Team Awesome

OOGASalad Final Design Plan

Genre

We are setting out to build a turn based strategy game engine. The inspiration for this came from turn-based "war-like" games in which one player controls an army of soldiers against a computer's army. Some examples of such games (with gameplay videos for clarity) are as follows:

Fire Emblem (probably already better than what our final product will be…): https://www.youtube.com/watch?v=SAQpa4xYB_o

Fire Emblem: Radiant Dawn (much prettier version of the same idea): https://www.youtube.com/watch?v=UzGcCBnWicY&list=PLF007A82BEC04F667&t=1110

X-Com: Enemy Unknown (even cooler!): https://www.youtube.com/watch?v=YY8Rkyps_PU&t=380

The user will not be limited by our authoring environment to making games of this exact form. Using our engine, users will be able to build games in which a primary player moves units / characters across a grid-like map on his / her turn and all other "players" (computer AI + possibly other human players) are allowed to move their units / characters before the player can move again. This would ideally allow a creative mind to create variants of board games, puzzle games, and more in addition to the "obvious" route of making a turn-based strategy game.

Modes

- Authoring Mode
- Gameplay

**User should be able to toggle between game mode and design mode at will

Additional Design Goals

- Implement an AI Engine, since our game is turn-based. Also allow for AI stats to be under the control of the user.
- Toggling between game mode and design mode should be as little of a hassle for the user as possible; we would like to keep the view of the game grid in the same window, whether the user is creating or playing a game. Options in the side bar will change, according to which mode the environment is in.

Design Specifications

Game Engine (Teddy, Grace, Andy)

package: engine

ViewController is an interface that was created with the intention to follow the model/view/controller paradigm so that we could fully separate the back and front end.

Controller implements ViewController and interacts with StateManager to add the code into the gui representations of the objects. Also interacts with the LevelManager to make the information about about the level appear the way it is. Code allows images to appear and disappear from the level, and the objectives to be edited.

- private void initStates() sends all of the states to the StateManager, including enemy mode, and level complete.
  Can also let you play the game

The controller evolved throughout the project to have a great deal of public API methods. An alternative would be to couple the front end and back end more closely. This would save us from having this behemoth of a controller class, but would ultimately result in less clean, reusable code elsewhere.

Game/LevelManager Lets the user make levels and holds them in a map with a String name as the key. Important objects include the active level (activeLevel), a list of the levels made (levels), and a map of games with game names (games).

- newLevel()
- removeLevel()
- activateLevel()
- clearAll() Allows the user to erase all of the information that currently defines the levels.

Level contains a map (myMap) of the current level that associates locations (JGPoints) with GameObjects. Also handles creating objects, placing an object in a specific position deleting an object, getting an objects range, and more. It also keeps track of all the enemies and players that exist in the level, which is useful for figuring out when either side's turn has ended. The level keeps track (Stack placementHistory & Stack undoHistory) of the last several actions made by the author so they can undo/redo actions. This might not be the best place to keep this information, as in an object oriented model this information should probably be held by the authoring environment itself.

- addObjective()
  creates an objective before putting it into the list of objectives.
- getCompleteObjectives()
  returns a list of completed objectives.
- placeObject(GameObject toPlace)
  Places a new object on the map.
- moveObject(GameObject toMove, x, y, row, col)
  Moves an object to a given (x,y) pixel coordinate / (row,col) grid coordinate.
- Set<JGPoint> getRange(Unit unit, boolean isAttacking)
  Gets either all of the JGPoints that a unit can move to or all of the JGPoints that are within its attack range. Uses the recursive, creatively named helper method rangeRecurse.
- hideAll() / showAll()
  Hides / shows the contents of this level.
- undoPlacement() / redoPlacement()
  Undos / redos the last actions done in the authoring environment.

