**Software Design Description**

**Author:** Augusto Celis

Debugging

**Abstract:** This document describes the software design for Metro Map Maker, an application which will be used to help users make their own custom metro maps, and save for later reference.

**Based on IEEE Std 1016TM-2009 document format**

Copyright © 2018 Debugging Enterprises

*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**

To make your way around a new city, you may have some trouble at first so one may go to the internet for solutions. Cities usually provide a graphical representation of their subway, showing stations, lines, transfers, etc. These maps let people map out their trip, what line they have to take, and how many stops it will be before they arrive at their location. This graphical representation also usually lets a user figure out what route is optimal for arriving,

The ***Metro Map Maker (i.e. )*** application will let a user build their very own customized graphical representation of city subway systems. This will include naming the lines, naming the stations, intersecting lines/stations, and landmarks. It will also provide a tool for calculating the optimal route to make it from one stop to another. It will also provide an export feature that will export the map information to a format that can be understood by a corresponding Web application that will make use of it.

* 1. **Purpose**

The purpose of this document is to specify how the Metro Map Maker program will work, and a general blueprint for how it is going to come out. The design will use UML class diagrams to provide complete details regarding all packages, classes, instance variables, class variables, and method signatures to build the application. UML sequence diagrams will also be provided to specify object interactions after the initialization of the application, in response to a variation of user input and events.

* 1. **Scope**

The whole purpose behind this project will be to help a user create a map and edit their maps. There will be an emphasis on ease of use, and user friendliness. There will be an export format for all maps, so they may be used by a uniform application.

* 1. **Definitions, acronyms, and abbreviations**

**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.

**GUI** – Graphical User Interface, visual controls like buttons inside a window in a software application that collectively allow the user to operate the program.

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

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

**JavaScript** – the default scripting language of the Web, JavaScript is provided to pages in the form of text files with code that can be loaded and executed when a page loads so as to dynamically generate page content in the DOM.

**XML** – This is the language used to have our own customized language and pictures on the canvas. This is used in tandem with all the buttons displaying images, tooltips, and error messages.

**Json** – This is the language used to store data and save it to a document. This is used in tandem with the saving, loading, and exporting information prompts in the program.

**CSS** – This helps the design of the elements on the screen, by giving each box their appropriate widths, colors, and spacing upon startup of the program.

**Stylesheet** – a static text file employed by HTML pages that can control the colors, fonts, layout and other style components in a Web page.

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

graphically.

**M3** – This is the standalone application itself, also short for Metro Map Maker. This is the title that will appear once one is working with the application in real time.

**Djf** – This is the software framework that is developed to work alongside the metro map maker application. This will allow for ease of use when creating the application.

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

**Transaction** – When any of the UML diagrams have transaction appended to the end of the word, that means that this method is to edit the map, and will be undoable.

**Canvas Controller** – This is the name given for the class that will primarily take care of whenever someone clicks, drags, or releases the mouse

**Map Controller** – This is the controller that will take care of anything that happens when one wants to edit anything such as adding stations, removing lines, adding labels, rotating labels, etc.

**Properties Manager** – This is a package that is given by professor McKenna which loads up all the properties from any extra XML files that we may use.

* 1. **References**

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

**Dr. Richard McKenna –** Helped develop the djf framework within this project

* 1. **Overview**

The SDD will define how the Metro Map Maker application should look an operate upon running of the application. Section 2 of this document will provide the Package-Level Viewpoint, specifying the packages and frameworks to be defined. 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.

1. **Package-level Design ViewPoint**

This design will encompass both the djf framework and the m3 application that will be used in the construction of the GUI. This will also consist of the Properties Manager class which will help load the properties from the class for the GUI. In building both of these we will rely on Java API heavily to provide multiple packages and services four our code. Following this are descriptions of how the Java API are used.

* 1. **M3 and djf Framework**

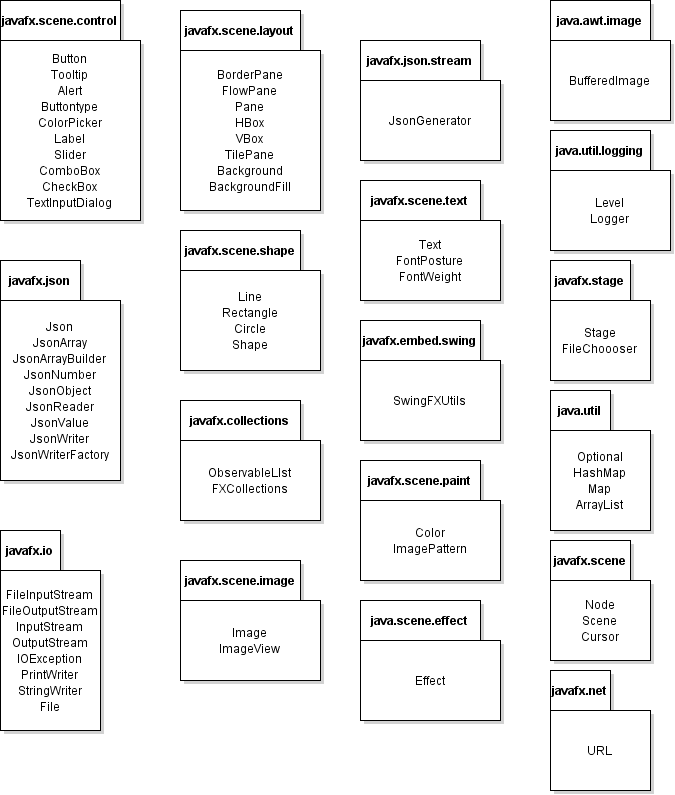
The djf Framework and M3 framework will be designed and developed in tandem. Figure 2.1 specifies all the components that will have to be developed and places all of them into their appropriate packages.

|  |  |
| --- | --- |
| Djf Framework | M3 Framework |

**Figure 2.1: Design Overview Packages**

**2.2 Java API Usage**

Both the framework and the m3 classes will use java API. Due to that we will use each of the classes specified in figure 2.2



**Figure 2.2: Java API and Packages to be used**

**2.3 Java API Usage Descriptions**

Tables 2.1-2.18 below summarize how each of these classes will be used.

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Button** | The various different GUI buttons that one can press |
| **Tooltip** | When one hovers over a button, a tooltip explaining the button will appear |
| **Alert** | To set off various different pop up windows, such as the about button box |
| **Buttontype** | This helps set up the Buttons for alert and TextInputDialog boxes |
| **ColorPicker** | The rendering and color choice for stations, background, and metro lines |
| **Label** | Text name for certain buttons and boxes |
| **Slider** | For scaling station and line thickness |
| **ComboBox** | For storing names of stations, metro lines, and route origin/destination |
| **CheckBox** | For activating the visual grid for a user |
| **TextInputDialog** | To set up other pop up windows, such as text modifications to draggable text instances. |

**Table 2.1: Uses for classes in the Java API’s javafx.scene.control package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **JSon** | Helps to create Json objects for storing data about the map |
| **JSonArray** | An array of Json objects. Could be an array of any Json object type |
| **JSonArrayBuilder** | Helps to build the Json array itself |
| **JSonNumber** | A number value for Json objects |
| **JsonObject** | A Json object value which represents any int or double. Usually used for width, height, coordinates. |
| **JSonReader** | Helps to read from Json files such as saved maps |
| **JsonValue** | A Json value which is abstractly defined for our use |
| **JsonWriter** | Write a Json object to an output source, helps with saving to files |
| **JsonWriterFactory** | Factory which help creates the JsonWriter instances |

**Table 2.2: Uses for classes in Java API’s javafx.json package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **FileInputStream** | Helps to obtain bytes from the file system to load files |
| **FileOutputStream** | Helps to write data from app to file, such as writing to a directory |
| **InputStream** | This is the abstract class for all input streams which helps to read files |
| **OutputStream** | This is the abstract class for all output streams which helps to write out to a file |
| **IOException** | Usually used when trying to read images from a file |
| **PrintWriter** | Helps to format objects into a textually pleasing document |
| **StringWriter** | Helps to collect its output using string buffer, which well then construct the string itself |
| **File** | An abstract class for all file and directory names, and to use file writers/readers |

**Table 2.3: Uses for classes in Java API’s javafx.io package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **BorderPane** | Helps to layout nodes on top, left, bottom, and right which is used for the workspace itself |
| **FlowPane** | Helps to set up children that will be wrapped, such as the toolbars |
| **Pane** | Base class for panes, which helps to set up canvas, toolbars, and file toolbar |
| **HBox** | Helps to set up children in a horizontal row, such as the bold/italic buttons side by side |
| **VBox** | Helps to set up children in a vertical column, such as the 2 route station choices |
| **TilePane** | Helps to set up a grid of uniform “tiles”, which will help show the grid |
| **Background** | The background of the region, which will help set up the color for the background, or add an image |
| **BackgroundFill** | Helps set up the fill for the background itself |

**Table 2.4: Uses for classes in Java API’s javafx.scene.layout package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Line** | Creates a line at specified start and end coordinates. Used for making the metro lines |
| **Rectangle** | Creates a rectangle of specified shape and size. Will be used to help make draggable images, which will be put as a fill for the rectangle |
| **Circle** | Creates a circle with specified radius, which will help to keep consistent metro stations |
| **Shape** | The overall class, which will help to access all the methods of each shape in the java API |

**Table 2.5: Uses for classes in Java API’s javafx.scene.shape package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **ObservableList** | Allows listeners to track what happens, such as when one clicks a button |
| **FXCollections** | Helps to get all of the methods from the java.util.collections methods. |

**Table 2.6: Uses for classes in Java API’s javafx.collections package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Image** | Allows listeners to track what happens, such as when one clicks a button |
| **ImageView** | Allows nodes with custom images to be used, which will help with Alert dialog boxes |

**Table 2.7: Uses for classes in Java API’s javafx.scene.image package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **JsonGenerator** | Helps to write Json values one at a time, transferring over all the objects to a Json format |

**Table 2.8: Uses for classes in Java API’s javafx.json.stream package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Text** | Displays text which will be used as a node, which will help with creating Labels around the map. |
| **FontPosture** | Helps to create the posture of the text, such as italicizing the word itself |
| **FontWeight** | Helps to create the weight of the text, which will help with bolding the word |

**Table 2.9: Uses for classes in Java API’s javafx.scene.text package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **SwingFXUtils** | This will help convert between Swing AWT and JavaFX format which will help activate all the different Java classes that are being used. |

**Table 2.10: Uses for classes in Java API’s javafx.embed.swing package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Color** | This class will help create all the RGB colors, which will either be set manually or be picked from a color picker wheel. |
| **ImagePattern** | This will end up accounting for the draggable image. It will fill a shape with an image pattern and will be used just like a shape. |

**Table 2.11: Uses for classes in Java API’s javafx.scene.paint package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Effect** | This class is the abstract for all effect implementations, helping to create an effect to our shapes. |

**Table 2.12: Uses for classes in Java API’s javafx.scene.effect package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **BufferedImage** | This will help to buffer the image out, and grab the data from the image file |

**Table 2.13: Uses for classes in Java API’s javafx.scene.image package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Level** | This will help set the level of severity for our exception methods |
| **Logger** | This will help set up the logging event for the exception dialog. |

**Table 2.14: Uses for classes in Java API’s java.util.logging package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Stage** | This is the top level container, which will help to contain all the GUI space. |
| **FileChooser** | This will help with selecting files using a file reader. This will be used when choosing an image. |

**Table 2.15: Uses for classes in Java API’s javafx.stage package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Optional** | Container that will have either null or non null objects |
| **HashMap** | This will help to has value to specific keys, in our case being to help map out certain Json files and arrays |
| **Map** | The general interface for API that works similar to HashMap |
| **ArrayList** | A dynamically resizable array that will hold certain values, such as our jTPS transaction lists for undo redo. |

**Table 2.16: Uses for classes in Java API’s java.util package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **Node** | This class will help create all the functional shapes within this program, such as our DraggableImages and our DraggableTexts |
| **Scene** | This is the scene container for everything inside of the scene graph. This is placed into the stage. |
| **Cursor** | This class helps to detect the cursor, which will help to detect the different events that are happening at each moment such as clicking, dragging, and releasing the mouse. |

