API Design Document

Team Information

Name	NetID	Role
Yu Zhou Lee	yl218	Game Authoring (Front-end)
Tze Kang Ng	tn52	Game Engine
Amy Wang	alw53	Game Engine and Game Authoring (Back-end)
Dan Zhang	dyz2	Game Authoring (Front-end)
Viju Mathew	vjm7	Game Authoring (Back-end)
Tara Gu	wg37	Game Engine and Game Data
Zanele Munyikwa	ztm3	Game Engine
Ashley Qian	aq11	Game Player and Game Engine
Timothy Shih	ts130	Game Engine
Trey Bagley	dsb25	Game Engine and Game Data

Genre

Our game Genre is the Role Playing Game - the RPG has the following general characteristics:

1) Growth - User controls character/characters where character advances through a concept of self improvement, either through development of parts of the character or through advancements towards some goal. This could include gaining more skills, more tools, improved skills, improved tools, or a combination of these.

2) Exploration - There is a world for the player to travel through and explore. The size of the world varies depending on the game, as well as whether or not the player can manipulate the environment. A common theme through RPGs is interaction with the explored environment which could lead to picking up new items or meeting new characters.

3) Storyline and setting/map through which the game progresses - There is often a goal or set of goals that drives the player through the game. This goal could be concrete, such as obtain 10 coins, or something more abstract, such as make the best farm you can.

4) The primary player will usually choose a unique starting character from a range of options that have different appearances, skills, abilities, actions etc.

Variations

1) Standard RPG (Maplestory, Final Fantasy, etc,)

2) Slice of Life (Harvest Moon)

Overall Design Goal

Our overall design goals echo the fundamentals of Object-Oriented programming that we have been learning throughout the entire semester. These include readability, extensibility, modularity, and reusability. These goals are intertwined for code that is readable and extensible could very easily be modular and reusable as well.

Furthermore, we are hoping to design simple solutions to any issues that we come across. By this we mean that they will encompass the entire scope of the problem while remaining adhering to the principles mentioned earlier, specifically readability and extensibility.

Game Authoring Environment

Game Authoring Team Design Goals

Our primary goals were to create a flexible, functional structure for both displaying and selecting options/preferences on the front end and taking all these information to define, create and add characters, items, events, maps etc, so that our system would be easily extended to additional necessary changes and features. The backend would store the data that is passed from the frontend to the backend with a controller acting as the interface between the backend and the frontend. Since there is a difference in data requirements for views to be rendered in the frontend and data to be read in the Game Player, the backend also acted as a "translator", restructuring data in an agreed manner between the frontend's Data and the Game Engine's Data. Upon invoking the publishing command, the backend restructures the data format to fit requirements set by the Game Engine.

Front-end Design Goals

The Game Authoring Environment's design focused on the following goals:

1) Reusability of Code -

- We wanted to avoid hardcoding our views for every object that needed to be created in the game e.g. Map Tiles, Characters etc. Instead, we aimed to abstract the similarities between these objects in a class inheritance hierarchy and allowed applied the differences in each extension of an object.
- Furthermore, we wanted our views to be generated dynamically based on the various selections made by the user e.g. changing the input fields based on the various interfaces selected to plug into a character.

2) Extensibility -

- We wanted our code to be designed so that new types of Objects could be added without new code added to the program. Furthermore, we wanted to design our code in such a way that allowed us to modify/change the input values required for creation of objects. This would provide us with the ability to have some flexibility in terms of the types of inputs required for the game engine e.g. Our program did not need to change significantly should the game engine require different values.

3) Systemic/Reliable -

- We had agreed earlier on that a change in an object would affect all objects created from the same source e.g. making a particular tile unwalkable would make all tiles created from the same tile unwalkable. We made the decision because design wise, it allowed much more consistency, where we did not require us to keep track of every single object that was created on the map and also allowed for the user to make maps easily e.g. making all trees unwalkable in one click vs clicking through all trees to make each one of them unwalkable.

Back-end Design Goals

An important back end goal is to organize our classes well and structure everything appropriately so that there are clear packages that everything fits well into. There should also be clear relationships among the different packages so that we would know which packages and modules depend on each other and how to carry out their functions. The back end should also be flexible such that adding features would be easy to implement and hastle free. We should make sure that the code does not change drastically when adding new features, for example, when adding a new characteristic to an event, we should only need to recode one part of the code or as little of the code as possible, instead of making drastic changes to multiple aspects of the code in all the different packages. We also plan to carry this out using the MVC pattern for implementation, dividing the game authoring environment into three interconnected parts to separate internal representations of information from the ways that information is presented to or accepted from the user. We plan to have a controller called DataManager, which accepts input from the front end view and converts it to commands for the back end model or view. Also, we need to make out back end easy to publish. We plan to have our back end consist of various definition classes such as CharacterDef, PlayerDef, ItemDef, EventDef, MapDef, and TileDef, which all inherit an abstract publish method, in such a way that they all publish in the same way.

Primary Classes and Methods

Core Architecture

We are using the **MVC Framework** as our core architecture for the Game Authoring Environment, with the following defined as our Model, View and Controller:

1) Model – These are the definition classes that hold the data in a structured way e.g. PlayerDef, MapDef. Each time a view is created, an instance of an Object Definition found in the model is created. The current definitions defined are intended to be general and extensions can be made e.g. CharacterDef (with subclasses NPCDef, PlayerDef), EventDef, ItemDef, MapDef, TileDef

2) View – This mainview in the front-end of the Game Authoring Environment is the WorkSpace Manager, which creates all the Object Managers (views that create objects) and also instantiates a reference to the Controller. Each time a new object is created, it is passed to the DataManager and models of the object are created and stored in the DataManager. The created objects will be reflected in the view. If edits are to be made, the view will query the DataManager for the object definition and the relevant values will be rendered in the

3) Controller – A DataManager object that holds a list of the Object definitions each time a new object is defined in the view. These lists will be classified under various broad categories e.g CharacterList, PlayerList, EventList and this controller will store the most updated data of the current game. When the view wants to edit the data, the view will query the data for a definition to show the user and will update the controller with the relevant data when it is saved.

