Team: MISSINGNO

OOGASALAD - Initial Design Document

I. GENRE

Our platform will support the creation of turn-based strategy games. Such games involve players placing units on a map and strategically controlling their behavior. A victory condition such as controlling a given number of points or eliminating enemy units determines when the game is completed. Features of this genre that our API will support include:

- Definition of victory conditions. The designer will be able to create and apply a variety of conditions that define how the game is completed and who wins.
- Creation of a variety of mobile units. Each unit will have specific behaviors and attributes.
- Creation of custom map tiles and map layout design. The designer can define the exact way in which a map is organized to support game customization. Further, different tiles may interact with units differently.

II. DESIGN GOALS

Our current expected high level flow of information:
The model acts as a class library for the GAE, Data, and Engine. All data components of the model extend a Stat framework, so that both GAE and Data can use the same function calls (getStat() and getStatCollection()) to retrieve all of the pertinent state for any class in the model. The GAE uses the getStat calls to populate interfaces so that a developer could create and modify units, abilities, conditions, and tiles. This information is passed to the Data team as the objects themselves. The Data team converts the objects to xml and back to the objects that get passed to the Game Engine. The Game Engine draws the game state on a JGame Canvas. There will be limited user input between the Game Engine and model facilitated by a controller class.

- Complex conditions based on a modular set of fixed conditions (gather x, defeat x, create x, move x to y). Conditions can define:
    - Win conditions
    - Turn conditions
    - Unit ability usage
- Units modular through Composition
    - Customizable unit stats (ie different kinds of units)
    - Customizable unit abilities (ie fireball spell, rallying cry, or spawn minion)
        - use modular “effects” that include modifying stats and creating other units
        - based on customizable parameters like radius, range, and power
        - all units can attack, move, and harvest by default, though setting the associated unit stat values to zero effectively removes them, allowing for unit ‘buildings’
- Customizable map and terrain (defines shape of habitable map and unit traversal)
- Allow multiple players to play on the same machine

- Potential to utilize one or more AI players
    - rank set of conditions to drive behavior

Assumptions

- All tiles are square
- Maps are rectangular (though playable areas may be any shape… e.g. by using impassable borders)
- Units have the following attributes: Max Health, Health, Base Attack, Attack, Base Defense, Defense, Base Stamina, Stamina, Range.
- Units move by spending stamina to traverse terrain.
- The following behaviors are available for a unit: Attack, Move, Harvest, Interact, InteractedWith, CustomAbility.
- Custom Abilities either modify stats or create other units.

Structural Components

The key structural components of a turn-based game are: the map layout, the functions of various units and how they are created, and the win conditions for that game.

Game Variety

Our platform will allow for great variety in games designed mainly through these game elements:

Map layout - the configuration of different tiles (impassable, farmable, etc.) creates different points of contest for players. Perhaps resources are central on the map, prompting early aggression, or they are on the fringes, encouraging a late-game.

Game objectives - designers will have free reign over what dictates the end of the game: king of the hill, kill all enemies, survival, etc.

Unit abilities - allowing the designer to set the attributes (health, base attack, etc.) and abilities of all units and the steps to create them. Different unit configurations make for drastically different gameplay.

Opponents - play with other humans on the same machine (switch turns) or verse an AI Computer.

III. PRIMARY CLASSES AND METHODS

Model

Units are modular through composition. When a fresh unit is created, all composed elements are set to default values, which are editable in the GAE. An attributes object stores numeric stats. Different unit behaviors are introduced by changing the unit's Abilities object, which contains a collection of abilities, including: Attack, Move, Harvest, Interact, InteractedWith, and CustomAbility. CustomAbilities may alter unit stats or create new units (as discussed above in Assumptions).

State within the model is modified through the use of Abilities. Abilities can be abstracted to two method calls: prepAbility() and useAbility(). PrepAbility() prompts the user for any necessary information (sometimes none is needed), and the useAbility() carries out the defined behavior based on the user input, or lack thereof. There are three predefined ability behaviors--namely move, attack, and harvest. Move() updates position state based on user input. Attack() updates unit attribute state, based on a different interpretation of user input. Harvest() interacts with resource state of a particular tile. All other state modification is contained within the flexible CustomAbility class, which functions by prompting the user for a number of targeting parameters and then applies a number of effects to those targets. The Effect classes modify unit attributes or create other units. These effects are modular.