ResourceLoader can load some desired information from a properties file. This is normally used when we're doing something with reflection. For example, we use the resource loader to get all the different images that can be used for enemies, or the classpaths of the different algorithms for AIUnits. We could have used the Reflection util package here, but we prefer the tag parameter we use in load() to having a ton of different properties files, and Teddy wrote this (shoutout to Jordan Ly for the idea) before that util package was posted, anyway.

- static Map<String, String> load(String tag, String fileName) Loads all of the information from the given properties file that starts with the given tag. ie: if I wanted to get a list of all the images that can be used for enemy units, I would call load(“Enemy”, “resources.images”);
  This would scan the images properties file for any entries that start with “Enemy” (ie: “EnemyGargoyle=medieval_enemy_64.png” but not “PlayerStandard”) and give us a map linking the name (“Gargoyle”) to the image path (“medieval_enemy.png”).

DeepCloner makes a deep copy of an object so that we can use public get methods effectively without worrying about others manipulating data. This is not our original code.

Objectives

package: objectives

An **Objective** is a task that the user should accomplish to complete the level. Examples include: leave no enemies alive (LeaveNEnemies), kill the boss (KillBoss), reach a designated square (CapturePoint), or survive 20 turns (Survive). All specific objectives extend Objective. The constructor for Objective takes the Level that it is an objective for and a list of relevant arguments. For example, in level 5 one might need to collect gems until there are no more than 2 remaining on the map -- this would be done using the LeaveNObjects objective. The objectives were initially in an inheritance hierarchy, with all objectives extending the Objectives.java superclass. We considered having a subclass of objectives that would serve as a superclass for objectives that required traversing the entire game grid to determine whether the objective was fulfilled or not. If we had a larger collection of objectives, this might have been useful for organizational purposes; however, at our current stage, we didn’t feel that doing so was necessary.

The Objective superclass holds the following methods:

- getNumberFlaggedObjects(String indicator): a method for traversing through the game grid and counting the instances of a certain collision- or team ID. Its parameter indicates whether team IDs or collision IDs should be searched for.
- isAchieved(): returns whether or not the objective has been achieved, based on the state of the game grid.
- isFailed(): triggers game over when the objective cannot be achieved. We wrote Objective so that this only occurs when the user has reached the maximum turn count without completing the objective. An alternative would be to make this method abstract and have each individual objective have failure conditions (ie: if enemies pick up 12 gems, the game ends).

Objectives are created using the **ObjectiveFactory**, which is based on the Factory design pattern. It takes the name of the objective that needs to be created and a list of arguments that are relevant for completing that objective.

Example:
To add an objective to the current level in which a user has to have any of their characters step on the square of the grid at the coordinates (1,1), one might write something like:

List<Integer> args = new List<Integer>(1,1);
currentLevel.addObjective("CAPTURE", args); // which will then call the objectiveFactory's create() method

The factory pattern here allows us to create instances of objects without using "if" trees and potentially importing the entire objectives package into a class.

Objects

package: objects

GameObject is our superclass that every object in the game will extend. We used JGame, so GameObject extends JGObject. An alternative would be to abandon JGame and create our own custom object interface. JGame is not particularly well designed, and their GameObjects are no exceptions: by controlling their own sprite they muddle the line between model and view components. They also have public variables for the x and y coordinates, and if there's one thing that this course has drilled into our heads, it's that public variables are only used by losers who belong in the 7th circle of programmer hell. However, we wanted the functionality of the JGEngine to make our lives easier, and we find the API for animating GameObjects exceedingly useful. If we had another month for the project, I would have been all for writing our own version of JGEngine, but JGame ended up being a necessary evil.

We initially planned to have two superclasses, one for objects and another for player and enemy characters (called "units" in our API). Instead, all of the following classes extend GameObject:

Unit defines a character in the game. This character can be controlled by either the computer or the player. Units have several actions: they can move, attack things, push things, or just wait. As they pick up items, they might get more abilities (i.e. an axe might let one cut down trees). These actions are all defined in the gameplayer package's action package (see below), and future programmers could certainly add new, custom actions. The Unit has stats that determine how far it can move in a turn, how far away it can attack, how powerful it is, etc., and these can all increase when the Unit accomplishes enough things (ie: kills enough enemies) to level up. Units can be controlled by AI, as discussed in the ArtificialIntelligence section below.

