API Design Document v2: Omega_Nu

Leeviana Gray - Data control
Kevin Jian - Backend grid management, team manager
Chris Murphy - Front end, GUI, splash screen
Patrick Schutz - Front end, GUI
Brooks Mershon - Editing views for units, items, actions
Andy Bradshaw - Combat, Units, Stage Controller
William Li - Front end for grid
Ken McAndrews - Backend grid management
Carlos Reyes - Combat, units, stage controller, pathfinding
Vincent Wang - Units, stage controller

Genre

Turn based strategy of certain predefined types (e.g. Fire Emblem, Farming Simulator, Final Fantasy Tactics).

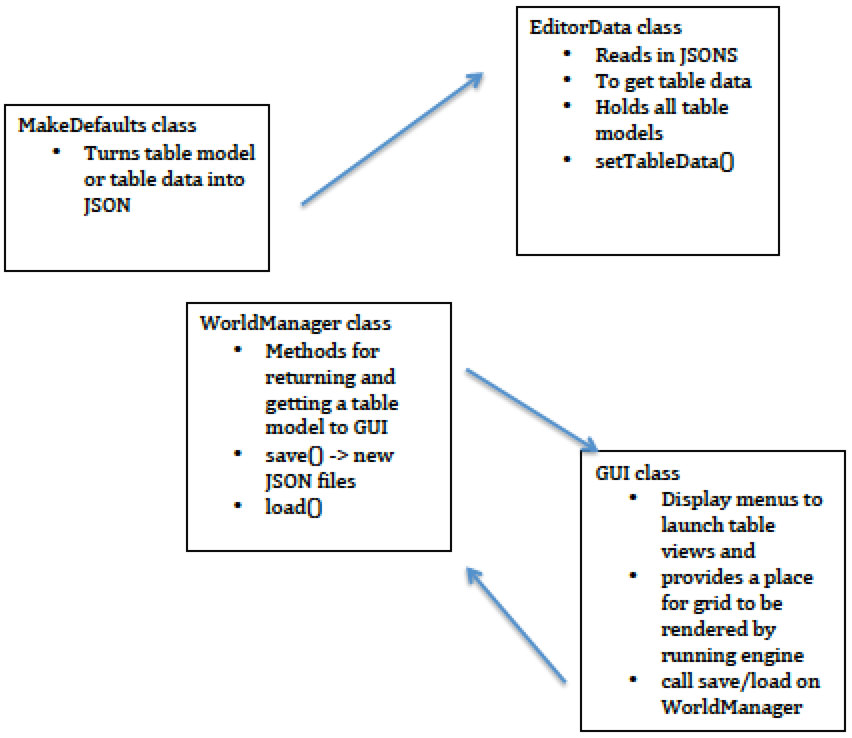
Design Goals

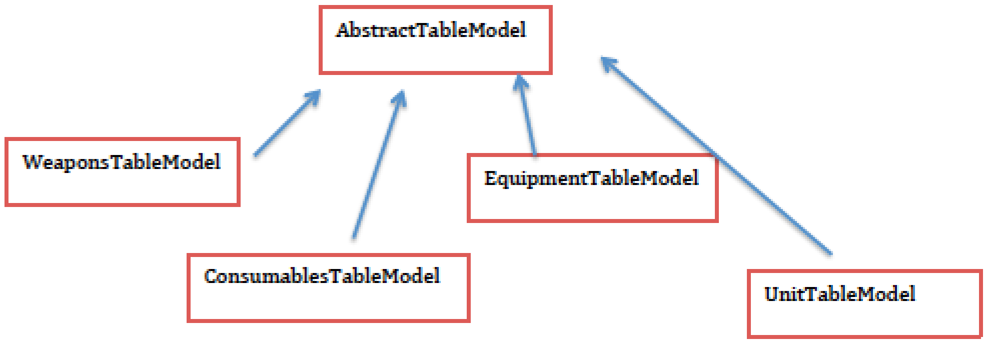
Our overarching design goal was to provide a framework that can support the gameplay seen in turn based games like Fire Emblem and Farming Simulator. The structure of these games contains a 2-dimensional grid for movement and a system of interactions between units found on the grid. We assume that the world the user wants to create will progress through various stages, so the user has to define each specific stage. We also assume that each stage has (a) required win condition(s), so we provide the user several predefined options that they can modify by some extent. An example of this would be a farming simulator, where the win condition could be for the main character to gather 3 gallons of milk from his cows. In another scenario, a fighting game such as Fire Emblem could require the player to kill a certain number of enemies, or a specific boss. Our goal is to allow the user to be able to create his own units and actions so that he can form his own unique take on the turn based genre.

