VOOGASalad, The Exception: Design Plan

Genre: RTS, Real-Time Strategy

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 objects 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 objects.

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 objects. 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

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.

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 aspects, and their opponent's units, each with different levels and limits of interaction.

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 different commands to be interpreted. 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 canvas with a bunch of editing UI elements, where each of the modules on that canvas have certain levels of interaction with one another. A module is defined as a unit, the player, non interactable objects in the environment, resources, projectiles, etc. Each one of these modules should be placeable and editable by the player on a number of different levels. The player should be able to dynamically create and place one of these modules where they see fit. They should also be able to change and edit physical values about each of the modules, such as its color, size, strength, health, who the unit belongs to, 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 game objects. We ultimately want each object to be able interact with each other module in any way we choose and in order to do this, each one of our modules will have detailed methods and classes that help define how modules interact with others. In our UI, we will give the user the ability to change how certain certain units and environment blocks can interact with each other. Furthering this capability, the UI will also be able to edit how projectiles affect other modules and how much control the player will have over certain components.

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.

Communications between each pane and other portions of the program will be handled via the mediator pattern by using request objects and the request handler. This design was created with the goal of allowing the editor and player to be easily adjusted and have additional functionality added without a severe restructuring of our program over the course of the development process.

Similarly, in order to support the handling of user interaction events in any way that the game designer specifies, a request system will be implemented. When the user interacts with the GUI, a Request is created that will be redirected to the correct part of the program that is equipped with the

tools necessary to fulfill the request. How the request is generated will always be identical; how the game designer decides to specify that the game engine responds to the request can and will be flexible. Furthermore, each object in the game will hold a mapping of conditional functions to reaction functions. This mapping will be able to be specified when units are created in the editor, and will be loaded from their JSON encodings. During the update phase of the game loop, each object will run all conditional functions and use their output to determine which reactionary functions it needs to run as well. All objects 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 XML to encode their GUI panes that they construct. 3) we assume that the users will use JSON to encode their game objects if they are creating them without the use of the editor.

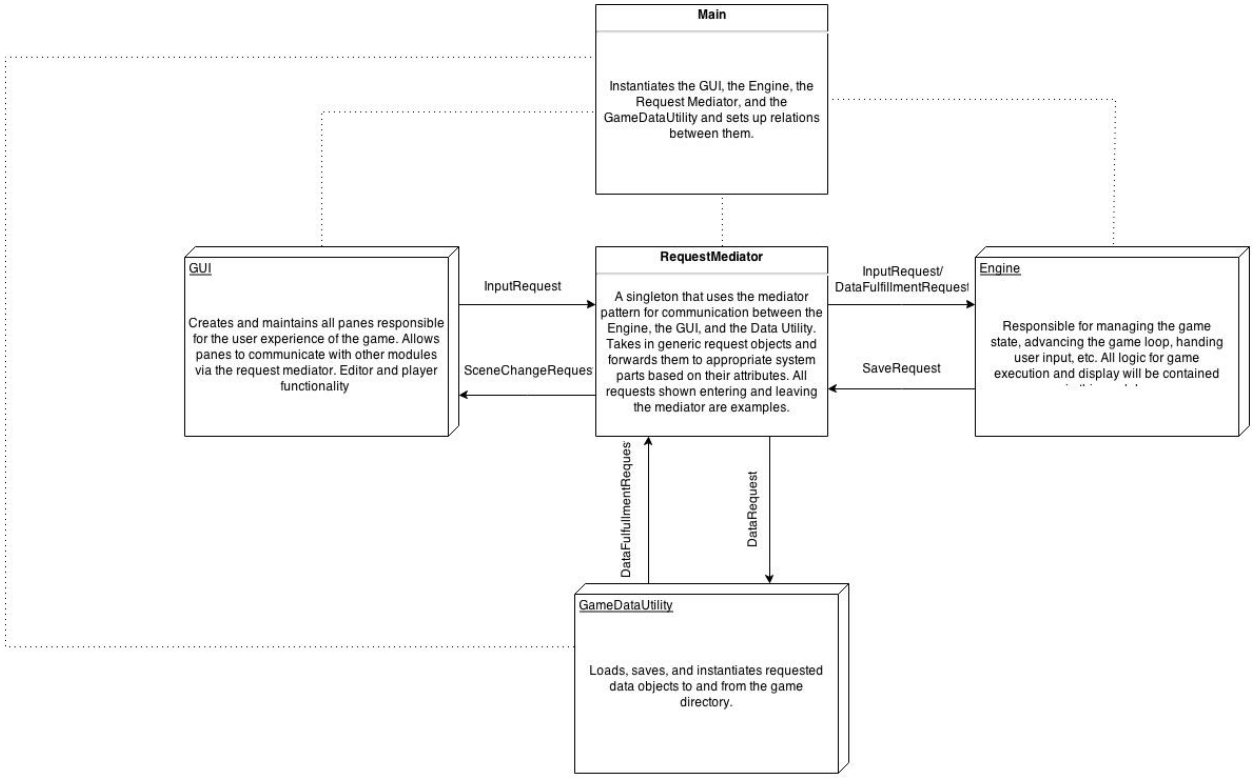
In order to play a game, a number of representations will be created. Firstly, XML representations of all of the GUI panes will be created. JSON representations of all game objects will also be created. These will be parsed into the game and represented as JavaFX nodes if they are drawable or as instance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