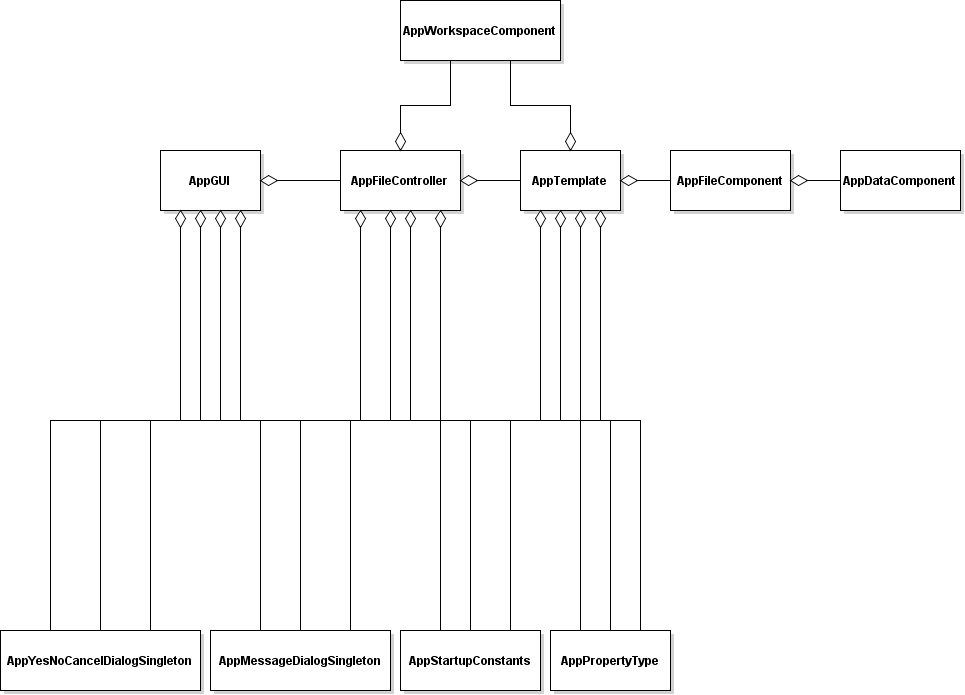
**Table 2.17: Uses for classes in Java API’s javafx.scene package**

|  |  |
| --- | --- |
| **Class/Interface** | **Use** |
| **URL** | This will help to find the info from our photos and navigate the file chooser paths. |

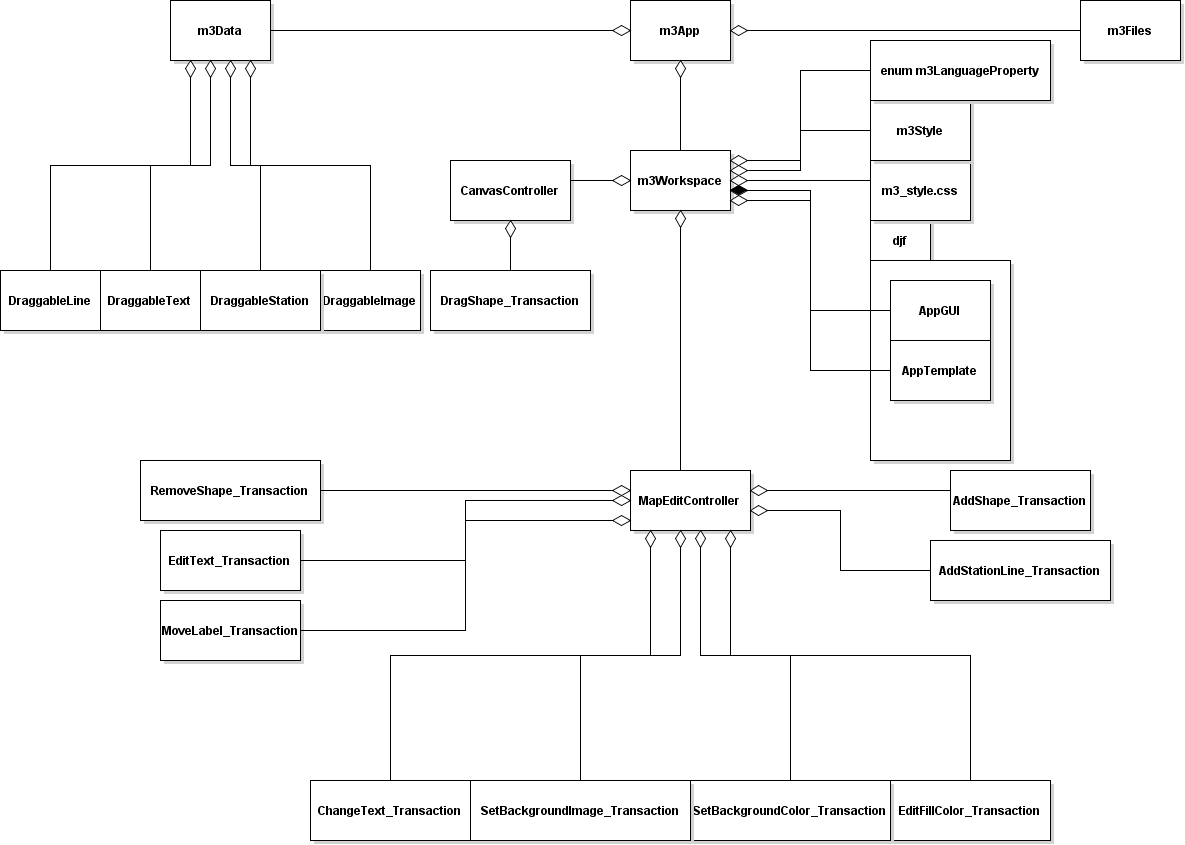
**Table 2.18: Uses for classes in Java API’s javafx.net package**

1. **Class-Level Design Viewpoint**

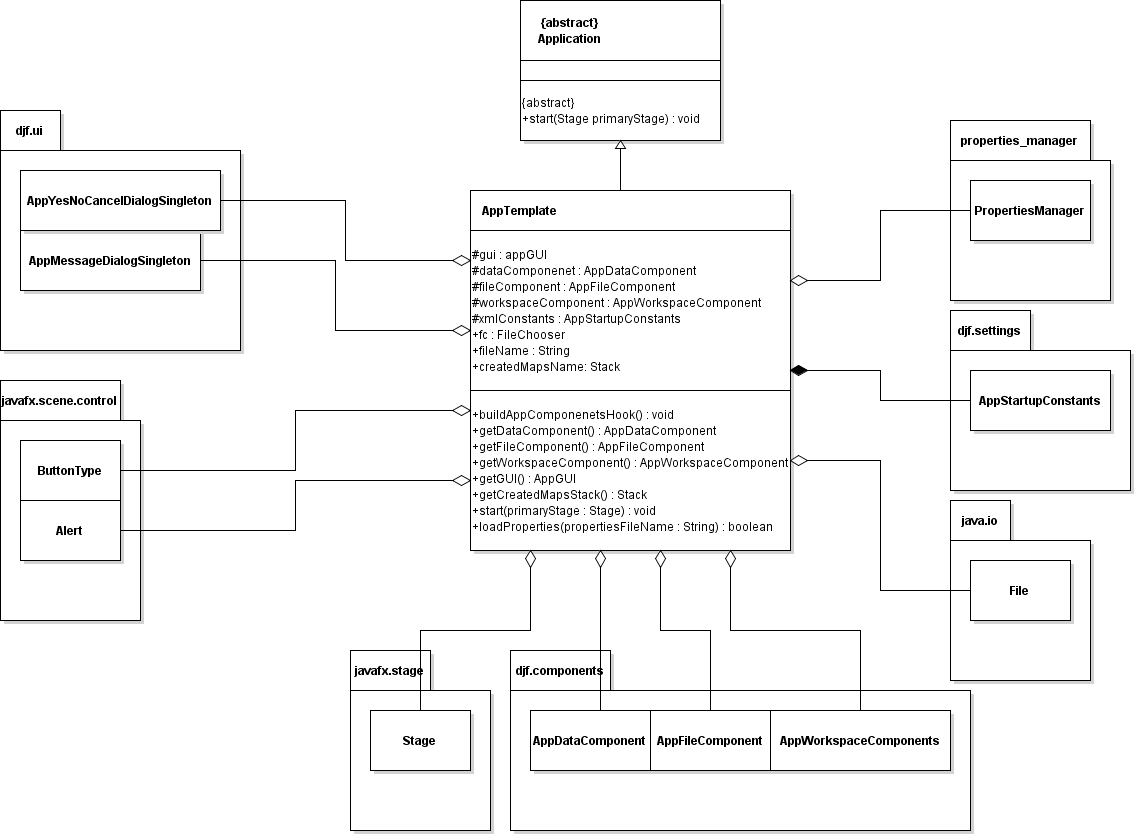
As mentioned before, the design will encompass the djf and the m3 frameworks which will help build the application itself. The following UML Class Diagrams reflect this. Due to the complexity of this project, we present it using diagrams ranging from an overview to very detailed diagrams. *Note:* Some of the packages are generalized to save space on the UML diagrams.



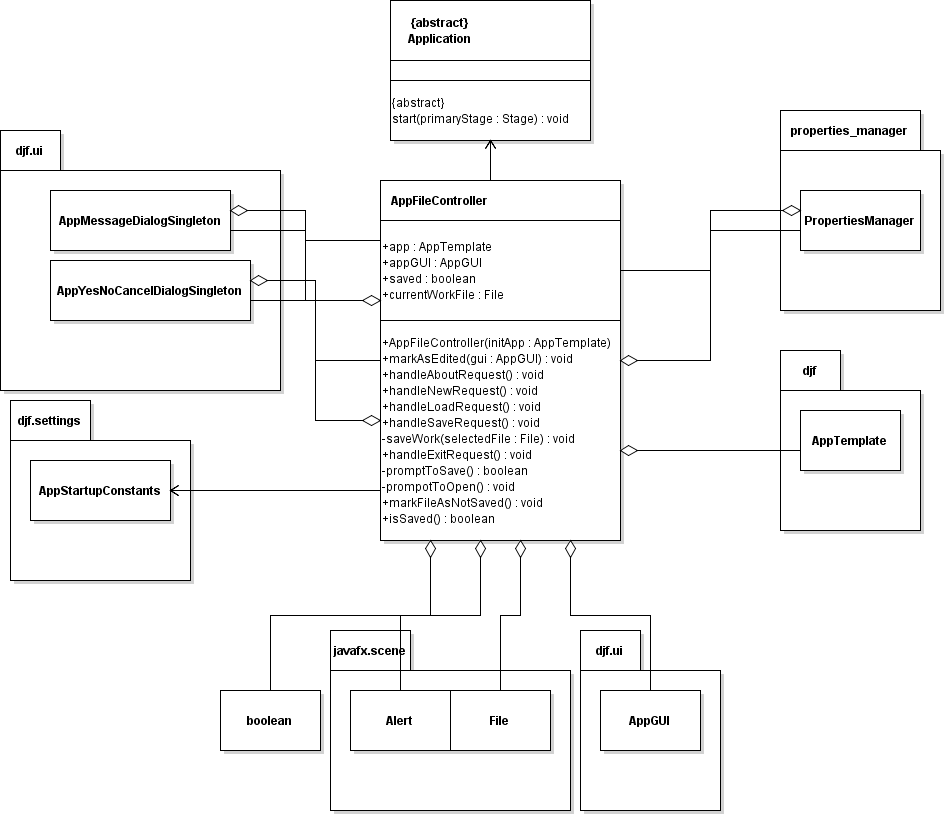
**Figure 3.1: Djf Framework Overview UML Class Diagram**

