OOGASalad iTeam
Design Document

Team Members:

Steve Siyang Wang,Anthony Olawo,
Talal Javed Qadri, Justin Zihao Zhang,
Nick Pengyi Pan, Isaac Shenghan Chen,
Katharine Krieger, David Chou

Genre

Describe your game genre and what qualities make it unique that your design will need to support

Our game genre is going to be **Side-Scrolling Platform**. There are several unique characteristics that our design will need to support:

1. Scrolling
    a. This means that whenever the actor object moves, you should be able to see the background shift as well. The background image stays immobile, but the background objects move.t
2. Interactions
    . The game should also be able to have objects interact with one another
    a. Objects can be destroyed, bounce off of one another, or even pass straight through one another
3. Levels
    . You should be able to advance between different levels of the game, and each level should be able to have different kinds of scenery and different kinds of objects.
    a. There are specific goals for beating each level, and each level can be created separately and loaded separately.
4. Game State
    . The player should have some idea of how he or she is doing in terms of different measurements: time, score, coins, enemies killed, etc.
    a. This is a crucial component for determining level.

The framework should allow the user to be able to create games like Mario, Tiny Wings, Flappy Bird, Raptor, Falldown, and Battlestar Galactica.

For a better understanding of these games, see the following links:
Super Mario Bros
Tiny Wings
Flappy Bird
Raptor
Falldown
BattleStar Galactica

The user should also note that a limitation of the framework is that the user is not able to create games that are in 3-D.

Design Goals

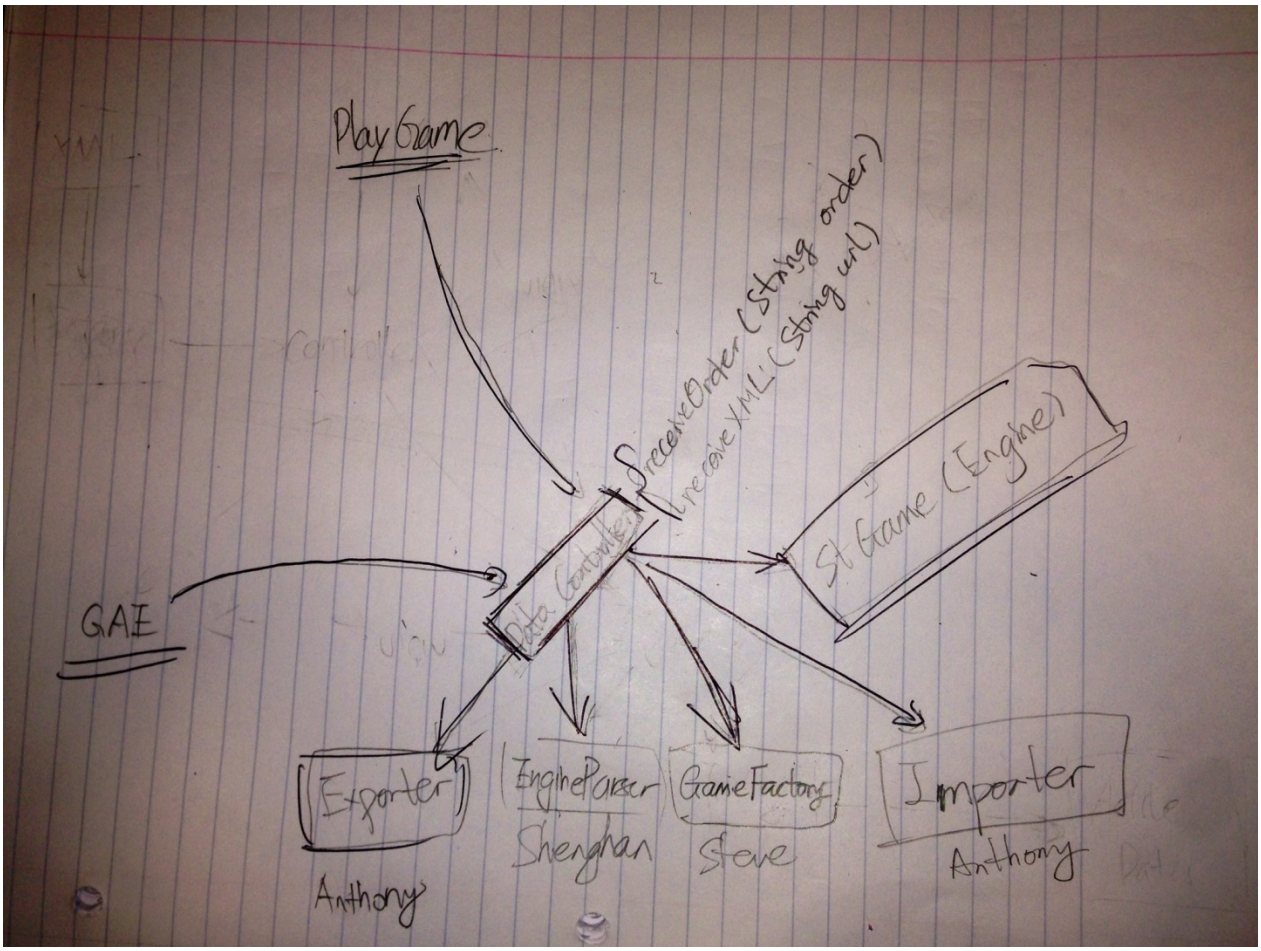
Describe the goals of your project's design (i.e., what you want to make flexible and what assumptions you may need to make), the structure of how games are represented, and how these work together to support a variety of games in your genre.

