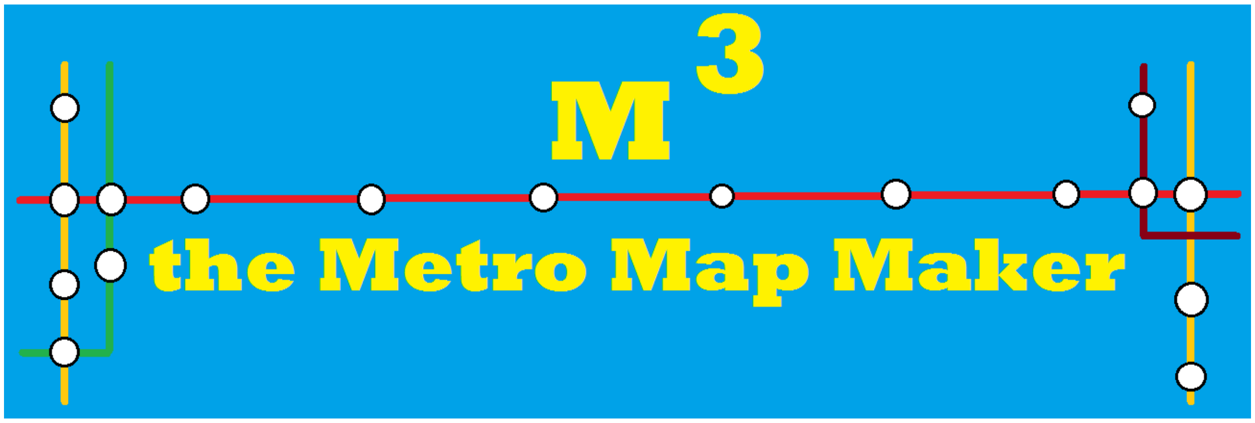
The *Metro Map Maker* ™0
Software Design Description



Author: Fanng Dai
Debugging Enterprises™
October, 2017
Version 1.0

Abstract: This document describes the software design for Metro Map Maker, a program developed to help map makers build graphical representations of public transportation systems.

Based on IEEE Std 830 ™-1998 (R2009) document format

1 Introduction

This is the software Design Description (SDD) for the Metro Map Maker™ 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 Metro Map Maker 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. The intended audience for this document is all the members of the development team, this includes the instructor, TA, and of course, myself. 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. Upon finishing this document, one should clearly visualize how the application will operate.

1.2 Scope

Metro Map Maker will be an application for map makers to easily make and edit public transportation maps. Tools developed for its construction includes DesktopJavaFramework, PropertiesManager, and jTPS, as some of the functionalities required for this application are provided or can be slightly modified to adjust. Along with the frameworks, this application will also use the external JavaX.JSON library to save and load JSON files. So, this design contains design descriptions for only the application unless modifications are made within the framework. Then, only the modifications are included. Note that Java is the target language for this software 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.

CSS – Cascading Style Sheet, a stylesheet language written in either HTML or XML which describes the appearance of the project.

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 within 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.

JSON – JavaScript Object Notation, human-readable file used to store and load data objects.

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

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.

Use Case Diagram – A UML document format that specifies how a user will interact with a system.

XML – eXtensible Markup Language, human-readable data used to store and load data.

1.4 References

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

Metro Map Maker™SRS – Debugging Enterprises' Software Requirements Specification for the Metro Map Maker application.

1.5 Overview

This Software Design Description document provides a working design for the Metro Map Maker software application as described in the Metro Map Maker Software 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.

2 Package – Level Design Viewpoint

As mentioned, this design will encompass the Metro Map Maker application. In building this application, we will heavily rely on the Java API as well as an external JavaX.JSON 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 Metro Map Maker Overview

The Metro Map Maker will be designed with minor edits to the DesktopJavaFramework, PropertiesManager, and jTPS framework. Figure 2.1 specifies all the components to be developed and places all classes in home packages.

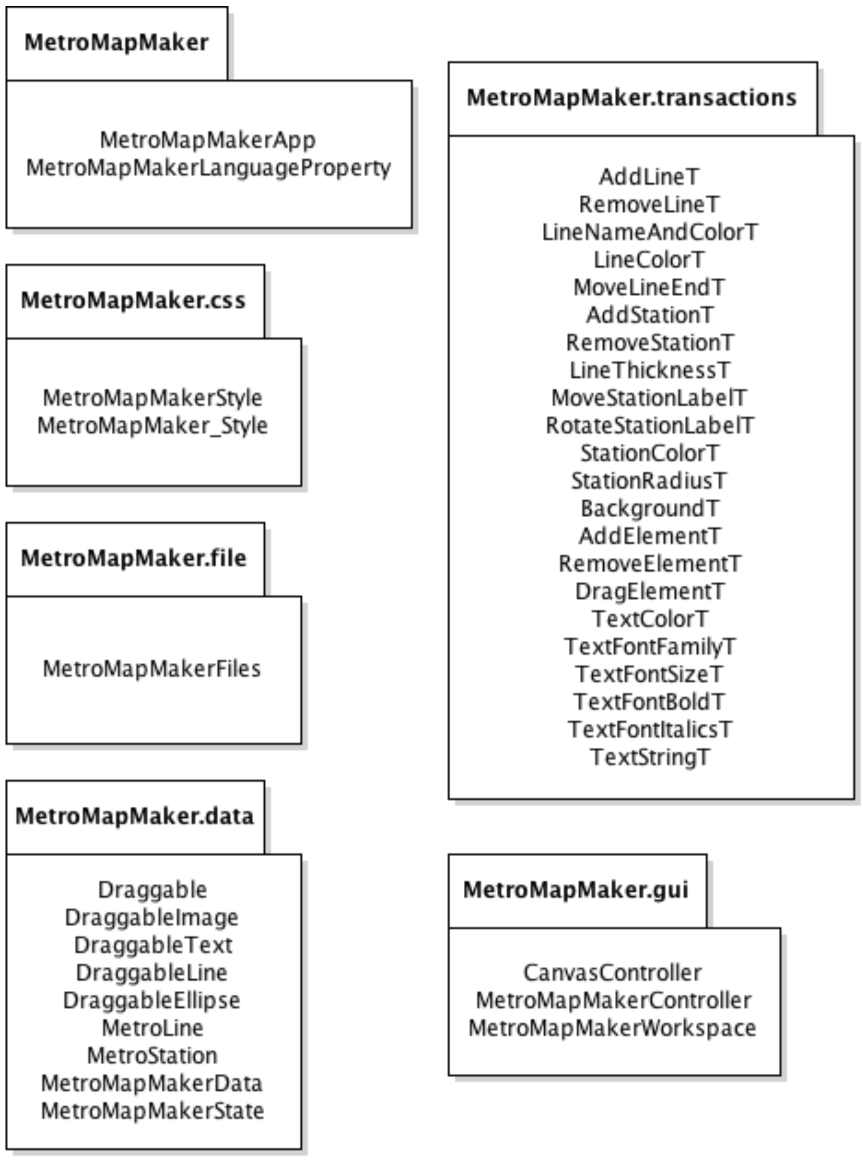


Figure 2.1: Metro Map Maker Package Overview

2.2 Java API Usage

The Metro Map Maker application will be developed using the Java programming languages. As such, this design will make use of the classes specified in Figure 2.2.

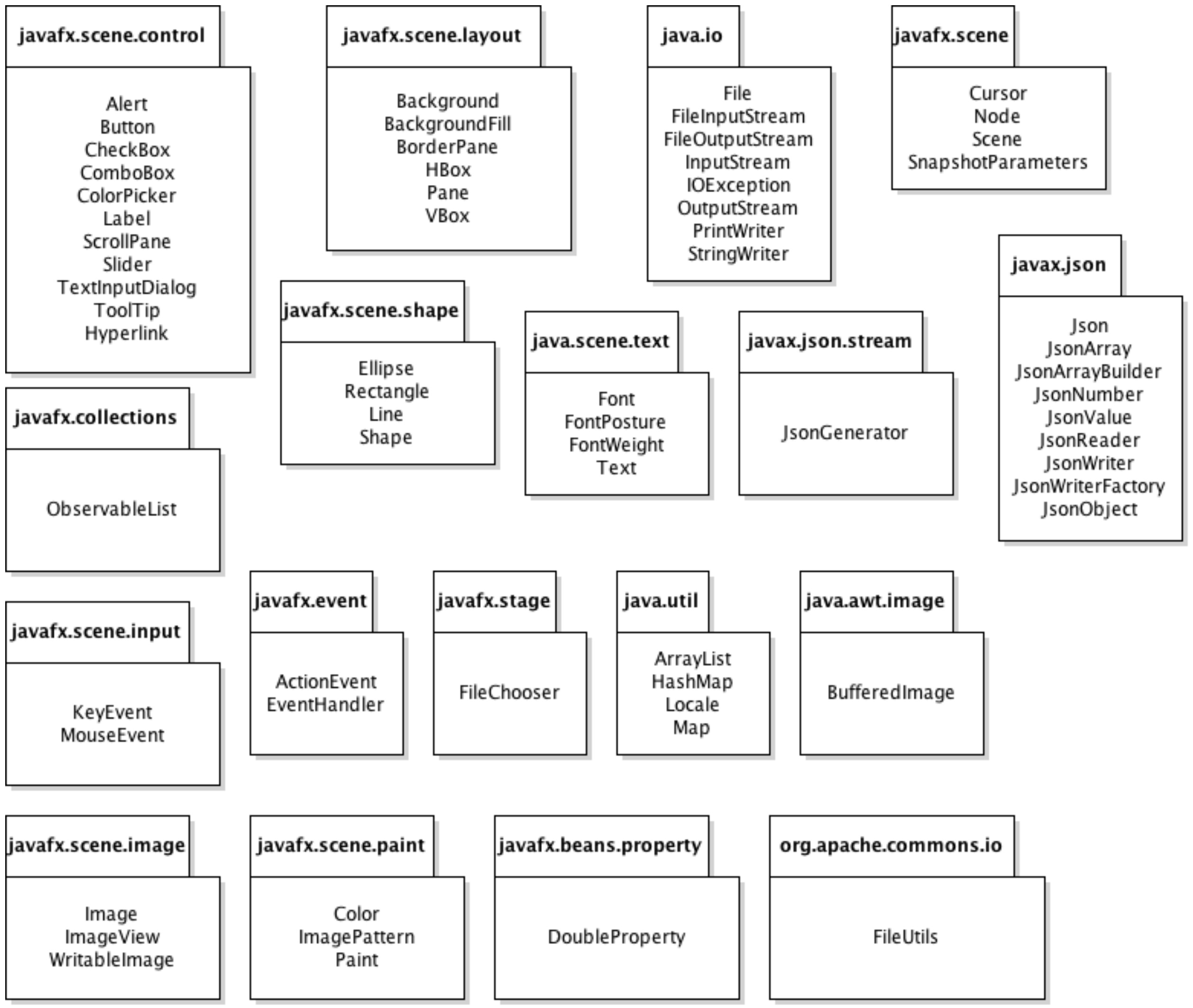


Figure 2.2: Java API Classes and Packages To Be Used

2.3 Java API Usage Descriptions

Tables 2.1-2.16 below summarizes how each of these classes will be used.

Class/Interface	Use
Alert	For informing user in occurrences of any changes or errors.
Button	For adding and removing stations, images, labels and elements. Also used for making a text bold and italics. Zooming in and out of the pane. (All elements in toolbar is a button.)
CheckBox	To enable and disable grid.
ComboBox	For choosing the metro lines and stations to edit. Also used to find the direction between two stations. Change the font size and family of the text.
ColorPicker	For coloring shapes of the user's choice. To color the metro lines and stations. For the background color unless image is applied.
Label	For identifying sections of the toolbar. (Left pane in workspace)
ScrollPane	For the use of when panes are not able to fit in the designated frame. For the left toolbar when user shrinks the size.
Slider	For changing the thickness of the metro line and the circle radius.
TextInputDialog	For prompting user for text input.
ToolTip	For showing additional information when node is hovered over by mouse.
Hyperlink	For showing user that the text is clickable. Used for welcome dialog recent works.

Table 2.1: Uses the classes in the Java API's javafx.scene.control

Class/Interface	Use
Background	For setting the background of the pane.
BackgroundFill	For filling the background with a color or image.
BorderPane	For the layout of the workspace.
HBox	For the layout of certain panes going from left to right. (Toolbar)
Pane	For the canvas (center) of the workspace.
VBox	For the layout of certain panes going from top to bottom. (Left pane of workspace.)

Table 2.2: Uses the classes in the Java API's javafx.scene.layout

Class/Interface	Use
File	For finding the path to a certain external file or saving a file given a certain path.
FileInputStream	For loading Json files.
FileOutputStream	For saving Json files.
InputStream	For loading data from a Json file.
IOException	For catching errors from InputStream and Outputstream. In case saving or loading a file is disrupted.
OutputStream	For saving data to a Json file.
PrintWriter	For converting an object to a text-output stream.
StringWriter	For writing information in a string format.

Table 2.3: Uses the classes in the Java API's java.io

Class/Interface	Use
Cursor	For setting the action of the mouse.
Node	For the purpose of generics. So all shapes can be called.
Scene	For the control over the applications window.
SnapshotParameters	For taking a screenshot of desired parameters in the canvas. Used for exporting image file.

Table 2.4: Uses the classes in the Java API's javafx.scene

Class/Interface	**Use**
Ellipse	For the stations in the application.
Rectangle	For the images of each station.
Line	For the metro lines in the application.
Shape	For the ease of changing appearances of ellipse, rectangle and line.

Table 2.5: Uses the classes in the Java API's javafx.scene.shape

Class/Interface	**Use**
Font	For changing the font of a selected text.
FontPosture	For declaring if the selected text is italics or not.
FontWeight	For declaring if the selected text is bold or not.
Text	For editing the labels of the metro lines and stations.

Table 2.6: Uses the classes in the Java API's javafx.scene.text

Class/Interface	**Use**
ObservableList	For storing objects (shapes).

Table 2.7: Uses the classes in the Java API's javafx.collections

Class/Interface	**Use**
JsonGenerator	For generating/writing the format of the Json file.

Table 2.8: Uses the classes in the Java API's javafx.json.stream

Class/Interface	**Use**
KeyEvent	For getting information about a key event, like which key was pressed.
MouseEvent	For getting information about a mouse event, like where was the mouse pressed?

Table 2.9: Uses the classes in the Java API's javafx.scene.input

Class/Interface	Use
ActionEvent	For getting information about an action event like which button was pressed.
EventHandler	For taking action when a certain action occurs.

Table 2.10: Uses the classes in the Java API's javafx.event

Class/Interface	Use
FileChooser	For choosing a file to either load, save or choose an image.

Table 2.11: Uses the classes in the Java API's javafx.stage

Class/Interface	Use
ArrayList	For storing objects in this case, the shapes (ellipse, line, and text).
HashMap	For storing (name,value) key pairs, we'll use it for storing our Images, accessible using their ID names.
Locale	For setting the language of the window and making sure all API libraries used corresponding to the location chosen.
Map	For mapping keys to values in a hashMap.

Table 2.12: Uses the classes in the Java API's java.util

Class/Interface	Use
Image	For loading images from a specified URL.
ImageView	For rendering the image so that it fits within its boundaries.
WritableImage	For writing export image.

Table 2.13: Uses the classes in the Java API's javafx.scene.image

Class/Interface	Use
Color	For filling a shape with a specified color, chosen from the colorPicker.
ImagePattern	For filling a shape with a specified image.
Paint	For filling a shape with specified color or gradients.

Table 2.14: Uses the classes in the Java API's javafx.scene.paint

Class/Interface	Use
BufferedImage	For storing image data where pixel information can be accessed and changed.

Table 2.15: Uses the classes in the Java API's java.awt.image

Class/Interface	Use
Json	Factory class for creating Json objects.
JsonArray	For storing Json objects.
JsonArrayBuilder	For the aid to help build the JsonArray.
JsonNumber	For storing numbers in the Json file.
JsonValue	For storing some value to the Json file. Such as JsonObject, JsonArray, JsonNumber, JsonString, JsonVAlue.TRUE, JsonValue.FALSE and JsonValue.NULL.
JsonReader	For reading and loading Json files.
JsonWriter	For writing and saving Json files.
JsonWriterFactory	Factory to create JsonWriter instances.
JsonObject	For loading the workspace with given objects.

Table 2.16: Uses the classes in the Java API's javax.json

Class/Interface	Use
BlurType	For soften a shadow effect.
DropShadow	For the purpose of highlighting a node which is selected.
Effect	For changing the feature of a certain node.

Table 2.17: Uses the classes in the Java API's javafx.scene.effect

Class/Interface	Use
DoubleProperty	For binding Nodes together. (MetroLine and MetroStation)

Table 2.18: Uses the classes in the Java API's javafx.beans.property

Class/Interface	Use
FileUtils	For making folders. (exporting)

Table 2.19: Uses the classes in the Java API's org.apache.commons.io

3 Class-Level Design Viewpoint

As aforementioned, this design will encompass the Metro Map Maker application. 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 to detailed ones.

