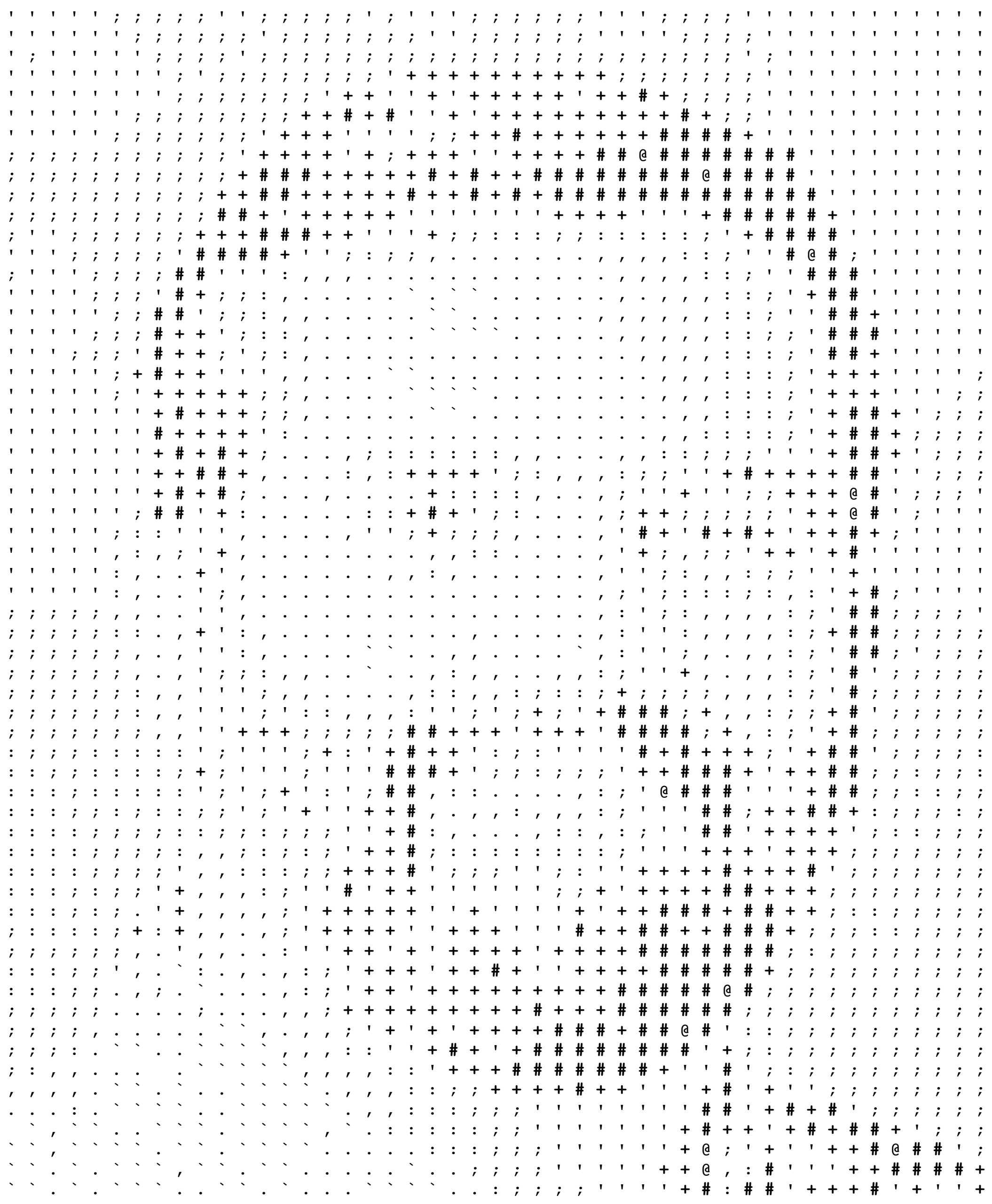
VOOGASalad: The Exception
Our Dope API and Illest Design Document



Note: This is a high-level description of our API. For a more detailed description, please see our Javadocs and our package.html file.

Genre

At its most fundamental level, a real-time strategy game is a top-down, real-time game in which the player must position and maneuver units and structures under their control to in order to secure areas of the map and/or destroy their opponents' assets. Unlike many other 2D game genres, an RTS game offers a top-down view in which the player is not represented as a single in-game unit but rather as a commander that has the ability to select and maneuver units under their command. The goal of classic RTS games is to use a starting set of units to gather resources and use these resources to create structures and factories to allow for the creation and strengthening of additional components. These components are then commanded to attack enemy units on the map with the ultimate goal of defeating the enemy's commander through the achievement of a predetermined list of objectives.

Due to the extremely open and interactive nature of RTS games, supporting the design and creation of RTS games quickly becomes a complex design problem. Game maps need to not only have support for different terrains for different unit types but they also must support interactable (and possibly renewable) resources and elements that the player's units must be able to interact with. In addition, terrains need to be able to support temporary changes of states as in the case of spells or traps being set for other element .

With regards to the game units themselves, a wide variety of behaviors and interactions need to be supported by our system as well. In RTS games, units must contain much more functionality than simple collision detection and movement. Due to the sporadic interaction between the player and the units, units must be able to have behaviors that are automatically executed when the units are not being controlled by the player. In addition, different RTS games necessitate the definition and interaction between any number of arbitrary parameters of game element. In order for our game engine to be successful, the definition of custom attributes and logic through which these attributes are interacted with must also be supported. Units in an RTS game can also have behavior that is not as simple as basic interaction with the parameters of other units and must also support complex behaviors such as the spawning and containment of other units.

In addition to the requirement of highly customizable terrains and game units, the gameplay itself must also be highly customizable. A level in an RTS game may not be as simple as killing all the enemy players; it might be a race to controlling a certain area of land, collecting the most resources, building the largest city, or any other combination of arbitrary objectives based on the state of the units or terrain in the game. For our engine to be truly robust in its creation of RTS games, we must also provide support for the definition of highly customizable goals paired with unique maps, unit placements, and scenarios for each level.

Looking at the requirements above, it is clear that an effective RTS engine will need to allow for a great deal more customization than most other game engines would allow the user. Users will need to be able to define their own custom terrains, units, goals, behaviors, and levels in order for our editor to be truly robust.

Design Goals

Our application's functionality can defined in two major groups: editing a game scenario and playing the game. Our game playing functionality will be the same as a typical RTS. Players face off against either another or multiple other opponents and ultimately try to achieve pre-defined goals. The player will be able to control multiple units at one time and direct them as a collective unit. The player can also select buildings, resources, environmental objects (like resources), and their opponent's units, each with different levels and limits of interaction.

Standard RTS game editors allow users to place, move, and delete units as well as create the game terrain using a wide variety of user-defined blocks. The game editor also allows the user to modify particular internal attributes of units and certain parts of the environment. A lot of newer and more popular game editors give the user the ability to also define specific unit paths, create unique game goals, and create campaigns and scenarios around the games they create.

We want our game playing functionality to be adaptable to different goals. For example, even though the commands and keys will usually not change during gameplay, different game types might require ubiquitous commands such as a specific keypress to be interpreted differently. Our game play will also be modular enough to support multiplayer capabilities, but that goal is not one of our top priorities.

Our game editor is essentially a group of editing UI modules, where each of the modules is capable of configuring a different aspect of a game element. Examples of elements modifiable by the game editor are units, non interactable elements in the environment (like rocks), resources, projectiles, etc. Each one of these elements should be placeable through the editor interface and editable by the player on a number of different levels. The player should be able to dynamically create and place these elements on the map from the editor. They should also be able to change and edit physical values about each of the elements, such as their color, bounding box, strength, health, etc. Our game editor will also allow the user to create paths, edit goals, and create specific campaigns based on their own preferences.

