API Design Document

Genre

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

We are going to make an RPG game building engine. Thus, it will need to support grid-based overhead movement, and a multiple-state system to handle map navigation, menu navigation, and battles. So we will need to have support for multiple states, tile setup, interaction with objects on tiles, items, and player and enemy stats.

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 4 main subteams for this project. The Data team will have its components communicate with the Authoring Environment and the Player teams. In addition, the Engine will interact with the Player.

The user will be able to create a game through the Game Authoring Environment, which will allow them to build their world map-grid of images, place GridObjects such as walls, enemy-encounter spots, NPCs, or other things that would emit dialogue and give/take items such as treasure chests. We will also allow the user to upload enemy images and stats to be used in a battle system associated with the random-encounter tiles, with character stats and leveling. A menu system involving items will allow interactions with the NPCs to be item-based and more complex, so we intend to add this as well. All of this will allow the user to do a plethora of things--they are only limited by the grid structure, and their ideas behind how the user is to interact with the NPC-type grid objects. The user could make a Final Fantasy-like game, a maze full of riddles, a multiple-storyline journey like The Stanley Parable, or any number of things.

As the user builds the game, the Authoring Environment gives its information to Data to be processed and stored. Player will then access this data and build the specific game the user has created, using the underlying framework created by Engine.

There are a few goals we see as essential to the design of this project:

Focused Flexibility: Given how open ended the RPG genre is, it is imperative that we focus our project on a subset of possible features within RPGs. However, within that subset, we want to be as flexible as possible with regards to how features or elements of the game can be utilized and combined. The goal here is that a user designing with our authoring environment immediately knows what specific type of game we support, but feels totally free to design a game within the bounds of that game type.

Clearly defined API's for each module: Given the massive scope of this project (with respect to the types of project we have had experience with), it is imperative that we define the publically accessible methods for each module early on and do not stray too far from our initial design unless absolutely necessary. Otherwise, we will spend far too much time having to work on the interfacing between

modules, rather than on features of our project

Enclosed modules: The only way each module team will be able to continually add, extend, and expand upon their code is if they do not have to worry about impacting other modules' code when making changes. As such, it is essential that data, game engine, player, and authoring environment teams focus on encapsulation.

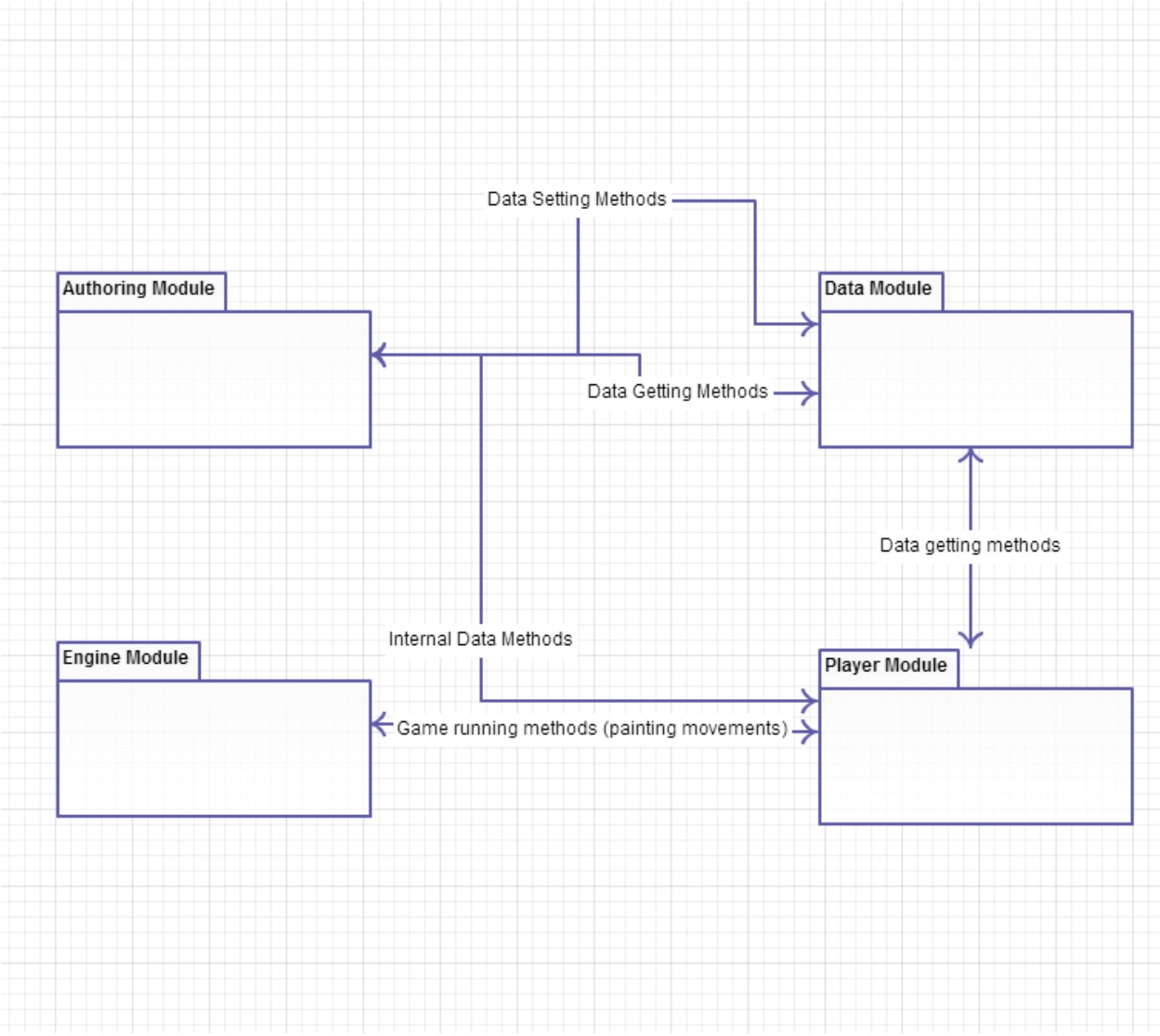
Test Driven Environments: Given the fact that multiple teams will be independently making changes, altering code, and pushing those changes back to the main repository, it is essential that teams are able to ensure that the changes they have made did not break the project. The only way to do this effectively and efficiently on a scale like this is to have all-encompassing, versatile tests for each aspect of Oogasalad.