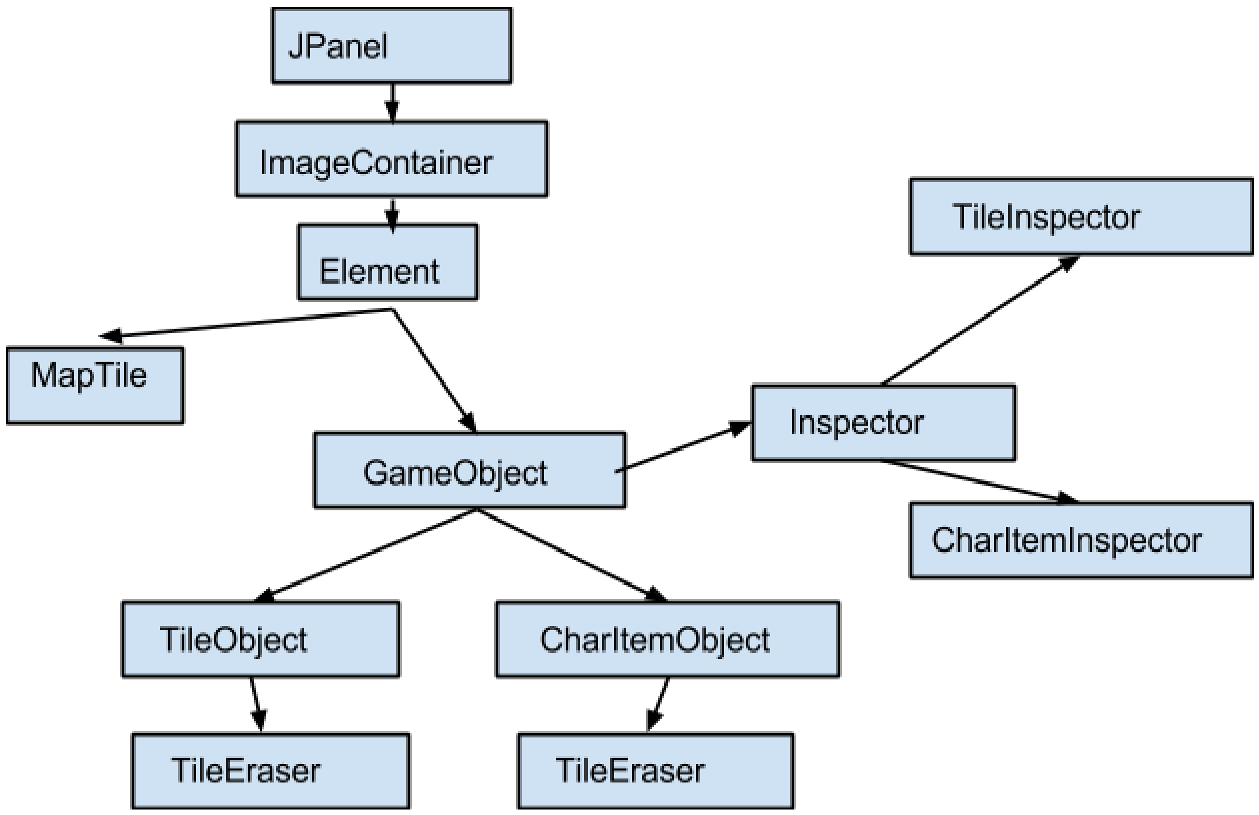
Front-end Modules / Class Structure and API

The following are the main modules in our Front-end:
1) Elements - Containing Main elements and tiles of the maps, including the container views for these elements, the MapCanvas.
2) Error - these are the exceptions raised in the GameAuthoring Environment, when errors occur
3) Workspaces - These were objects that contained “container classes” such as the object managers and main workspace manager etc.
4) Util - These contained methods that was rather general and was used throughout the code without much specification. It also contains a class called DefaultCreator, which creates the tiles in the authoring environment automatically, attaching each tile with a unique id.
5) Factories - Containing our GAInput factories as well as our GameObjectFactories which were used to create objects based on the input types
API and module structure
6) InputFields - These contain the GAInput classes that are used by the frontend to receive and display user inputs. These classes all extend various Swing classes, such as JTextArea or JCheckBox.

Front End Hierarchy

<u>Elements</u>
The smallest elements in the frontend were essentially “tiles” that contained information about the objects that they represented.

The implemented classes are MapTiles, representing a single grid on the Map as well as extensions of GameObject. Instead of storing any values in these Elements, it only stores IDs as well as a reference to the controller. Therefore, the moment there is a need to pull up information of an object, we make a call to the controller with the id of the object to get specific information on the object.



The Elements of the front end program followed the above hierarchy, with 2 main groups of objects, MapTiles and GameObjects.

The base element is an abstract class called Element with an abstract method setTile, requiring many of its subclasses to implement it. This setTile method is especially important as it is the method that allows all Elements to have different behaviors e.g. a TileObject's setTile method takes in a MapTile and sets the MapTile's tileImage to itself and the MapTile's mapTileID to its own mapTileID. . Furthermore, we added the hovering feature to Elements, allowing all objects extending this class to have its borders highlighted in white (for identification purposes) when the user hovers his mouse over the object.

MapTiles were objects that were placed on the map and differ from GameObjects in that they contained map tile objects as well as item objects or character objects. Furthermore, they could store more than 1 item object/character object. MapTiles were also where we implemented the “draggable technology”, allowing the user to paint the tiles by dragging their mouses over the map.

The Inspectors did not extend from their respective GameObject types as it did with erasers because we realized that the requirements for erasers and inspectors were too different. All the erasers had to do was to remove the respective object ID and image from a map tile while the inspector had to communicate to the WorkSpaceManager to dynamically reflect the current inspected object in an object manager - therefore, the inspector needed different references to top level objects to reflect such results, hence the difference in hierarchy.

The MapCanvas contained a method called saveMap that is invoked by its container, that allows each MapCanvas to save all the mapTiles it contains to the controller.

We also realized that many of these objects required a reference to the controller and also a method to set the program’s active tile (in order to paint the active object on the map) and all these objects were present in the Workspace manager. Therefore, in order to provide access to these methods in the Workspace manager, we created interfaces that allowed the Workspace Manager to behave in different ways and passed this Workspace Manager downwards as versions these interfaces - therefore these objects only have access to these methods and not all the methods of the Workspace Manager.

Benefits of Design: It was especially easy to extend from GameObject to create new GameObjects because extended classes came with the ability to add images, setMapTiles as well as have the hover effect for users to identify. It also worked well within the frame work as it made much sense as to where we needed to add code to e.g. setTile method as well as fields in the MapTile class, should it need to take a different type of Object.

Alternatives to Design: We were also considering implementing a GameObject interface between the objects that gave them the required methods and yet implementing all of them in various ways. This allows for greater customization between the objects and classes can contain methods that it needs only as well as the interface methods. Furthermore, it still allowed us to see each object as a “GameObject”, with the required methods for the GameObject to interact with its containers. However, while it allowed for more customization for each GameObject, it still ignored the fact that these objects were connected to each other and should be in a inheritance hierarchy. Furthermore, there were too many common methods that each of these Objects would need to have e.g. hovering effect, having a reference to the same types of objects that it is evident that a hierarchy structure would avoid duplicated code.

<u>Elements API</u>

The elements package has 13 classes but many of the classes have protected and private methods so that we do not expose values to the user that he/she does not need to know. The following methods are the public methods for each class.