In terms of game mechanic restrictions, we have started with a fairly broad allowance for the interaction of game objects within the game. Through our extensive use of maps of lambda functions, we are initially allowing for any object to have any number of user-defined parameters as well as being able to interact with any of the user-defined parameters of other objects. In our interactions between objects, we have removed the temporal aspect of interaction, i.e. one object can not see previous states of another object but can rather only see the other object's current state. In addition, we have restricted objects as being defined to have one owner class and have not included an allowance for multiple owners of one game object. In our design, we have also restricted an object's knowledge of the objects around it to only allow an object to know about the objects colliding with it or contained within its bounding box.

In addition to object 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 a top-down 2D scrolling camera and assumes a finite world size.

Primary Modules and Extensions Points

The High Level Design

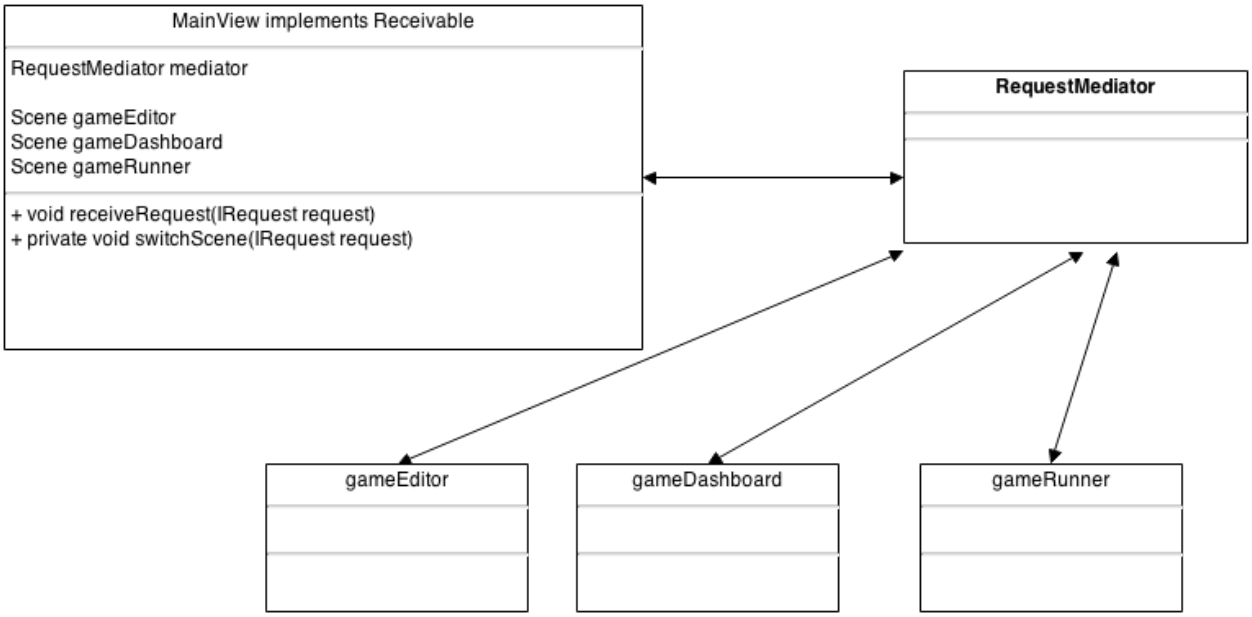


When the game is first started, the first class that is run is Main. Main is responsible for instantiating the GUI, the Engine, the Request Mediator, and the Game Data Utility. It is also responsible for establishing all the relationships between each of these four components. The request mediator is the primary class responsible for delegating tasks among the GUI, the GameDataUtility, and the Game Engine. Each of these three big sections need to both send and receive information from the other two, so we implement the mediator design pattern using the RequestMediator Class in order to decide where requests are coming from and where they are going. What the RequestMediator takes in is a generic request object and what its spits out is a specific request object.

The GUI is the physical interface of the project and is the display for the main splash page, the actual game, and the game editor. It allows all of the panes in the scene to communicate with other modules using the request mediator. For example, if a particular button event is fired in the GUI, its action is sent through the RequestMediator to the Engine to handle the button click. The GameDataUtility is the area of our application where we save, load, and instantiate requested data objects from the game directory. Whenever a particular state is saved in the GUI, it is sent as a request through the RequestMediator to the GameDataUtility to be saved as a file. When the Engine needs to run, its reads in the state it needs to be in from the GameDataUtility using the same request method. Finally the engine acts as the backend for our application and deals with all module interaction. It is

here where the animation loop is run, which calls update on all the modules to determine movement, collision, event handling, etc. It is in this section where all game logic and execution is contained, and then the scene is returned to the GUI as a request through the RequestMediator.