There are several main goals to this project:

1. We want to make sure that there is flexibility in the creation of different objects and interactions.
    a. For each object, you should be able to modify the image, size, x/y positions, x/y velocities, jump behavior, collision behavior, tile collision behavior, movement behavior, destruction behavior, and goal-satisfaction (scoring) behavior.
2. There should also be flexibility in preferences.
    . Not only should the user be able to modify the images from a current list of available images, but the user should also be able to load up new images.
    a. Lastly, the user should be able to save the preferences that he or she has created into an XML file.
3. These games can either be represented in a Game Player interface or a Game Authoring Environment interface.
    . Game Player
        i. Within the GamePlayer, the user will have full control of interaction in the game and will be able to command all of the individual player methods that are specified by the methods below.
        ii. The GamePlayer should also be able to specify any XML file levels that have been created within the Game Authoring Environment.
    a. Game Authoring Environment
        . In the GAE, there is more utility. The user is able to modify values of the game while the game is running, and able to add new features/objects during gameplay.
        i. The GAE allows the user to see all the objects and make modifications to them without having to actually type any code.
        ii. The user is able to see only one scene at a time, but is given the ability to switch between various scenes to modify the settings within those scenes.

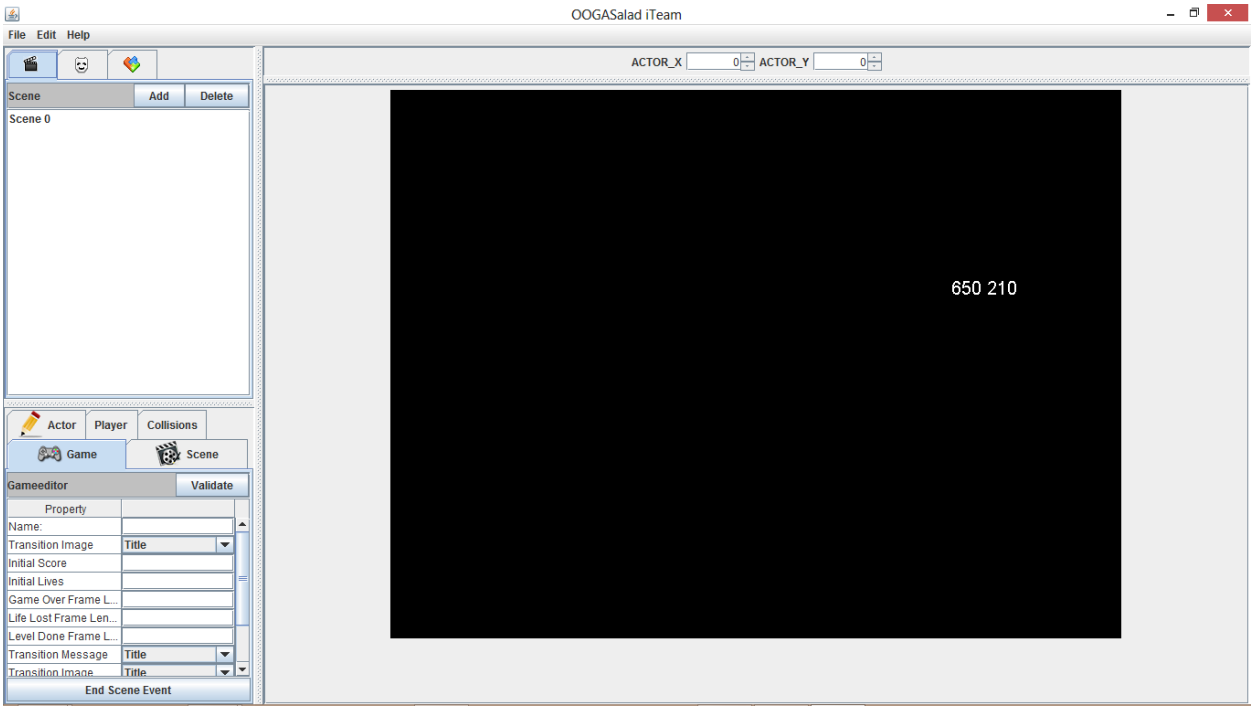
Primary Classes and Methods for Each Module

Describe the program's core architecture (focus on behavior not state), including pictures of a UML diagrams and "screen shots" of your intended user interface. Each sub-project should have its own API for others in the overall team.