One of the defining and unique attributes of our game editor will be the user's ability to effectively modify the relationship between elements of the game. We ultimately want each element to be able interact with every other element in any way we choose and in order to do this, each one of our elements will have detailed and agile interfaces that help define how it interacts with others. In our UI, we will give the user the ability to change how units and environment blocks can interact with each other.

In order to optimize the design of our program, we need to define the sections of our design that will be flexible / extensible. Our GUI will be comprised of modular panes. These panes will be dynamically generated from XML files and assembled into a cohesive, larger GUI pane.

Communication between the Model to and from the Engine and GUI will be implemented through an Observer-Observable pattern. In this schema the Model is the Observable, while the Engine and GUI are the Observers of the model. Furthermore, the Engine and GUI are also controllers for the Model, that can call methods to update the state of the model. Thus, upon instantiation, the Engine and the GUI are given references to the Model, to allow them to update the Model, e.g., when a new Level is created in the Editor the GUI can call createLevel() and pass it a level name and description to be added to the Model. For communicating from the Model to the Engine and GUI, whenever the Model is updated and the GUI or Engine to update its state, the Model calls notifyObservers() to elicit the changes based on the model state in the engine and the GUI. For example, if the user creates a new Archer then it sends the info about that archer (a bunch of strings) to the Model, the Model updates itself and notifies the GUI to update itself and GUI then when getting the list of possible characters in the game it sees the archer and adds it to the accordion showing all of the drag and droppable units. In addition, since the engine will also be observing the model, the engine will be notified of an additional unit being added to the game and will update its factories accordingly to allow for the creation of this new element type.

The Engine, which generates the view of the game state and does the complex calculations to animate the game, also has a connection using the Observer/Observable pattern to the Model. The Engine uses this connection in order to initialize the state of the game animation each time a major change is made. This mostly happens when the game is begun as well as any time a level (one concrete map with set goals) changes. This connection is also used to keep track of what elements are currently selected in GUI panes that the Engine has no direct control over, such as unit-to-be-placed selections within the Editor panes.

The Model holds a reference to the save/load utility and utilizes it to save and load JSON files from the disk in order to generate game state representations as Java Objects. The Model holds these elements and some others such as libraries that are necessary to fully characterize all possibilities for customization inside of the game. The Model exposes a selective API to the Editor such that the passive data structures that hold the game state cannot be corrupted by a frivolous user. The Engine, on the other hand, is able to gain direct access to the data structures, as users cannot directly influence the actions of the engine without doing so through Editor capabilities, thereby safeguarding data structures from corruption by the Engine.

Although our main update functionality is carried out through the use of a game loop, most game actions elicit their behaviour through a set of maps of conditions to actions, making this part of our design very similar to an event-based one. This mapping will be specified when units are created in the editor, and will be loaded from their JSON encodings when the game is to be played. During the update phase of the game loop, each element will run all conditional functions as appropriate and use their output to determine which reactionary functions it needs to run as well. For instance, all game elements will update using conditions that evaluate on self-state. Drawable game elements will need to also update based on conditions that evaluate on the elements' internal representation of the animation it is currently running. Selectable game elements, however, will also need to run conditions that operate on element that are colliding with the element in question, or those that are visible to the element in question, or even those that refer to actions required of the element by user interactions. All elements will be very extensible in this manner.

Some assumptions we make about the manner in which users will interact with our game are: 1) we assume that the user will have the ability to code in Java if they are looking to implement paradigm-shifting extensions to the project. 2) we assume that the users will use FXML to encode their GUI panes that they construct. 3) we assume that the users will use JSON to encode their game if they are creating them without the use of the editor.

In order to play a game, a number of representations will be created. Firstly, XML representations of all of the GUI panes will be created. JSON representations of all game element ects will also be created. These will be parsed into the game and represented as JavaFX nodes if they are drawable or as instance variables if they represent game state. This structure is entirely modular and allows for both simple and complex RTS games to be created.

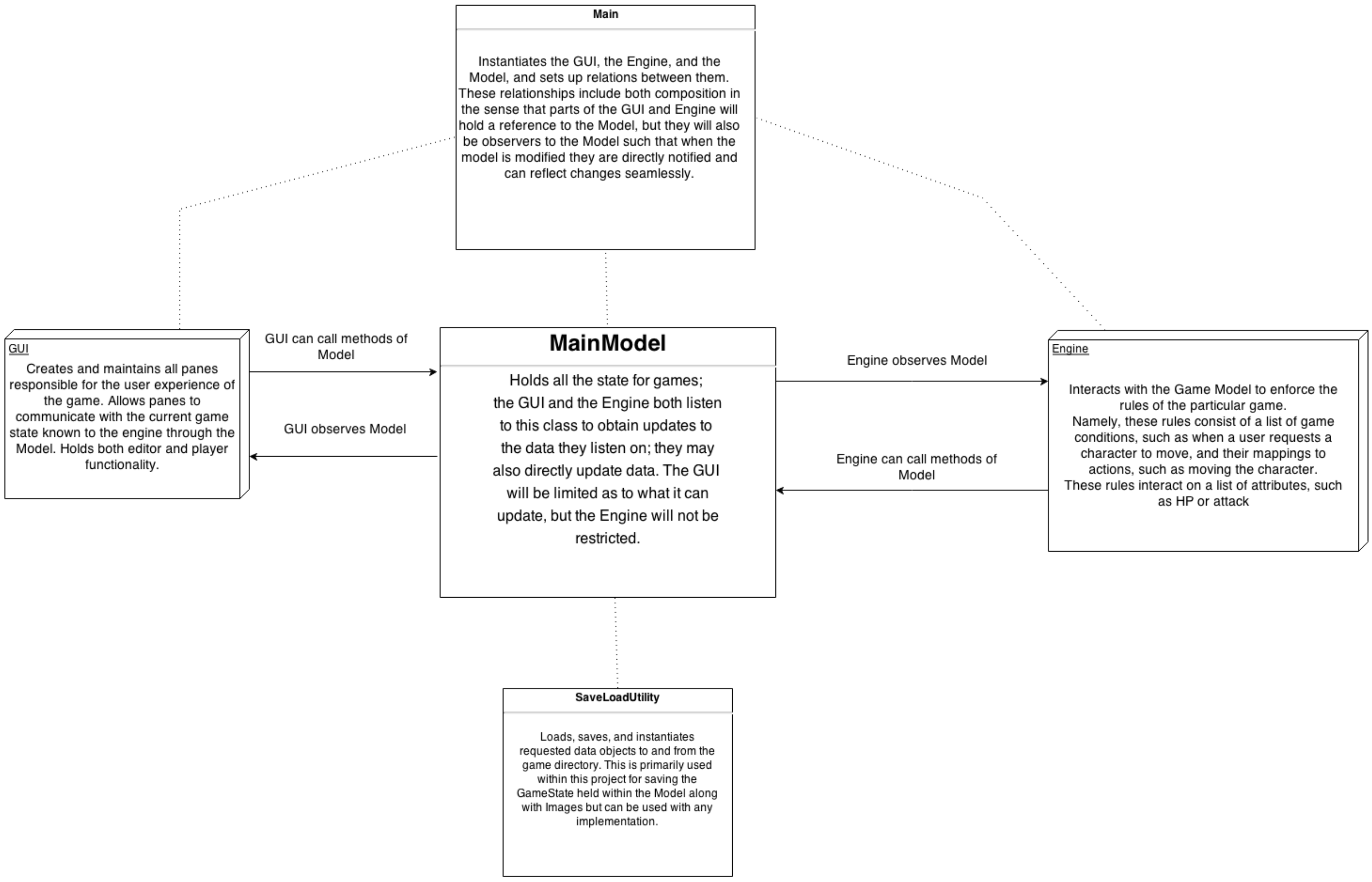
In terms of game mechanic restrictions, we have started with a fairly broad allowance for the interaction of game elements within the game. Because lambda functions are not serializable and allow a user using the editor to access much more information than they might be privy to, we have decided that these were not the best option in encoding our condition action pairs. Instead, we have created a basic scripting language, dubbed VScript (described in detail in the Engine section below), to encapsulate all possible conditions and actions. Our design will benefit from this decision as it not only provides a layer of abstraction between conditions and actions and the underlying code but it also will make it easier for the editor to allow for the dynamic creation of condition-action pairs. In addition, the creation of this simple scripting language will allow a user to write their own custom actions quickly without them needing to know java or any details of our underlying implementation. At their fundamental level, conditions allow for the comparison between any two attributes of objects and returns a boolean of the evaluation result of the conditions. In our game element design, all attributes are stored in maps allowing them to be referenced by these conditions. Although this does add a bit of complexity to our design as there is no guarantee that a parameter will exist in a given object, this adds power to the objects and creation of custom interactions as every attribute in an object can be cleanly referenced by a condition. Like conditions, actions are things that act on the attributes of objects. By pairing these conditions and actions, we have allowed a method for a user to cleanly add highly customizable behaviour to their units without having to write a single line of code.