<u>ImageContainer (and its extensions)</u>

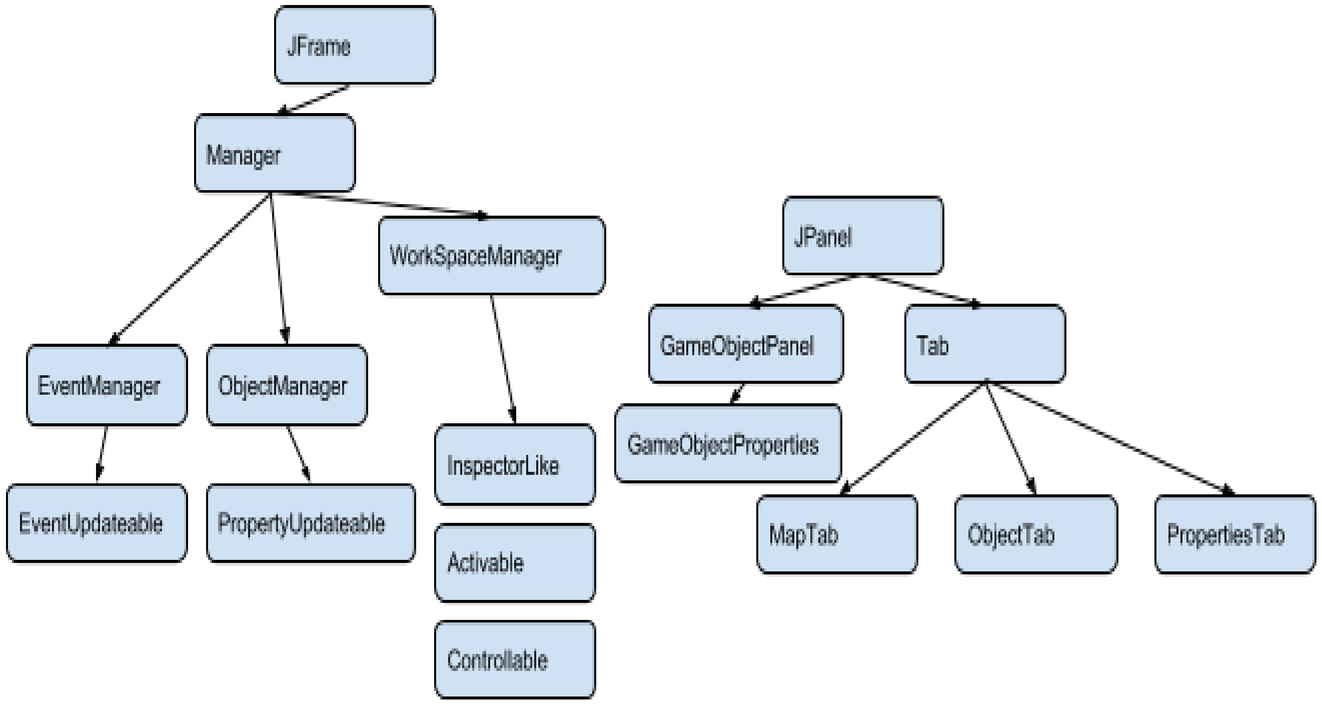
- addObject(String filepath)
- addObject(Image image)

<u>MapCanvas</u>
- getMapID()
- getMapName()
- saveMap(boolean isPremium, String code)

<u>MapTile</u>
- getMapID()
- getMapName()
- getMapTileID()
- getSurfaceID()
- getTileX()
- getTileY()

<u>WorkSpaces</u>

The workspace package included all of the elements that the user would interact with in creating a game. Any of the windows that are used during game creation are created from classes in the workspace package. The main hub of activity is the WorkSpace manager class, which creates all of the necessary elements, tabs, and other managers necessary in game creation.



As you can see from the hierarchy above, there are two main groups in terms of the organization, one stemming from the JFrame Swing component, and one extending from the JPanel Swing component. The former includes the manager superclass, and every manager inherits from it. The manager classes refer to the various windows and panes that are used during game creation. The content inside the managers is determined by the second group, which includes the GameObjectPanel and the Tab superclasses.

The Tab superclass defines the different tabs seen in the manager panes, including the MapTab, ObjectTabs, and PropertiesTabs. The map tab is on the main workspace manager, and displays the actual map to paint the game. The object tab displays the available objects for use in the game. The contents are determined by the actual object type, such as tile, or character or item. The properties tab is used to display the properties of a specific object after it is inspected.

The properties tab specifically is populated by the GameObjectProperties class, which extends GameObjectPanel. These classes read the respective properties from a data file, and dynamically generate the inputs from the data file. The GameObjectPanel is also used by the new object creation tool, as it uses the same attributes for each object type to define a new object. The GameObjectProperties class has an additional update view class, such that when an object is inspected, the properties tab will automatically update with the data regarding that specific object, which is pulled from the back end.

As manager superclass is the base of all of the GUI elements. It is what defines all of the other managers, including WorkSpaceManager, EventManager, and ObjectManager.

WorkSpaceManager is the primary class in the game authoring environment. It is the large window that includes the map on which the game is drawn. As such, the WorkSpaceManager is responsible for creating all of the different panes for various object creation, including character pane, item pane, tile pane, and the event pane. The workspace manager also implements the inspectorlike, activable, and controllable interfaces. This is so that abbreviated versions of the work space manager could be passed to subclasses, such that they had access to only the methods necessary to them, rather than an entire workspace manager. This utilized the interface segregation principle, described by the SOLID design principle.

While EventManager and ObjectManager both inherit from Manager, they are instantiated by the WorkSpaceManager. The ObjectManager is a single class that is responsible for creating the panes to create Characters, Items and Tiles. ObjectManager implements the PropertyUpdateable interface so it can update the properties tab with whatever information is necessary from the inspector. The ObjectManager contains and object tab and a properties tab, which are described earlier.

The EventManager was created separated from the ObjectManager because events were too different from the typical object to be created by the same class. EventManager pulls information from its own data file, and populates it depending on the type of event chosen. Event manager also includes its own eventinspector, so as to properly update the events pane when choosing various objects. It does so by implementing the EventUpdateable interface.

Benefits of Design: We minimized the amount of code we had to write by having ObjectManager apply to almost every object type, except for events. This kept our design extremely flexible, in that we only had to update the data file to change any attributes to an object, rather than the code itself. Moreover, the view is dynamically updated, and can be very easily changed if necessary, with virtually no change to the code. Our use of interface segregation was also good in that it did not give subclasses access to every class in the workspace manager.

Alternatives to Design: The inheritance structure, while it makes sense in terms of the type of panes, did not really seem to fit in practice. For example, even though the EventManager, ObjectManager, and WorkSpaceManager all extended Manager, EventManager and ObjectManager were instantiated by WorkSpaceManager, so it seems more intuitive to have them extend WorkSpaceManager. However, we felt that they were too different to successfully do so.