****

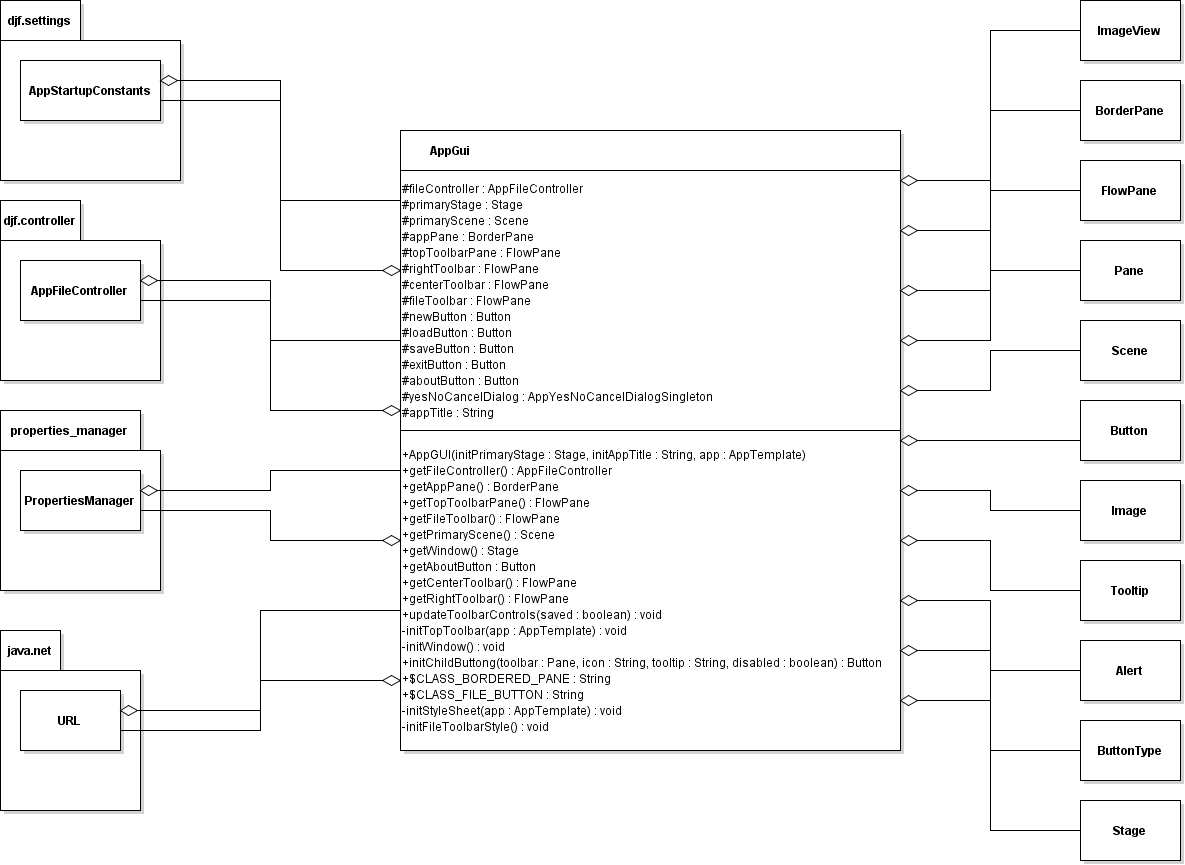
**Figure 3.2: M3 Overview UML Class Diagram**

****

**Figure 3.3: Detailed AppTemplate UML Class Diagram**

****

**Figure 3.4 Detail AppFileController UML Class Diagram**

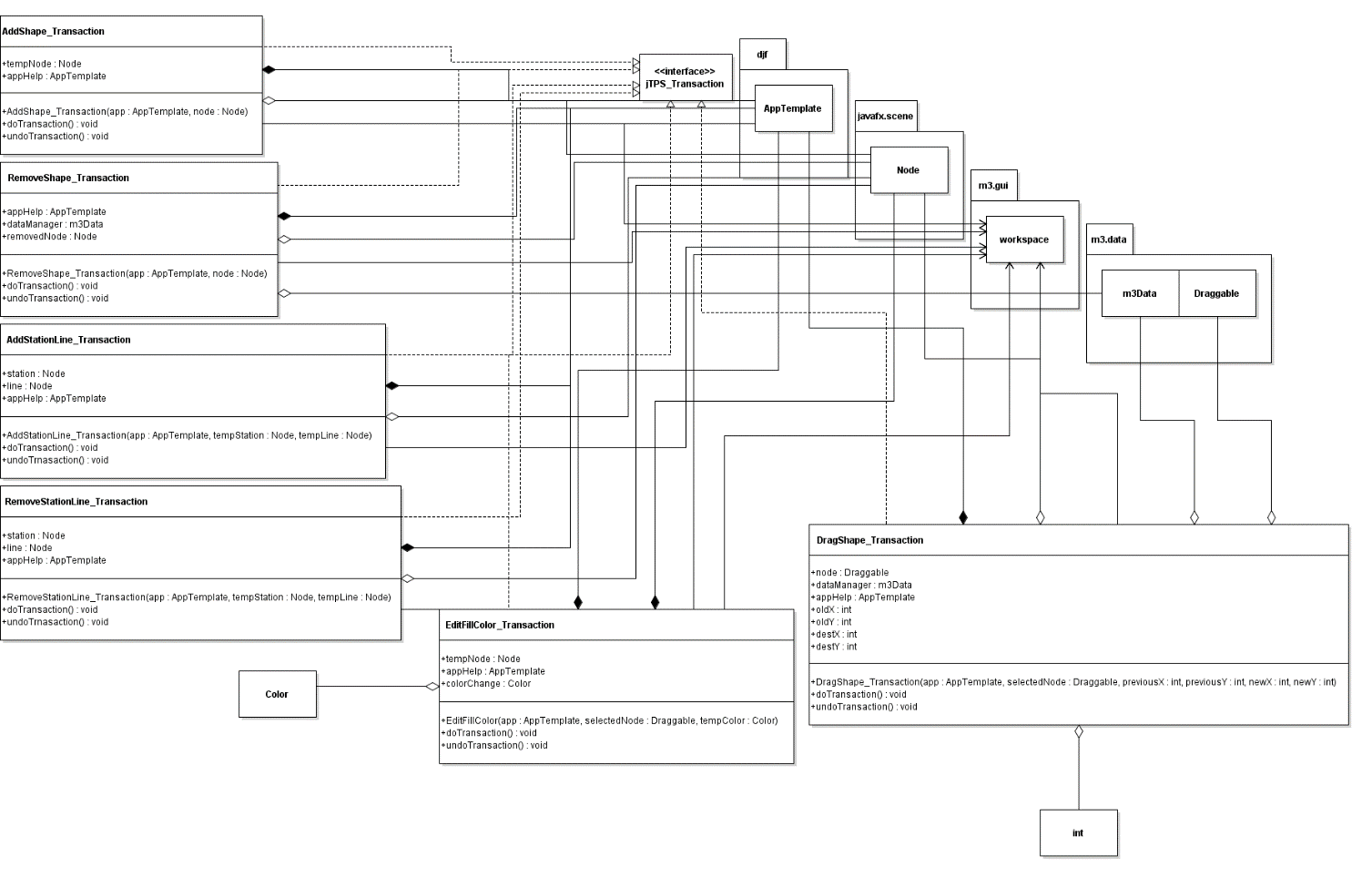
****

**Figure 3.5 Detail AppGUI UML Class Diagram**

|  |  |  |
| --- | --- | --- |
|  |  |  |

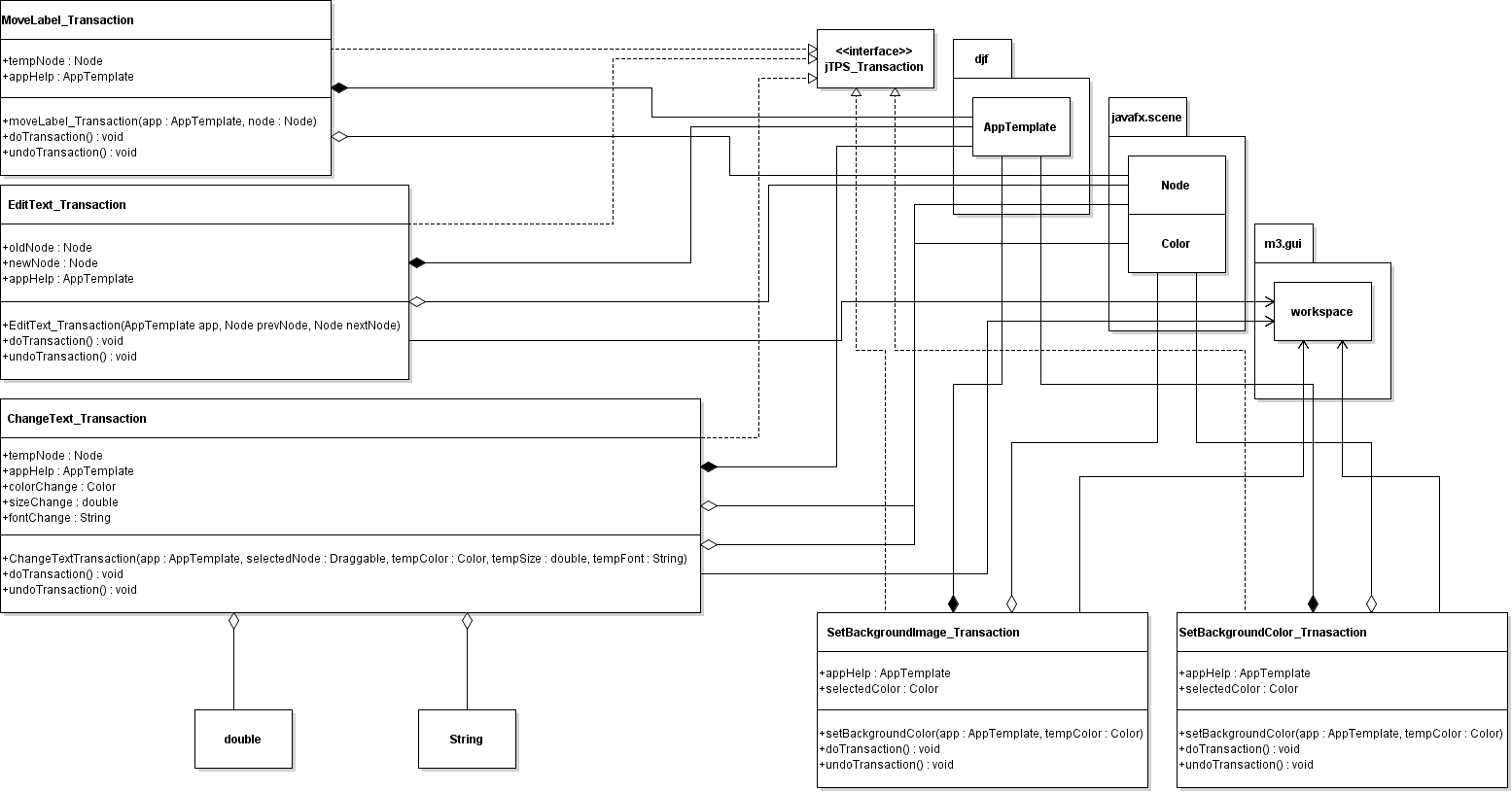
**Figure 3.6 Detail AppPropertyType, m3LanguagePropertyType, and m3State enum Class Diagram**

*Note:* This class encompasses the enum class for the djf and for the m3 classes, which are shown in tandem to save some space.



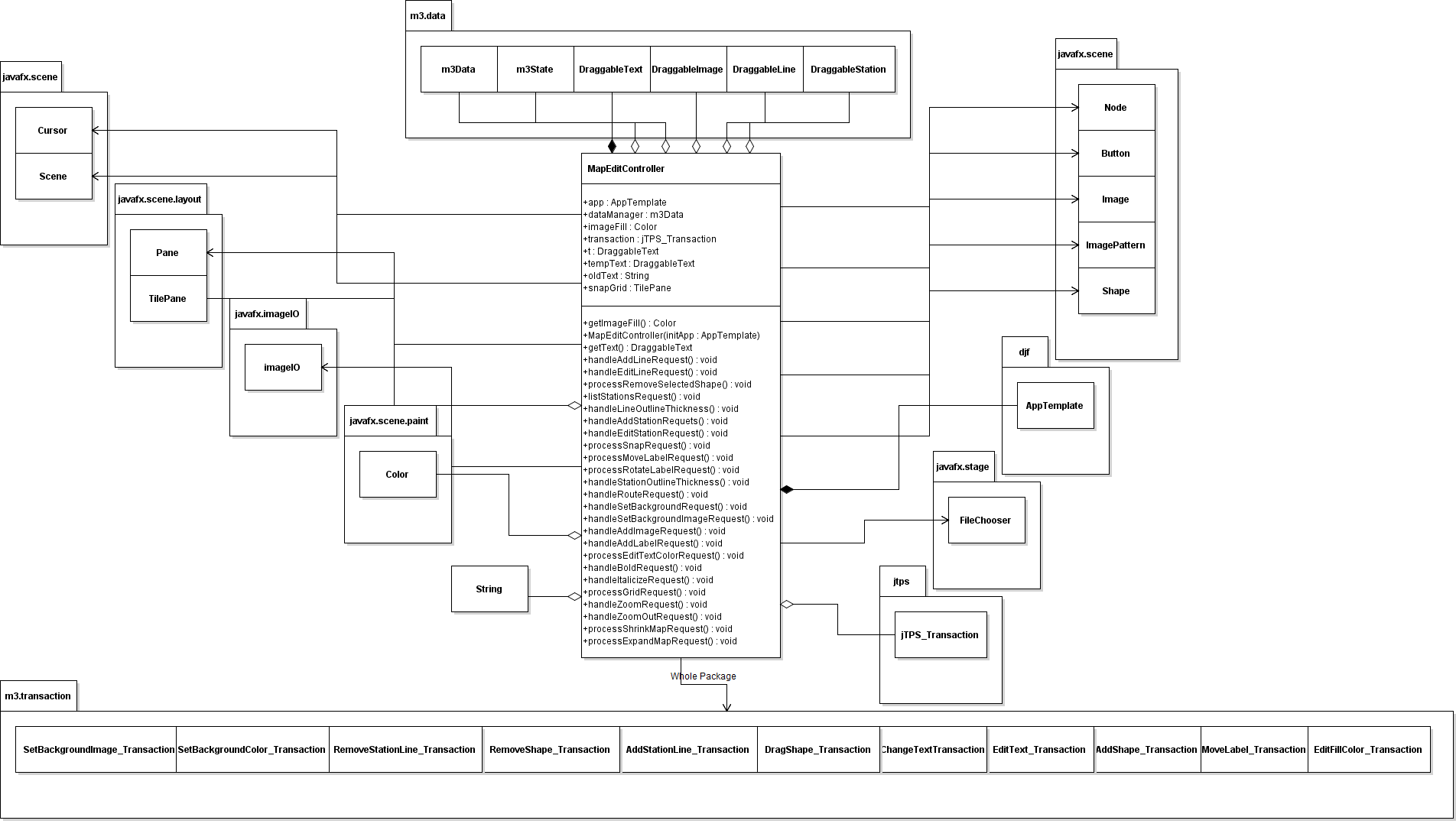
**Figure 3.7 Detail Tansactions UML 1 Class Diagram**

*Note:*This consists of Adding/Removing shapes, Adding/Removing stations from subway lines, and Dragging Shapes.

