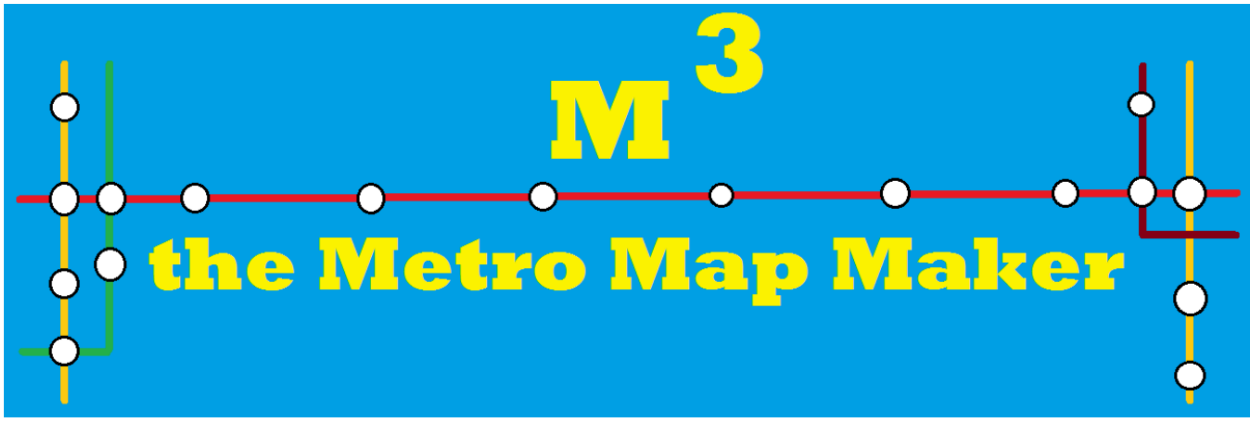
MetroMapMaker

Software Design Description



Author: Qi Yuan Fang
CSE 219 L02
Fall 2017

Abstract: This document describes the software design for MetroMapMaker, an application that provides users with tools enabling them to build graphical representations of city subway systems. MetroMapMaker also provides tools to calculate the best journey from any existing station to another existing station.

Based on IEEE Std 1016 – 2009 document formal

Table Of Contents

1 Introduction

This is the Software Design Description for the MetroMapMaker application. Please note that the document format is based on the IEEE Standard 1016-2009 recommendation for software design.

1.1 Purpose

The purpose of this document is to be a blueprint for the making of the MetroMapMaker application. This document will use UML diagrams to detail the packages, classes, instance variables, class variables, and method signatures that will be used to build the application. There are also UML sequence diagrams that help specify how objects will interact after the application has been initialized, to user interactions.

1.2 Scope

For this project, users will be able to create and edit custom subway maps. An important aspect of this application is its emphasis on ease of use. All exported subway maps created by this application are saved in a common export format to ensure compatibility.

1.3 Definitions, acronyms, and abbreviations

Framework – In object oriented languages, frameworks are collections of classes & interfaces that provide a basis for building applications or other frameworks that have common needs that the framework can provide for.

GUI- GUI stands for Graphical User Interface, examples of GUI elements are the window of the application itself, buttons, combo boxes, etc. that allow users to interact with the program.

IEEE – IEEE stands for the Institute of Electrical and Electronics Engineers

JavaScript – JavaScript is a widely used scripting language of the web. It can be loaded and executed when a page on the web loads which allows dynamic generation of page content in the DOM.

Stylesheet- Stylesheets are static text files used by HTML pages that control style components on web pages.

UML – Unified Modeling Language, is a standard set used for design phase of application dev

Use Case Descriptions – A formal format used to specify how users will interact with a system.

1.4 References

IEEE Std 830 TM 1998 (R2009) – IEEE Recommended practice for Software Requirements Specification

1.5 Overview

The following SDD document's purpose is to provide a definition for how the **Metro Map Maker** application should look and how it should function. The purpose of this document is to specify how we will construct the software in question using UML. Section 2 of this document will provide the package- level viewpoint which specifies the packages and frameworks that will be designed. Section 3 will provide a class-level viewpoint which using UML Class Diagrams, we will specify how the classes used in this application will be constructed. In section 4, we will provide the Method-Level System Viewpoint, which describes how methods will interact with one another. The purpose of section 5 is to provide deployment information such as file structures and formats to use. Section 6 provides a Table of Contents, an Index, and References. For this SDD, all UML Diagrams were created using the VioletUML editor.

2 Package-Level Design Viewpoint

As previously mentioned, this design encompasses the MetroMapMaker application and the frameworks used in its construction. In building this application, we will utilize the Java API which provides us with convenient to use tools which will more easily enable us to build MetroMapMaker.

2.1 Metro Map Maker overview

Figure 2.1 specifies the components that will be developed and places all classes into their respective home packages. **The provided frameworks (DesktopJavaFramework, JTPS, PropertiesManager) have also been included in this diagram.**

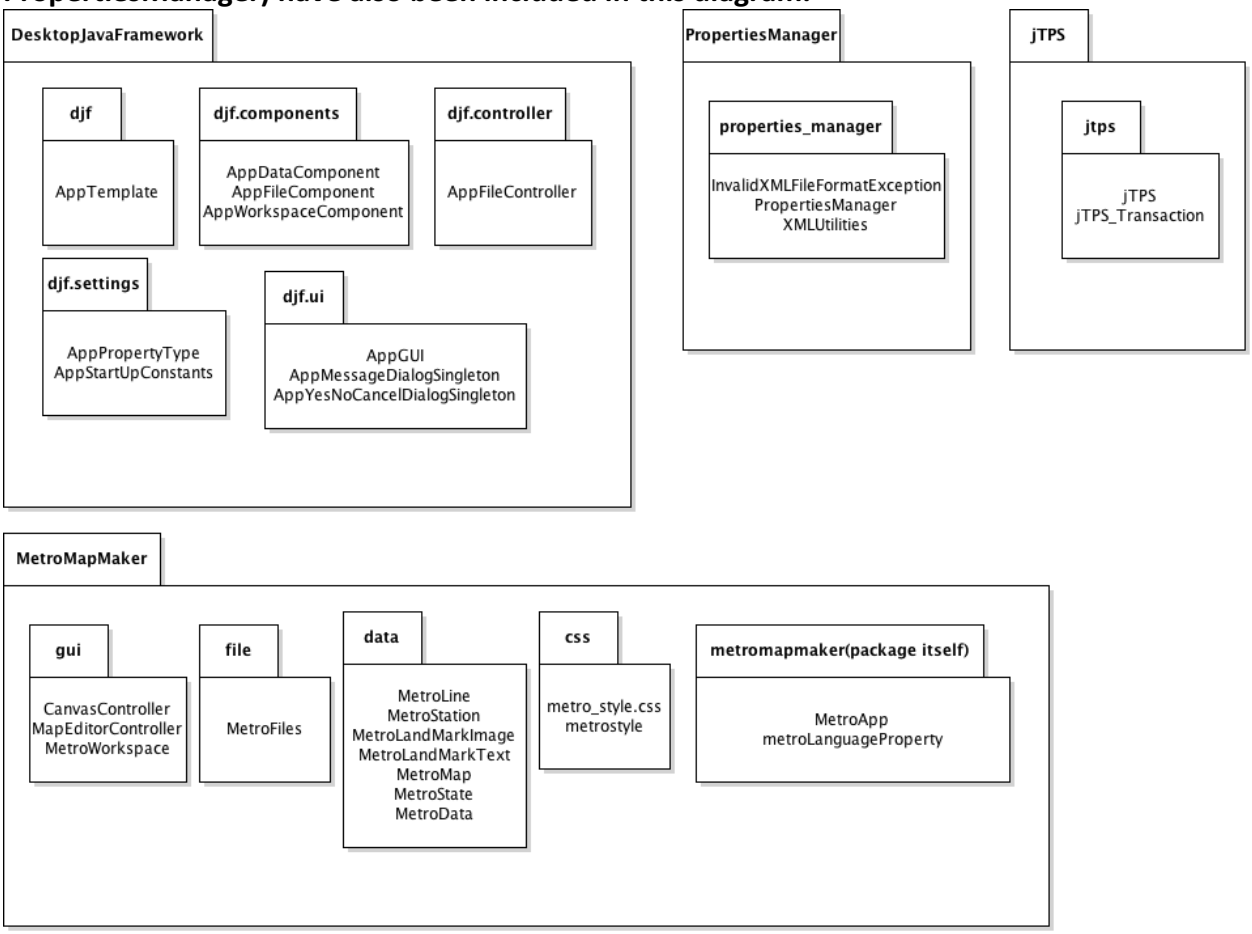


Figure 2.1 : Design Packages Overview

Brief description and explanation of design rationale (More detailed explanation in 3.1):

The gui sub package contains the MetroApp class which is the application class itself, the MetroWorkspace which is the workspace of the application that deals with static aspects of the UI and setting up controllers. The controller classes of this application are CanvasController and MapEditorController. CanvasController's purpose is to provide controllers for user interactions. The CanvasController deals with canvas interactions while MapEditorController deals with actions relating to the map itself such as adding elements, getting routes, adding background

images etc. The MetroFiles class inside the sub package file is responsible for loading and saving maps utilizing JSON. The MetroLine, MetroStation, MetroLandMarkImage, MetroLandMarkText are map elements represent train lines, train stations, and image & text landmarks respectively. They contain member variables and objects and methods that help in the functionality of the program. The MetroState enum class provides application state enum states that gives the application input on what state the program is in to enable/disable certain UI elements. The MetroData class is the data management component of the application.