- Constructor: Unit(final int collisionID, final String name, final String graphicsName, final int x, final int y, final int tileSize, final int team, final StatSheet stats, final StatSheet growths, final String type, final String origPath)
  Sets up the Unit and defines its initial actions.
- void attack(Unit other)
  This allows this unit to attack another unit.
- void damage(int amount)
  This damages this unit by a given amount.
- Set<String> getDestroyableIDs()
  This lets someone know all of the different items that this Unit can destroy.

We have two main object categories, both of which extend GameObject:

TerrainObject defines objects that are parts of the terrain, such as trees, rocks, etc. Can be movable or immovable, but are not collectible.

- Constructor: the boolean value for whether the object can be collected is set to false. This is the essence of a TerrainObject.
  **a TerrainObject can still be movable (i.e. our rocks can be pushed).

PickupObject defines objects that can be collected, and can affect the game in some way (improve the holder's health, increase the user's score, etc.)

- Constructor: the boolean value for whether the object is collectible is set to true.
- setStats(int maxHealth, int bonusHealth, int power, int defense, int speed, int range):
  creates a new StatSheet object for the PickupObject, using the parameters passed in.
- pickUpItem(Unit unit, int maxLife, int bonusLife, int power, int defense, int speed, int range):
  calls boostStats(StatSheet boosts) on the unit, which will take the StatSheet of the PickupObject and change the unit's stats accordingly. This method then removes the PickupObject.

Currently our code is structured around a few key game object types--player, enemy, terrain, item (stock items, specifically), and weapon; this is evident in the cluster of type tags in our Constants class (display package). This is a possible weakness in our design because in order to add objects without significantly adding to the code library, users would have to fit all objects that they want to add in these categories. While these categories cover a wide variety of objects, we can't assume that all objects to be desired in a game would fit in these categories.

Additionally, to add a new object and implement it fully within the game authoring environment, the user must add lines of code to the image properties file in the resource package and a few lines in the Constants class, at the very least. (A new image in the images folder of the resources package would also be nice.) To create a new action for a new object, users would have to add a line to the common properties file in the resources package, as well as a class in the actions package of the gameplayer package that extends UnitAction. Adding a sound effect would require updating the sound properties file, the sounds package, and the Constants class.

Thus, adding a new object to the game authoring library requires code that could range from a few lines to a class, and then some. This lack of standardization in adding new objects is another weakness in our code, because the process is very diffuse (sometimes not immediately obvious), and is also dependent on the object itself. Adding a new class to the actions package would probably be unavoidable at this point if a user wanted to add a new action, but we feel that some of the other steps in adding a new object could be refactored out.

StatSheet, which does not extend GameObject but provides an added dimension of complexity to our objects, maps stats (currently: current health, max health, power, defense, speed, range) to integer values. This can be used either to store the stats of a unit and the percent chance each stat has to grow (in the growths field). This conveniently allowed us to group all stats together, so that we could minimize the number of instances where we would have to deal with numerous strings and boolean values.

<u>**Artificial Intelligence**</u> (Teddy, Matt)

We created a fairly complex AI engine for the enemy units in our game using a couple of different design patterns. Every **AIUnit** has a **Strategy**. This strategy is defined by the degree to which the AI pursues certain actions (in particular: attacking the user, protecting itself, going for pickup items, and defending the objective) and an overall intelligence score. The AI inputs these values into its set of intelligence algorithms (discussed below) to determine which square to move to. Evan learned the idea of a reward structure in his AI class, and we're just applying that here.

