Pokémon Text Adventure

Report #2

Team 3:
Rahil Sagarwala
Gary Ray
Luis Siavichay
Chad Mendenhall
October 15, 2019

Individual Contributions

`

Rahil Sagarwala

1. Shared links to Group about Github/Git tips
2. Finished startScreen class
3. Proposed idea of dual language feature integration
4. Finished integration of dual language feature
5. Finished integration of adjusting text speed and text size
6. Finished integration of direct swapping options within game
7. Started superclass and subclasses for TrainerSuper, Rival, and Trainer
8. Proposed new group strategy (division of labor in pairs and rotation)
9. Notified Group about keeping Report 1 up to date
10. Gave group tasks for coding
11. Notified group of errors in code and when updated repository
12. Added and edited interaction diagram descriptions
13. Finished general class diagram layout
14. Added feedback website to program
15. Integrated dynamic font size setting in game (partially working) as opposed to manual settings (fully working)
16. Added data type, signature, and general descriptions for class diagrams
17. Added logo icon to game
18. Completed Hardware Requirements
19. Created own logo and added both logos to game
20. Added Project Management Milestones and Use Cases
21. Finished Algorithms and DataStructures
22. Finished User Interface and Design Implementation
23. Added to Design of Tests
24. Changed name of game
25. Revised proposal

Gary Ray

1. Created the Sequence Diagram
2. Created the Communications Diagram
3. Completed the Persistent Data Storage Section
4. Discussed the need for Network Protocol Section
5. Contributed info to the Coverage of Test Cases Section
6. Contributed info to the Integration Test Strategy
7. Adjusted some grammatical errors throughout report
8. Added references

Chad Mendenhall

1. Proposed using Maven for build tool
2. Battle - Timing Diagram
3. Began researching using maven for build tool
4. Adjusted some report layout mistakes

Luis Siavichay

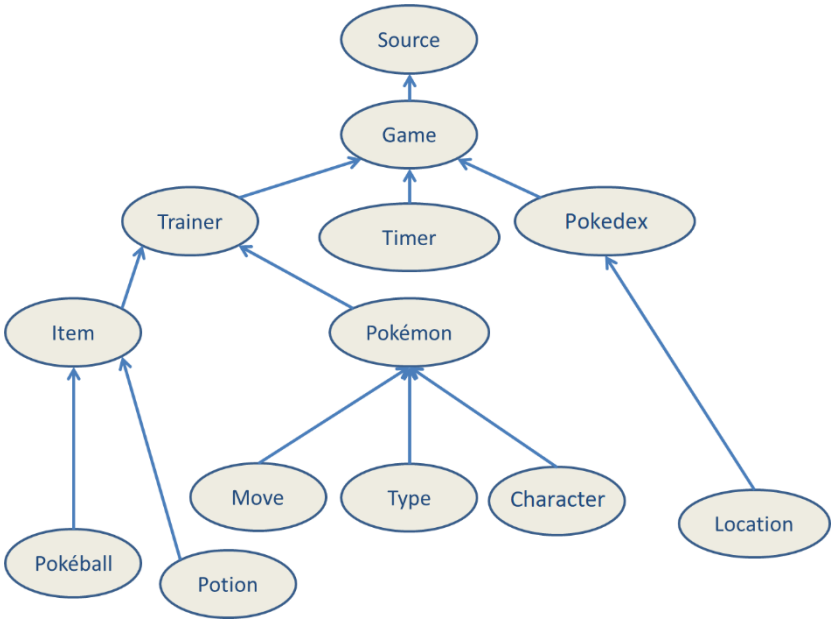
1. Analyzing Ideas and Formats for Report 2 – Part One
2. Created Cover Page and Redesign Logo for App use on smart devices and game consoles
3. Added Individual Contributions Breakdown
4. Added Table of Contents
5. Added Section 1: Interaction Diagrams and included examples and analysis
6. Added Project Management and Plan of Work
7. Added Extra References
8. Enhanced Page Layout, Adjusted Tables, and Headings for Uniformity
9. Edited Report for Conversion to PDF
10. Created Simple and Complex Interaction diagrams with details
11. Included emphasis of system sequence, communication, and timing diagrams
12. Added resized and updated logo icon in PNG and ICO formats for game
13. Created System Architecture and System Design section with the tasks needed
14. Included UML package diagram
15. Added general descriptions for Architectural Styles, Identifying Subsystems, Mapping Subsystems to Hardware

Table of Contents

Section 1: Interaction Diagrams

As in the term interaction, it is clear that the interaction diagram is used to describe some type of interactions among the different elements in the models. This interaction is a part of dynamic behavior of the system which is represented in UML by diagrams such as sequence diagram, communication diagram, and timing diagram.

Simple interaction diagram:

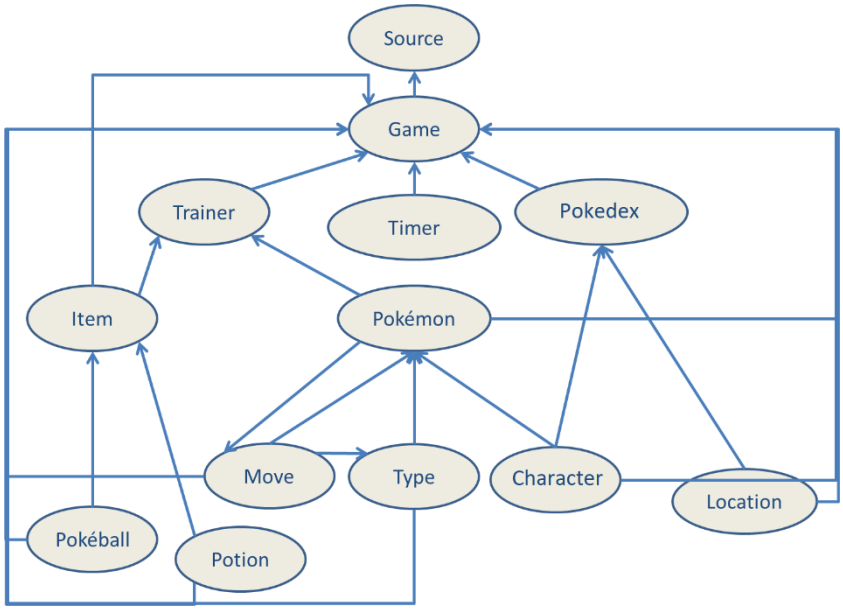


The source code contains the files needed to run the game. The game displays text on the screen by using a Timer for a typewriter effect similar to the original games. The trainer which includes all NPC's who can battle and the player's character, has access to the Pokémon objects along with items objects. The most basic of items are Pokéballs and potions. Pokéballs can be used to catch Pokémon which will be added to the Character's object. Potions can heal Pokémon. A Pokémon can initiate an attack object with specific properties and types. Pokémon have their own types as well. Within Pokémon's data, there is a summary of its information which is added to the Pokedex. The Pokedex also contains information about the Pokémon's location.

Alternatively what we could have done is separate location and give it a connection all of the other elements as locations can include Pokémon, trainers, items, the player character, dialogue which contains timer, and the Pokédex which is accessed from the menu available at any location. The reason why we have kept it only connected to Pokédex is because location is not necessarily an object with characteristics inside the game. The location object only dictates where things are located and thus it has no other purpose. It is sequential in the sense that the player needs to know where certain Pokémon are located, however anymore would be outside the scope of this game on a visual level as the purpose is not for the game to be visual.

Another alternative route we could have taken was to combine Trainer and player character into one as player character is essentially a trainer. We decided to settle for a separation with this diagram in order to stay consistent with which objects will eventually contain different methods and variables. That is the only reason why we consider both objects to be slightly different although both are part of the same superclass.

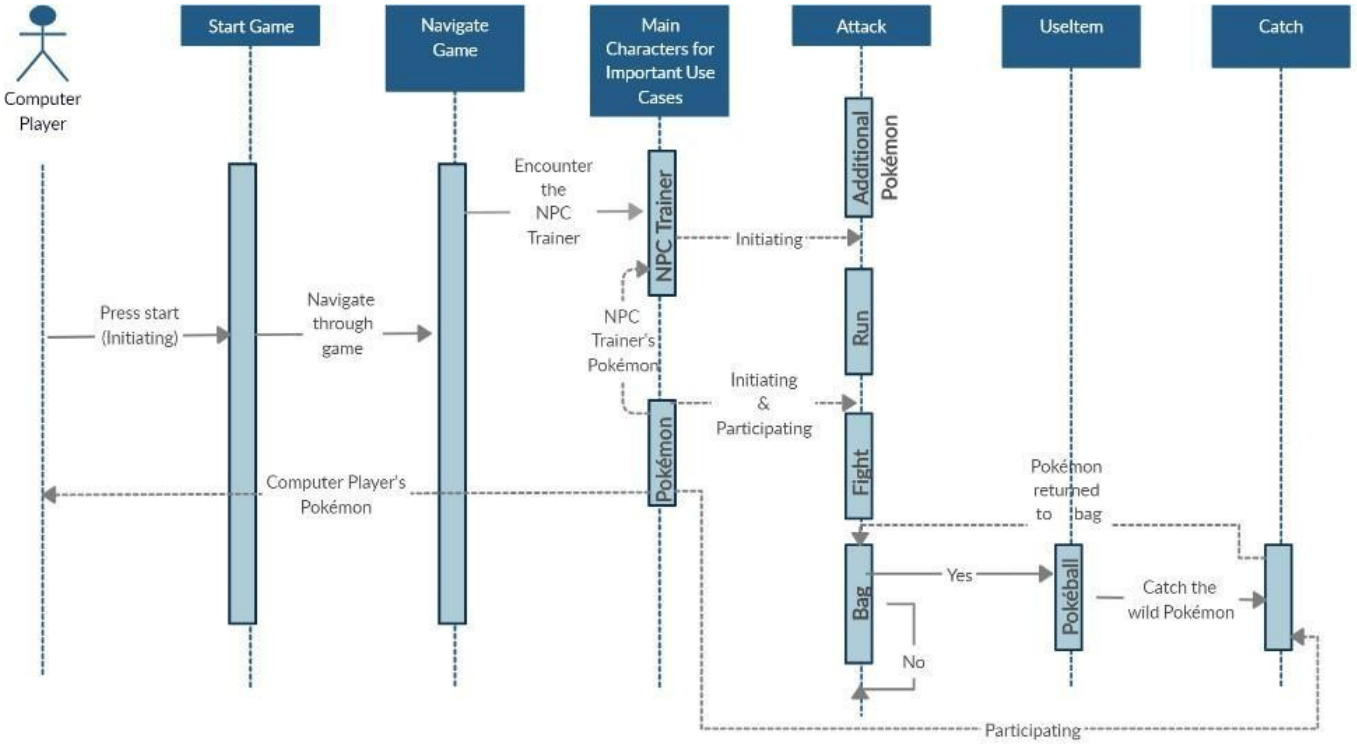
Complex interaction diagram:



The interaction of objects in a complex manner can be seen by the central buble Pokémon. Pokémon can be initiating and participating actors and thus they serve a dual purpose. For this reason, it is the central object that contains attack objects, can utilize these attacks, has a specific type that influences the turn of events in battle, can be obtained within a Trainer object (including player character), located in specific locations, caught by using Pokéballs, and information extracted into a summary of that Pokémon in the Pokédex. Trainers (including the player character) have access to items of various types and Pokémon. The user of the game which is also the player character, but not considered as an object, can visit various locations, access the Pokédex, items, and Pokémon. The Timer class is what is in charge of displaying text in a typewriter effect similar to the actual games and is connected to the Game which consists of all elements including dialogue and GUI.

Alternatively, we could have not allowed everything to point towards Game because of the level of complexity. When starting the code however it became clear that this design is inevitable as every object must be passed on from one screen to the next. This would be the case for being able to save/load the game along with simply keeping an instance of an object throughout the game. Perhaps the level of structure could be made hierarchical, although this would require being able to map out specific objects and where they belong in the scope of a large game.

System Sequence Diagram



The system sequence diagram emphasizes the time sequence of messages.

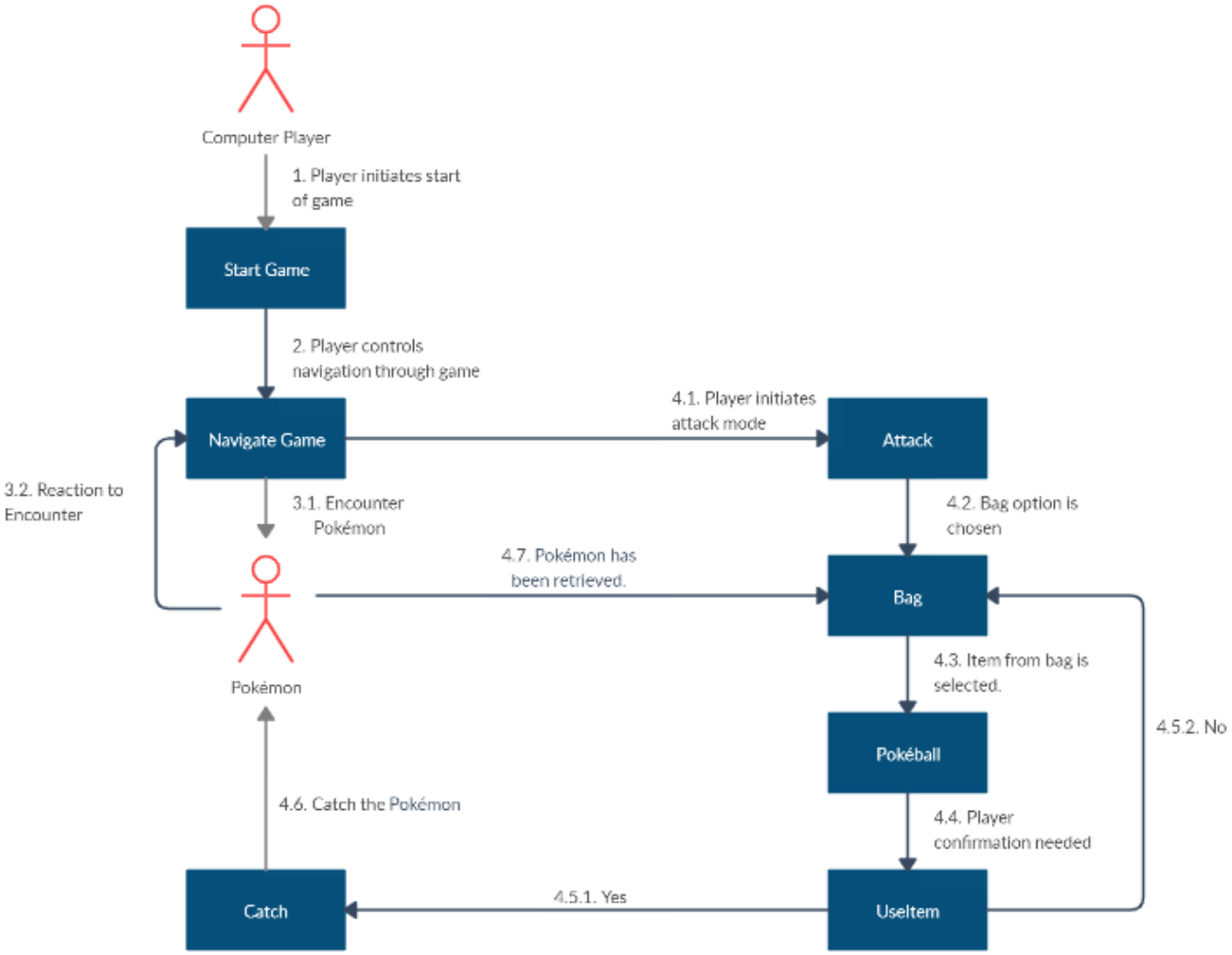
The system sequence diagram covers 3 of the most important use cases for Pokemon Text-Audio game. The attack, UseItem, and Catch use cases are represented in the diagram. The player starts a game and navigates multiple screens until a battle sequence is encountered. This can happen in two ways: 1) A random Pokémon initiates a battle or 2) An NPC trainer built into the game initiates a battle. The catch use case can only be initiated if it is a random wild Pokémon battle. Likewise, the use case run can only be initiated when battling against a random wild Pokémon. When battling against an NPC trainer or a wild Pokémon, use of items except for a Pokéball item are permitted. The fight sequence is turn based until one Pokémon's HP is 0. This goes on until there are no Pokémon left on a trainer and immediately ends when a wild Pokémon's HP is 0. The catching mechanism on a wild Pokémon, happens through using the Pokéball item. Subsequently, if a Pokémon is caught (random), it is added to the player object's Pokémon array either in the PC or the box.

An alternate solution would consist of separating Pokémon as initiating and participating actors to allow for separate sequence diagrams. We chose one however, because of the nature of the use cases which fluctuate in similarity per situation. When a Pokémon is an initiating actor, it is a wild Pokémon battle and when a Pokémon is a participating actor, it is a trainer battle. Additionally for the sake of simplicity, we excluded the attack class in the sequence diagram. The attack class consists of the various attack moves that a Pokémon can use. In essence a Pokémon contains attack objects but the attack objects are not the primary objects in question. The link between a Trainer's items and Pokémon (including the player) is in the same way a question of which class is the main class that can hold various other objects

Communication Diagram

Sequence diagrams are meticulously related to communication diagrams as they are both interaction diagrams. Many modelers often do not know when to use sequence versus communication diagrams. Communication diagrams emphasize the structural organization of the objects that send and receive messages.

For example:

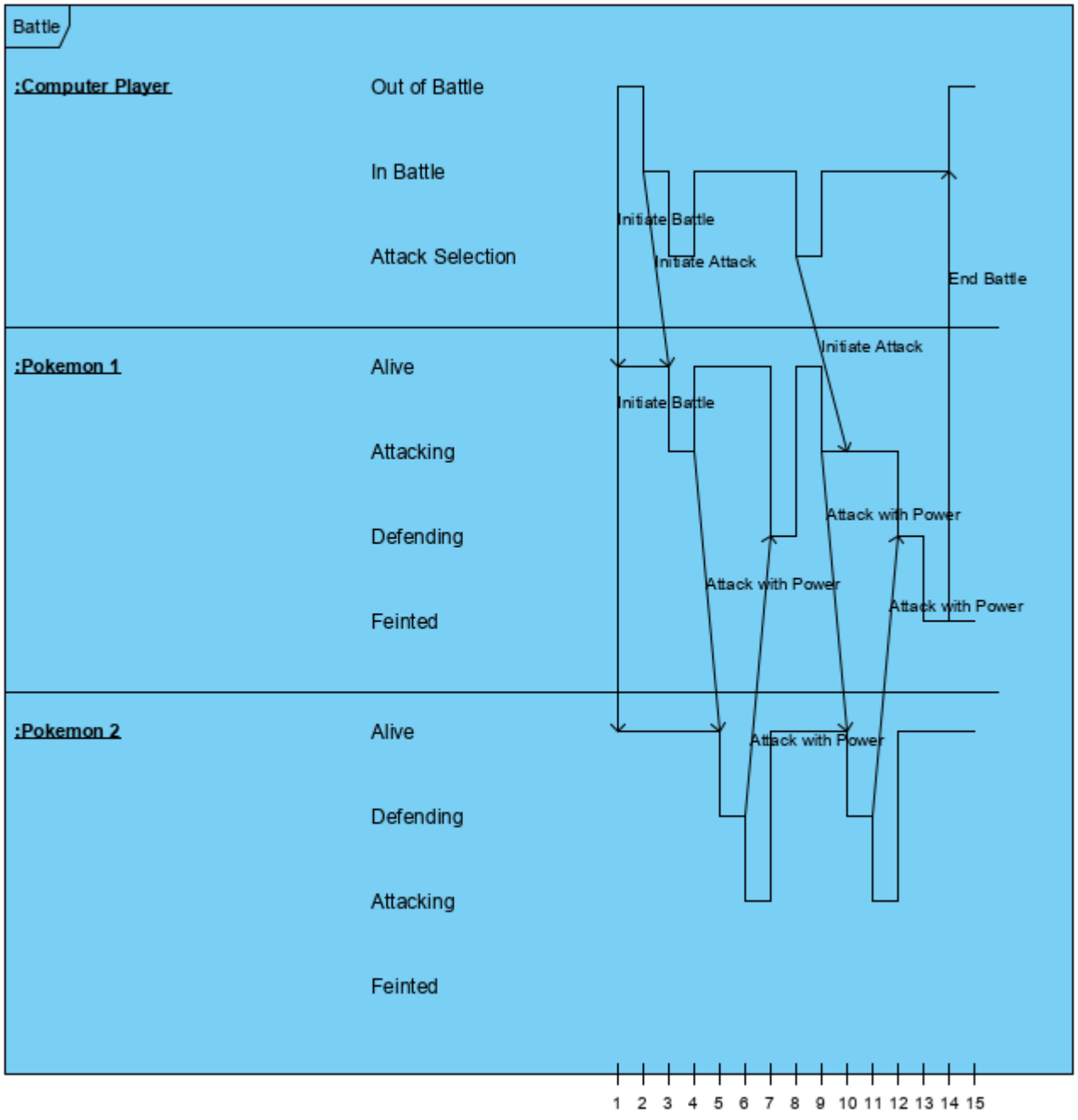


The order of steps is just like in the sequence diagram, however it is easier to see the connection between Pokémon and the player character. Although Pokémon can be programmed as set defined objects in the game, what makes this an element of a game is that the player character must catch Pokémon in wild random battles in order to obtain them. The player can use his/her own Pokémon in order to damage the other Pokémon by using attacks which makes the catch process much easier. If the player's Pokémon's HP is getting low, then the player can use the Item option in order to heal his/her Pokémon. The communication of events occurs in a sequential and random level. The random nature of catches, attack strength, and the encounters themselves allow for a reaction from the player character which dictates the next turn of events.

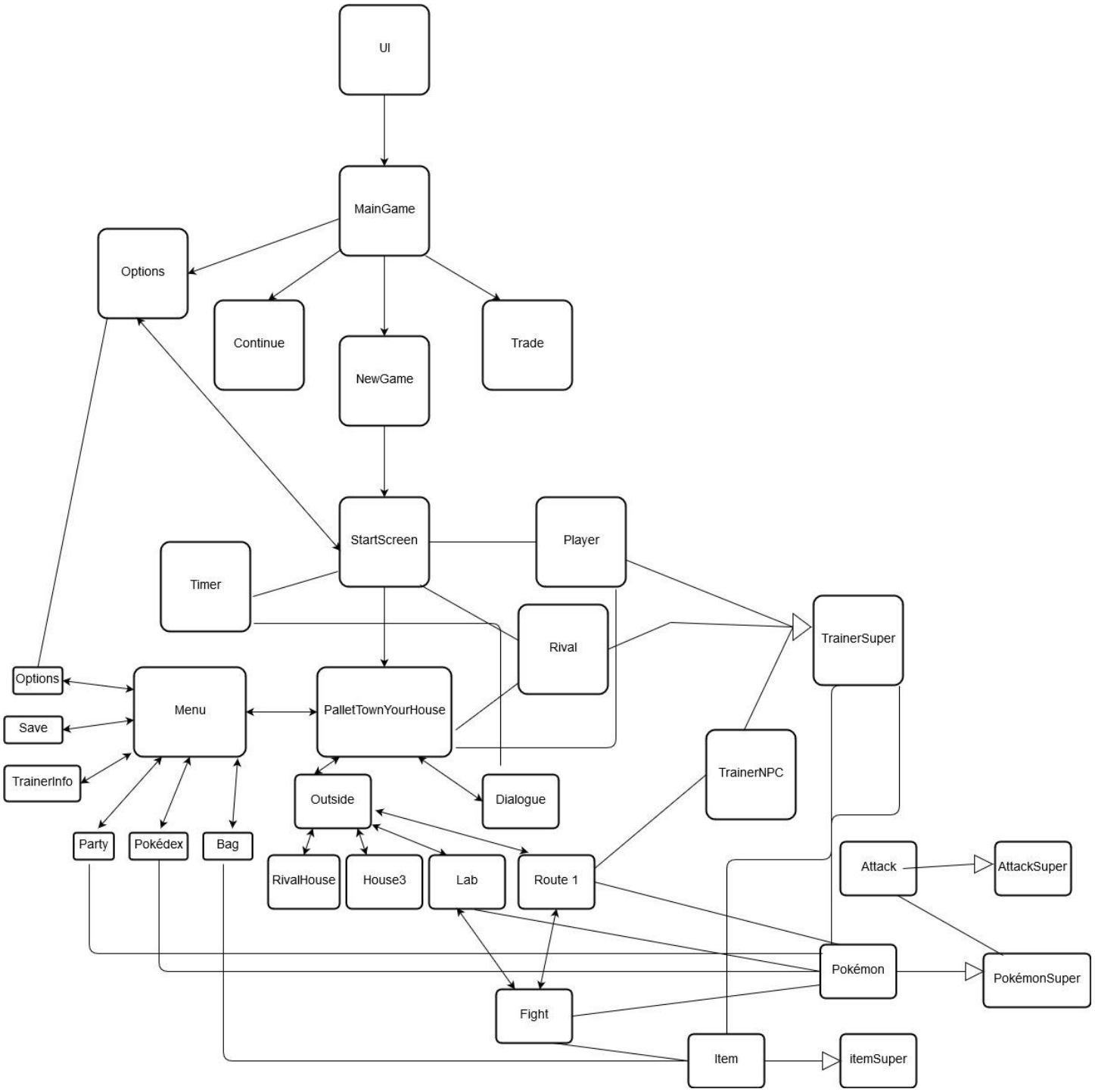
Timing Diagram

Timing diagrams are another type of interaction diagram. A timing diagram focuses on showing the time constraints involved with interactions, which is especially useful for real-time systems. It emphasizes on analysis, identifying states, participants, states for the different participants, and also when events are triggered. Timing diagrams are used to detail interactions based on time such as when events occur, how long they take for other participants to react, and how long it would take to complete each individual interaction.

For example:
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-timing-diagram/

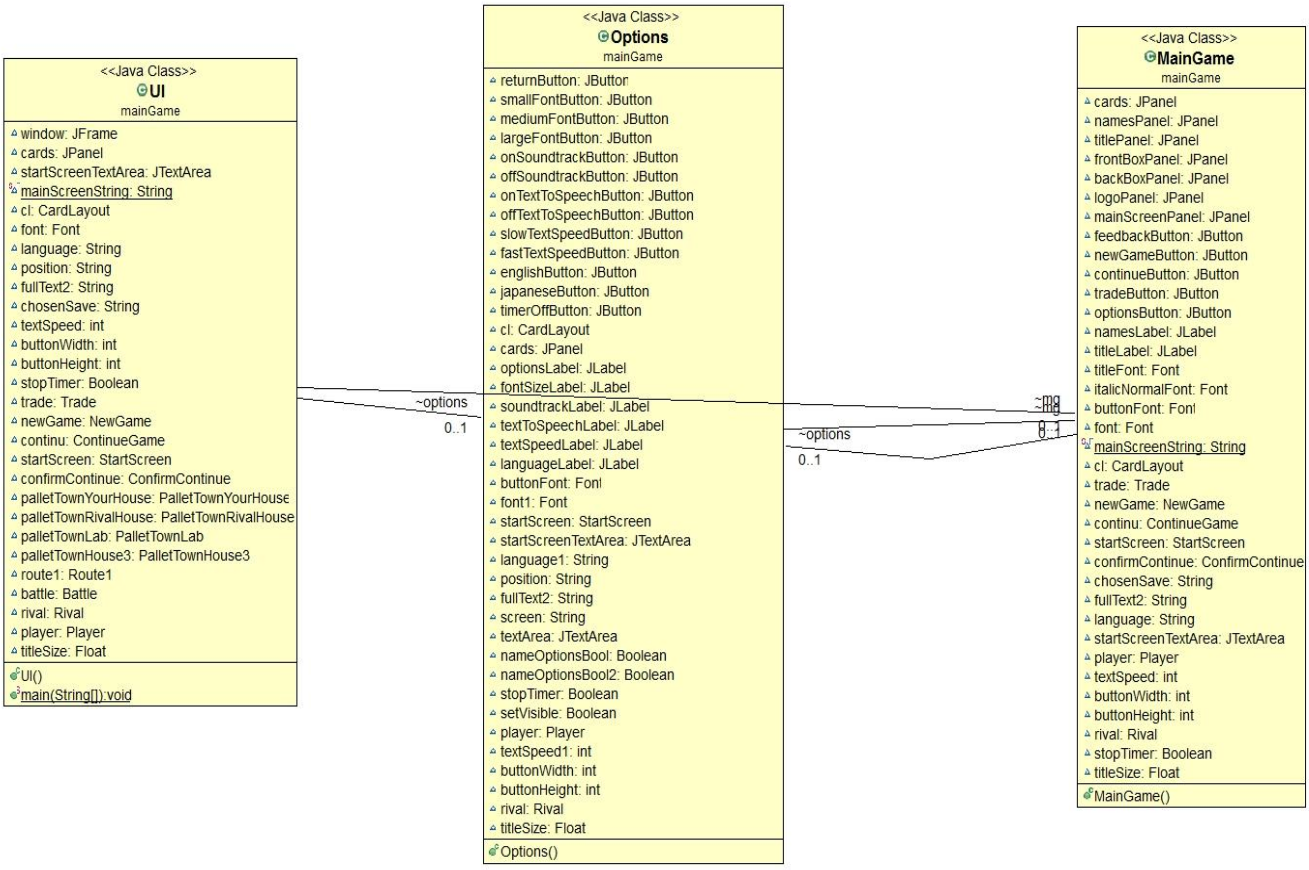


Section 2: Class Diagram and Interface Specification



The UI class initializes each class's relevant JPanel that will be shown. UI however only shows MainGame at first. MainGame is the main area that users see when running the .jar file. It consists of a title, names, and a few buttons. These buttons redirect towards different classes (Options, Trade, Continue, NewGame). NewGame is only for confirmation. When the user starts the actual game, the user goes to StartScreen. The link between Options and MainGame along with Options and StartScreen is the exact same. One can configure options and see them directly from MainGame or StartScreen. The Player object and the Rival object are initialized with their names in the StartScreen class and passed over subsequently. When the game progresses, the user goes to PalletTownYourHouse. Instead of Options, now there is a Menu

