Ramble On
Software Design Description

Author: Yvonne DeSousa
CSE 219 Stony Brook University
March 2013
Version 1.0

Abstract: This document describes the software design for Ramble On, an educational mini-game for learning geography and facts about countries and other regions.

Based on IEEE Std 1016TM-2009 document format

Copyright © 2013 Yvonne DeSousa, with additional frameworks provided by Richard McKenna under authorization

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

1 Introduction

This is the Software Design Description (SDD) for the Ramble On mini-game application. Note that this document format is based on the IEEE Standard 1016-2009 recommendation for software design.

1.1 Purpose

This document is to serve as the blueprint for the construction of the Ramble On application. This design will use UML class diagrams to provide complete detail regarding all packages, classes, instance variables, class variables, and method signatures needed to build the application. In addition, UML Sequence diagrams will be used to specify object interactions post-initialization of the application, meaning in response to user interactions or timed events.

1.2 Scope

Ramble On is an educatonal mini-game about learning the geography and other facts about countries and other regions. A base framework called MiniGameFramework in the Java programming language has been provided by Richard McKenna, and will be used in the design.

1.3 Definitions, acronyms, and abbreviations

Class Diagram – A UML document format that describes classes graphically. Specifically, it describes their instance variables, method headers, and relationships to other classes.

IEEE – Institute of Electrical and Electronics Engineers, the “world’s largest professional association for the advancement of technology”.

Framework – In an object-oriented language, a collection of classes and interfaces that collectively provide a service for building applications or additional frameworks all with a common need.

Java – A high-level programming language that uses a virtual machine layer between the Java application and the hardware to provide program portability.

Mini-Game – A standalone game that is a subset of a larger game application, typically sharing the primary game theme with that parent game application.

Mini Game Framework – The software framework to be developed in tandem with the Zombiquarium game such that additional mini-games can easily be constructed. Note that in the Zombiquarium SRS this was sometimes called the “Mini Zombie Game Framework”, but has been renamed the “Mini Game Framework”, since it’s not Zombie-specific.

Ramble On - The name of the mini-game developed, which has an educational focus where one must learn to "ramble on" the names/flags/leaders of a certain region.

Region – A place on earth (continent, country, state) that contains attributes such as a name, flag, and leader. Some may be able to be used as a game level, and then would contain Subregions.

Sequence Diagram – A UML document format that specifies how object methods interact with one

another.

Sprite – a renderable, and sometimes movable or clickable image in the game. Each Sun, Zombie, and Brain will be its own Sprite, as will GUI controls.

SpriteType – a type of Sprite, meaning all the artwork and states corresponding to a category of sprite. We do this because all the suns share artwork, so we will load all their artwork into a common Sprite Type, but each Sprite has its own position and velocity, so each will be its own Sprite that knows what Sprite Type it belongs to.

Subregion - A Regions that belongs to a set of Regions that make up a larger one. An element of a Region's subdivision, such as states to a country.

UML – Unified Modeling Language, a standard set of document formats for designing software graphically.

1.4 References

IEEE Std 830TM-1998 (R2009) – IEEE Standard for Information Technology – Systems Design – Software Design Descriptions

Ramble On SRS – Software Requirements Specification for the Ramble On application, as per the website of CSE 219

Zombiequarium(tm) Software Design Description – Richard McKenna, distributed by Debugging Enterprises(tm), November, 2011

1.5 Overview

This Software Design Description document provides a working design for the Ramble On software application as described in the Ramble On Requirements Specification. Note that all parties in the implementation stage must agree upon all connections between components before proceeding with the implementation stage. Section 2 of this document will provide the Package-Level Viewpoint, specifying the packages and frameworks to be designed. Section 3 will provide the Class-Level Viewpoint, using UML Class Diagrams to specify how the classes should be constructed. Section 4 will provide the Method-Level System Viewpoint, describing how methods will interact with one another. Section 5 provides deployment information like file structures and formats to use. Section 6 provides a Table of Contents, an Index, and References. Note that all UML Diagrams in this document were created using the VioletUML editor, with some minor graphics editing in GIMP.

2. **Package-Level Design Viewpoint**

As mentioned, this design will encompass both the Zombiquarium game application and the Mini-Game Framework to be used in its construction. In building both we will heavily rely on the Java API to provide services. Following are descriptions of the components to be built, as well as how the Java API will be used to build them.

2.1 Ramble On overview

The Ramble On design will be complimented with heavy reliance on the MiniGameFramework that has been provided for this project. Figure 2.1 specifies all the components to be developed and places all classes in home packages for the Ramble On design.

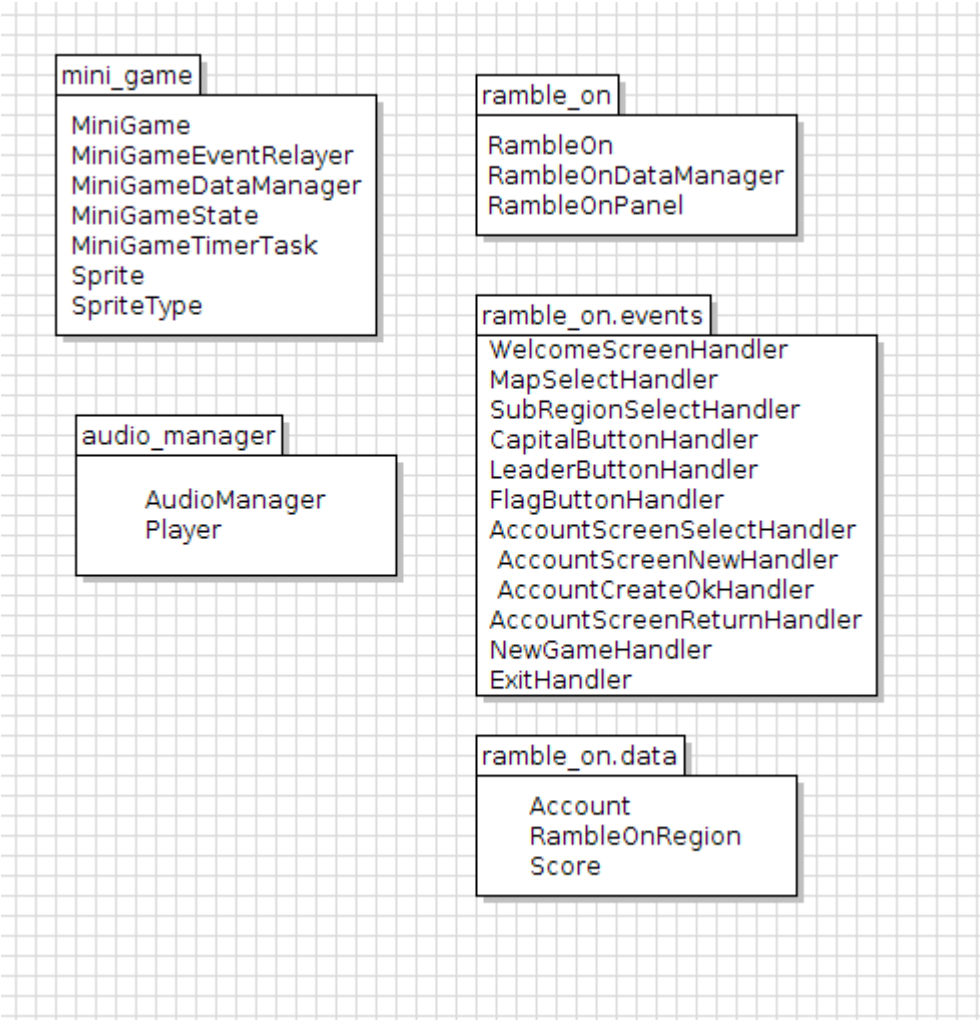

Figure 2.1: Design Packages Overview

2.2 Java API Usage

Both the framework and the mini-game application will be developed using the Java programming languages. As such, this design will make use of the classes specified in Figure 2.2.