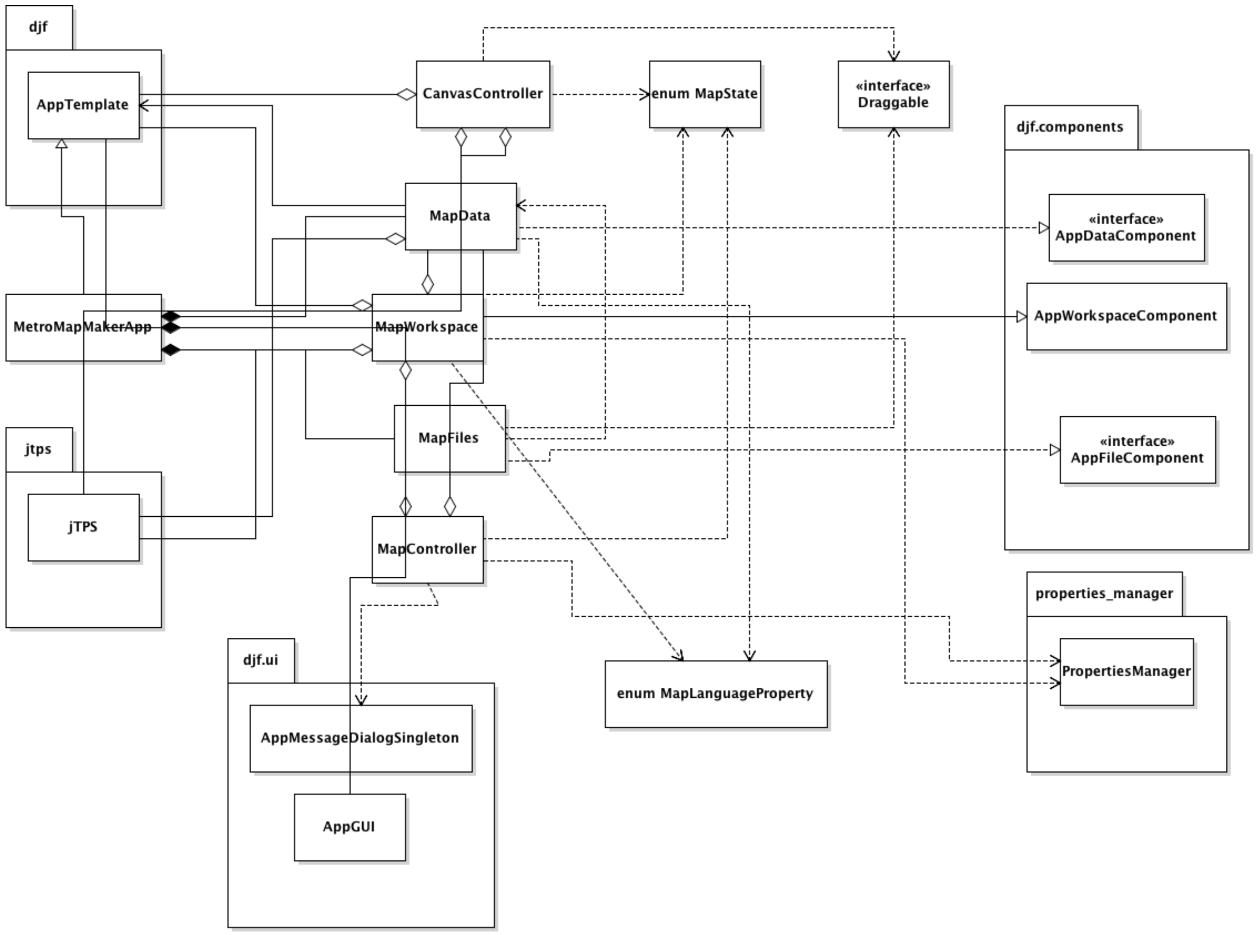


Figure 3.1: Metro Map Maker Overview UML Class Diagram

Note: This is how all the classes interact with each other. For more details, please go to the UML Class diagram for that class.

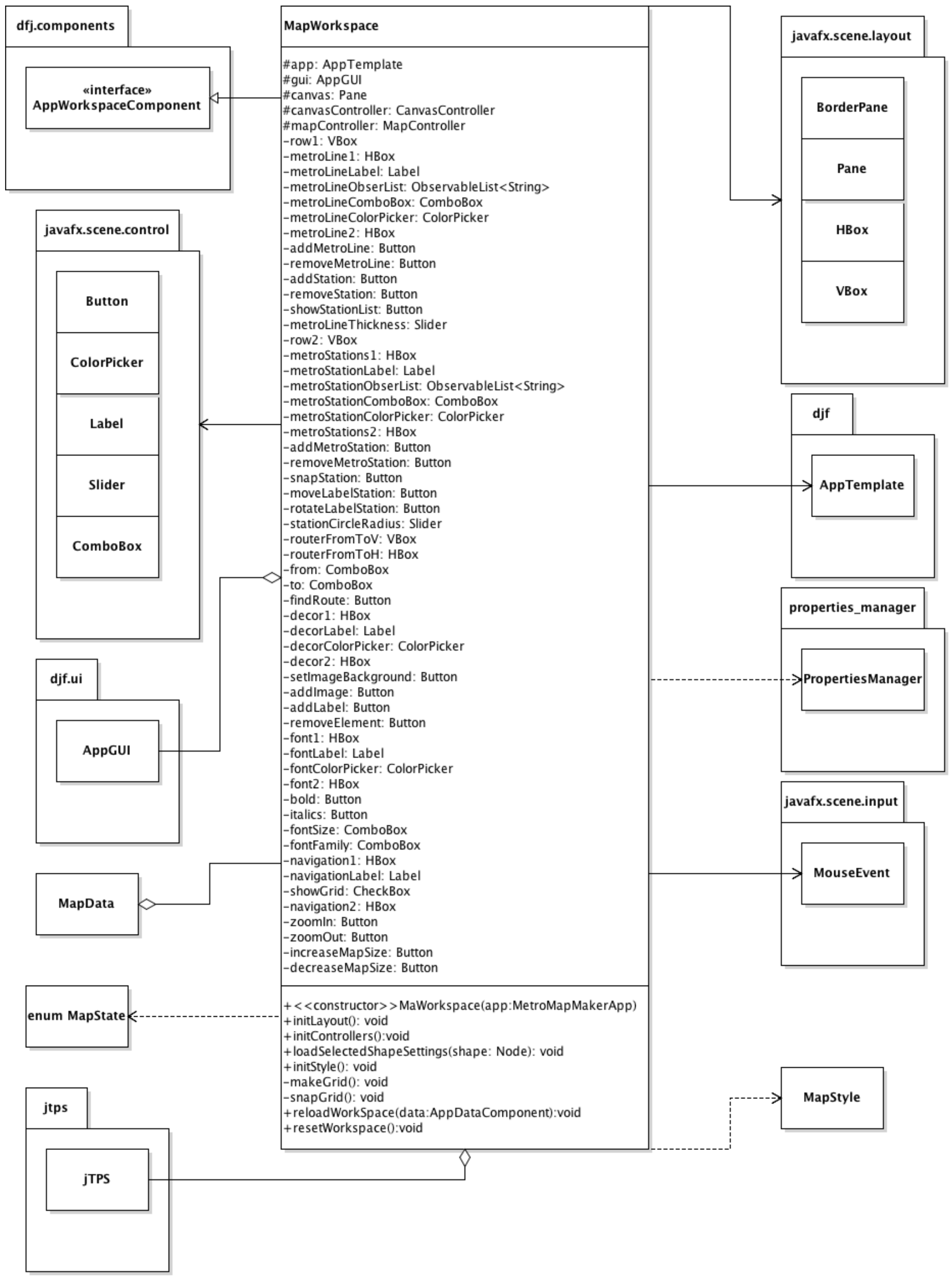


Figure 3.2: Detailed MapWorkspace UML Class Diagrams

Note: Workspace class organizes the whole canvas and left pane of the workspace.

This class is used to design the layout of the program. Where the buttons are located, where the canvas will be, the shape and color of each button is all located here. In additional, it also handles actions to the canvas and the buttons in the left toolbar. This class also deals with showing the grid pane and snapping each node to the grid.

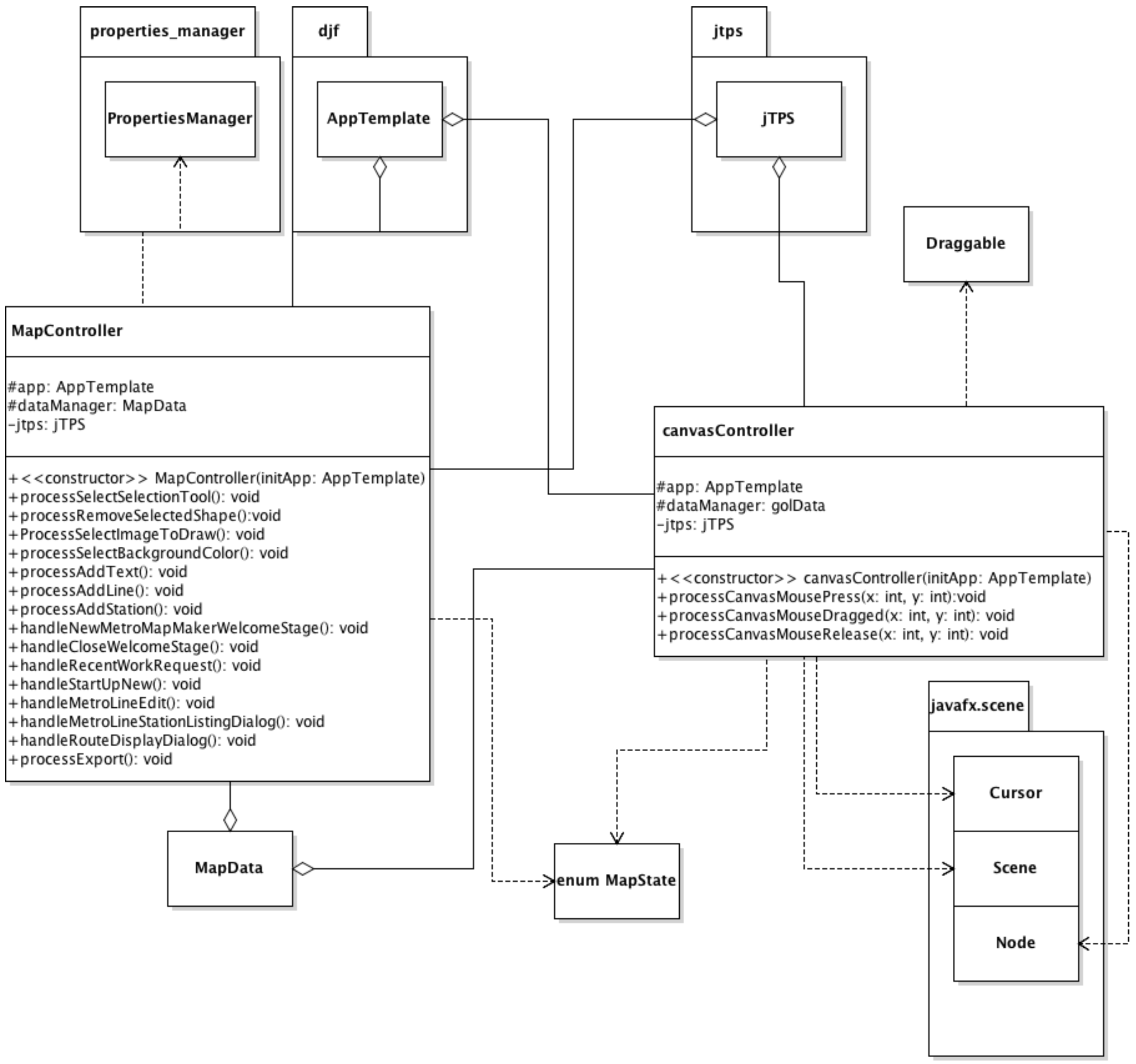

properties_manager
PropertiesManager
djf
AppTemplate
jtps
jTPS
Draggable
MapController
#app: AppTemplate
#dataManager: MapData
-jtps: jTPS
+<<constructor>> MapController(initApp: AppTemplate)
+processSelectSelectionTool(): void
+processRemoveSelectedShape():void
+ProcessSelectImageToDraw(): void
+processSelectBackgroundColor(): void
+processAddText(): void
+processAddLine(): void
+processAddStation(): void
+handleNewMetroMapMakerWelcomeStage(): void
+handleCloseWelcomeStage(): void
+handleRecentWorkRequest(): void
+handleStartUpNew(): void
+handleMetroLineEdit(): void
+handleMetroLineStationListingDialog(): void
+handleRouteDisplayDialog(): void
+processExport(): void
canvasController
#app: AppTemplate
#dataManager: golData
-jtps: jTPS
+<<constructor>> canvasController(initApp: AppTemplate)
+processCanvasMousePress(x: int, y: int):void
+processCanvasMouseDragged(x: int, y: int): void
+processCanvasMouseRelease(x: int, y: int): void
javafx.scene
Cursor
Scene
Node
MapData
enum MapState

Figure 3.3: Detailed MapController and CanvasController UML Class Diagrams

Note: Distributes the command given by the Workspace class such that what needs to be changed will be changed accordingly.

These classes handle the effects to when a user does a certain action to a node. When the user clicks on a node on the canvas, the node is selected and highlighted. This is dealt in the Canvas controller class. This class also deals with dragging the shape and sizing the image. MapController deals with the buttons located in the left tool bar. Using these classes will make the program more comprehensible.

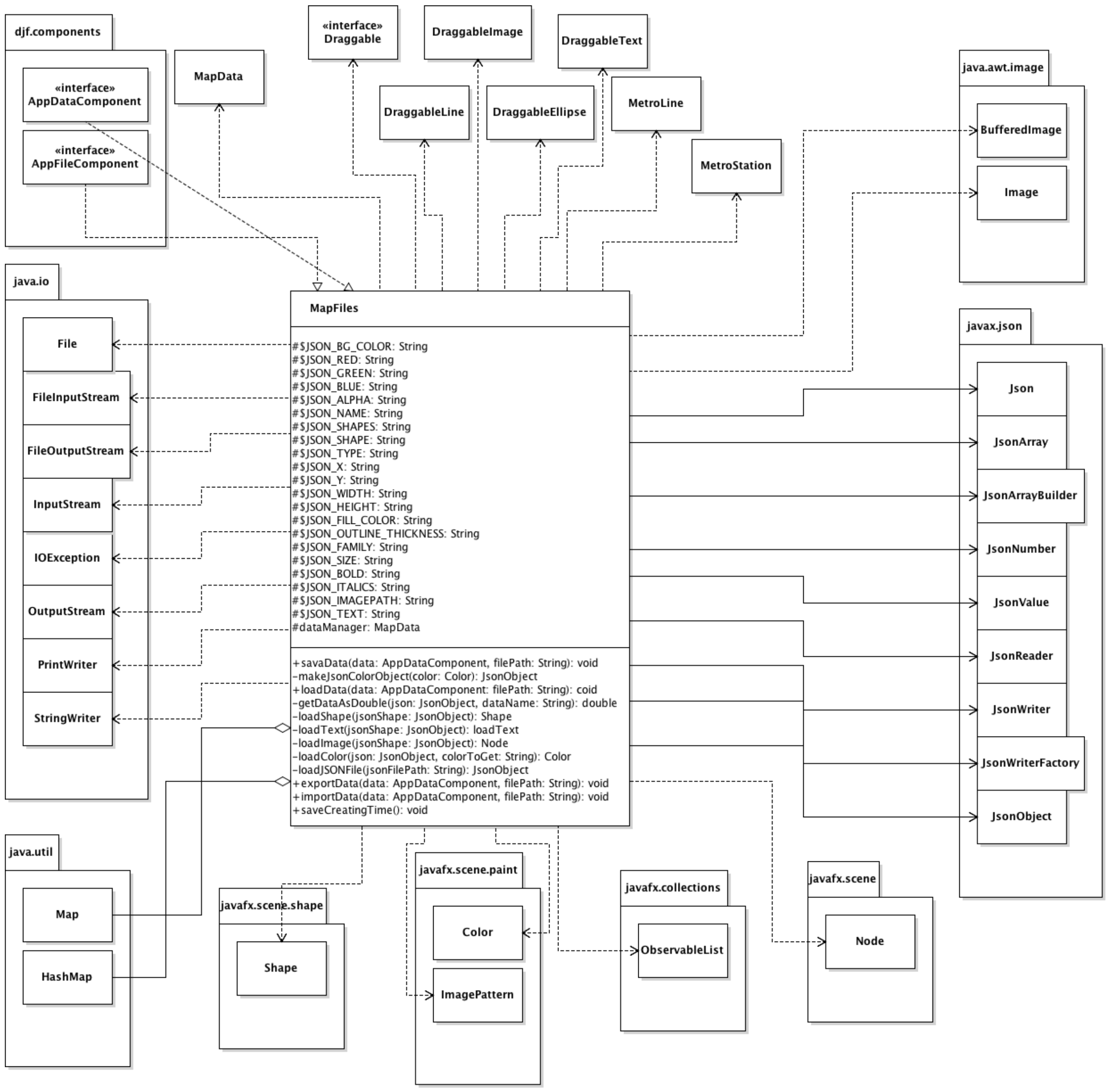


Figure 3.4: Detailed MapFiles UML Class Diagram

Note: Saves and loads files.

