction

Tower construction will be very similar to Enemy 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 enemies a certain tower can shoot as well as the image and speed at which the projectile that the tower shoots travel. A list view of all the possible enemies (all the enemies created/loaded in the previous phase) will be displayed at the bottom of the tower-creation screen. This list-view will be multiple select, and when the tower is saved all of the enemies selected will be saved as enemies that this tower can destroy.

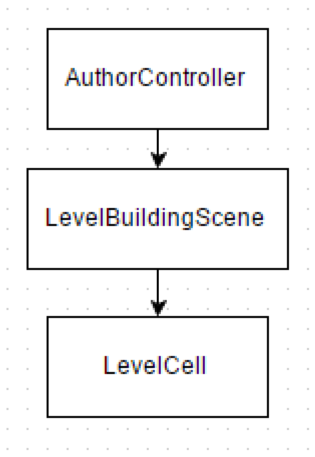
Since enemy and tower creation are so similar we consider their design together:



Once again we have an observable pattern implemented between AuthorController and the scenes. When the user is finished creating a type of actor, say enemies, the AuthorController is notified so it can switch scenes and it is passed the list of enemies which will eventually be written to a JSON file using GSON. EnemyBuildingScene and TowerBuildingScene both extend ActorBuildingScene which has a BuildingSceneMenu that allows the user to proceed to the next scene. These creation scenes also have a CreatedActorsDisplay, a listview, that displays the image and name of the actors that are created. The major difference between the two actor creation scenes is that in building a tower, the user must specify the enemies that the tower can attack. These enemies are presented to the user in the EnemiesCanAttackDisplay which is a multi-select JavaFX ListView. Another major difference, is that the TowerBuildingScene will implement its CreatedActorsDisplay a little different in that towers are upgradable. This means that an element of the ListView will feature multiple towers (a tower and its sequential upgrades) rather than a single actor.

Level Construction

The last stage of the authoring environment is the level construction. The levels will be represented in a ListView. To begin the user will add a level by specifying the types of enemies that will be present in the level as well as the number of each Enemy that will appear. This data will be stored in a map that maps a BaseEnemy to an Integer. The user will also be able to specify how much money the player earns by completing the level as well as the time length for the level. The time length is the amount of time in seconds that it will take to spawn all the enemies. Therefore, having a ton of enemies and a small time length will make for a challenging level!



The design for the level building is very simple. The level building scene has a level list view that makes use of custom LevelCell. A LevelCell shows all of the enemies and has text fields for the user to specify how many of each enemy type will appear. The user will also be able to specify the length of the level. If we have time, we may also add the ability for different towers to be allowed/disallowed from certain levels.