Communication: The only way a group of this size can be successful in handling the various issues that come up in the process of designing a project so large is if the members communicate well. This includes actively working with members of the group outside of your team so that the interface between modules is designed well in addition to constantly updating members within your team so that people are on the same page.

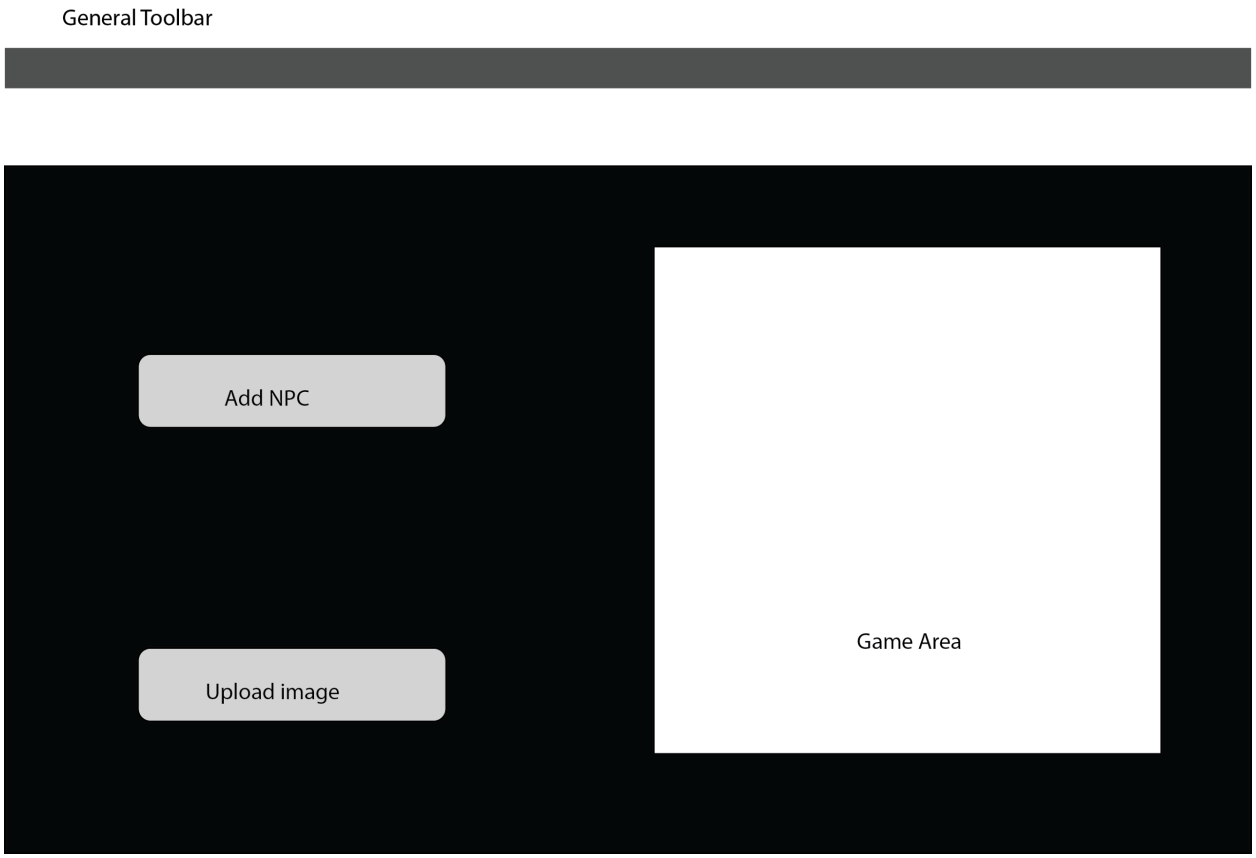
Primary Classes and Methods for each module