This is where all the files are saved and loaded. Image has its own class while loading because it has to consider different attributes such as the path it needs to load the image. Similarly, text has a string value which the other nodes do not have. Since ellipse and line are both just shapes with similar value, when loading these, we just use load shape. When saving a file, we need to take into account the recent time made so that when the program reopens, the user can view the most recent files.

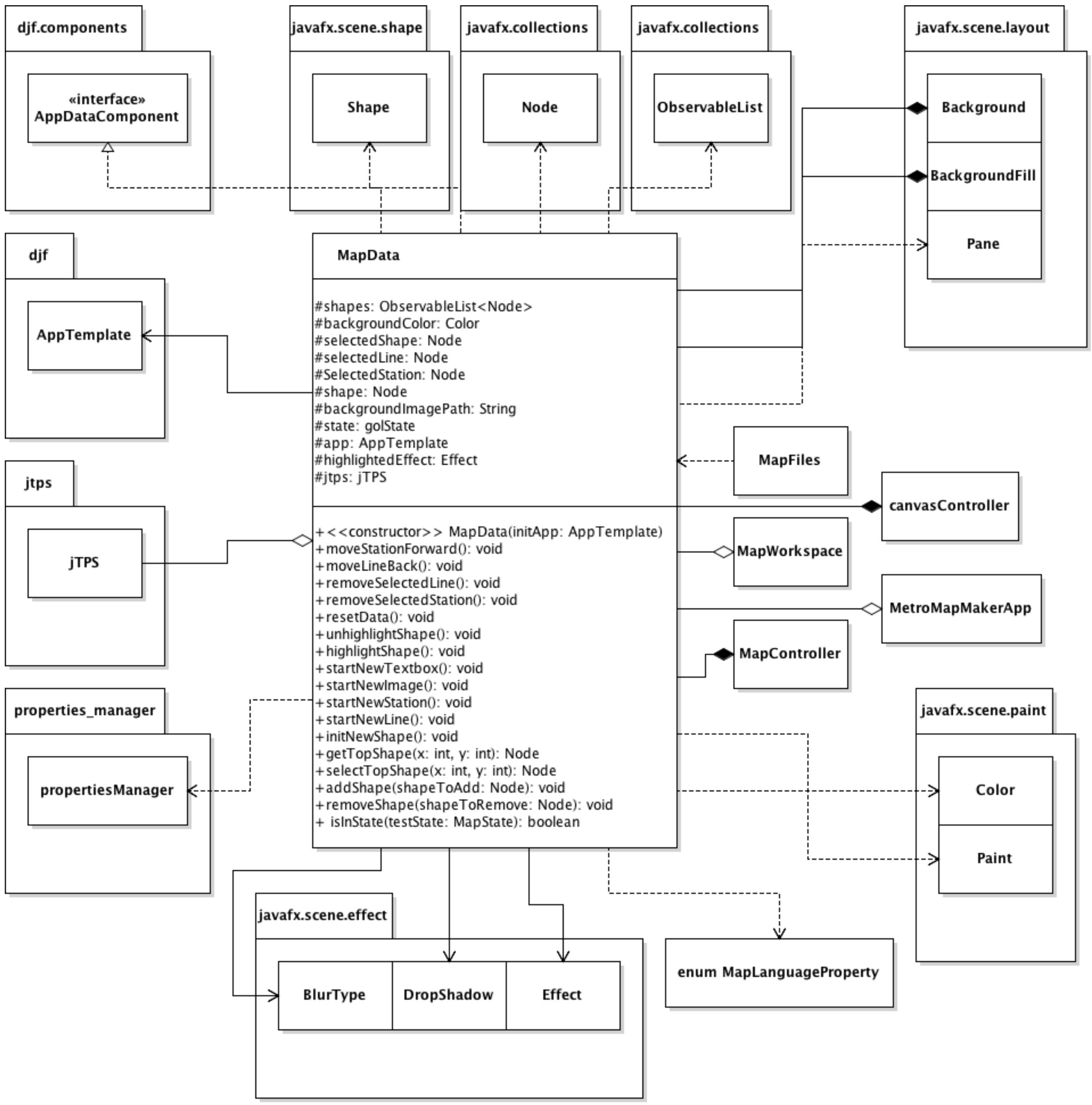


Figure 3.5: Detailed MapData UML Class Diagram

Note: If an object needs to be modified in anyway, MapController will call this class which will process the command.

This class deals with the nodes the user clicks on. Whether it is to change the color, highlight, highlight, remove, add a node, it is all dealt with here.

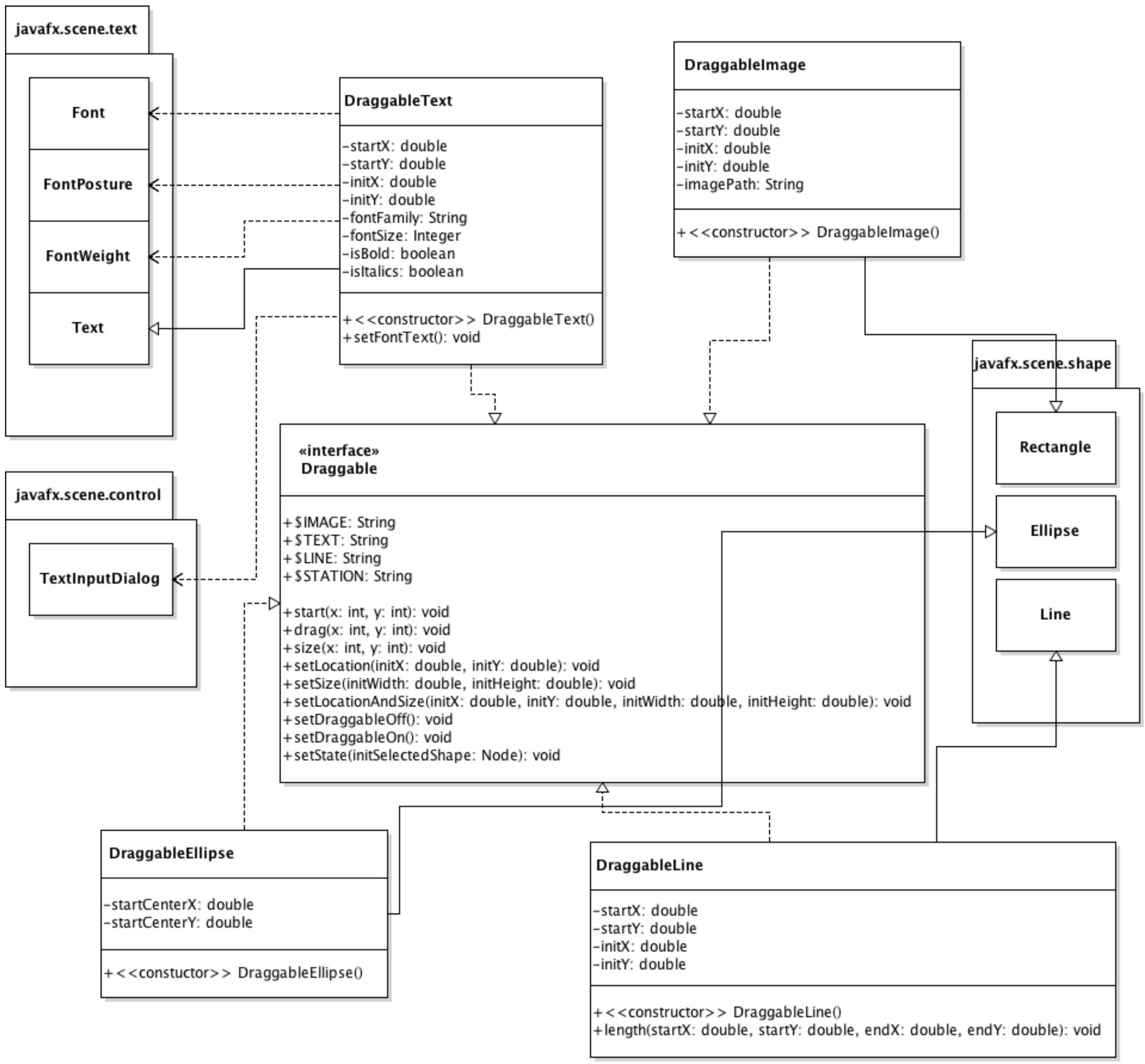


Figure 3.6: Detailed Draggable, DraggableText, DraggableImage, DraggableLine, and DraggableEllipse UML Class Diagram

Note: These classes are all the nodes which we will be dealing with. They are all draggable and sizable. DraggableImage is used for when the user wants to add an image to the canvas. DraggableText is for the label on the cavas and the name for the station and line. DraggableLine and DraggableEllipse is used for the line and station. It will bind with the DraggableText. This is a factory pattern because you don't need to know what type to use.

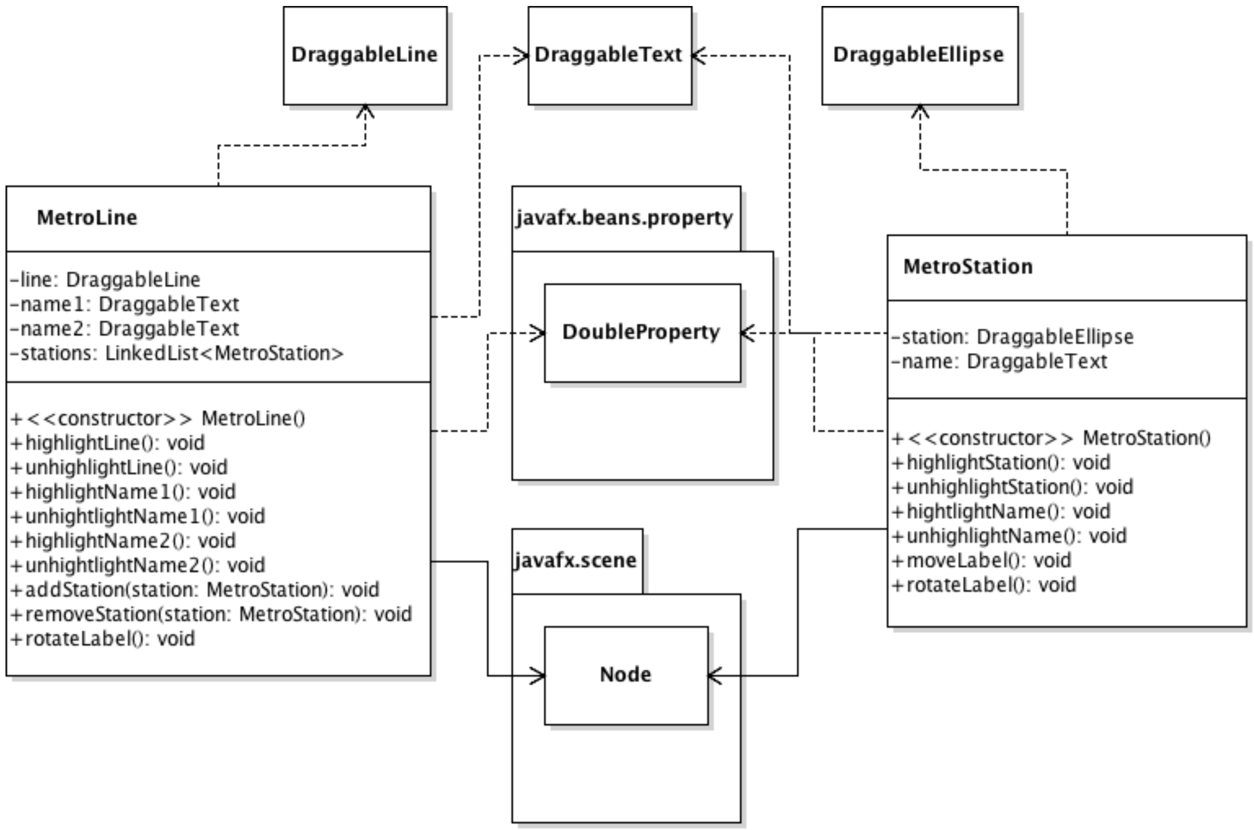


Figure 3.6a: Detailed MetroLine and MetroStation UML Class Diagram

Note: These two classes deal with the line and station. MetroLine consists of 2 DraggableText and a DraggableLine which will bind. MetroStation consists of 1 DraggableEllipse and 1 DraggableText. Since when clicking on the label of that line or station, another node which it is bind to is not highlighted, we need individual methods to highlight each node.

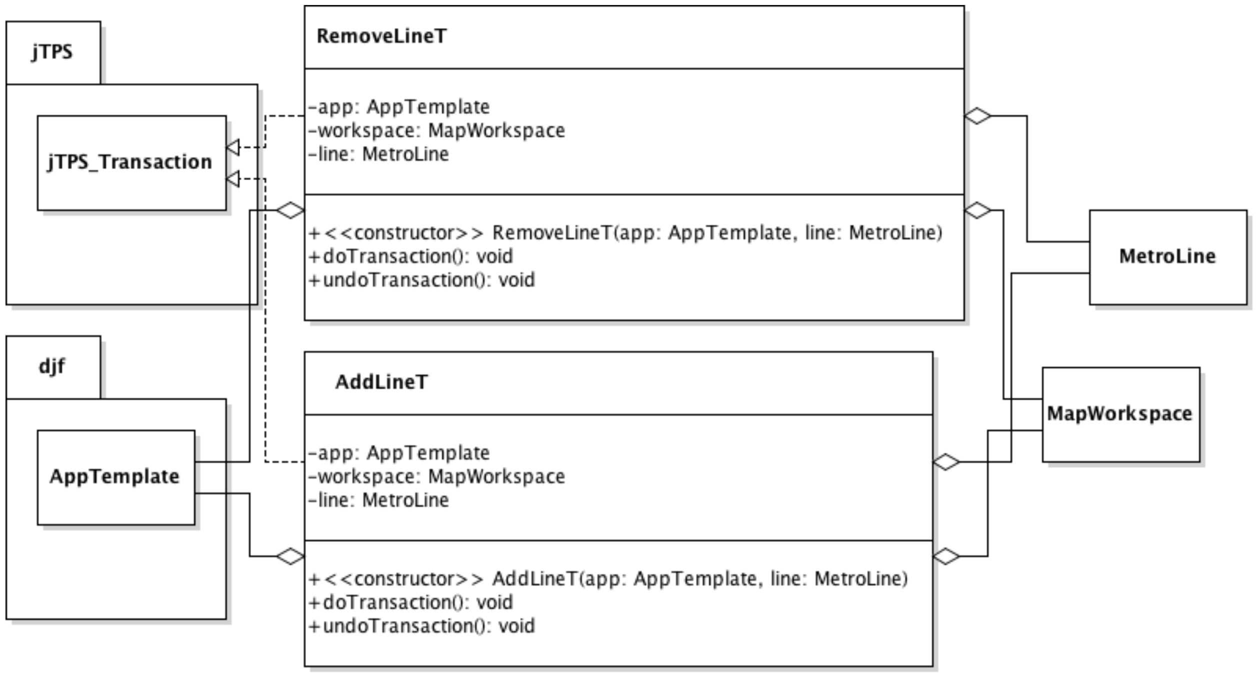


Figure 3.7: Detailed AddLineT and RemoveLineT UML Class Diagram

Note: RemoveLineT and AddLineT are the transactions to removing a line and adding a line. It will take the line and either remove or add it.

All classes that use the jTPS framework uses momento design pattern. Using this will be most efficient because it remembers all the attributes of an object to restore its original state.