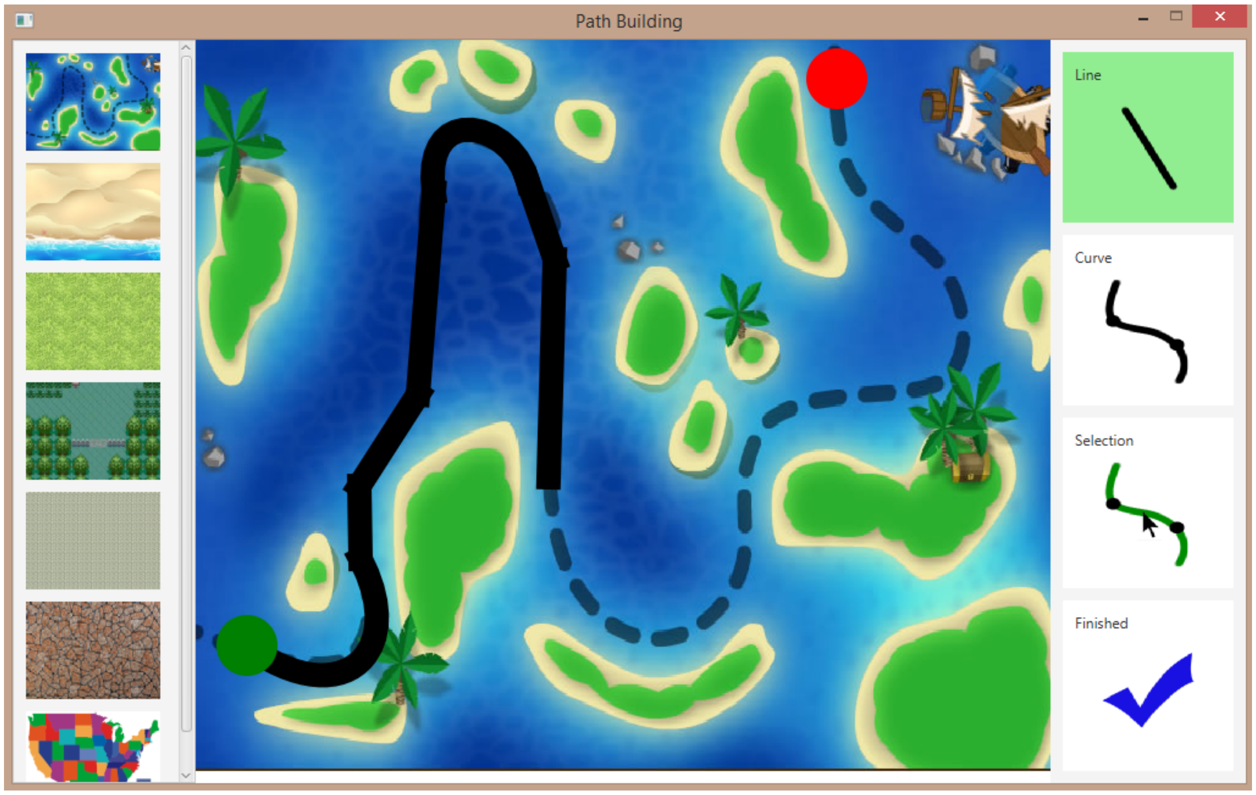


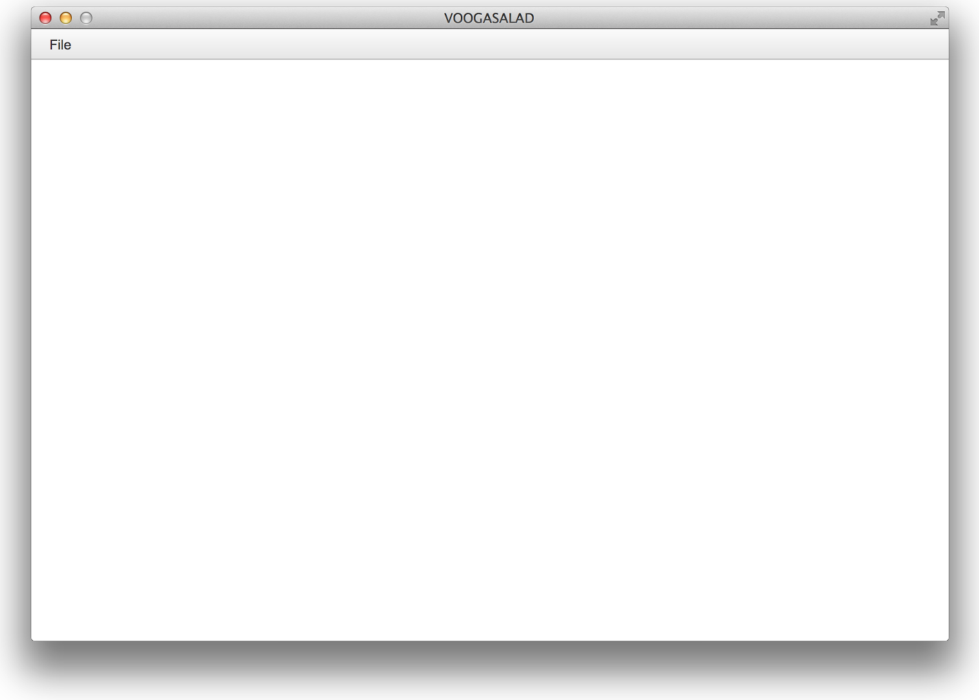
Figure: Screenshot of Path Building

Note on API

The game authoring environment will not offer an API. 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. If we have time, we will also hook the game engine up to the Authoring Environment so that a game simulation can be run while in the authoring process. This will require an API call to the engine to pass the level, tower, and enemy data in order for a GameLoop to be run.