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.

A basic model of how we envision the various modules in the program interacting is shown below. Further details about the methods and classes of each module follow later in this document.



On the next page is a basic module of how we envision the authoring environment GUI to look.



Authoring

Public API Methods:

```
/**
* Adds a new level to the game
* @param s Name of the level
* @param md MapData of the added level
*/
public void addLevel(String s, MapData md) {
	myLevels.put(s, md);
}
/**
 * Returns the current map (the map the user is currently viewing)
 * @return MapData of the current map
 */
public MapData getCurrentMap(){
	return myLevels.get(currentMap);
```

```
    }
    /**
     * Returns the arena labels currently being used
     */
public String[] getArenaLabels(){return arenaLabels;}
/**
* Returns the name of the current Map
* @return String name of map
*/
public String getCurrentMapName(){
    return currentMap;
}
/**
* Returns the File corresponding to the specified image name
* @param fileName Name of the image
* @return File of image
*/
public File getImage(String fileName) {
    return myImages.get(fileName);
}
    /**
     * Returns the song with the given filename
     * @param fileName Name of the song
     * @return File of the song
     */
    public File getSong(String fileName) {
           return mySongs.get(fileName);
    }

    public String getSongString(String fileName) {
           return mySongs.get(fileName).getPath().replace("\\", "/").substring(3);

    }
    /**
     * Returns the map with the specified name
     * @param s Name of the map
     * @return MapData of the given map
     */
    public MapData getMap(String s) {
```

```
        return myLevels.get(s);
    }
    /**
     * Returns all maps saved in the game
     * @return Mapping of map names to their MapData
     */
    public Map<String, MapData> getMaps() {
        return myLevels;
    }
    /**
     * Returns all items stored in the game
     * @return Mapping of item names to their itemData
     */
    public Map<String, ItemData> getMyItems() {
        return myItems;
    }
    /**
     * Gets the random enemies stored in the game
     * @return
     */
    public List<RandomEnemy> getMyRandomEnemies(){
        return myLevels.get(currentMap).getMyRandomEnemies();
    }
    /**
     * Gets the current player data
     */
    public PlayerData getPlayData() {
        return playerData;
    }
    /**
     * Returns the current primary map
     */
    public String getPrimaryMap() {
        return primaryMap;
    }
```

```
	/**
	 * Gets the weapons currently saved
	 * @return Mapping of weapon names to their WeaponData
	 */
	public Map<String,WeaponData> getMyWeapons(){
		return myWeapons;
	}

/**
* Saves a healer
* @param myHealer HealerData to be saved
*/
	public void saveHealer(HealerData myHealer) {
		myLevels.get(currentMap).saveHealer(myHealer);
	}
	/**
	 * Saves a song
	 * @param s Name of song
	 * @param f File of song
	 */
	public void saveSong(String s, File f) {
		mySongs.put(s, f);
	}
	/**
	 * Saves an item
	 * @param n Name of item
	 * @param it ItemData to be saved
	 */
	public void saveItem(String n, ItemData it){
		myItems.put(n,it);
	}
	/**
	 * Saves the player
	 * @param player PlayerData to be saved
	 */
	public void savePlayer(PlayerData player) {
		playerData=player;
		//myLevels.get(currentMap).savePlayer(player);
	}
```

```
    /**
     * Saves a weapon
     * @param n Name of weapon
     * @param wp WeaponData to be saved
     */
    public void saveWeapon(String n, WeaponData wp){
            myWeapons.put(n,wp);
    }
     * Sets the current map with the string name
     * @param s Name of the map
     */
    public void setCurrentMap(String s){
     currentMap = s;
    }
/**
* Sets the primary map
* @param s Name of the map to make primary
*/
public void setPrimaryMap(String s){
    primaryMap = s;
}
    /**
     * Saves an enemy to the current map
     * @param enemy EnemyData to be saved
     */
public void saveEnemy(EnemyData enemy) {myLevels.get(currentMap).saveEnemy(enemy);;}
    /**
     * Saves a shopkeeper to the current Map
     * @param shopkeeper to be saved
     */
public void saveShopkeeper(ShopkeeperData myShopkeeper) {
            myLevels.get(currentMap).saveShopkeeper(myShopkeeper);
}
```