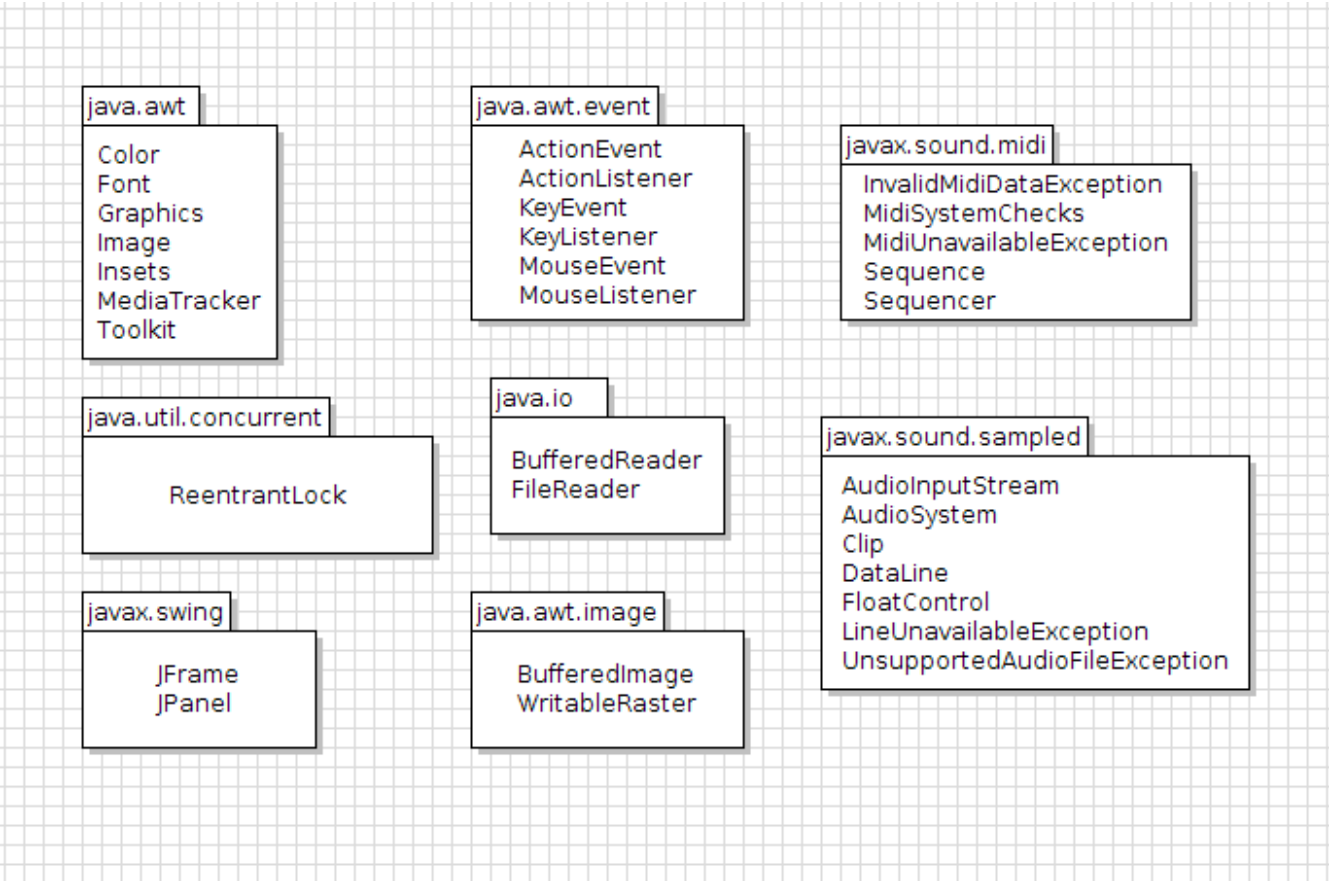

Figure 2.2: Java API Classes and Packages To Be Used

2.3 Java API Usage Descriptions

Tables 2.1-2.9 below summarize how each of these classes will be used.

Class/Interface	Use
Color	For setting the rendering colors for text and the progress bar
Font	For setting the fonts for rendered text
Graphics	For rendering text, images, and shapes to the canvas
Image	For storing image data
Insets	For changing component margins
MediaTracker	For ensuring synchronous image loading
Toolkit	For loading images

Table 2.1: Uses for classes in the Java API's java.awt package

Class/Interface	Use
ActionEvent	For getting information about an action event like which button was pressed.
ActionListener	For responding to an action event, like a button press. We will

	provide our own custom implementation of this interface.
KeyEvent	For getting information about a key event, like which key was pressed.
KeyListener	For responding to a key event, like a key press. We will provide our own custom implementation of this interface.
MouseEvent	For getting information about a mouse event, like where was the mouse pressed?
MouseListener	For responding to a mouse event, like a mouse button press. We will provide our own custom implementation of this interface.

Table 2.2: Uses for classes in the Java API's java.awt.event package

Class/Interface	Use
BufferedImage	For storing image data where pixel information can be accessed and changed.
WritableRaster	For changing pixel data in a BufferedImage. We'll use this to add transparency to pixels loaded with the color key.

Table 2.3: Uses for classes in the Java API's java.awt.image package

Class/Interface	Use
BufferedReader	For reading text files, we'll use this for loading some game data at startup.
FileReader	For reading files.

Table 2.4: Uses for classes in the Java API's java.io package

Class/Interface	Use
ReentrantLock	For ensuring only one thread has access to program data.

Table 2.6: Uses for classes in the Java API's java.util.concurrent package

Class/Interface	Use
JFrame	Provides the window for our GUI.
JPanel	Provides a canvas for the game to be rendered onto.

Table 2.7: Uses for classes in the Java API's javax.swing package

Class/Interface	Use
InvalidMidiDataException	Checks for errors in identifying Midi file
MidiSystemChecks	Checks midi file
MidiUnavailableException	Checks for errors in utilizing Midi file
Sequence	Evaluates midi data
Sequencer	Plays midi data

Table 2.8: Uses for classes in the Java API's javax.sound.midi package

Class/Interface	Use
AudioInputStream	Stream of audio input
AudioSystem	System of audio data
Clip	Segment of audio data
DataLine	Segment of data
FloatControl	Controls the data from the audio system
LineUnavailableException	Throws exception for invalid input
UnsupportedAudioFileException	Throws exception for invalid file

Table 2.9: Uses for classes in the Java API's javax.sound.midi package

3. Class-Level Design Viewpoint

As mentioned, the implementation will include the MiniGameFramework, which will be relied on heavily by the Ramble On application. In addition, the AudioManager class, which has also been provided, will be utilized. Some methods and design has been drawn from the RegionPickerSolution program as well.

The following UML Class Diagrams reflect this. Note that due to the complexity of the project, we present the class designs using a series of diagrams going from overview diagrams down to detailed ones.

More detail will be given to those that will be constructed from scratch, especially the RambleOn project itself. For all intensive purposes, all provided projects and code will be utilized without altering, unless unexpected updates or errors occur in later development that require a such a change.