All objects that are stored by data extend either Stat or StatCollection.

The Stat object stores a name and a double. This allows information such as a Unit's health, for example, to be easily displayed on the front end by calling the stat's name and value, for example: "Health" = 10.0. Further, by storing this data as objects, the GAE may alter the attributes of a unit during creation simply by adjusting the stat, not requiring a call back to the model.

StatCollection extends Stat, such that any StatCollection, such as a Unit, can have a getStats() method called, which can be looped through to receive all information that comprises that unit. Under this setup, overall data storage may be viewed as a tree, where a given StatCollection branches into other StatCollections (further branching) or Stats (leaves). For example, a Unit contains an Attributes object, with further contains Stats that specify health, attack, etc.

[Please refer to any of the following classes in our online repository, which we have Javadoc'd: model.stats.Stat, model.stats.StatCollection, model.unit.Unit, model.Attributes, model.Abilities, model.abilities.Attack]

Game Authoring Environment

Controller - the controller holds references to each of the panels and enables communication between panels and other swing components. addPanel() method allows the Workspace to add the EditPanels to the controller. Methods that edit panel information are:
postBoardData(int x, int y) - sets the board size to the input value
postPlayers(int numPlayers) - sends the intended number of players to the PlayerPanel, which adds or removes players to match the input value.
postProperties(List<Stat> props) - sends the Stat properties to the EditPanels, to be posted in the ObjectsPanel

removeTask(TaskViewItem tvi) - sends the given task to the EditPanels to remove like types from the TaskPanel.
createMap(List<String> data) - creates the map, an instance of JGEngine, in the MapPanel and initializes it.
Additionally the current model state can extracted from the panels be saved to an XML file using the **getAndSaveState()** method.

EditPanel - extends JPanel, holds and updates a BoardList based on calls from the controller. Holds a reference to the Controller to allow communication from one EditPanel to another. Holds default actions (do nothing) for the method calls necessary for the controller. Each of the following is overridden in the subclass that implements the intended effects.
postProperties(List<Stat> properties) -posts the properties of a clicked object in the ObjectPanel
postPlayers(int numPlayers) -modifies the number of players to match the input.
createMap(List<String> data) -creates the GUIMap in the MapPanel
removeTask(TaskViewItem tvi) -removes tasks with the same subtype as the given example from the TaskPane
addViewItem() -adds a ViewItem to the given panel.
giveStateObjects() -places model side objects needed to populate a game to the given GameElements object and returns it.

https://www.lucidchart.com/publicSegments/view/528ab95d-84fc-4ade-8d7a-27170a004b21/image.png

MapPanel- child of EditPanel. Component responsible for displaying and maintaining a view of the game's map. Holds references to instance of **GUIMap**, the frontend JGEngine representation of the Map state, as well as **GameMap**, the backend representation of the Map state.

BoardList- extends JList, updates a list of ViewItems based on calls from EditPanel. Defines a custom subtype of **DefaultListCellRenderer** that defines the precise way in which ViewItems are displayed to the user. Methods include:
addNewItem(item:ViewItem) : adds a new ViewItem to the list model.
removeItem(item:ViewItem): removes from list model
sendData(data: List<Stat>): sends a list of the object's stats to the Controller for the purpose of updating the GUI appropriately when selected. Called when ViewItem clicked.
createCustomType(): launches a new InputDialogue to collect the user-inputted data.
postInput(input:List<Stat>): sends data to the selected ViewItem to update fields with user input. Called by actionlistener in InputDialogue.

Child classes: **ConditionList, PlayerList, TaskList, UnitList, TileList**

ViewItem -Abstract Object that represents the item to be placed in a BoardList, controls what should be done if clicked. Stores Model representation and JGObject representation of the same object. Child classes will implement onClick to specify behavior when clicked. Contains all information necessary to properly display in BoardList.