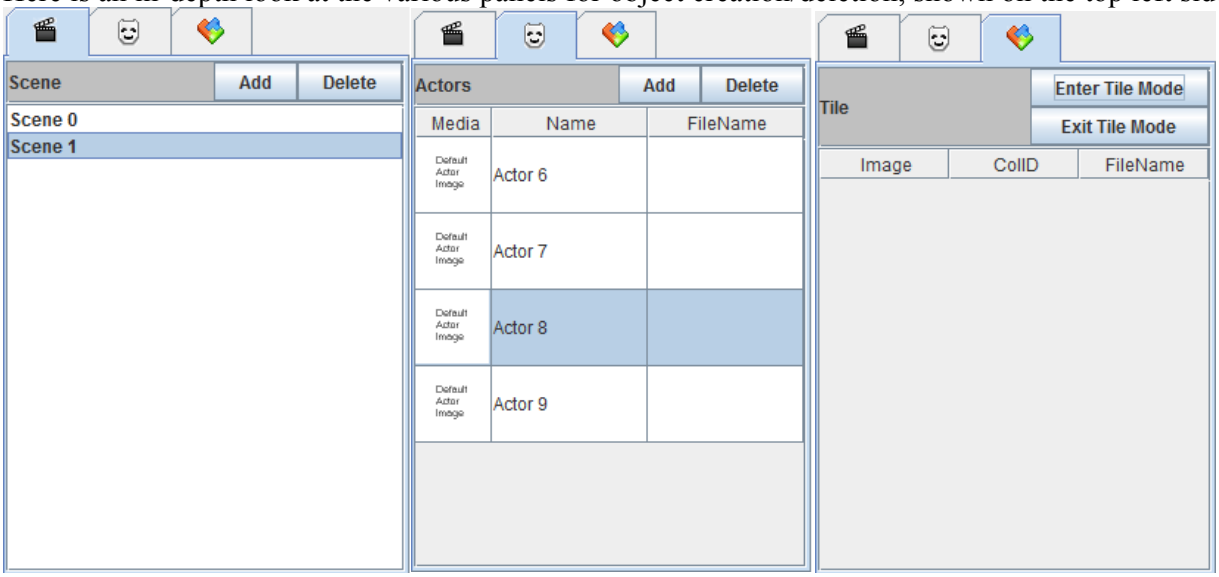


Shown above is a picture of the general framework for our API.

Below is a screenshot of the current user interface:

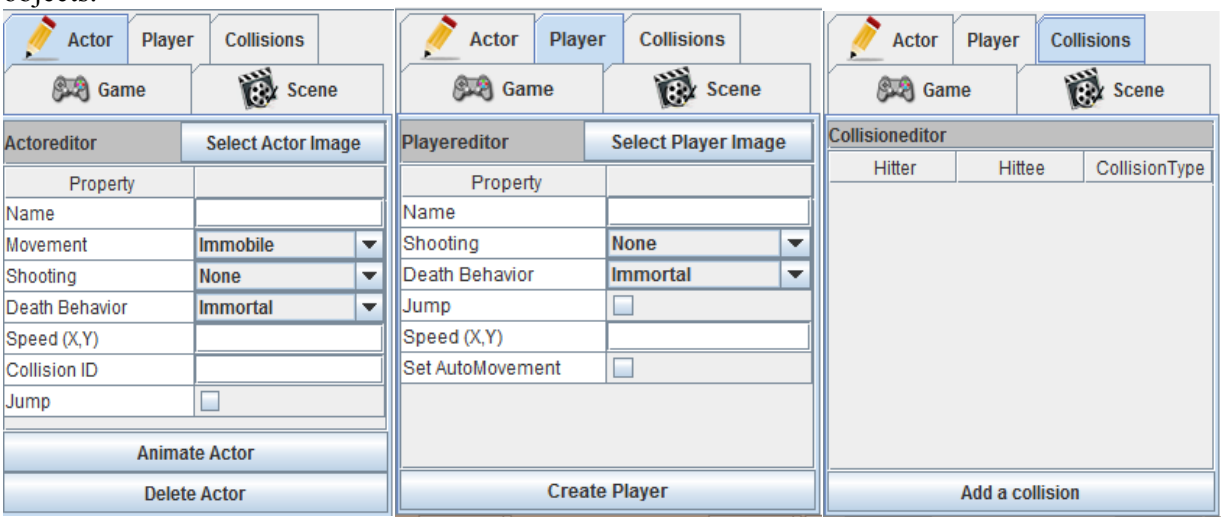


Here is an in-depth look at the various panels for object creation/deletion, shown on the top left side:



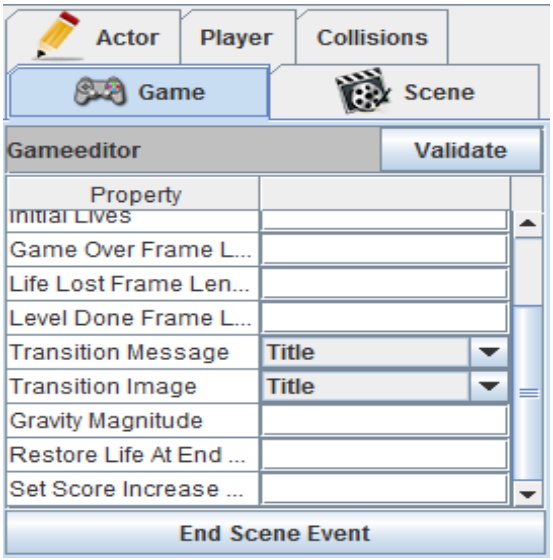
In the first tab, you are able to toggle the different scenes. Clicking on each scene will automatically allow you to display all of the objects and images in that scene. The second tab allows you to add and delete various actor objects and see which images are assigned to each actor. Lastly, we have the tile tab which allows you to specify the various types of tiles you would like, and then determine whether you would like to enable dragging for creation of tiles.