For the view, the goal was to use modal dialogs to guide user interaction by focusing attention on a current list of units, items, or actions in relation to their use by units placed on a grid. Further, multiple states are managed by a tabbed pane system with a grid for each stage. This design ensures that individual problems are modular both in code design, but in visual design as well, for dialogs do not need to fit along with other panels in a monolithic GUI. Views for editing specific objects are only seen when they need to be.

Primary Classes and Methods

The following diagram illustrates the WorldManager interaction with creating table models to render views of objects to be edited. The diagram illustrates which classes are responsible for creating the table models from JSON files. The TableModels which extend AbstractTableModel are created by using their addNewXxx() methods with a specific number of objects per row required. TableModels are created when a game is loaded for editing maintained by the EditorData class for viewing when a menu item in the GUI is called to spawn an editing dialog that contains the appropriate JTable in a JDialog.


MakeDefaults class
Turns table model or table data into JSON
EditorData class
Reads in JSONS
To get table data
Holds all table models
setTableData()
WorldManager class
Methods for returning and getting a table model to GUI
save() -> new JSON files
load()
GUI class
Display menus to launch table views and
provides a place for grid to be rendered by running engine
call save/load on WorldManager



WorldManager - The intermediary between views and EditorData and Grid, stores List of Stages

- addStage/getStage/setActiveStage, set/getGameName, getObjectImage(x,y), set/getImage() etc.

EditorData - deals with the storage and handling of default/custom objects

- EditorData(), for use by the save JSON deserializer (information comes from save JSON)
- EditorData(String folderName), initializes and reads in information stored in JSONs found in "JSONs/folderName".
- loadObjects(), helper method that does the actual deserializing and reading in
- get(String type), returns the JTable data associated with the type requested.
- setCustomizable(type, JTable data), sets the JTable data associated with the type requested

MakeDefaults - standalone runnable class that creates default data JSONs. Methods include makeTiles(), makesObjects() and makeConditions().

Grid
- keeps track of all objects, tiles, and dimensions of the grid
- constructor - creates a new grid with the passed in dimensions with all default tiles
- move - changes the position of a selected unit
- action - finds the affected units of a chosen action
- findRange - gets the valid movement range of a given object, or action range of an action
- get and place objects and tiles - gets and places objects and tiles at given coordinates
- draw - draws the grid

Tile
- keeps track of a tile instance, which contains whether or not it is active, its stat modifiers, its move cost, and its image
- draw - draws the tile

Coordinate
- keeps track of the x and y location of a coordinate on the grid
- getters and setters for the x and y location of the coordinate

Stage
- get/setName, WinCondition, pre/post story, Grid

Condition

-stores data necessary to determine if it's fulfilled. Examples include PositionCondition (if a unit is on a certain tile then you fulfilled that condition), UnitCountCondition (if a certain side has a certain number of conditions, most commonly 0 to indicate all enemy units are gone), and TurnCondition (after a certain number of turns have passed, one side wins).

WinCondition

-List of conditions to be fulfilled to determine the win state.

Stage

-Contains the grid, win conditions, and all units. A game consists of several stages (levels), and controls how battle, movement, and actions work by putting together these parts and sequencing them correctly. Also contains the story to be displayed both before and after a level.

view

Calls WorldManager methods

action

CombatAction

- Stores stats and weights that affect the outcome of certain interactions (e.g. The attacker's magic stat compared to the defenders resistance stat will affect the degree to which outcomes occur).

- Actions are executed when a unit becomes engaged in combat. Outcomes of action are applied to attacker and defender.

gameObjects
- GameObject
  -
- GameUnit
  - Stores all of the data relevant to a character in the game
  - add, remove, and select objects for units to use