BoardListViewItem - Abstract type representing backend objects that will be directly applied to the Board. Holds reference to **myProperties**, a list of Stat objects for the subtype's particular backend class. This list is used by the InputDialogue to display the correct input fields as the designer creates new custom types. Method **createModel()** takes a List of Stats as an argument and updates the view item's backend object with the user's custom data. Child classes: **TileViewItem**, **ConditionViewItem**, **UnitViewItem**. Each holds reference to an instance of backend object.

TaskViewItem - abstract type representing a distinct task that the user must complete in order to save the game. Contains a method **onClick()** that will launch an InputDialogue for the user to input information necessary to complete the task. For instance, the user must specify map dimensions before the map is created. Child classes:
BoardSizeTaskViewItem, PlayerTaskViewItem

MapObject- custom subtype of JGObject to be placed on the map. Paints itself on the board, drags and drops itself on mouse drag, references stats on mouse click. Contains a reference to the particular **BoardListViewItem** that created it.

InputDialogue- Pulls necessary fields and populates text fields based on the backend object it creates, sets model instance variables based on user input to create the specific object.
Methods include:
createGutsPanel(): returns a JPanel containing all the appropriate fields for the user to enter input.
postInput(): called when input submitted. Sends data back to the BoardList source.
GetDataAction: custom ActionListener calls postInput when user is ready to submit data.
Child classes: **PlayerDialogue**, **BoardSizeDialogue**, **SubUnitDialogue, UnitCreationDialogue**

<u>Data</u>

GameElements

GameElements class is the collection of data that contains all the objects of game elements. When “export to xml” is invoked in the game authoring environment, this object will be filled with objects created by the game designer and be passed to the encoder to serialize objects into xml file. On the other hand, when the game player loads the xml file to start the game, ‘Data Manager’ will read in the file and make decoders to create corresponding elements. It will finally pack all the objects into GameElement object and send it to the game player.
getter and setter methods

SaveHandler
This class will receive a GameElements object, which contains all of the information that needs to be stored, through the **initializeSave()** method. This method is invoked by the game authoring environment when it is ready to save the relevant information and pass it to the data side. SaveHandler also has the method **initEncoders()**, which initializes all of the Encoders needed to process the information. Afterwards, each Encoder will be called through Encoder’s abstract encodeData() method.

Encoder
This is the abstract class which contains the method **encodeData()**, which will serialize the information stored in a GameElements object and format them into an XML file which will be saved for later use. It contains methods specific to converting the information into an XML file such as **formatXML()**, which adds appropriate indentations to the file, and **addXmlElement()**, which adds an element to the XML file. These methods are extended and implemented by subclasses that extend the Encoder class, **MapEncoder, UnitEncoder,** and **PlayerEncoder**.

DataManager
The data manager receives the xml file of defined game elements. It converts the XML file into the dom document object, pass it to the decoders by calling **processDecoders()**. It also collects the objects back from the decoders, finalizes the deserialization, and sends the game element object to the game player.

getGameElements(): this method receives the xml file and returns the GameElements object that is filled with the objects defined in the xml file.

Decoder
Abstract Decoder class has an abstract method called **decodeData()**, which receives the ‘dom’ document object from the data manager, instantiates the right elements and load them back to the Data Manager.

Concrete Decoder class: UnitDecoder, MapDecoder, PlayerDecoder.

* The Data manager and Decoder assume that the XML file is in the right format that we agreed. Otherwise, it will crash or throw the exception. If the user wants to add the new element that is not defined in the model, the decoder is still easily extensible since he or she only needs to implement new concrete decoder class.

Game Engine

GameViewer
The GameViewer will be the parent container for all the components of the game's engine and user interface. It defines the layout of these components, and is comprised of a Menu Bar for loading games, the GameEngine. The GameEngine will be placed in the center spanning from the left edge to the right, and will have a text at the top that will show the score for each player. Beneath the Game Engine we have a button to end the current player's turn and two dynamically changing list menus for choosing units and their respective actions. Followed, in a FlowLayout with the button and lists, we also have a JPanel that contains two text areas: one for reporting selected unit stats such as health, attack, etc., and another for player stats such as resources.