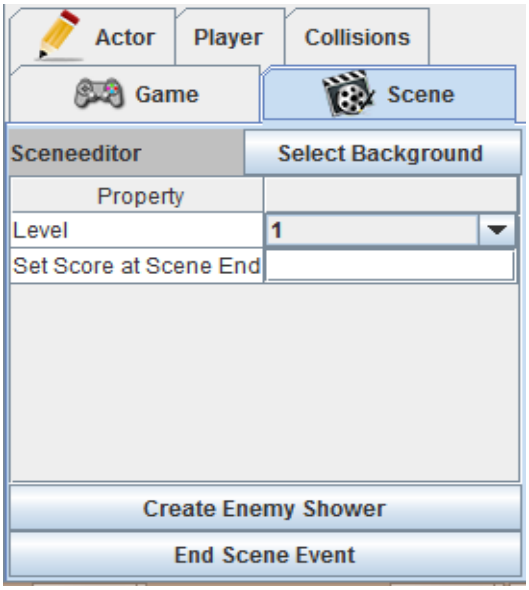
On the bottom left side, you can see the possible modifications that you can make for all of the objects.



The top three tabs for the bottom left tab allow you to modify all of the characteristics of actors (multiple instances of these can exist in the game), the player (there can only be one), and the collisions between the player and other objects.
The lower two tabs are for the game and scene modification.

Inside of the game tab, you are able to write the various characteristics of the game itself. Essentially, these characteristics are the different states of the game and are independent of the objects within the game.





The scene tab allows you to modify the number of levels, the end score that is required for each level, and enemy behavior in the levels.

The following modules below describe how each of the components operate with one another

Graphic Authoring Environment: This package contains all of the various components that have been added to the visual workspace operated by the user to develop a new game.

- The first component that is necessary is a panel that displays the Engine. Because our program uses live editing, the user is able to modify the characteristics of the objects while the animation images, movements, and behaviors are operating.
- There are additional components for panels in various actions for players (the user-controlled object) and actors (non-key-controllable objects) and the various interactions between the two.
- Lastly, there are tabs that allow the user to specify how to traverse between levels, and the various ways scores, goals, deaths, and scenes are implemented.

Panel: Panel.java is a separate abstract class within the GAE package. The majority of panels in the GAE package extend this class and this class essentially ensures that the factory design pattern can be used for creation of majority of panels and subpanels within the GAE..

- construct()
- makeSubPanel()
- makeSubPanelItems()
- getType()

MenuBar: This class creates all the menus for the GAE and it is present inside the GAE package. The menu items for all the different menu bars are created by adding the required methods to their listeners via reflection. The MethodAction class from the util package made this process much easier. Most of the listener methods are present inside this class whereas some of them are obtained from the GAEController.

- makeMenuItem(Object target, String label, String method) creates a menu item with the string “label” as its name and adds the method of the target object, specified by the input string “method”, to its action listener.
- saveGameFile() saves the game information as an XML file
- popUpAndSetKey() pops up a cheat key setting dialog

GAEController: Acts as the interface between the viewable items and the back end’s controller; takes messages from the user and translates them into object creation/modification
*It is important to note that the functions listed below may have parameters but they were not listed because many of them follow the same format. They are viewable from the following link: Data Formats

- There are creation objects for scenes, levels, and objects.
- There are modification methods for the following items in player: speed, image, ID, collision ID, position, remove behavior, shooting behaviors, jumping behavior, upwards behavior, key bindings.
- There are similar modification methods for the items in the actors, although key bindings are not available for actors (non-user-controlled objects)
- Additionally, there are methods that allow for modification of all scenes, levels, and background images, as well as collision behaviors and managers for the following: score, life, triggers, transition states, and health bars.