- unit can attack with a combat action associated with their active weapon

gameObjects.items

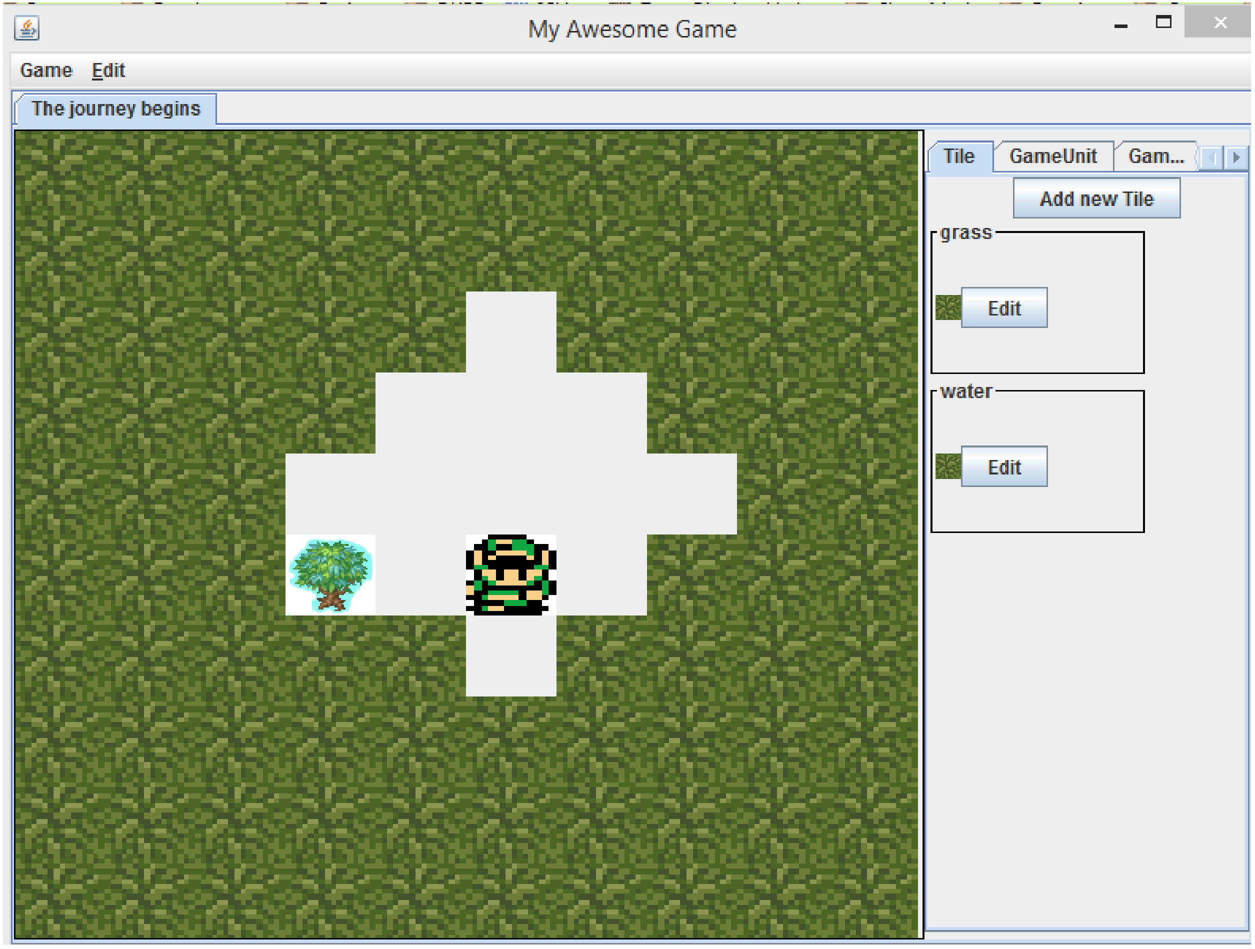
- Item
    - All an item requires is a name and quantity.
- Equipment
    - Stores stat modifiers that impact the character they are owned by/equipped to.
    - Can apply themselves to a units base stat as needed by combat.

parser

JSONParser

- createObject - makes object from JSON
- createJSON - makes JSON from object

GUI



__**Example Code and Games**__

Game save:

```
[ "controllers.WorldManager", {
  "myGameName" : "test",
  "myStages" : [ "java.util.ArrayList", [ [ "stage.Stage", {
    "myWinCondition" : [ "stage.WinCondition", {
      "conditionsNeeded" : 0,
      "conditions" : [ "java.util.ArrayList", [ ] ]
    } ],
    "name" : "stageOne",
    "affiliateList" : [ "java.util.ArrayList", [ ] ],
    "grid" : [ "grid.Grid", {
      "myWidth" : 10,
      "myHeight" : 10,
      "myTiles" :
       [ *4 THOUSAND LINES OF TILE CODE REMOVED* ],
    "myUnits" : [ "java.util.ArrayList", [ ] ],
    "myPassStatuses" : [ "java.util.HashMap", { } ],
    "width" : 10,
    "height" : 10,
    "gameUnits" : [ "java.util.ArrayList", [ ] ]
  } ]
} ]
```