Workspace Manager API

- createMapTab()
- destroyWorkspace(int i)
- clearWorkSpaces()
- getActiveWorkspace()
- getDefinitionData(String id)
- getDefinitionsOfType(string type)
- getNewLongID()
- getNewShortID()
- inspect(String, List<String>)
- inspect(String, String)
- nameGame(String gameName)
- publish (String name)
- setActiveTile(GameObject gameObject)
- setActiveWorkSpace(int i)
- updateDefinition(String, String, Map<String,String>)

InputFields

We created wrappers for all of the JComponents that we used in our game authoring environment. These wrappers allow us to generate our views very easily and extend the amount of interfaces each object can have easily, regardless of the values that they should contain. We created this wrapper class (by implementing an Interface GAInput) that had only 2 main methods amongst other methods it provides, getValue and setValue - this allowed us to allocate the generation of the views to each class and simply allow us to get the value and set the desired value in the format set out in these classes.

By making all input components a GAInput, all we needed to do was to call a getValue and setValue methods when saving values and rendering these values in views. The power of this implementation shows its best in an Input Field that requires an algorithm - GADecisionTree, which required a BFS to generate its views as well as a BFS to read the decision tree's values. Instead of worrying how to do that, we simply added it as a GAInput and placed the algorithm to generate views in a JPanel, acting as a GAInput.

Benefits of Designs/Alternatives: The first alternative we attempted to do was reflection on native components - how ever, we discovered many problems with reflection - firstly, if we were to do reflection on native components, there were too many different methods that changed views e.g. different methods to set values for JCheckedBox vs JTextArea. Furthermore, the number of arguments to create these objects differred too much and reflection could not be used to create all of these objects. Furthermore, we realized that there were too many differing requirements of each input (e.g. Decision Tree) where an algorithm is required to restructure data from the view and read it in later on. Instead, we abstracted behaviors of these inputs into an interface and created a factory to create all these objects.

<u>API for a GAInput</u>
- getValue(String string)
- setValue(String string)
- setUneditable();
- isEditable();

<u>Overall</u>

Back-end Modules / Class Structure and API
The design of the Back-end includes that of the DataManager and the Definition classes.

The definition classes such as ItemDef and PlayerDef will contain a list of possible keys it may have it its definition e.g. Relationships, Value etc, which will be set based on the map of key value pairs passed to it in its constructor. All definition classes will be extended from an abstract class called Definition which will have an abstract method publish, which will return a JSON object. This is intended for 2 main reasons:

1) To ensure that when the user wants to save the object, the Data-manager can just loop through all the objects that are currently stored and call publish on them to save them
2) To allow different types of Definition classes (e.g. map etc.) to have a different way of publishing their JSON to file

The following are the Modules/Classes with the public API that are intended to be included in our design:
NOTE:
1) All definition classes have a public method publish that will return the JSON representation of that file
2) The rest of the back-end will have no API exposed to the user except for the class of the DataManager

Manager Module
<u>DataManager</u>
NOTE: One design challenge is that each of the types of objects needs to have a create, update and get method while will be saved to a specific list. Therefore we are thinking of a better way to invoke these data to solve them in separate components.

- publish() - returns a JSON file that is the representation of all the objects in the game
- loadJSON() - opens a saved JSON file to create definitions that was previously created

- updateCharacter(Map<String, String>)
- createCharacter(Map<String, String>)
- getCharacter(String id)
- getCharacters();

- updateMap(Map<String, String>)
- createMap(Map<String, String>)
- getMap(String id)
- getMaps();

- updateTile(Map<String, String>)
- createTile(Map<String, String>)
- getTile(String id)
- getTiles();

- updateEvent(Map<String, String>)
- createEvent(Map<String, String>)
- getEvent(String id)
- getEvents();

- updateItem(Map<String, String>)
- createItem(Map<String, String>)
- getItem(String id)
- getItems();

Object Module

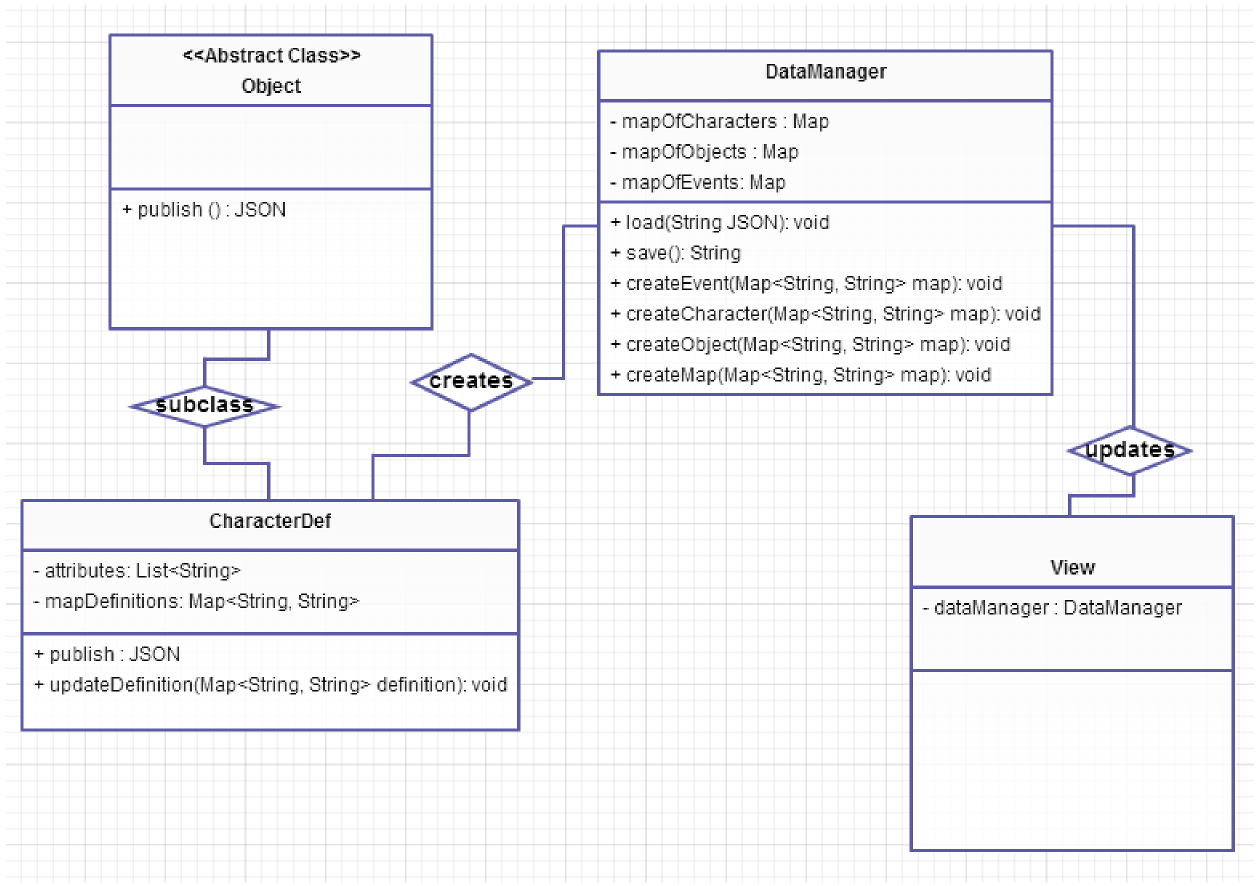
- abstract class for sub-modules

Item Module
- WeaponDef
- ArmourDef
- ToolDef

Character Module
NPCDef
PlayerDef
MonsterDef

MapModule
TileDef
MapDef

UML Diagrams


<<Abstract Class>>
Object
+ publish () : JSON
subclass
creates
DataManager
- mapOfCharacters : Map
- mapOfObjects : Map
- mapOfEvents: Map
+ load(String JSON): void
+ save(): String
+ createEvent(Map<String, String> map): void
+ createCharacter(Map<String, String> map): void
+ createObject(Map<String, String> map): void
+ createMap(Map<String, String> map): void
updates
CharacterDef
- attributes: List<String>
- mapDefinitions: Map<String, String>
+ publish : JSON
+ updateDefinition(Map<String, String> definition): void
View
- dataManager : DataManager

<u>Example Code</u>

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.