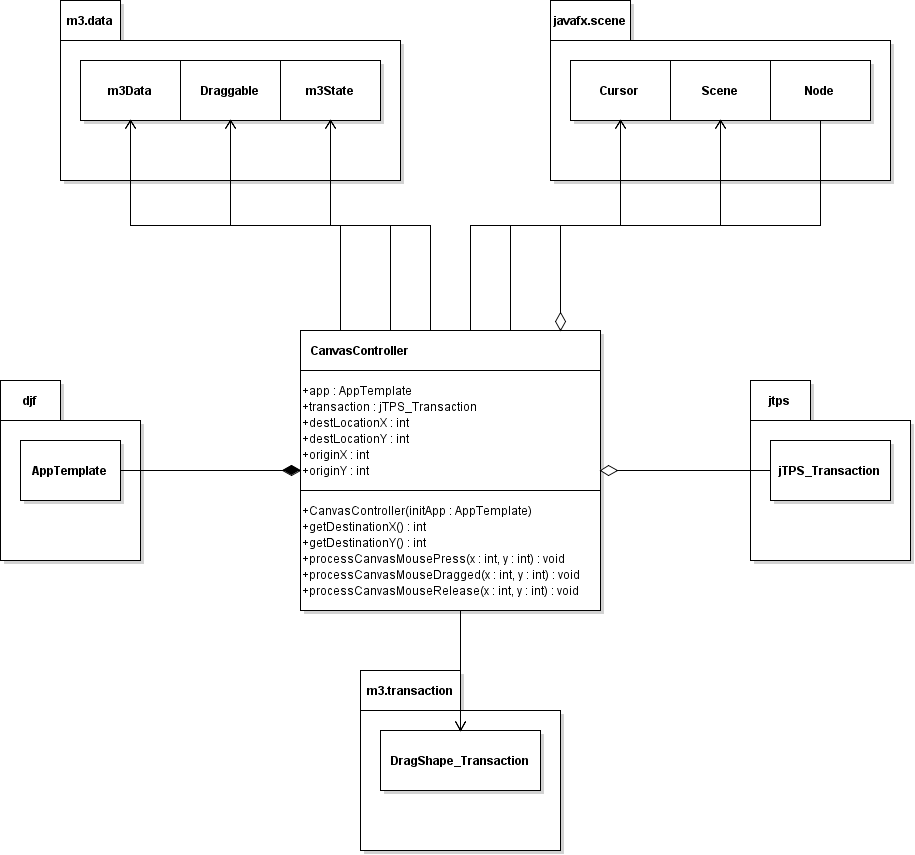
****

**Figure 3.8 Detail Transaction UML 2 Class Diagram**

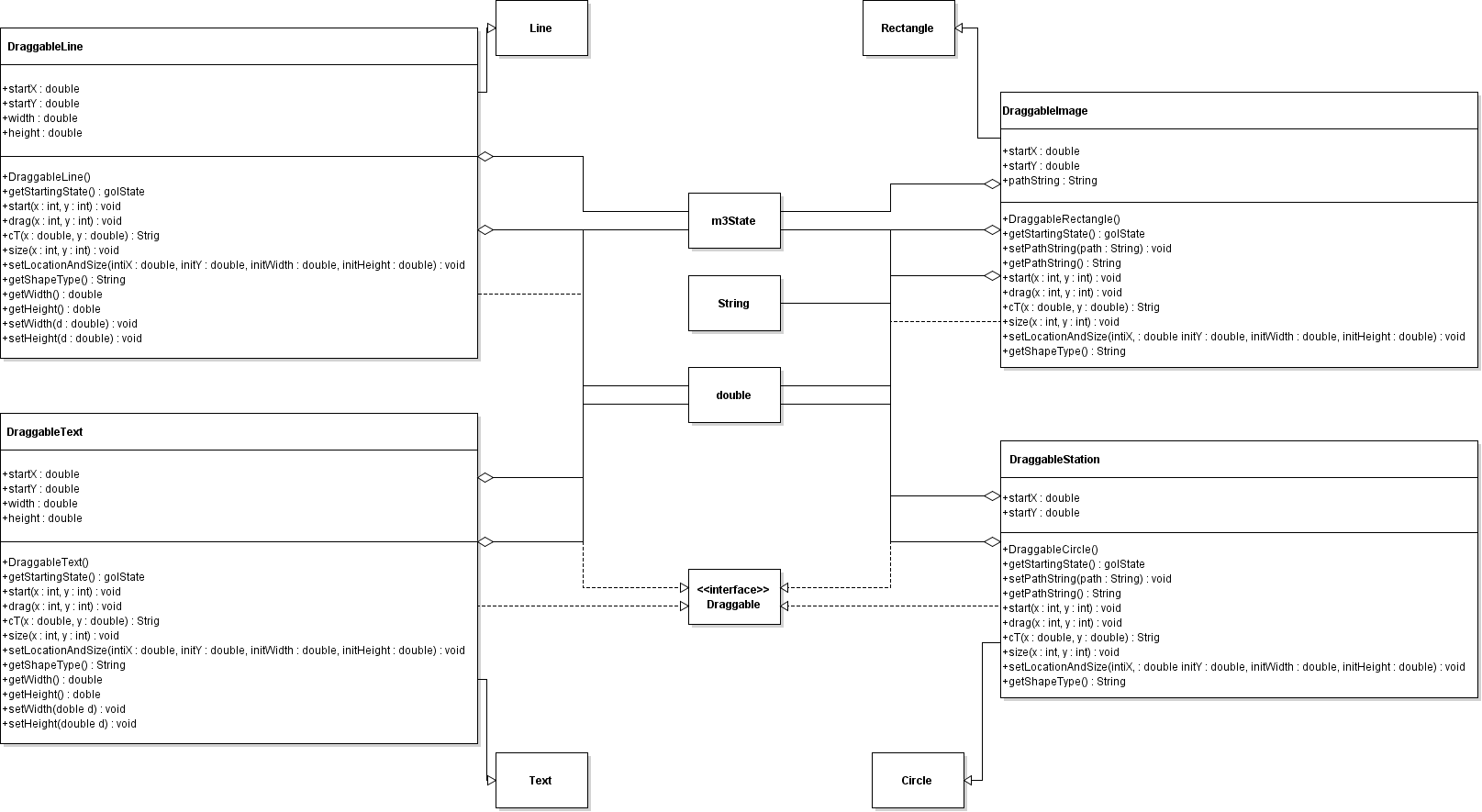
***Note:*** This consists of Shifting label around a station, Editing text, Changing text color/font/size, Setting the background as an Image, or setting the background Color.