DataPrimer
The DataPrimer class is responsible for loading the File Chooser. The **selectAndParseXML()** method which selects an XML file and calls the parser created by the Data team to parse it. The resulting data structure is used to populate the map with its initial tiles and units.

GameLoader
This class manages the creation of game objects from their respective model objects (eg. Tile ----> GameTileObject) in the GameEngine, and the distribution of units amongst the players contained by the GameEngine. This is accomplished by the use of the **setPlayers(), initializeUnits() and initializeTiles()** methods which the GameEngine class implements. It calls these methods after obtaining Lists of model objects from the GameElements object returned by the DataManager class (Game Data) when the DataPrimer is activated.

GameEngine
The single instance of the GameEngine takes care of the actual game. As an instance of JEngine, it will be placed in the top-center of the GUI via the GameViewer class and will display our map (tiles) and units. Most basically, the class starts by initializing the canvas to a specific size, and then populates the map with tiles and units.
To do so, the GameEngine has methods **initializeTiles()** and **initializeUnits()** to create game tile and unit objects to place on the map when the user calls a file created from the Game Authorizing Environment. Both methods iterate through unit data structured by the Data team after having parsed an XML file provided by the GAE.

After having initialized the playing field, classes such as **MouseObject**, **MouseListener** will take care of user interaction with the engine. These two classes in particular respond to the user clicking on certain units to perform certain actions such as move, attack, spawn, etc. The primary method will be to use collision between the units and the mouse when clicked. Actions such as move, attack, and spawn will have a second interaction after the initial click to determine where the unit will move, attack, or spawn another unit. After having selected a specific action, such as move, the Model will return a collection of highlighted Tiles that the unit can move to. Afterwards, we intend to implement a method such as **getHighlightedTiles()** so that they appear on the GameEngine. Clicking on one of the highlighted tiles will then be listened to again, and the method **getSelectedTile()** will then return a specific tile that the unit will interact with (move, attack, etc.).
GameEngine will also implement methods that the model will use to update the GameEngine - i.e. remove unit objects that have been deceased and initialize new unit objects that have been created. **createUnits()** and **removeUnits()** will fulfill these functions. **updatePlayerScore()** will be used by model to alter different players' scores depending on the outcome of player actions.

IV. EXAMPLE CODE

Our final platform will be able to create analogous versions of the following turn-based games:

· Civilization V – this is likely the most in-depth game that will be catered to. A game such as this will have further modification with different game types, such as king of the hill or elimination. The most intensive part of designing this from our platform would be creating tech trees (or the in-game hierarchy of units that may be spawned).

· Chess – the most significant feature of Chess is (obviously) movement of the pieces – whereas most Turn-based games will have units' movement limited by distance and the pass-ability of a given tile, Chess pieces have set, allowed movement schemes.

· Farming Simulator – this is an example of a turn-based game in which there is no opponent. Rather, in this case, the computer AI might randomly introduce conditions impacting the user's play, such as introducing a swarm of locusts. Further, multiple players *could* be incorporated in which the players may cooperate or compete.

V. DESIGN

The current design separates the entire project into the most compartmentalized components: GAE information flows in one direction to Data, which flows in one direction to the Engine. All three of the GAE, Data, and Engine use class definitions from the model that have been designed to be as read-friendly as possible given the various usage requirements of each component. Please see the respective class and method section for a justification of that component's low-level design decisions.

VI. TEAM ROLES

Game Authoring Environment (GAE)

· Bradley Sykes - set up edit game perspective, map tiles, victory conditions, toolbar functionality, interaction with game data team.

· William Shelburne - set up GUI flow from title screen to edit game perspective, objects, players, interaction with game engine team.

Model

· Timo Santala and John Godbey - have pair-programmed / worked together on all portions of the model.

Game Data

· Alex Song - design and implement the encoder that translates information from the game authoring environment to XML files.