The two different games in our genre that we will consider are Maple Story and Harvest Moon. Maple Story is an RPG focused on fighting and leveling up while Harvest Moon is an farm simulation RPG focused on tending to your farm and collecting items. Our goal is to have the save and load processes for different game types to be flexible enough to not require drastic structural changes. Our design is very heavily influenced by composition over inheritance design, thus the keys in our data file (json) are dependent on the interfaces that are implemented in the game engine. For every interface e.g. Fightable, Talkable, there will be certain fields that we know will exist e.g. speech or damage. Therefore, in packaging the data, we will look at the interfaces that are defined/selected for that object and include the key value pairs for that.

Maple Story:
In maple story, the key aspects are map creation, quest creation, character stats building, and monster/item creation and interaction.

Harvest Moon:
The key aspects behind Harvest Moon involve item creation, event creation, and character development/relationships between people.

Character creation: Characters/creatures that we can have relationships with may have a Friendable interface implemented. To reflect that in our data, we will identify that the creatures character have the Friendable option selected and include that as one of the attributes selected. When JSONifying an object, we will recognize that the friendable option was chosen and therefore, we will know that certain friendable options (e.g. type of relationships, benefits of relationships) must have been defined in the map that was originally given that should be saved. Therefore, we will search for that data in the map and subsequently add those key value pairs to the JSON that will be saved

In both cases, such as Map design, where the data should be saved in the same way, we will save the data in key value pairs such as:

```
{
        Name: “Map01”,
        id: “1”,
        MapDesign: “########
                         ###...###
}
```

These definitions will then be held as id → definition maps in the DataManager. Should the view query for the data, the definitions will be reflected. The benefit of this design would mean that we are essentially saving the information in the same way. However, the types of information that is saved for each object changes dynamically according to what attributes the object has, reflecting the philosophy behind the game engine/game itself.

Alternatives to design

For our front end, we made our design choices after working through what it would be like for someone to create a simple game in the authoring environment. Our design has panes of different object types created at the start (character, plater, item, event, map, etc.) with default things created already. When/if the user wants to create a new object of type something (character, item, etc.), the front end will create a new window/pop up for that type of object editor, and once the object is created, it gets passed to the controller. The front end queries the controller for updated objects that are stored/created, and when the user is ready to publish the game, the back end will compress all of the objects/information into a data file (most likely json).

Alternatives to our design include getting rid of a distinguished front end and back end; we could technically store all of the objects and information in the current front end with the panels etc., and it would be okay. We decided against this though, because we felt that the front end and back end design leaves more room for extensibility/flexibility and allows for easier publishing, with a possible “publishable” interface.

Another alternative could be passing everything to the back end and essentially just having action listeners in the front end. This means that whenever the user wants to create anything, we send the request to the back end, the back end creates the pop up window for object creation, and then passes it back to the front end for display. We felt that this was extremely inefficient and created dependencies between the front end and back end that were unnecessary.

Game Data

Game Data Design Goals

- *Reusability*. We strive to make small components that are specialized doing specific tasks and can be reused. For example, the Translator class, the PropertiesFIleReader, and the ExpectingKeys.properties file, which are submitted to the util packagem read json file and process information in a way that is defined by the user in the properties file. Helper classes MediaProcessor writes information to JGame's media table, and the GameDataLibrary stores information of all interfaces and types of objects that are expected in the json file.

- *Extensibility*. It is very easy to update the parsing if the json file format is change, or a new item is added. The only two classes that need to change are the GameDataLibrary (new interfaces or types of objects need to be added to the list of expecting interfaces/objects) and ExpectingKeys.properties. Furthermore, it is very easy extend if the user decides to handle extra processing before arguments are stored or objects are instantiated. For example, if the user decides to write information to the media table when an image path is being read in the json file, he or she can change the expecting type in the properties file to "ToMediaTable", like we do in our project, and in the translator class, add an if check and call the MediaProcessor, which calls the MediaTableWriter.

- *Robustness*. We consider the fact that JGame can only instantiate certain number of JGObjects at one time. Therefore, we separate processing and object creation: the arguments of constructor of GameObject are stored in class GameData, and objects are only instantiate when they are needed so that JGame will not crash if too many objects are being made. Game Data also holds information of maps and tiles that will be accessed when switching game maps and states. The same principle is applied to the Translator class: the user can choose to only obtain the arguments, and can call instantiateProcessedObjects to create objects.

Benefits of Design

- *Follows the Open-Close Principle*. The Translator, PropertiesFileReader, MediaProcessor, and MediaTableWriter are the core of our data processing. Helper classes such as GameObjectFactory/Manager, MapsAndTilesFactory/Manager are added for the storing of objects and are useful for the engine to access objects, but the four classes are only used within the Game Data package. Furthermore, the GameObjectFactory and MapsAndTilesFactory both have only private and protected methods, which means in order to create game objects or maps, the user has to go through GameObjectManager and MapsAndTilesManager. This encloses the creation of objects.

- *Our design is very reusable and extensible*. Different helper classes that hold different types of information can be easily added. For example, when we made the decision to potentially include game intro, the GameIntroParser and GameIntro are added easily, and only the properties file needs to be updated for the expecting information in the json file.

Alternatives to Design

- *Separate parsing classes according different parsing methods rather than the type of information that we are parsing*. Although we do not have repeated code in the factory and manager classes for different types of information, we realize that they do very similar tasks: use the translator to process information and hold it. If we had more time, we would implement two separate factories/manager: one for GameObjects and GameIntro, and the other one is for Maps/Tiles. Maps and tiles require more processing and interact with JGame directly through the setTiles and setTilesSetting methods, and therefore should be separate from GameObjects.