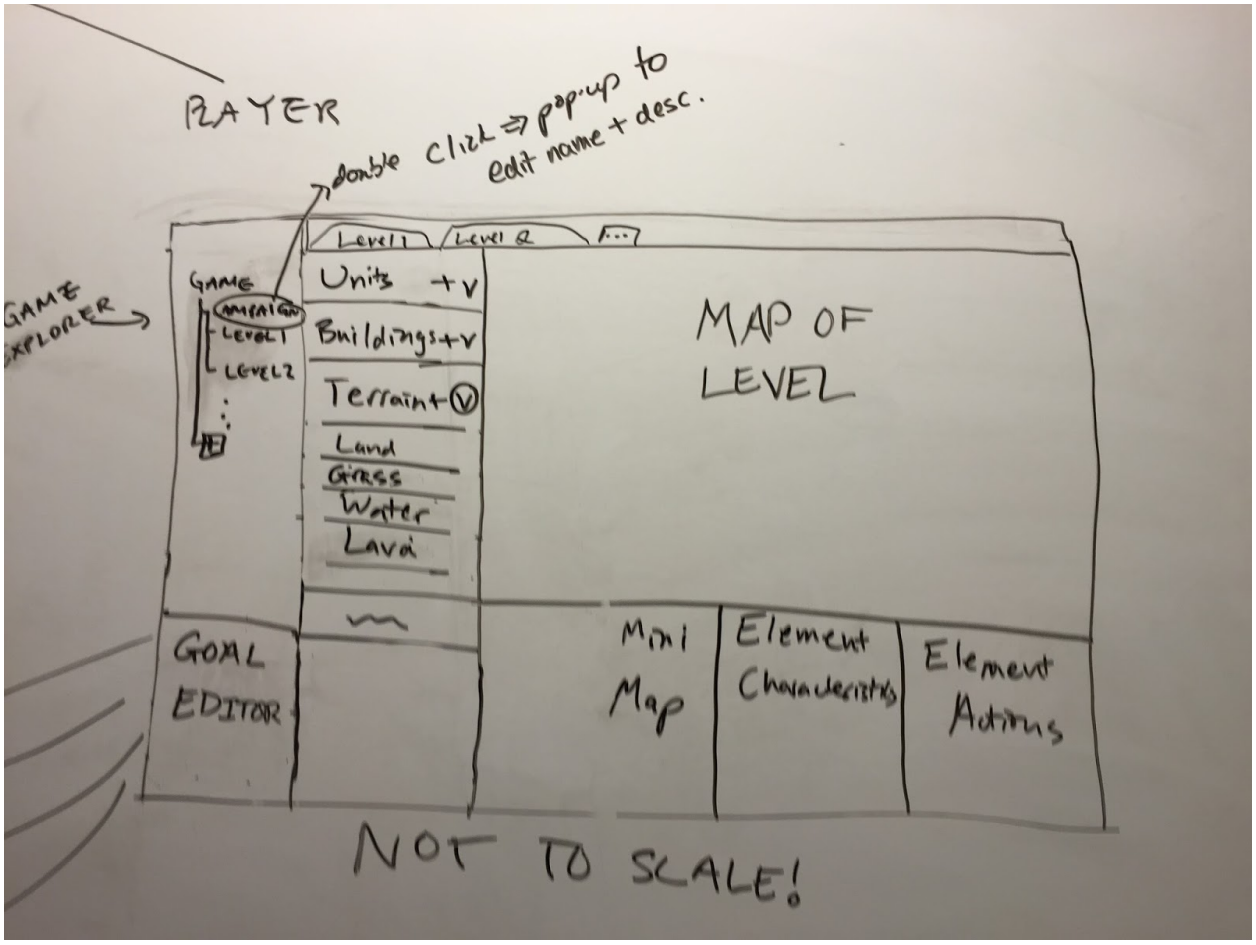
The GUI Design



The Main GUI is initialized when the Main class (see high level design) is run. This instantiates the MainView--the overarching GUI that holds the Stage representing the program view. When this is instantiates, it also instantiates the scenes that can be held in the MainView, namely the GameEditor module (interface for editing and creating games), GameDashboard module (interface for choosing whether to edit/play game as well as what game to edit/play), and GameRunner 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 the RequestMediator by sending and receiving requests.