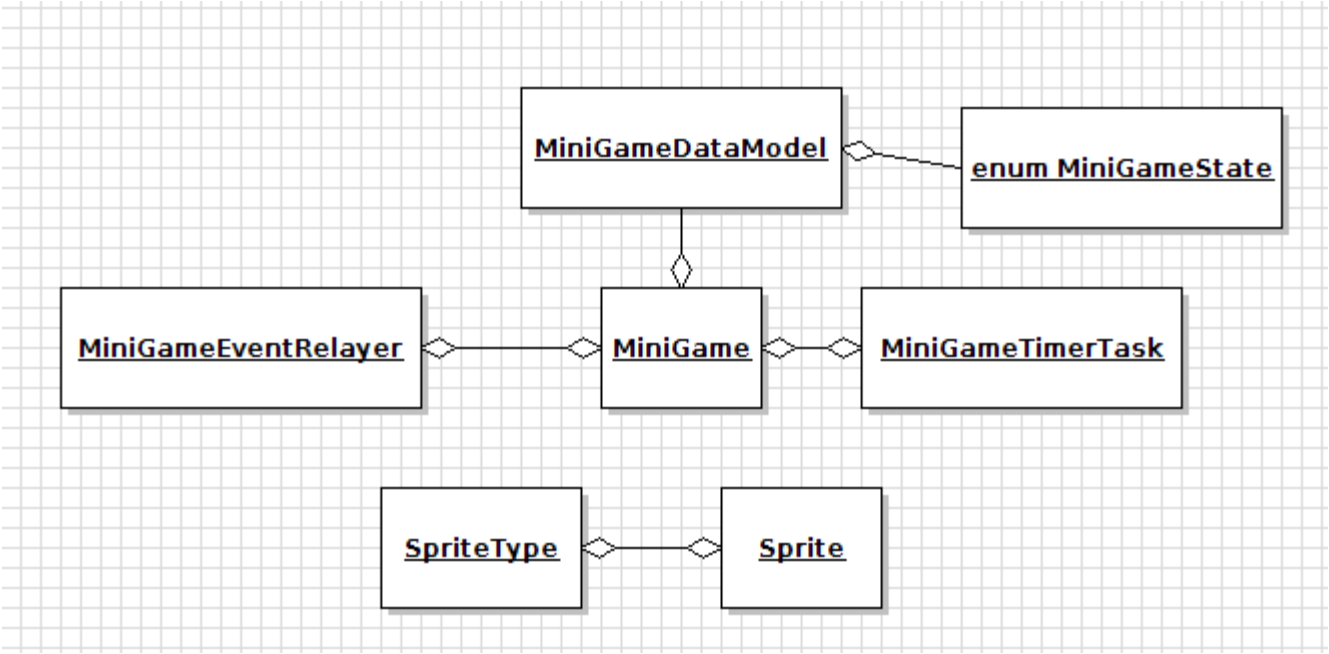

Figure 3.1: Mini-Game Framework Overview UML Class Diagram

A brief overview of the MiniGameFramework, provided with full license from the author, Richard McKenna for development in this application. The classes will be utilized with the intention of not editing any of the files. For a more detailed explanation of the framework, please reference the Zombiequarium(tm) Software Design Description by Richard McKenna (distributed by Debugging Enterprises(tm), November, 2011).

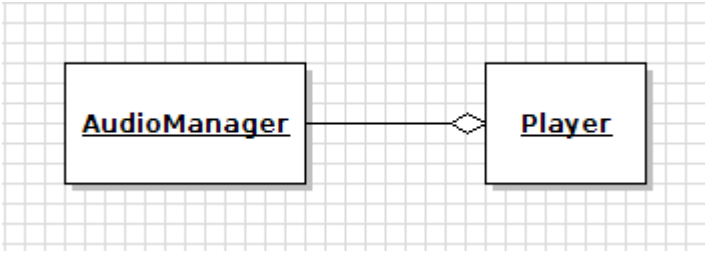

Figure 3.2 Audio Manager Overview UML Class Diagram

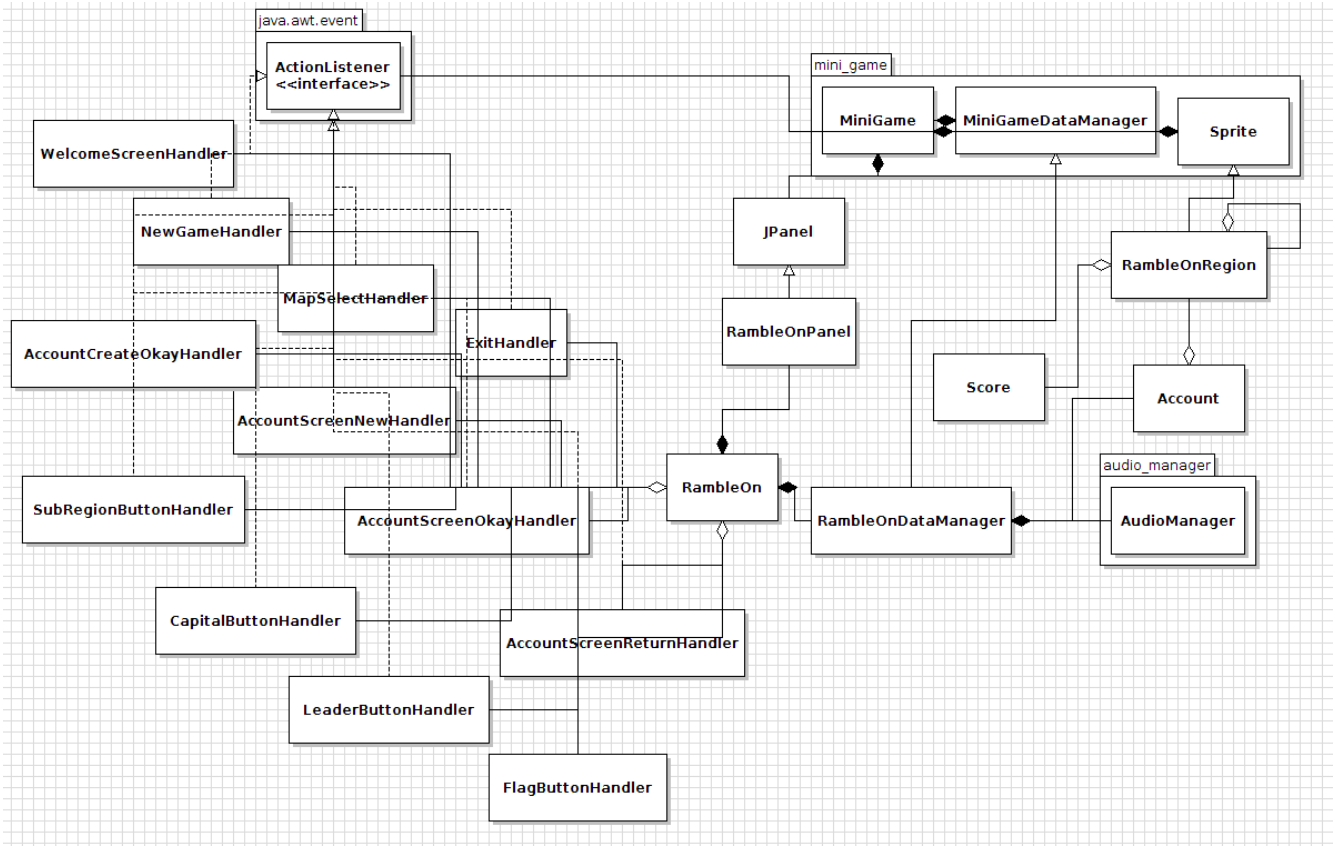

Figure 3.3: Ramble On Overview UML Class Diagram

A brief overview of the entire RambleOn project plans, showing the relationship of all of the to-be created classes and the direct hierarchies the utilize. On the left are the event handlers, and to the right are the data and GUI implementations. Further explanation and specific clarifications follow immediately:

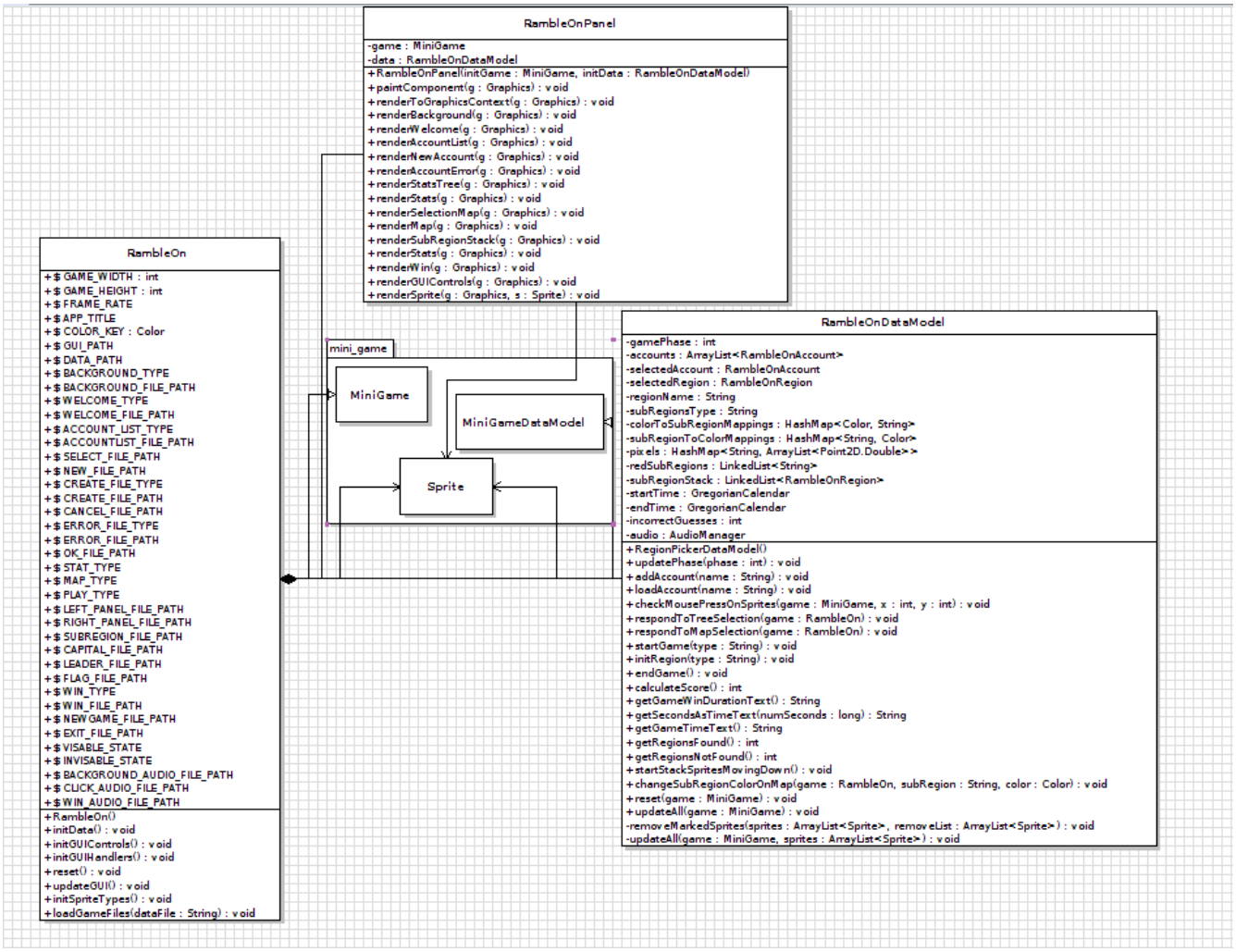

Figure 3.4: Detailed RambleOn, RambleOnPanel, and RambleOnDataModel UML Class Diagram

The RambleOn main program inherits the framework from the MiniGame class, as such with RambleOnDataModel with MiniGameDataModel, and RambleOnPanel with Jframe.

As of now, the RambleOn main program holds all of the paths for the GUI and audio locations, and adds these elements to the data and displays of the RambleOnDataModen and RambleOnPanel. However, it is desired to have a potentially cleaner solution for managing this immense amount of Strings, both for ease of reading by programmers, and simple editing by code-clueless graphic designers and composers. The potential for a separate data file is lucrative, but will ultimately depend on elements on data saving as a whole that will be further explained in section five.

The RambleOnPanel is structured to consistently update, checking with data from RambleOnDataModel to be sure as to what exactly to display. The RambleOnDataModel will rely on user input to keep track of the user's choices as per account creation, stat tree viewing, map selection, and ultimately gameplay. The exact mechanics of such are still hazy, but are intended to be sectioned off into “phases” of gameplay, akin to the RambleOn class's system of seeing what GUI and Sprite elements to display. A way to connect the two systems seems like a potential possibility, but and independent system has been theorized as to not risk counting my blessings, and ending up with a

flawed plan and no back-up.

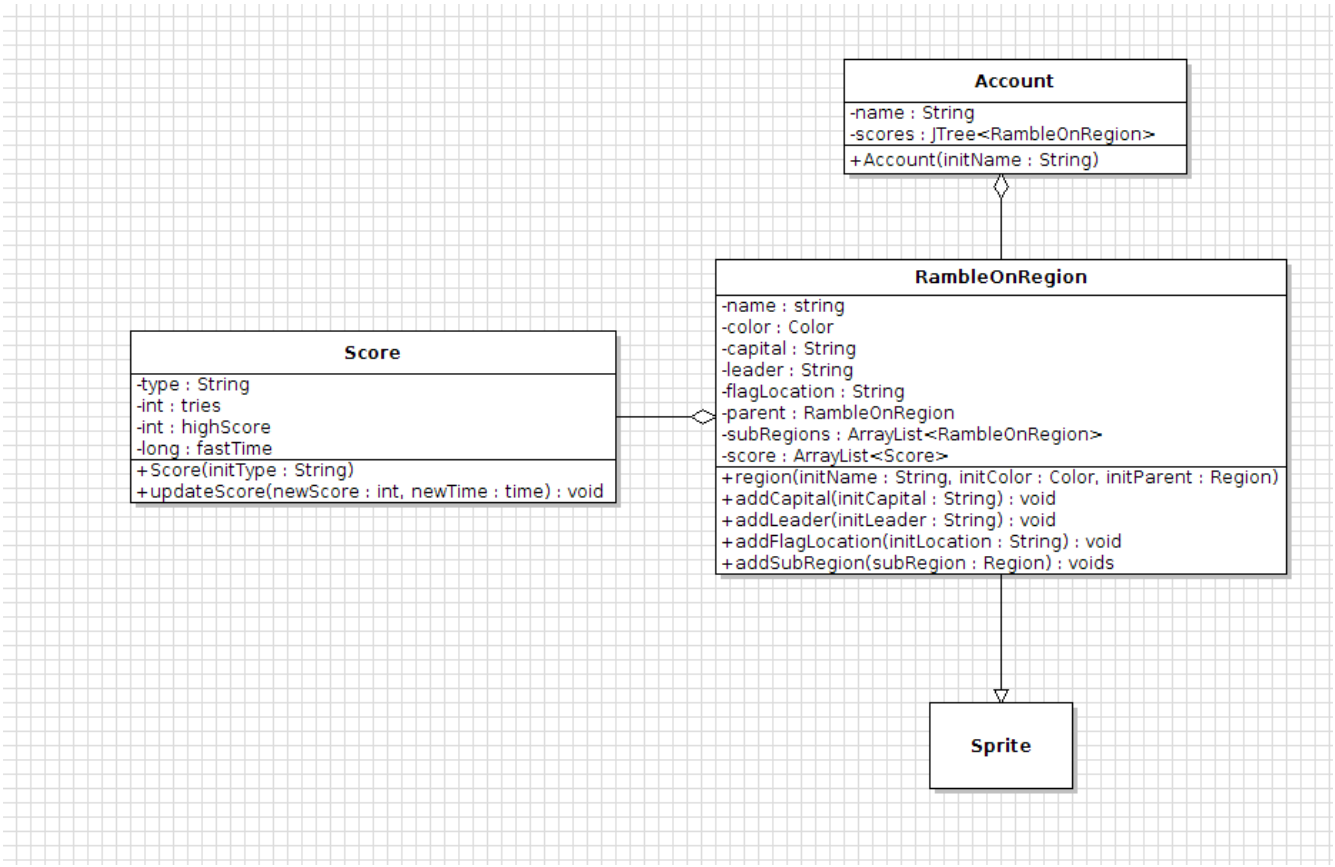

Figure 3.5: Detailed Account, RambleOnRegion, and Score UML Class Diagrams

An Account is a collection of RambleOnRegions assigned to a name. Adding a separate tree of RambleOnRegions for each account was planned, as the RambleOnRegion is intended to have an Score object, which will allow for dual usage in both the score viewing phase, the area selection phase, and the ultimate gameplay. There lies the possibility of too much data being stored, if the accounts were to ever get to a large amount, but in the scope of this project, this will be the most efficient and simple.

If on a later date one was to edit this for such a wider audience, one should just keep a tree of a modified object containing region names and an array of scores for each gameplay type. The now excluded data should then be stored in an reference tree that will be referenced for retrieving data. The constant need to reference this unchangeable tree may present problems though, especially if in the context of a more widely used application. (Though due to the lack of network capabilities, one would probably never run into this problem, lest the application was distributed to a classroom environment, or an extremely large household.)