- All methods in PropertiesFileReader class should be changed to protected because this class is a helper class of Translator

Game Data Module/API

Translator (from Tara's util package): Takes in a json file, looks up properties file for items to expect and their types, and output the processed data into a list of arguments and they're ready to be instantiated whenever the user wants by calling instantiateProcessedObjects
- List<Object[]> translateSpecifiedItem(String itemKey, String jsonFilePath, String propertiesFilePath)
- List<Object[]> translateJsonFile(String jsonFilePath, String propertiesFilePath)
- Object instantiateProcessedObject(Object[] args, String packagePathPlusClassName)
- Map<String, String> getUniqueIDSymbolMap()

PropertiesFileReader (from Tara's util package): Helper class of the Translator class; reads the properties file and looks for items and their types in the user-defined properties file
- String getSplitPattern(): Get the split pattern the user defined in the properties file
- List<String> getSubObjects(String item): Return the list of subobjects in the properties file for the given item

ExpectingKeys.properties (from Tara's util package): Properties file to define keys to expect and their types

MediaProcessor: A helper class of Translator: write necessary information to the media table as well as assign collision IDs

- String processMedia(String type, JSONObject subObject): write information of the json subobject to media table
- void addCreatedTile(Object[] tileArgs): this method is used to assign collision IDs to tiles. It adds the input tile to the list of tiles have already been created. This method should be protected instead of public because it is only used by the Translator class and exposes the implementation of assigning collision IDs

MediaTableWriter (in src/util): A class that takes in all the information required to make an entry in a media table and, if an entry with that name doesn't already exist within it, writes it as a new line in a designated text file

- void write (): after checking the text file, use all the Strings and the int passed in by the constructor to make a new entry using a FileWriter

DataConstants: an interface that holds static final constants and tokens that can be accessed from anywhere in the project

GameData: Class that holds the input arguments of ONE GameObject constructor

- String getName()
- String getType()
- double getXPos()
- double getYPos()
- void setXPos()
- void setYPos()
- Object[] getData(): get all input arguments for class GameObject
- String getMapUniqueID90
- String getGFXName(): get the graphics name in JGame's media table

PlayerCharacterData: Class that extends GameData to store extra information of a single PlayerCharacter such as stats

- List<InventoryItem> exposePlayerItems()
- PlayerStats getPlayerStats()
- List<Object> setPlayerStats()

GameIntro: Class that stores information of introduction information of the game for the splash screen

- String getBattleBackgroundGFX(): get the background image name for battles in the media table
- String getInstruction()
- String getGameName()

GameIntroParser: Class that parsers game intro informaton such as instructions and game name

- GameIntro parserIntroInformation(): use Translator to parse game intro information

MapsAndTilesManager: Class that keeps track of the maps including the creation and creating actual JGame tiles in the game
- void createMaps()
- GameMap getMapByUniqueID(String uniqueID)
- int[] getMapDimensions(String uniqueID)

GameObjectDataParser: Class that takes in a json file and return a list or arguments that are saved for future GameObject creation
- GameDataManager createData(): create data and store them in GameDataManager
- PartyObject createPartyObject(): create a PartyObject that are consists of all PlayerCharacters in this game

GameDataManager: Stores all the data of the constructor inputs for GameObjects
- void createObjectsInMap(GameMap map)
- void saveObjectData(GameObject object)
- GameObjectManager getGameObjectManager()
- String getFirstMapID(): get the map ID of the map that the game should start with

GameObjectManager: Class that keeps track of the GameObjects that are load from the file
- List<GameObject> getActiveGameObjects()
- List<Object> enable (String type, Map<String, List<Object>> propertyInputMap, Map<String, String> propertyImplementationMap): Return the list of CharacterLike objects (abilities) to enable the character to perform certain tasks in the game
- void setPlayerCharacterDisplay (PlayerEngine playerEngine): Set the display (JGEngine) class of player character
- void setEventManager(GameEventManagerInterface myEventManager)
- void addActiveObject (GameObject gameObject)
- void destroyActiveObjects(): when going to a new state, destroy active and save them as GameData

GameObjectManagerInterface: Interfact of GameObjectManager to restrict access
- List<Object> enable (String type, Map<String, List<Object>> propertyInputMap, Map<String, String> propertyImplementationMap): Return the list of CharacterLike objects (abilities) to enable the character to perform certain tasks in the game
- void setPlayerCharacterDisplay (PlayerEngine playerEngine): Set the display (JGEngine) class of player character

GameMap: Class that store information of a map
- int[] getTIlePosition()
- String toSymbolRepresentation(): return the symbol representation that will be the input of setTiles method in the engine

- List<String> getNotWalkableSymbols: get the list of symbols of tiles that are not walkable
- void addTile(GameTile tile)
- List<String> getStringRepresentation(): return the list of string representation that are consists of unique IDs of tiles in the map
- boolean hasThisTile(GameTile tile)
- List<GameTile> getTiles()
- String getUniqueID()
- void turnOffSubscriptions(): make this map not a user-subscribed map
- boolean getSubscriptionBoolean()
- String getCode(): get the subscription code
- String getName()
- int getWidth()
- int getHeight

GameTile: Class for tile objects which makes up a map
- void setMap(GameMap map)
- void addToGame(JGEngine engine)
- GameMap getMap()
- boolean isWalkable()
- String getUniqueID()
- String getSymbol()
- void setSymbol (String symbol)

GameDataLibrary: Another helper class of parsing objects. It holds all the abilities and their implementations. Again, the methods in this class should be protected instead of public because they are only used in the game data package.
- void init(): Initialize property library, create all the lists and maps, and execute necessary stuff to setup
- Object createPropertyObject (String propertyName, String implementationName, List<Object> inputs): instantiate the implementation of an interface (e.g. instantiate a Talk object for the interface Talkable)
- List<String> getFieldsForGameObjects (): Return an unmodifiable list of the fields in the properties file (name, x, y, uniqueid, etc)
- List<String> getFieldsForMaps(): Return an unmodifiable list of the fields for maps
- List<String> getFieldsForTiles(): Return an unmodifiable list of the fields for tiles
- List<String> getAbilitiesForGameObjects(): Return an unmodifiable list of all the abilities a GameObject can have
- String getProperty (String implementation): Get the name of the interface from implementation

QuestData: Class that stores quest information
- boolean checkQuestCompletion(String, String)
- String getQuestID()

- String getQuestDescription(String questType, String objectID)

<u>**Game Engine Environment**</u>

Game Engine Team Design Goals

- Process user commands received from Game Player (frontend): the HUDController class
- Game logic

Benefits of Design

- *Extensibility*. We strive to make it as easy as possible to add new features. We are using composition over inheritance as the principle behind our design: GameObjects, which extends JGObjects, can have different "properties" or "abilities" that are represented by interfaces. For each interface, there are two main implementations (although it is easy to add more): can and cannot. For example, the interface Talkable has implementations Talk and CannotTalk. Therefore, to add a new feature, such as Fightable, only two classes Fight and CannotFight need to be added.