There are probably more sophisticated ways of implementing AI, but we like this because it provides ways for game authors with a varying skillsets to create custom AI's of different complexities. A game author can create their own strategy by entering custom values for the AIUnit's intelligence and its different reward functions. They might make a super greedy AIUnit who takes all the pickup items, or an aggressive AIUnit who will attack the user with no regard for his own safety. Game authors with heavy programming experience can easily add sophisticated algorithms that the AIUnits can use to determine how exactly they will go about achieving their goals.

The intelligence of the enemy is particularly important as it determines which set of algorithms the Unit uses to decide which grid square to move to. These algorithm sets extend the **AIAlgorithm** abstract class. An enemy with low intelligence might get assigned the DumbAIAlgorithm (does nothing), whereas an enemy with high intelligence might get assigned the SmartAIAlgorithm (which is pretty complex). The AIUnit has a field myAlgorithm which stores the AIAlgorithm that it will use. This is an application of the Strategy design pattern. The AIAlgorithm has several protected utility methods that may be called on by any algorithm, and the following abstract methods:

- **evaluateAttack(), evaluateDefense(), evaluateMoney(), and evaluateObjective()**: send some information about the AIUnit (and potentially the whole map, in more complex algorithms) through an algorithm to determine the reward that moving/attacking a given JGPoint provides with regards to each respective strategy. The values received here are multiplied by the respective reward value specified by the AIUnit's strategy to determine which of all possible moves provides maximum benefit to the unit or the unit's team.
- **evaluateHit():** same idea, but used for determining which square the AIUnit should attack.

The algorithms in each of these methods are not assigned based on hard-coded intelligence values. Instead, the AIUnit looks at the algorithms.properties file to see the total number of algorithms and their complexities, and the **AlgorithmFactory** chooses an AIAlgorithm of appropriate complexity. This was designed using the Factory design pattern so that future programmers could write many, potentially better or more specific AIAlgorithm implementations and use them in their games by just adding one line to the algorithms.properties file that specifies its name, relative complexity, and classpath.

Authoring Environment (Xin, Matt)

(swing stuff - panels, windows, buttons)

- Preferences: size of grid, splash/bg images, settings (game speed, monster lives, etc.)
- Panels: JGEngine, ButtonPanel, GameObjectSelector, State, EnvironmentObjects
- Menu bars
- Class ButtonPanel which manages all buttons, e.g. Play, Save, Load, Preferences
- Packages and their classes:

- display
  - Constant
  - FinalResultsView
  - LevelSettingView
  - LoadGameView
  - MainView
  - ObjectiveSettingView
  - PlayerStatsTableView
  - PreferenceView
  - UnitStatsView

  I chose to make this part of the GUI to make a bit more general as an exercise to see how much we could refactor some parts of our code. I changed it to use the Chain of Responsibility design pattern so that each respective object type's selector view / instantiation code has its own class. Information provided by the user or contained in the different panels of the selector view is passed through a preset hierarchy for these classes until one can handle it.

- display.jgame
  - GameCursor: cursor for indicating which grid square is selected.
  - GameEnv: extends JGEngine; contains the elements of playing a game, such as initializing the game playing GUI, responding to commands made by the user during game play (such as moving objects and making new objects), and convenience methods that delete objects, select certain grid squares, return certain values (such as whether the game is in play mode or not), etc.
  - GridLine: used for delineating the grid squares in the game.
  - GridObject: a superclass for objects related to the grid. All classes with a "Grid" prefix in this package extend this class.
  - GridSquare: an individual grid square.