In our effort to make this language as simple as possible, we have removed the temporal aspect of interaction, i.e. one element can not see previous states of another element but can rather only see the other element’s current state. In addition, we have restricted element as being defined to have one owner (as in one player that controls a given unit) and have not included an allowance for multiple owners of one game element. In our design, we have also restricted an element’s knowledge of the element around it to only allow an element to know about the element colliding with it or contained within its vision box.

In addition to element restrictions, we also have made certain essential assumptions about gameplay mechanics. The biggest of these is the camera through which the game is viewed. Our system allows only for the camera defined by the animations being used and assumes a finite world size.

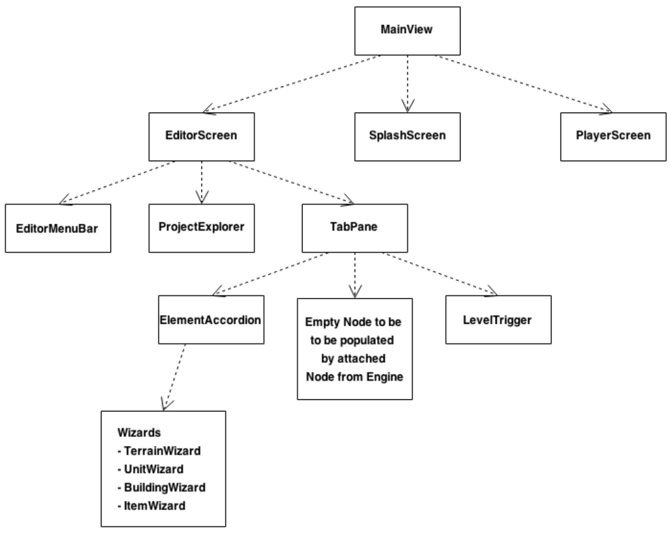
Primary Modules and Extension Points

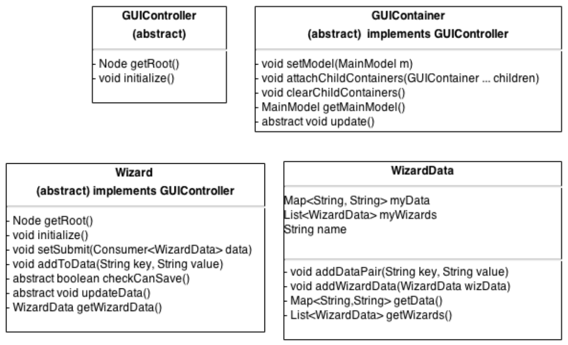
The High Level Design:


Main
Instantiates the GUI, the Engine, and the Model, and sets up relations between them. These relationships include both composition in the sense that parts of the GUI and Engine will hold a reference to the Model, but they will also be observers to the Model such that when the model is modified they are directly notified and can reflect changes seamlessly.
GUI
Creates and maintains all panes responsible for the user experience of the game. Allows panes to communicate with the current game state known to the engine through the Model. Holds both editor and player functionality.
GUI can call methods of Model
GUI observes Model
MainModel
Holds all the state for games; the GUI and the Engine both listen to this class to obtain updates to the data they listen on; they may also directly update data. The GUI will be limited as to what it can update, but the Engine will not be restricted.
Engine observes Model
Engine can call methods of Model
Engine
Interacts with the Game Model to enforce the rules of the particular game. Namely, these rules consist of a list of game conditions, such as when a user requests a character to move, and their mappings to actions, such as moving the character. These rules interact on a list of attributes, such as HP or attack
SaveLoadUtility
Loads, saves, and instantiates requested data objects to and from the game directory. This is primarily used within this project for saving the GameState held within the Model along with Images but can be used with any implementation.

Our design is centered around the coordination of four different main parts: the GUI, the game engine, the model, and the save/load utility. The GUI holds all the panes and groups of the application and is responsible for switching between the Splash Screen, the game editor page, and the game player scene. The editor and the engine both handle user inputs and update the UI and the game internals depending on the user's interaction. Both can be considered part of the view but neither interacts directly with the other. The view provided by the engine is what a user would see when playing the game whereas the view provided by the GUI is a set of panes that surround the view from the engine that allow for modifications of attributes of the engine. The model is what is responsible for storing the state of all objects in the game and the game's state itself. It has an observer/observable relationship with the editor and the engine, where the model is the observable and the editor/engine are the observers, and is also responsible for calling update on the engine or on the editor when, for example, the user decides to change a level. The save/load utility allows for loading and saving of GSONable data objects (non-Java FX, functional interfaces) as well as loading and saving of media (such as images). It is here where we save game states after closing our application. The load/save utility interacts directly with the model.

GUI:



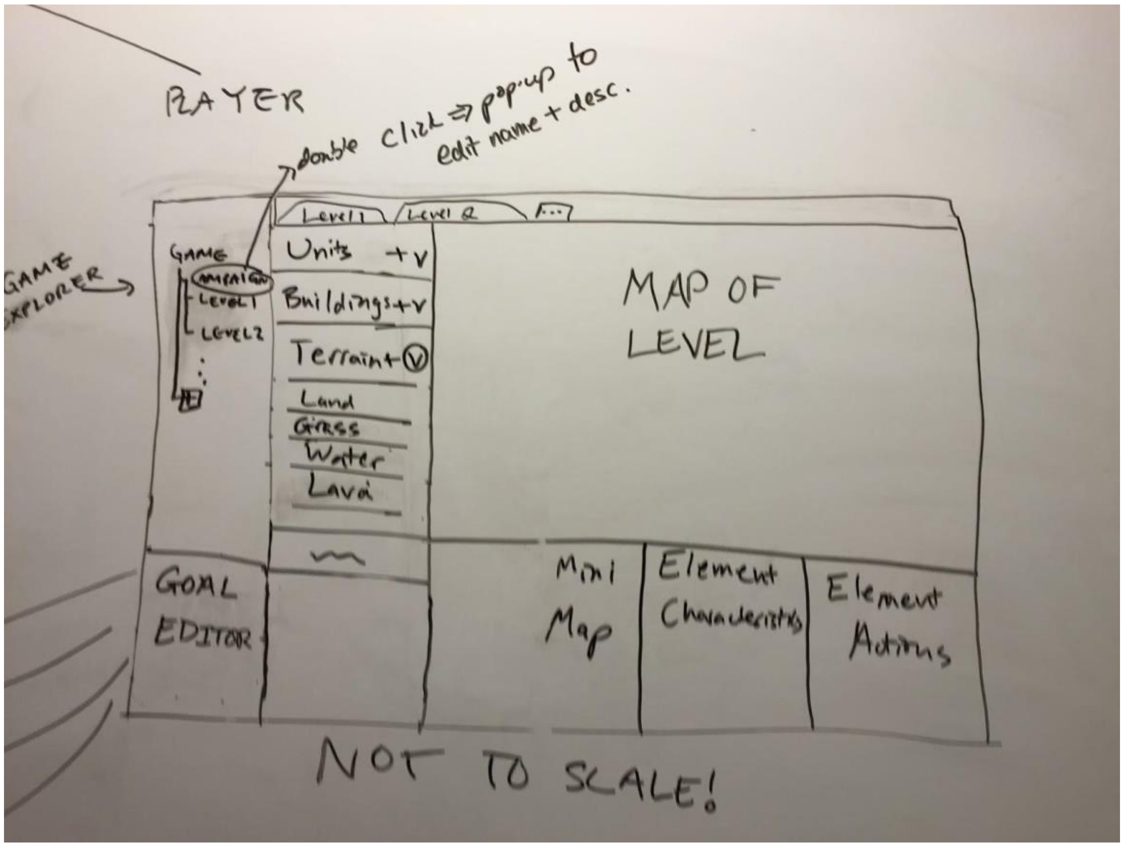

GUIController
(abstract)
- Node getRoot()
- void initialize()
GUIContainer
(abstract) implements GUIController
- void setModel(MainModel m)
- void attachChildContainers(GUIContainer ... children)
- void clearChildContainers()
- MainModel getMainModel()
- abstract void update()
Wizard
(abstract) implements GUIController
- Node getRoot()
- void initialize()
- void setSubmit(Consumer<WizardData> data)
- void addToData(String key, String value)
- abstract boolean checkCanSave()
- abstract void updateData()
- WizardData getWizardData()
WizardData
Map<String, String> myData
List<WizardData> myWizards
String name
- void addDataPair(String key, String value)
- void addWizardData(WizardData wizData)
- Map<String,String> getData()
- List<WizardData> getWizards()

The MainView is initialized when the Main class (see high level design) is run. This is the overarching GUI that holds the Stage representing the program view. When this is instantiated, it has the ability to set scenes that can be held in the MainView, namely the EditorScreen module (interface for editing and creating games), SplashScreen module (interface for choosing whether to edit/play game as well as what game to edit/play), and PlayerScreen module (interface for running and playing the game). The GUI components will all communicate between each other (e.g., switching the scene to be displayed in the MainView) through holding a reference to the MainView and then specifying an enumerated type to switch between view.

These three primary views are all classes which extend the abstract class GUIContainer which holds a reference to the MainModel and have the ability to listen to it. Furthermore, they can delegate any GUIControllers that they hold to be GUIContainers and pass them the model. While every nested pane is a GUIController, such that it has a @FXML initialize() method and holds a reference to its root node, not every controller necessarily should hold the model in order to encapsulate the data and keep the code shy. It is also important to mention that all Java classes within the GUI are merely controllers which have been linked to an FXML file which designates a controller with a tag along with the ids of its elements. Therefore, we can significantly clean up the java code to focus on logic related to the editor and enable the UI layout to be simply specified in FXML and then nest Controllers within each other with significant ease.

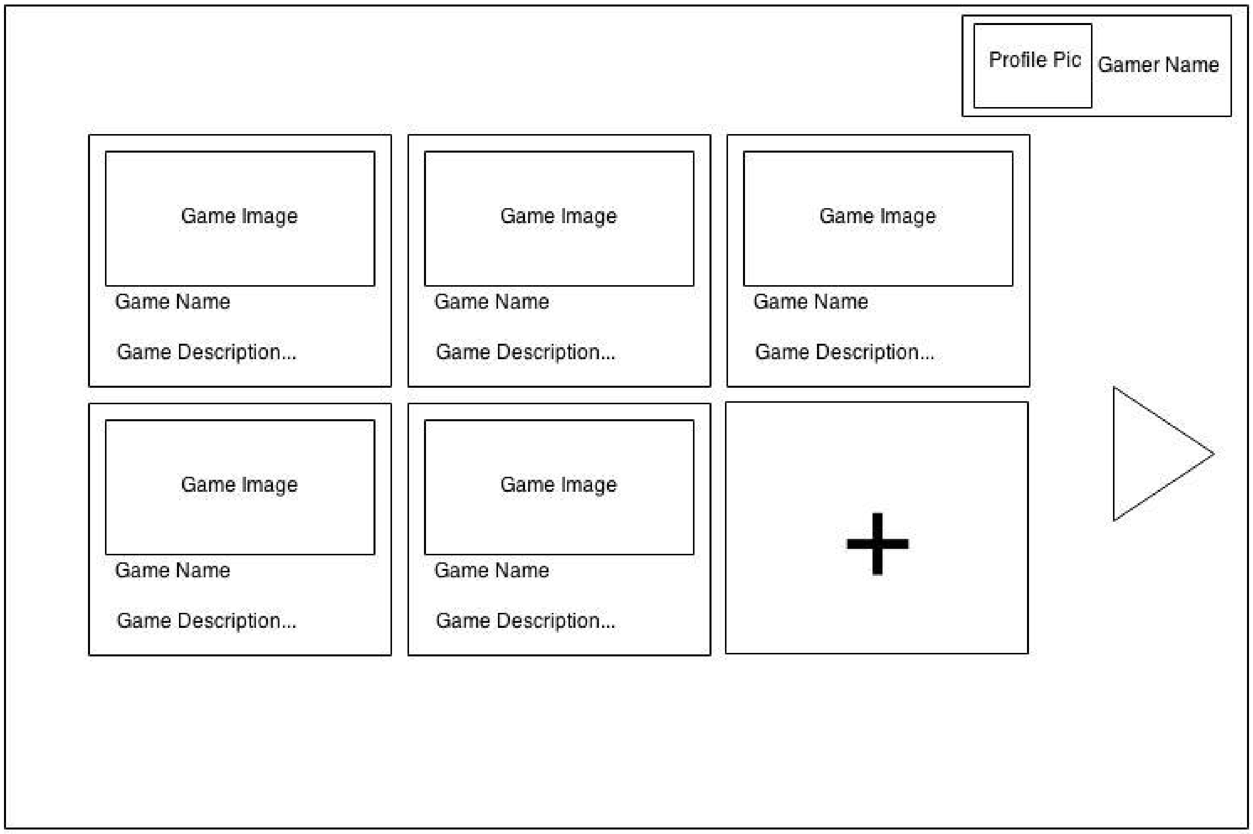
Lastly, we have defined an alternative GUIController type called a Wizard within the Editor. Any Wizard requires a save button and an errorMessage text box. These wizards allow for the creation of any sort of data that will ultimately be sent to the MainModel. This data is stored as a WizardData which holds a Map<String,String> along with a list of WizardData that it hold such that nesting controllers can easily be facilitated. Ultimately, this well designed data structure ensures that user input can be sent to the model in an organized fashion while completely extracting and game information away from the Wizards.