button which will contain game related options and system related options within the Menu that redirects to the Options class. From Outside, 3 options go to a general dialogue screen and 1 option goes to the next location which is Outside. Outside has a few options that lead to other locations or general dialogue screen. The Lab is the first time that the player receives a Pokémon and thus the Pokémon object is initialized there and passed on subsequently. Likewise, the Lab is the first time the user faces a battle and thus Battle screen is called passing over any relevant information. The battle screen has access to the Items class and to certain Player information. When a Pokémon attacks, it is calling a move from the Attack class using one of its methods. The game progresses fully in the manner described thus far.



The UI first initializes the window using a JFrame and then creates a first JPanel called cards. This initial JPanel carries all other JPanels that the game will switch to. Each of these JPanels are represented by creating their class objects since each of the pertinent objects extend JPanel. A few exceptions are Trainer and Player which simply have to be initialized in order to be passed from screen to screen. Each class object has its own constructors which will be described individually. This means that all other variables in this class such as Strings, Boolean, Int, etc. are initializing variables for the constructors and hold default values. The main method in UI calls the empty constructor of UI which contains all of the code necessary to initialize all the class objects and the main JFrame window.

The Options class which extends JPanel, is a screen with various buttons and labels. Thus, many JButtons and JLabels are present. Each button has its own function and must change a variable in its own class in order to pass it on to another screen object. Thus, many variables for initialization exist such as the String “language”, the int “textSpeed”, the Font “font”, and the Boolean “stopTimer”. The language can be set to “English” or “Japanese”, the text speed can be set to “slow” or “fast”, the font size can be set to “small”, “medium, or “large”, and the stopTimer can be set to true or false. All the variables are self-explanatory, except for stopTimer.

Whenever a class has dialogue that should be displayed using the typewriter effect, the TimerClass is called with the value of true or false. False will display the typewriter effect and true will not. The Options class passes this value to subsequent screens. The constructor for the Options class contains the following:

```
(final CardLayout layout, final JPanel cards, String position,

                  JTextArea textArea, String screen, Boolean nameOptionsBool,

                  Player player2, Boolean setVisible, String language, int
textSpeed, String fullText,

                  Rival rival2, Font font, Boolean stopTimer1, Boolean
nameOptionsBool2, Float titleSize2,

                  int buttonWidth2, int buttonHeight2)
```

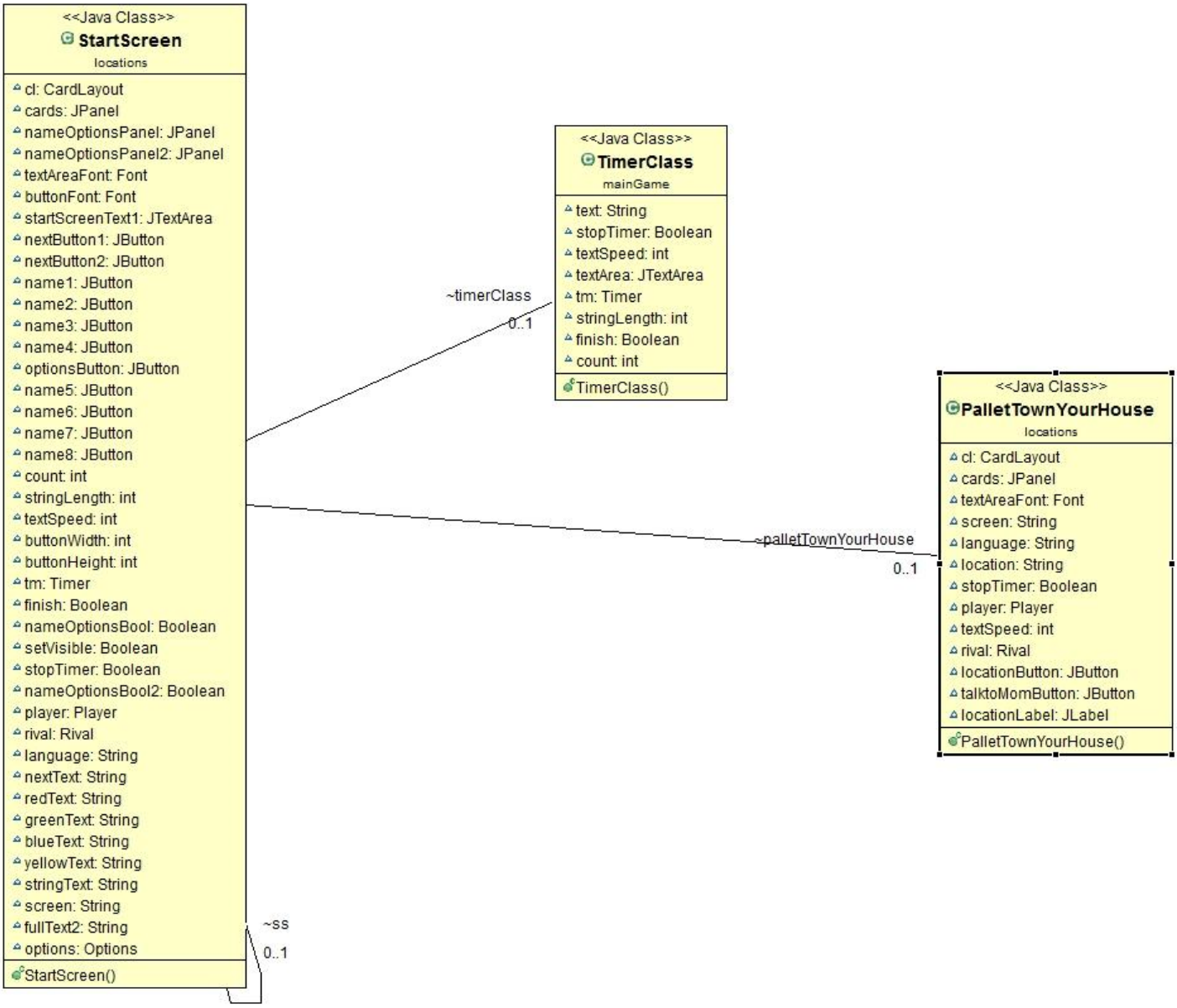
The layout variable and cards variable will be seen in every class that contains a screen because of the way CardLayout is setup. This is only possible because each class with its screen, including the current class of discussion, extends JPanel. The position variable is a String that can be inherited from other objects when the Options class is called. This dictates where the return JButton should go since in the game, the Options screen is present in the Main screen, the starting sequence of the game, and within the menu in subsequent parts of the game. The textArea variable is only included for provisional purposes and if needed. This is due to the fact that changes were made in order to pass dialogue text from another screen to the Options screen and to be able to return to the previous screen with that text specifically. Essentially, almost every variable in the constructor exists in order to pass them back to the previous screen.

The MainGame class also has its own GUI setup, similar to Options, except with an added font made specifically for the upper JLabel called “titleLabel”. The reason why we also have a “titleFont” is because the title has its own specific font as well that can be manipulated in size just as any other JLabel. All other JLabels thus far, rely on the standard variable “font” that is passed from one class to another. Each screen is defined by a String in order to show that screen by using the CardLayout. The only difference with MainGame is that we have a private mainScreenString to define the value needed to switch over to this screen. Each button in this class leads to a new class and thus, objects for other classes are created. The constructor for MainGame is the following:

```
(final CardLayout layout, final JPanel cards, Font font,

            String language, int textSpeed, Boolean stopTimer, Float
titleSize, int buttonWidth, int buttonHeight)
```

Each of the constructors have the same purpose as in the Options class, except for buttonWidth and buttonHeight. This was added for experimental purposes in order to also adjust the size of buttons alongside the size of the text.