These assorted classes are intended to make the storing and retrieval of game data easy and universal. However, such plans are also extremely flexible, depending on further implementation and guidelines. The likely utilization of XML files to store data raises further questions of how to access and save files, but until then, and as such, these plans are created assuming that further information and/or approval will be given at a later date.

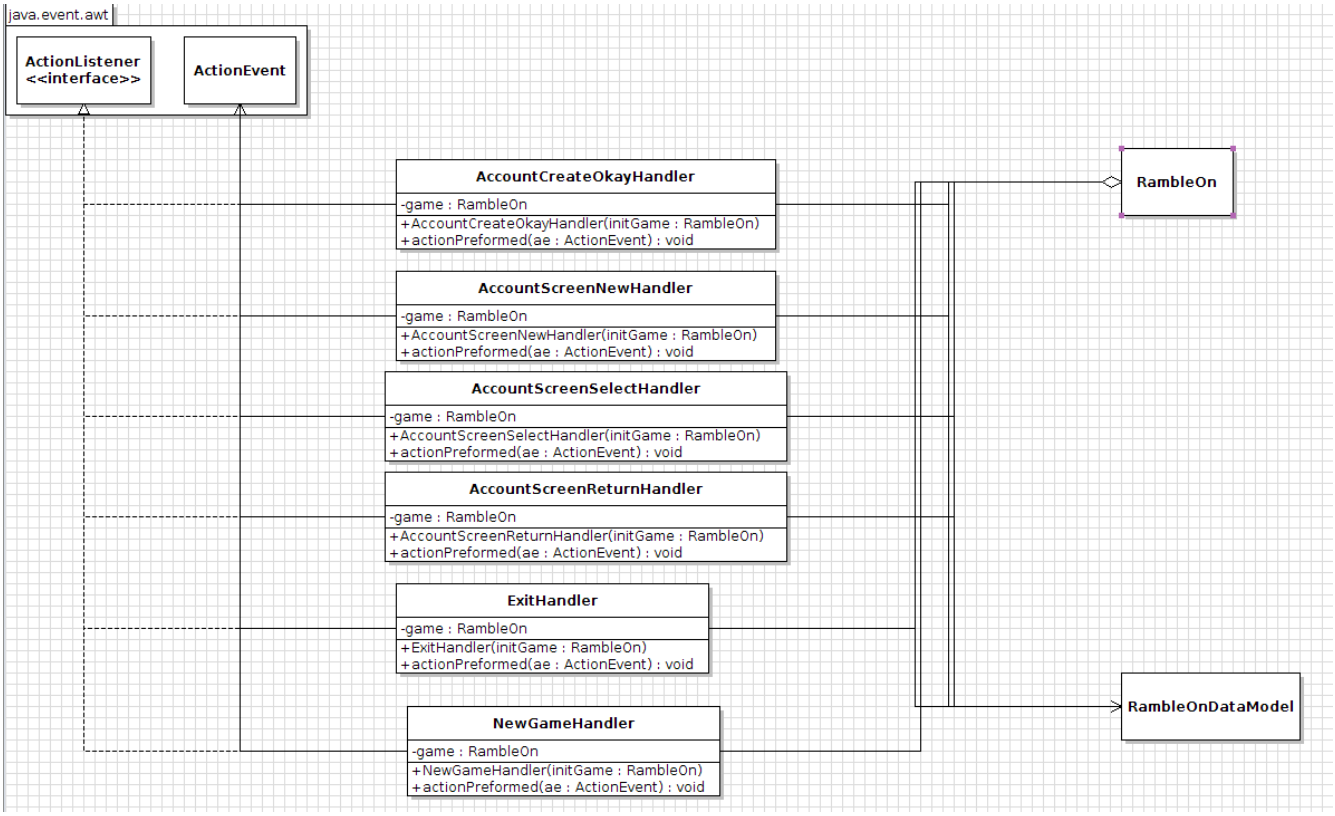

Figure 3.6: Detailed AccountScreenSelectHandler, AccountScreenNewHandler, AccountCreateOkHandler, AccountScreenReturnHandler, NewGameHandler, and ExitHandler UML Class Diagrams

Part One of the assorted handlers that will be utilized for this program. All of which inhered the ActionListener interface, and utilize the ActionEvent class.

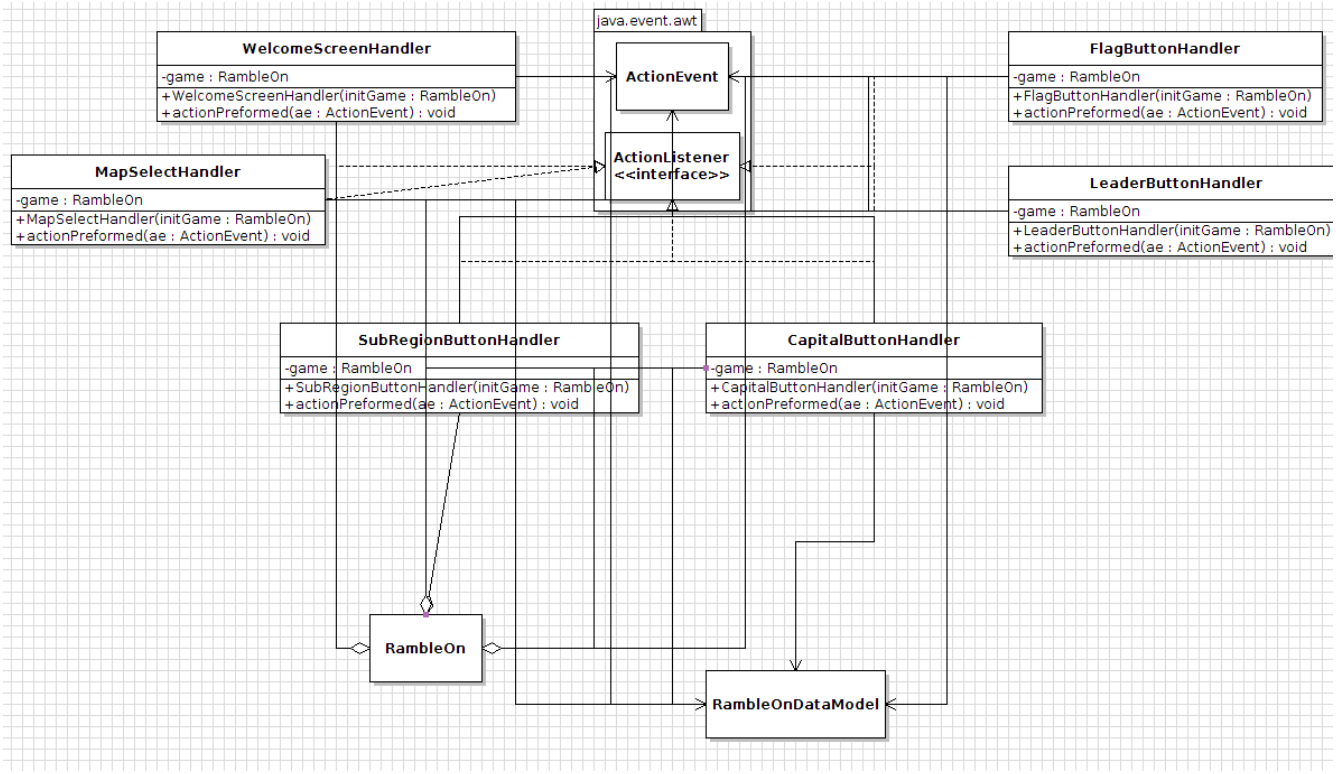

Figure 3.7: Detailed WelcomeScreenHandler, MapSelectHandler, SubRegionSelectHandler, CapitalButtonHandler, LeaderButtonHandler, and FlagButtonHandler UML Class Diagrams

Part Two of the assorted handlers that will be utilized for this program. All of which inhered the ActionListener interface, and utilize the ActionEvent class.

4. Method-Level Design Viewpoint

Now that the general architecture of the classes has been determined, it is time to specify how data will flow through the system. The following UML Sequence Diagrams describe the methods called within the code to be developed in order to provide the appropriate event responses.