Alternatives to Design

- The implementation of "property" interfaces should be instanted only once and be stored as private instance variables in the GameObject class, instead of instantiate it in the individual methods such as talk(). For example, the implementation Talk of the interface Talkable should be stored as an instance variable myTalk in the GameObject class. This is because it allows more processing of the myTalk object in other methods.
- The EventManager class would benefit from being split into two classes: EventAggregator and EventInvoker. As it currently exists, it violates the single responsibility principle. This decision was largely made because since we have a queue of events, the job of invoking the the events is somewhat complicated. The EventInvoker could serve as a mediator that listens to the event aggregator. Then, it would run it's logic and process the events (aka the "management" part) of the event aggregator
- While the event invocation is very extensible and flexible, adding a new event does require and understanding of jgame and how the different objects interact, so there are some dependencies here. Passing arguments to the events requires understanding what the constructor of the new event that you've created looks like, and that may lead to difficulty in debugging and errors when first creating a new event. Here, there may have been some sacrifice of readability for the sake of high flexibility. One alternative would have been further abstracting away event notification within the hit method of GameObjects. Instead of explicitly stating that you're adding an event, you could simply send a notification to the EventAggregator (by implementing the EventAggregator Interface) and it would parse the details based on "who" had notified it of a new event. This would be a slight change to the existing design, because the notification in the

GameObject currently passes the arguments for the event's construction to the EventFactory.

Primary classes and methods:

- Interfaces classify objects and therefore defines the player's interaction with them:
  - All interfaces related to characters are included in the interface CharacterLike:
    - Talkable: the ability for a GameObject to talk
    - Fightable: the ability for a GameObject to fight
    - Questable: the ability for a GameObject to hold quests
  - All interfaces related to items are included in the interface ItemLike
    - Pickupable: the ability for a GameObject to be picked up by the Player Character
    - Equipable: the ability for a GameObject to be equipped by the Player Character
  - All interfaces related to transitions are included in the interface DoorLike
    - Transitionable: when the Player Character collides with a transitionable object, the object will take the Player Character to another position of another map

- Class GameObject, which extends JGObject, includes all objects in the game except map tiles
  - Class GameCharacter extends GameObject, and class PlayerCharacter and NonPlayerCharacter further extends class GameCharacter. The main difference between PlayerCharacter and NonPlayerCharacter is that NonPlayerCharacter don't have controls; there should be lots of overlap in terms of graphics/animation, movement, location information, reaction to obstacles, use of Events (mentioned below)
    - class PlayerCharacter: one player character
      - move() method that listens for keys
      - Instance variables for stats and skill tree
    - class NonPlayerCharacter: not able to be controlled by the player. This encompasses both enemies and friendly characters
  - Class MapItem extends GameObject, and contains a map of interfaces that are implemented as well as the information needed to do so
    - All the items in our games implement Pickupable - this means that when the item is touched while in the map view, the item itself disappears, and a new kind of item class called an InventoryItem appears in the inventory. This is achieved through an event prompted upon the Party object's collision with the MapItem, and the InventoryItem is able to be created using the data stored in the

Pickupable interface. This event destroys the original MapItem after gleaning the information needed.

  - The InventoryItem does not extend JGObject or GameObject, it is merely a vessel for information and has no physical properties in the game; it contains its own name, its type (we distinguished between three types of items in our games: Weapon, Single Use Combat, and Quest Item), and the list of the effects it has on the list of player stats when utilized

Party

In an RPG, a party (PartyObject) basically is all the characters the player controls or uses. Naturally, it made sense for us to make use of this concept. The party is useful because it provides a single place to maintain or change attributes of the characters, and is also a convenient place to put other pertinent information like inventory items and quests. One can also specify which character should be used to represent the party on the map. Note that because we use party as the map character, it is also a JGObject. We use PartyObject to implement move and hit. This is useful because if a collision is detected, the PartyObject is immediately available from the hit method, and can be modified/passed along to perform the appropriate reaction.

Party API

void setActivePlayer(String name) - sets active player character by name. If name doesn't exist in party, nothing happens.

void getPlayerStats() - returns the active character's stats object

List<InventoryItem> getItems() - returns items held by the party

Map<String, PlayerStats> getAllPlayerStats() - returns all players stats

boolean isCharacterName(String name) - returns true if name is a character's name, false otherwise.

List<PlayerCharacterData> getAlivePlayerCharacters() - returns a list of PlayerCharacterData

List<String> getQuestDescriptions() - returns quest descriptions

void incrementStat(String stat, int incrementBy) - increments a stat by a given amount

void addToInventory(InventoryItem newInventoryItem) - adds a inventory item to the party void addQuest(QuestData quest) - adds a quest to the party

void clearQuest(String questID) - removes a quest from the party

checkQuestCompletion(String questType, String objectID) - checks if a quest is completed

Events

In our game, we implemented a combination of several different design patterns throughout the course of our event listening, creation, and management. For event listening, we utilized a modified observer pattern. Our main issue with the observer pattern was that it works best with a one to many dependency, meaning that when an object changes state, it notifies its dependents. However, here, we have an undefined number of objects to listen to so we utilized an event aggregator. With the traditional observer pattern, in an extensive game, we can have hundreds of items and NPCs, all of which would need listeners. Instead, our design allows various objects to notify the event aggregator when they think an event needs to be added to the queue. Our eventmanager allows for multiple publishers of events, and multiple receivers. This makes it so that our GameObjects do not need direct reference to game state, the party object, or other objects within the game.

Once a notification is sent to the event manager, it notifies the factory to create an event based on who notified it, and the attached message that they sent. This factory utilizes the factory pattern, constructing the correct object based on the message and an array of arguments. To implement this, we utilized the Reflection util package. In terms of the actual event structure, events hold various information about the game's state at time of it's creation. Here, we utilized the Command Pattern, where the GameEvent is the command and the individual types of events were all concrete commands. We encapsulated the invoker within the GameEventManager (will be further discussed below), and here we have a large number of receivers, based upon the type of event. the In every frame, the engine runs all the events that are in the queue by calling each event's execute method. This queue design allows for a faster and smoother running game, as the engine does not stop every time the event aggregator receives an event that it is listening for.

Quest

Quest is also another GameEvent. Quests controls a lot of things in our implementation, including setting the buttons on the ActionPanel for the user to issue commands, parsing commands, and controlling the quest logic and progression. Our simple version of Quest implements these things.

The QuestEvent checks for whether the quest has been accepted by the player or not and if it is accepted, adds the quest to the player's quest panel through the HUDController.

Quest API - same as GameEvent

Event API
Event Hierarchy

- GameEvent Interface
  String execute(PlayerEngine engine)
  - QuestEvent
  - TransitionGameEvent
  - ItemGameEvent
    - ItemPickUpEvent
  - NPCGameEvent
    - NPCTalkEvent
    - NPCStartBattleEvent
    - NPCEndBattleEvent
    - NPCTalkEvent

Event Management API
GameEventManagerInterface:

    void add(Object[] args)

GameEventManager:

    GameEvent next()
    void add()
    int doNext()
    void runAllEvents()

GameEventFactory:

    GameEvent makeEvent()

States

In the engine, we attempted to make use of JGame's existing functions and improve upon them. One way we did this was with states. While JGame allows for states, it requires pseudo-hardcoding them within the JGEngine, making it somewhat inflexible.