Game Factory: This class uses reflection to control for how objects are created and modified within the game; essentially, it serves a separate component to allow for data altering from data containment

- GameFactory
  - processOrder()
  - oneStepReflect()
  - twoStepReflect()
- OrderList: serves as containment unit for data formatting

DataController: Acts as the interface between the front end's controller and the back end engine; takes orders from the front end to create and modify objects. The existence of DataController makes the coding process much easier because it greatly reduces the dependence between the front end and the back end.

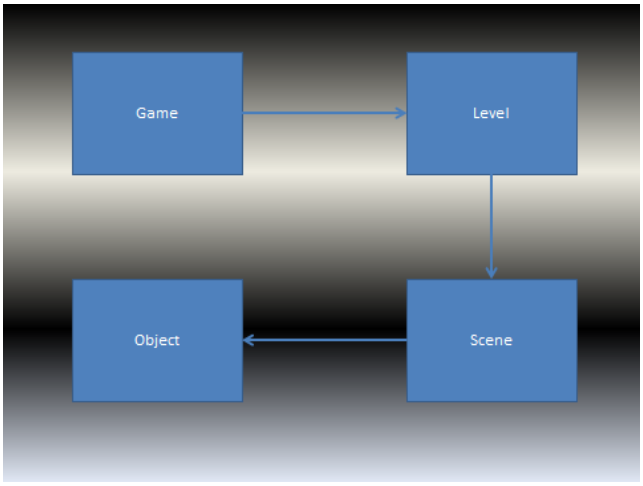
- callFactoryToProcess(String)
- initGameEngine(boolean)
- receiveOrder(String)
- exportXML(String)
- readXML(String)
- uploadImage(int, int, String)
- reviveObject()

Parser: Translates XML files into Strings for object creation and saves strings back into XML files

- class GameSaverAndLoader
  - save(List<String>, String)
  - load(String)
- class XMLReader: loads the XML
  - read(String)
- class XMLWriter: saves the XML
  - write(List<String>, String)

Engine Module: Uses JGEngine to deal with all visuals, progressions through levels, and interactions between objects

- The engine has functions for setting up the entire game and allowing the user to see: game start, game end, level done, high score, life lost
- Additionally, there are methods that check for interactions between objects and object behaviors – these are mainly for checking collisions and tracking movement of the objects



- Lastly, there is the possibility of modifying the objects
- Important Methods:
  - paintFrame...
  - start...
  - createTitles

Game Module: This module is stored in the stage package. It deals with the various hierarchies which from any given game and stores all the information related to the Game

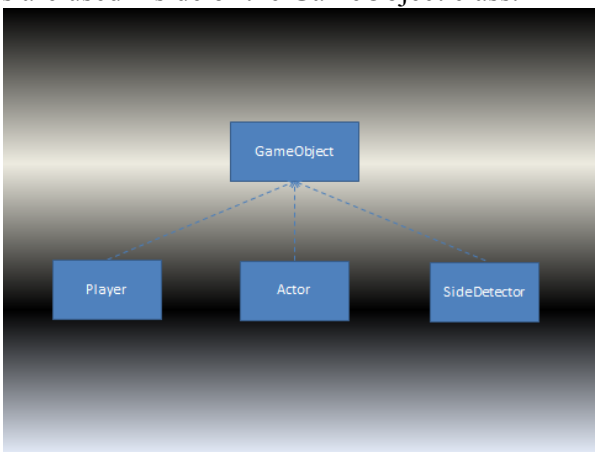
- Game Objects are placed in Scenes; Scenes are placed in Levels; Levels are placed in the Game; Managers are placed in the Game
- Every scene, level, and Game Object has its own id
- The main purpose of the classes in the Game module is to act as containment units for the object under it. This allows the user to create and remove levels, scenes and Game Objects
- An important thing to note is that the Game module also holds all of the global managers that are used in the Game
- Important Methods:
    - addLevel()
    - addScene()
    - addNonPlayer()
    - removeLevel()
    - removeScene()
    - removePlayer()