- display.animation: package for handling the animations in our game, such as when a player is attacked or when a player performs a boost action. The frames for the animations are in the animations package of the resources package, and the animations are defined in the animations properties file of the resources package.
  - Animation: animation superclass.
  - FadeAnimation
  - OneTimeAnimation: prevents animations from looping!
  - WordAnimation: animation class for displaying messages to the user.
- display.objectSelector
  - DefaultSelectorProtocol
  - EnemyHandler
  - ItemHandler
  - ObjectSelectorHandler
  - ObjectSelectorView
  - OtherHandler
  - PlayerHandler
  - SelectorProcessor
- display.util: this package contains the pieces of the GUI that can be instantiated for different purposes. Almost all classes below extend Class Panel which encapsulates a JPanel instance.
  - ButtonPanel: a generic panel to place buttons

        - addButton method: add a new button into the panel with the name of the button and a ActionListener instance
    - CheckListView: a combination of JCheckBox and JList
    - FileChooserView
        - saveFile method: prompt a separate panel for entering the save path
        - loadFile method: prompt a separate panel for entering the load path
    - ImageListView: display images inside of JList
    - ImageSelectorView: display images inside of JComboBox
    - ImageTableView: display images inside of JTable
    - Layout: a generic vertical display of different components
    - ListView: a wrapper for JList, add supports for multiple selections, toggling and button
    - MenuBar: menu bar manager
    - Panel
    - SelectorView: a wrapper class for JComboBox
    - SpringUtilities
    - TableView: a wrapper class for JTable, supports multiple selections, toggling and button

Controller (see introductory note for more information)

- connects GameEngine, Authoring Environment with Game Data
- Fields: Mode (Edit/Play)
- Methods: put objects on the GameEngine, call methods in utility class to load and save, AI algorithm to compete with player

- public void makePrefVisible();
- public void setPref(PreferenceView p);
- public void newLevel();
- public void newLevel(String gameName, String levelName, Integer rows, Integer cols, Integer tileSize, String difficulty);
- public String getCurrGameName();
- public String getCurrLevelName();
- public Level getCurrLevel();
- public void setMainView(MainView mv);
- public void addObjectiveBinding(String name, Object[][] s);
- public Object[][] getObjectLayout(String name);
- public void showObjectiveSettingView(String name);
- public void showListSelectorView();
- public void showListSelectorViewWithPrevData(GameObject unit);
- public void setObjective(String obj, List<Integer> args);
- public void removeObjective(String obj);
- public void focusBackToMainView();
- public void setJGameEngineEnabled(Boolean b);
- public void setJGameEngineSelectable(boolean b);
- public boolean checkDuplicateName(String gameName, String levelName);
- public void switchGame(String gameName);
- public void switchLevel(String gameName, String levelName);
- public void deleteGame(final String gameName);
- public void deleteLevel(final String gameName, final String levelName);
- public Preferences getUserDefaults();

- public void newObject(String name, int collisionId, int team, String graphic, StatSheet stats, StatSheet updates, String type);
- public void deleteObject();
- public GameObject getObject(int row, int col);
- public void moveObject(Unit unit, int row, int col);
- public boolean isPlaying();
- public void startPlayMode(int level);
- public void setPlaying(boolean b);
- public void startEditMode();
- public void displayStatus(String s);
- public void setStatsVisible(boolean b, StatSheet stats, JGPoint p);
- public boolean isSelectable(int row, int col);
- public void addSelectables(Set<JGPoint> points, boolean exclusive);
- public void setObjectSelectorPanelEnabled(boolean b);
- public StateManager getStateManager();
- public void playActionSound(String description);

Game Data (TC, Thanh-Ha)