The Authoring module will interact with the Data module to provide all necessary information for a game to be created. The user will be able to upload images for tiles, and on-tile objects. When the user uploads an image for later use, they will give it a name and they will use a file chooser to select a file it goes with. Then we would use Data's storeImage(String name, File image) method to let Data know to add this image to its map of strings to images. Later, Authoring and Player would be able to use Data's

getImage(String name) method to retrieve that image for later use. For map storage, we would use Data's storeWorldData(String name, WorldData world) method to let Data know to add this map's WorldData object to its map of strings to WorldDatas (the Player module could then use getWorldData(String name) to retrieve these maps, which would contain getTileData(int x, int y) that would allow them to retrieve TileData objects corresponding to tiles on the map). TileData objects that the WorldData returns would contain coordinates, the name of their corresponding image, and a list of the GridObjectData objects corresponding to all the things on that tile. For GridObjectData storage, we would use Data's storeGridObjectData(String name, GridObjectData t) method to store GridObjectData objects in their string to GridObjectData map. Then we could use getGridObjectData(String name) to retrieve these objects for display and potential edit. This allows for smooth communication of Data with the Player, as all they need to do is retrieve WorldData objects to build the map (using its getTileData method to access coordinates and images of tiles), and retrieve all the information it needs for GridObjects by calling getGridObjectData method.

Classes

- AuthoringViewer
    - Uses Feature objects in features package
        - MapBuildFeature
        - GridObjectFeature
        - ImageAddFeature
        - EnemyAddFeature
        - ItemAddFeature
        - PlayerEditFeature
    - Data-Passing Objects
        - WorldData
        - TileData
        - GridObjectData

Data

Data will be responsible for storing classes retrieved from the authoring environment, parsing them in JSON and passing on any required information to the player class. The authoring environment will communicate with the data to let data know what setters they want to call on the information in data. The player will communicate with the data by letting data know what information they want to be able to retrieve. The authoring environment is going to pass WorldData objects to data which player can then retrieve to access to retrieve the information they need.

Methods Other Modules Call:
Authoring Environment will call on Data:
storeImage(String name, File image)
getImage(string name)
storeWorldData(String name, WorldData world)

getWorldData(String name)
storeGridObjectData(String name, GridObjectData t)
getGridObjectDatas()

Player will call on Data:
getWorldData()
getImage(String s)

Module 1 (interaction with authoring environment)
The data is going to have “front end” storage and “back end” storage. The front end storage is essentially going to be updated incrementally as the authoring environment passes information to data using the storeData method. When the create game button is called, the incrementally stored data will be converted to a JSon file.

Module 1 Classes
DataCompiler: The “front end” back end compiler. Stores the information being sent to us, loads the correct image so the front end can get them, compiles data to be stored
-This may include subclasses/partner classes that help store specific components
DataConverter: Converts data to the JSON file.