The StartScreen class is longer than most classes because it reinitializes its own screen to display different text and GUI elements. A couple of the unused variables due to changes in the code are: buttonFont and finish. The interesting part about this class is that it has a String variable called “screen”. This is initially set to “1” and increments each time the “Next” JButton is clicked to create a new instance of the StartScreen class in order to know which “fulltext2” String to set inside the TimerClass variable “tm”, in order to manipulate StartScreen’s JTextArea “startScreenText1”. This also works by passing any relevant variables when creating a new instance of its own class. Subsequent screens have different JButtons that represent names for the Player variable “player” and the Rival variable “rival”. The reason why there are 2 JPanels and 8 JButtons (nameOptionsPanel1, nameOptionsPanel2, name1, name2, name3, name4, name5, name6, name7, and name8) is because of the fact that separate objects need to have their name created which caused complications when only having one set. Thus, there are Booleans “nameOptionsBool1” and “nameOptionsBool2” in order to control the visibility of the panels containing the buttons. The various Strings with colors in their names decide which language the JButtons should have text in. This is the same way for all of the JButtons. Last, there is an Options object represented through the “options” variable that creates a new instance of Options, setting all relevant constructor parameters that need to be passed on. The StartScreen constructor contains the following:

```
(final CardLayout layout, final JPanel cards, Font font,

                  JTextArea textArea, String screen,

                  Boolean nameOptionsBool, Player player2, Boolean
setVisible,

                  String language, int textSpeed, String fullText,

                  Rival rival2, Boolean nameOptionsBool2, Boolean stopTimer,
int buttonWidth, int buttonHeight)
```

The TimerClass class displays text with a typewriter effect. In order to do this, it must be provided with text and a text area in order to implement the changes necessary. At first, we tried to just include the text area and not the text. This led to disastrous consequences such as, text not being able to be directly manipulated back and forth especially when switching languages. Thus, TimerClass has the String variable “text” and the JTextArea variable “textArea” The Timer object itself which is implemented in this class by using the variable “tm”, takes in a parameter that dictates the speed to display the text. Such is the reason for having an int variable “textSpeed”. The next variable is the int “stringLength” which takes the String text’s length in order to know when to stop displaying the text. Not including this would result in an infinite loop of displaying text. The int variable “count” is included for provisional purposes. Its intention was replaced by the Boolean “stopTimer”. This variable essentially decides if the text should be displayed directly or with the typewriter effect. This comes in handy as sometimes it is necessary to automatically stop the typewriter effect when needed such as exiting the Options class. The constructor for TimerClass is the following:

```
(String text, Boolean stopTimer, int textSpeed, JTextArea textArea)
```

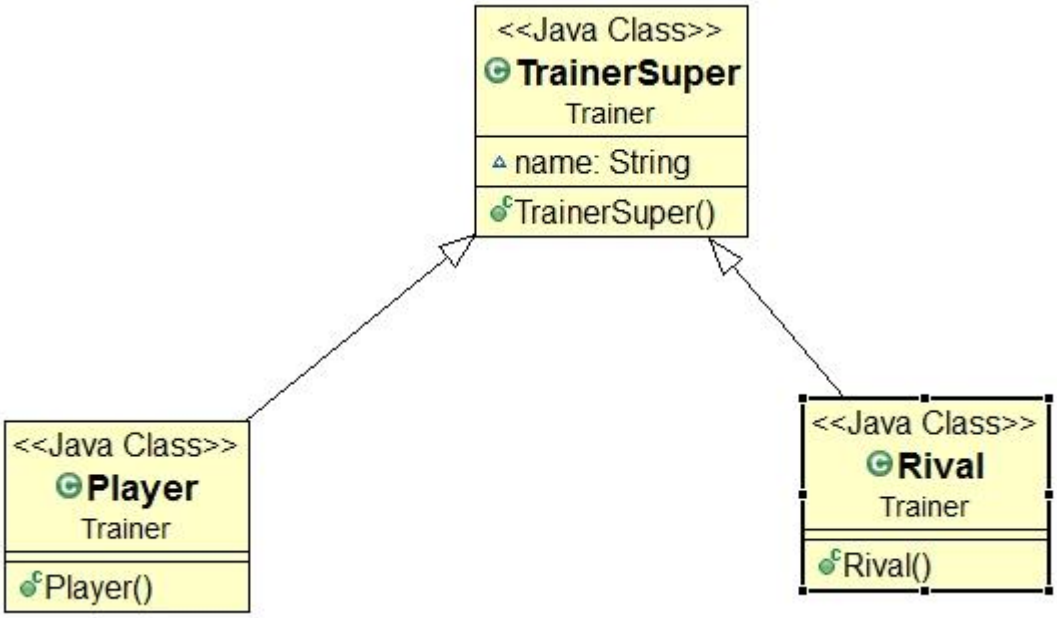
Each one of these parameters is used in the above description as each screen passes on these parameters.

PalletTownYourHouse is the screen that is displayed after StartScreen. It has a layout with various buttons that can lead to various places. This screen in particular has 3 buttons that lead to the same dialogue screen. The 4th option leads to a screen similar to this one. Every pertinent piece of information from StartScreen is passed over such as the language, font, textSpeed, rival, and player. Thus its constructor is the following:

```
(final CardLayout layout, final JPanel cards,

                  Font font, String screen, String language, int textSpeed,

                  Player player, Rival rival, Boolean stopTimer, String
location2)
```



As of writing this, TrainerSuper and its child classes do not have many variables or constructor parameters. The connection between the super class and the child classes are important however, as this same situation will be seen in future super classes such as items, Pokémon, and attacks. The TrainerSuper class will allow the creation of new NPC Trainer characters in the game that have set and defined qualities. The child classes, for now rival and player, can be manipulated based on the choices made. As of now, the TrainerSuper class only has a String name attribute with a getName and setName method. Both rival and player use the super definition for the String name. Thus the constructors for each are the following:

Trainer Super: `(String name)`

Player: `(String name)`

Rival: `(String name)`

The ContinueGame class for right now, only has 3 JButtons with their JLabels set to the corresponding save number. There are plans for future implementations in the future, however for right now clicking on a save leads to the ConfirmContinue Class which displays the text of the save file number passed onto it from ContinueGame.

The constructors for both are the following:

ContinueGame: `(final CardLayout layout, final JPanel cards)`

ConfirmContinue: `(final CardLayout layout, final JPanel cards, String name)`

The NewGame class is the class that leads to StartScreen. Thus there is a connection from MainGame to NewGame. This is essentially a confirmation screen with a “yes” button that leads to StartScreen and a “no” button that goes back to MainGame. When clicking on “yes” a new StartScreen object is created, hence the “ss” variable. This serves the purpose of creating a

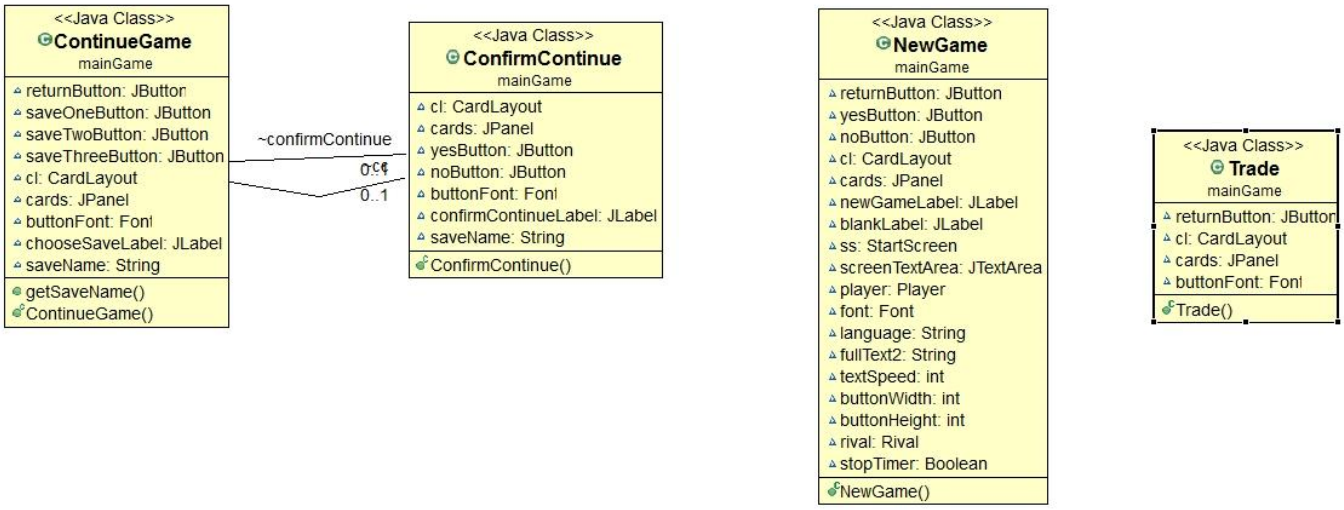
default “screen” String as defined above. This is one of the cases in which a variable is not needed, although included. For this reason, other variables are created such as Player player, Rival rival, Boolean stopTimer, Font font, String language, and String fullText2. There is also a blank JLabel called “blankLabel” that serves the purpose of aligning this screen properly when using the GridBagLayout. The constructor parameters for NewGame are the following:

```
final CardLayout layout, final JPanel cards, Font font, String language, int
textSpeed, Boolean stopTimer,

                    int buttonWidth, int buttonHeight
```

The Trade class, just like the ContinueGame class, is for provisional purpose right now. Nothing has been implemented inside, except for a return JButton. The constructor parameters are the following:

```
final CardLayout layout, final JPanel cards
```



Section 3: System Architecture and Design

Architecture Styles

In this section, we discuss the activities and tasks that are relevant in developing a good system architecture. A system architecture is the conceptual model that defines the structure, behavior, and views of a system. An architecture description is a description and representation of a system which is organized in a way that supports reasoning about the structures and behaviors of the system.

Since our application is complex in terms and form of the architecture, we use several architectural styles together in order to model the game system. In this section, we will discuss the architectural styles of our game system from a higher level and then break it down into several pieces for discussion.