Tile:

```
[ "grid.Tile", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "statMods" : [ "java.util.HashMap", { } ],
  "moveCost" : 2,
  "active" : false,
  "imagePath" : "resources/water.png",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ],
  "name" : "water"
} ]
```

Object:

```
 [ "gameObject.GameObject", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/tree.png",
  "name" : "tree",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ] ]
```

Unit:

```
[ "gameObject.GameUnit", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "affiliation" : 0,
  "experience" : 0.0,
  "controllable" : true,
  "itemList" : [ "java.util.ArrayList", [ ] ],
  "health" : 20.0,
  "activeWeapon" : null,
  "stats" : [ "gameObject.Stat", {
    "myStatList" : [ "java.util.HashMap", {
      "movement" : 3
    } ]
  } ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/hero.png",
  "name" : "hero",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ]
```

A Condition (for winning)

```
  [ "stage.PositionCondition", {
    "myData" : [ "java.util.HashMap", { } ],
    "neededData" : [ "java.util.ArrayList", [ "x", "y", "affiliation" ] ],
    "data" : [ "java.util.HashMap", { } ],
    "name" : "",
    "imagePath" : "resources/grass.png",
    "image" : [ "java.awt.image.BufferedImage", {
      "alphaPremultiplied" : false
    } ]
  } ]
```

Example game: Fire Emblem is a fantasy turn based tactics game that has combat and RPG elements such as items. This game has a constant cast of unique characters that you develop over certain stages. Characters grow stronger over time and retain their statistics and equipment across levels/stages. This takes place on a 2 dimensional square grid and once a character is gone it is gone forever. In this game, you can trade items between units. There are numerous ways to win a stage in Fire Emblem, including defeating all enemy units, surviving for a certain number of turns, and having a unit stand on a certain tile.

Fire Emblem Tile:

```
  [ "grid.Tile", {
    "myData" : [ "java.util.HashMap", { } ],
    "neededData" : [ "java.util.ArrayList", [ ] ],
    "passableList" : [ "java.util.ArrayList", [ ] ],
    "statMods" : [ "java.util.HashMap", { } ],
    "moveCost" : 4,
    "active" : false,
    "imagePath" : "resources/mountain.png",
    "image" : [ "java.awt.image.BufferedImage", {
```

```
    "alphaPremultiplied" : false
  } ],
  "name" : "mountaintile"
} ]
```

Fire Emblem Object:

```
 [ "gameObject.GameObject", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/tree.png",
  "name" : "tree",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ] ]
```

Fire Emblem Unit:

```
[ "gameObject.GameUnit", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "affiliation" : 0,
  "experience" : 0.0,
  "controllable" : true,
  "itemList" : [ "java.util.ArrayList", [ ] ],
  "health" : 20.0,
  "activeWeapon" : "Falchion",
  "stats" : [ "gameObject.Stat", {
    "myStatList" : [ "java.util.HashMap", {
      "strength" : 10,
      "health" : 14,
      "defense" : 10,
      "speed" : 20,
      "skill" : 15,
      "movement" : 6
    } ]
  } ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/chrom.png",
  "name" : "Chrom",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ]
```

Fire Emblem Condition (for winning)

```
[ "stage.TurnCondition", {
  "myData" : [ "java.util.HashMap", { } ],
   "neededData" : [ "java.util.ArrayList", ["affiliation", "turnCount" ]
],
  "data" : [ "java.util.HashMap", { } ],
  "name" : "",
  "imagePath" : "resources/grass.png",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ]
```

Example Game 2: Advance wars is a fictional turn based tactics game with combat but more RTS elements such as resource management and unit creation. Armies are formed with the production of units on a single map, and after a stage is over (through destroying all enemy units or capturing their headquarters), all produced units are not in play for the next stage. Thus, units are not retained across levels. This also takes place on a 2 dimensional square grid. Identical units can be produced if a unit of the same type is destroyed. Units do not grow stronger over time with the acquisition of items, so they also cannot trade. The main way to win in Advance Wars is by destroying all enemy units.