The rationale behind the design of this application in this particular way is that the application is in ways similar to previous homework projects we have done. Since the application we are using is similarly laid out to the previous ones, I have decided to utilize a similar workspace component. Likewise, the controller classes in this application are similar to those already used in previous assignments due to similarities such as draggable objects on a canvas, dragging objects on a canvas etc. Similarly, the data management and file management classes are similar. The saving is done in JSON in a similar way (with modifications due to difference of objects being saved). The pros to this approach to application design is familiarity and speed of development as many of the frameworks and design patterns/structures in this application are already familiar to me. The cons is that this approach is rather restrictive as it forces the application developer to develop in a narrower mindset, one that is similar to the previous assignments. I considered other different application architectures that were totally different from the one I ended up choosing but I decided the familiarity was worth the tradeoff. Design wise, I decided to make the Metro Lines a collection (ArrayList) of lines which connect stations to make sure that the user can easily drag stations around and have the “Line” move with the station. Furthermore, the “Lines” contain ArrayList of the station themselves and this makes it easier to find optimal routes. I decided to go this route instead of a different approach, having a single line because this provides greater visual appeal and user friendliness.

For information on how the subsystems interact with each other, please view section 3.1 Overview diagram or the detailed explanation proceeding it.

2.2 Java API Usage

The frameworks (DesktopJavaFramework, JTPS, PropertiesManager) and the MetroMapMaker application make use of the Application Program interface of the Java programming language. Because of this, the design of this application will use the classes specified in Figure 2.2.

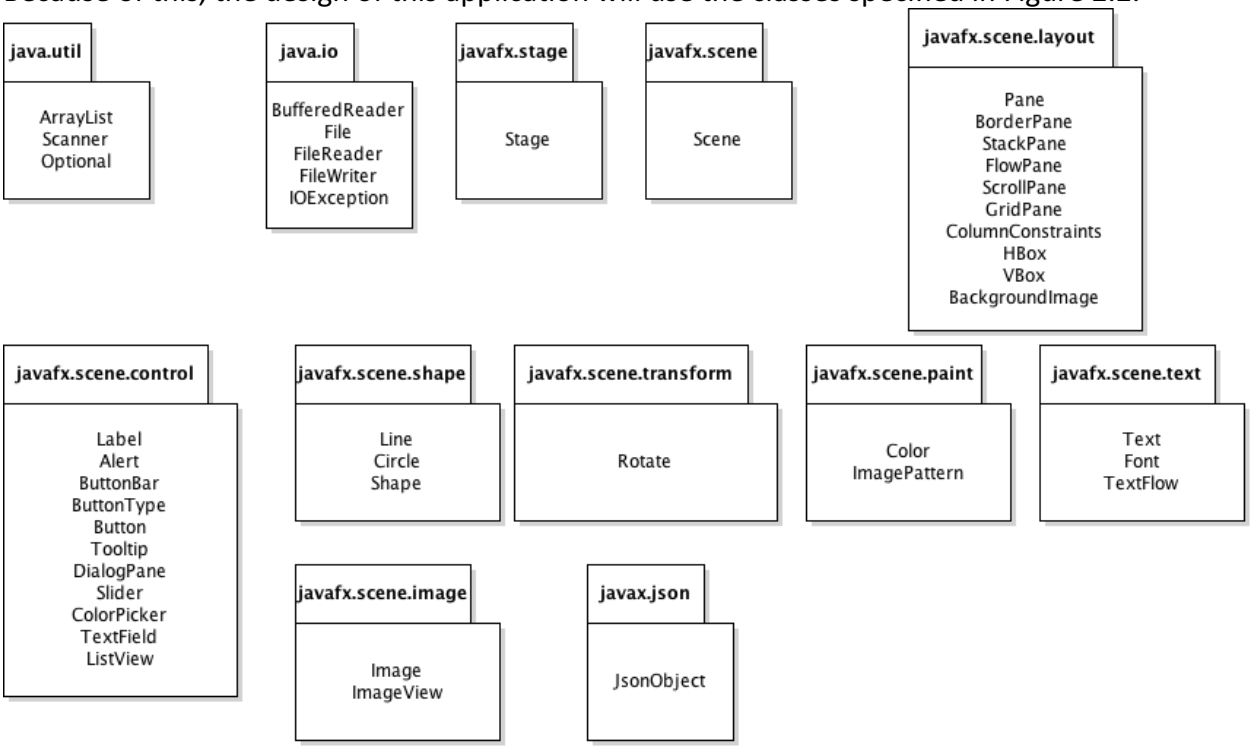


Figure 2.2: Java API Classes and Packages To Be Used

2.3 Java API Usage Descriptions

Tables 2.1-2.12 below summarize how each of the classes of the Java API that are utilized in this application and its used frameworks will be utilized.

Class/Interface	Use
ArrayList	For jTPS (Undo/Redo)
Scanner	For file I/O
Optional	For alert options

Table 2.1: Use cases for the classes in the java.util package

Class/Interface	Use
BufferedReader	For file I/O
File	For file I/O
FileReader	For file I/O
FileWriter	For file I/O
IOException	For file I/O exception

Table 2.2: Use cases for the java.io package

Class/Interface	Use
Stage	For application window

Table 2.3: Use cases for the javafx.stage package

Class/Interface	Use
Scene	For application scenes

Table 2.4: Use cases for the javafx.scene.scene package

Class/Interface	Use
Pane	For the “Canvas” of the application
BorderPane	For UI purposes
StackPane	For UI purposes
FlowPane	For UI purposes
ScrollPane	For scrollable map
GridPane	For layout of toolbars
ColumnConstraints	For layout of toolbars
HBox	For layout of toolbars
VBox	For layout of left bar
BackgroundImage	For the addition of background images to maps

Table 2.5: Use cases for the javafx.scene.layout package

Class/Interface	Use
Label	For labels, line names, station names
Alert	For alerts such as
ButtonBar	For custom alerts such as the Welcome popup
ButtonType	For custom alerts such as the Welcome popup
Button	For button controls in application
Tooltip	For to provide
DialogPane	For dialogs in application
Slider	For selecting line thickness
ColorPicker	For picking colors in application
TextField	For text entry (Metro Line Details etc..)

ListView	For list view of stations in popup dialogs

Table 2.6: Use cases for the javafx.scene.control package

Class/Interface	Use
Line	For adding metro lines
Circle	For adding representations of stations
Shape	Parent of Line and Circle used to aggregate both types in collections.

Table 2.7: Use cases for the javafx.scene.shape package

Class/Interface	Use
Rotate	For rotation of labels

Table 2.8: Use cases for the javafx.scene.transform package

Class/Interface	Use
Color	For color picker, station fill colors, background color, text color
ImagePattern	For Image background

Table 2.9: Use cases for the javafx.scene.paint package

Class/Interface	Use
Text	For setting text of labels
Font	For changing font of text in the map
TextFlow	For clickable text in welcome dialog

Table 2.10: Use cases for the javafx.scene.text package

Class/Interface	Use
Image	For adding images to the map
ImageView	For adding images to the map (JavaFX container object for Image)

Table 2.11: Use cases for the javafx.scene.image package

Class/Interface	Use
JsonObject	Used for saving of data by MetroFile class in JSON format

Table 2.12: Use cases for the javax.json pacakage

3 Class- Level Design Viewpoint

The design of the MetroMapMaker application will utilize both the MetroMapMaker package and the provided frameworks (DesktopJavaFramework, JTPS, PropertiesManager) . The UML diagram provided below reflects the interconnectedness of these packages. The class diagrams are presented going from overview diagrams to more detailed ones. However, as only the MetroMapMaker package is modified in any way from what is given (DesktopJavaFramework, JTPS, PropertiesManager), the diagrams provided in section 3 are of those that pertain to MetroMapMaker. For the overview diagram of MetroMapMaker, detailed package level view of frameworks used are illustrated. For detailed class level diagrams only the classes that are used in the frameworks are visualized for readability purposes.

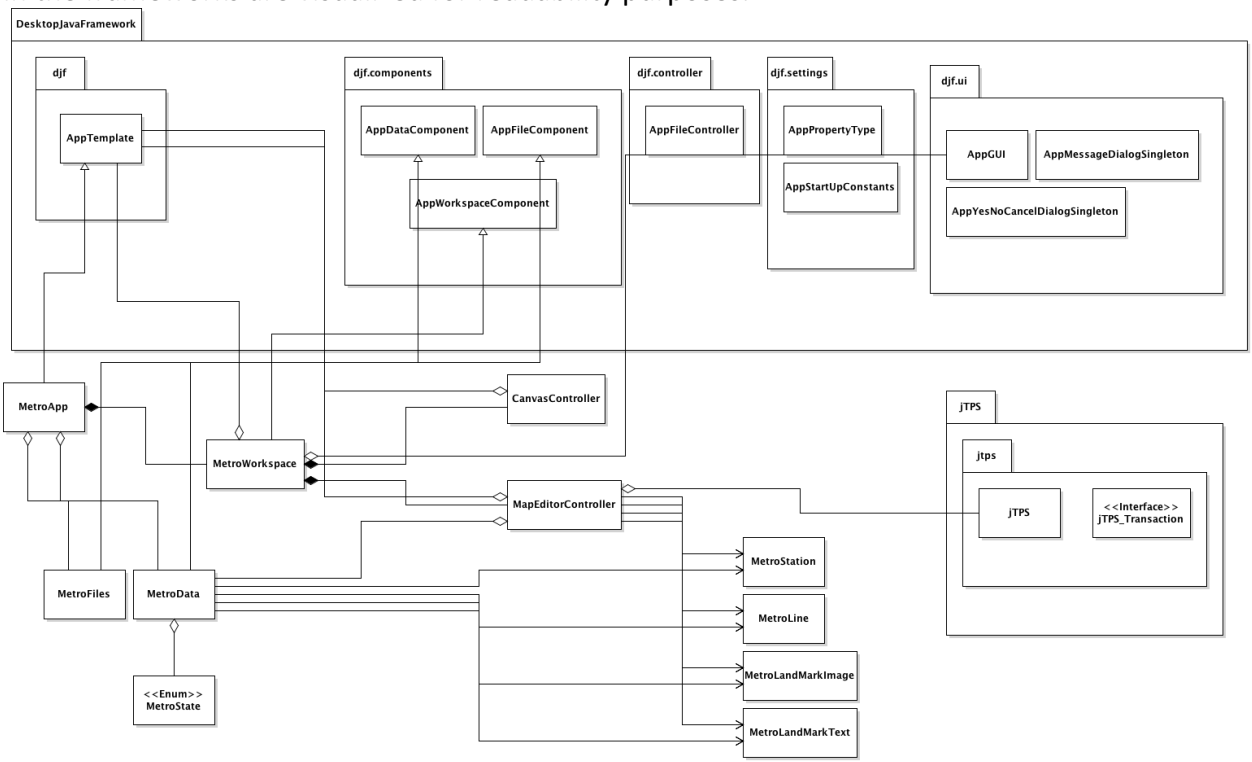


Figure 3.1 MetroMapMaker Overview UML Class Diagram

Design diagram explanations:

The application MetroMapMaker heavily relies on DesktopJavaFramework for its basic structure. The MetroApp class inherits from AppTemplate and is composed of the workspace (MetroWorkspace), contains instances of MetroFiles, and MetroData which extend from AppFileComponent and AppDataComponent in DesktopJavaFramework. The workspace (MetroWorkspace) serves as the layout class of our application in which all static aspects are initialized, including the welcome dialog. It contains an instance of AppGUI, and AppTemplate from the DesktopJavaFramework class and extends from AppWorkspaceComponent (also from DesktopJavaFramework). Furthermore, the MetroWorkspace class is responsible for initializing the controller classes of the application of which it contains the instances of canvasController and mapEditorController, the controller for the canvas of the application and the controller for the map editing buttons. Both of these controllers contain instances of AppTemplate from djf.

The workspace is organized into a left pane where all of the toolbars with map editing controls are located and a canvas on the right which displays the map and includes the use of a gridpane for snapping functionality.

The purpose of the CanvasController class is to translate user inputs such as mouse clicks on our canvas to information that our program can use. Examples of such user inputs include clicking objects on the canvas, releasing the mouse in the canvas, and dragging objects on the canvas. The purpose of the MapEditorController class is to set up controller classes for all of the map editing buttons/controls as set up by the MetroWorkplace class. This is done by a method for each button/controller which creates transactions that implement Jtps_Transaction. Each transaction is represented by a nested inner class in the MapEditorController class. The Controller class includes a JTPS object that contains the transactions for undo/redo functionality. The MapEditorController class also contains numerous dialogs that are responsible for the addition of stations, editing of stations, and route finding pop-up windows. The route-finding functionality is implemented using the MapEditorController which contains a method called processFindRoute that gets from UI combo boxes the start and end stations and finds the optimal route between them and displays the resulting path with a dialog. Lastly, the MapEditorController contains a stack for copy/paste functionality (may or may not be implemented depending on user friendliness).

Most of the controller methods and transactions deal with the addition of map elements to the canvas such as stations, lines, and landmarks. To implement such functionality in this application, the MetroStation, MetroLine, MetroLandMarkImage, and MetroLandMarkText are used. The MetroStation class's purpose is to provide an object to represent stations, in which is contained a Circle object that represents the stop and a Text object for the name of the station. The MetroLine class is the object that represents the metro lines of the map, with and ArrayList of stations to represent all the stations included in the line and an ArrayList of Lines objects that connect each station of the line. Special Line objects are also members of each instance of MetroLine which indicate end lines that can be dragged. A method redrawLines is provided to help redraw lines between stations when the stations are dragged. Also included are the two mentioned LandMark objects which are intended to be objects that help create images and text on the canvas and make saving them in JSON easier.

The MetroData class in this application extends AppDataComponent and is responsible for holding a list of map elements that are to be added to the canvas and also other important aspects of the map. The MetroFile class is responsible for file I/O (saving and loading maps) and does this via JSON. The methods in MetroFile lets the user save maps into JSON files via the help of JsonObjects and loads data from JSON files into maps that the user can edit. The MetroState enum class contains the states that the program is in which is utilized by classes in the program to determine what is and is not possible to do at the time.