****

**Figure 3.9 Detail MapEditController UML Class Diagram**

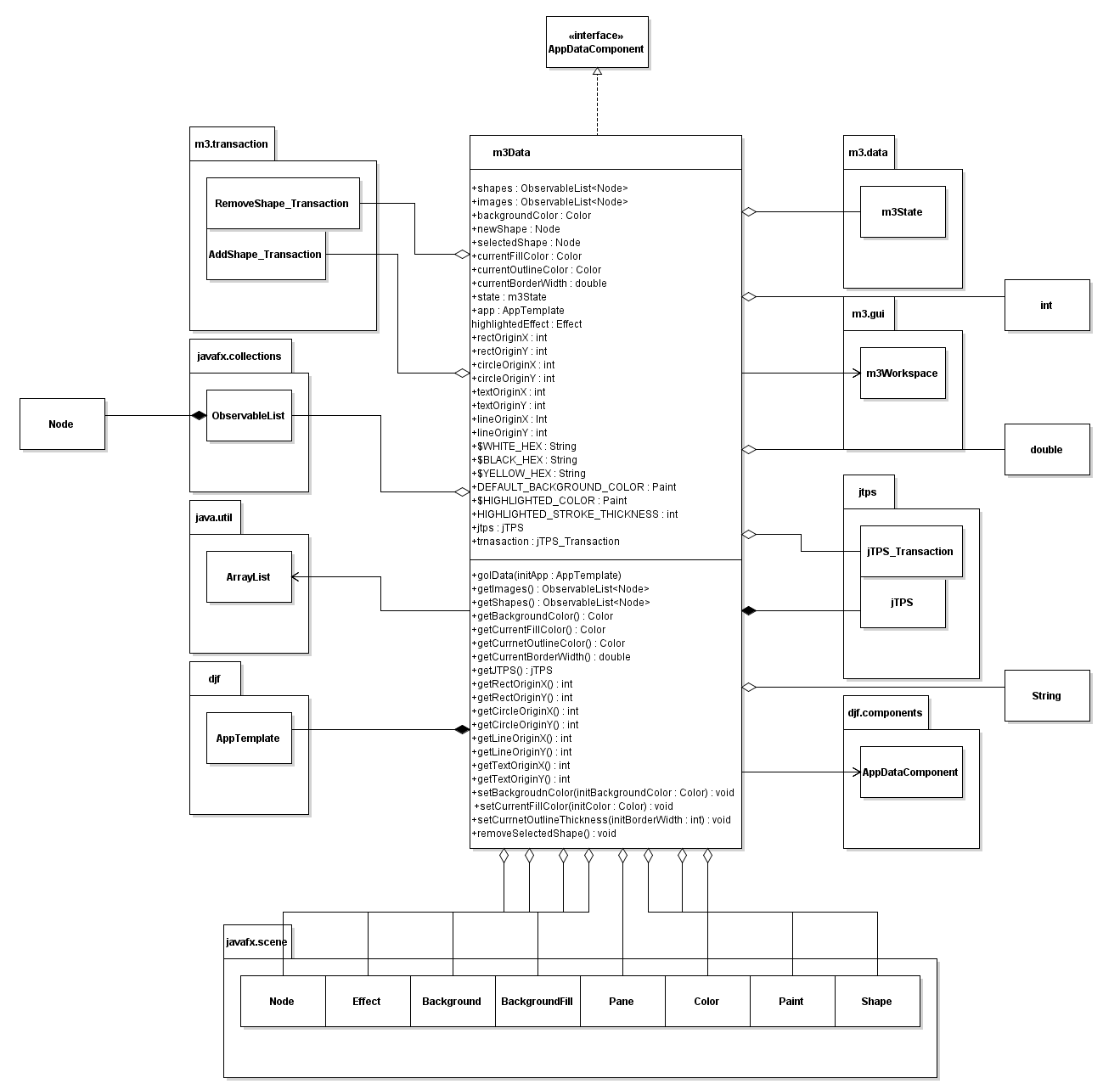
****

**Figure 3.10 Detail CanvasController UML Class Diagram**

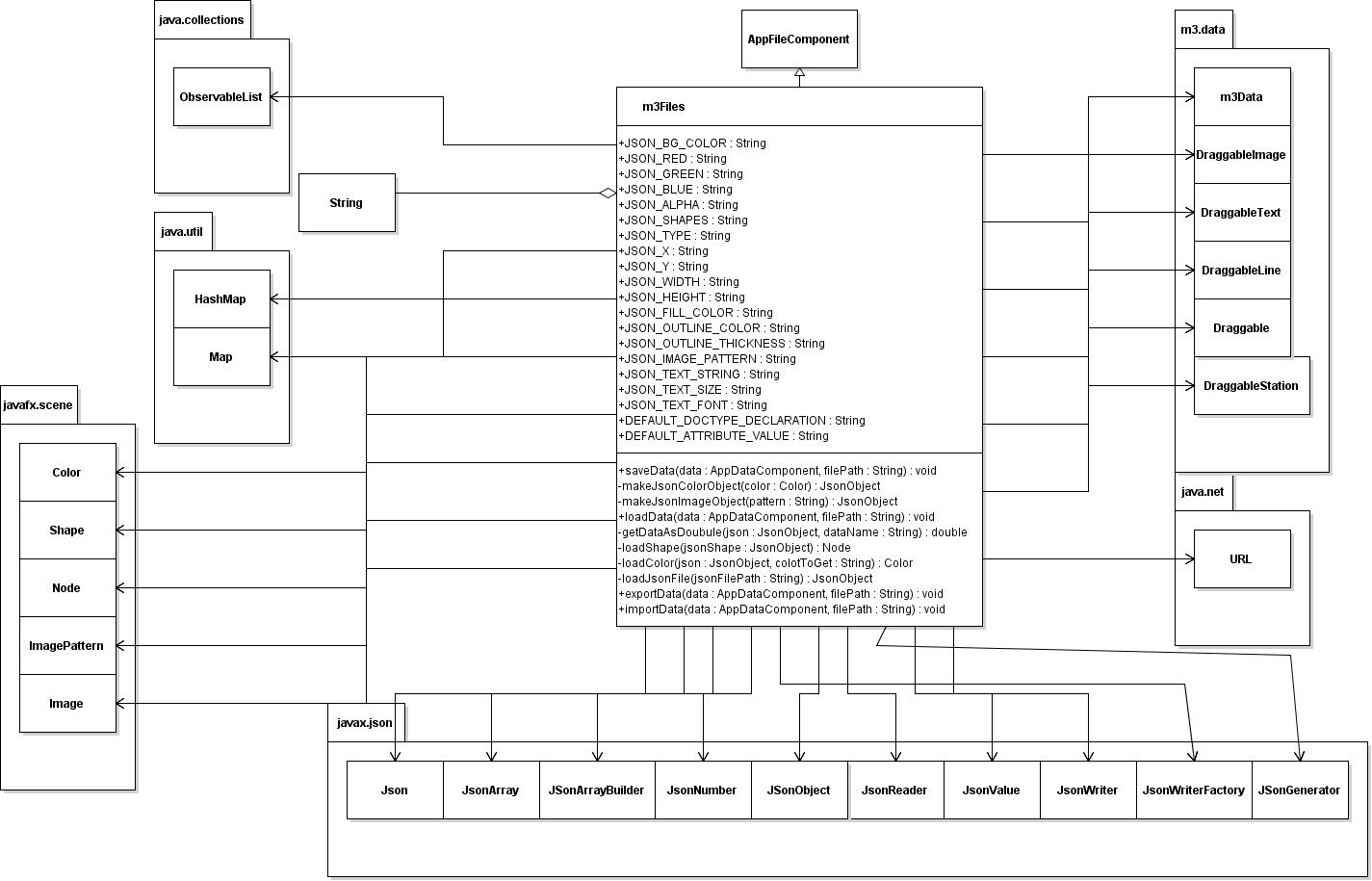
****

**Figure 3.11 Detail Draggable Classes UML Class Diagram**

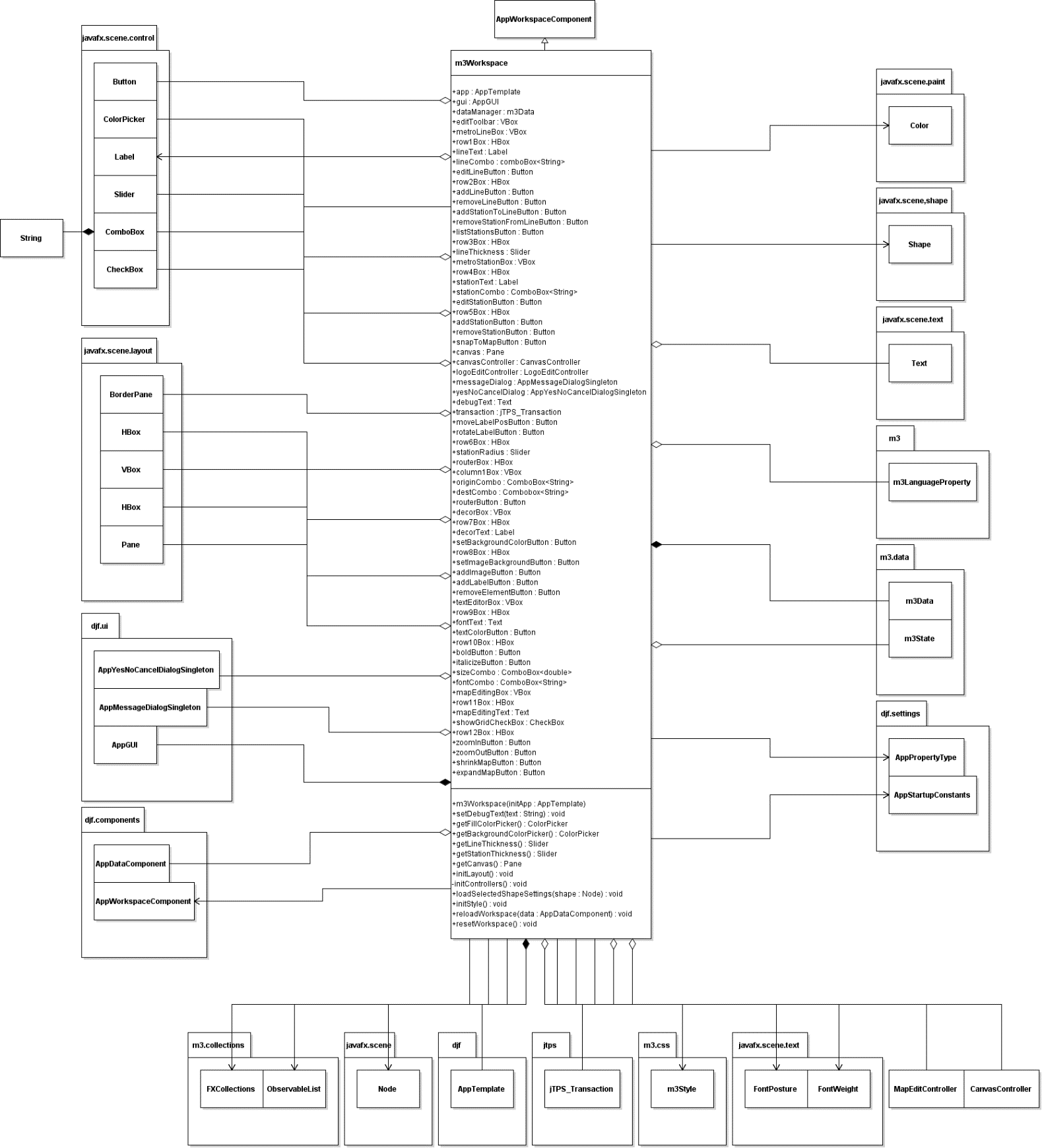
*Note:* This consists of all the draggable classes including DraggableText, DraggableStation, DraggableImage, and DraggableLine.



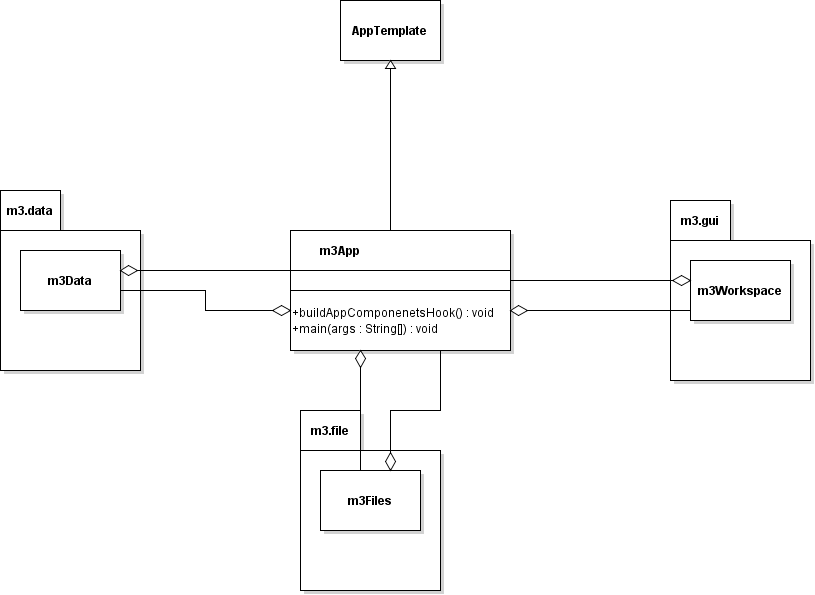
**Figure 3.12 Detail m3Data UML Class Diagram**