Module2 – Load module (interaction with player)
The front end will select the data they want to load at which point the data will call the load data method. When the load data method is called, the stored Json files will be parsed and the necessary information will be passed to the player class (i.e. WorldData objects).

Module 2 Classes
Parser/Data Reading: This looks at the data files selected via the front end

Player

Classes (in gameview package):

- GameFrame
    - Extends RPGEngine and essentially is the game.
    - Sets up the tile images and grid objects onto the canvas

- MapDataParser
    - Parses through the data to create the lists of tile images and grid objects
    - Uses reflection to create the generic grid objects
    - MapDataParser creates grid objects and gets tile images
- Creator
    - Creates the player, items, and weapons from data
    - The player, items, and weapons are objects saved separately from the grid
    - Creator is different from MapDataParser in that it creates the special objects not contained in the grid
- GameChooser
    - Creates the 'game chooser' screen and is the first 'world' that initializes the rest of the game

  Title Manager

  Creates the 'title screen' where the user can type in a string to load a game.

Classes (in engine.menu):

- Contains manager package and node package. The classes in these packages provide the interface for the frontend connection from the data. Moreover, the Menu classes provide the front end interface to load and save in game.

Methods Player Calls:

From Engine:

- initializeCanvas(int width, int height)
- setWorld(World world)
- setTileObject(GridObject obj, int xTile, int yTile)
- setTileImage(int i, int j, String fileName)

From Data:

- DataManager.getWorldData(String fileName)
- World.setWorldDataManager(WorldDataManager wdm)

From Authoring:

- WorldData.getMaps()
- WorldData.getSongString()
- WorldData.getPrimaryMap()
- WorldData.getPlayData()
- GridObjectData.getArguments()

The Player module will interact with both the Engine and the Data. The Data team will parse the XML/JSON file Player can serve as a view for the Data. For instance, sprites, maps, etc can be displayed from the Player module through a GUI.

Interaction with the Data team will largely be through getMethods(). The Player team will be communicating the Authoring team to decide on what features/properties the Data team needs to set() and get().

The interactions between the Player and the Engine are that the Player will utilize the methods/classes implemented in the Engine to create the game. For example, the Engine will be responsible for defining classes such as Tile, Tile Object, as well as methods such as setTileXY, setImage. The Player will actually use these to create the map, and the NPCs and the playable character.

Game Engine

The Game Engine will provide a functionality interface to the Player, allowing the Player to run a given game given specific data. Given that, the Game Engine will not be communicating with any other part of the overall program.

We plan on creating our own Engine and not using JGame at all. Specifically, the Engine will provide the Player module to be able to instantiate objects on the screen, move objects given user input, control movement of sprites on the canvas, transition between different "worlds," and keep track of game state. See below for the API we provide to the Player module.

API
Methods called by Player

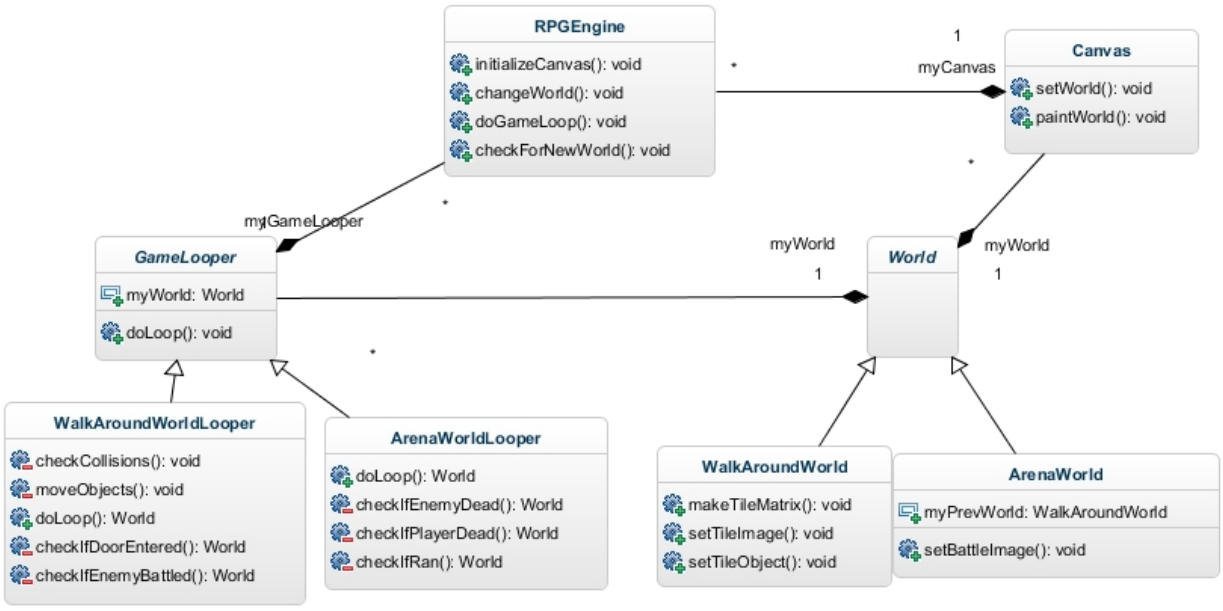
- initializeCanvas(int width, int height)
- setWorld(World world)
- setTileObject(GridObject obj, int xTile, int yTile)
- setTileImage(int i, int j, String fileName)