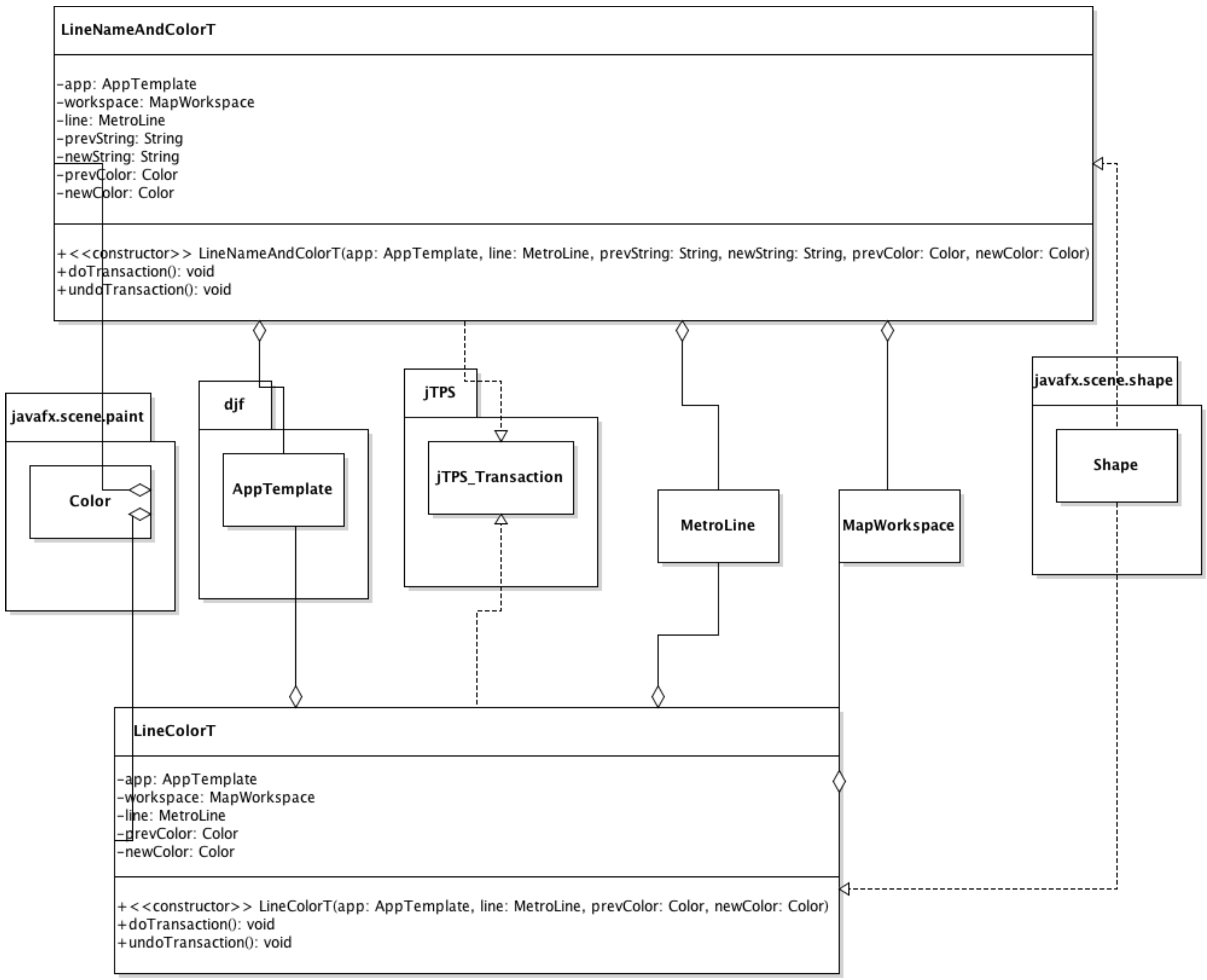


Figure 3.8: Detailed LineNameAndColorT and LineColorT UML Class Diagram

Note: LineNameAndColorT will change the line name and color. Since both are changed when the user clicks "ok" on the dialog, we need to change them and undo them together. We will need the previous attributes and the ones which we want to change. LineColorT changes the line color. By changing the color of the line, the DraggableLine and DraggableText will both change to the selected color from the colorpicker.

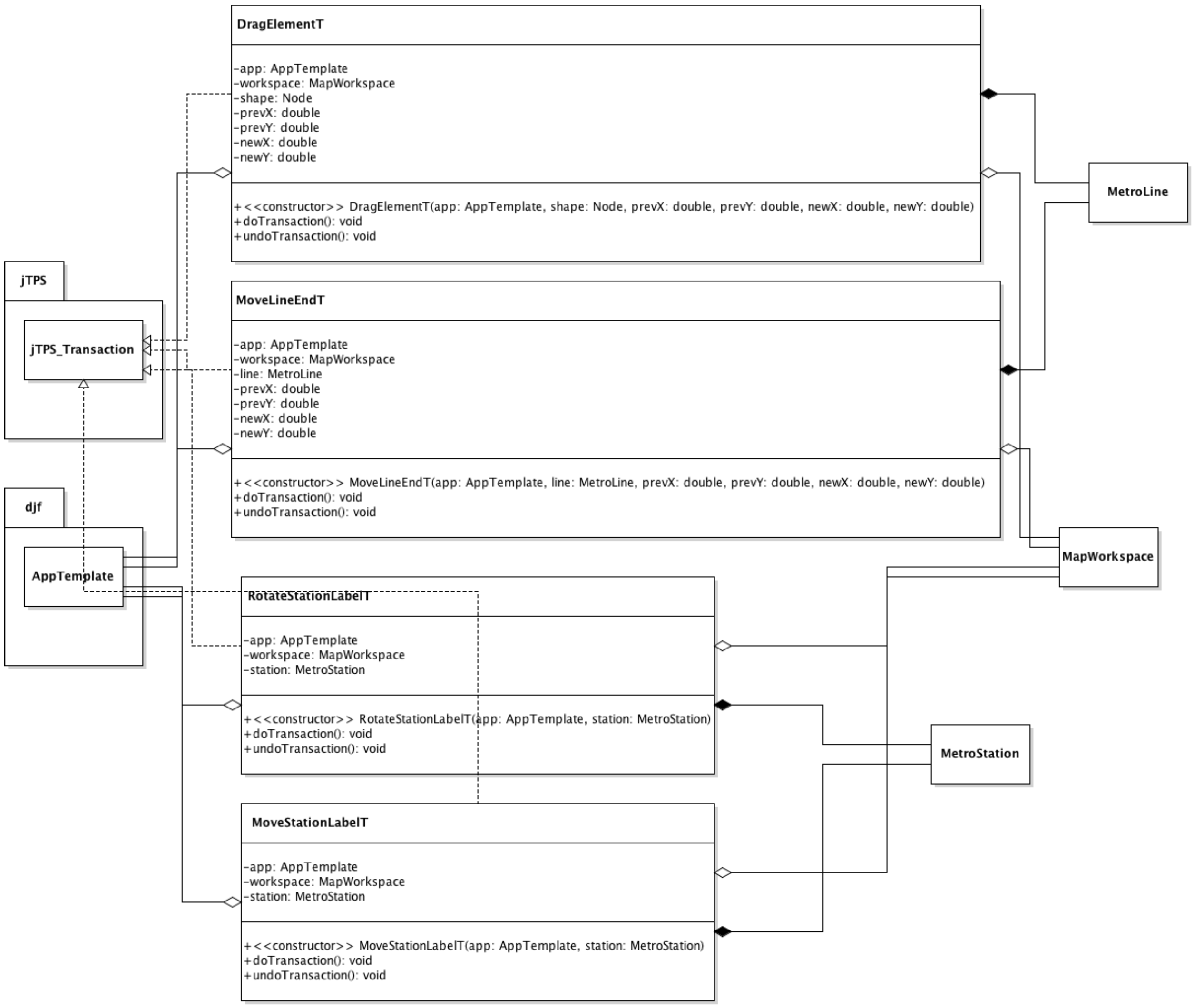


Figure 3.9: Detailed DragElementT, MoveLineEndT, RotateStationLabelT, and MoveStationLabelT UML Class Diagram

Note: DragElementT drags all nodes which is draggable except line. The reason for this is because when dragging a line, one end cannot more while the other does. The other nodes must all move as one unlike the line. This, we have the MoveineEndT which only moves or rotates the line as the user drags the node. RotateStateLabelT will rotate the station label/name. MoveStationLabelT will move the station label/name to one of the designated 4 locations.

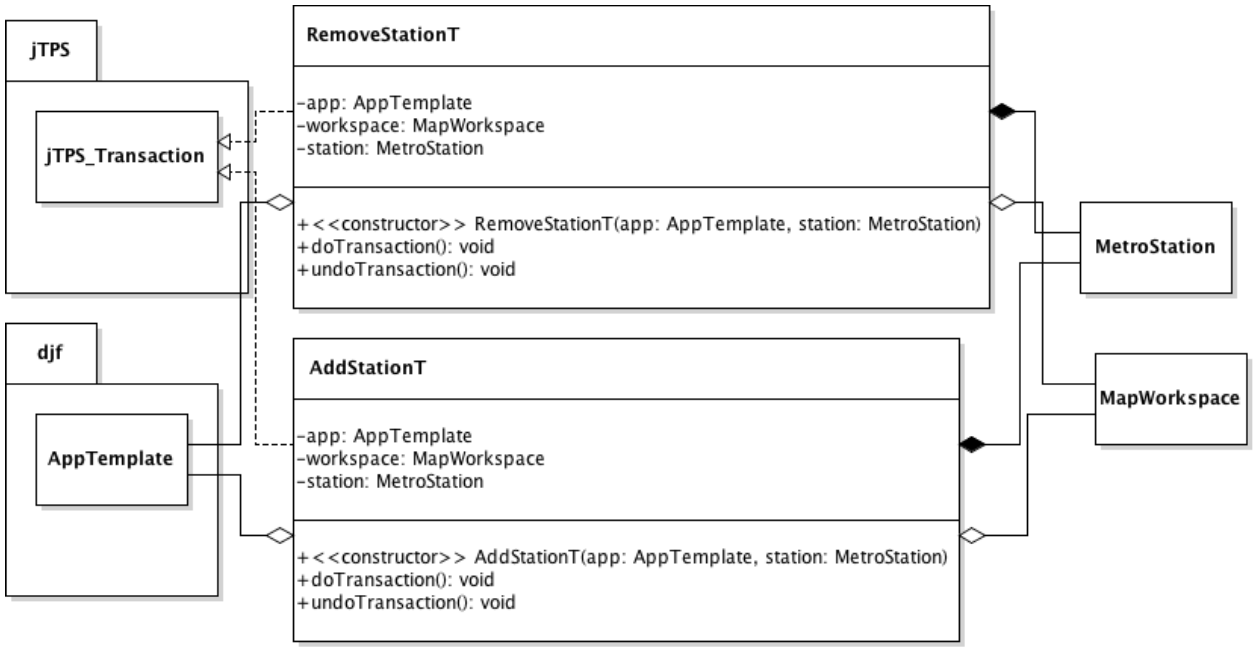


Figure 3.10: Detailed AddStationT and RemoveStationT UML Class Diagram

Note: RemoveStationT will remove a station and AddStationT will add the station. This can be undone so it must be part of a transaction class.

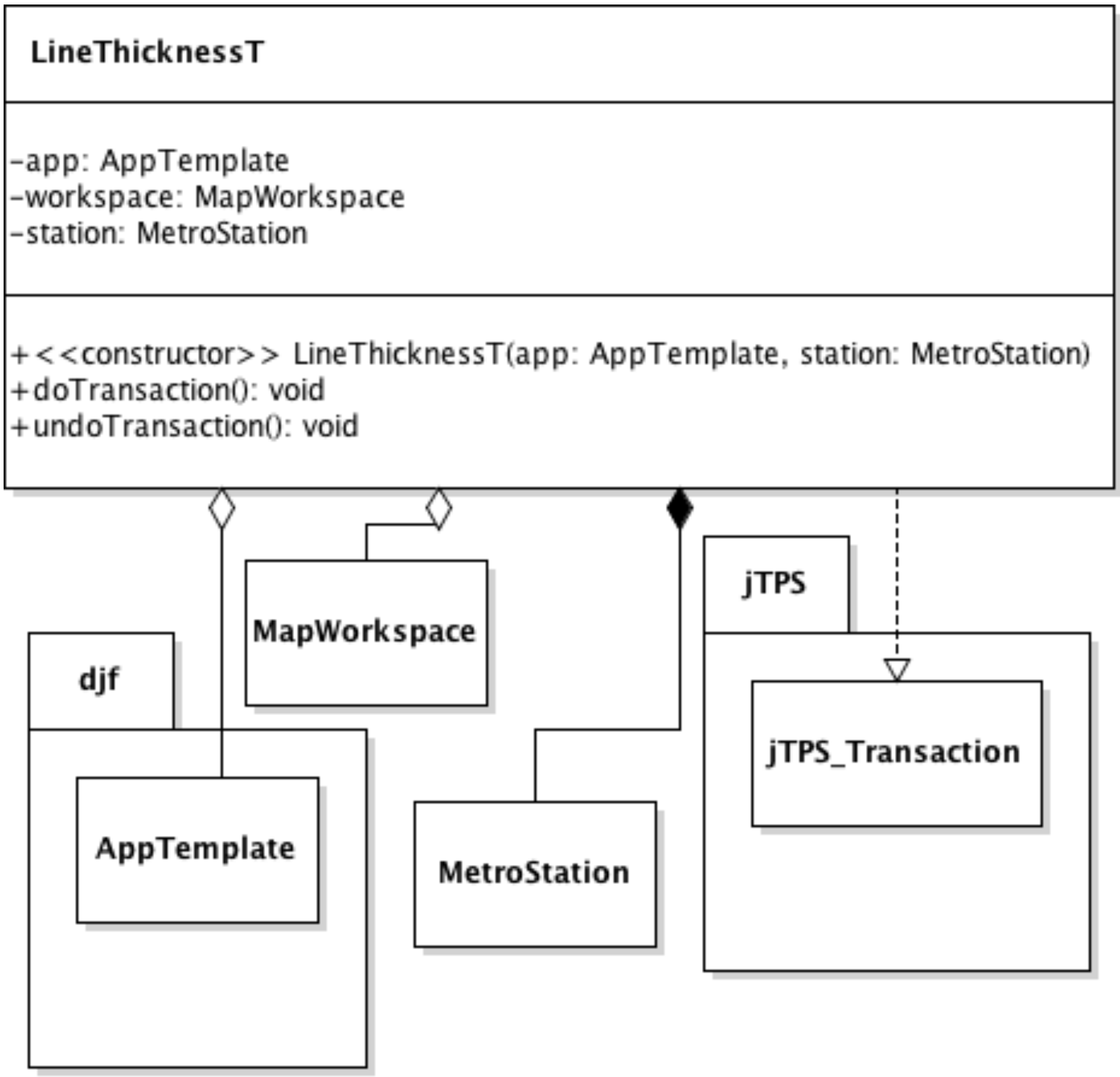


Figure 3.11: Detailed LineThicknessT UML Class diagram

Note: This changes the thickness of the line. The user accesses this when they use the slider located in the left toolbar. This transaction can be done and undone.

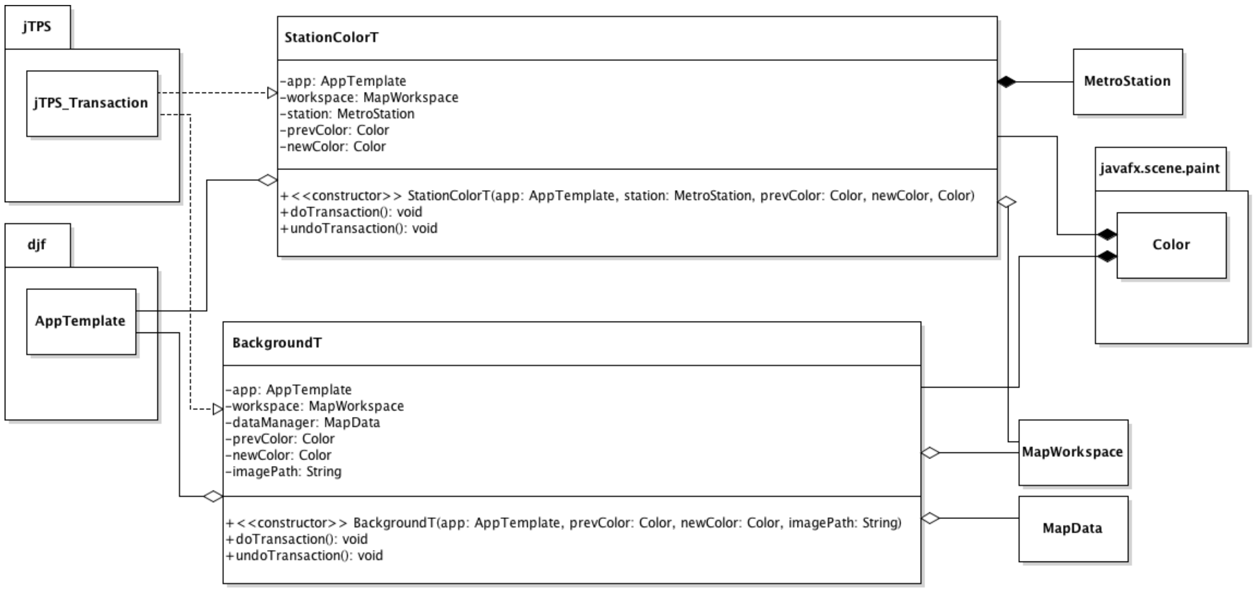


Figure 3.12: Detailed StationColorT and BackgroundT UML Class diagram

Note: StationColorT changes the color of the DraggableEllipse and the DraggableText it is associated with. BackgroundT changes the color of the background.