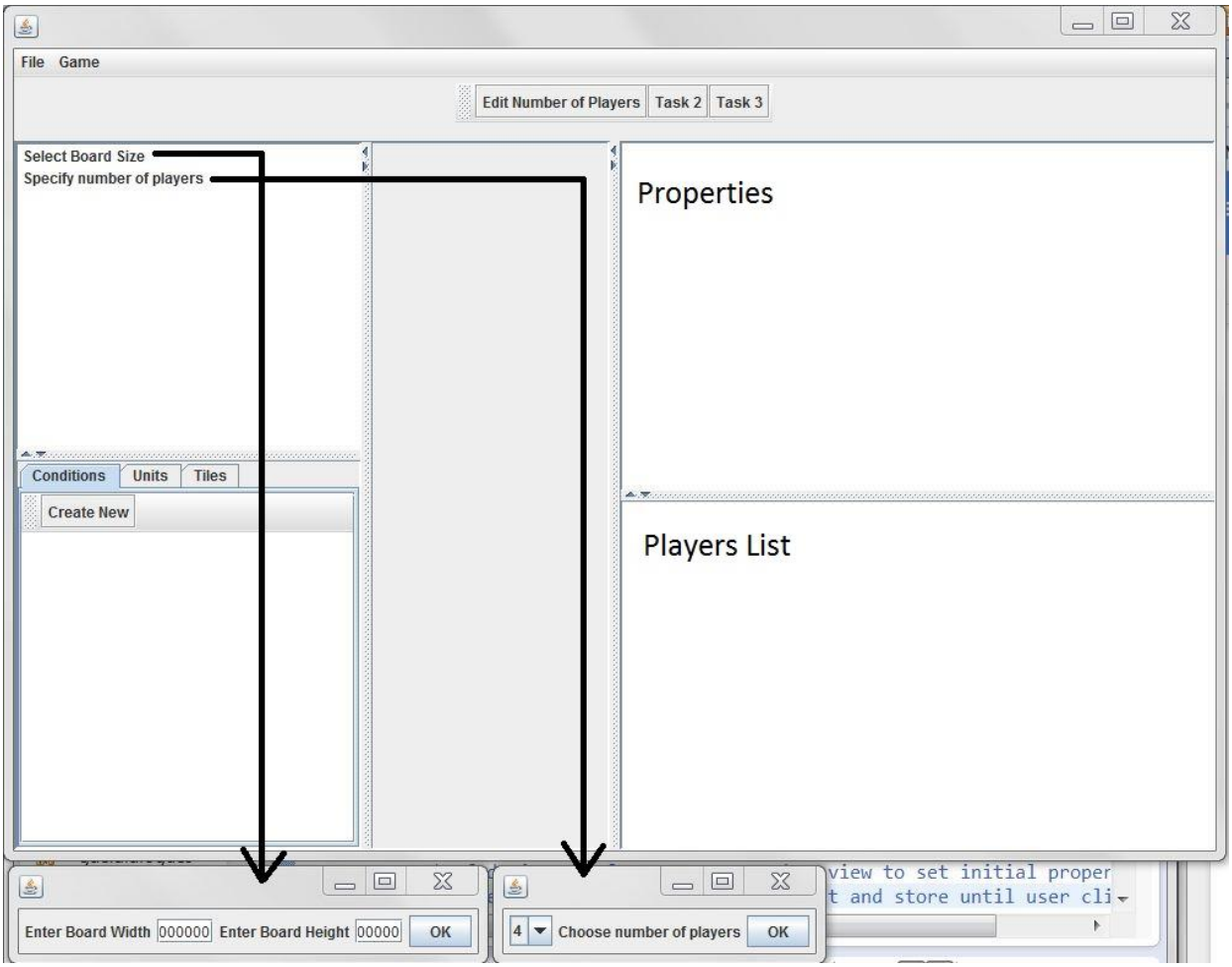
· Seunghyun Lee - design and implement the decoder that converts the XML to information that the game player can access.