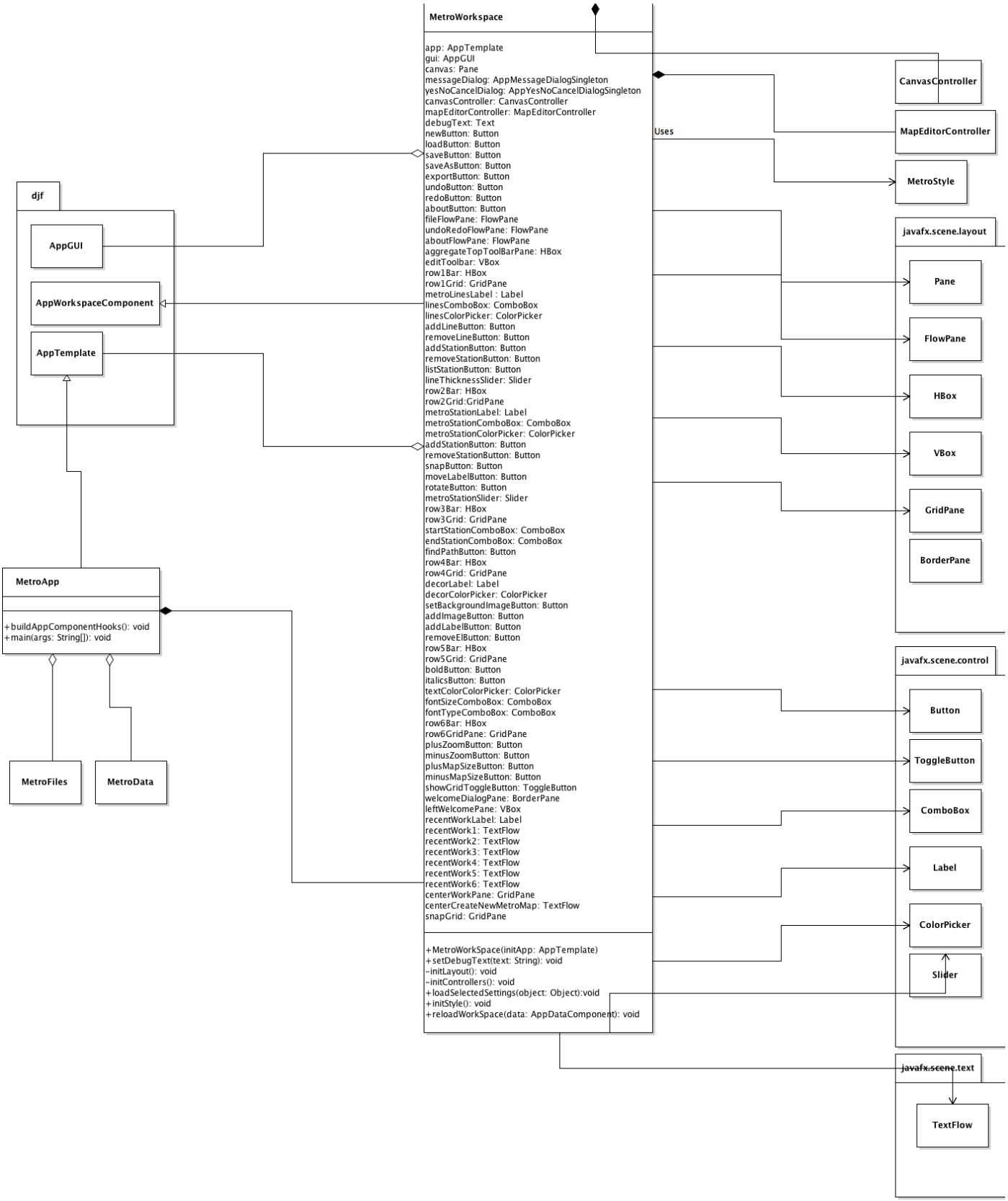


Figure 3.2 Detailed MetroApp and MetroWorkspace UML Class Diagrams

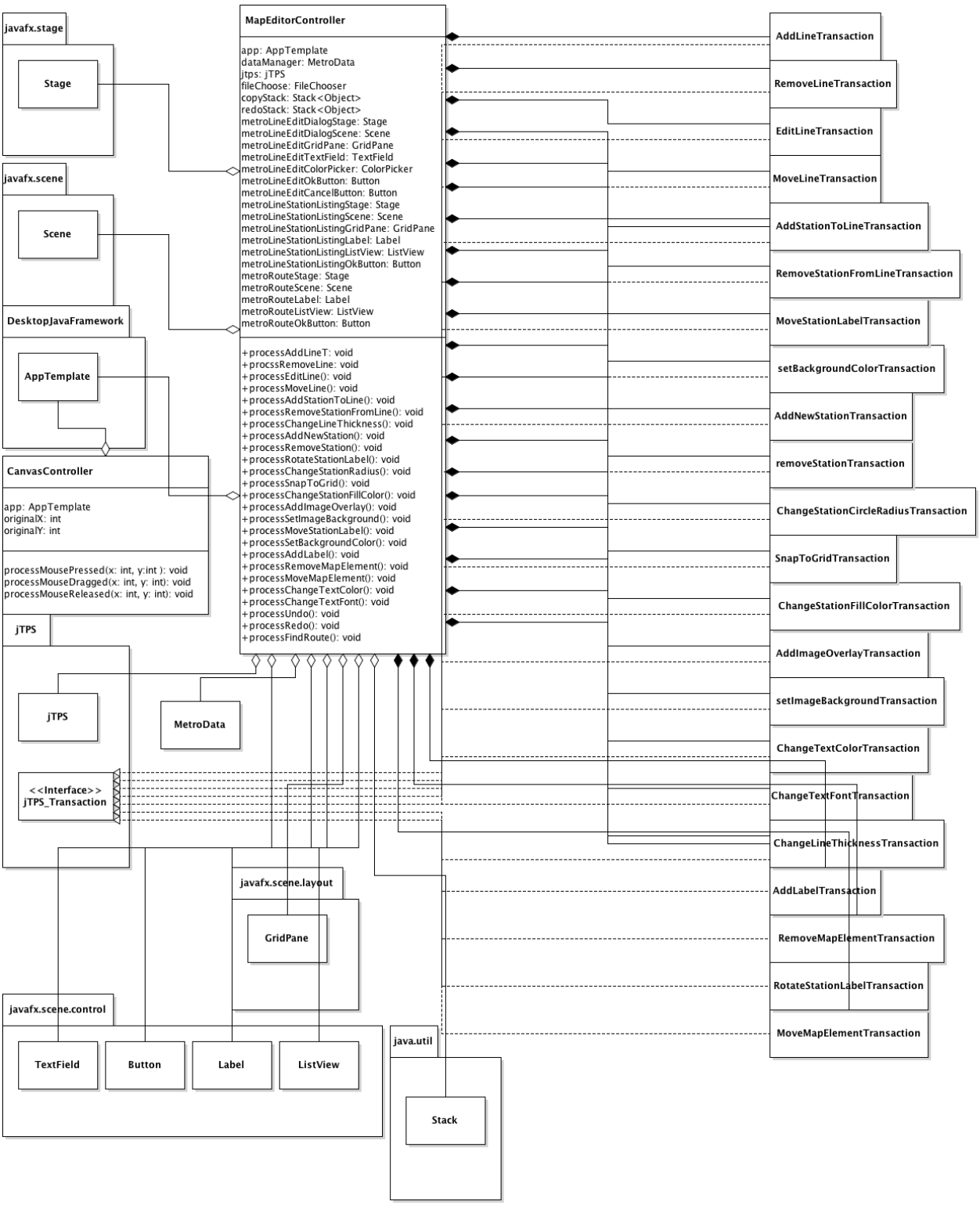


Figure 3.3 Detailed CanvasController and MapEditorController UML Class Diagrams. The nested inner clases (which end in transaction and implement Jtps_Transaction) are shown in this diagram. For abstraction reasons as well as cluttering issues, their specific implementations are not shown in this diagram.

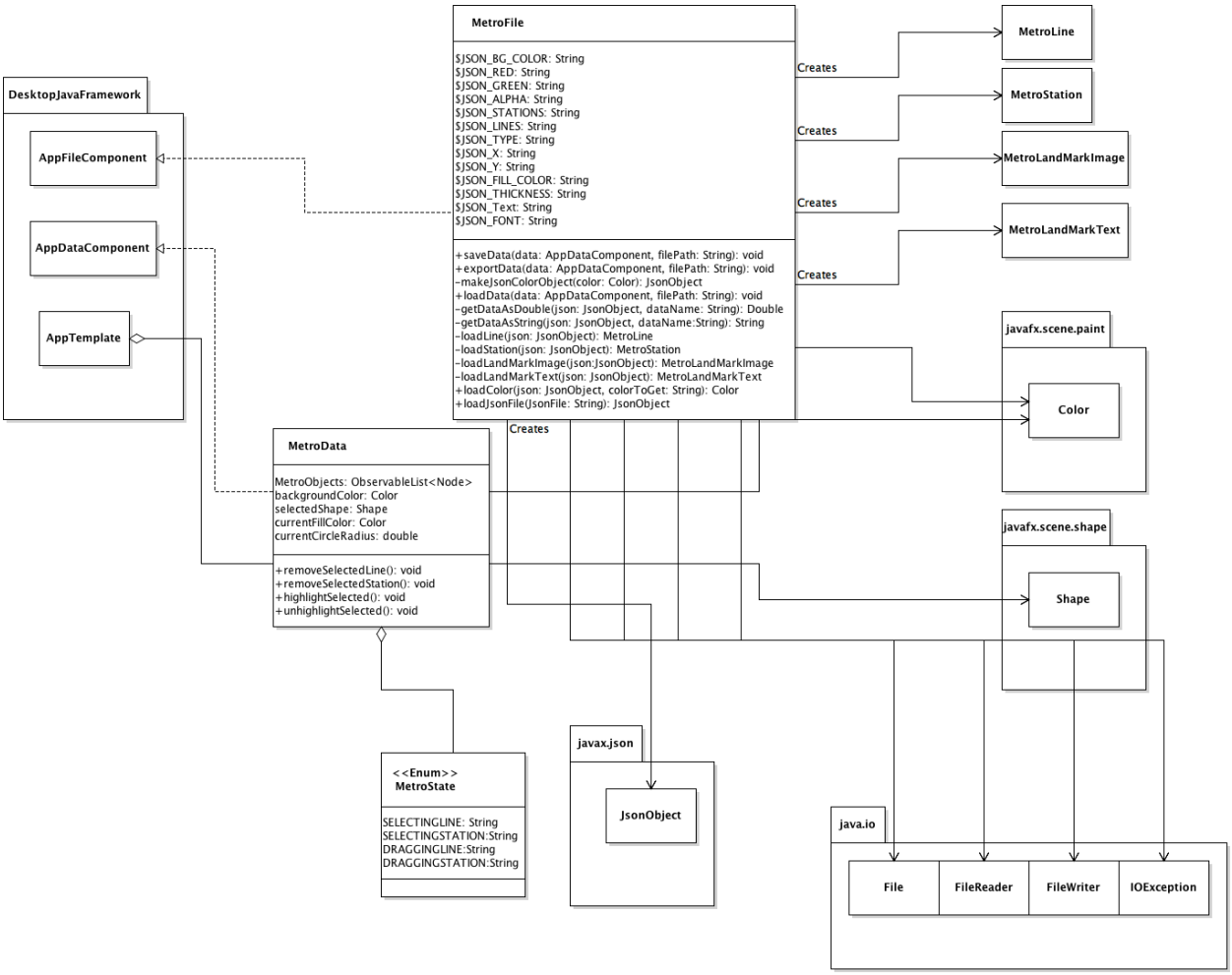


Figure 3.4 Detailed MetroFiles, MetroState and MetroData UML Class Diagram

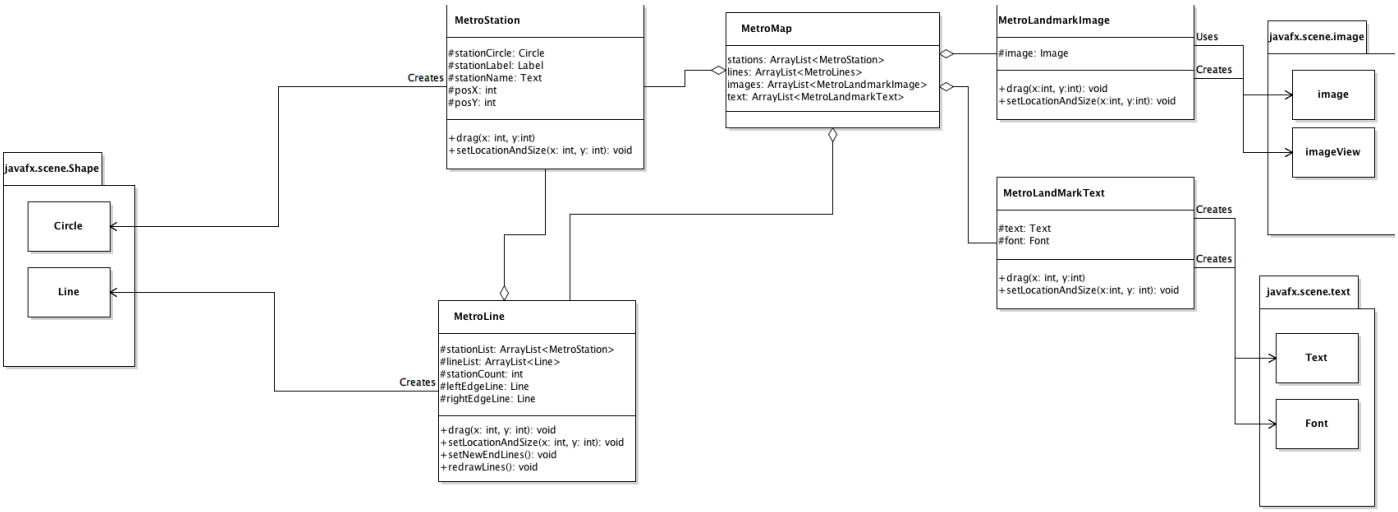


Figure 3.5 Detailed MetroLine , MetroStation, MetroLandMarkImage, MetroLandMarkText , Metro Map UML Class Diagram

4: Method Level Design Viewpoint

Now that the general architecture of the classes of this program have been determined, the following UML sequence diagrams describe the methods called within the code to be developed in order to provide the appropriate event responses.