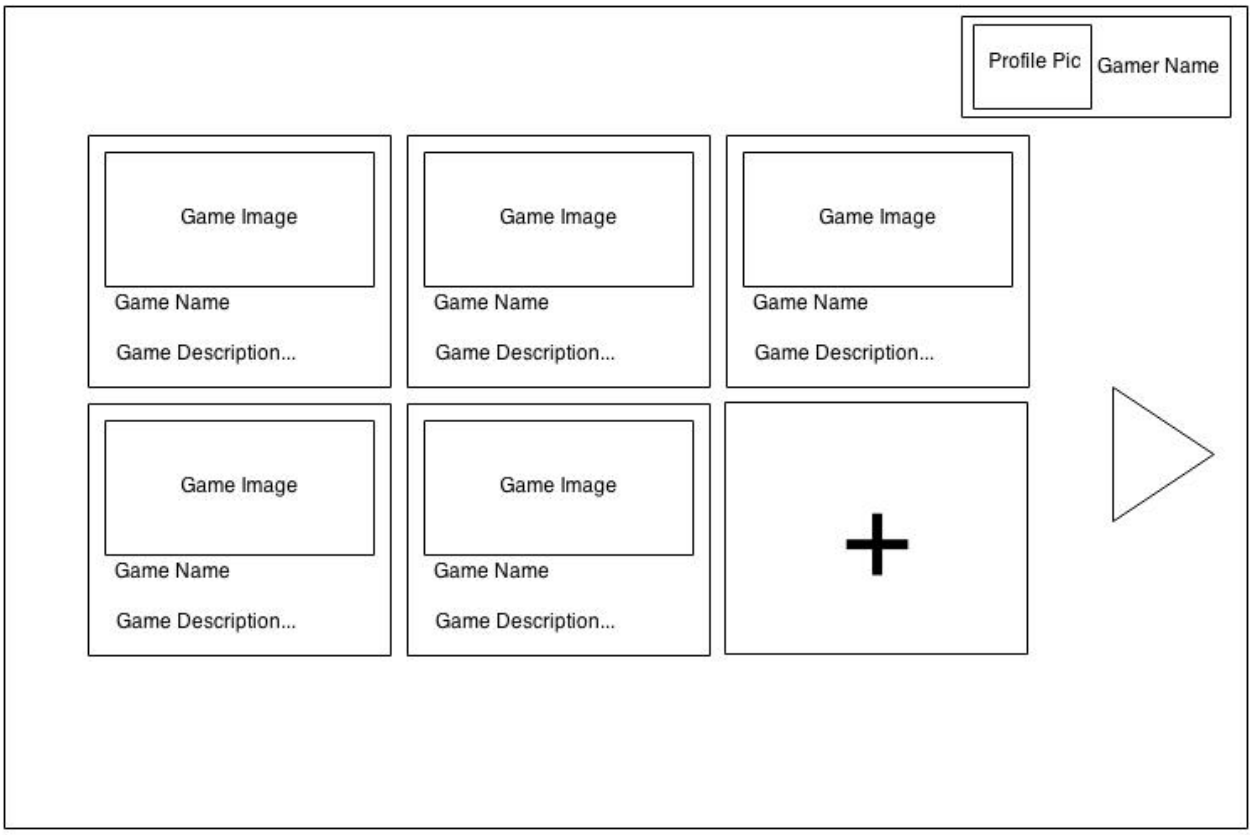
GUI Design Mockups:

Editor:

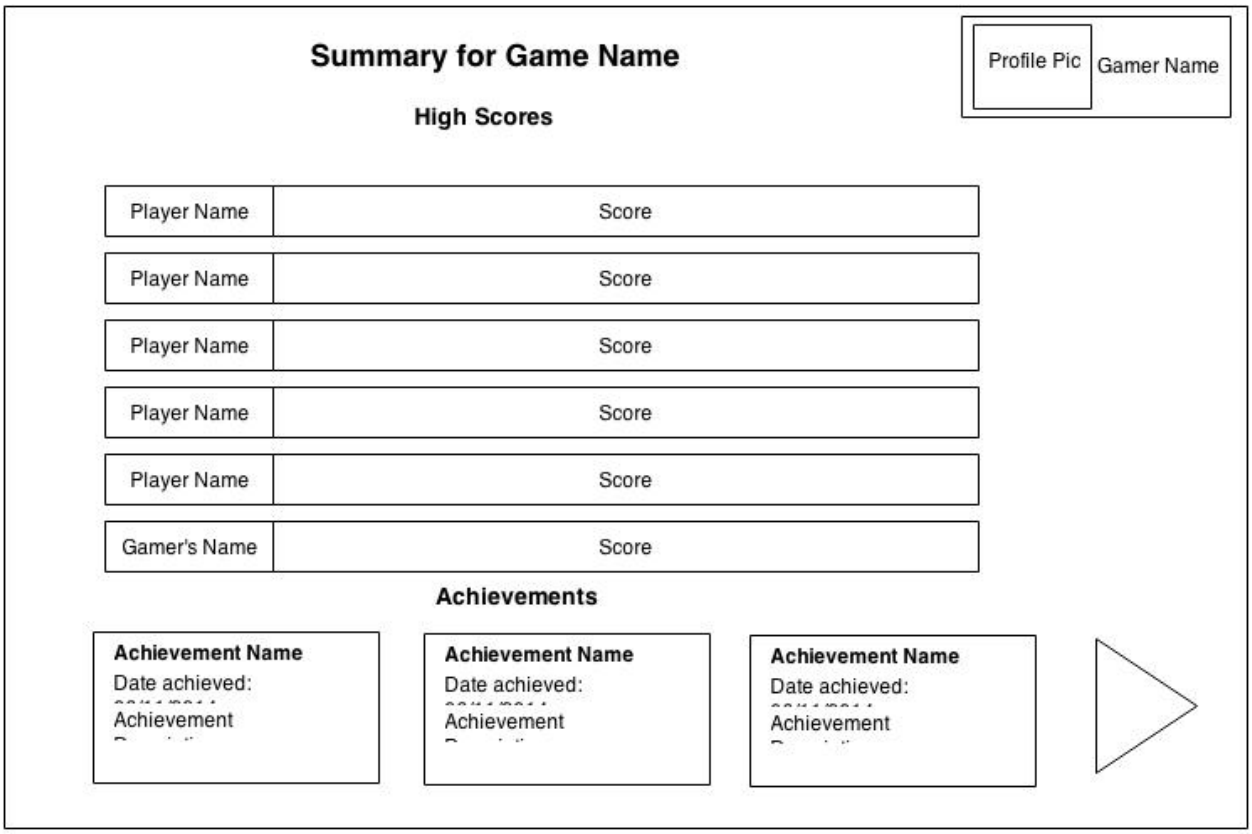


The above drawing represents a basic visual of what the Editor will look like in a Level Editor mode. The far left tab represents a hierarchy of the currently loaded game which has sub hierarchies for each Campaign and the individual Levels. Double clicking on these individual labels should launch a 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 window which enables the user to create a new GameElement with fields to edit Name, Description, Attributes, along with a fully functional trigger editor that lets a user create conditions and associated actions based on dropdown menus populated by currently available attributes in the engine.

Player:

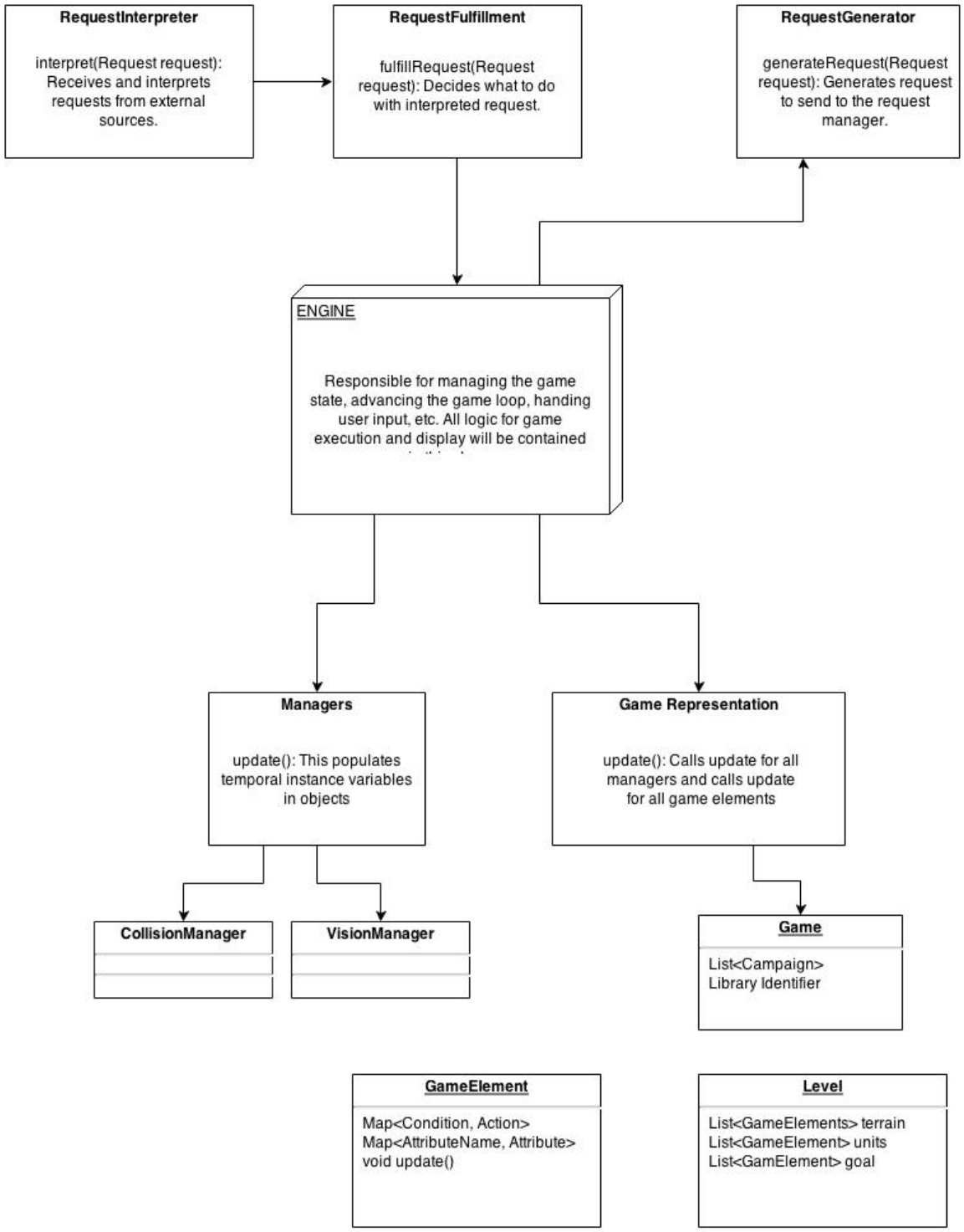


This is an image of how the game selector page should roughly look like. If the application user creates several different kinds of games, then each of these games are saved, and the user can come back to this page and select which game he wants to play. Each game is represented by a box containing an 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.

The Game Engine Design



The Game Engine contains the internal representation of the objects in the game. When initialized, it uses factories to create all GameElements that are represented, as well as all managers and levels that are defined in the given files that it loads. All objects in the game are represented by a

GameElement (including units, buildings, terrain). Each of these elements contains maps of actions to execute based on certain conditions, a map of attribute names to their values, and contains an update method. All of these are defined in the data that is loaded. Part of the Element data is it's current visual representation and location on the map. This map is the canvas visual component that is passed to the UI elements to display.

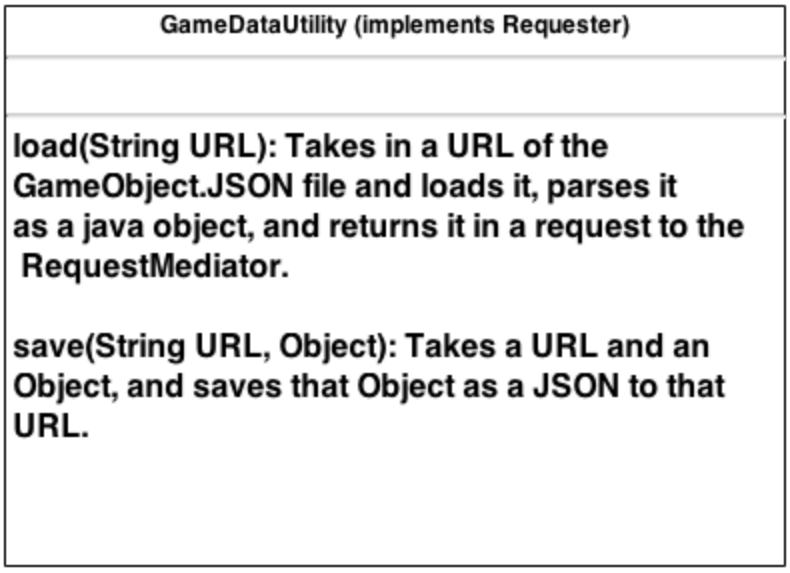
Elements get their inputs in the form of requests. Requests come from the UI and from the Data Storage. For example, the UI sends a request to the engine when the screen is clicked by the user. The request interpreter decides in what way the request is to be handled, while the requestFulfillment class defines how each request is actually to be handled. Each GameElement will define in it's update() how it responds to requests and how it's state changes from frame to frame.

The updating of the game's state is defined by the behavior in the game engine. Once a game is running in the GamePlayer, for example, the logic for how objects behave and react to each other is all defined in the engine.

Each game is controlled by a Level. The level will contain the necessary definitions of the game's terrain, units, and goals, all of which are dynamic and can be changed by incoming Requests or events during gameplay.

Utility Modules:

The GameDataUtility Design



The load method takes in the URL of the JSON to be loaded, parses it, and sends a request object with the parsed JSON as a payload out. The save method takes in a URL and an object, parses that object to JSON, and saves it to that URL as a .json file.