Game Player

The Game Player is responsible for presenting available games to the user and allow the user to play a game using the game engine. When the application is started, the game manager presents a UI such as the one below, with the center section displaying the different available games (and descriptions) in the directory that the user has navigated to. After the user selects the game to play, the game player manager constructs a Game View and instantiates a Game Engine object, passing it the Game View and the game data files.



The Game View consists of modules that the Game Engine uses for displaying game visuals and controlling game interactions. The main modules of the Game View are listed below:

GameView "API"

TowerStore - Displays towers available to be purchased by the user

- Public behavior:
    - Display towers available based on collection provided by back-end Engine
    - Click and drag a tower into GameWorld
    - Check amount of money available to determine validity of tower selection

HUD - Shows current in-game stats and data, including current level, score, money, health (or lives)

- Public behavior:
    - Statistics will be properties bound to properties in the back-end Engine (binding allows for the stats display to be updated without constant looping to check for updates)

UpgradeStore - Allows the user to upgrade existing towers to improved towers; displays the upgrade information for the currently selected tower

- Public behavior:
    - When tower is clicked in GameWorld, displays upgrade information for that tower
    - Choosing to upgrade will notify back-end Engine to replace with an updated tower by passing an ID number for the selected tower

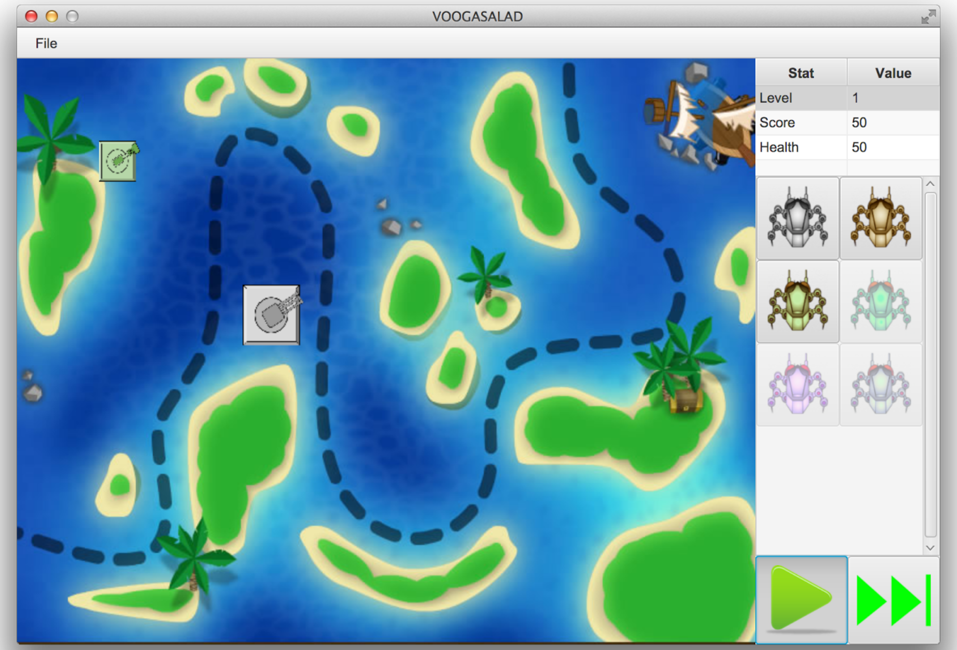
ControlDock - Gives the user capability to play, pause, and change game speeds (normal or fast)

- Public behavior:
    - Can start, pause, resume, and fast-forward game looping within the back-end Engine
    - Accommodates continuous gameplay (no stoppage between levels)