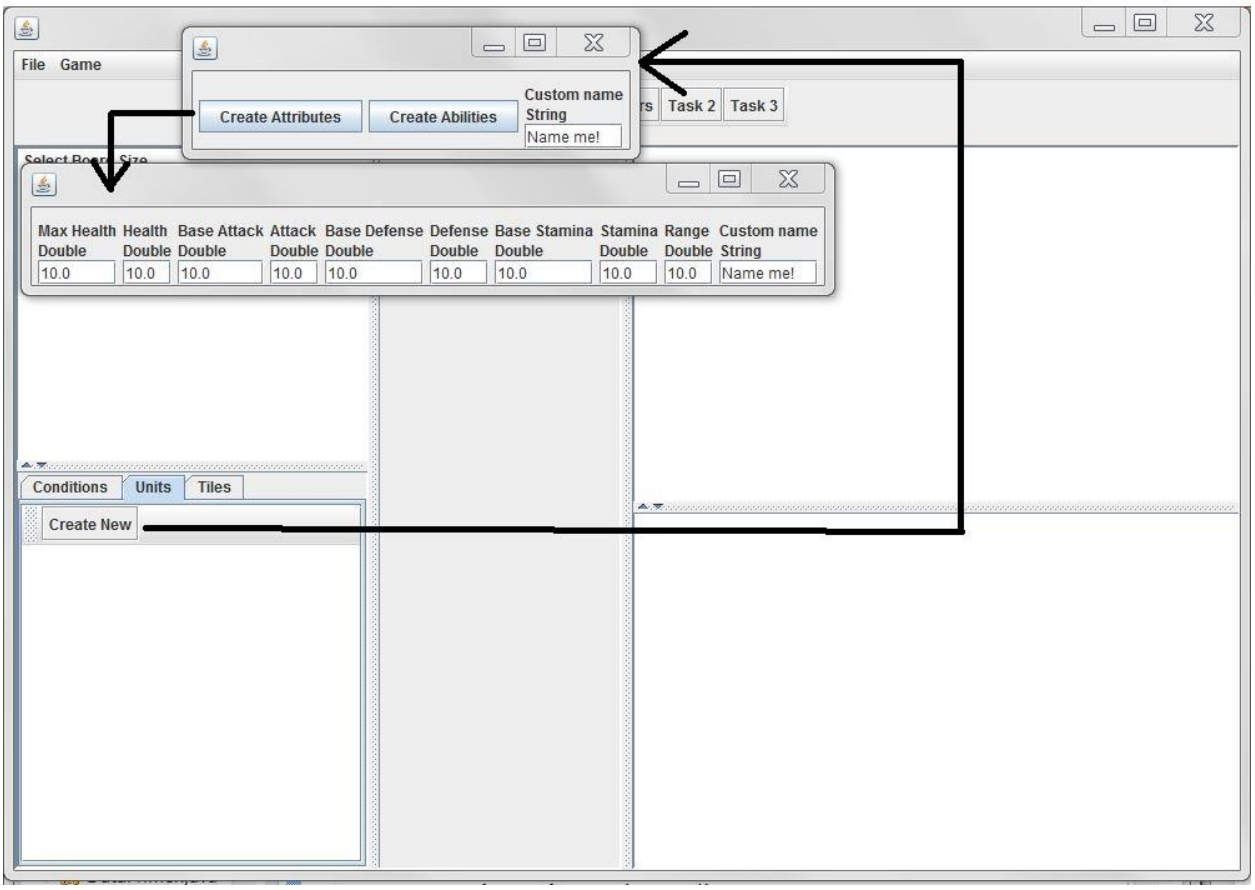
Game Engine

· (Both members were not able to attend our Design Document meeting: specific roles will be determined at a later date.)

VII. ScreenShots

GAE




File Game
Create Attributes
Create Abilities
Custom name
String
Name me!
Task 2
Task 3
Max Health
Double
10.0
Health
Double
10.0
Base Attack
Double
10.0
Attack
Double
10.0
Base Defense
Double
10.0
Defense
Double
10.0
Base Stamina
Double
10.0
Stamina
Double
10.0
Range
Double
10.0
Custom name
String
Name me!
Conditions
Units
Tiles
Create New