15 Required UML sequence diagrams:

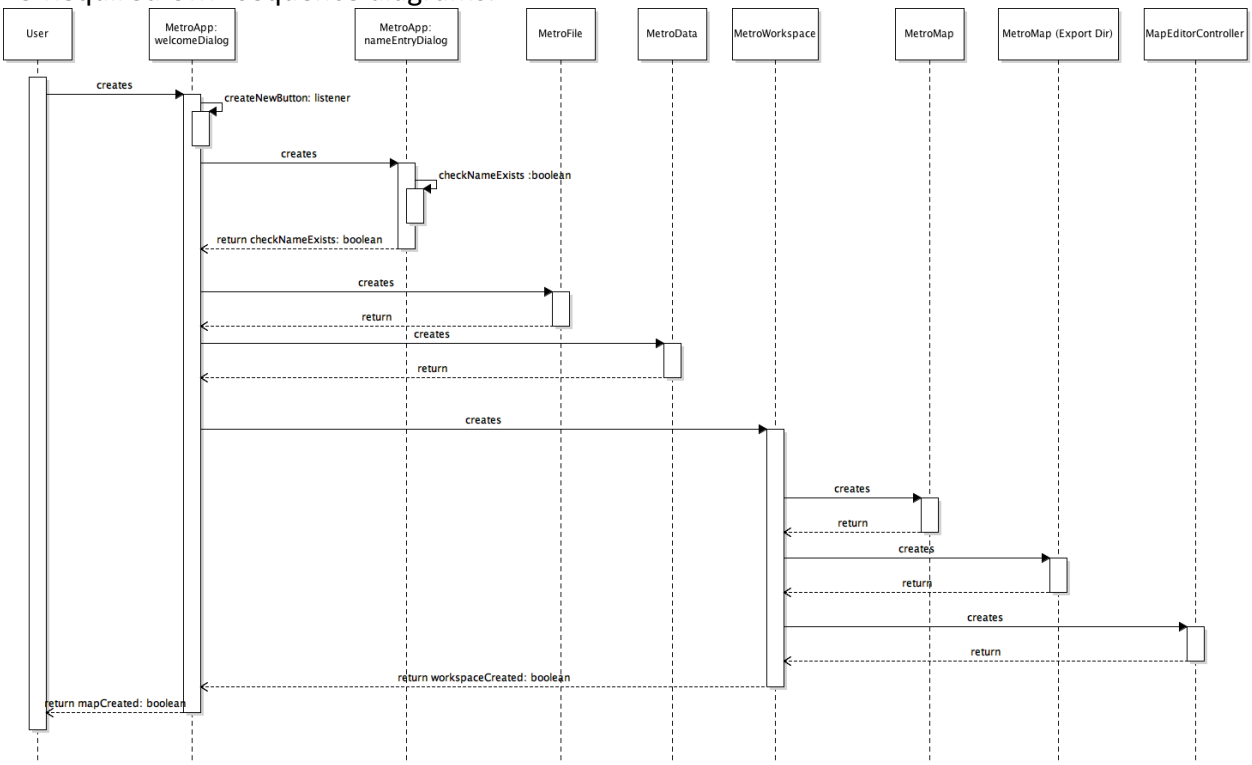


Figure 4.1 Create New Map UML sequence diagram

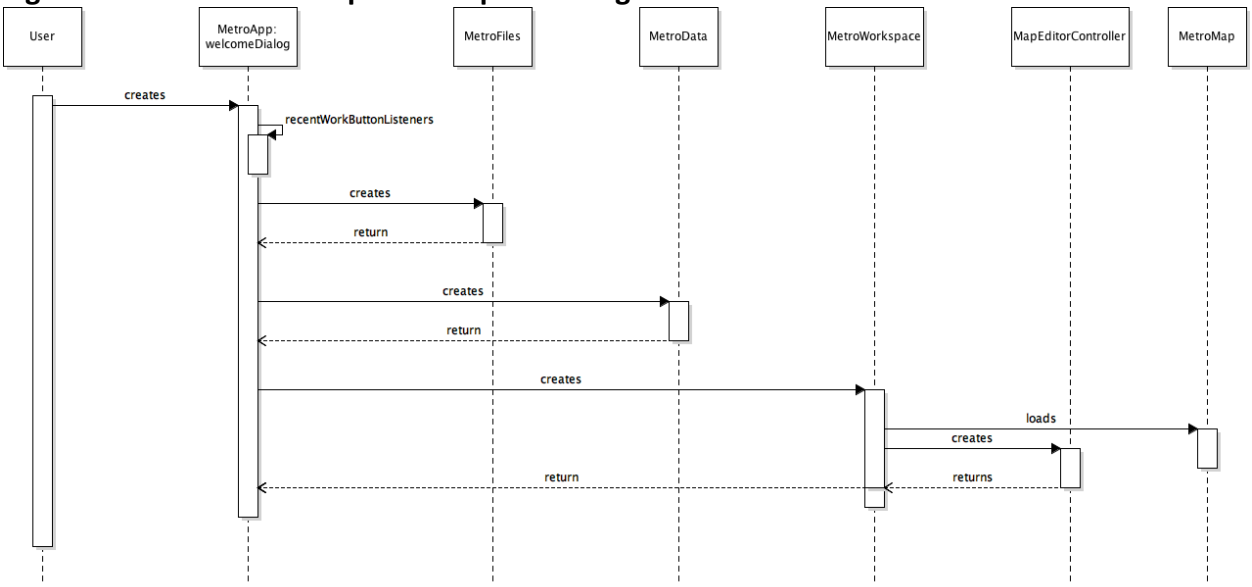


Figure 4.2 Select Recent Map to Load UML sequence diagram

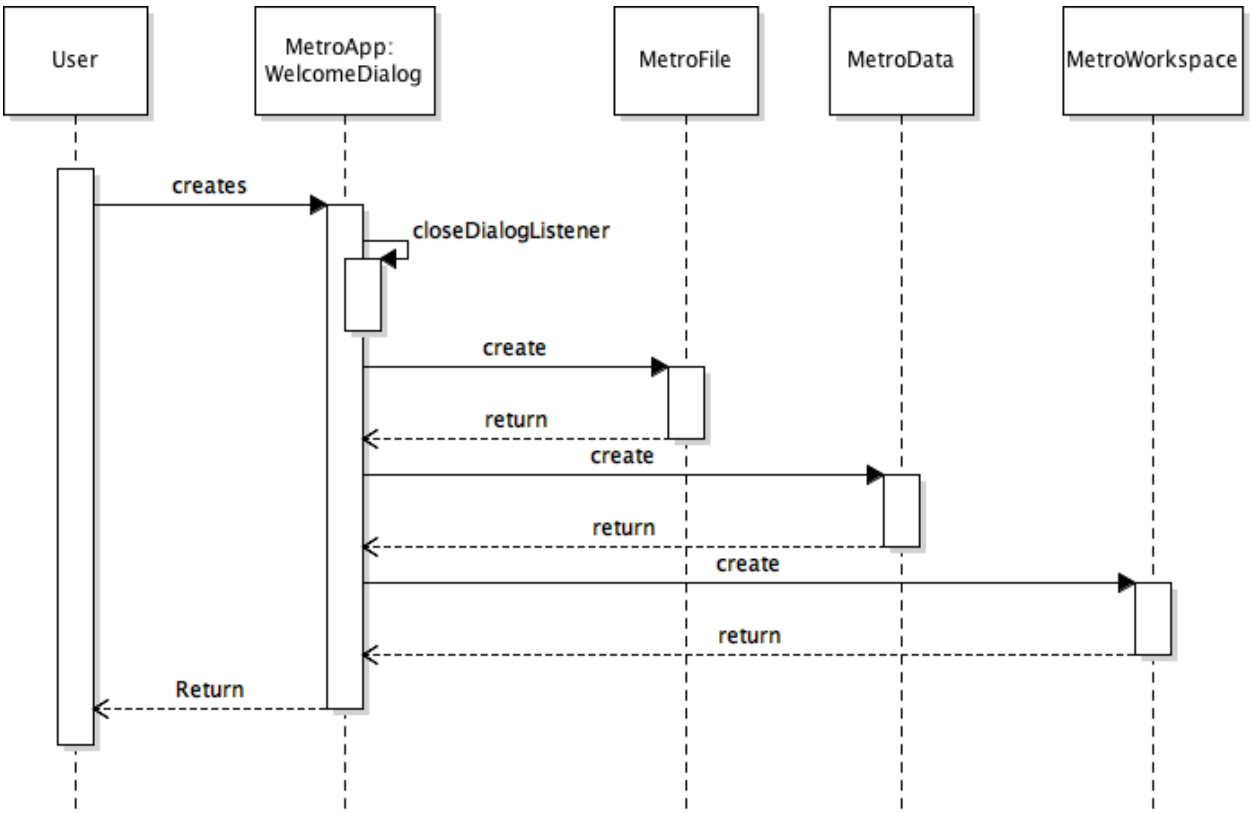


Figure 4.3 Close Welcome Dialog UML sequence diagram