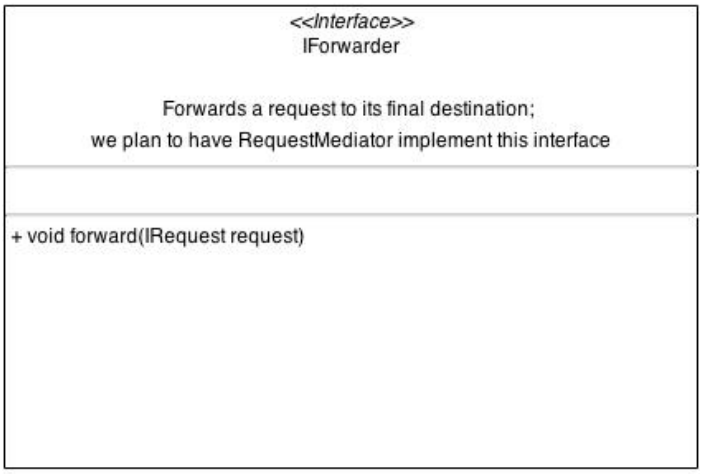
The MultipleLanguageUtility Design

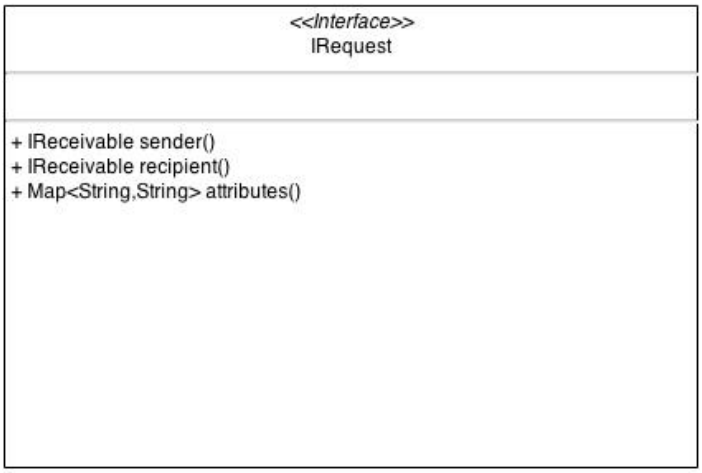
Allows the user to switch their game's languages- for example, switching directions from English to French.

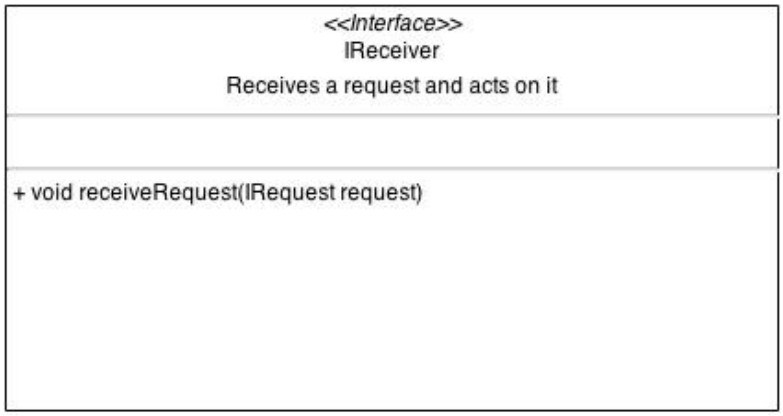
The FXMLElementGenerationUtility Design

Generates FXML elements via JavaFX resource reading

The RequestManager Design







Takes a Request object and decodes where to send the Request to, and sends a casted version of that request to the proper authority. Some examples of requests include Engine requests, Save/Load Requests, and GUI Requests.

Exceptions will also be handled through the Request Manager--whenever a Requestable component has an exception (e.g., stop the program), then the Request will take priority over the rest of the requests and tell all of the other components that an Exception has taken place and they should handle it.

Example Code:

Game Example 1:

Basic RTS game (Starcraft, Warcraft): players (including AIs) control armies and workers to gather resources with the goal to eliminate all Objects, both units and buildings, of the opposing team/ players

In the basic RTS game, the player's GameElements include units that can move and can be issued commands, buildings that have abilities and/or attributes. These attributes can be dynamic, and each Object can have separate abilities. For example, resource-gathering units can be issued commands to go to map-defined Resource objects and decrement the resources from the object while adding resources to the player's count. In addition (or alternatively), units can have the ability to fight against other enemy units. Fighting units have the abilities to fire projectiles which can result in various actions being performed when collided with other units. These projectiles could also include things such as spells which are invisible but have the ability to modify attributes of Objects such as Buildings, the Terrain, and other Units. Buildings can have the ability to create new Unit Objects. The player can select multiple units at once and command them to move to new locations. Units can have the ability to interact with other units via 'abilities' (attribute allowing them to attack enemy units).

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 Objects simultaneously.

Game Example 2:

Simple RPG (Navigating an obstacle course while having to kill enemies in your way / Hero Quest): 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 Objects 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.

Game Example 3:

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.

In order to implement a these games, the game engine will need to operate on all of the constituent GameElements that make up the games. A GameElement is a ubiquitous object that will hold a map of conditions to actions, a map of attributes with their names, and an update method. The conditions and actions are lambda functions; the update method will iterate over the attributes and evaluate appropriate condition functions and their paired action functions while doing so.

In order to implement a unit such as a knight, a GameElement will be composed as follows: The attribute map will hold information such as the owning player, the health and mana points, damage, armor, attack and movement speed, that it is a land unit, and temporal variables such as the current list of other elements that it is colliding with or units that it can see, *etc*. The map of conditions to actions will define behavior such as: if the knight is attempting to move into water, stop movement; if the knight is colliding with an enemy, attack; if the knight can see an enemy, run at it in order to attack; if the knight is colliding with a friendly healing projectile, to increase health. The knight will also have condition functions that determine if it should respond to attributes of the terrain it is currently inhabiting.