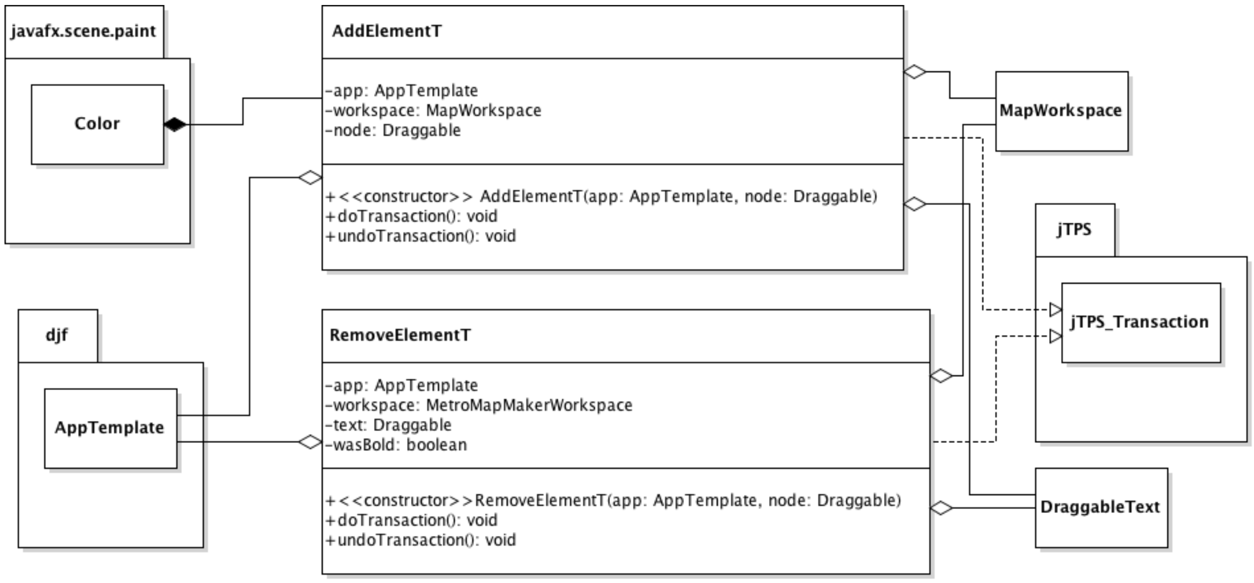


Figure 3.13: Detailed AddElementT and RemoveElementT UML Class diagram

Note: AddElementT and RemoveElementT adds and removes images and labels. It cannot add and remove lines nor stations.

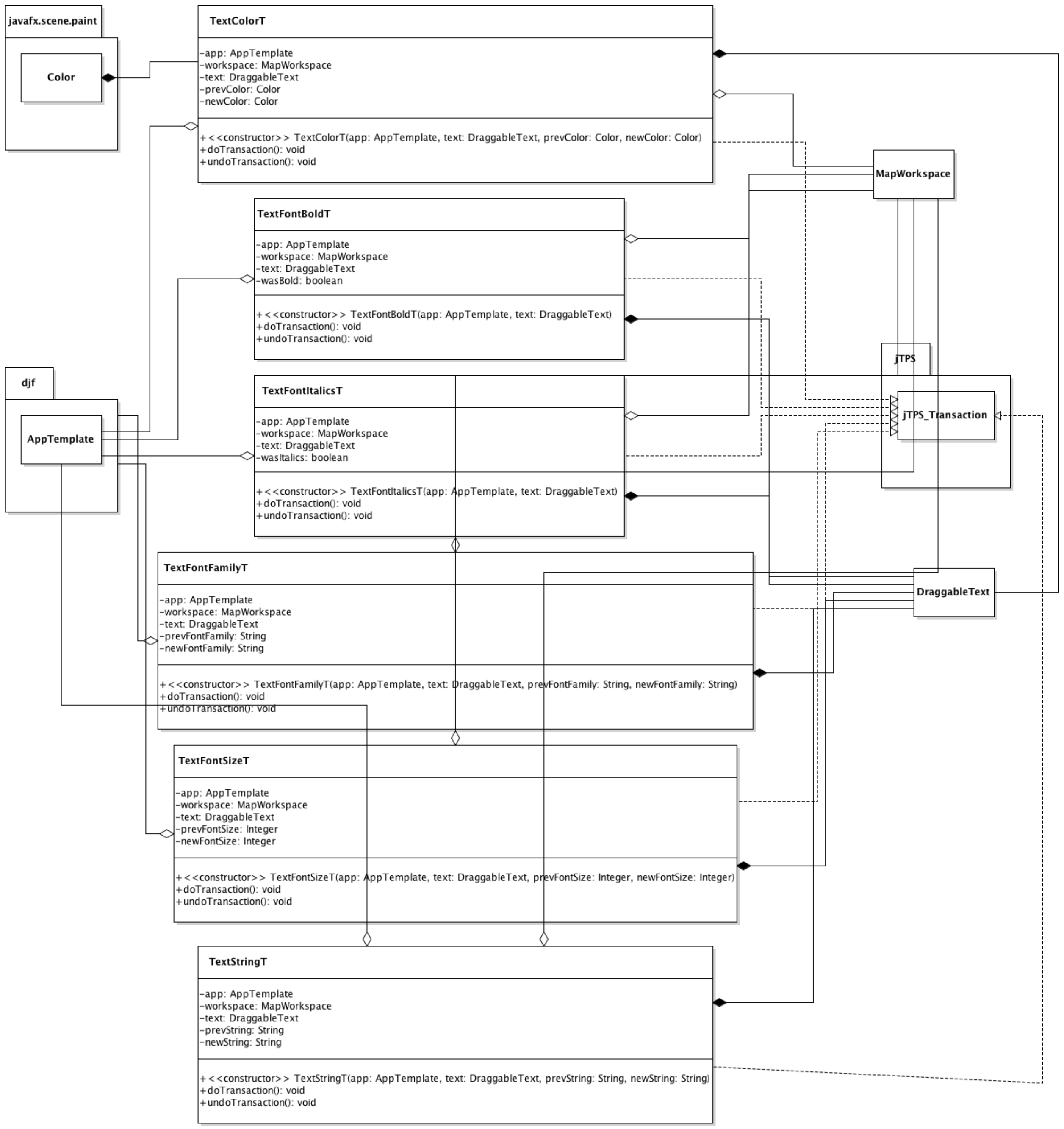
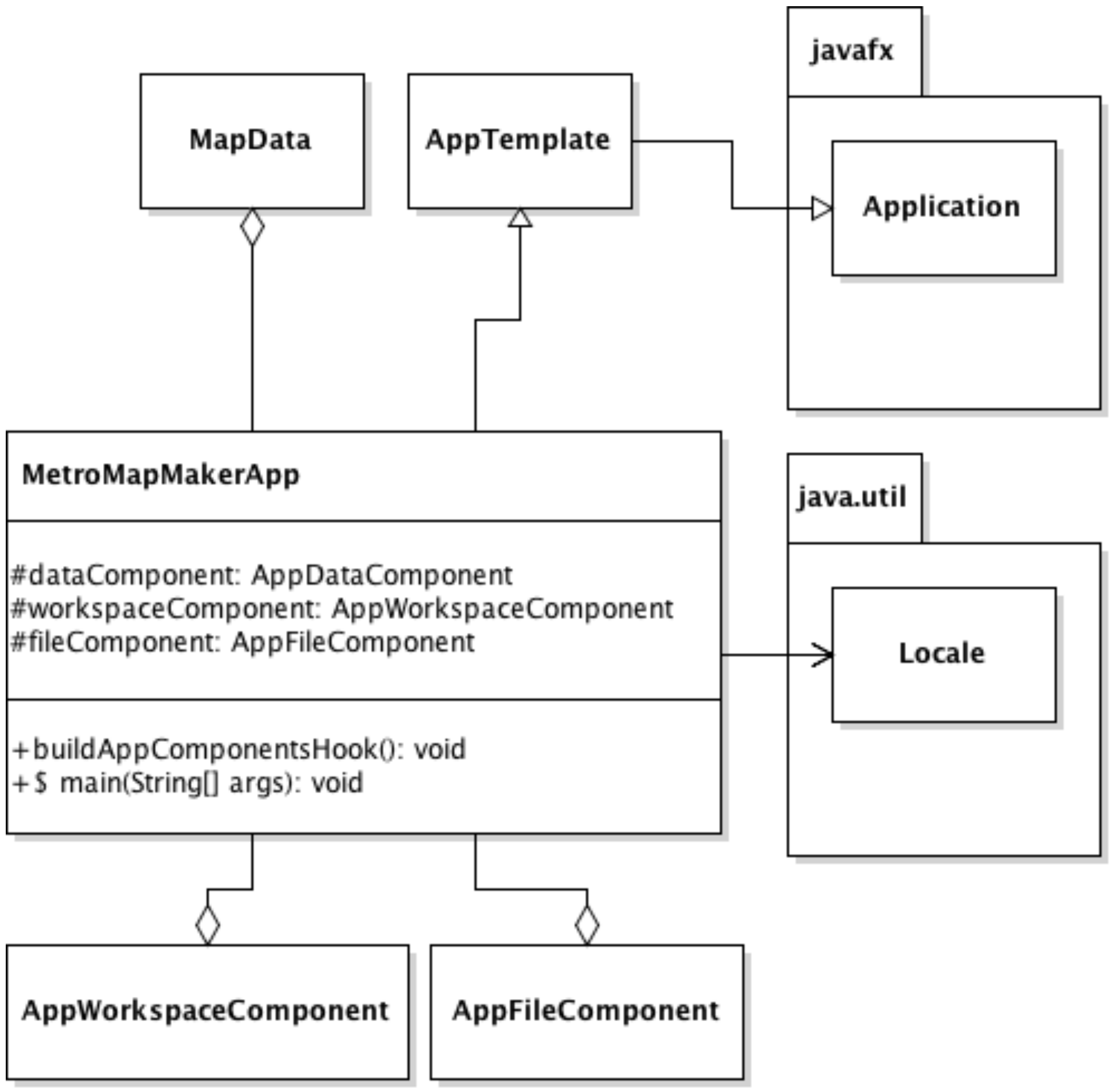


Figure 3.14: Detailed TextColorT, TextFontFamilyT, ChangeTextFontSizeTransaction. TextFontBoldT, TextFontItalicsT, and TextStringT UML Class diagram

Note: These classes change the font values. These values include color, bold, italics, family, size, and the string value it has.



```
#dataComponent: AppDataComponent
#workspaceComponent: AppWorkspaceComponent
#fileComponent: AppFileComponent

+buildAppComponentsHook(): void
+$ main(String[] args): void
```

Figure 3.15: Detailed MetroMapMakerApp UML Class Diagram

Note: This class holds the main class. It gives out the initial orders of when the program runs.

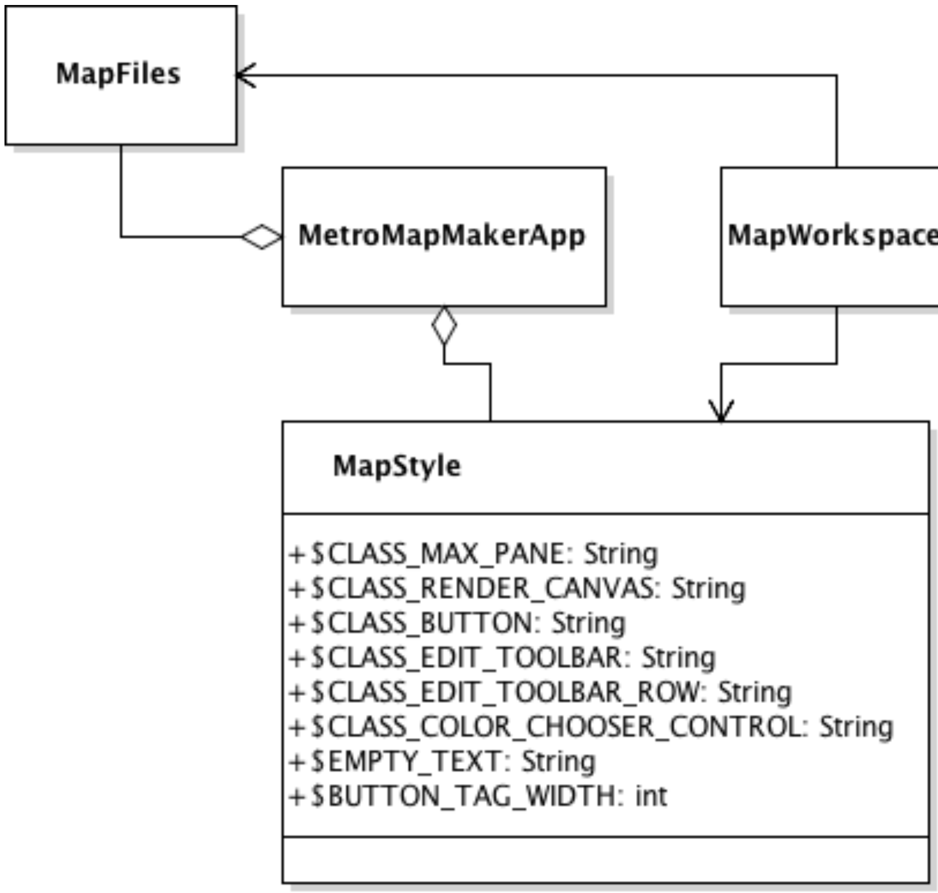


Figure 3.16: Detailed MapStyle UML Class Diagram

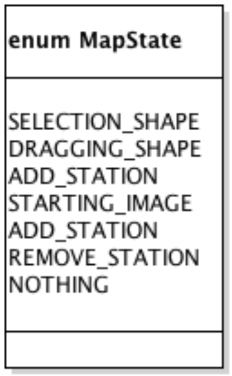


Figure 3.17: Detailed MetroMapMakerState UML Class Diagram

Note: Classes it is associated with is in figure 3.1.
These are the states which the program can be in. Nothing stands for the user clicked on the canvas and at that x and y value, there is no node. Selection_shape is for when the user clicks on a node. Dragging_shape is for when the user is dragging a shape. Add_Station is for adding a station to a line. Starting_image is for adding and sizing an image. Nothing stands for the program is waiting for the user to give it a command.

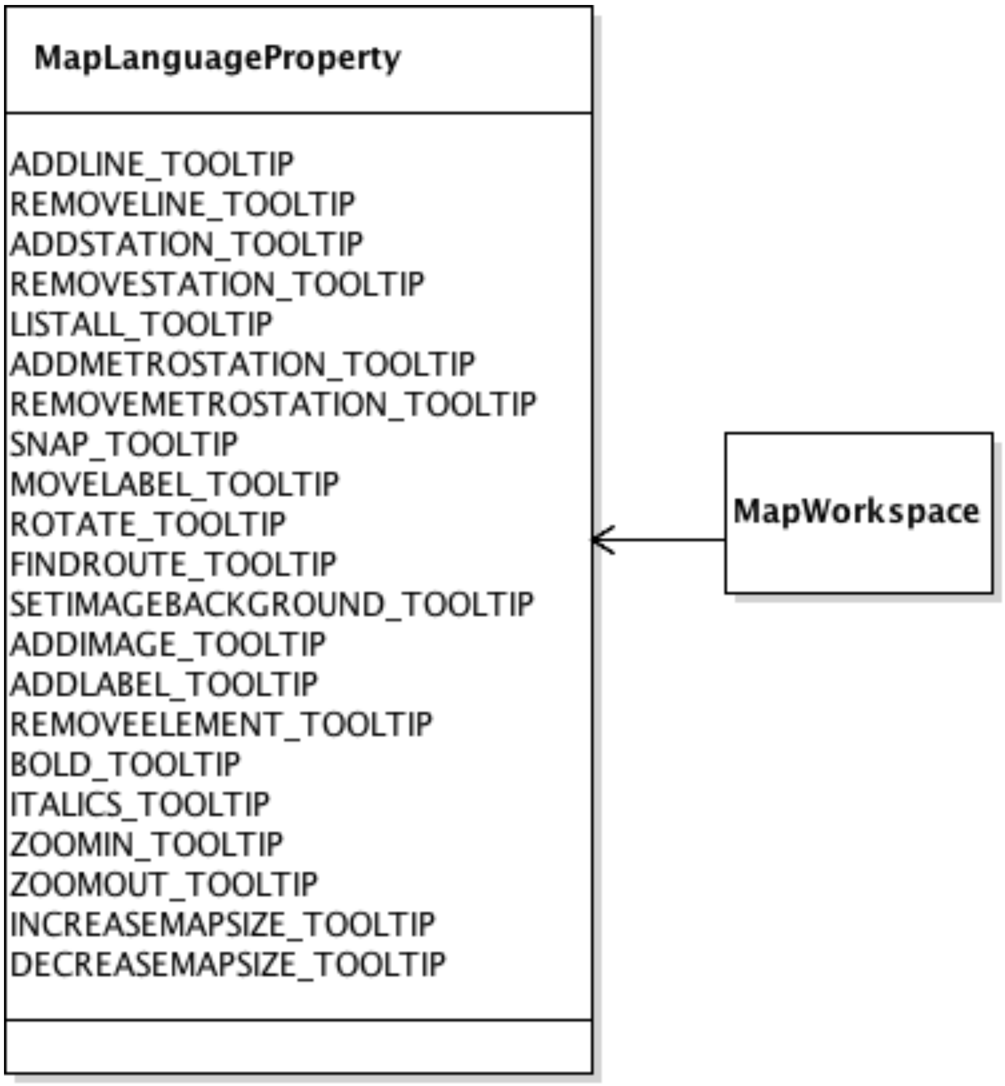


Figure 3.18: Detailed MetroMapMakerLanguageProperty UML Class Diagram

Note: This class is for declaring the tooltips associated with each button on the left toolbar.