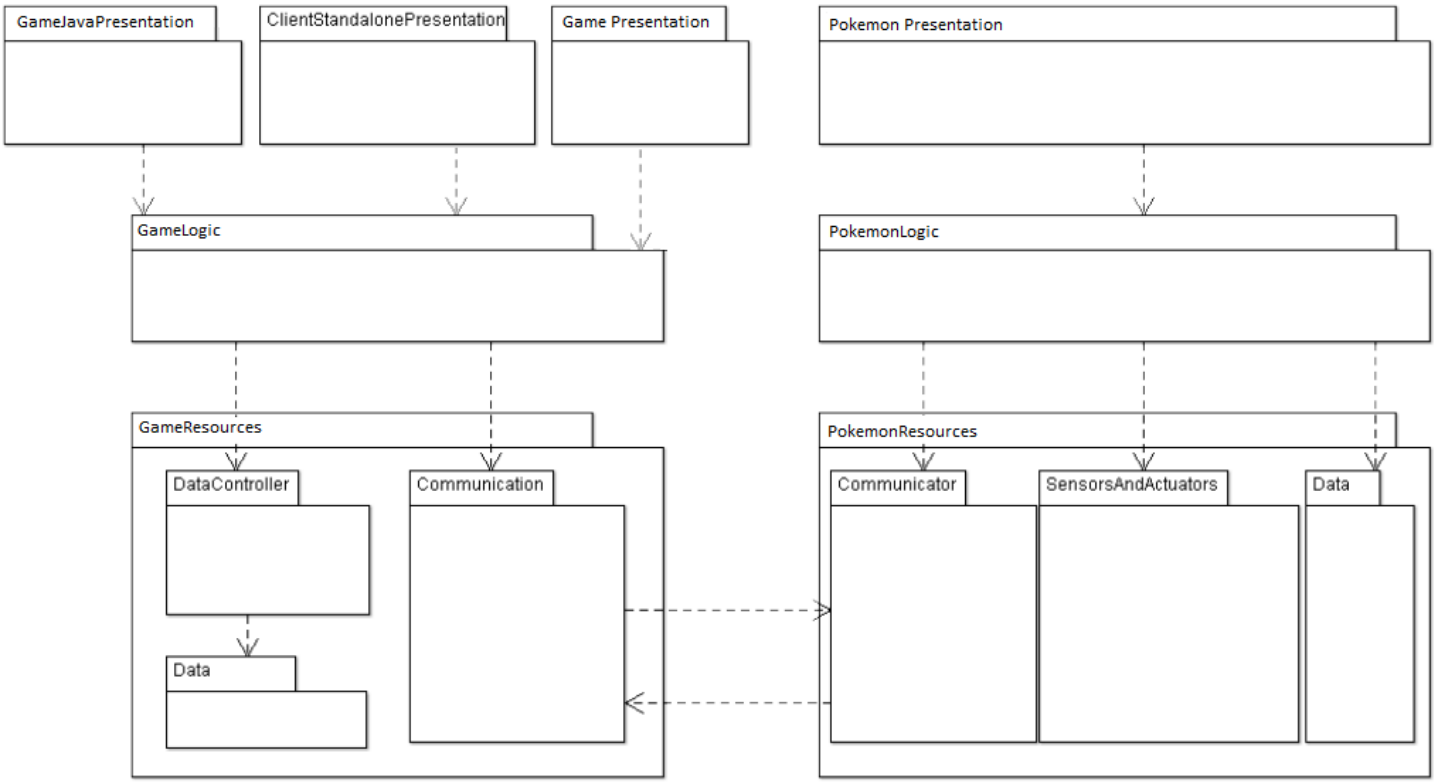
We can generally identify the structure of our system with the following components::

- Java Client
- Game Server
- Pokémon

The three subsystems are categorized as Java client, game server, and Pokémon) into an architectural style. The client/game server architecture emphasizes the setup of communication between the player and the game. Essentially, we are using a typical client/server architecture.The purpose of this architecture is to allow access for one or more clients to numerous resources or functionalities in a central server. A client such as a Java application in a PC allows multiple clients to communicate with the game server at the same time. A client/server architecture has been known to be more stable and less complex to implement. It is important to recognize the association of a system embodied in its components along with their relationships to each other and to the environment. It is also significant to recognize the principles governing the design and progression. If documented, a description of the design and contents of our game system may include information such as a detailed inventory of current hardware and software and networking capabilities. This would allow for future and potential plans and priorities on future purchases along with plans for keeping up with upgraded systems and new innovations in the gaming industry.

Identifying Subsystems

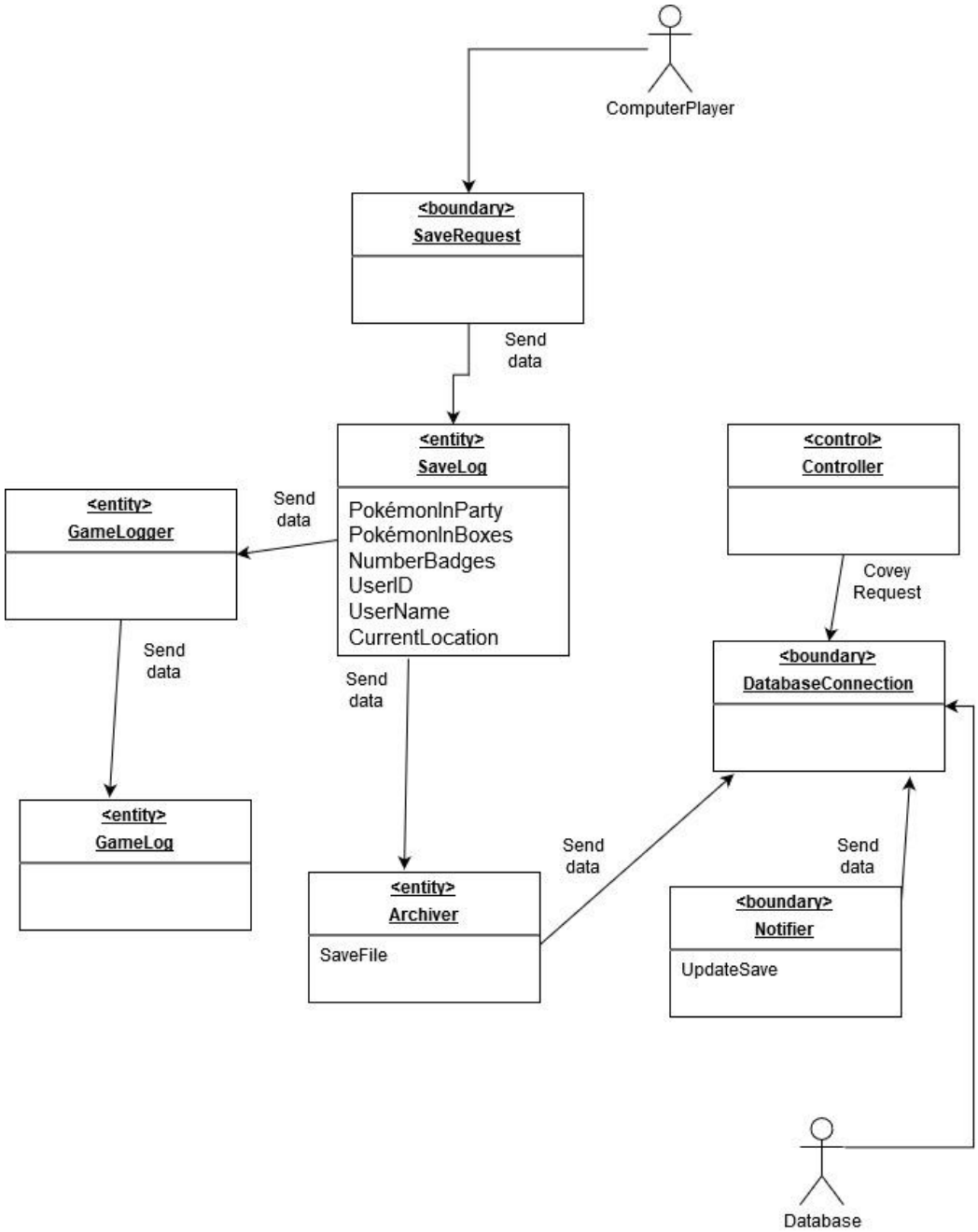
A system architecture primarily concentrates on the internal interfaces among the system's subsystems or components. It also focuses on the interfaces between the system and its external environment, especially the user. In the specific case of computer systems, this latter, special, interface is known as the computer human interface or the human computer interface. To give an overview of the core packages in our gaming software and the dependencies among them, we used the UML package diagram shown in the figure below:



Mapping Subsystems to Hardware

The subsystems can certainly be mapped into the hardware components and peripherals such as a keyboard, mouse, printer, Nintendo switch controllers, etc. In the UML package diagram, the ServerLogic and ServerResources packages are allocated in the game server. The PokémonPresentation, PokémonLogic and PokémonResources packages are allocated in the Pokémon. The GameJavaPresentation is allocated in the game server, and the user needs a Java client to access this presentation. The GameJavaPresentation will be considered as a singular part of future work. ClientStandalonePresentation package is allocated in the client's PC. Finally, the GamePresentation is allocated in the gaming devices such as Nintendo Switch, Gameboy emulators, or any other system with capabilities to play games. There will be many options to consider for future work on this project including porting to Android. On the other hand porting to an operating system such as iOS would complicate the nature of the java code as iOS requires objective C code.

Persistent Data Storage



Persistent Data Storage has to be evaluated for any data that will have to outlive one single execution of the system being used. For the game we are creating, this function will mainly be used when the user decides to save their progress during one specific game session. The diagram above details the process starting with the user choosing to save the game progress which filters through all aspects until the data is saved into the Player database.

The player starts the save game process by choosing that option within the main menu. The system will then prompt a “Yes or No” question to ensure the player is ready to save their progress. Once the player clicks on “Yes”, the system evaluates all of the objects that are

defined in the save log. Various objects like location, UserId, UserName, and others are defined with the SaveLog box. Once the system has compiled all of this data, it transfers it to the archiver. Next, the system will prompt the user to create a specific ID to attach to this specific game progress. Once the ID is confirmed, this data will be sent to the Player database. This database will hold all of the saved data that is associated with the player that is currently using the system. The database feature is only used for the trade feature. In other words, loading an instance of a game occurs within the game file itself unless the program needs to look up information regarding save data inside the database.