In order to implement a terrain element such as a patch of grass, a GameElement will be composed as follows: The map of conditions to actions will be empty, as the grass cannot act. If the grass is long and difficult to walk through, or someone has cast a slow spell in the area, the grass attributes map will hold a speed modifier. The grass allows units to interact with these modifiers at their own will by using their own sets of condition functions. The update function of a terrain GameElement will be empty.

In order to implement a goal, a GameElement will be composed as follows: for instance, if there is a goal that expires after a certain amount of time, the map of attributes will simply hold the current time elapsed. The map of conditions to actions will hold one condition-action pair, where the condition would evaluate to true if enough time had elapsed and the action would be to end the level, *etc.* The update method would evaluate the condition and act appropriately.

In the engine, the animation loop will have the general structure of :

1) updating temporal instance variables in all GameElements
2) tell all GameElements to update()
3) respond to user requests

Alternate Designs:

Game Loop vs. Event Loop:

Originally, this discussion involved deciding between using a game loop to handle the changing state of the game versus using an event loop to handle the changing state of the game. We realized that, for an event loop, we'd be pushing events onto a queue that would be executed in the order they arrived. This is problematic, since the order in which events are executed is dependent on the time at which they were pushed into the queue. That means there may be unexpected out of order executions. For example, lets say two soldiers collide. It's reasonable that the collision events thrown by each of the soldiers might end up being pushed into the event loop in a fairly arbitrary order- meaning, one soldier might register the collision, and act on it before the other does. If, instead, we stuck with a game loop implementation, the soldier's collisions would be executed in a definite order. So, if one soldier collided with another, and was meant to bounce back, by the time the other soldier updated, it would not again act on a collision.

JSON Data vs. Other data types

JSON data was compared to a few other ways of storing data, including XML. JSON, overall, is much quicker to write by hand, and easier to understand by inspection. It is also in general less character intensive than XML, meaning smaller files to transmit when we end up doing multiplayer. If this was a simple file, we'd imagine that the file size difference would be negligible for the same data. However, we imagine our GameObject.json files containing quite a large class, meaning that the file size difference will be significant.

2 GUI (Player and Editor separate) vs 1 GUI

Initially we discussed creating separate Game Player and Game Editor GUIs (started by separate Mains). This would low create a separation between the way that games are actually run (implemented in the player), and how game data is edited visually (in the GUI). However, we decided that this separation was ultimately unnecessary. The outward extensions for the Game Engine that specify how units move should be common, and the Game Player is functionally a part of the Game Editor, just with the ability to also start the game. This led us to the conclusion that it would result in less copied code if the Game Player and the Editor were part of the same GUI, which is now the GUI Pane Holder.

1-Way Requests (GUIPaneHolder -> RequestManager) vs. 2-Way Requests(GuiPaneHolder <-> RequestManager)

Originally, requests from the GUIPaneHolder were one-directional. Using JavaFX property binding, the engine would obtain references to dynamically modifiable fields within the editor (such as text boxes) at the beginning of the program. In this way, the engine would be able to appropriately set the information in these fields based on information passed to it from the GUIPaneHolder via the RequestManager. However, we realized that we needed a bi-directional reference. Specifically, in the case that the engine needs to notify the individual GUIPanes to return to the game dashboard (high scores, choice of games, etc). This is why the decision was made to have 2-way requests.

Team Roles

Component	Team Members
Game Engine	Names: ● John Lorenz - Request Handling ● Michael Deng - Game Loop ● Steve Kuznetsov - GUIPane Requests ● Zachary Bears - Input event handling ● Stanley Yuan - Controlling Responsibilities: ● Java class framework for playing RTS games ● Runs game loop, updates internal game objects states, and responds to requests from GUIPane events ● Editor Controller vs. Player Controller (Liskov Substitution Principle) that gets swapped out based on editor/player mode due to different kinds of interaction
Map Editor	Names: ● Josh Miller - UI Design

	● Nishad Agrawal - UX Design ● Jonathan Tseng - Communication Responsibilities: ● Develop editor for game creation ● Develop Front End utilities for generating GUI elements ● Interacting with load/save utilities ● Communicating with engine
Game Player	Names: ● Zachary Bears - FXML pane creation ● Stanley Yuan - UI Design ● Michael Ren - Request handling for game player Responsibilities: ● Creating GUIPaneHolders and GUIPanes for game play ● Interacting with load/save utilities ● Communicating with engine
Game Data	Names: ● Rahul Harikrishnan - JSON Design ● Michael Ren - Request passing / communication ● Joshua Miller - Save and Load Utilities Responsibilities: ● Develop JSON data structure and communication between multiple players