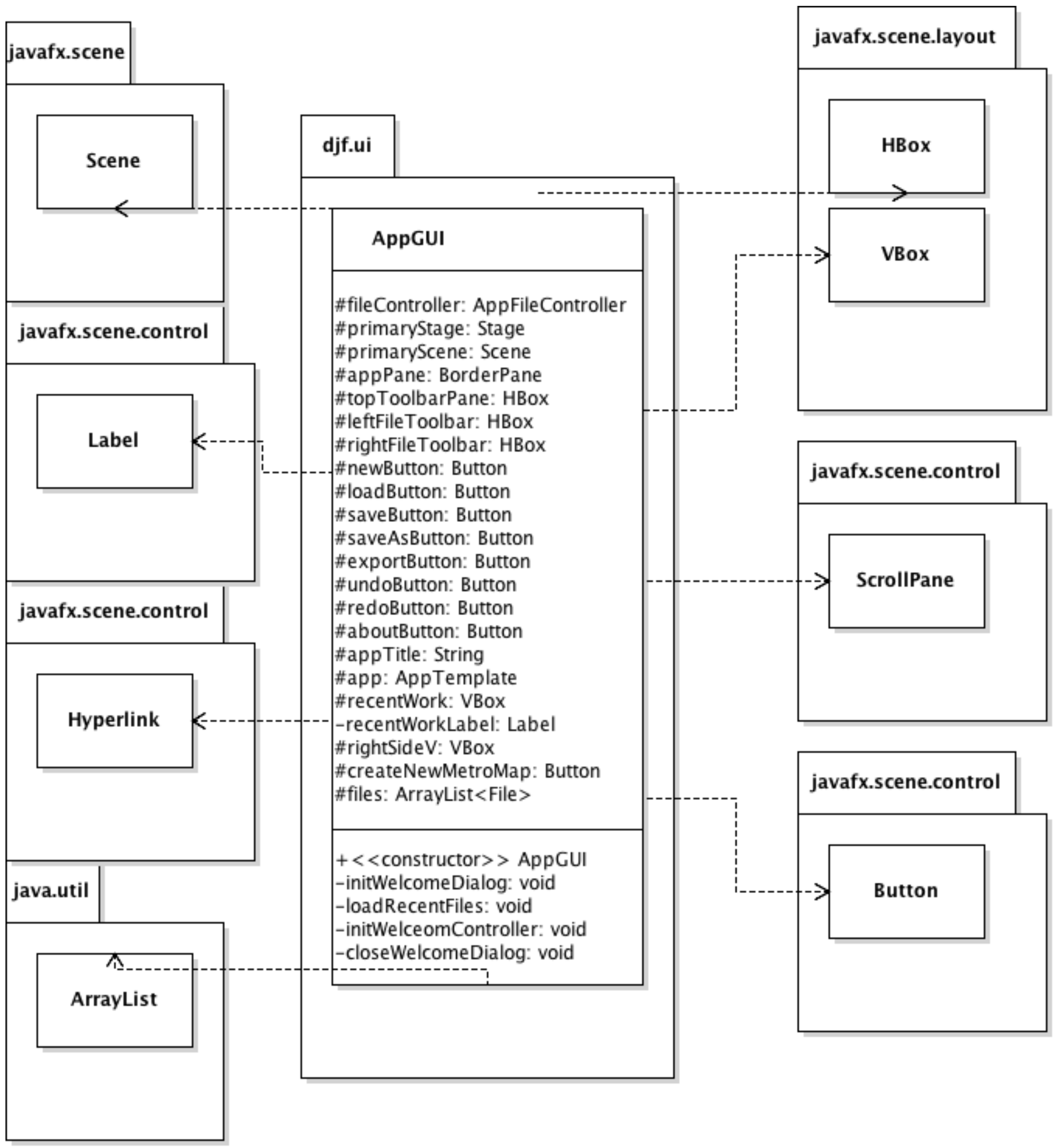


Figure 3.19: Detailed DesktopJavaFramework djf.ui.AppGUI UML Class diagram

Note: This class deals with the welcome dialog and the toolbar for the main UI.

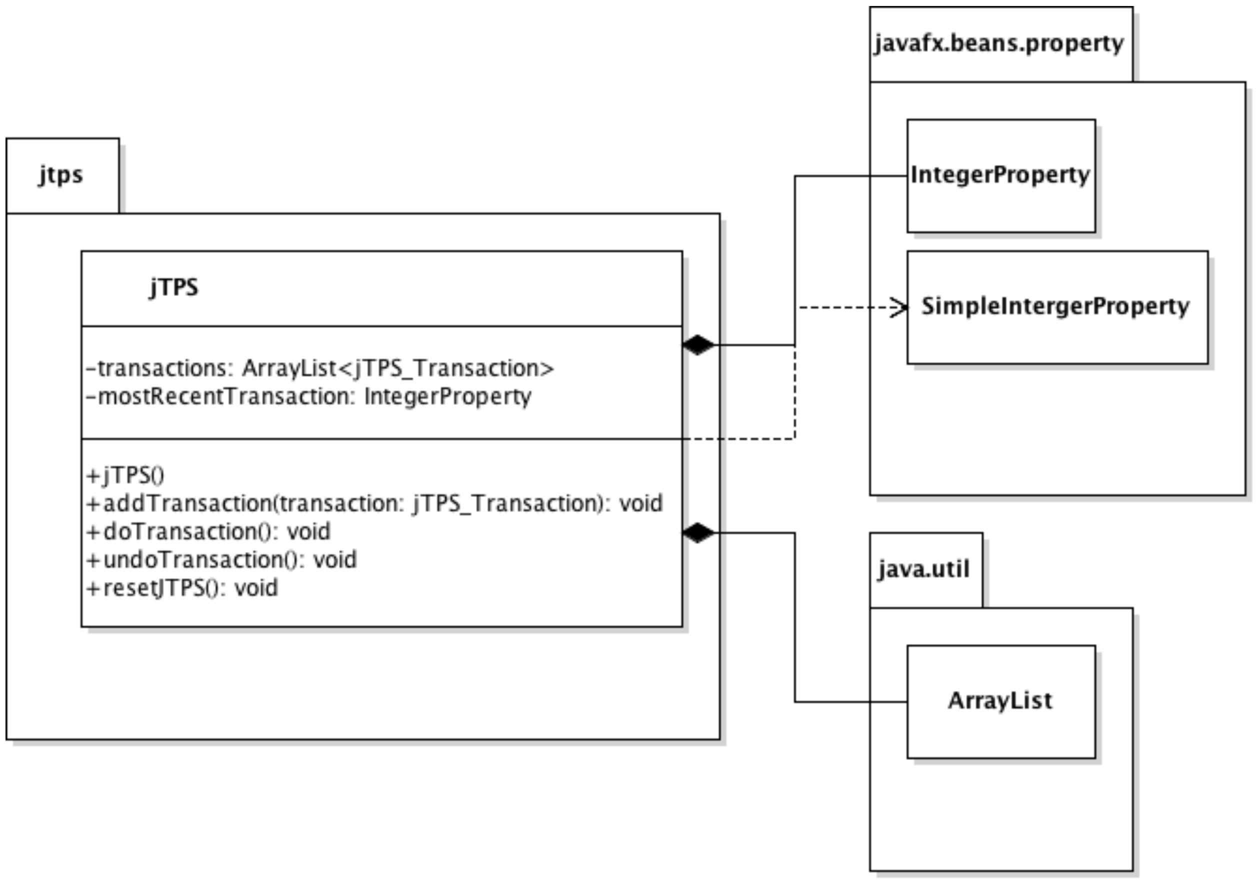


Figure 3.20: Detailed jtps.jTPS UML Class diagram

Note: The jtps class has been modified such that we can access it's information from workspace so that we can disable and enable the undo and redo button accordingly.

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 describes the method called within the code to be developed in order to provide the appropriate vent responses.

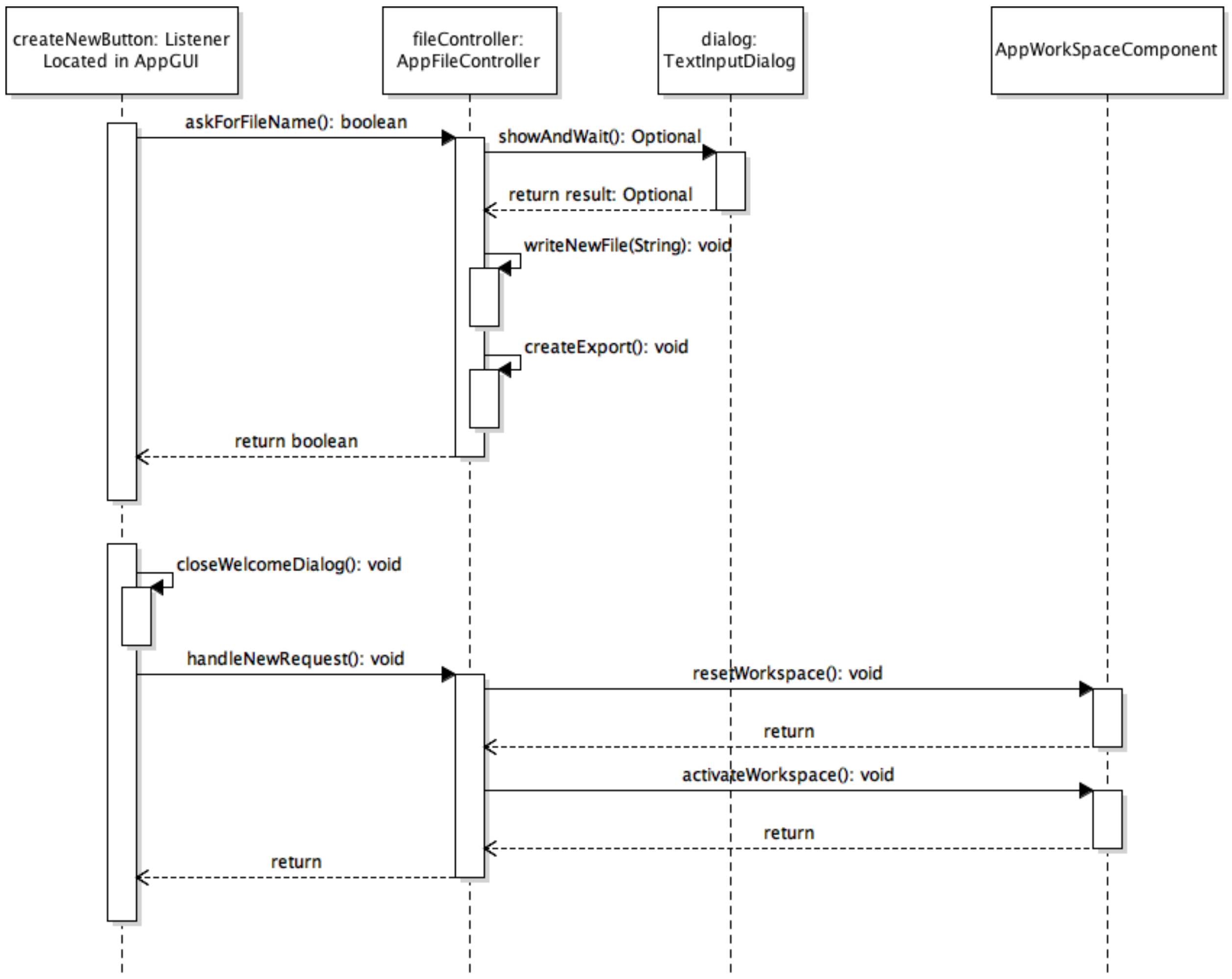


Figure 4.1: Create New Map (Welcome dialog) UML Sequence Diagram (use case 2.1)

Note: There are many ways to do. I chose to put my welcome dialog in my AppGUI class because it is more convenient to switch scenes rather than stages. Another reason is because the main UI has a lot of common controls which the welcome dialog uses. For the way that I did it, it can be easily switched back to the welcome dialog if requested in the future.

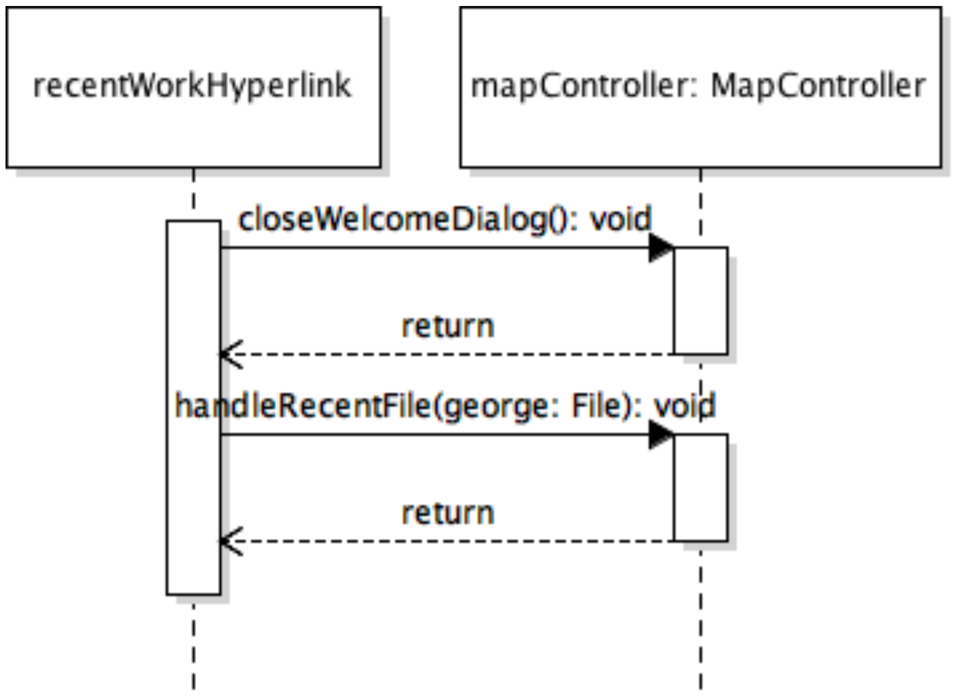

recentWorkHyperlink
mapController: MapController
closeWelcomeDialog(): void
return
handleRecentFile(george: File): void
return

Figure 4.2: Select Recent Map to Load (Welcome dialog) UML Sequence Diagram (use case 2.2)

Note: When the user clicks on a recent hyperlink in the welcome dialog, the welcome stage is closed and the main UI is loaded with the file that user chose. There are many ways of doing this. For example, the user can close the whole stage and restart but I chose to remove all the scene and use the same stage. This would be less complex and will work better with the framework since AppGUI is already called.

When organizing the load recent hyperlinks, I had two options. I could either make a JSON file of the most recent work or I could read from my work folder. I chose the second option since the first one can lead to bugs in the future. By reading from the work folder, the program does do more work by accessing the memory rather than reading but it can make sure that all the files are present. When reading the JSON file, the file could have been deleted by the user which can lead to bugs.