Manager Module: Managers use the delegation pattern to keep track of all object behaviors and abstract much of the functionality from the Game Objects and the Game (which stores the information about all of the objects). Statistics Manager is the super class of most of the managers so that the managers can be stored in a list within some classes (i.e. GameObject) to reduce the number of instance variables for a better and cleaner design. Please note that some of the managers use composition over inheritance technique to be fully extensible to new behaviors. For example, ActionManager manages all action behaviors for a Game Object. It has only 3 methods (the 4th one is for getting the attributes), which are setBehavior(String, Object...args), doAction(String). Whenever a doAction(String) is called, the ActionManager will use Reflection to locate its relevant behavior class (stored in the behaviors package) and perform the action. The same technique applies to InputManager, CollisionManager, TriggerEventManager, etc. By using composition over inheritance, a programmer can easily add new actions/behaviors/features without changing any codes within the managers. They just need to add the relevant action class within the behaviors package and update the properties file used by the corresponding managers for Reflection.

- There are two main types of managers: game object managers and game managers. Each manager is stored in the object according to its respective type.
- GameObject Managers
    - AnimationManager
        - updateImage(String) – for actual object
        - modifyImage(String, String) – modifies the key/value pair in map
    - ActionManager
        - setBehavior(String, Object ... args)
        - doAction(String)
        - bounce()
- Game Managers
    - BloodManager – keeps track of health
        - update() – has several overloaded methods
    - CollisionManager
        - addCollisionPair(String, int, int)
        - addTileCollisionPair(String, int, int)

        - hitObject(String, int, int)
        - hitTile(String, int, int)
        - setDirectionalCollisionBehavior(String, int, int)
        - setDirectionalTileCollisionBehavior(String, int, int)
    - InputManager – checks what keys are being used
        - initCheckKey(GameEngine)
        - checkKey()
        - setKey(int, String)
    - LiveManager
        - setInitLives(int, int)
        - changeLive(int, int)
        - addPlayer(Player)
    - RevivalManager
        - undo()
        - addRemovedObject() – adds the most recently removed object to the stack
    - ScoreManager
        - restore()
        - update(...)
    - SoundManager
    - StatisticsManager
        - update(...)
    - TriggerEventManager
        - initTEM(GameEngine)
        - update(...)
        - performEvent()
        - setEventOrTriggerBehavior()

Object Module: This class serves as the primary basis for all objects/actors/players that are created inside of the game. A simple inheritance hierarchy is used for these various objects with GameObject being the superclass for all other objects. Many managers are used inside of the GameObject class. For more information on how they are used, see above.



- GameObject (extends JGObject)
    - setXHead(), setYHead() – these determine what direction to shoot objects as well as movement direction
    - resume() – unfreezes objects
    - suspend() – freezes objects
    - bounce()
    - stop()
    - ground()
- Player (extends GameObject)
    - moveUp/Down/Left/Right()
    - setCanMoveInAir()
    - setPlayerSpeed()
- Non-player (extends GameObject)
    - move()
- SideDetector (extends GameObject)
    - move()

- changeBlood()
- stop()
- ground()

Example Data Files

Rather than providing specific Java code, you should describe two or three example games from your genre in detail that differ significantly and the data files that represent them and could be saved and loaded by your project. You should use these examples to help make concrete the abstractions you have identified in your design.