We implemented our own version of states to increase flexibility. The interface for states (GameStateInterface) requires two methods be implemented, paintFrameGameState and doFrameGameState, which are analogous to similarly named methods in JGame. Instead of hardcoding the names of each state's methods within JGame, we simply have the Engine use the default game state and call each states methods, swapping out GameStates as necessary, like if Events are called to switch to a battle state. This allows us to easily add GameState implementations without cluttering up the JGEngine. Each implementation is encapsulated and handles the work the Engine needs to do. The downside is that the states and the Engine become very coupled, as GameStates need a reference to the JGEngine to do anything useful. This is not that much different from how they were initially in JGame, so it is not a huge issue, and if there is too much access, it is simple enough to create an interface to limit method access from the state.

Battle

Battle is just another GameState. Battle controls a lot of things in our implementation, including setting the buttons on the ActionPanel for the user to issue commands, parsing commands, and controlling the battle logic. Our simple version of Battle implements these things, however, other versions of battle state could use their own implementations.

During the BattleState, the main classes involved are BattleState, PlayerStats, PartyObject, PlayerCharacterData, PlayerEngine and the HUDContoller. A new battle state is triggered off during a collision between the player character and a NPC. A new battle event is created and called, which creates a new BattleState object taking in as parameters the party and the NPC involved. The same sequence of events/methods for any game state is then carried out. The active objects are destroyed and the screen cleared before a new background is set, and the images of the objects then painted in the standard 'battle positions', with their stats displayed next to them. BattleState interacts with the HUDController through the engine, which in turn interacts with the ActionPanel and displays the fight and item options for the user to choose. In doFrame, the engine constantly looks out for the (human) player's inputs which is sent over from the action panel. Depending on whether fight or item is chosen, the corresponding method will be called which takes in both PartyObject and the NPC, and alters their stats based on values set by the game author. The changes in these stats and attributes will be displayed in the action panel as the battle progresses. The battle state is programmed to be turn based, hence after the player attacks, the NPC will attack, and this continues until the battle ends. When the battle ends, a BattleExitEvent is called, and the game state is changed back to the map state (or the previous state), and the display repainted back to its previous state.

One thing I think we could have done is refactor our current BattleState, perhaps even make a hierarchy for different types of battle. However, I think the concept makes sense, and it would be possible with how states currently work to have multiple types of battles in the same game.

BattleState API - same as GameState

GameState API

doFrameGameState() - called by the JGEngine (PlayerEngine) every frame to do the game logic instead of doing it within the engine itself.

paintFrameGameState() - called by PlayerEngine every frame to paint anything else necessary in the frame.

Other useful GameState public methods

MapState

- getActiveMap() - returns the active GameMap used by the MapState
- setGameMap(GameMap map) - set the GameMap to be used.

HUDController (Controller)

The HUDController functions as the tie between the GameEngine and the ActionPanel. The controller contains the appropriate methods to call from the ActionPanel class according to the parameters that each game event will give. The HUDController also contains necessary information that is gathered from the ActionPanel, like user inputs, and necessary information that is sent from the game events that is to be put on queue for display.

HUDController API

- updateActionPanelStats(PartyObject party) - uses the PartyObject to get stats and display the stats and repaint the stats panel
- updateActionPanelStats(PlayerStats stats) - updates the stats panel for stats using the PlayerStats class directly
- updateActionPanelDialouge(String DialogueString) - displays text in the ActionPanel based off of a string
- updateActionPanelDialouge(List<String> actions) - displays buttons for user to click on based off of a list of strings
- updateActionPanelInventory(List<InventoryItem> items) - displays buttons for user to click on in the inventory panel based of a list of InventoryItems
-

Game Player (Frontend)

The GamePlayer a JFrame that contains the HUDController, the ActionPanel, and the PlayerEngine. It handles the initialization of the engine as well as the MenuBar for the user to select a JSON file to load into the GameEngine. The goal of the GamePlayer is to setup and connect all the moving parts that it contains once a JSON file is loaded. This means that the HUDController, ActionPanel and the PlayerEngine must be initialized, and the the PlayerEngine must have access to the ActionPanel through the HUDController.

ActionPanel

The ActionPanel is a JPanel that contains tabs for the different aspects of the game. There is a notification tab, that receives information from the HUDController to update the notification subpanel. The UserInput panel is another subpanel in the notifications panel that will create buttons for a variety of possible options for responses for the user. The stats panel show the stats of the main player, the inventory shows the current items that the user has and will use those items once clicked upon. The party panel shows the multiple players that could be in your working RPG group and shows the respective stats as well. The ActionPanel utilizes the ActionListener that that will receive user input based off of the button and push it into a queue for the HUDController to process.

PlayerEngine

The PlayerEngine extends the GameEngine and has all the methods that the GameEngine in JGame has. This is where the JSON gets parsed through the MapDataParser and all the Characters get created and placed on the map. The PlayerEngine is a JPanel that displays our game and reflects all the user interactions with the PlayerCharacters, NPCCharacters, Items and etc. in our RPG. The PlayerEngine also houses the PartyObject that defines the party of players in the game. The PlayerEngine also detects collisions per frame and executes GameEvents based off of the collisions.

Files

- We'll be using the JSON filetype to store and retrieve data. There exist some useful libraries that are easy to use and parse JSON, and writing the file simple as well. We will make use of the nested structure of JSON. We will specify a schema to standardize the syntax.
  Example for a NPC:

```
{
         "uniqueid":"2075647454",
         "abilities":"Talkable",
         "imagepath":"media\/resources\/Character\/Character4.png",
         "name":"Amy",
         "map":"460759282",
            "Talkable":"Hey     Zan!::Where     are     you:1::Yay     you:2@@I     know
right::Argh:3@@Boo::FLYYYY:4@@Seee MEEEE@@Flyer",
         "type":"NonPlayerCharacter",
         "y":"3",
         "x":"8"
     },
```

Division of Work

Timothy Shih and Tze Kang Ng - GameState Package, battling, overall design, debugging Engine control flow.
Tara Gu - GameData Package, properties package, propertiesimplementation package, top level gameengine package, GameObject hierarchy and the implementation of composition over inheritance, TimeDecorator, TimeOfDay, and Season
Trey Bagley - MediaTableWriter, MapItems, implementation of class Pickup and class Equip
Zanele Munyikwa - Event Aggregation and Management, Quests, Integration of Events into battle state, BackEnd of DialogueTree and DialogueNode, TimeDecorator, TimeOfDay, and Season, Debugging Engine Control
Ashley Qian - GamePlayer, PlayerEngine, ActionPanel, HUDController, overall design planning stages and debugging final features
Amy Wang - GameStates, PartyEvents, Events design, Entire Party integeration