GameWorld - Visualization of gameplay; displays all of the actors and the interactions between them as they move throughout the map

- Public behavior:
    - Dragging and dropping (or clicking to place) a new tower notifies back-end Engine to instantiate a new tower, passing the tower type and the location coordinates
    - Clicking on an existing tower opens its information in UpgradeStore

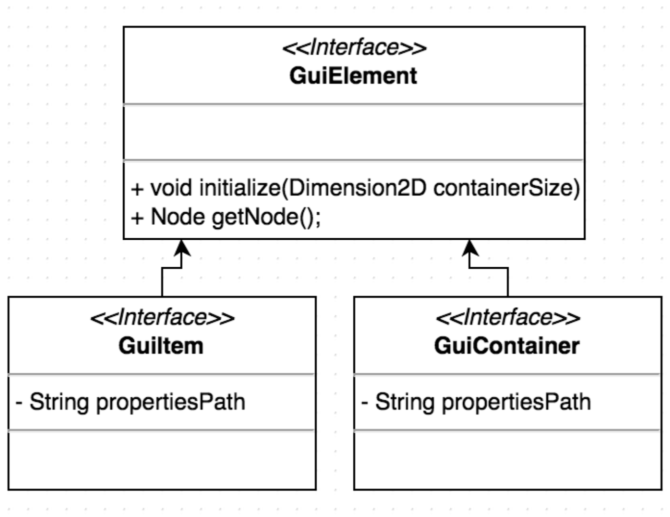
The following image shows an example of the Game View in-game UI (the UpgradeStore is not shown in this image)



Stat	Value
Level	1
Score	50
Health	50

Design
In the Game Player, every visible item on the GUI is considered a stand-alone module that is hosted within a container. For example, the menu bar is a module, complete with its own behavior and is hosted in the container at the top of the scene. These GUI individual items are sized based on the size of their container, using ratios specified in XML files. Each container has an XML file to specify a size ratio and a list of GUI modules to include (which it initializes via reflection). Storing these parameters in XML files to be parsed rather than hard-coding JavaFX coordinates is preferred because it makes modification easier and more extensible. New items can be added to containers or existing items can be moved to different containers without difficulty. We use our GUI Manager to handle the behavior of each of the GUI modules and to facilitate interaction with the back-end Engine. However, the individual GUI modules see the GUI Manager as a specified interface (specific for each module) rather than as the whole manager, so they are only able to call the methods that are needed for that specific module.

In the UML diagram given below, the GuiElement interface defines the universal behavior of GUI modules, which includes initializing themselves and returning their Node for representation in their host container. GuiItem and GuiContainer are tag interfaces to differentiate between a passive container and an active item that has extensive behavior.



Game Engine

GameManager

The point of communication between the player and the engine will be through the GameManager. The GameManager will be the one to handle the game loop, updating the backend elements of the game per frame. From the player's perspective, the GameManager will be the one to actually run the game, after the game itself has been loaded. Furthermore, the game engine was designed with the goal of minimizing unnecessary communication with the game view, and hence adding new actors will be done in the GameManager directly rather than passing them to the Player to add.

The GameManager is also designed in a way so that the outward facing communication methods can be used by both the game view and the game authoring environment. The purpose of this design is so that while creating the game in the authoring environment, game developers can do a sample run of the level for the purpose of testing. This is done by allowing the outside to load levels in manually instead of having to load a game file.

On the topic of loading, the main public methods of the GameManager is loading games, levels and game progress states, and saving progress states. Furthermore, the game loop in the game manager will have the ability to be safely paused and stopped from the outside, so that loading new levels mid game can be done safely without any problem of concurrent access.

Actors

Before going further, it is necessary to talk about what actors are. Actors are considered to be the dynamic elements of the game itself; it is what allows the application to be defined as a game. The best way to explain this is through examples. The BaseActor superclass branches into two sub-branches - RealActor and Projectile. Furthermore, RealActor branches into BaseEnemy and BaseTower. Seeing as how our game is a Tower Defense game, we felt that these were the core elements of the game that could not be removed; Towers are required as the name implies, while enemies are required to pose as obstacles for gameplay. Finally, we believed that projectiles also deserved to be actors instead of just a subcomponent of Towers, so that they could be more dynamic in behavior.

