## 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

**1. GameEngine Interface (Teddy, Grace, Andy)**

* engine
  + **Controller** helps the back end communicate with the front end. Interacts with the 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

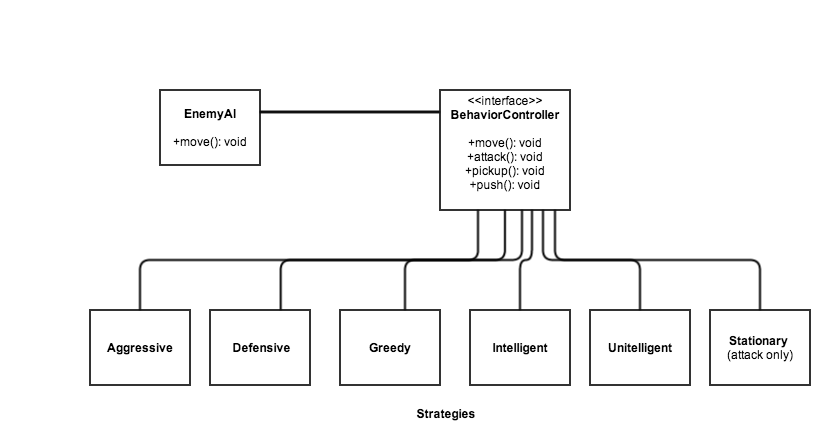
* + **Game/LevelManager** Lets the user make levels and holds them in a map with a String name as the key. Important objects include an active level, aa list of the levels made, and a map of games with game names.
    - clearAll() Allows the user to erase all of the information that currently defines the levels.
    - **Level** Makes a map of the levels and all the information (Strings of object names to objects on the levels) in the level. Also handles, creating objects, placing an object in a specific position, and deleting object, and any other information about what the game is.
    - addObjective() creates an objective before putting it into the list of gameObjects.
    - getCompleteObjectives() returns a list of completed objectives.
  + **ResourceLoader** loads in all of the game’s property files including the images, objectives, sounds, and the player stats.
  + **DeepCloner --** *throws DeepClonerException*
    - 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

Objectives are things that the user should accomplish to complete the level!

* + **ObjectiveFactory** 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.
    - The create() method in ObjectiveFactory takes the string name and the parameters of that particular objective, and creates an instance of the objective. Also throws an exception in case of an invalid objective name.
  + **Objective** is the superclass that all the specific objectives extend. Its constructor 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 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.

*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); //objectiveFactory calls create()*

* objects
  + **GameObject** extends JGObject
    - **Unit** defines a character in the game. This character can be controlled by either the computer or the player.
      * 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)
      * Set<JGPoint> getRange(GameObject[][] map, boolean isAttacking)   
        Returns a set of JGPoints on the map that the unit can move to (if !isAttacking) or attack (if isAttacking) on the current turn.
      * void attack(Unit other)   
        This allows this unit to attack another unit.
      * void damage(int amount)  
        This damages this unit by a given amount.
      * move() -- Implements one AI Strategy (see below) using the IBehavior behavior variable but this is not used if the user controls this unit.
    - **TerrainObject** -- 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. a rock).
    - **PickupObject** -- objects that can be collected, and can affect the game in some way (improve the holder’s health, increment the holder’s points, 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.
  + **StatSheet** maps stats (currently: current health, max health, power, defense, speed, range) to integer values. This can be used both to store the stats of a unit and the percent chance each stat has to grow (in the growths field).
* AI strategies (Matt, (Teddy?))
  + **EnemyAI** - based on strategy design pattern
    - Different types of concrete strategies of the enemy will implement the BehaviorController (aggressive, defensive, greedy, stationary, unintelligent, intelligent, etc)
    - void setBehavior(BehaviorController behavior);
    - BehaviorController getBehavior();
    - move()
  + **BehaviorController** (interface)
    - Looks over the whole map and sees what moves are in range.
      * These moves are assigned point values based on what particular strategy is being implemented.
      * Each strategy assigns different values to achieving certain goals (collecting a gem, attacking a unit, etc.) and then calculates the move based on the strategy



**2. 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
* Classes:
  + ButtonPanel
  + GameCursor
  + GameEnv
  + GridLine
  + GridSquare
  + GridObject
  + ImageListView
  + Layout
  + LevelSettingView
  + ListView
  + MainView
  + MenuBar
  + ObjectiveSettingView
  + Panel
  + PreferenceView
  + SelectorView
  + TableView
  + UnitStatsView

**3. Controller**

* 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);

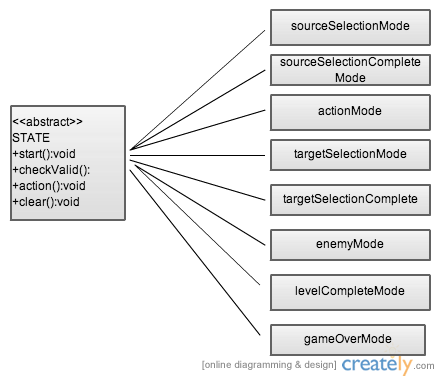
**4. Game Data (TC, Thanh-Ha)**

* 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](http://docs.oracle.com/javase/7/docs/api/java/beans/XMLDecoder.html) and [XMLEncoder](http://docs.oracle.com/javase/7/docs/api/java/beans/XMLEncoder.html) classes, respectively.

**5. Game Player ( Evan, Kanchan)**

* Module which handles gameplay when program is in “Play” mode
* States: track current gamestate
  + State (Abstract Class)
    - Fields: String label, ViewController controller, StateManager stateManager
    - Methods
      * getStateName(): returns name of state
      * start(Set<JGPoint> selectables): initializes state
      * checkValid(int row, int col): checks for valid user input
      * action(): performs action
      * clear(): clears state information

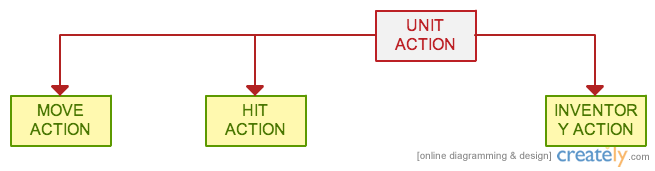
1. SourceSelectionMode: initial state of game, prior to object selection
2. SourceSelectionCompleteMode: after object has been selected
3. ActionMode: user selects action to perform (e.g. move, attack)
4. TargetSelectionMode: select target of action (e.g. empty cell, opponent)
5. TargetSelectionCompleteMode: 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 no levels remain or losing conditions have been met



* Actions: actions to be performed by a unit in a given turn (selected in Action Mode, executed after Target Selection Complete Mode)
  + UnitAction (Abstract Class)
    - Fields: String label, ViewController controller
    - Methods
      * applyAction(Unit unit, int row, int col): executes selected action
      * cheackValid(int row, int col): checks for valid user input

1. HitAction: calls unit’s “attack” method on selected enemy
2. MoveAction: calls moveObject() method from level to move selected unit
3. InventoryAction: allows user to access inventory of items

Action Diagram and State Diagram



Examples games (probably can be expanded):

* 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

### Design Alternatives

#### Game Engine Interface

There were several possible implementations for objects and objectives, and our design underwent several structural changes before it arrived at its current form. 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.

For the Objects class, we initially planned to have two superclasses, one for objects and another for player and enemy characters (called “units” in our API). Now, both of these types of objects extend a superclass called GameObject, which holds basic information about all game objects.