Classes

- RPGEngine (abstract class, games will create a class that extends RPGEngine)
    - this manages which of the worlds is currently active
    - Initializes the canvas
    - Allows for changing worlds
    - Contains main game loop (repaints and manages the current gameLooper)
- World (abstract)

- WalkaroundWorld: Keeps track of gridobjects in the world
  - List of tiles and gridObjects in the world
    - Tile: contains one or more objects and an image
  - Keeps track of the collision matrix/collision handlers for all grid objects
  - Keeps track of a list of random enemies that can be encountered in the world
  - Collision matrix: keeps track of each of the objects and their respective collision handlers
    - Collision handlers include bump collision, enter collision, battle collision, null collision (Mediator Pattern)
- Arena (turn based)
  - Two sprites in arena
  - TextDisplayer : Currently highlighted Option, List of possible Options
  - BattleManager: Keeps track of two sprites, sets battle nodes and their actions (attack,weapon,bag,run)
  - BattleCalculator: Keeps track of who attacks first, calculates attack damage/effects, check if the enemy has been defeated
  - BattleAI: Chooses an attack and weapon for the enemy

- GameLooper: Changes to a specific looper depending on the world you are in. The main function of the game looper is to change the world. The doLoop method returns a world and is monitored by the main game loop in rpg engine (Observer Pattern). Whenever the doLoop returns anything other than a null, the world is changed in RPGEngine
  - WalkAroundWorldLooper: Active when the player is in a walkaroundworld. Checks for collisions and moves objects. Changes to a new walkaround world if a door is entered. Changes to an arenaworld if an enemy is encountered.
  - ArenaWorldLooper: Active when the player is in an ArenaWorld. Checks if the player won the battle, the player lost the battle, or the player ran from the battle. Returns the WalkAroundWorld that the player was just in.
- Canvas: Paints the objects and images on the World. Basically the front end portion of the world.



This diagram represents the basic interaction between the RPGEngine, Canvas, World, and GameLooper. The RPG Engine deals with changing worlds and contains the main game loop. The RPGEngine has a GameLooper (abstract) that has two functions: 1 - completes any tasks that need to be done to the world every iteration of the loop (including checking collisions and moving objects) and 2 - signals the RPGEngine if the world changes (by entering doors, entering battles, or ending battles). Each Game Looper has an instance of a World. The world keeps track of the objects on the screen. The RPGEngine also has an instance of a Canvas which paints the current world.