Actors will use composition over inheritance to define the various behaviors. This will allow us to avoid creating a massive hierarchy by simply assigning a variable number of behaviors to specific objects that the actors will hold. These behaviors will have a separate hierarchy that defines, as expected, the behavior of the actor. Each behavior will implement the behavior interface and be extensible so many types of specific behaviors can be made. For example, the movement behavior will have many implementations, such as linear movement where the actor only moves along a set of lines or movement along a free path where the movement is dynamic as new obstacles appear affecting its path. 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 Enemy 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 Attack. Then they can create a new instance of Enemy with their new attack 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 Enemies simply by creating new behaviors, and without modifying an existing hierarchy.

- Actor (These objects store behavior objects listed below)
    - RealActor
        - Tower
        - Enemy
    - Projectile

- IBehavior
    - BaseMovement
        - LinearMovement
    - BaseAttack
        - ClosestAttack
    - BaseDefend
    - BaseEffect

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**
  - Enemies 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 Enemies down the path. The user will have access to a wide range of towers with varying abilities as the game progresses and the Enemies become more difficult to handle. The towers available are defined during game authoring, and as are the behavior and difficulty of Enemies. This is a design in the style of games like Bloons Tower Defense.

- **Game 2 - Red Light, Green Light**
  - Enemies 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 Enemies as they travel down the various paths, so that limited projectile towers are able to take out the Enemies 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 Enemies at all, and the user can only strategically place a limited number of traffic towers to delay the Enemies 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", Enemies 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 Enemies themselves will also specify how the Enemies move across the environment. The user will be able to place towers across the screen to stop the Enemies from crossing (with a high score for the least number of Enemies allowed to cross). This game will rely more heavily on AI-pathfinding as part of its gameplay, as “smart” Enemies will seek to avoid towers to make it across the environment “alive”.

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.

- **Game Player**: We considered using a single large container class to hold all of the GUI components, with their locations and sizes hard-coded as part of initializing that class. We decided that this container class would get overly cumbersome and that we could come up with a better way of separating the components, and hopefully a better way of fitting them all together without hard-coding in JavaFX coordinates for every element. We settled on our current design, which separates the GUI into smaller containers. The properties of the containers and the elements they contain are specified in separate XML files, so modifying them is as simple as changing a few lines of XML rather than finding and changing hard-coded values and positions. The GUI items themselves are sized based on the size of their container rather than having a hard-coded absolute size, which is more preferable because modifying the containers will appropriately modify the items’ size without needing additional coding.

- **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 Enemies 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:** To traverse paths we considered using a 2D grid, but we decided against this decision for a few reasons. Most notably, we figured out that path finding would not operate effectively if two paths overlapped, so we decided that having separate objects represent each path was a more effective strategy. The other major design decision we made was to include javafx node in the backend of the engine. This choice was made after much debate because we felt that it would limit the interaction between the front end and the back end, allowing our program to run more efficiently. Other design topics that were discussed include the structure of the behavior interface, and whether or not there should be multiple interfaces or just one. We decided that having one was more extensible because actors now take a list of behaviors, so adding a new one does not require modifying existing code. The other topic discussed was the hierarchy of the actors. It was difficult to decide between having the towers and enemies behave in the same way, as their behaviors are, in general, very similar. However, in order for the authoring environment to accommodate upgrading towers we decided that they should separated in the hierarchy.
  Another alternate design we considered was dedicating another thread to do the main game loop. This would potentially help prevent dropping frame rates and increasing accuracy when speeding up the game. This would have been done by separating the updating and rendering of the actors. However, we decided that the potential concurrency problems introduced by multi threaded would not be worth this optimization.
-

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
    - Secondary: Chase, Scotty, Duke, Allan
    - **Extensions**: Open Arena
- Game Engine
    - Primary: Scotty, Duke, Timesh, Chase
    - Secondary: 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