Advance Wars Tile:

```
[ "grid.Tile", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "statMods" : [ "java.util.HashMap", { } ],
  "moveCost" : 4,
  "active" : false,
  "imagePath" : "resources/cementhighway.png",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ],
  "name" : "Cement Highway"
} ]
```

Advance Wars Object:

```
 [ "gameObject.GameObject", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/steelandglassbuilding.png",
  "name" : "Steel and glass Building",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ] ]
```

Advance Wars Unit:

```
[ "gameObject.GameUnit", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "affiliation" : 0,
  "experience" : 0.0,
  "controllable" : true,
  "itemList" : [ "java.util.ArrayList", [ ] ],
  "health" : 10.0,
  "activeWeapon" : "Machine Gun",
  "stats" : [ "gameObject.Stat", {
    "myStatList" : [ "java.util.HashMap", {
      "strength" : 10,
      "health" : 14,
      "defense" : 10,
      "speed" : 0,
      "skill" : 0,
      "movement" : 10
    } ]
  } ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/recon.png",
  "name" : "Recon",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ]
```

Advance Wars Condition (for winning)

```
  [ "stage.UnitCountCondition", {
    "myData" : [ "java.util.HashMap", { } ],
     "neededData" : [ "java.util.ArrayList", ["affiliation", "unitList" ]
  ],
    "data" : [ "java.util.HashMap", { } ],
    "name" : "",
    "imagePath" : "resources/grass.png",
    "image" : [ "java.awt.image.BufferedImage", {
      "alphaPremultiplied" : false
    } ]
  } ]
```

Example Game 3: Omega Nu Farming Simulator (OMFS) is a ***FICTIONAL*** realistic fiction turn based strategy game. In this game, the main difference is that the violence in OMFS is limited between man and cow. There will be control of multiple farm animal units such as cows, chickens, goats, and pigs) and farmers. If a farmer unit and animal unit are next to each other, the user will be able to harvest certain items from the animals. Win conditions include having a certain number of items and getting a certain number items under a turn limit.

Farming Simulator Tile:

```
  [ "grid.Tile", {
    "myData" : [ "java.util.HashMap", { } ],
    "neededData" : [ "java.util.ArrayList", [ ] ],
    "passableList" : [ "java.util.ArrayList", [ ] ],
    "statMods" : [ "java.util.HashMap", { } ],
    "moveCost" : 0,
    "active" : false,
    "imagePath" : "resources/grass.png",
    "image" : [ "java.awt.image.BufferedImage", {
      "alphaPremultiplied" : false
    } ],
    "name" : “grass"
  } ]
```

Farming Simulator Object:

```
 [ "gameObject.GameObject", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/watertrough.png",
  "name" : “watertrough",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ] ]
```

Farming Simulator Unit:

```
[ "gameObject.GameUnit", {
  "myData" : [ "java.util.HashMap", { } ],
  "neededData" : [ "java.util.ArrayList", [ ] ],
  "affiliation" : 0,
  "experience" : 0.0,
  "controllable" : true,
  "itemList" : [ "java.util.ArrayList", [ ] ],
  "health" : 1.0,
  "activeWeapon" : “Udders”,
  "stats" : [ "gameObject.Stat", {
    "myStatList" : [ "java.util.HashMap", {
      “strength” : 0,
      “health” : 1,
      “defense” : 0,
      “speed” : 0,
      “skill” : 0,
      "movement" : 1
    } ]
  } ],
  "passableList" : [ "java.util.ArrayList", [ ] ],
  "imagePath" : "resources/cow.png",
  "name" : "Cow",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ]
```

Farming Simulator Condition (for winning)

```
[ "stage.TurnCondition", {
  "myData" : [ "java.util.HashMap", { } ],
   "neededData" : [ "java.util.ArrayList", ["affiliation", "turnCount" ]
],
  "data" : [ "java.util.HashMap", { } ],
  "name" : "",
  "imagePath" : "resources/grass.png",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ]

[ "stage.ItemCondition", {
  "myData" : [ "java.util.HashMap", { } ],
   "neededData" : [ "java.util.ArrayList", ["affiliation", "turnCount",
"unitList"] ],
  "data" : [ "java.util.HashMap", { } ],
  "name" : "",
  "imagePath" : "resources/grass.png",
  "image" : [ "java.awt.image.BufferedImage", {
    "alphaPremultiplied" : false
  } ]
} ]
```

The largest difference in genre implementation would be reflected in differing "data" passed to various types of Conditions that have been added to WinCondition.