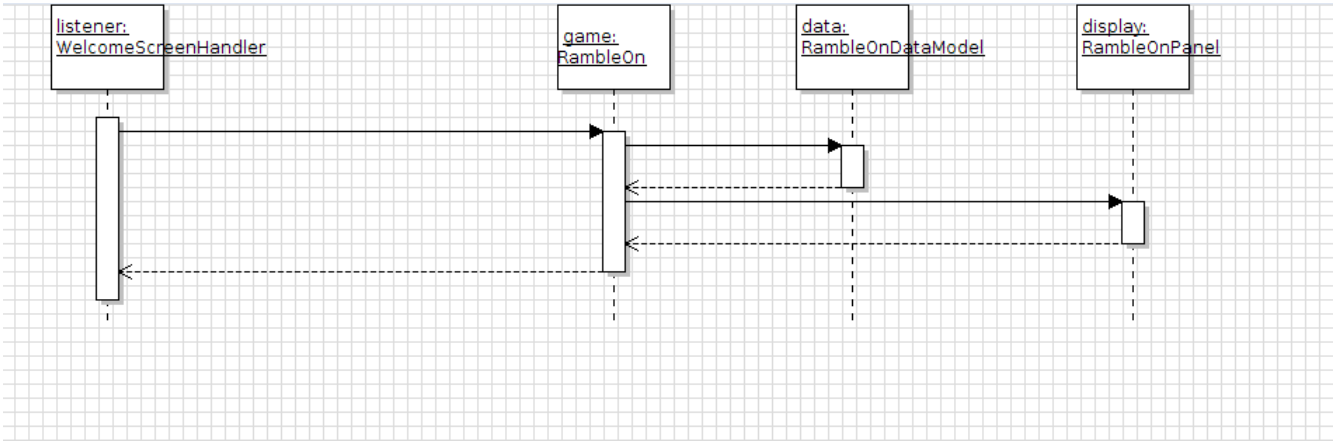

Figure 4.1: WelcomeScreenHandler UML Sequence Diagrams

The WelcomeScreenHandler checks for a sole click from the user, after which it refreshes the display to the Account Display, and tells the RambleOnDataManager to switch to such a phase.

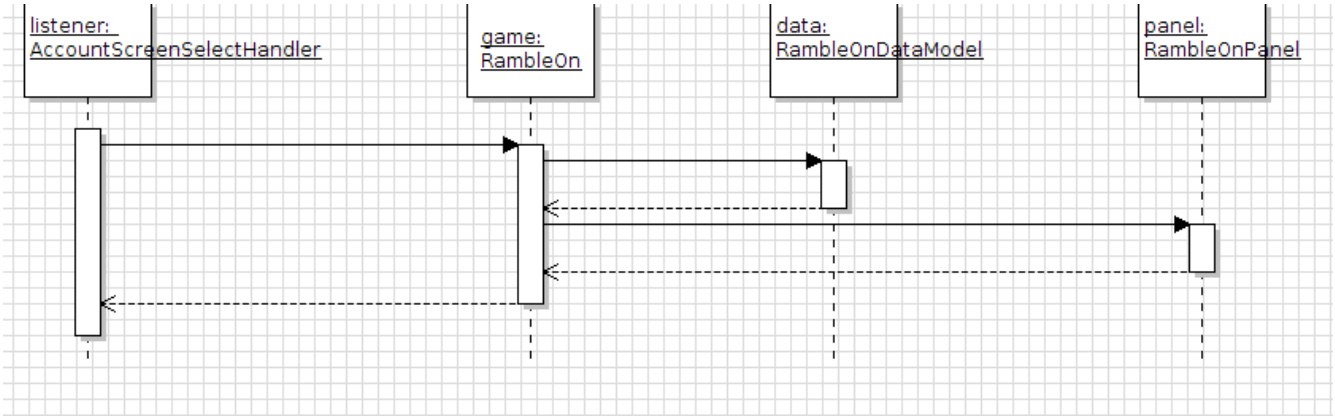

Figure 4.2: AccountScreenSelectHandler UML Sequence Diagrams
The AccountScreenSelectHandler is called on the click of the "Select" button of the Account Screen Selection phase. This phase should automatically select the top element of the account list if available, or disable the button to prevent this handler from being called without sufficient data.

After an account is selected, RambleOnDataManager should retrieve the Account object from the list, and load it before going on to the Stat Tree selection.

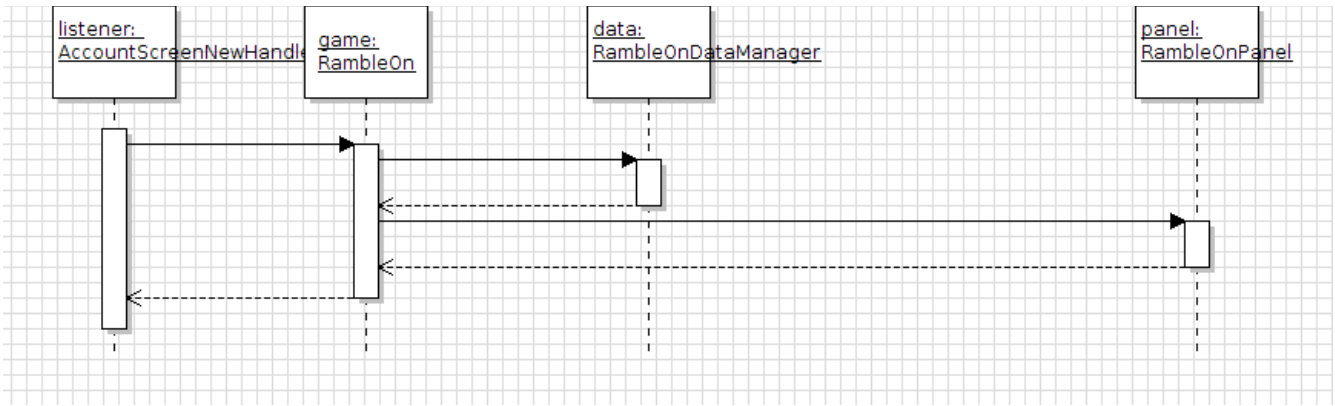

Figure 4.3 AccountScreenNewHandler UML Sequence Diagrams
The AccountScreenNewHandler is called when a user seeks to create a new account for use in the game. This handler should call the New Account Phase, which displays a box giving instructions to type a name in a text field, and to then either confirm or return back to the Account Screen list.

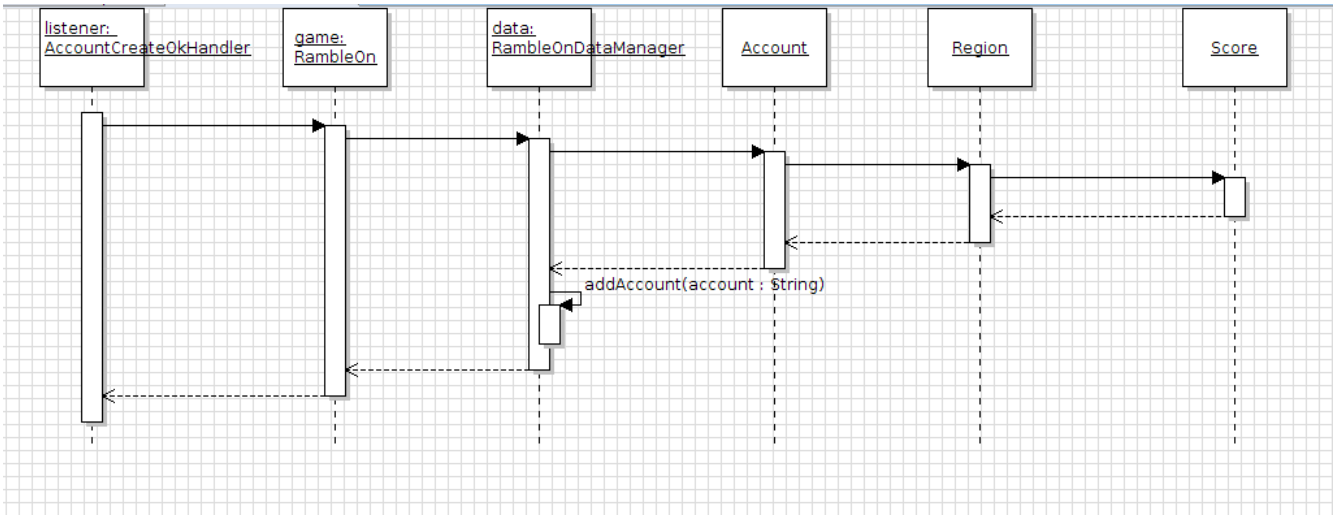

Figure 4.4: AccountCreateOkHandler UML Sequence Diagrams
After calling the AccountCreateOkHandler, the input in the textbox should be retrieved and checked to make sure that it contains a valid input, that is, a string of symbols that are not already listed as an account. If not, the program should display an error box displaying the error, and with a button that leads to the AccountScreenReturnHandler.