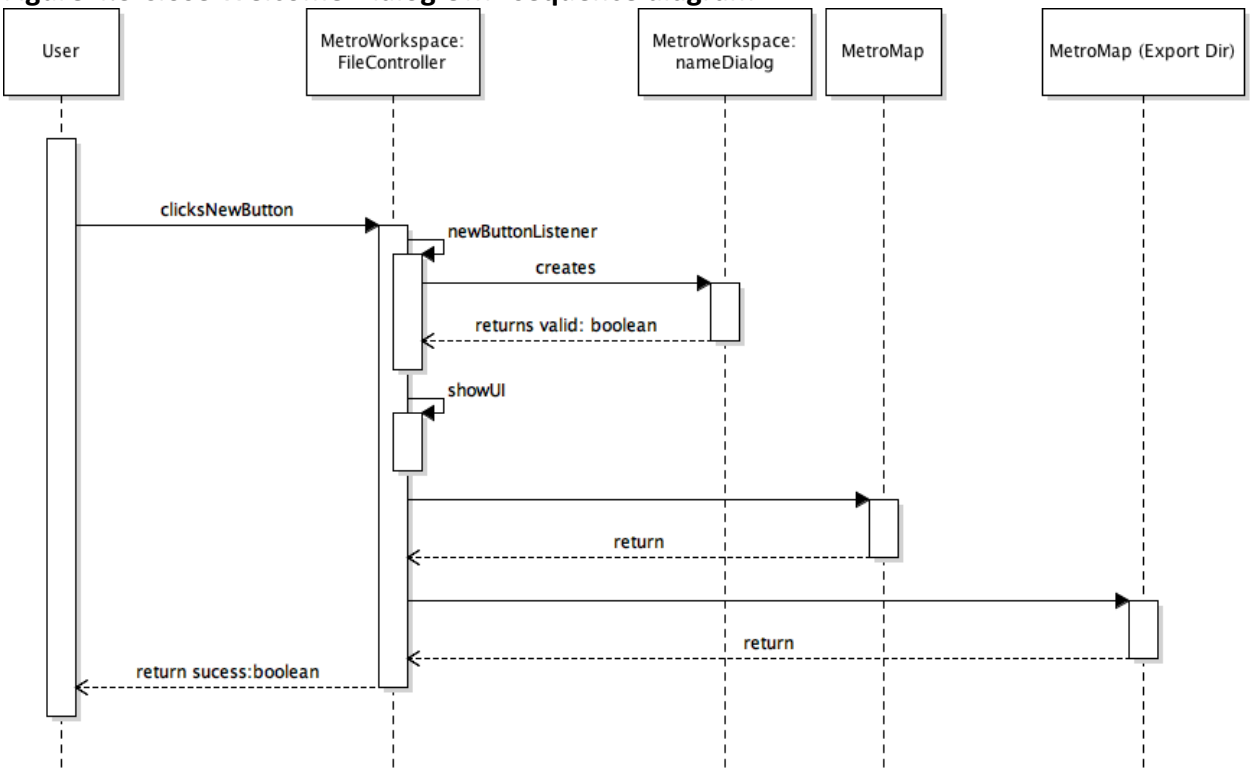


Figure 4.4 Create New Map UML sequence diagram

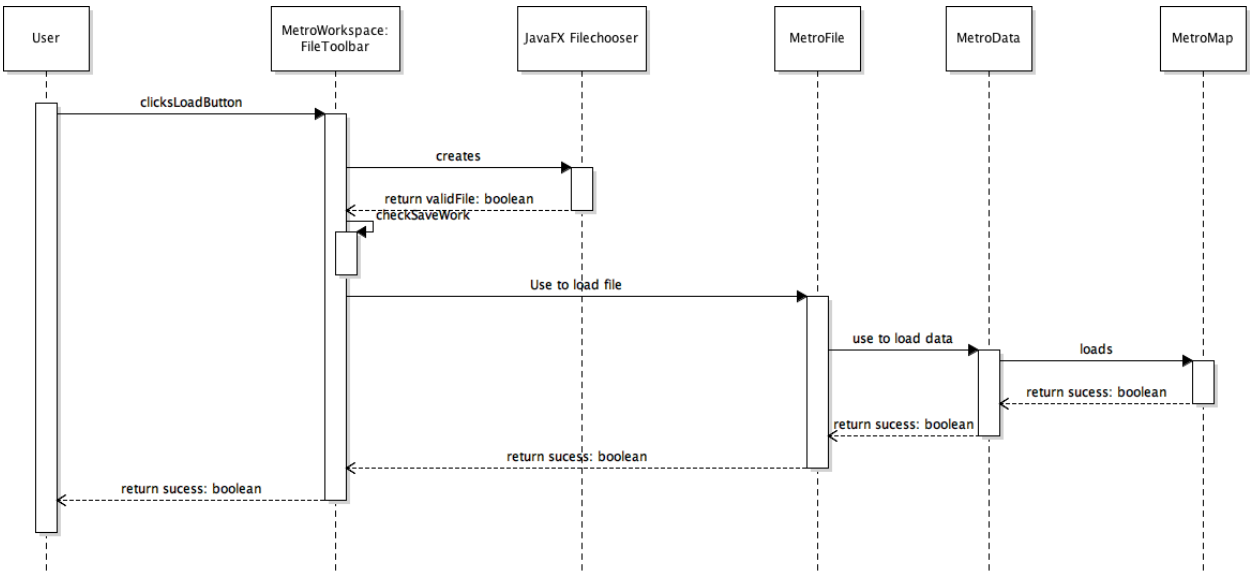


Figure 4.5 Load Map UML sequence diagram

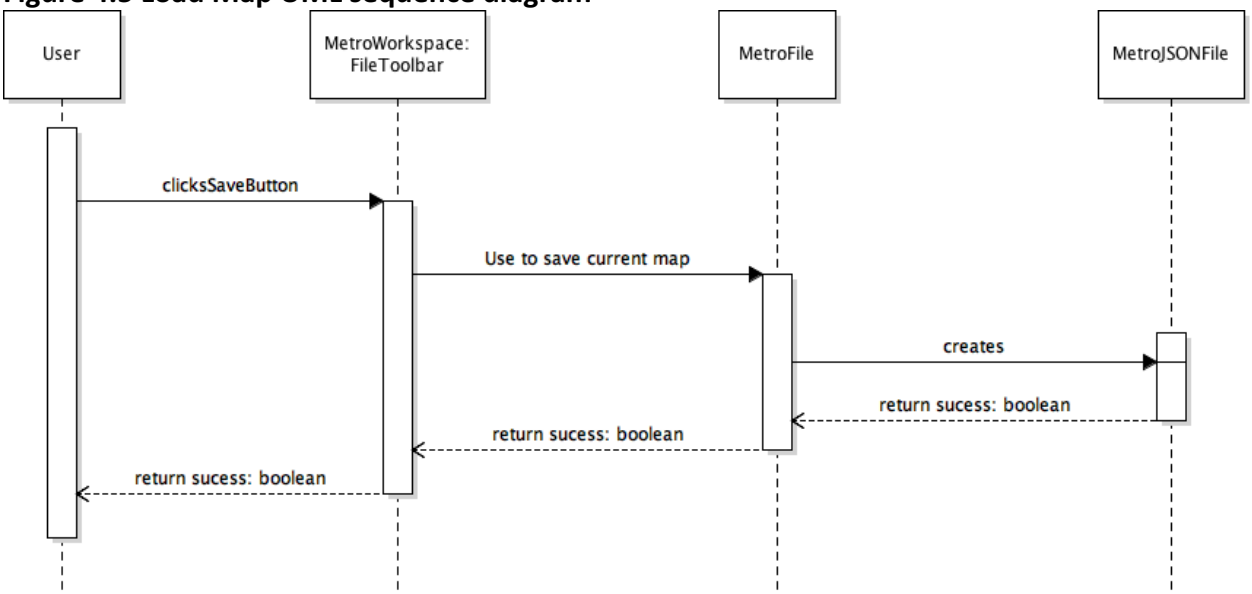


Figure 4.6 Save Map UML sequence diagram