****

**Figure 3.13 Detail m3Files UML Class Diagram**

****

**Figure 3.14 Detail m3Workspace UML Class Diagram**

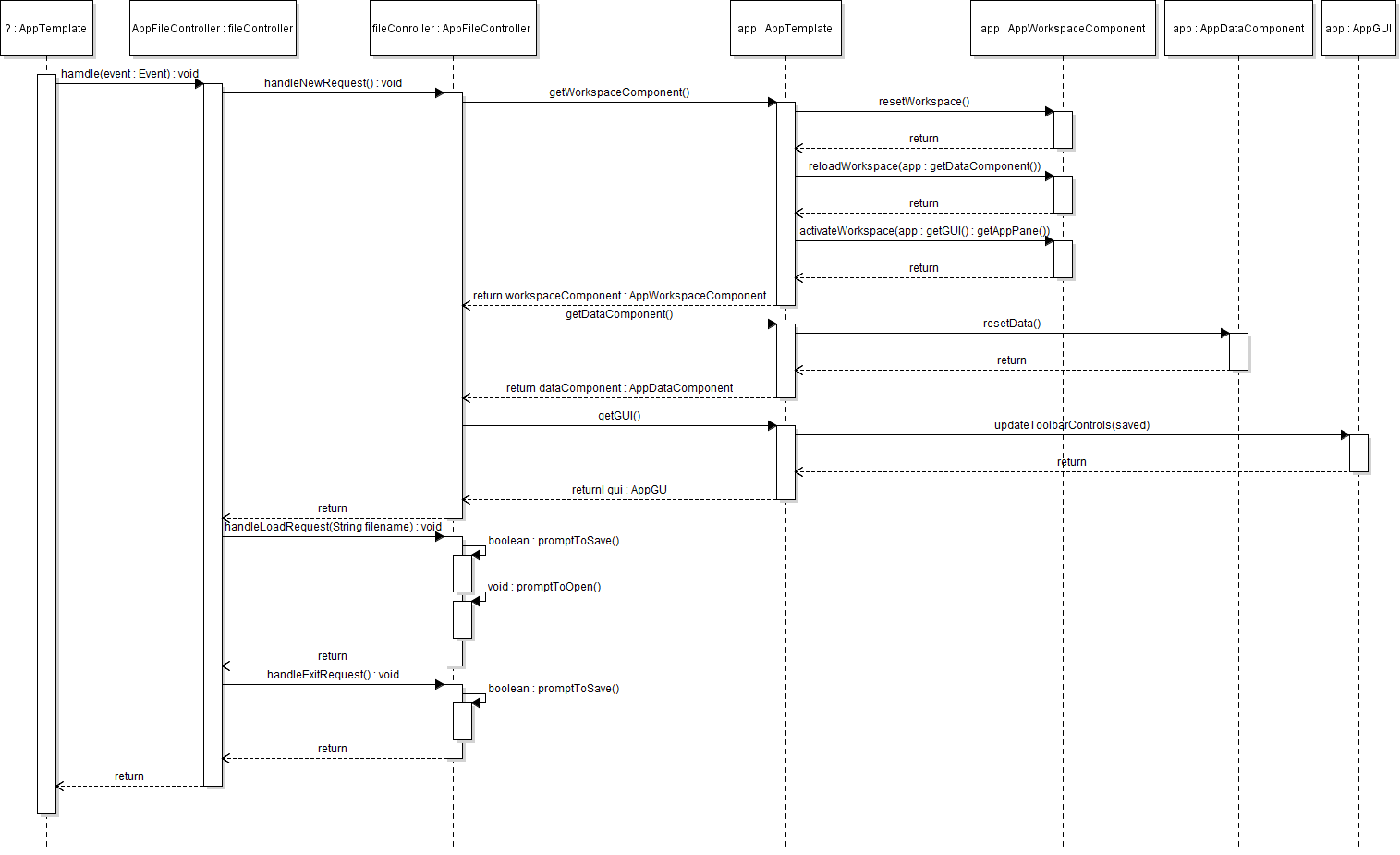
****

**Figure 3.15 Detail m3App UML Class Diagram**

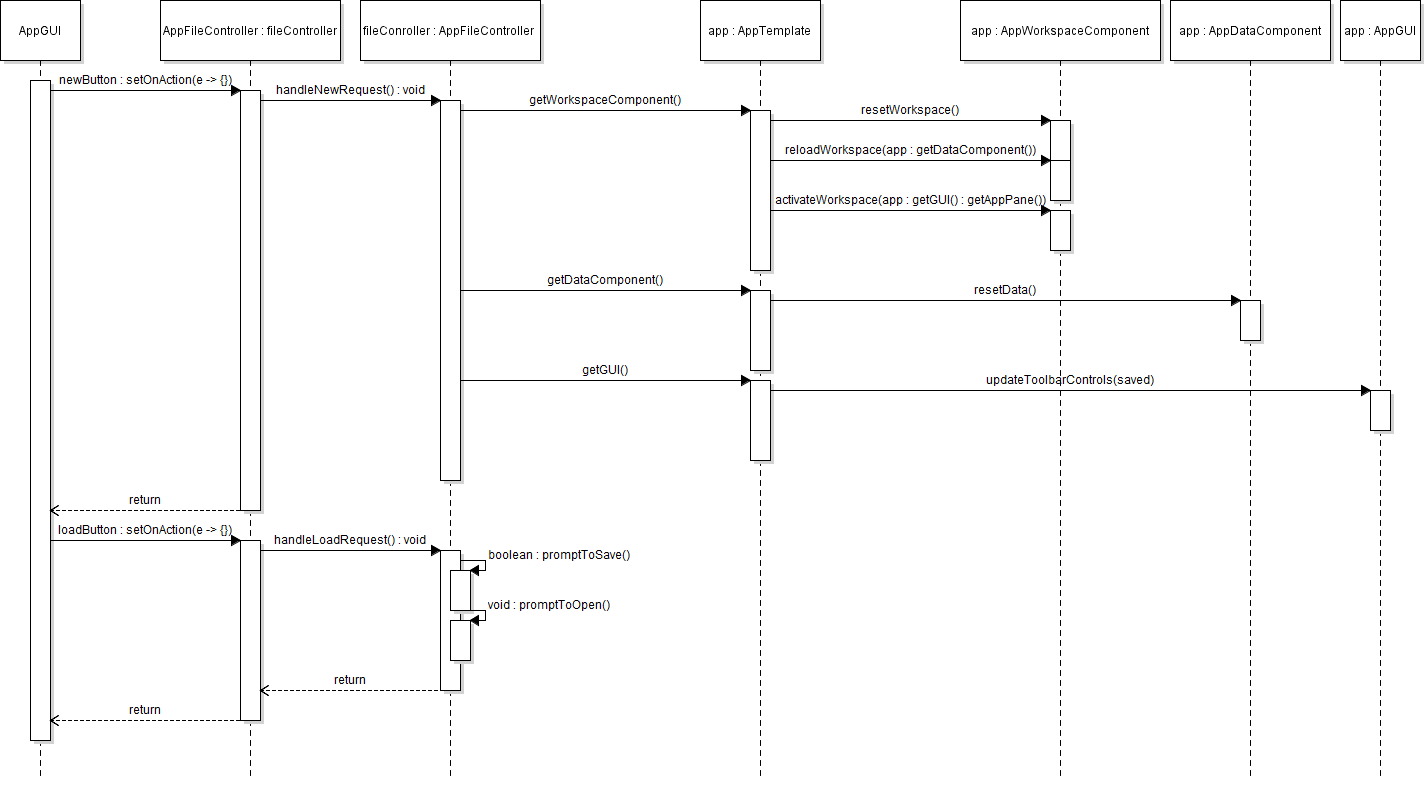
Any classes that UML’s were not made of, were made by Professor Richard Mckenna. Any classes in the djf UML’s were originally Professor Mckenna’s with some modifications made.

1. Method-Level Design Viewpoint

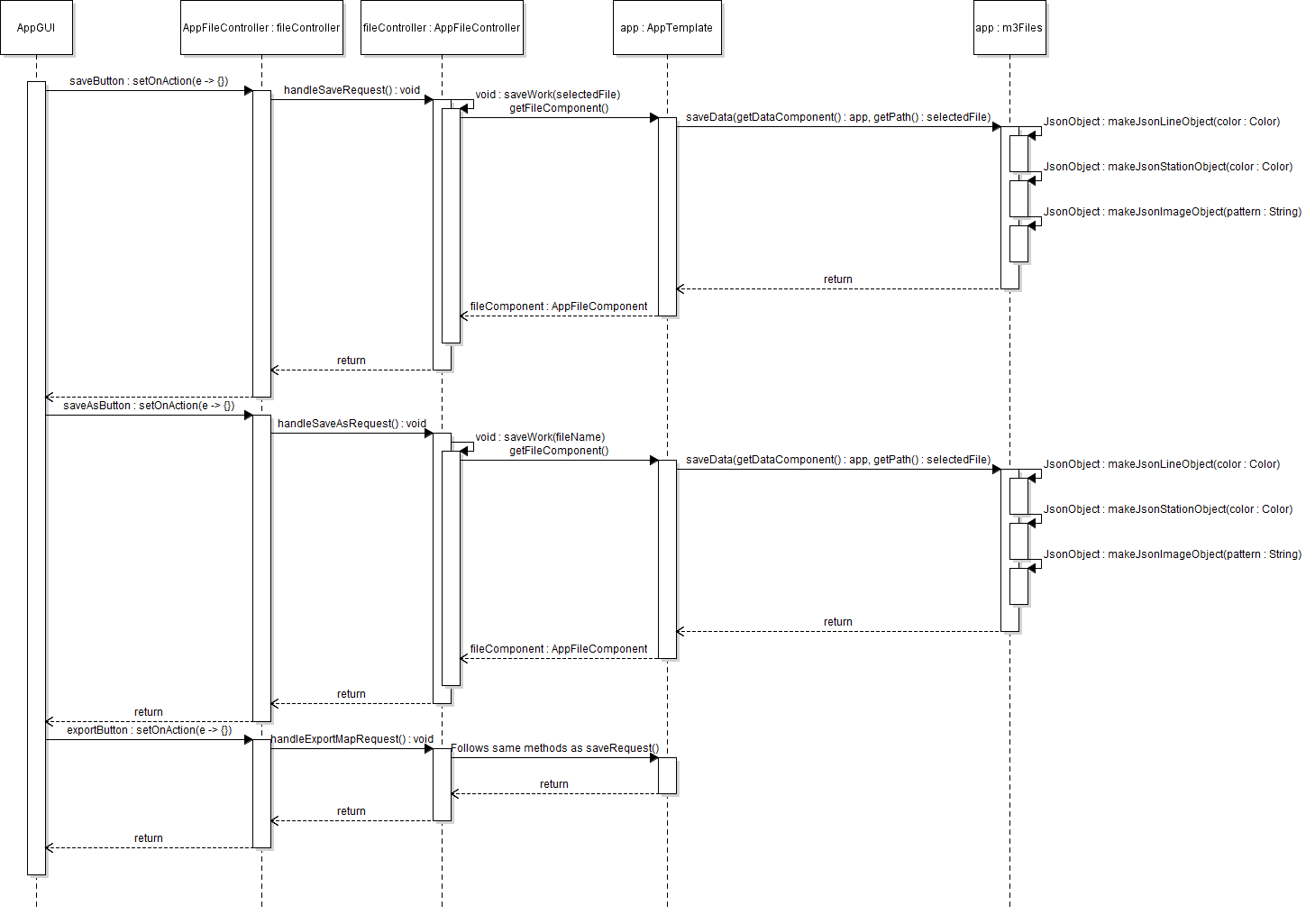
Now that the general architecture of the classes has been determined, it is time to specify how data will flow through the system. The following UML Sequence Diagrams describe the methods called within the code to be developed in order to provide the appropriate event responses. Some event responses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

****

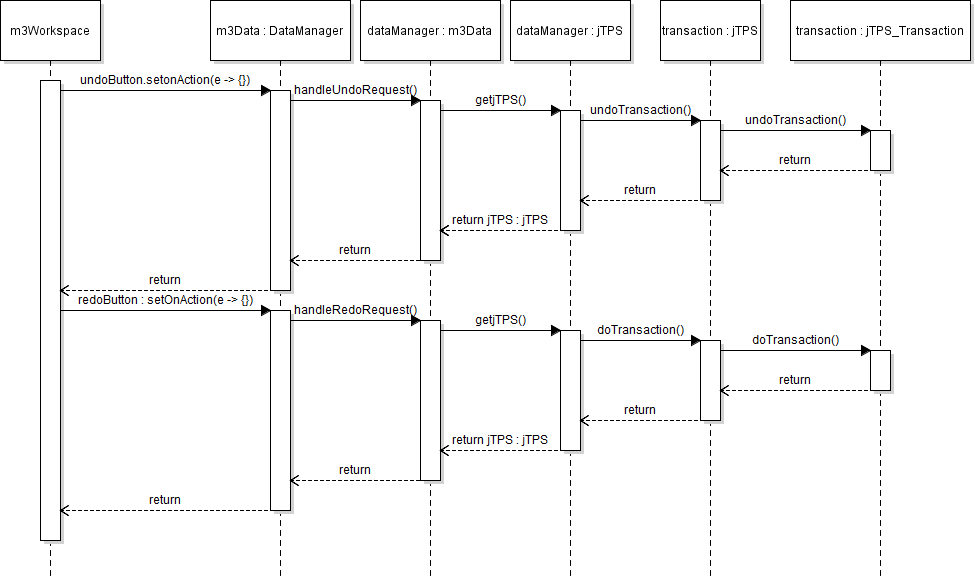
**Figure 4.1 WelcomeDialog UML sequence diagram**

****

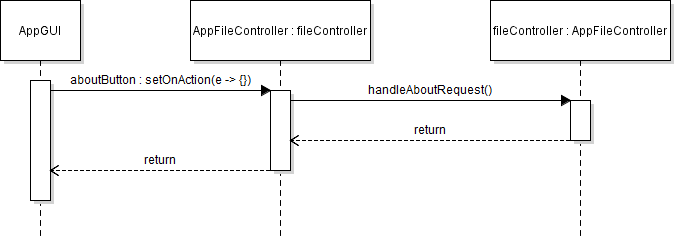
**Figure 4.2 File Toolbar Sequence (New and Load button)**