Game.xml

```
<list>
  <string>ModifyGravity,Magnitude,0.0</string>
  <string>CreateLevel,ID,1</string>
  <string>ModifyLevel,ID,1,SetInitialScene,0</string>
  <string>CreateScene,ID,1,ID,0</string>
  <string>SwitchScene,ID,1,ID,0</string>
  <string>ModifyScene,ID,0,PlayerInitialPosition,0.0,0.0</string>
  <string>CreateTile,TileMap,0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 </string>
  <string>CreateScene,ID,1,ID,1</string>
  <string>SwitchScene,ID,1,ID,1</string>
  <string>ModifyScene,ID,1,PlayerInitialPosition,0.0,0.0</string>
  <string>CreateActor,ID,1,Image,actor_blank.png,100,100,Position,100.0,200.0,Name,Actor
1,Colid,1,Lives,1</string>
  <string>ModifyActor,ID,1,RegularMove,RegularMove,1.0,1.0</string>
  <string>CreateActor,ID,2,Image,actor_blank.png,100,100,Position,100.0,200.0,Name,Actor
2,Colid,1,Lives,1</string>
  <string>CreateActor,ID,3,Image,actor_blank.png,100,100,Position,100.0,200.0,Name,Actor
3,Colid,1,Lives,1</string>
  <string>CreateTile,TileMap,0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 0000000000000000000000000000000000000000
0000000000000000000000000000000000000000 </string>
  <string>ModifyLiveManager,RestoreLifeByLevel,true</string>
  <string>ModifyScoreManager,InitialScore,0</string>
  <string>ModifyGame,SetName,Game</string>
</list>
```

Alternatives to Design

Explain some alternatives to your design, and why you choose the one you did.

Player/GAE Separation

One possible way we could have designed the program was to have the player available for instantiation in the game authoring engine, rather than having the **Game Player** and **Game Authoring Environment** have separate views. The reason we decided to separate them in the end was because even though gameplay between the two were quite similar, the **GAE** would also need to have a slurry of additional listeners added that were not necessary inside of the view of the **Game Player**. Because of this, we decided that it would be better to have the hierarchy shown in the UML Diagram, which displays the **GAE** and **Game Player** on the same level.

Multiple Engines

Additionally, we also considered having multiple engines for each level/stage of the game. This idea was given moderate consideration but pretty quickly thrown out as we realized that it would be more efficient to have a list of different stages inside of each engine. Each stage would then be able to distribute the proper information for object creation whenever some currentStage variable was increased/decreased. That way, instead of having to instantiate a new engine every time, what we could do instead would be to just instantiate a new view for the stage level, and the **Game Engine** would hold all of the possible stages inside a list. A main reason we decided to do this would be so that you do not need to have multiple engines running simultaneously and have to hide all the engines you are not running on. Although such a variant would not necessarily affect gameplay, it certainly makes working within the **Game Authoring Environment** more efficient.

A Little Less Reflection

A final aspect of the design that could have been altered would be to use more HashMaps to store data instead of using so much reflection. Having multiple layers of reflection makes it difficult to implement things on the front end because the changes between on reflection method and another are generally pretty small, but the front end is required to make a new component that results in repeated code. It is possible that using the Features utility class that was created by Jacob Lettie would have allowed the front end workers to minimize the work they would have to do in order to create new panels and tabs for the front end.

Division of Labor

List of each team member's role in the project and a breakdown of what each person is expected to work on.

The project is broken down into subteams and the individual components that each person is expected to work on is listed below:

- Engine and Game Player
    - Steve Siyang Wang (GameFactory, TriggerEventManager, Eventable)
    - Justin Zihao Zhang (DataController, GameObject (NonPlayer & Player), Game, Scene, Level, ActionManager, BloodManager, LiveManager, ScoreManager, StatisticsManager, InputManager, all classes in behaviors package except those related to events (Eventable) module), Gravity, Integration of GameStats Util, AttributeMaker, SaladUtil)
    - Isaac Shenghan Chen (GameEngine, SideDetector)
    - David Chou (Player, AnimationManager, RevivalManager, ImageBuffer)
- Game Authoring Environment and Data
    - Katharine Krieger - GAE & Controller
    - Anthony Olawo - Parser/Data/ XML
    - Talal Javad Qadri -GAE & Controller
    - Nick Pan - GAE & Controller

Thanks for reading!