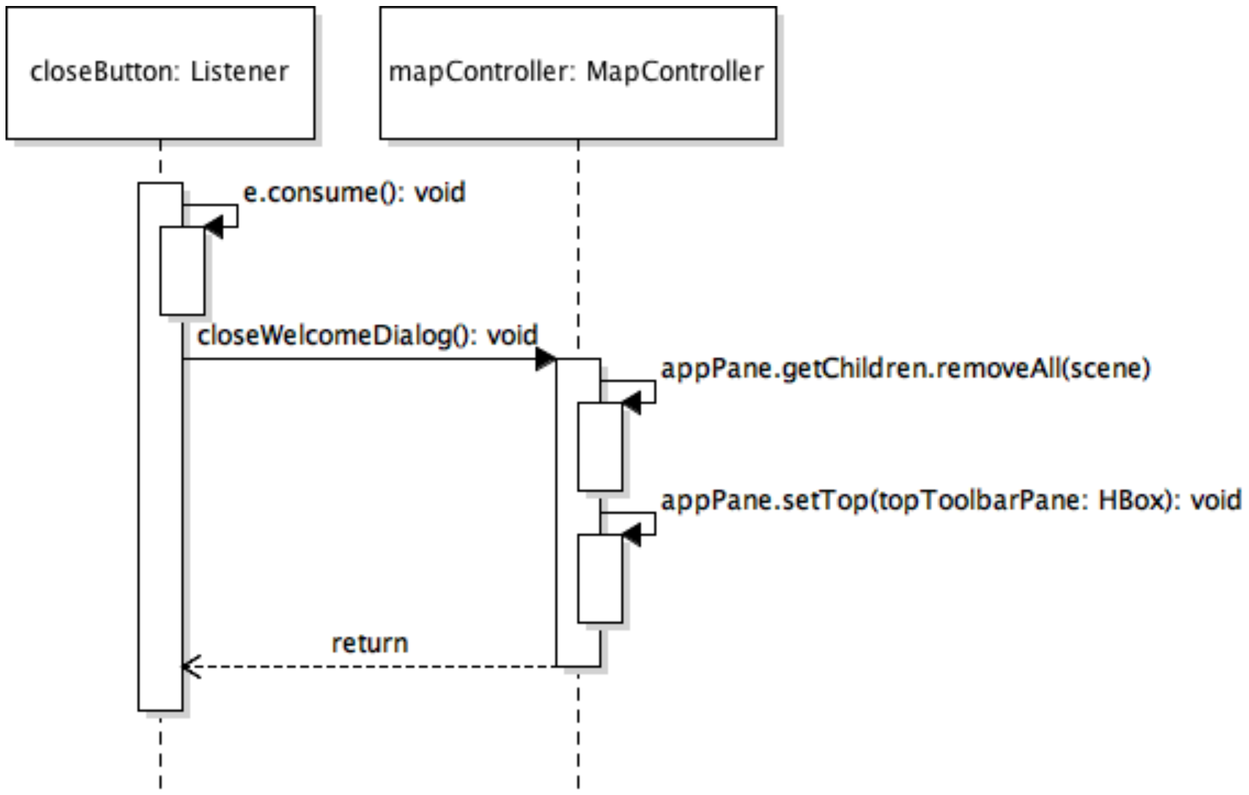

closeButton: Listener
mapController: MapController
e.consume(): void
closeWelcomeDialog(): void
appPane.getChildren.removeAll(scene)
appPane.setTop(topToolbarPane: HBox): void
return

Figure 4.3: Close Welcome Dialog UML Sequence Diagram (use case 2.3)

Note: When the close button is pressed in the welcome dialog, the welcome dialog is closed and the main UI is loaded without the workspace being loaded. Only the toolbar is activated. The program is now resizable.

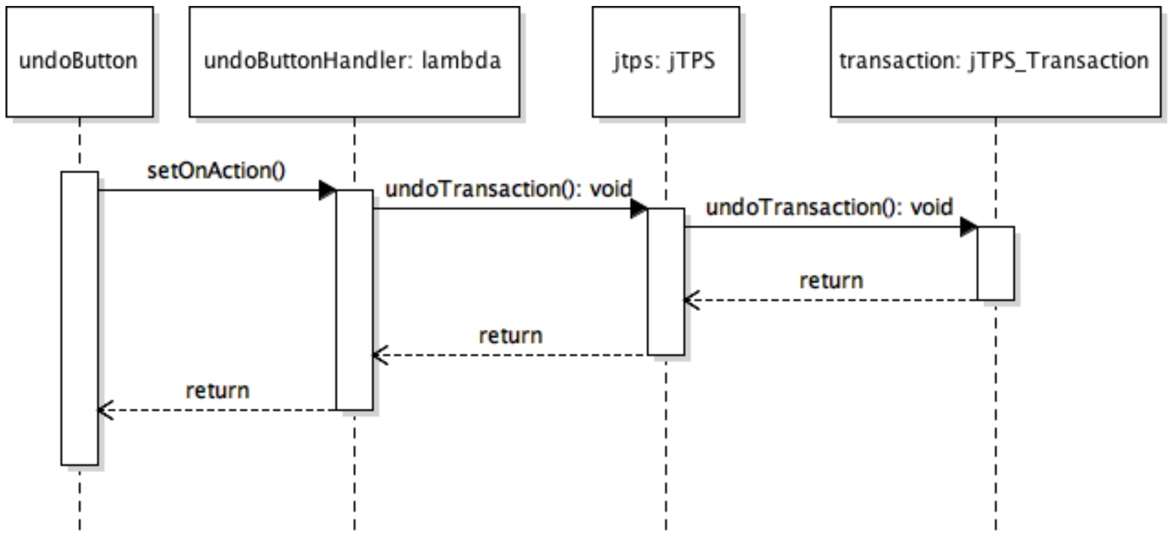


Figure 4.4: undo Edit UML Sequence Diagram (use case 2.9)

Note: This action is done every time the user clicks on the undo button. It will undo the most recent transaction. This is a similar process when the user clicks on the redo button. This uses the memento design pattern.

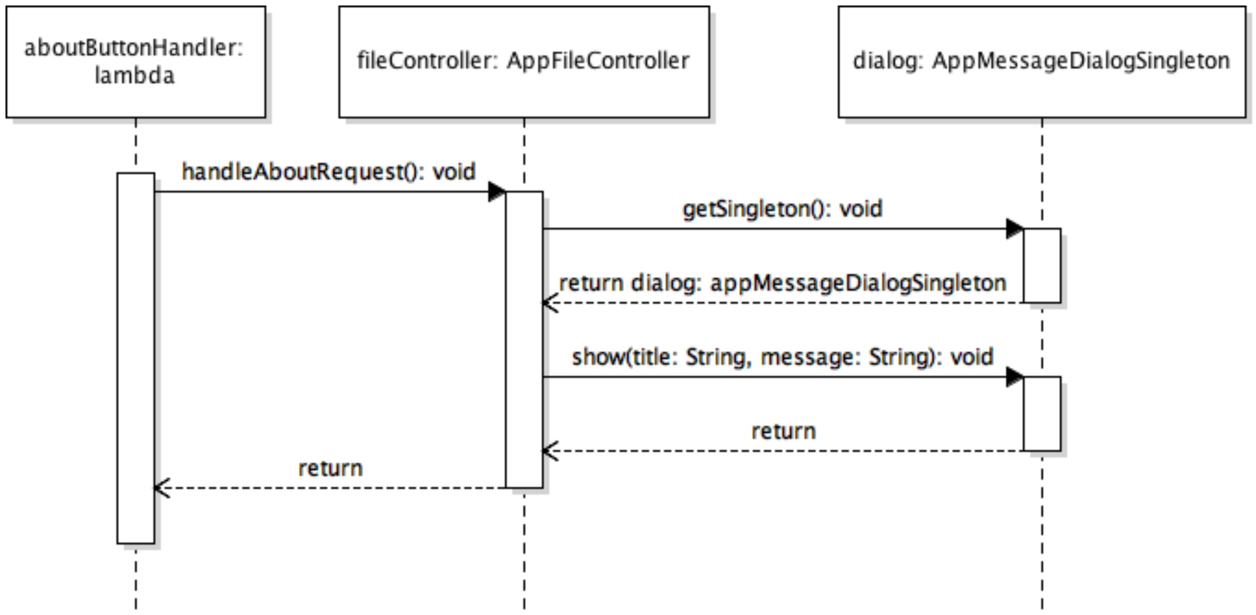

aboutButtonHandler: lambda
fileController: AppFileController
dialog: AppMessageDialogSingleton
handleAboutRequest(): void
getSingleton(): void
return dialog: appMessageDialogSingleton
show(title: String, message: String): void
return
return

Figure 4.5: Learn About Application UML Sequence Diagram (use case 2.11)

Note: This is handled in the AppFileController. A pop out dialog will show which shows the information which the programmer put in.

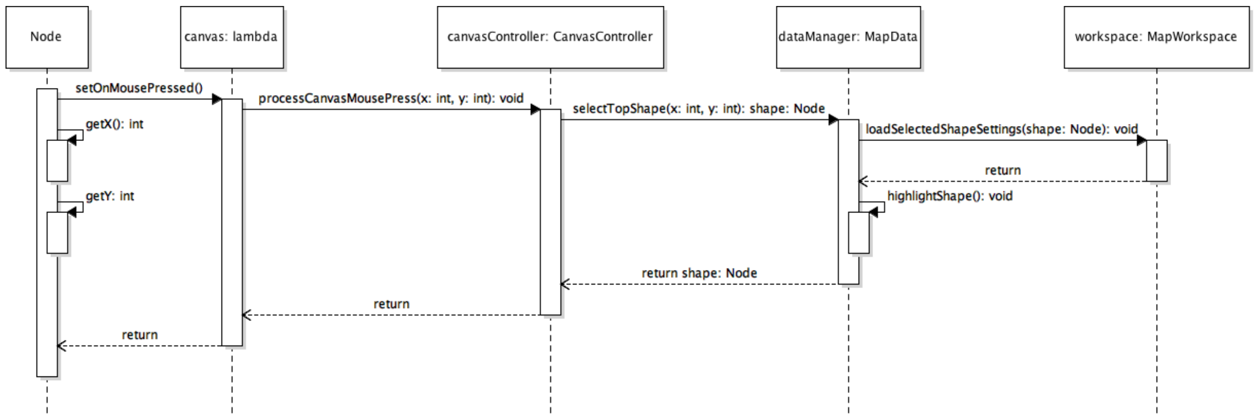

Node
canvas: lambda
canvasController: CanvasController
dataManager: MapData
workspace: MapWorkspace
setOnMousePressed()
processCanvasMousePress(x: int, y: int): void
selectTopShape(x: int, y: int): shape: Node
loadSelectedShapeSettings(shape: Node): void
getX(): int
getY: int
return
highlightShape(): void
return shape: Node
return
return

Figure 4.6a: Select node by clicking UML Sequence Diagram

Note: When the user clicks on the text or end of the line, the text of that line (only one side) will be highlighted. As a result, the user is able to drag this end anywhere within the canvas. When remove line button is clicked, the whole MetroLine object/node will be removed from the canvas.

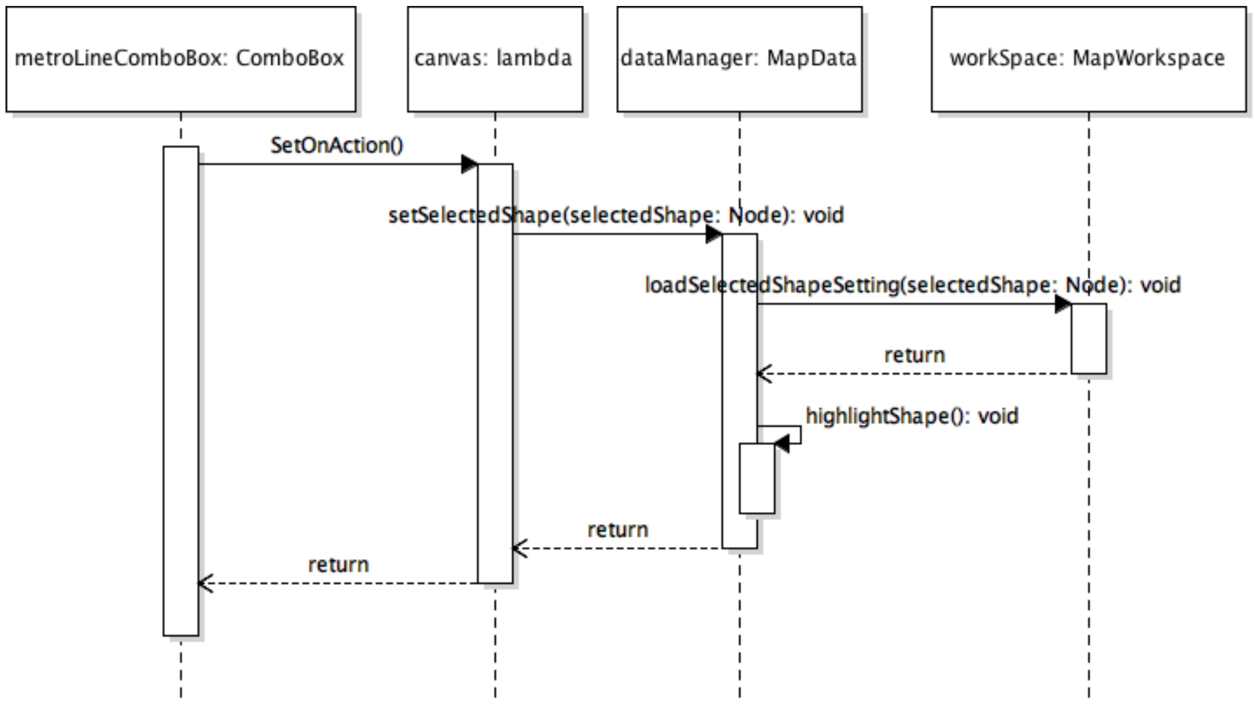

metroLineComboBox: ComboBox
canvas: lambda
dataManager: MapData
workSpace: MapWorkspace
SetOnAction()
setSelectedShape(selectedShape: Node): void
loadSelectedShapeSetting(selectedShape: Node): void
return
highlightShape(): void
return
return

Figure 4.6b: Select line through combo box UML Sequence Diagram

Note: This action selects a line when the user chooses a line from the combo box. As a result, the whole line is highlighted. When the user wants to drag, the line will unhighlight and only the end which the user clicks on will highlight.

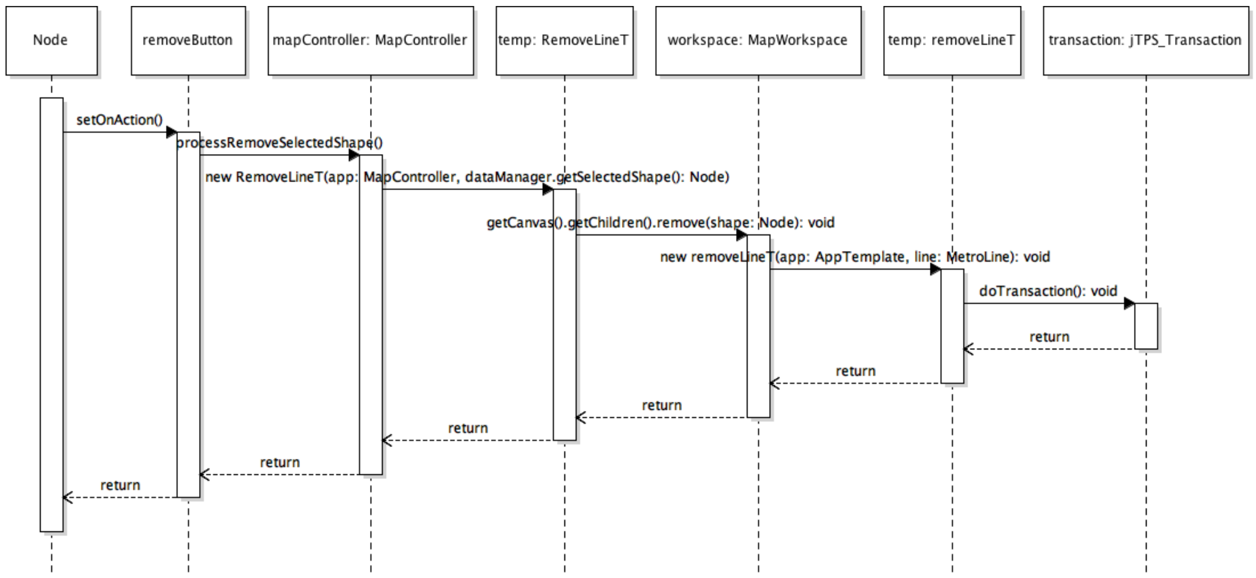

Node
removeButton
mapController: MapController
temp: RemoveLineT
workspace: MapWorkspace
temp: removeLineT
transaction: jTPS_Transaction
setOnAction()
processRemoveSelectedShape()
new RemoveLineT(app: MapController, dataManager.getSelectedShape(): Node)
getCanvas().getChildren().remove(shape: Node): void
new removeLineT(app: AppTemplate, line: MetroLine): void
doTransaction(): void
return
return
return
return
return
return

Figure 4.6: Remove Line UML Sequence Diagram (use case 2.13)

Note: Refer to figure 4.6a and 4.6b to select node. User clicks on the remove line button and the selected line will be removed. If no line is selected, nothing will happen.

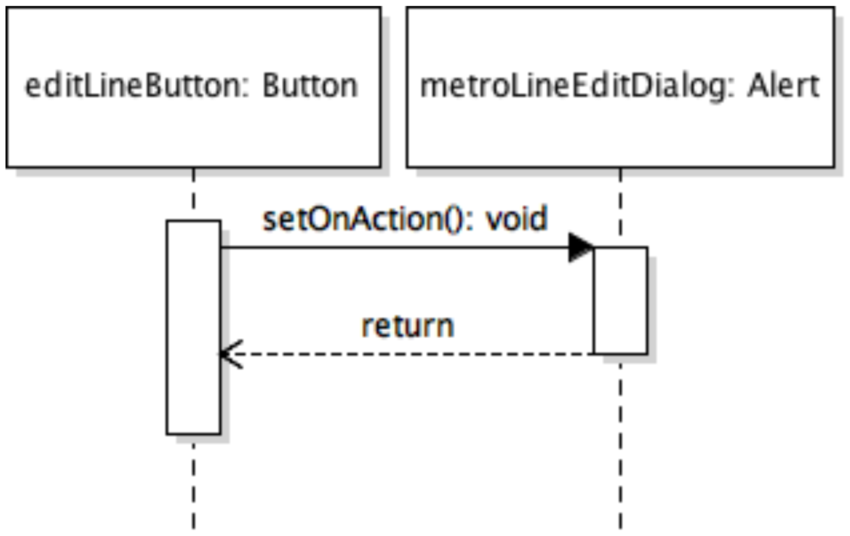


Figure 4.7a: Edit Line Button show dialog Sequence Diagram

Note: This will show the edit line dialog.