Editor:

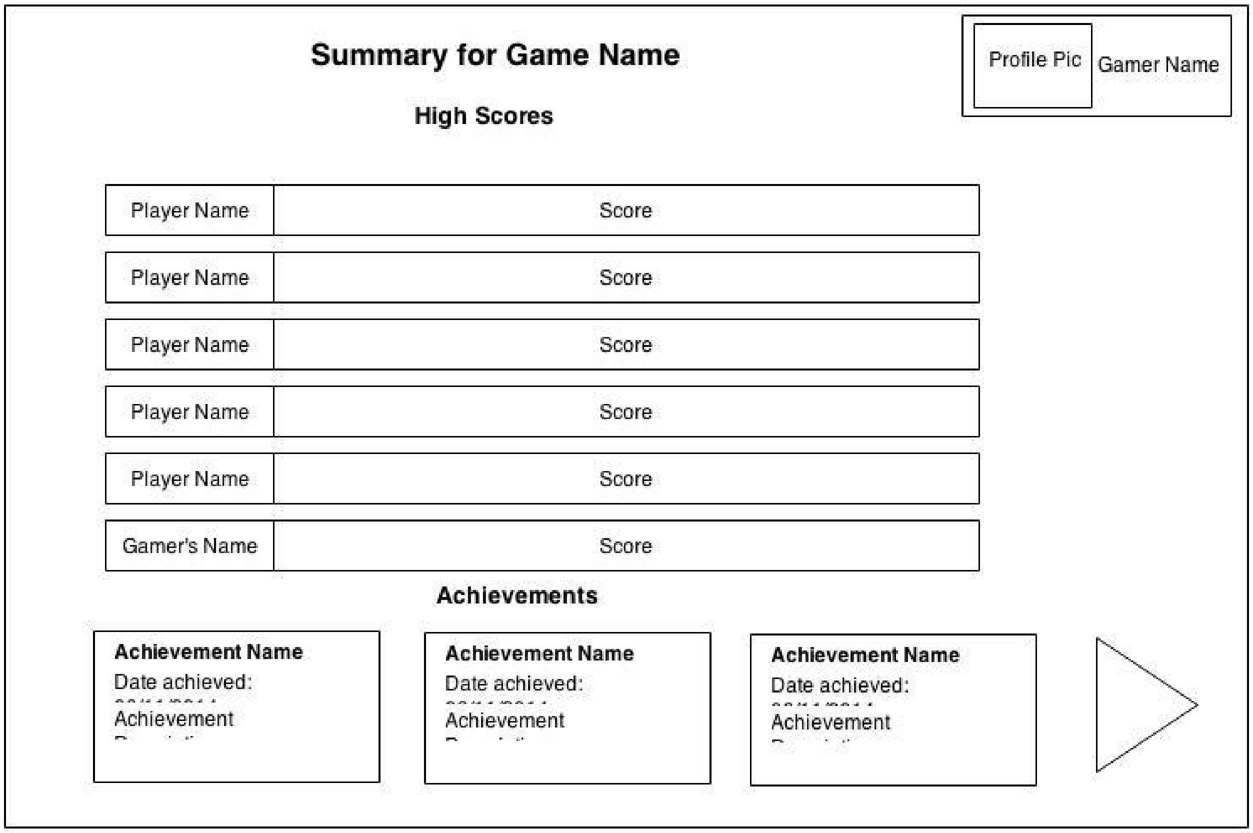


The above drawing represents a basic visual of what the Editor will look like in a Level Editor mode. The far left tab represents a hierarchy of the currently loaded game which has sub hierarchies for each Campaign and the individual Levels. Double clicking on these individual labels should launch a pop up menu that enables the creator to modify the name and description of the level, campaign, or game. Clicking on a Level label should load a new tab into the main tab view of the editor where the map of the level is displayed along with a mini map, an element characteristics display window and an element actions window. Furthermore, each Level tab has an associated accordion folder of Items that can be added to the Level. Each item can be expanded and also features an “add” button which should launch a new pop-up Wizard which enables the user to create a new GameElementState with fields to edit Name, Description, Attributes, along with a fully functional Condition Action Wizard that lets a user create conditions and associated actions based on dropdown menus populated by currently available attributes in the engine.

Player:



This is an image of how the game selector page should roughly look like. If the application user creates several different kinds of games, then each of these games are saved, and the user can come back to this page and select which game he wants to play. Each game is represented by a box containing an image of the game, the name of the particular game, and a small description of the game's goals. There is also the option to add a new game and to scroll through more games if there are more than six.



This is the interface page that displays after a particular game is completed. It displays the high scores on this particular game and the players who earned those scores. There is also a section on this page that lists the achievements won in the game.

Engine:

The Engine is instantiated with a reference to the Model and its first order of business is to attach itself as a listener to and ask for the current GameUniverse and LevelState from the Model. This allows the Engine to operate, as without the LevelState there is nothing to animate. The next order of business is to create a Level from the LevelState, thereby also creating GameElements from their respective States. As this action creates both GameElements and their visual representations, the Engine next uses the Level to instantiate the GameLoop and GameElementManager. An InputManager is created to handle inputs.

The GameLoop holds a timeline which fires the update method every frame. This method utilizes all of the Computers in order to give each Element the other Elements that are pertinent. Currently this includes elements that the game element is colliding with or that it can see currently. Then, the game loop asks each element to update itself. The elements update based on internal state, the elements it is interacting with, etc. This is done through Condition/Action pairs.

The fundamental way by which our game elements interact with each other is through the use of Condition and Action pairs. As mentioned in the Design Goals section, condition/action pairs are created through the use of the VScript scripting language that we created. In creating this language, we have decided that there are two essential types of attributes that need to be contained within GameElements to allow them to interact: string and double parameters. All of these parameters are stored in their respective maps and can be compared by conditions. Through the stringing of conditions, it is possible for elements to react to complex interactions in their environment. VScript syntax for specifying these interactions as well as sample code snippets are shown in Appendix A. It is important to note that user-written code is not currently in the design specification and that the code simply functions as a way to save and send conditions and actions. At their fundamental level, all conditions and actions rely on the idea of a Parameter. A parameter is an object that takes in the current elements of interest around an object and allows for the getting and setting of that attribute. Evaluators then take these parameters and perform specified actions on them. For example, an evaluator for a >= condition will simply perform the >= function on two parameters whereas an evaluator for getNearestElement will contain pre-written java code to perform a complex function on two elements. Above the evaluators lie the operators which allow for the stringing of evaluators, allowing conditions and actions being chained. The real power in this approach lies in the fact that both evaluators and operators follow the composite pattern which allows for complex chaining of evaluators and operators, allowing our game elements to have complex behaviors using a relatively small set of fundamental commands.

Model:

The model will hold all the state for games; the GUI and the Engine both listen to this class to obtain updates to the data they listen on; they may also directly update data. The GUI will be limited as to what it can update, but the Engine will not be restricted.

Several different states will exist:
- GameState, which represents the initial state of a game level
- CurrentLevelState, which represents the state of the game during play
- CampaignState, which represents the state of the campaign the player is playing
- GameElementState, which represents the state of a particular element in the game, such as a soldier

Accessors will exist to obtain the current game state, such as the current game level. There will also be methods to create new instances of game states, such as creating a new level or campaign. The user will then use the UI to edit the new game state to his or her liking.

The Model contains the data leveraged by the editor and the engine. It contains methods for modifying a GameState held in the model, as well as methods to save and load GameState Objects from JSONs.

The Model serves as the fondling point between the engine and the editor. Both the engine and the editor observe the model, and use the model to modify the GameState. For example, in the editor, when the user creates a new GameElement, it gets sent to the model as a package of data, which the Model assembles into a GameElement, which it then adds to its held GameState. Then, if the user were to play the game, the engine would then update the Model to reflect the user's interactions with the game. That way, switching between edit and play modes will be extremely simple, since they're essentially based on the same interactions with the Model.