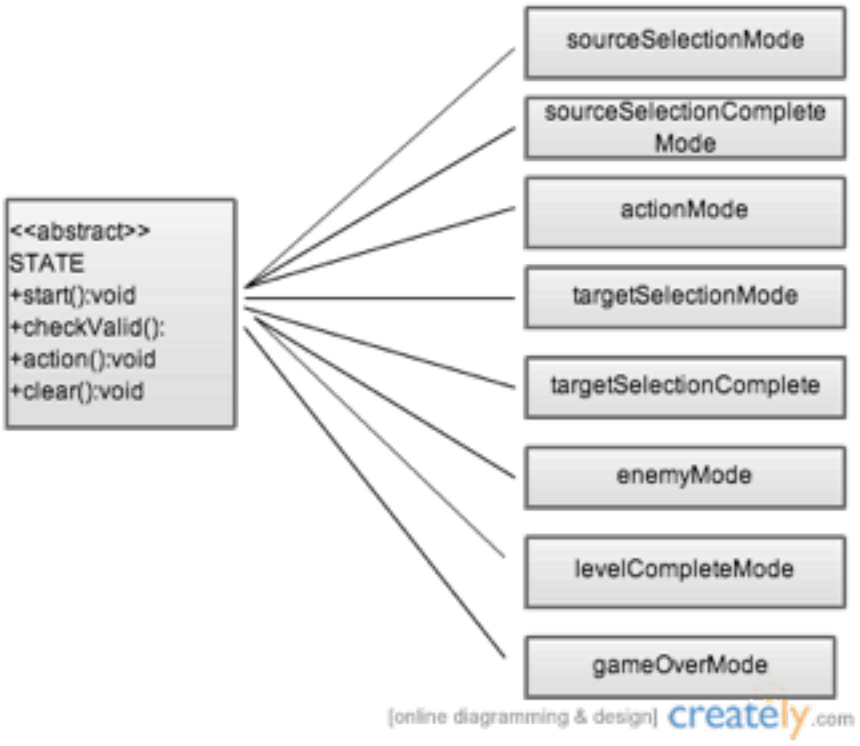
- Save GameData(Object [ ][ ] map), utility class to serialize game data into files
- Load GameData(): utility class to load files into map
- Store all game objects and preferences: class Slide (as one level) which stores all current objects and preferences, class SlideManager which stores all slides and open APIs to access active one
- JavaBeans = "JavaBeans are reusable software components for Java. They are classes that encapsulate many objects into a single object (the bean)."
    - You read and write beans in XML format using the XMLDecoder and XMLEncoder classes, respectively.

Game Player (Evan, Kanchan)

- Module which handles gameplay when program is in "Play" mode. The gameplay loop is constructed as a finite state machine, consisting of states and actions to be performed at each state, both of which will be further explored below.
- States: track current game state. The collection of states is stored in the gameplayer.modes package.
    - State (Abstract Class)
        - Fields
            - String label: used to obtain the name of the state.
            - ViewController controller: used to access methods from the program's controller
            - StateManager stateManager: StateManager object is used to update the game environment and information about the units/ objects in accordance with the changes that occur from state to state within the FSM
        - Methods
            - getStateName(): returns name of state
            - start(Set<JGPoint> selectables): initializes state given the input of the user. Depending on the current state, this input may be selected units, actions (hit, move, etc.), or destinations on the grid.

- checkValid(int row, int col): checks for valid user input. For example, when in SourceSelectionMode, a valid selection is one which represents a row and column at which one of the player's units exits
- action(): performs action as defined by the applyAction() method from the action selected in ActionMode

1. SourceSelectionMode: initial state of game, prior to object selection
2. SourceSelectionCompleteMode: after object has been selected
3. ActionMode: user selects action to perform (e.g. move, attack)
4. TargetSelectionMode: select target of action (e.g. empty cell, opponent)
5. TargetSelectionCompleteMode: after target has been selected
6. EnemyMode: when current turn belongs to AI opponent
7. LevelCompleteMode: on level completion; moves to next level
8. GameOverMode: when losing conditions have been met
9. AllOverMode: when no levels remain