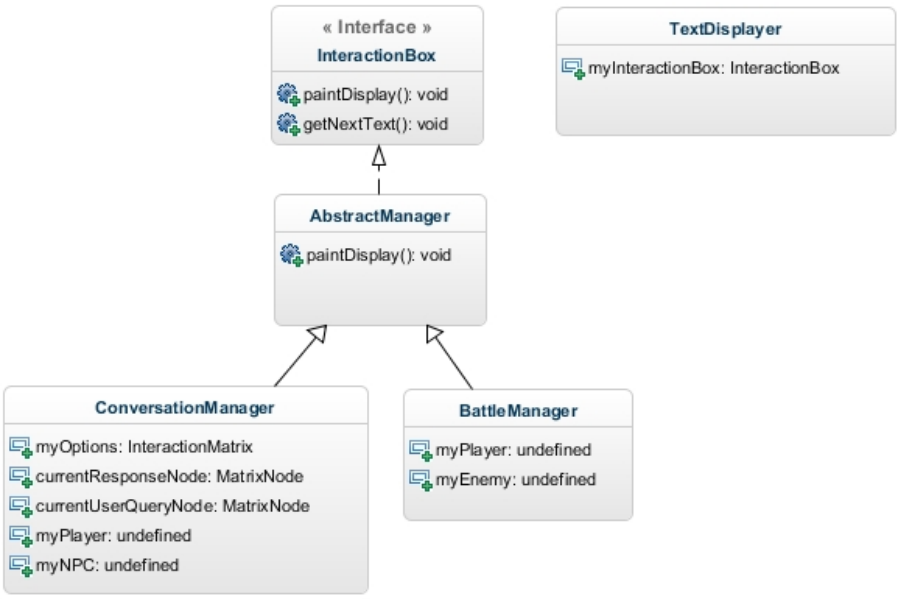
- GridObject (abstract): Any object in a WalkAroundWorld, has a position, tilex, tiley, interaction box. An object can have any collisionhandler
    - Subclasses
        - Barrier: A grid object that doesn't move. A building, terrain element, or other object.
        - Door: has a world that can be either a walkaroundworld (regular door) or an arenaworld (random enemy encounter).
        - Person: A grid object that can move. Has money, experience, items, weapons
            - Player: The grid object that the user controls. Reacts to key presses, checks surroundings, levels up
            - NPC: A non playable character. Can move based on a predefined set of movements (static, back and forth, or follow player). Has response nodes for conversations. Can give items to the player or see if the player has items to influence the game.
                - Enemy: Does a battle. Has an arena world so that a battle can be entered
                - Healer: Heals the player on interaction and has predefined dialogue
                - Shopkeeper: Has predefined dialogue and has items that cost

money that can be sold to the player

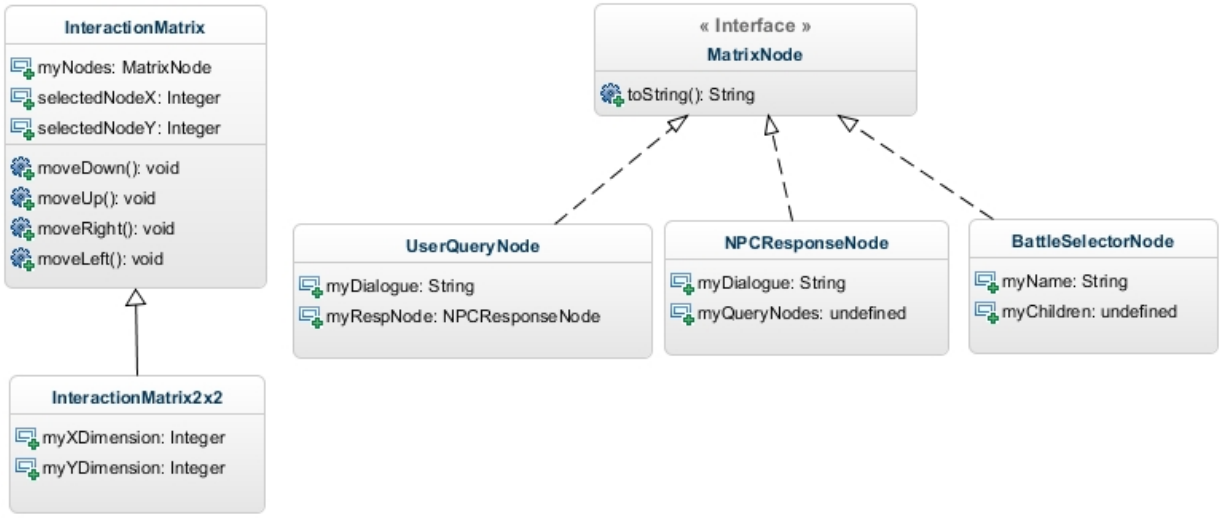
- State: controls what the keypresses do depending on whether you are in a WalkAroundWorld, battle, conversation, bag, etc...