Save/Load Utility:

The Save/Load utility is intended to take the Model's GameElementState objects (representing things such as terrain, goals, characters) and saving it in the JSON file format using GSON. For resources such as images, the Model invokes the saveImage(Image image, String filePath) method. These save routines are made possible via the following API calls:

```
/**
 * Save a JSONable object to a library file
 *
 * @param object
 *         that can be converted to JSON format
 * @param filePath
 *         to which object should be saved
 * @throws IOException
 *          in case of trouble reading
 */
public void save (JSONable object, String filePath) throws IOException;

/**
 * Save an image from a particular file path
 *
 * @param image
 * @param filePath
 *         at which image is located
 * @return filePath where the image is saved to
 * @throws IOException
 */
public String saveImage (Image image, String filePath) throws IOException;
```

In order to load Java class objects from JSON or media such as images, the save methods have corresponding load methods shown below. The distinction is made between Java class objects that can be recreated from JSON files (loadResource (Class className, String filePath)), whereas the other load method (loadImage(String filePath) can be used to load images at a particular file location :

```
/**
 * Load a resource from a given file and return Java object representation
 *
 * @param className
```

```
  *        Java class object that is being loaded
  * @param filePath
  *        that resource to be loaded from
  * @return Java class object
  */
 public <T> T loadResource (Class className, String filePath);
/**
  *
  * Load an image at given file path
  *
  * @param filePath
  *        of image
  * @return Image object loaded if exists
  * @throws IOException
  */
 public Image loadImage (String filePath) throws IOException;
```

Extensions

Extension 1: Web Deployment

As one of our extensions, we decided to do a web deployment of our project. To do so we built a Ruby on Rails server with a jQuery/Bootstrap 3 front end that delivers the .jnlp file that contains our project and runs it in the browser. This was fairly difficult, as Java's security issues make it difficult to configure a host's as well as a user's security settings to be able to run it in the browser.

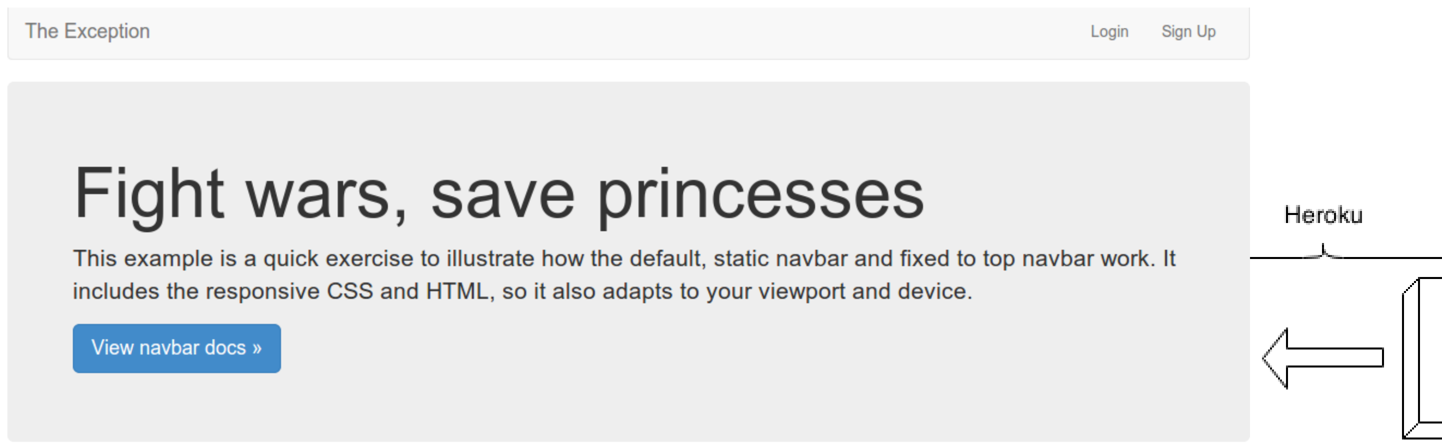
We chose Ruby on Rails since it is fairly easy to add user authentication and easy to host. First, we created the server, and added the Devise gem (*https://github.com/plataformatec/devise*). Devise allows for easy set up of user authentication. Then, we added a SQLLite database to store user information.

For our front end, we chose Bootstrap (*http://getbootstrap.com*) and jQuery (*http://jquery.com/*). Bootstrap is a common CSS framework that allows for quick deployment by relying on a grid system that allows positioning and styling of DOM elements to be extremely simple. jQuery allows us to make our page interactive with some neat JavaScript. Note that Java is to JavaScript as ham is to hamburger.

Lastly, we hosted our server on Heroku (https://www.heroku.com) , which is a cloud hosting platform that allows for easy deployment. In production, Heroku uses Postgres as opposed to our development database SQLLite.

Java security was the main struggle with this extension. To get the site to work,we first had to self-sign the JAR, which could only be done with NetBeans IDE or a command line application. We decided to use NetBeans, since NetBeans also autobuilds your project for deployment. Then, we had to configure my browser to allow Java execution. Last, we had to write list the Heroku deployment to get my Java installation to trust the site as a source of Java.

The result was a fully deployed server, *https://peaceful-earth-5221.herokuapp.com*.
Below is a diagram explaining the stack, as well as a screen shot from the deployment.


The Exception

Login Sign Up

Fight wars, save princesses

This example is a quick exercise to illustrate how the default, static navbar and fixed to top navbar work. It includes the responsive CSS and HTML, so it also adapts to your viewport and device.

View navbar docs »

Now that our server is up, our next goal is to allow network compatibility of our game. This will require the addition of classes to the engine responsible for establishing a network connection and populating the player list based on connected computers. Once a connection is established, syncing game play should be fairly simple as the models will simply need to pass their game representations between each other and

Extension 2: Java and an Xbox controller

As one of our extensions, we decided to use an Xbox controller as input for our game. To do so, we used the JXInput library (http://www.hardcode.de/jxinput) , which allows us to treat the Xbox controller as a joystick. Using this library, we're able to map the Xbox's joysticks and buttons to the control keys.

Extension 3: Java and a Leap Motion

As one of our extensions, we decided to use a Leap Motion as input for our game. To do so, we will use the default Leap Motion Java SDK, which allows us to treat the Leap Motion as a controller. Using this SDK, we will attempt to map the Leap Motion's hand positions to key presses.

Use Case 1

The player controls a soldier. The user sees an enemy soldier far away from him. He wants to fight and kill that soldier using a melee attack. The user clicks on his soldier and right clicks on	The user click on the canvas generates a selection box (which, for the event of a single click, has zero width and zero height). The engine determines which elements the selection

the enemy soldier to initiate the attack.	box touches, and changes their state to selected. This triggers the display of doodads around the animation such as a health bar as well as unit interaction objects in menus around the GUI. The right click is similarly interpreted and decoded to mean an attack as it occurred over an enemy unit. This creates a number of condition-action pairs in the soldier game element. A pathing action pair is created which moves the soldier to the enemy. When the soldier collides with the enemy, pre-existing collision conditions kick in that begin attack.

Use Case 2

The player controls a worker and a lumber mill. He sees some tree resources several steps away from him. He wants to chop down the tree and harvest wood from that tree. Then, he wants to deposit the wood into the lumber mill and have the wood be credited to the player's stockpile. The user clicks on his worker and right clicks on the tree to initiate the harvest.	The same code path as above applies - the worker is delegated a pathing condition to the tree, and another that determines that when he collides with the tree, he will begin to harvest it. When he reaches his carrying capacity of wood, a third condition is triggered and the resulting action asks the the GameElementManager for the closest available place for him to return the wood to - the lumber mill. Another condition action pair is set for him to path to the lumber yard to deposit the wood and then back to the tree to harvest more lumber.