Save Log	Pokémon In Party	Contains core information such as names, levels, stats, moves
	Pokémon In Boxes	Contains core information such as names, levels, stats, moves, box number
	Number of Gym Badges Obtained	Essentially game completion progress, there are 8 gym badges
	User ID	Random ID created from initializing/starting a game
	User Name	Name that user chooses at start of game
	Current Location	Location that user saves from

The above table details the 6 objects that will be evaluated when saving a player's progress. All of the items will be saved into a log file to help the system regain its current state when the user decides to continue playing the game during the next session.

Global Control Flow

The game server will be implemented as an event driven system that maintains the state of the game. It will loop and wait for events sent from the clients that it will then process and respond to. For example movement through the map will involve the server keeping track of that particular players current position and when it receives a move event it will process the event, determine any variables and then return the status to the client. The client will also be event driven, it will maintain a state that is in sync with the server state. It will wait for input events from the user and trigger further events to update the server state, and any changes that need made in the graphics output based on the received input events.

The game client will run in a loop and maintain a timer in the loop that will be used to maintain an output frame rate. Each iteration of the game loop will be controlled by this timer and render one frame of output. There will also be a seperate timer used to control input and output input text in a typewriter effect. This real-time system will be periodic based on the render loop timer and input timer.

The system will run several concurrent threads. There will be a render loop thread that is controlling graphic output. The server will run in a thread waiting for events from the clients. The client thread will run a loop waiting for user input. We will need to avoid any race conditions and

deadlocks between these threads. For example the graphic output thread will be rendering the same state data that the server thread is updating constantly. We will need to ensure the graphic output thread is only a reader of the state and any changes written to the state happen in the server thread to avoid any race condition. Since we will be using an event driven system we will need to make sure that all events trigger something and don't result in two threads waiting on each other causing deadlock.

Hardware Requirements

Our game has a window set by default to 1280 x 680 resolution. We have plans for implementing other resolution styles or even scaling the contents to the window based upon device screen size. For now, the user can change the font which can help to mitigate issues when playing on a smaller or larger screen than the default. The display screen must support colors as the game has some font in different colors and a colorful logo, although for the most part, the colors are black and white.

On my Windows 8.1 machine, the game takes 83 MB of memory when running the game and less than 1% of CPU in most cases, although it can go as high as 5-10% due to the TimerClass typewriter effect. The .jar file itself is small at about 362 KB at the moment. If the Trade class is implemented, the game will require a network connection only for that aspect of the game.

The game will also be using audio output for game sound effects and verbal descriptions of the game. So any hardware will require a speaker output. We will be implementing a save game option and this will require some persistent storage of a small save game file. Save game files will be written to whatever storage the device they are running on is using. We won't need to worry about specifics as we will use an abstract JVM call to write output to whatever storage it is setup on.

Section 4: Algorithms and Data Structures

Algorithms

- Items
    - Check for player item array list, Add each item object's name to button, and stop when reaching the size of the item Array List
    - Some items such as potions, revive, or full heal have their own methods to update a Pokémon's stats, HP, or even revive them
    - Map has a method to show a new screen containing a map image
    - Player has initial methods to use, however the main object methods are within the objects themselves. This allows the programmer to call the method on Player for simplicity. This has been done for the Map item object thus far.
    - Items can be sold to update player's money
    - Items have a quantity property that can be incremented or decremented
    - Different Pokéballs (considered items) have a differing range of randomness for catch rate determined through a random method with the base value determined through the type of the Pokéball.
- Pokémon Stats
    - A Pokémon's HP, Defense, Attack, Sp. Attack, and Sp. Defense can be updated throughout the game from using items, battling, or leveling up.
        - Methods must be implemented for the player to call methods on other objects.
    - Pokémon have their own types that are effective or non-effective against other types
        - Type method will have to be implemented to decide the baseline

for randomness in terms of super effective, not effective, or standard.

- Pokémon Encounter Rate
    - Each Pokémon has a specified range of percentage in order to encounter it in the wild
        - Method to generate random value based off of a property of a Pokémon (encounterRate).
- Pokémon Catch Rate
    - Each Pokémon has a specified catch rate in a range also dictated by randomness and which Pokéball one uses
        - Method to generate random value based off of a property of a Pokémon (catchRate).

- Attacks
    - Each attack creates a random number based upon its power stats
    - Attacks have their own types that are effective or non-effective against other types
    - Attacks have a property called PP that must be incremented upon using a special item or decremented upon using that attack.
    - Attacks have their own accuracy. The value of the property (0-100) is the actual percentage and must be mapped accordingly.
- Items
    - Check for player item array list, Add each item object's name to button, and stop when reaching null object from item array list
    - Dynamically map Items in itemArrayList to buttons in buttonArrayList by looping through each element and setting the ith position of itemArrayList as a button's text.
    - Add methods for each item button through using if statements in both languages, checking for the name of the item inside itemArrayList and making sure it is mapped to a specific button.
    - Some items such as potions, revive, or full heal have their own methods to update a Pokémon's stats, HP, or even revive them
    - Items can be sold to update player's money
    - Different Pokéballs have a differing range of randomness for catch rate

Data Structures

- Array List for flexibility with Trainer Items as It can be updated and manipulated often. Arrays have the disadvantage of not being able to stretch out the size but is not as dynamic
- Arrays for static list of items for other trainers
- Arrays for player's party Pokémon and NPC trainer's Pokémon
- Multidimensional Array for Pokémon in boxes. 12 Boxes of 30 Pokémon inside each.
- Sorting algorithm is implemented inside ArrayList. Any data structure that requires sorting, will use an ArrayList for ease of use, instead of dealing with different types of sort algorithms.

All of the above decisions are made by deciding flexibility and so much performance as that is something that can be taken care of after the initial implementation. Most of the

focus is going towards being able to dynamically add new objects to the game without having to worry about the specific decisions that we had made in the implementation. In some cases, arrays are used for ease of use when not many objects are necessary inside of one. Both have their advantages and disadvantages in certain situations such as static or dynamic size, built in sorting mechanisms, and type conversion from one to another. In other words, both data structures can be converted from one to another without much difficulty. If one must become the other for certain circumstances, such as an array to ArrayList to increase speed, then that can be done. At this point, these are the only data structures that we will be using although we are open to using more as the game progresses and becomes more complex.

Section 5: User Interface and Design Implementation

At this point of the game, we have done exactly as we had described with the initial mock ups in terms of layout. We decided on simplicity at the beginning and what we have ended up with is a product that illustrates ease of use in terms of one who is not knowledgeable with games. There are a few areas however that we have changed. One of these areas is the fact that initially, the menu button was not planned to hold the options button and the placement of both these buttons were not determined at the time. In creating the game, we decided that it would be better to allow the user to dynamically change settings in almost every part of the game, except for areas where it would not make sense. Furthermore, we made changes to the confirmation dialogue boxes that show up when clicking on certain options and items thus far. We may actually implement this part, although it requires more user effort than expected especially since this would have to be done for over 50 elements at this point in time.

Another thing that we have changed at this point in time is that fact that some areas of the game, we had initially mocked up the illustration that navigation would only have a couple of buttons. In order to finish the GUI in a timely manner, we have decided to reuse the layout of 4 outside buttons and 1 inside label that shows the location. If an area does not make use of certain buttons, it is kept blank with no action listeners. This also makes it easier for the layout that we are primarily using (GridBagLayout), which we had initially not determined that we would use. The last thing that we did not realize from the beginning is that we have an initial Map item in the game which should logically switch to a new screen and have some sort of layout. In the process of finding clean images for this screen, we had decided to use a blank interface for the map that is similar to the actual games. By doing this, we have been able to manually add the text of the locations which is not something one would find in English and in Japanese while browsing the Internet. At this point in time, we prioritize ease of use and simplicity as opposed to being flashy.

Section 6: Design of Tests

Test Cases

1. JUnit Test case for names of objects (items, attacks, Pokémon) depending on the language
2. JUnit Test Case for range of random values associated with an attack's power on hit, Pokémon stat production, Pokémon encounter rate, and Pokémon catch rate
3. Integration Testing of combination of objects that produce random values
    a. Pokémon stats yielding attack power + Attack stats yielding attack power
    b. Pokémon stats yielding attack power + Pokémon type + Attack stats yielding

attack power + Attack type

c. Pokémon stats yielding attack power + Pokémon type + Attack stats yielding attack power + Attack type + critical hit (rare)
d. Pokémon stat production + Pokémon encounter rate production
e. Pokémon stat production + Pokémon catch rate production
f. Pokémon encounter rate production + Pokémon catch rate production
g. Pokémon encounter rate production + Pokémon catch rate production + Pokémon encounter rate production

Coverage of Test Cases

Test coverage measures the degree to which the specification or code of a software program has been exercised by tests. Test Coverage aims to measure the effectiveness of testing in a qualitative manner. It determines whether the test cases are covering entire functional requirements. We have to evaluate this section from a standpoint that the test cases are not written based on code but based on user requirements or expected functionality.

There are many different types of testing that will be used before we get to the final stage of our project. The most popular style of testing for our project will be unit testing. Unit testing is being used often as we code. For unit testing, we are breaking down our code into “units” instead of evaluating it initially as a whole. As we code a function or method, we immediately test that unit of code to ensure that it fulfills the requirement of its initial purpose. Any issues within that unit are resolved at that moment so that other units afterwards are not affected. Unit testing also prevents you from getting to the end and having multiple bugs that have to be diagnosed and sorted through.

The tests will cover as many objects as we have produced by the time of each demo. As most of the test cases deal with random number values (integer or decimal), the accuracy of the tests is solely determined by what makes sense for a battle inside a game. For instance, a level 5 Pokémon should not be able to produce an attack value of 100. Its range would be ~5-10. Any value inside an expected range is acceptable and should hold true for the rest of the game. The use cases will cover the mathematical models detailed which follows strict rules within the game. For instance, levels go from 5-100. There is no other value associated with level. Once the units have been tested, the code will be combined and we will transition to the integration stage.