- Item (abstract): Can be held by a person and used
    - KeyItem: does nothing when used. Used to progress in the game by talking to NPCs
    - StatBuffer: changes a statistic when used
- Miscellaneous Classes:
    - Weapon: Can be held by a person. Has a list of attacks and a speed/damage
    - Attack: An attribute of a Weapon. Has a speed/damage and an optional effect
    - Effect: An attribute of an Attack. Can change a person's statistic
    - Statistic: has a value, a max value, and a name. Persons have a health, defense, damage, speed and damage statistic. Attacks and Weapons also have speed and damage statistics. Contains helpful methods for changing statistics and setting them to a maximum value

- Dialogue: See UML diagram below
- MenuSystem: See UML diagram below. It's pretty much identical to the Dialogue UML diagram in that the menu system uses managers to display all of its information, an interaction matrix that provides it's selection grid, an interaction box that interacts with the high-level canvas, and nodes that connect menu classes.



As one can see, the TextDisplayer takes in an InteractionBox. This could be a menu system, or dialogue



UML Diagram

Example code

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.

Our design will be extensible enough to create many different games within the RPG genre. Beyond the obvious example of any sort of Final Fantasy/Pokemon type game, choose-your-own-adventure esque games such as The Stanley Parable would be possible thanks to NPC interactions, items, map changes, etc. Other possibilities are maze games, puzzle games, and even dating simulators!

All game types will use JSON data files, which will be handled by Data and passed to Authoring and Player when necessary.

Alternatives

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

Engine

- For our project we decided to implement the Control (what button presses do), in the Player. Because of this, we were able to implement a composition pattern where our Player object had a State object that would do what it was supposed to do (move the player, continue through dialogue, pick up object) depending on what type of State object it was. This was great and allowed for versatility in what we could achieve in terms of different state in our game. This design choice, however, complicated somethings as any time we wanted to change the state of our game, the Player had to be involved. Our managers would have instances of Player and so would other classes, which at times was not optimal. In the end, though, we decided to keep

Player in control.

- We implemented the world managing using game loopers. The loopers contain a doLoop() method that is constantly monitored by the main game loop in the RPG engine. If the doLoop() method returns a world (or, in other words, not "null"), the RPG engine sets this world as the new world and instantiates a new game looper. An alternative to this would be to have a World Manager that oversees all of the worlds, changes worlds, and keeps track of the states of every world. This design would not be as elegant as the one we used because all of the worlds would be mushed into one array. In our design, only one world matters: the current world, and all other worlds are comprised in objects (doors and enemies). The gameLoopers act as an observer which merely change the worlds; they don't keep track of all of the worlds.

Authoring Alternatives:

For the tile, it was recommended that NPC characters and events be inherent methods of a tile. But because such a relation could cause many dependencies, it was decided that NPC characters and events would be separate from the tile, and lay on a separate "layer" on top of the tiles that make the world/game map. With the chosen design, each tile will have a list of events and NPCs that are related to the respective tile. This separation between the tile and events/NPCs provides a layer of abstraction that allows for maintainability and extensibility.

One alternative design was to have Data and Player parse the JSON files. However this was deemed to be too cumbersome. The task would be repetitive, and it would be better for the Data to have a flexible API that the Player can take advantage of. Therefore, the Data will be parsing the JSON file.

The Data suggested that all objects be saved before rendering to the view panel and processing the player and game engine. However, to make the game interface more usable, it was suggested that instant object retrieval be possible. Therefore, Data should save as the creation of the game progresses. The end result would be the possibility of a soft and hard save.

Game objects could have been created used a Factory pattern. Since players, map enemies, and random enemies are types of people, the user doesn't actually need to know the subtype of person. A createPerson() method within the Factory could return a player, map enemy or random enemy. With the Factory the user would only be able to access and see the necessary value fields for the type of person (this would be different from the fact that we currently gray out and disable fields).

The Factory pattern could also be used for various types of NPCs such as healer, shopkeepers and interactive items.

Instead of having different creation GUI for each game object, we could add reflection to create a certain instances of the required game objects. This would reduce the number of classes significantly in the authoring package. Utilizing reflection with the Factory pattern would make the authoring system much leaner compared to its existing form.

Roles

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

Authoring Environment

- Jacob
- Pritam
- Richard

Inside the Authoring Environment we hope to be able to eventually compartmentalize our work enough to be able to work independently,

Engine

- Parker
- Lee

Data

- Davis
- Sanmay

Player

- Peter
- Brandon
- Benson