Use Case 3

The player controls a worker. He wants to build a Town Center so that he can produce more workers. He selects an area of ground that fits a Town Center and begins building the Town Center. When the Town Center is complete, he produces a single, additional worker from that Town Center, subtracting 50 food from his stockpile. The user selects the worker, then clicks on an interactive pane in the GUI that enumerates all available buildings that can be built by that worker, selecting the Town Center. Then the user clicks on an available plot of land,	The same code path as before is used to select the worker. When the worker (or any unit) is selected, the unit interaction panes update as necessary to show the unit interaction modes. Clicking on these GUI elements is equivalent to firing off an Action without a Condition being first evaluated - in this case, the engine takes the click into consideration so that the following left click can be handled with the appropriate action. This is done by switching out the left click action in the input controller. The next left click is therefore interpreted as a building

beginning the construction process. Once the building is built, the user selects the building and selects the “make worker” button from the available modes of interaction. This begins the creation of a worker; when the creation finishes, a worker is spawned at the building.	placement and not a selection. The placement of the building also adds a new condition to the currently selected worker - to path to the building site and begin construction once he collides with the building foundation. Once the building is finished (as evaluated with a condition), the worker stops working and the building allows for interaction modes when selected. Clicking on the create worker button similarly adds a condition and action pair into the building’s list - firstly, while unit completion is under 100%, increment it, update the progress bar, etc. When 100% is reached, an action pings the GameElementManager to spawn a correct unit in the location; a GameElementFactory is used to do that by loading a base GameElementState from the GameUniverse held by the Model and building the GameElement from that State. This Action also decrements the total amount of Food in the User meta-attributes.

Use Case 4

The player controls a tower. The player sees an enemy soldier in the distance. The tower should auto-attack the soldier.	Before letting each GameElement update, the GameLoop will find all colliding elements and visible elements and inform them of their status. This way, during the update loop each Element holds as much information as is feasible - each SelectableGameElement will have a list of InteractingElements - elements it is currently colliding with, elements it can see , etc. The tower will be informed by the vision computer of the soldier when the soldier enters its’ visionbox. A condition will always be evaluated on visible elements - if they are owned by an enemy team and inside the range of the tower, an attack will occur.

Use Case 5

The player controls a mage. The mage attacks an enemy unit, debuffing the enemy units' movement speed for two seconds.	When the spell projectile collides with the enemy soldier, it not only decreases his HP but also decreases his movement speed and adds a condition-action pair that evaluates to true only at a certain point in the game clock - two seconds later. The action of that condition is to increase the movement speed of the unit once again to the point it was at before.

Possible Types of Games:

The flexibility of our design as demonstrated in the use cases allows for a number of different types of almost fundamentally different games to be built within our game engine.

The obvious example is a Basic RTS game such as Warcraft or Starcraft where players control armies and workers with a goal of eliminating the units and buildings of the opposing teams. The defining factor to the basic RTS is the ability to manage both buildings and units, as well as controlling resources. It allows the player to control and issue commands to multiple GameElements simultaneously.

Another example would be a Simple RPG (Navigating an obstacle course while having to kill enemies in your way / Hero Quest): the player controls a single unit, potentially a group of units, with dynamic attributes to accomplish a goal such as reaching a specific destination or achieve a specific kill count, obtaining resources or defeating the enemy. Here the user is only able to control a single unit which has the ability to gather resources, but resources can be gained through means other than simply gathering them with workers. For example, a unit could pick up a deposited item on a ground which requires no actual gathering. In addition, the attributes of GameElements are dynamic, in that the player's unit can gain new abilities and augment it's statistics based on the current game state including elements such as kill count, time elapsed, and location. Based upon this, the user controlled unit has a broader set of final goals than a generic RTS, whose sole focus is to usually destroy the enemy, while user control is significantly compared to RTS. However, aside from this, much of the general functionality is very similar to a regular RTS.

A final example would be a Basic Tower Defense: Players build objects (units/ buildings) that are stationary, with dynamic attributes (allowing for upgrading), that react to incoming computer units that travel along a statically set path. For more sophisticated Tower Defense games, the path can be dynamically be determine by the enemy units based upon the locations of user-created units. This requires the ability to change the attributes of default units. For example, units that in the RTS game were able to move to new locations, they need to be able to be set as stationary. In addition, the ability to upgrade units is integral, similarly to in the RPG version. In a simple tower defense game, there is no requirement that a unit creates a new building, because buildings can be created without a user. However, if there is a building-creating unit, this unit lacks the ability to kill computer units, since only buildings have the ability to attack with projectiles.

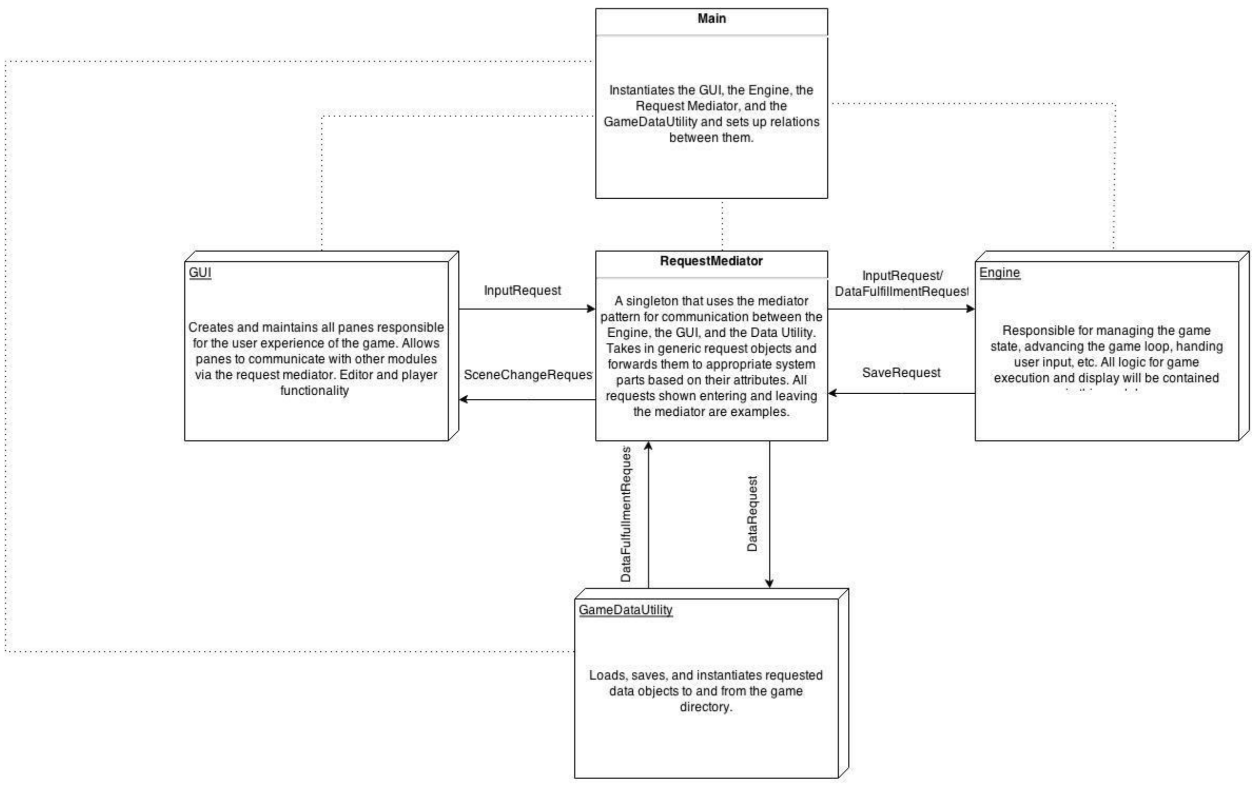
Patterns and Specific Designs:

- Observer/Observable: The model observes the engine and the editor and calls update on them whenever needed.
- Mediator: The individual UI panes talk to a central UI pane manager class which takes care of notifying the Model
- Factory: Factories used to create conditions and actions
- Inheritance: A multitude action classes will be created, which inherit from attribute changing actions and object creating actions. Both of these subclasses inherit from the abstract super-class Action.
- Composition: Used to add requests dynamically.
- GSON: Used GSON to save and load game and object states to retaining information across application opening and closing
- Proxy: We will use a Proxy to restrict the GUI's access to the Model

Alternate Designs

Shifting away from Requests:

An alternate design that we considered, which we originally wrote about, was the usage of requests. We originally considered the use of a request mediator that would handle all communication between the GUI, Engine, and SaveLoadUtility. All communication between the GUI, Engine, and SaveLoadUtility (which is pretty much all communication in general as previously constructed) would be passed through the form of different requests. There would be InputRequests, SaveRequests, DataRequests, etc. The RequestMediator would simply convert the request into another request that would be passed on to either the GUI, Engine, or SaveLoadUtility, which would then handle it.