If so, the program should create a new Account object with the name, and add it to the Accounts list. The program should then load the Jtree of RambleOnRegions, and utilize them in the Stat Tree phase.

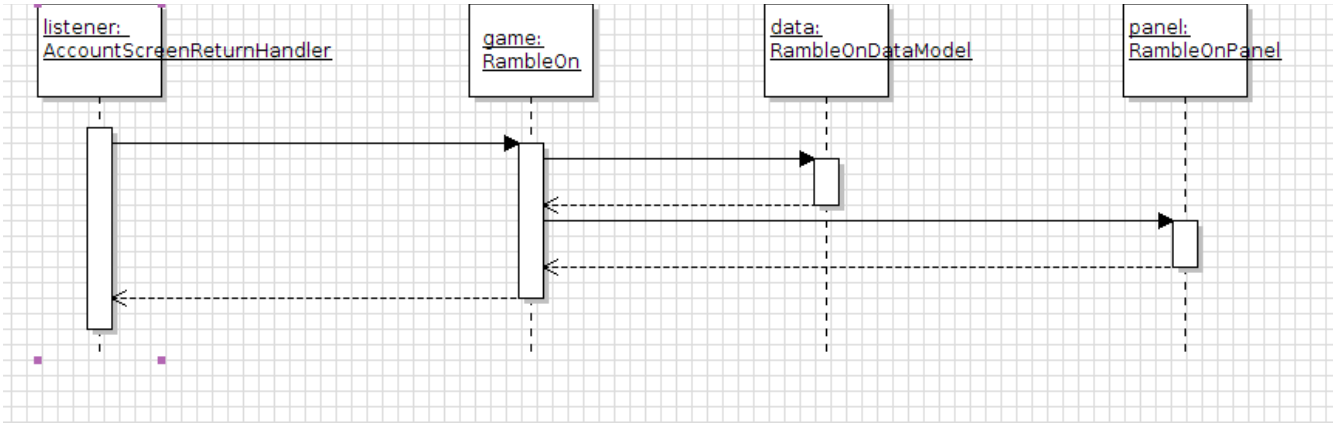

Figure 4.5 AccountScreenReturnHandler UML Sequence Diagrams UML Sequence Diagrams
The AccountScreenReturnHandler will be utlized in multiple contexts, including returning to the Accounts List phase after canceling a new account creation, or entering an invalid input. This handler will also be used in the subsequent phases, so that the user can easily start a new game without having to restart the entire game.

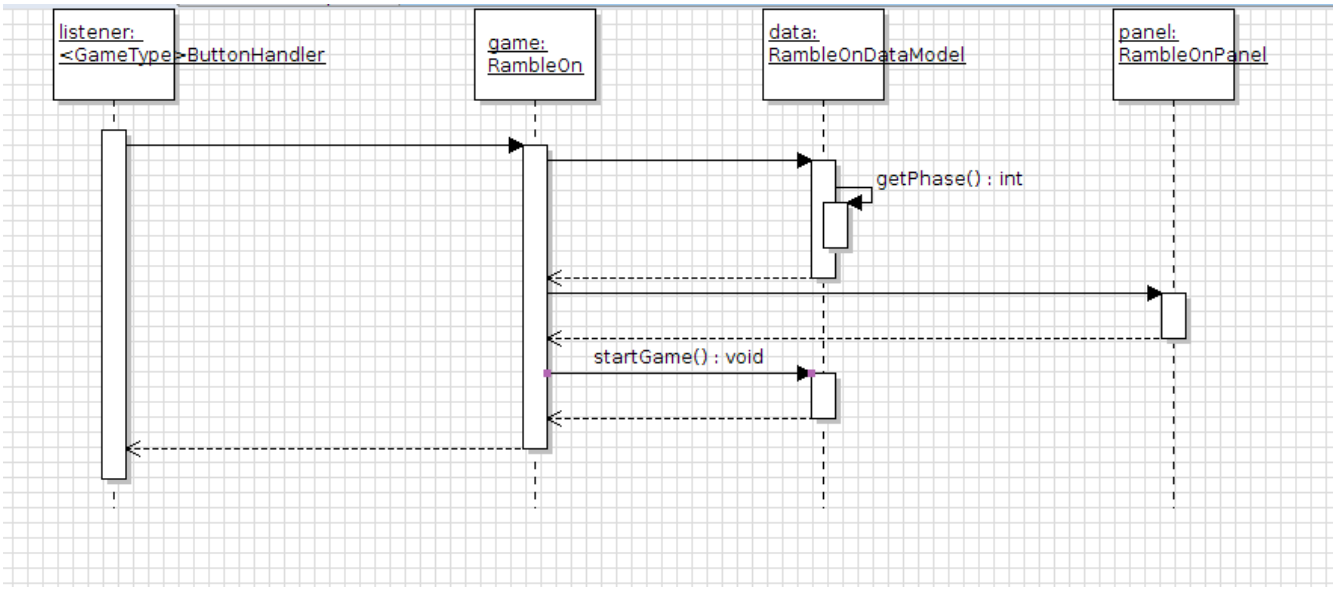

Figure 4.6: SubRegionButtonHandler, CapitalButtonHandler, LeaderButtonHandler, FlagButtonHandler UML Sequence Diagrams

These handlers will be active during the Stat Tree and the Map Selection phases, and will either display the saved statistics for a selected region's gameplay in Stat Tree, or start the gameplay for the selected region in Map Selection.

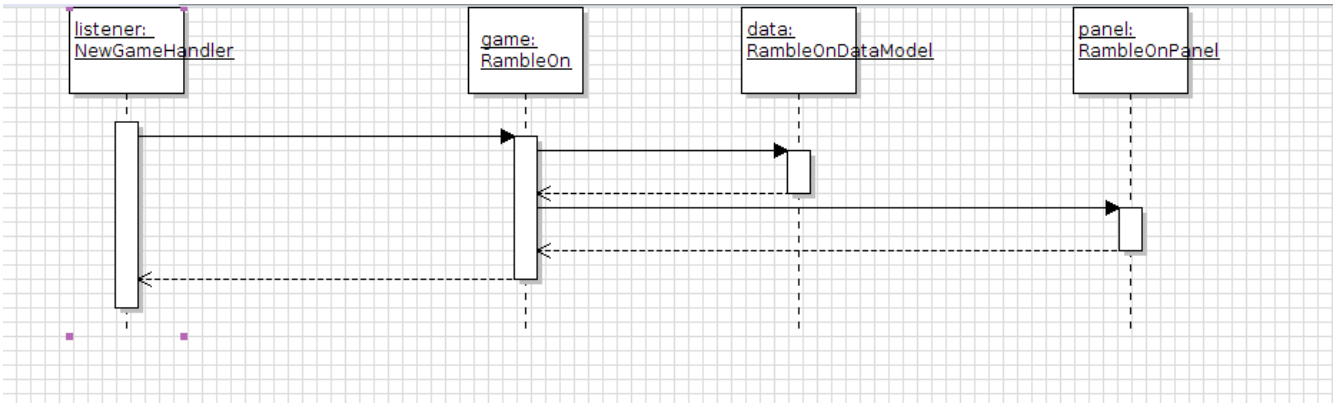

Figure 4.6: NewGameHandler UML Sequence Diagrams
This handler will be active during the Stat Tree phase, and will allow the user to access the Map Select phase to browse the availible levels to play, and to start a game with one.