Integration Testing Strategy

Maven is a popular option for integration testing which we plan to use at the moment. The singular JUnit test cases can be done through Eclipse after a set of object implementations is complete along with the methods required to produce values that require testing (random values). Integration testing will only be done once all of the object pieces fit together accordingly with their methods.

During our integration testing stage, we will be using the big bang integration approach. All of the methods and functions will already be tested during the unit testing phase. Now we are integrating all of the units into one working system. We will start with the main interface and then work our way down by priority. The goal is to also create uniformity amongst the functions/methods since different group members may do separate functions/methods during the coding phase. The integration testing stage will be a great opportunity to gauge how close or how far we are from having a fluid and complete system.

Project Management and Plan of Work

`

Product: A Pokémon text audio game meant for people who have great nostalgia for the original games. This product is meant for the visually impaired as well. It is meant to be an easy game to pick up and play for anyone. This game will be in Java with a hand-coded layout and will introduce complex features not seen in games similar to this one.

Responsibilities So Far:

Rahil:

- Completed coding the beginning of the game
- Integrated features into game such as dual language support, changing text size, and changing text speed
- Provided group with links, ideas, helpful tips, comments, and suggested edits for section 1 of Report 2
- Kept group aware of updates needed for Report 1
- Created video to explain code to group: https://www.youtube.com/watch?v=t28rItN_ZGM&feature=youtu.be

Gary:

- Communication Diagram
- Persistent Data Storage Section
- Continue coordinating tasks with teammates

Chad:

- Notified Group of Maven
- Timing Diagrams
- Begin Implementing Classes
- Global Control Flow
- Hardware Requirements
- Project coordination and Progress
- Plan of Work

Luis:

- Communicated with team on responsibilities for report and project
- Researched GUI code and shared links to support project
- Worked on design and layout of report
- Included research and text on report sections
- Created Google Docs for file sharing
- Provided group with links, ideas, helpful tips, comments, and made corrections for section 1 & 2 of Report 2

- Kept group aware of deadlines for equal contributions towards reports and project

Project Coordination and Progress Report

1. Functional Use cases
    a. UseItem
        i. Map item has been implemented in ArrayList. The same concept can be applied to other objects except their methods will not pertain to switching screens.
    b. MultiLanguage
        i. This use case has not been mentioned before, but it replaces the trade, soundtrack, and textToSpeech use cases. It is fully working and one can switch to English/Japanese whenever he/she chooses in the options.
2. In Progress use cases
    a. Fight
        i. We are right about the point in which we implement this one as the first battle is coming up (rival battle)
    b. Catch
        i. After rival battle, route 1 is accessible which will allow trainer to catch Pokémon
    c. Save/Load
        i. Towards the end of route 1, we will implement save/load

Plan of Work

* = Unfinished

1. GUI creation with locations
2. User Interface Options Integration
3. English Dialogue
4. Dual Language Dialogue
5. Initial Object creation
    a. 3 Trainers
6. Initial Item Integration
    a. Map
7. *Finish Object Creation*
    a. 4 Pokémon
    b. 3 Trainers (add methods and properties)
    c. 8 Attacks
8. *Finish Item Integration*
    a. Potion
    b. Pokéball
9. *Mathematical Methods for Battle*
10. *Implement Battle Screen*
11. *Implement Catch Use Case*
12. *Implement Save/Load Use Case*

Future Responsibilities:

We hope to share all responsibilities.

Rahil:

- Work on reports
- Work on code

Gary:

- Work on reports
- Work on code

Luis:

- Work on reports
- Work on code

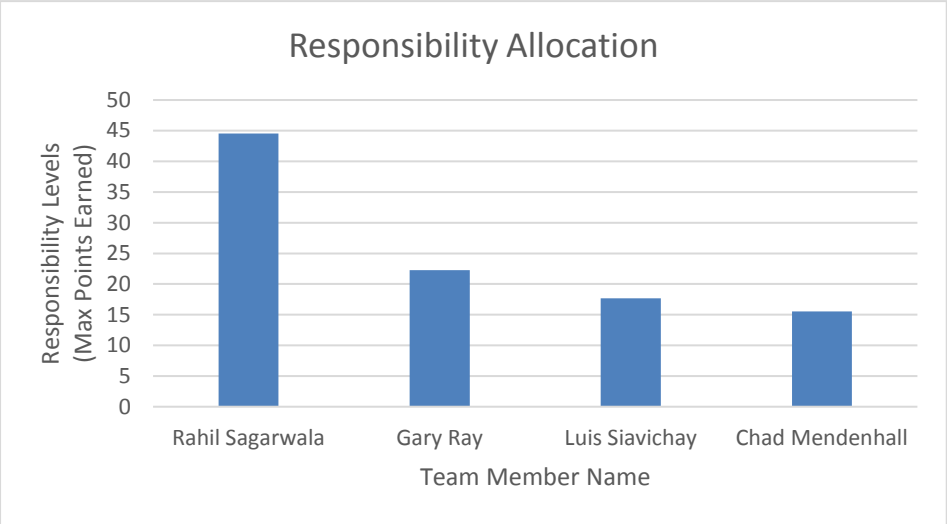
Chad:

- Work on reports
- Work on code

Responsibility Matrix

		Team Member Name			
		Rahil Sagarwala	Gary Ray	Chad Mendenhall	Luis Siavichay
	Project Management (18 Points)	30%	20%	30%	20%
Responsibility Levels	Sec.1: Interaction Diagrams (30 points)	25%	25%	25%	25%
	Sec.2: Class Diagram and Interface Specification (10 points)	80%	10%	0%	10%
	Sec.3: System Architecture and System Design (15 points)	15%	25%	15%	45%
	Sec.4: Algorithms and Data Structures(4 points)	100%	0%	0%	0%
	Sec.5: User Interface Design and Implementation (11 points)	100%	0%	0%	0%
	Sec.6: Design of Tests (12 points)	50%	50%	0%	0%

Responsibility Allocation



Rahil: (0.3 x 18) + (0.25 x 30) + (0.80 x 10) + (0.15 x 15) + (1.0 x 4) + (1.0 x 11) + (0.50 x 12) = 5.4 + 7.5 + 8 + 2.25 + 4 + 11 + 6 = **44.55**

Gary: (0.20 x 18) + (0.25 x 30) + (0.10 x 10) + (0.25 x 15) + (0 x 4) + (0 x 11) + (0.50 x 12) = 3.6 + 7.5 + 1 + 3.75 + 0 + 0 + 6 = **22.25**

Chad: (0.30 x 18) + (0.25 x 30) + (0 x 10) + (0.15 x 15) + (0 x 4) + (0 x 11) + (0 x 12) = 5.4 + 7.5 + 0 + 2.25 + 0 + 0 + 0 = **15.55**

Luis: (0.20 x 10) + (0.25 x 30) + (0.10 x 10) + (0.45 x 15) + (0 x 4) + (0 x 11) + (0 x 12) = 2 + 7.5 + 1 + 6.75 + 0 + 0 + 0 = **17.65**

`

References

Bulbapedia.bulbagarden.net. (2019). *Statistic - Bulbapedia, the community-driven Pokémon encyclopedia*. [online] Available at: https://bulbapedia.bulbagarden.net/wiki/Statistic [Accessed 20 Sep. 2019].

Bulbapedia.bulbagarden.net. (2019). *Catch rate - Bulbapedia, the community-driven Pokémon encyclopedia*. [online] Available at: https://bulbapedia.bulbagarden.net/wiki/Catch_rate [Accessed 20 Sep. 2019].

Downloads.khinsider.com. (2016). *Pokemon Red, Green, Blue & Yellow MP3 - Download Pokemon Red,*

En.wikipedia.org. (2019). Unified Modeling Language. [online] Available at: https://en.wikipedia.org/wiki/Unified_Modeling_Language#Interaction_diagrams [Accessed 25 Sep. 2019].

Gamefaqs.gamespot.com. (2007). *Pokemon Red Version - Game Script - Game Boy - By mtkennerly - GameFAQs*. [online] Available at: https://gamefaqs.gamespot.com/gameboy/367023-pokemon-red-version/faqs/48982 [Accessed 20 Sep. 2019].

Green, Blue & Yellow Soundtracks for FREE!. [online] Available at: https://downloads.khinsider.com/game-soundtracks/album/pokemon-red-green-blue-yellow [Accessed 20 Sep. 2019].

Homepage.cs.uiowa.edu. (2019). [online] Available at: https://homepage.cs.uiowa.edu/~tinelli/classes/022/Spring15/Notes/chap11.pdf [Accessed 22 Sep. 2019].

M.bulbapedia.bulbagarden.net. (2019). *Experience - Bulbapedia, the community-driven Pokémon encyclopedia*. [online] Available at: https://m.bulbapedia.bulbagarden.net/wiki/Experience [Accessed 20 Sep. 2019].

Marsic, I. (2012). *Software Engineering*. New Brunswick.

Miles, R. and Hamilton, K. (2006). Learning UML 2.0. Cambridge: O'Reilly.App.creately.com. (2019). *Creately*. [online] Available at: https://app.creately.com/diagram/iREP4saBgEy/edit [Accessed 20 Sep. 2019].

Visual-paradigm.com. (2019). What is Timing Diagram?. [online] Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-timing-diagram/ [Accessed 25 Sep. 2019].

Visual-paradigm.com. (2019). *What is Communication Diagram?*. [online] Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-communication-diagram/ [Accessed 27 Sep. 2019].

Sealights. (2019). *Code Coverage vs. Test Coverage: Pros and Cons | SeaLights*. [online] Available at: https://www.sealights.io/test-metrics/code-coverage-vs-test-coverage-pros-and-cons/ [Accessed 14 Oct. 2019].