Main
Instantiates the GUI, the Engine, the Request Mediator, and the GameDataUtility and sets up relations between them.
GUI
Creates and maintains all panes responsible for the user experience of the game. Allows panes to communicate with other modules via the request mediator. Editor and player functionality
InputRequest
SceneChangeReques
RequestMediator
A singleton that uses the mediator pattern for communication between the Engine, the GUI, and the Data Utility. Takes in generic request objects and forwards them to appropriate system parts based on their attributes. All requests shown entering and leaving the mediator are examples.
InputRequest/
DataFulfillmentRequest
SaveRequest
Engine
Responsible for managing the game state, advancing the game loop, handing user input, etc. All logic for game execution and display will be contained
DataFulfillmentReques
DataRequest
GameDataUtility
Loads, saves, and instantiates requested data objects to and from the game directory.

This design would have originally allowed for each part to be independent and closed. We started to discuss what kinds of information would be required in the request, and eventually realized that there was no point to having a RequestMediator - we could simply send the information directly without re-implementing Java type-checking and exception-handling or rolling our own message queue. In addition, due to the introduction of the Model, we completely changed how information was tracked and sent.

Engine Alternate Design

We originally said that we would have GameElements compose everything in the game (units, terrain, goals/objectives, etc.) under the idea that they would all be united in having the condition to action map and name to attribute map. This would have made sense even for something not physical like a goal because at its core, a goal is simply winning the level (or completion of a certain objective within a level) when a certain condition has been met - thus condition to action. However, we realized that there were some fundamental differences that were just too large to ignore. There are GameElements that are Drawable, which must also have an image and some kind of visual representation and animation. There are also a subset of DrawableGameElements that are Selectable, which are fundamentally different from DrawableGameObjects because of they are the ones that react to collisions.

View/GUI Alternate Design

For the View/GUI team (Jonathan, Nishad, Rahul, Josh), one design question that quickly arose in terms of acting as an observer of the Model was how exactly we would get every piece of the GUI to update when the Model tells the View to update. One possibility we came up with was to allow each each component of the View to have a reference to the MainModel in order to allow them each to directly call methods on the model and be observers of the model. While this would make writing the code easier (attach all the things as observers and then directly query the model for info when notified and tell the model to update), it would make the panes less reusable (they would only be applicable in the context of being an observer of an instance of our program's MainModel). Thus, we decided to go with the current design. In order to be as modular as possible (to allow full flexibility in changing layouts (only having to change the FXML file that placed the subpanes) and would also allow us to reuse GUIPanes (FXML and Controllers for them) and even in different contexts). In order to implement this schema, we essentially divided our GUI componenets into 2 types: GUIContainers and GUIPanes. GUIContainers (e.g., the Entire EditorScreen) held GUIPanes, while GUIPanes were a single thing (e.g., the MenuBar). GUIContainers were then mostly for organizing multiple GUIPanes and would have a reference to the MainModel, while GUIPanes would not have any concept of the MainModel. This allows the GUIPanes to be general by passing them lambdas for what to do when certain buttons are pressed.

Data Alternate Design

An alternate design that we considered was having the Save/Load Utility being observable by both the frontend and backend. In this way, any changes in the data saved or loaded would notify observers (editors and engine).   The primary setback with this design was the discovery that functional interfaces could not be "GSONified."   Due to this limitation, it was important for us to devise an alternate method for representing condition/actions represent separate "state information" that can be converted to and from GSON with those that cannot, such as the visual components rendered during game play and game editing (JavaFX nodes). Another alternate design for saving and loading functional interfaces (lambda expressions) was using Java's Serializable class. However, we rejected this alternative as well because adding a Serializable cast due to our functional interfaces would not make for very good design. A third alternative to which we gave a passing thought was trying to generate classes on the fly. In order to workaround the inability to "GSONify" lambda expressions, we decided to represent condition/action pairs as strings that would then be parsed by VScript, our scripting language on the backend.

Team Roles

Component	Team Members
Game Engine	Names: ● John Lorenz - Terrain and I/O interaction ● Michael Deng - Actions and Game Loop ● Steve Kuznetsov - Animation, Spritesheets, etc ● Zachary Bears - Input event handling, Condition creation, action creation ● Stanley Yuan - Controlling ● Michael Ren - Writing specific default action/condition pairs and game resources files Responsibilities: ● Java class framework for playing RTS games ● Runs game loop, updates internal game element states, and responds to requests from GUIPane events ● Uses a system of conditions and actions to continuously update, create, and delete different states and element in the game. ● An observer to the model. Updates whenever the model tells it to update.
View/GUI/Editor	Names: ● Jonathan Tseng - Project explorer, Accordian panes, GUI loading, positioning, language support ● Nishad Agrawal - GUI loading, FXML, positioning, Wizards ● Josh Miller - Wizards ● Rahul Harikrishnan - scene switching for different views Responsibilities: ● Creating the framed layout of the View, including a Splash screen, an editor screen, and the game runner screen ● Modular GUIpanes that allow placement and reuse of GUI objects in different contexts with different actions ● Polling info from the model (on update) and updating the view based on the info in the model (Observer-Observable relationship)
Model	Names: ● Jonathan Tseng - MainModel, DescribableState (Game/Level/Campaign) hierarchy ● Nishad Agrawal - GameElementStateFactory, GameElementStateHierarchy ● Rahul Harikrishnan - GameUniverse Responsibilities: ● Holding all savable (JSONable state) in wrapper classes ● Implementing observer-observable pattern for model (observable) and view and engine (observers) ● allowing queries for information to allow view and engine to update

Load/Save Utility	Names: ● Rahul Harikrishnan - All Responsibilities: ● Allow for loading and saving of GSONable data objects (non-Java FX, functional interfaces) as well as loading and saving of media (such as images).

Appendix A: VScript syntax

Below is the syntax for conditional statements in VScript

Subjects

this -> refers to the current element that is having a conditional evaluated on it

other -> refers to any other object other than the current element. As of right now, this can be any lowercase string that is not “this”

Object -> A capital first letter refers to a global element or a special object. i.e. Tank will expose attributes of all the tanks (currently only used for counts) and Game will refer to the game object and will allow for the referencing of important game attributes like time elapsed

Attribute Grabbing

in the below examples, x is a subject. Subjects have 2 possible value types: doubles and strings. To get a value, reference it by its tag. If the tag does not exist, the value will be set to 0 or “” for doubles and strings, respectively.

x.double(“ValueTag”) -> gets a double

x.string(“ValueTag”) -> gets a string from a value

Supported Logical Operators

&& || < > <= >= == !=

Notes:

One space must exist before and after each logical sign to ensure easy parsing

Example Code:

```
this.double(“health”)  == 5 && other.string(“type”) == “tank”
Tank.double(“number”) >= 0 || this.double(“magicalBullets”) == 1
```

Types of Actions

The syntax for actions will be essentially the same for conditions except for logical operators being replaced by the names of actions. After examining many possible features that need to be supported in our games, we have identified three main action types that need to be supported by VScript:

-change attribute

-add/remove condition action pair

(each of these above two will have three things they can apply to: the object of interest, another object, or a global set of all other objects)

-make an object

To make user use of actions simpler, we will write all the actions that need to be evaluated on parameters. Examples of these are “+=”, “findClosest”, “navigateTo”, etc.

The syntax for an example actions are shown below:

```
//Inflict damage on an enemy

enemy.double(“health”) -= this.double(“strength”)/enemy.double(“armour”)

//Move to the given enemy

this moveTo enemy
```