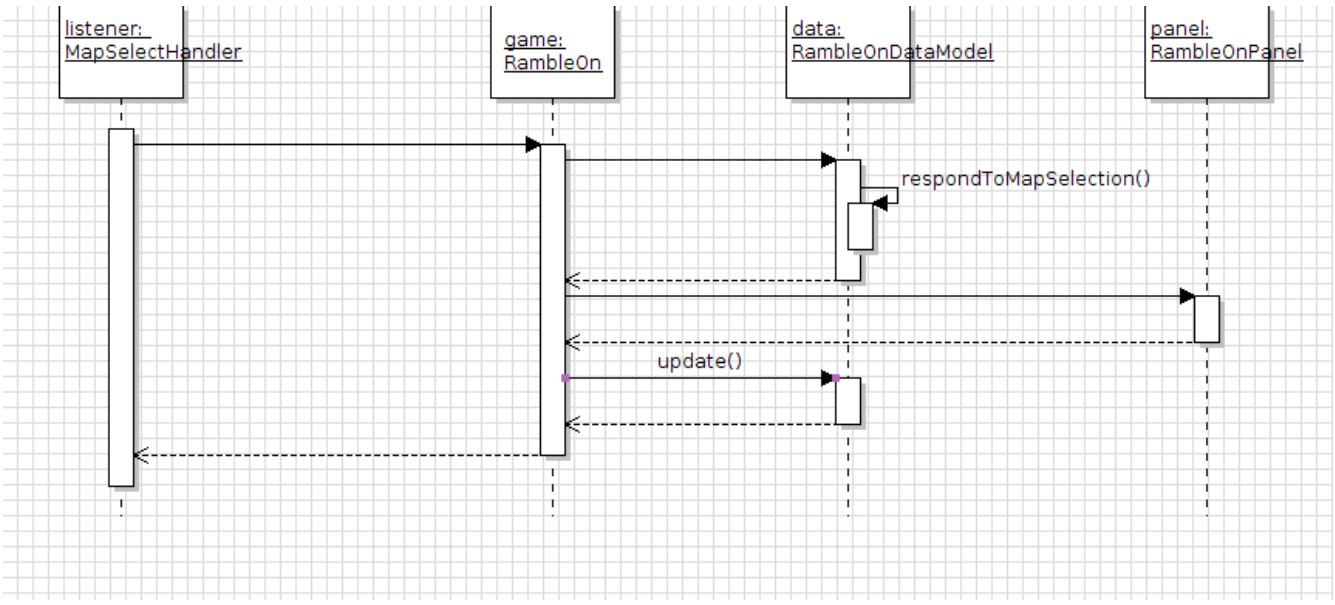

Figure 4.7: MapSelectHandler UML Sequence Diagrams

This handler may be redundant, but I prefer to be on the safe side. This handler would either evaluate clicks on any Map graphics, and process them according to the phase. In Map Select phase, it would see what button was clicked, and see if it correlates with the location clicked. In the Game Play phase, it would process clicks as per the rules of the game.

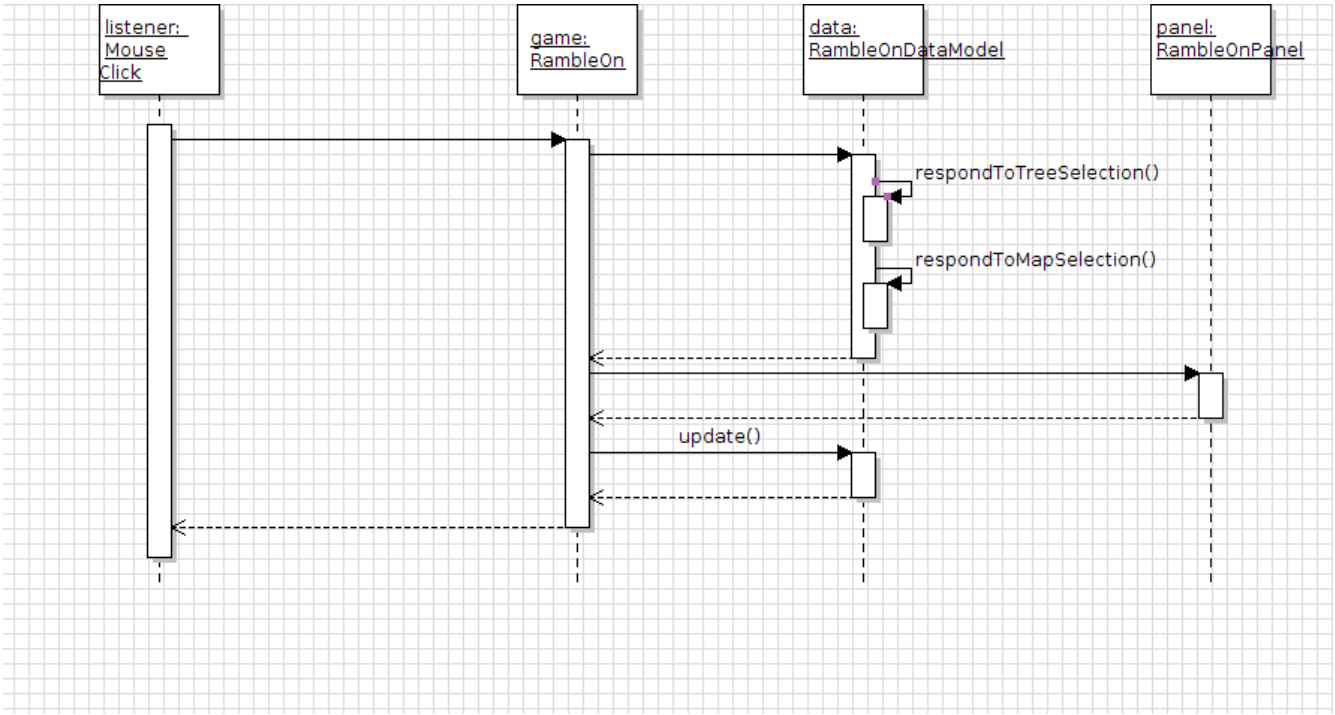

Figure 4.8: Mouse Click UML Sequence Diagrams

This handler will control additional mouseclicks as per the MiniGameFramework, and may be further relied on in case MapSelectHandler is redundant.

5. File Structure and Formats

Note that the Mini Game Framework will be provided inside MiniGameFramework.jar, a Java Archive file that will encapsulate the entire framework. This should be imported into the necessary project for the Ramble On application and will be included in the deployment of a single, executable JAR file titled RambleOn.jar.

Note that all necessary data and art files must accompany this program. Images will be created in GIMP, and the number and nature of such will be flexible, and potentially evolve in complexity if extra time for customization is necessary. Sound files will be "MIDI" files, as per the provided package, and will include, at the minimum, a gameplay song, victory song, and sound effect. Again, if additional time is found, further customization will expand on this. One possibility is to utilize the national anthems of the listed countries for each different level, but this will ultimately depend on an unknown save file issue, and again my time and sanity.

Figure 5.1 specifies the necessary file structure the launched application should use. Note that all necessary images should of course go in the image directory.

[Ramble On Folder]
- RambleOn.jar
- [Set Up Folder]
  - RambleOnGameData
  - [Images Folder]
  - [Sound Folder]

Figure 5.1: Ramble On File Structure

The RambleOnGameData file provides the file and state names for all sprite states in the game, in addition to player accounts, and other set-up information to be used by the main program.

The file is a collection of data, the specifics of which to be used have been promised to be further explained at a further date as per the curriculum of the course. Ideally, the document should include the following:

- Set up for sprites and GUI, including states and image locations
- Set up for audio, including file locations
- Information about the playable regions, possibly linking to further development with XML files, as utilized in HW #2
- Some sort of way to easily store the information stored in player accounts, as well as the ability to easily retrieve this from the main program
- Possible consolidation of the String data stored as static variables in the main RambleOn program.
- :Possible way to add custom music (ex: national anthems) for each level

Figure 5.2: Ramble On Saved Data Information

6. Supporting Information

Note that this document should serve as a reference for those implementing the code, so we'll provide a table of contents to help quickly find important sections.

6.1 Table of contents

6.2 Appendixes

N/A