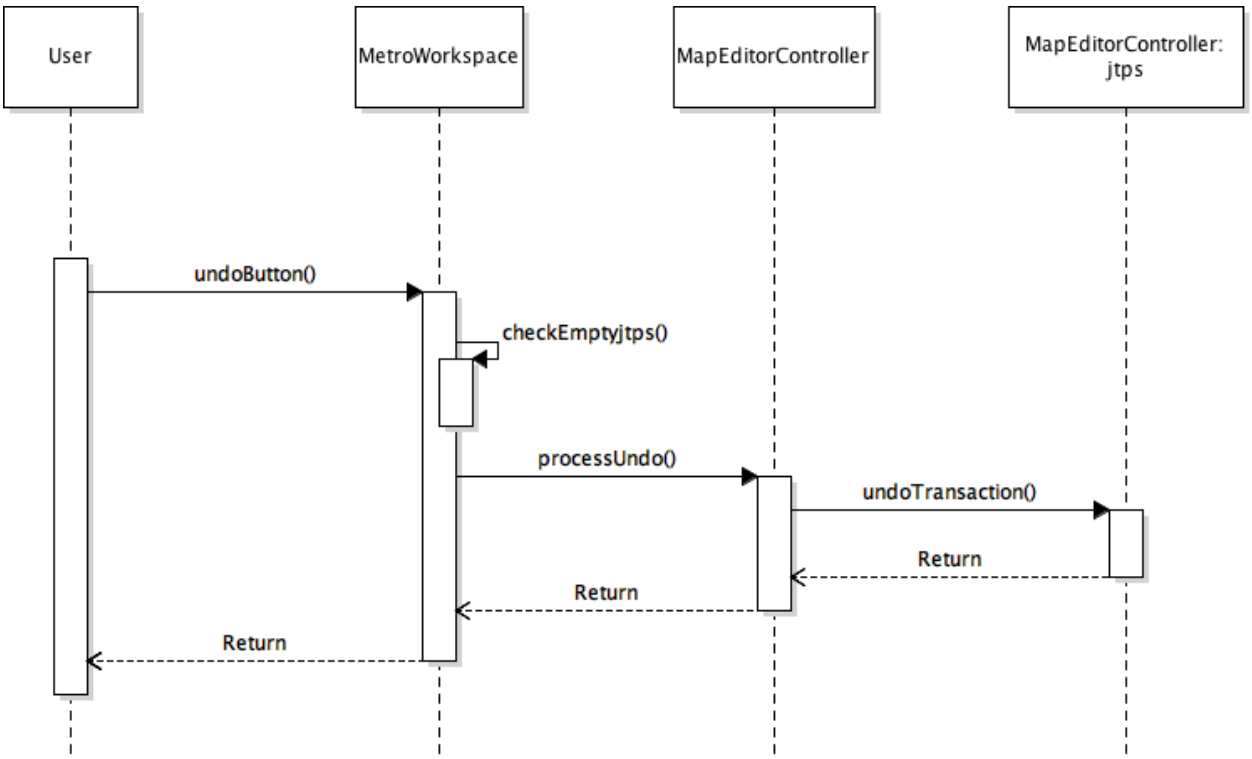


Figure 4.9 Undo Edit UML sequence diagram

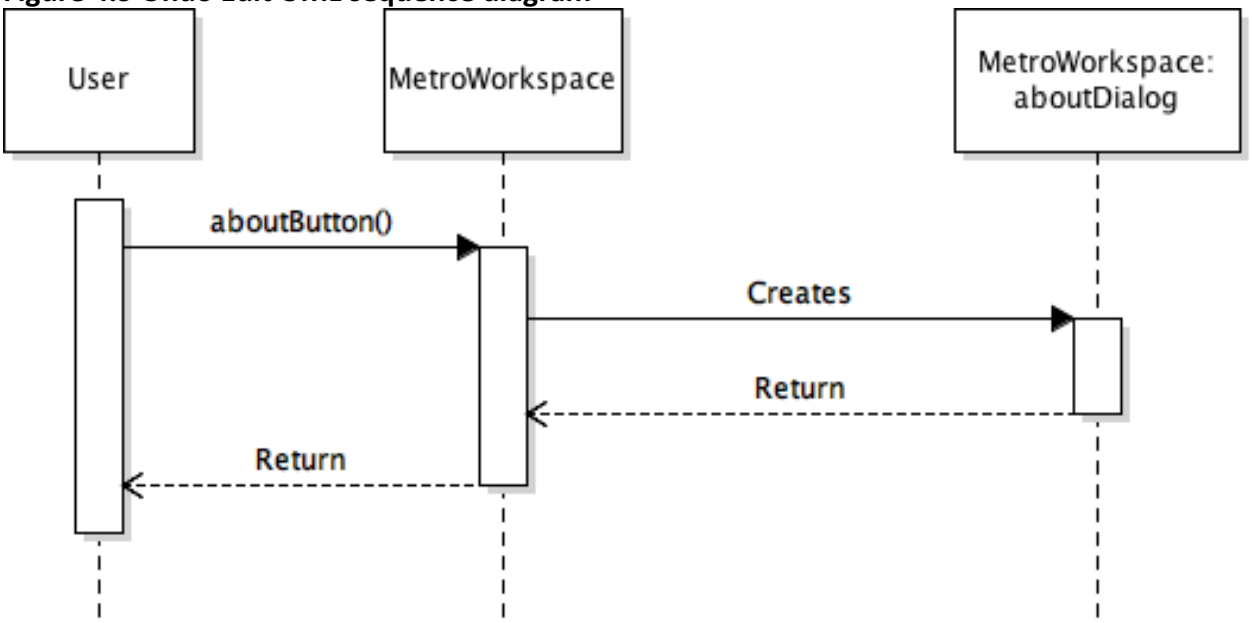


Figure 4.11 Learn About Application UML sequence diagram

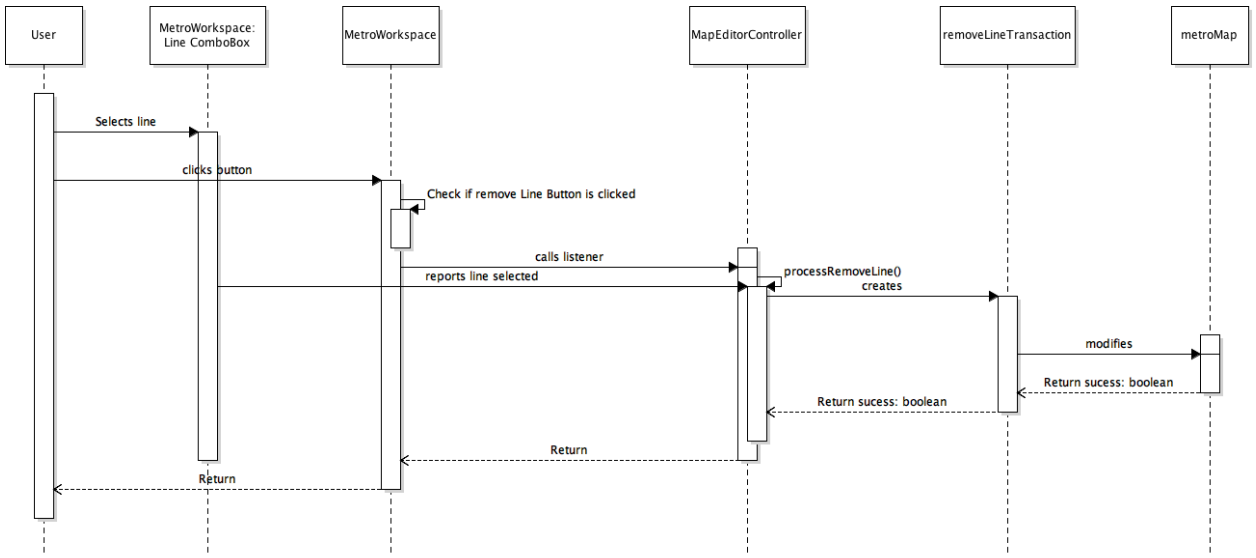
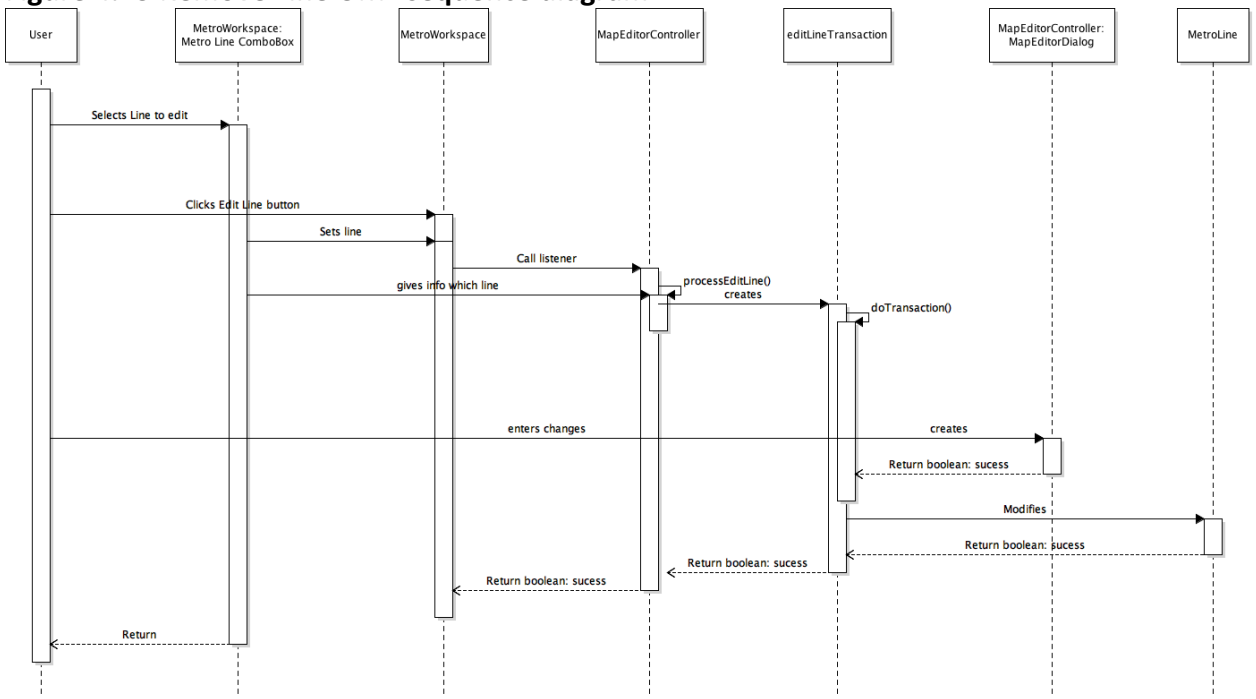
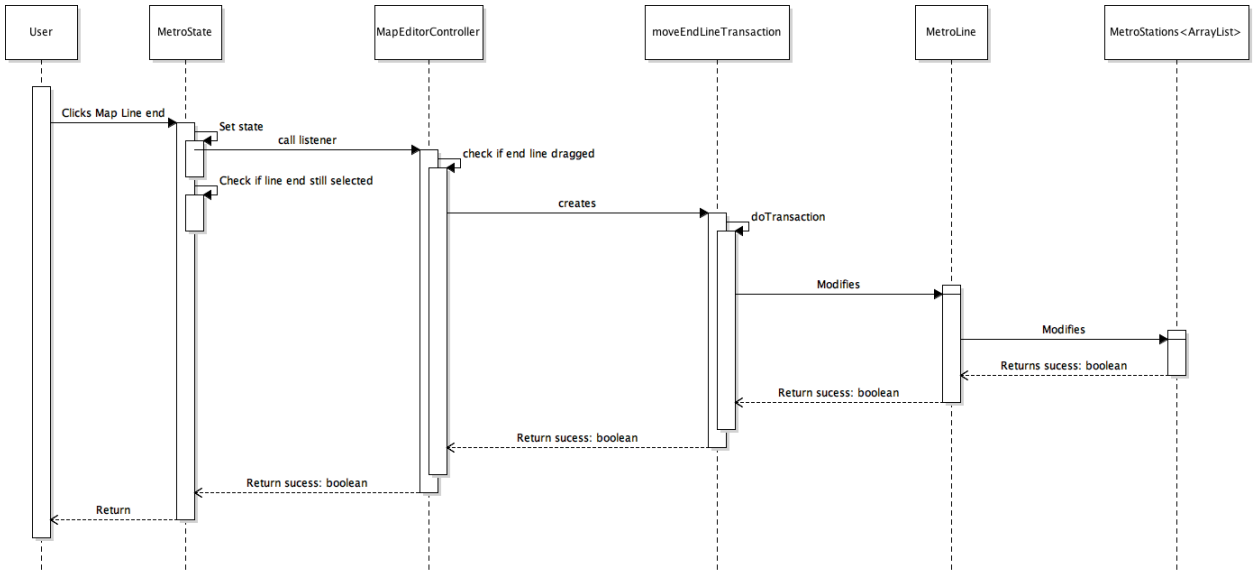