****

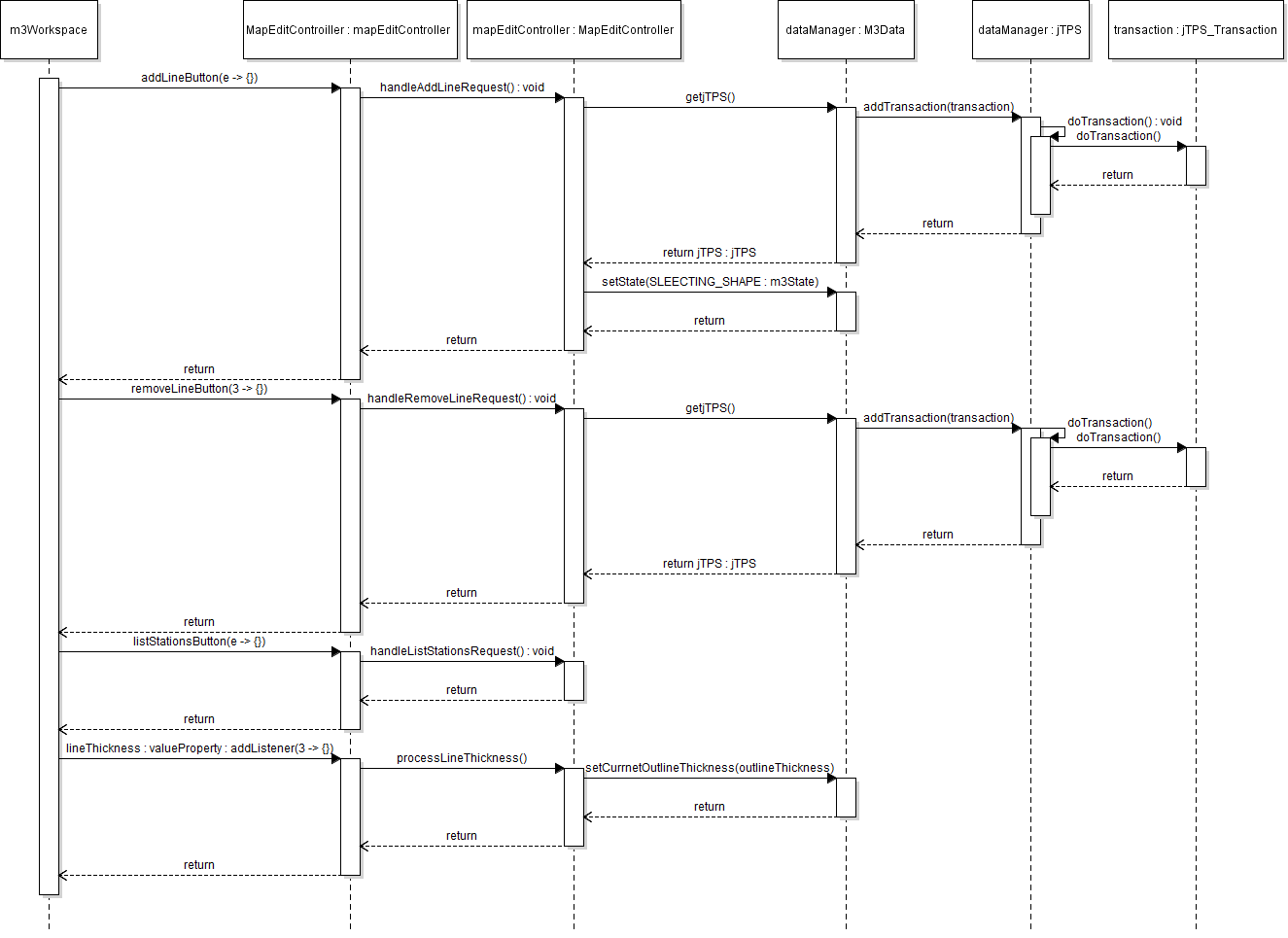
**Figure 4.3 File Toolbar Sequence (Save, Save as, and Export button)**

****

**Figure 4.4 Undo/Redo Toolbar UML sequence diagram**

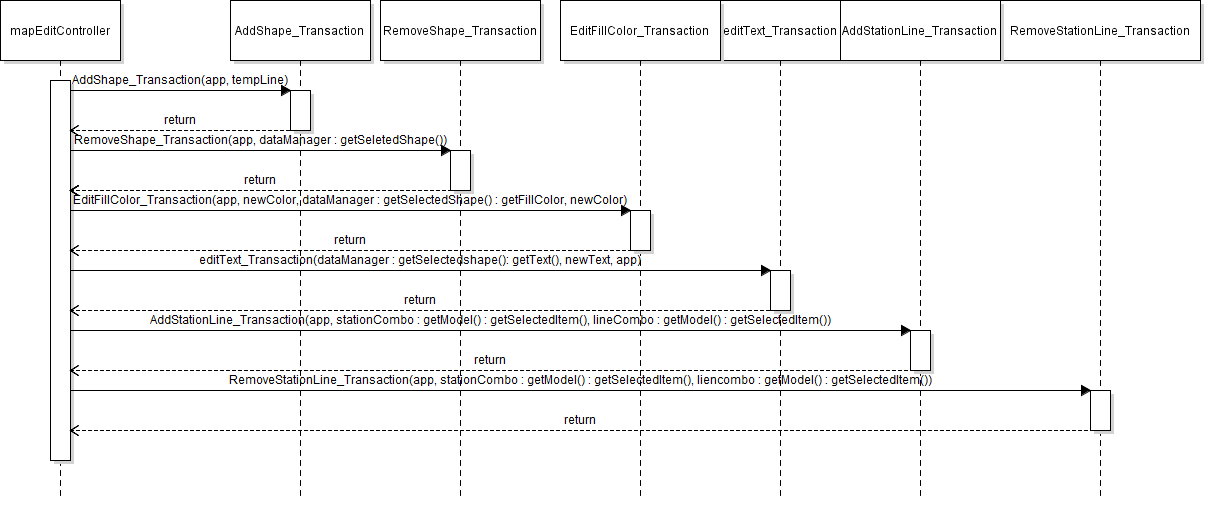
****

**Figure 4.5 About Toolbar UML sequence diagram**

****

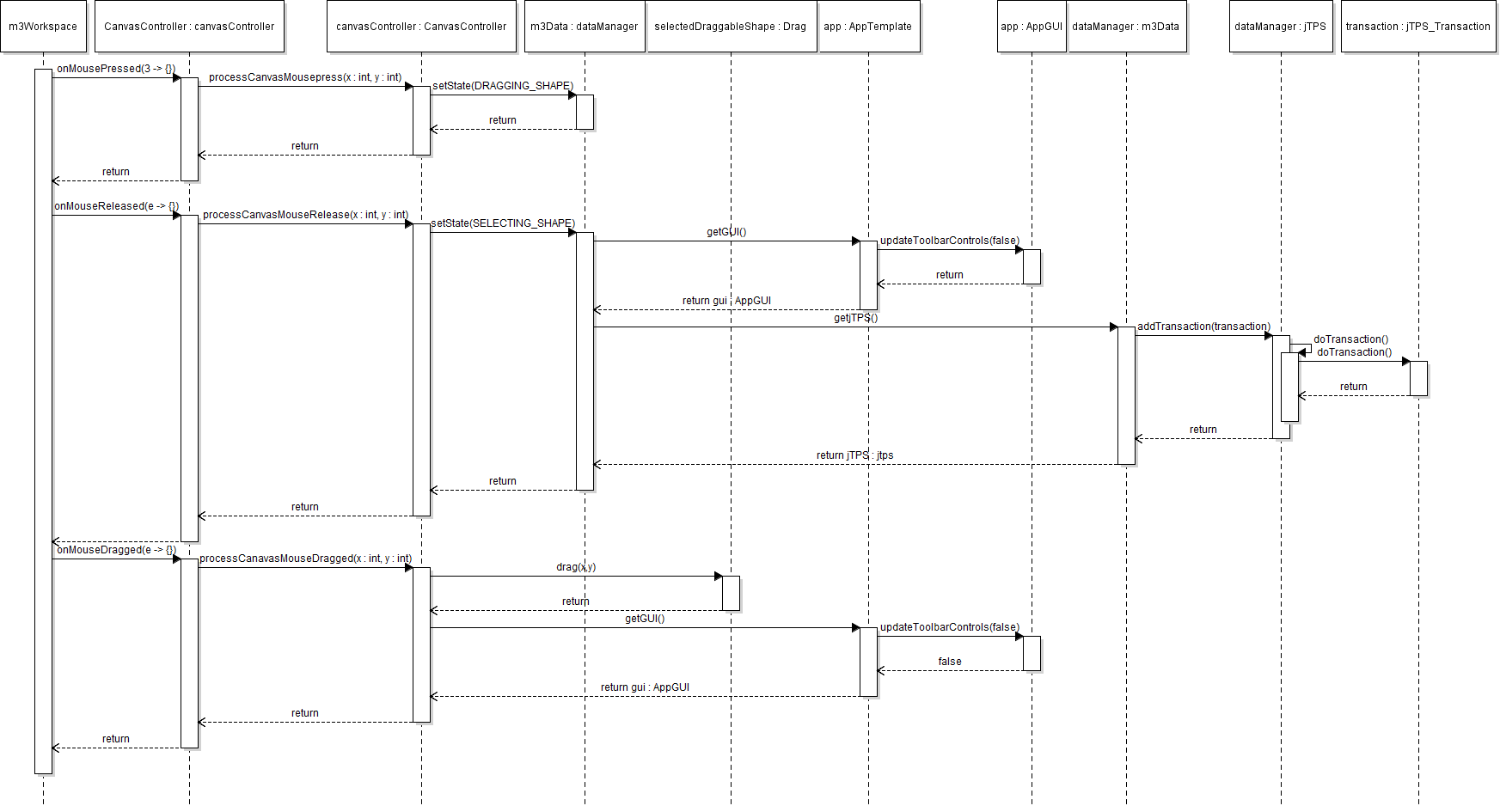
**Figure 4.6 Metro Line Toolbar UML sequence diagram**

*Note:* editLineButton, addStationToLineButton, removeStationfromLineButton buttons all look the same as the addLine and removeLine UML, so to save space we only describe their transactions in the next chart.



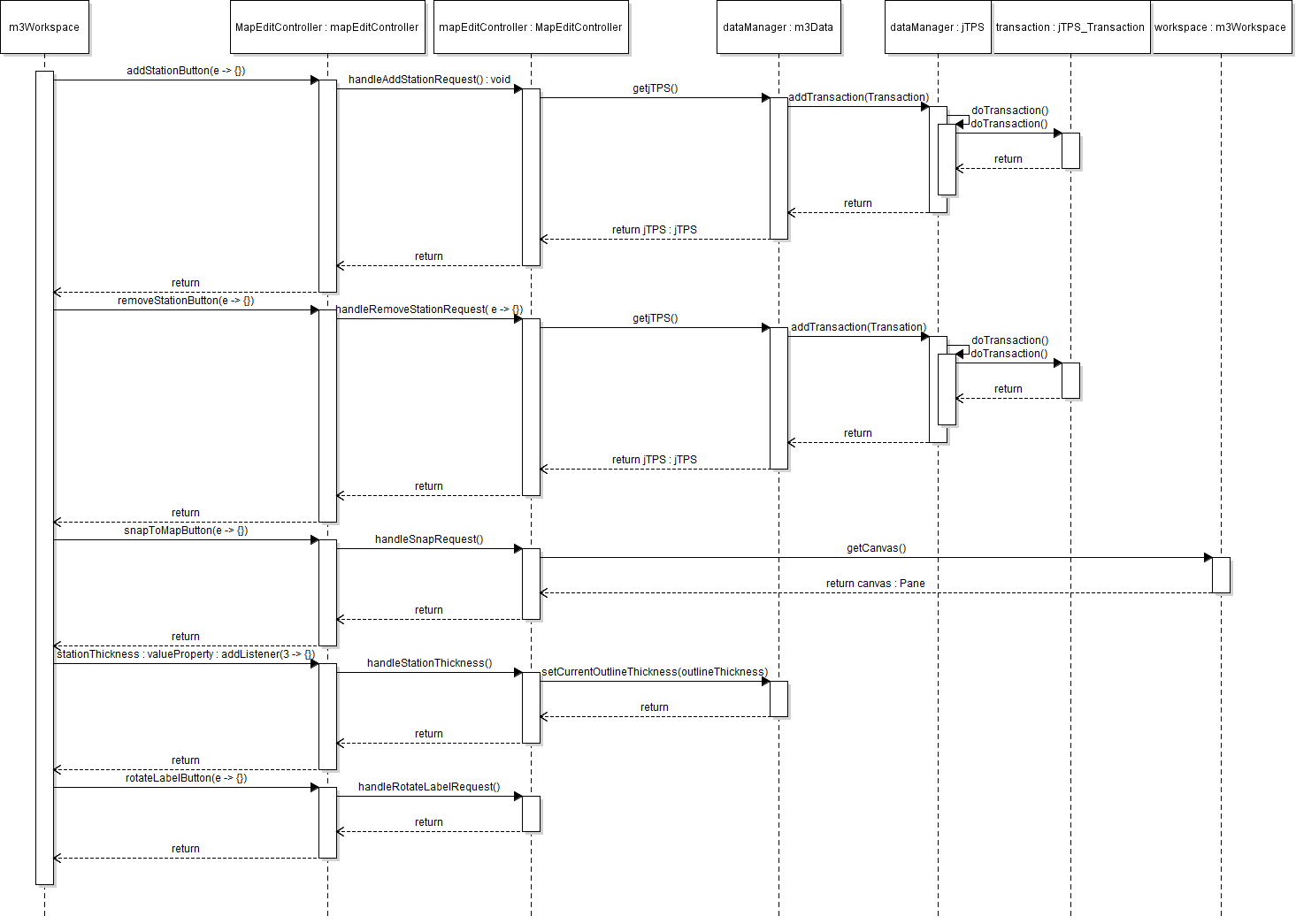
**Figure 4.7 Metro Line Toolbar Supplement UML sequence diagram**

*Note:* This is the transaction UML. Editing the line had editText and editFillColor transactions.