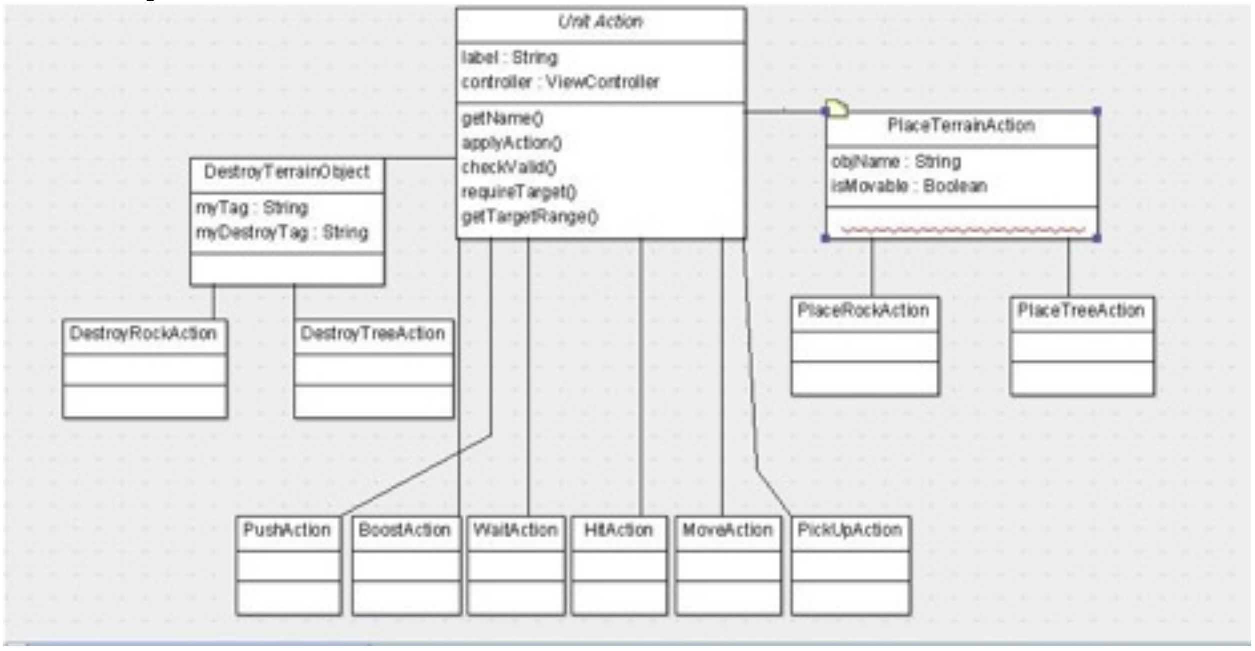


- Actions: actions to be performed by a unit in a given turn (selected in Action Mode, executed after Target Selection Complete Mode). The collection of actions exists in the gameplayer.actions package.
    - o UnitAction (Abstract Class)
        - ▪ Fields:
            - String label: name of action
            - ViewController controller: used to access methods from the program's controller
        - ▪ Methods
            - getName(): returns the name of the action (label)
            - applyAction(Unit unit, int row, int col): executes selected action
            - cheackValid(int row, int col): checks for valid user input
            - requireTarget(): returns a boolean indication whether or not the action requires a target location/unit/object

- getTargetRange(): gets the range of the target object to ensure it is within range of the source unit

1. HitAction: calls unit's "attack" method on selected enemy
2. MoveAction: calls moveObject() method from level to move selected unit from current location to destination
3. BoostAction: boosts statistics for a given unit when that unit acquires a valid object
4. Destroy[Rock/Tree]Action (extends DestroyTerrainAction): destroys/removes rock or tree from playing field
5. PickUpAction: unit picks up object and adds it to the unit's inventory, removing it from the playing field
6. [PlaceRock/PlantTree]Action (extends PlaceTerrainAction): allows player to place rock or tree at desird location on the grid
7. PushAction: selected unit moves "pushable" object one grid cell in the direction opposite the vector between the object and the unit pushing it
8. WaitAction: does nothing, moves to next turn/state

Action Diagram:



- StateManager: the StateManager maintains information about the game's current state, selected objects/units/actions, player units and enemies, and the current turn (as this is a turn-based game)
  - Fields
    - ViewController controller: gives StateManager access to controller methods
    - GameEnv gameEnv: the JGEngine canvas
    - Map<String, State> states: map of all the game's states
    - Deque<State> currentStates: Deque of current states
    - State currActionState: current turn's selected action state (Move, Hit, etc)
    - UnitAction currAction: current selected action
    - Unit currUnit: player unit which is being used in the current turn
    - JGPoint targetUnit: target of selected action in current turn

        - Set<JGPoint> potentialSpots: all available grid locations to which the selected unit can move/attack/etc.
        - Map<JGPoint, Unit> players: map of all the player's units on the grid
        - Map<JGPoint, Unit> enemies: map of all the enemy's units on the grid
        - Set<JGPoint> willPlay: set of grid locations which house units designated to play in the current turn
        - boolean playerTurn: stores the team eligible to move in the current turn (true if player's turn, false otherwise)
        - int maxLevel: highest level of the game
        - int currLevel: current level of the game (initialized at 1)
        - boolean playerWin: represents whether the game has been won by the player
        - boolean actionFinished: indicates whether the currAction has been completed
        - boolean moved: indicates whether a unit has been moved
        - int myTurnCount: tracks amount of turns taken in a current level (for use when a certain objective hsa turn constraints)
        - WordAnimation currAnimation: for use in animating text on canvas
    - Important Methods
        - resetAll(): resets StateManager, clearing out appropriate fields and reinitializing others to their orgininal values
        - addState(State state): used to add a state to the state map
        - setState(...): sets current game state based on user selections and the current state/action values
        - initLevel(boolean playerFirst, int max, Map<JGPoint, Unit> p, Map<JGPoint, Unit>): initializes level with that level's map and units
        - start(): begins FSM loop in SourceSelectionMode or EnemyMode, depending on the team eligible to move in the current turn
        - nextPlayer(): moves on to the next player from the set of players eligible to move or to the next team if the current turn has ended
        - checkObjectives(): checks level's objectives to see if they have been satisfied
        - nextLevel(): moves to the next level
        - delete/move/pushObject(...): used to remove/move/push units or objects on the grid
        - checkValidSelection(int row, int col): confirms that selected row or column contains a valid source or target, given the current state of the game
- HighScoreManager: The scorekeeping functionality exists in the gameplayer.highscore package. The HighScoreManager is used to read and write scores to a "leaderboard" for the game. The Score objects are defined by two fields: name (a String) and score (an integer). These scores get written to a .dat file upon completion of the game, which is stored in the project's highscores folder. Filename corresponds to the name of the game. There is also a HighScoreReader class for reading the contents of a score file in the console, since .dat files are otherwise unreadable.
    - Fields
        - ArrayList<Score> scores: list of scores associated with current game
        - String currGame: name of current game
        - int currentScore: points accrued in the current playing of currGame
        - File currFilePath: location of .dat file associated with currGame
        - String defaultPath: used to expedite the process of locating the game's score file within the file system

        - ObjectOutputStream outputStream: outputstream for writing a new score to the designated file
        - ObjectInputStream inputStream: inputstream for loading the current game's score file
    - Methods
        - initGame(String gameName): sets the currFilePath to the file which contains gameName's scores. If no file exists, calls createNewFile() to create a template score file.
        - getScores(): returns a sorted list of the current game's scores
        - addScore(String name, int score): loads .dat file for game and writes new score defined by name and score parameters to file
        - getHighScoreData(): returns a two-dimensional array of Strings which is used to populate the table displayed at the conclusion of gameplay
        - awardPoints(int points): increment currentPoints by specified amount
        - getCurrScore(): returns currentPoints
        - resetScore(): set currentPoints at 0

Example games:

- Kill everything in the world to win/advance
- Kill a boss to win/advance
- Reach a certain capture point (a certain tile) to win/advance
- Collect a certain amount of objects before time runs out, etc.
- Survive for a certain number of turns
- Any combination of these objectives