Design alternatives:

Data management:

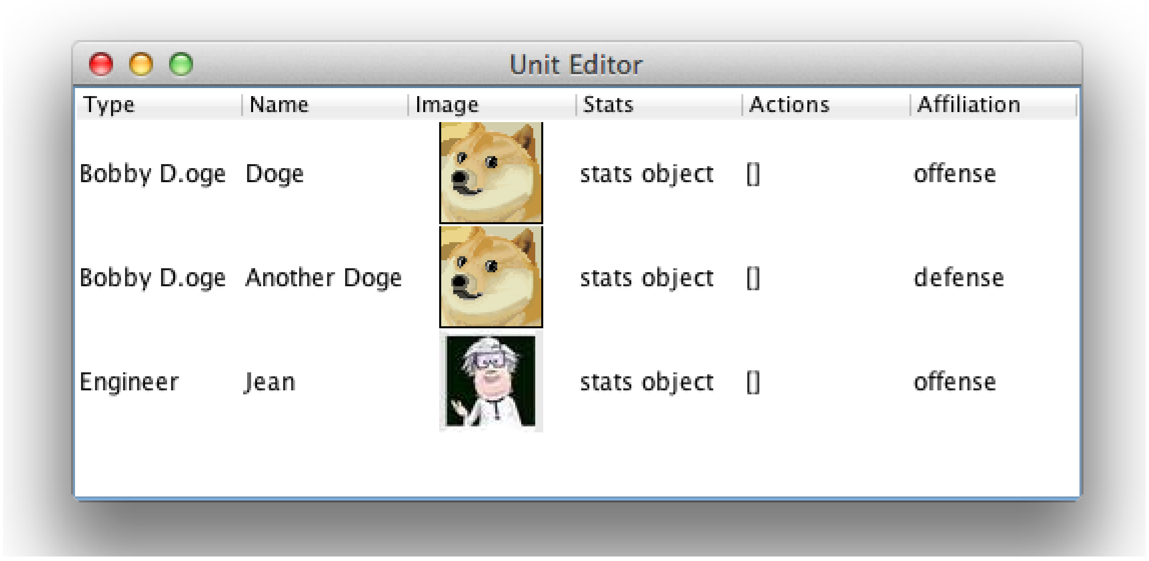
To store custom/default objects, units, actions etc. we considered several options. First, as far as format goes, we chose to store objects in JSON format early on because between the two most viable options, XML and JSON, we found that JSON was more readable and had a large enough following for there to be several comprehensive JSON libraries for Java. Upon playing around with the two formats, we ultimately did choose JSON for it's ease of use and flexibility. Once we got past that we still had to deal with what exactly we were serializing to JSON.

Default tiles (and custom tiles eventually) were originally stored in a list of specific objects and then serialized/deserialized. It worked out fairly well until we got to the actual sending and reading of information from the GUI, when it seemed that we would need many customized reading and packaging/unpackaging classes to deal with that.

In order to view and edit the various objects and edit units, items used by units, and actions that items can execute, JTables were used. JTables are backed by models which extend AbstractTableModel. These models reflect the natural organization of all objects used in gameplay into rows which can be added, deleted, and edited in individual cells. Cells can represent an ImageIcon, String, or object used in gameplay such as a Stats class or an Item class. JTables require custom editors and renderers which provide a means for unique views for editing cells to be called up as JDialogs. The models backing all JTables store data as a `List<Object[]>` in order to allow insertion and deletion of rows, as well as a generic array to hold images, and text. The non-generic parts of each type of object is reflected in the number and type of columns in the respective JTable model class. A particular table model has a List of Object arrays of a length that varies with the particular view that table supports. For example, a unit view holds the following information:

Type (drop down selection)	Name	*GRAPHIC*	Stats Object (an object with a spefic renderer)	Items (a list of object with a specific render)	Affiliation (drop down selection)

*more rows for each unit definition in the game



Type	Name	Image	Stats	Actions	Affiliation
Bobby D.oge	Doge		stats object	[]	offense
Bobby D.oge	Another Doge		stats object	[]	defense
Engineer	Jean		stats object	[]	offense

Screenshot from our Unit Editor which shows a JTable driven by a UnitTableModel

with each cell rendering being defined by a CellRenderer class that extends AbstractCellRenderer or implements TableCellRenderer. The models driving these tables can then be retrieved and packaged only when needed by an appropriate controller class (and saved in JSON format).