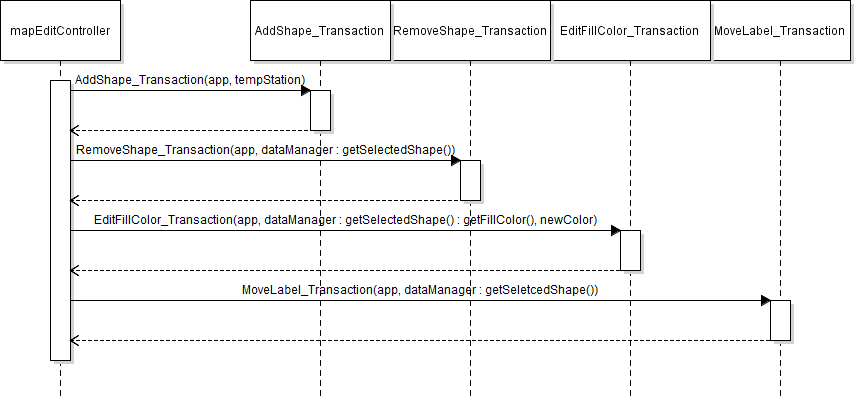
**Figure 4.8 Canvas Controller UML sequence diagram**

*Note:* This canvas controller is simply for dragging around a shape. When it comes to moving the end of a metro line, the system will set the state to DRAGGING\_LINE\_END which will help to extend the line to where it needs to go.

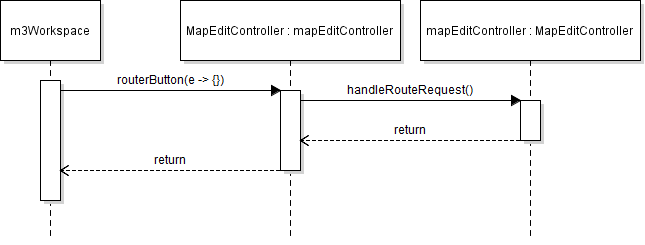
****

**Figure 4.9 Metro Station Toolbar UML sequence diagram**

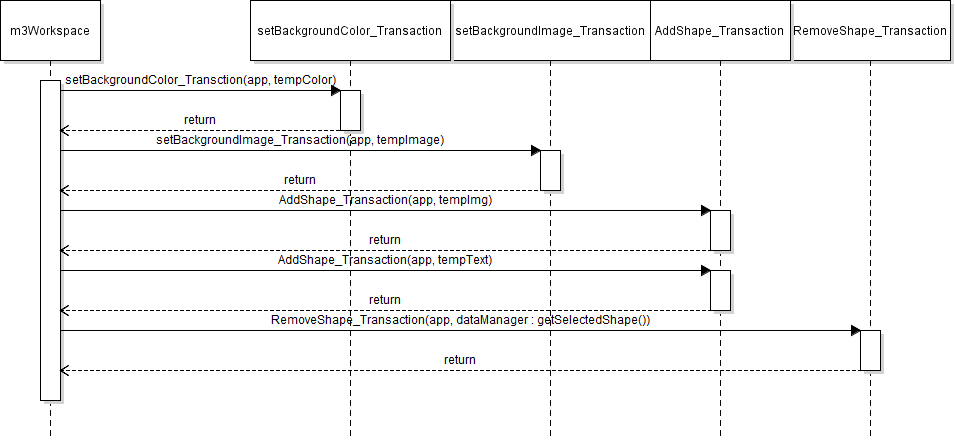
*Note:* moveStationLabel, changeStationFillColor buttons work the same as addStationButton and removeStationButton, so we will make UML for their individual functionality in the next chart.



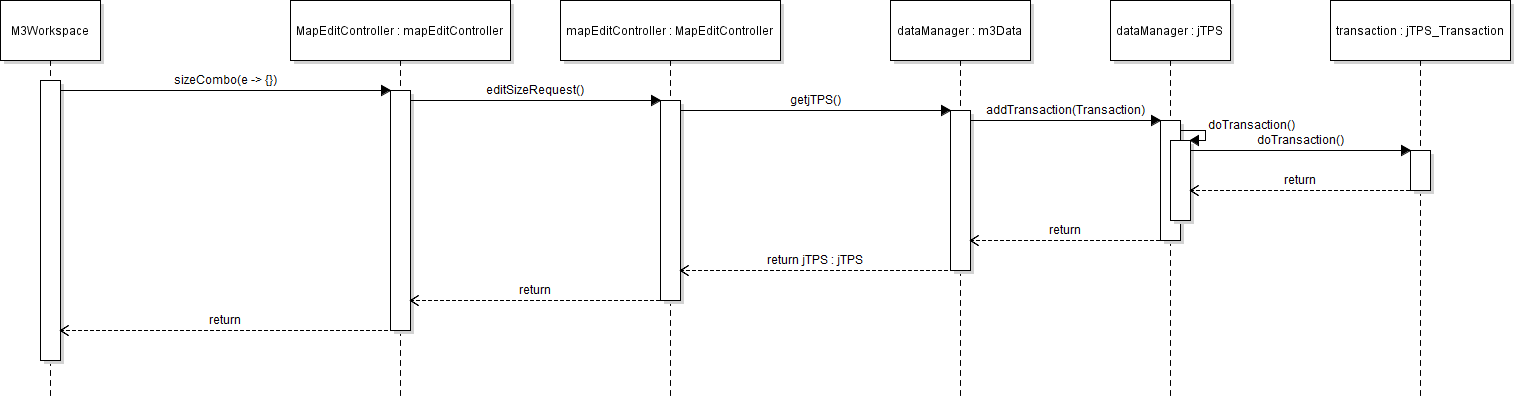
**Figure 4.10 Metro Station Toolbar Supplement UML sequence diagram**

****

**Figure 4.11 Station Router Toolbar UML sequence diagram**

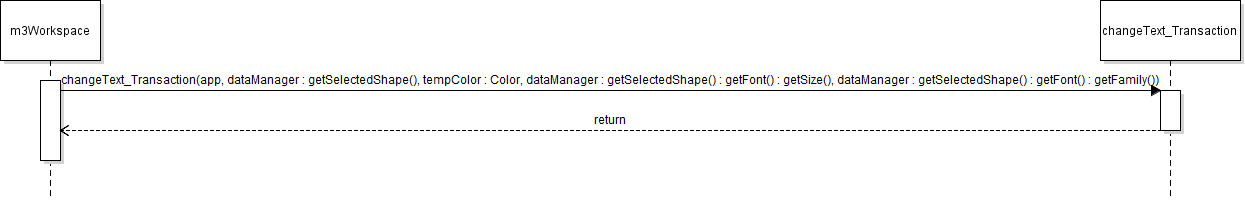
****

**Figure 4.12 Décor Toolbar UML sequence diagram**

****

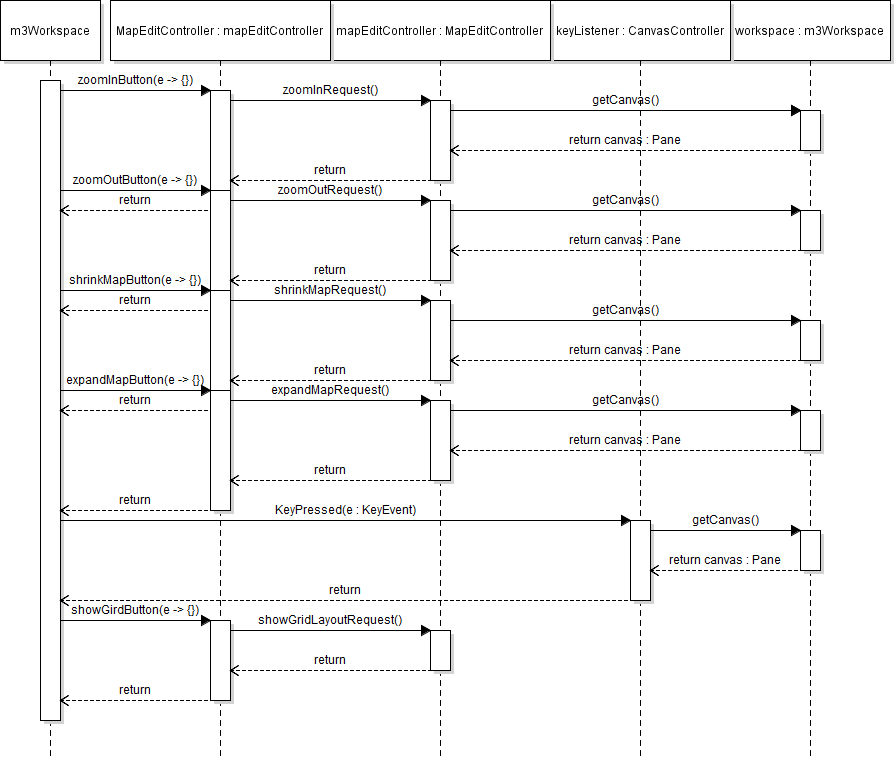
**Figure 4.13 Font Toolbar UML sequence diagram**

*Note:* fontCombo, boldButton, and italicizeButton requests all work the same as sizeCombo, so we are going to use one UML to represent them all, and show their appropriate transactions on the next UML sequence diagram

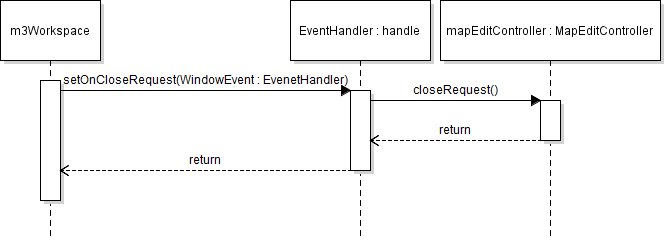


**Figure 4.14 Font Toolbar UML sequence diagram**

*Note:* This transaction is the only one necessary for when one bolds, italicizes, changes font color, font size, or font family. This will account for each, and if the use undo then they can go back to the previous version of the text.



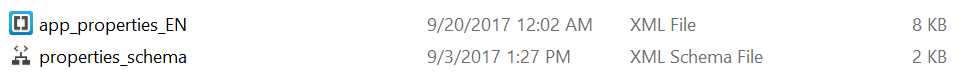
**Figure 4.14 Navigation Toolbar UML sequence diagram**

****

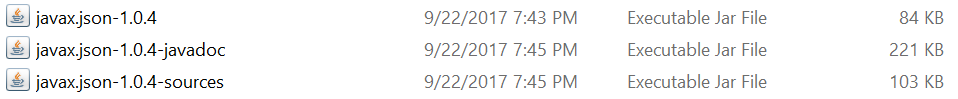
**Figure 4.15 Title Bar UML sequence diagram**

1. **File Structures and Formats**

Note that the djf framework, properties manager, and m3 files will be provided inside a zip file which will be ready once one downloads, and imports the project. The file structure will have a java package, with their individual classes inside of it. For example, we have a package called m3.transaction which consists of all the java classes for their appropriate transactions. Once importing, the classes will be grouped in the way they need to be grouped, including their art files for the buttons as well. Alongside everything, there will be a data class that will consist of 2 XML files. These 2 XML files will be contained inside of a data folder inside of the m3Folder. These 2 files are named app\_properties\_EN and properties\_schema.



These 2 will be included inside of the data folder when importing the project. The project will also consist of a json, Javadoc, and sources JAR files for json to work properly. This will help to run Json files properly when loading, saving, and exporting files.



1. **Supporting Information**

This document is a reference for how to go about implementing the m3 code when programming. This next section will include a table of contents which will explain where all the appropriate sections are in the SDD.

* 1. **Table of contents**

1. Introduction 2
2. Purpose 2
3. Scope 2
4. Definitions, acronyms, and abbreviations 2
5. References 4
6. Overview 4
7. Package-Level Design Viewpoint 5
8. M3 and djf framework overview 5
9. Java API Usage 6
10. Java API Usage Descriptions 7

3. Class-Level Design Viewpoint 12

4. Method-Level Design Viewepoint 27

5. File Structure and Formats 36

6. Supporting Information 36

1. Table of contents 36
2. Appendixes 36
   1. **Appendixes**

N/A