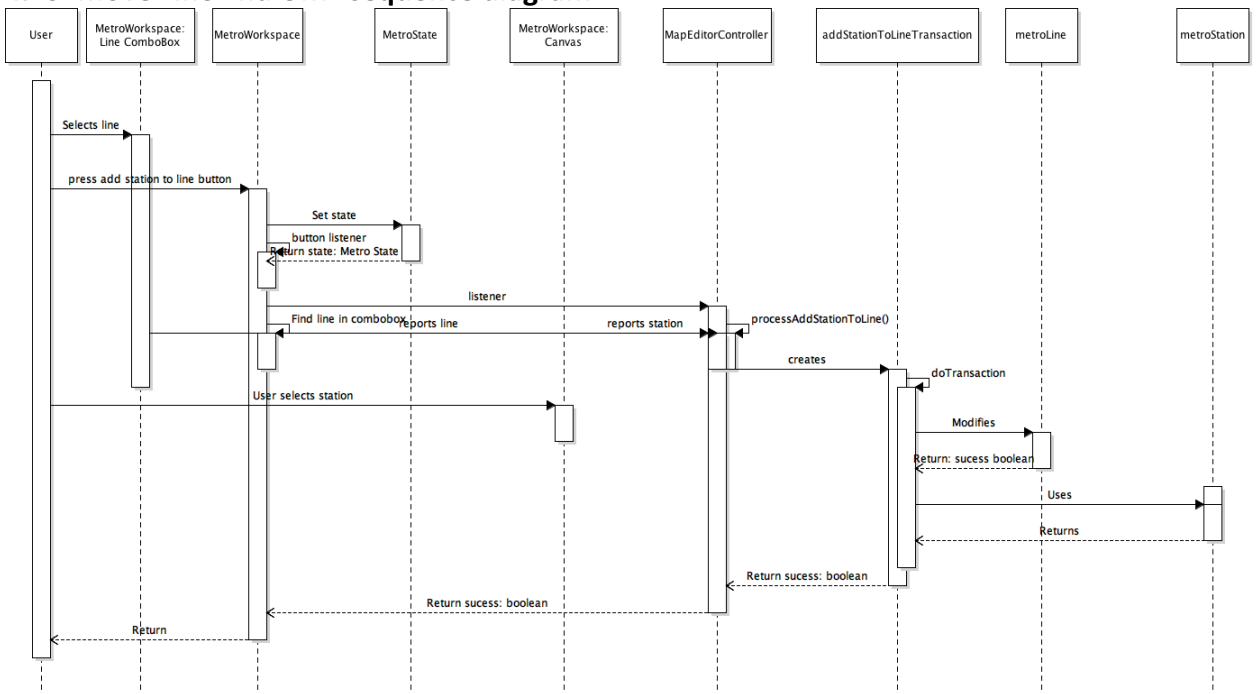
Figure 4.13 Remove Line UML sequence diagram



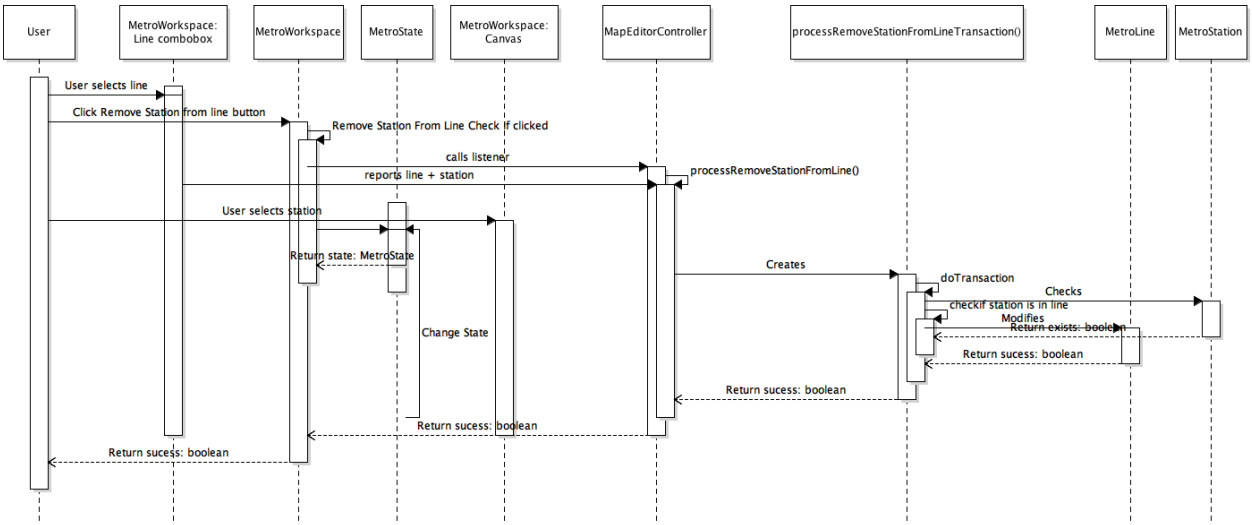
4.14 Edit Line UML sequence diagram



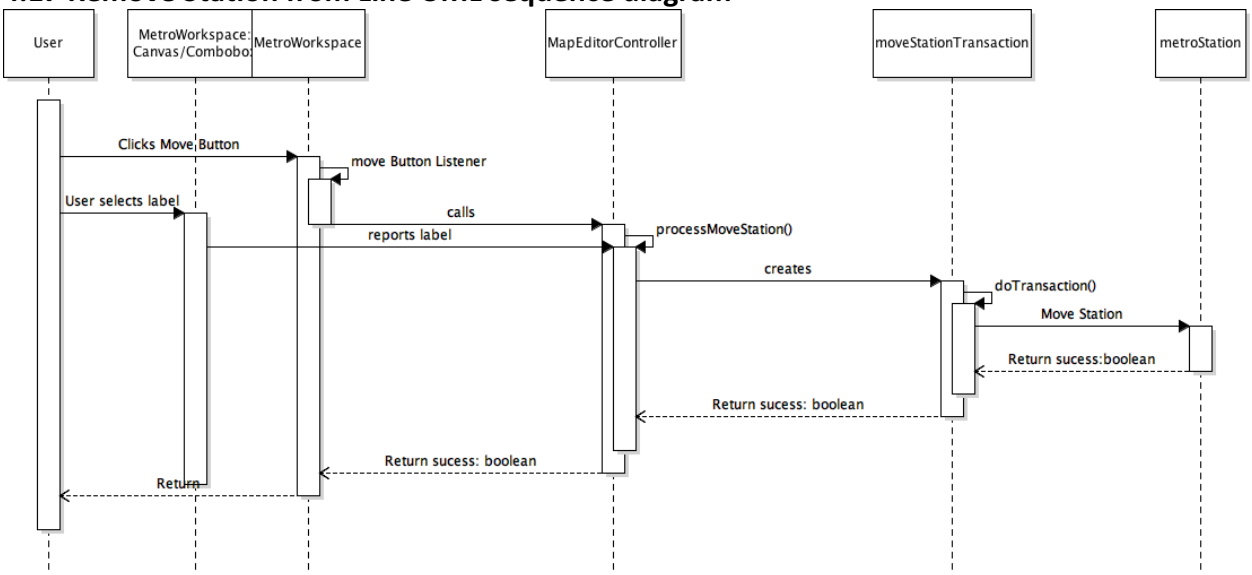
4.15 Move Line End UML sequence diagram



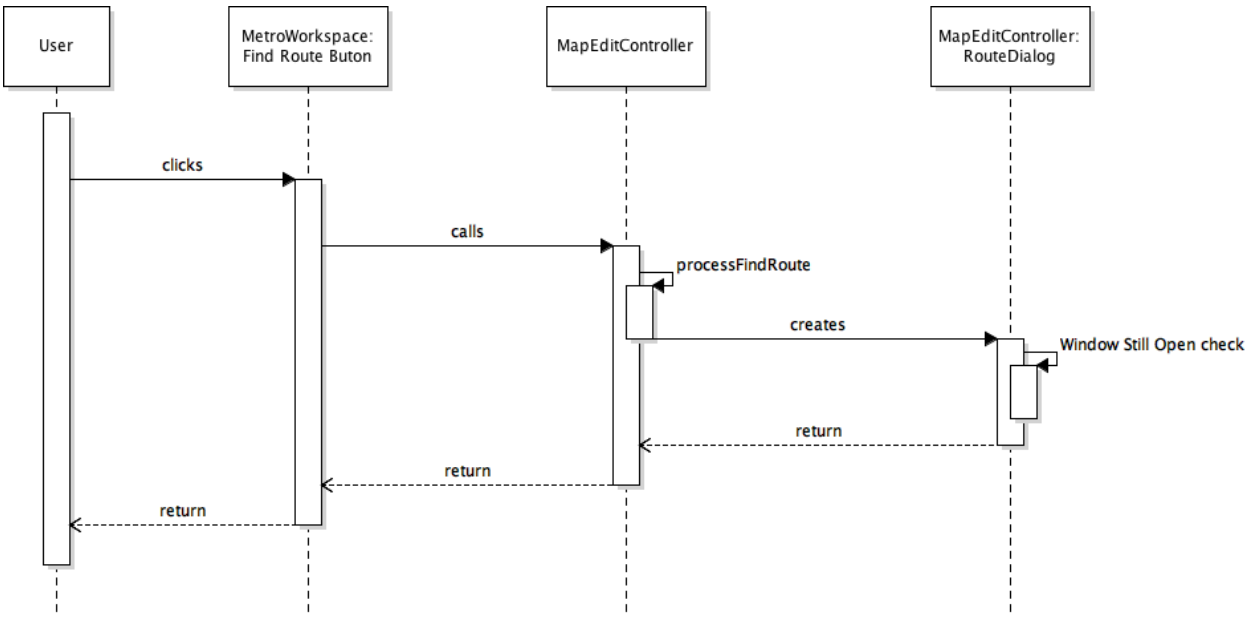
4.16 Add Stations to Line UML sequence diagram



4.17 Remove Station from Line UML sequence diagram



4.23 Move Station Label UML sequence diagram



4.27 Find Route UML sequence diagram

5: File Structure and Formats

The general file structure for this project is to be divided into the supporting frameworks and the main application. The supporting frameworks (DesktopJavaFramework, jTPS, PropertiesManager) will be their own separate packages. The main application will be located in the package MetroMapMaker and the inner file structure has already been provided. Necessary images will go into a folder named images. This will mainly be used for application logos and images of the sort. All work saved as JSON file are saved into a folder named work inside the package.

The following methods in MetroFiles (located inside src/file) are used to load and store JSON files for save load functionalities.

+saveData(data: AppComponent, filePath: String): void

: This method saves the data on the canvas with the help of a few helper methods into JSON files.

+loadData(data: AppComponent, filePath: String): void

: This method loads the data from a JSON file with a help of helper methods onto the canvas.

+exportData(data: AppComponent, filePath: String): void

: This method exports the data as an image.

Figure 5.1 : Example JSON File showing general format of JSON file which is used for saving and loading. Note it is a general layout.