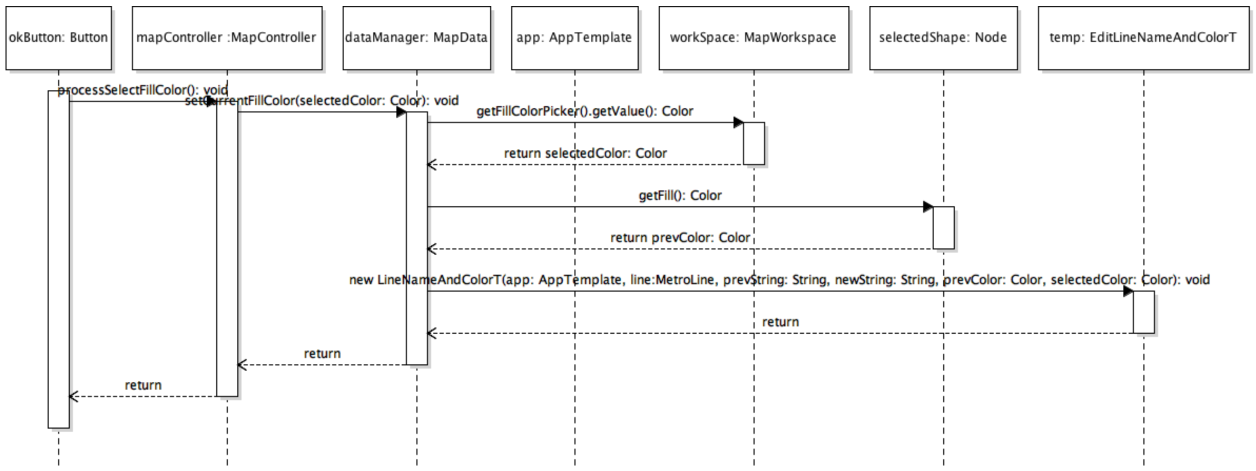

okButton: Button
mapController :MapController
dataManager: MapData
app: AppTemplate
workSpace: MapWorkspace
selectedShape: Node
temp: EditLineNameAndColorT
processSelectFillColor(): void
setCurrentFillColor(selectedColor: Color): void
getFillColorPicker().getValue(): Color
return selectedColor: Color
getFill(): Color
return prevColor: Color
new LineNameAndColorT(app: AppTemplate, line:MetroLine, prevString: String, newString: String, prevColor: Color, selectedColor: Color): void
return
return
return

Figure 4.7: Edit Line UML Sequence Diagram (use case 2.14)

Note: Refer to Figure 4.6a and 4.6b to select line. Refer to Figure 4.7a to show dialog.

The edit line dialog will show with the color and name value of the current line. If the user decides to change any attribute, program will record the transaction and change the value of the line. If the user clicks ok but does not change any attribute or clicks cancel, nothing occurs. This is to prevent too many null transactions.

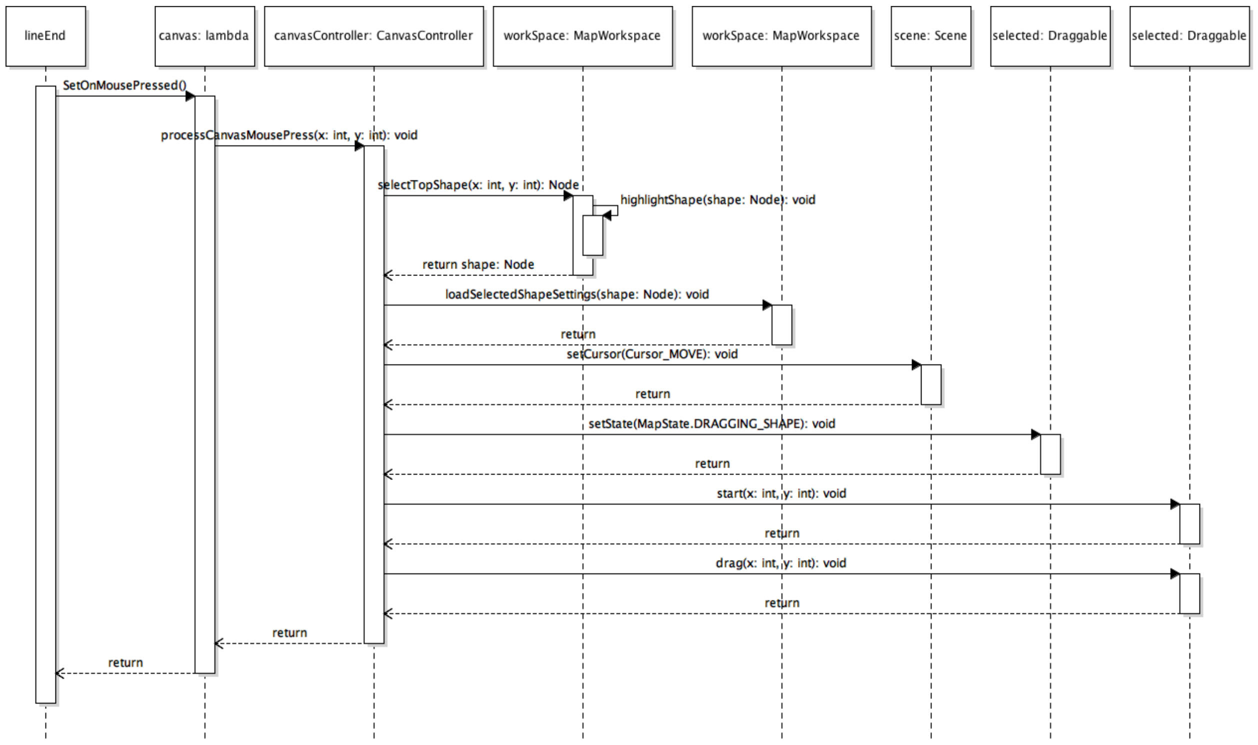

lineEnd
canvas: lambda
canvasController: CanvasController
workSpace: MapWorkspace
workSpace: MapWorkspace
scene: Scene
selected: Draggable
selected: Draggable
SetOnMousePressed()
processCanvasMousePress(x: int, y: int): void
selectTopShape(x: int, y: int): Node
highlightShape(shape: Node): void
return shape: Node
loadSelectedShapeSettings(shape: Node): void
return
setCursor(Cursor_MOVE): void
return
setState(MapState.DRAGGING_SHAPE): void
return
start(x: int, y: int): void
return
drag(x: int, y: int): void
return
return
return

Figure 4.8a: Select node and put state to dragging mode UML Sequence Diagram.

Note: This action will select a node and change the state of the program to dragging mode which will drag the end of the chosen line.

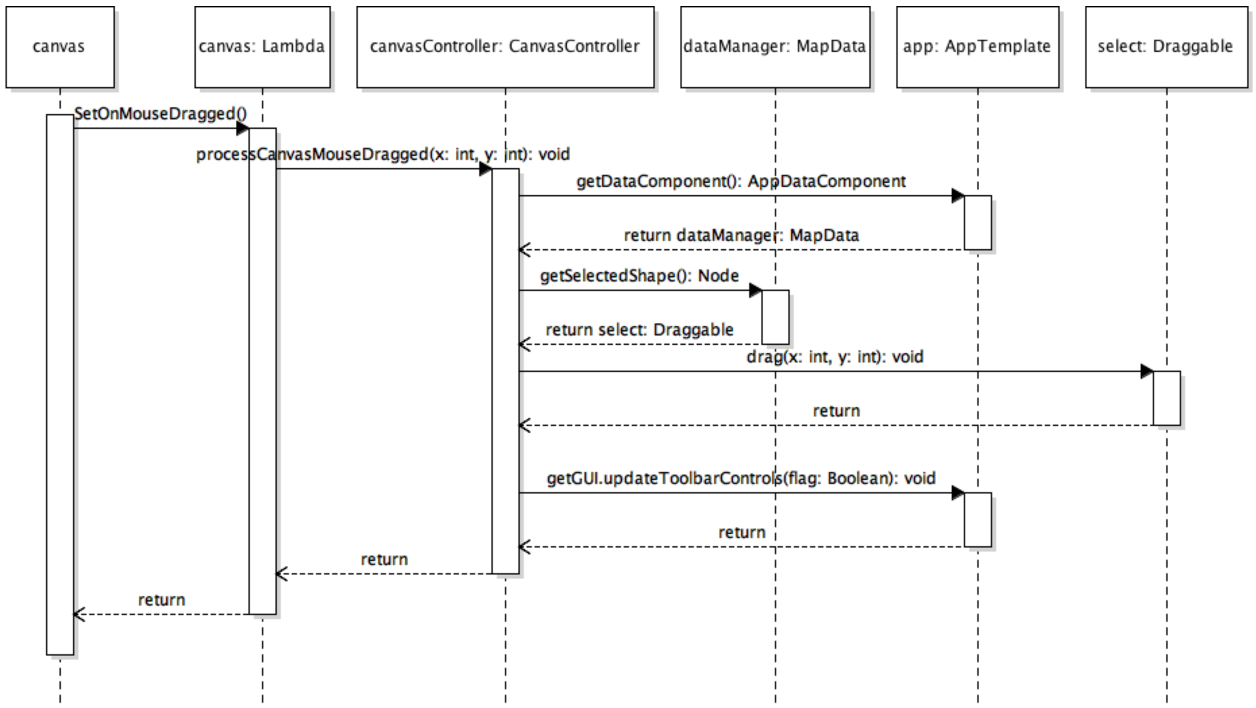

canvas
canvas: Lambda
canvasController: CanvasController
dataManager: MapData
app: AppTemplate
select: Draggable
SetOnMouseDragged()
processCanvasMouseDragged(x: int, y: int): void
getDataComponent(): AppDataComponent
return dataManager: MapData
getSelectedShape(): Node
return select: Draggable
drag(x: int, y: int): void
return
getGUI.updateToolbarControls(flag: Boolean): void
return
return
return

Figure 4.8: Move Line End UML Sequence Diagram (use case 2.15)

Note: Refer to Figure 4.8a to change mode. This action will go through jTPS upon mouse release. This action will drag the chosen line.

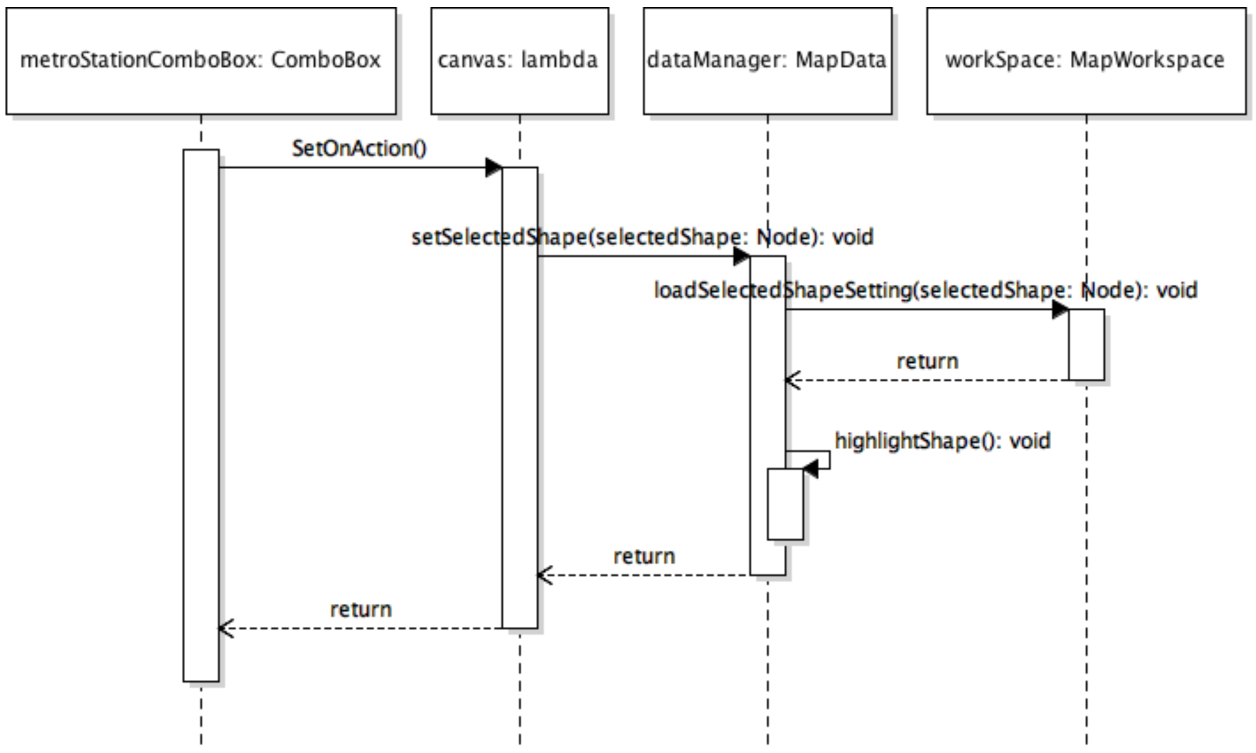

metroStationComboBox: ComboBox
canvas: lambda
dataManager: MapData
workSpace: MapWorkspace
SetOnAction()
setSelectedShape(selectedShape: Node): void
loadSelectedShapeSetting(selectedShape: Node): void
return
highlightShape(): void
return
return

Figure 4.9a: Select station from combo box UML Sequence Diagram

Note: The user will choose a metro station and that station will be highlighted in the canvas. All values such as the color of this station will show in the left toolbar. If the station is not on the canvas, nothing on the canvas will occur.

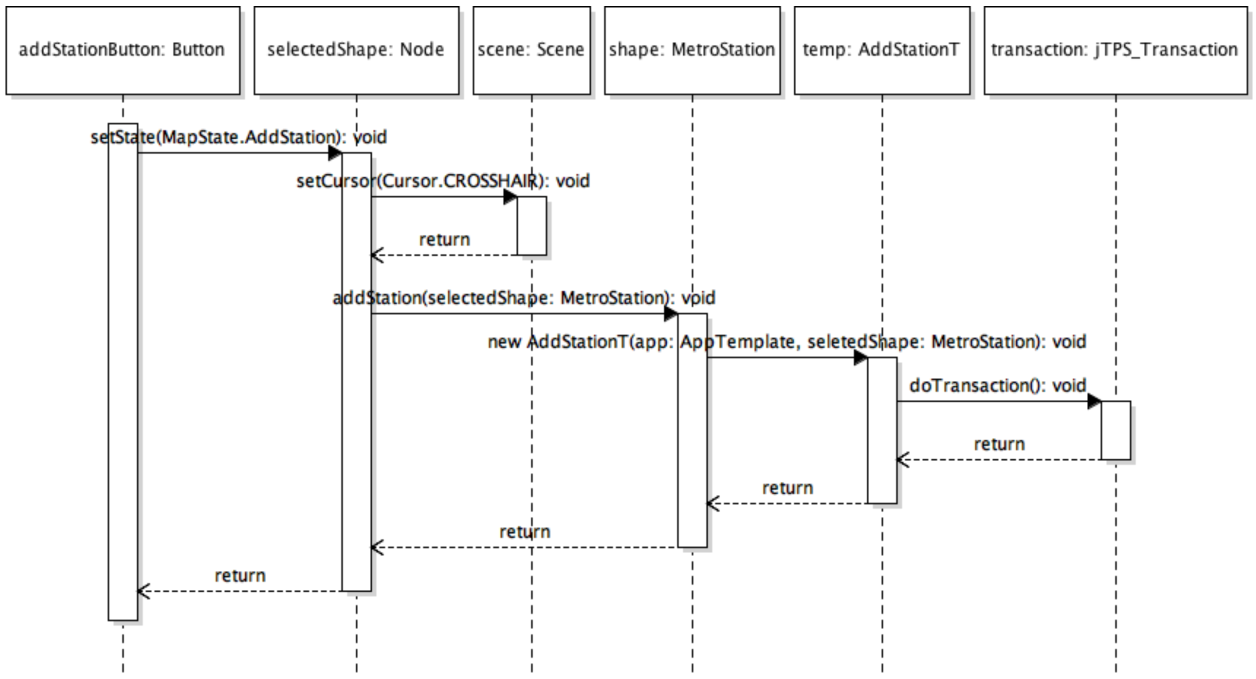

addStationButton: Button
selectedShape: Node
scene: Scene
shape: MetroStation
temp: AddStationT
transaction: jTPS_Transaction
setState(MapState.AddStation): void
setCursor(Cursor.CROSSHAIR): void
return
addStation(selectedShape: MetroStation): void
new AddStationT(app: AppTemplate, seletedShape: MetroStation): void
doTransaction(): void
return
return
return
return

Figure 4.9: Add Stations to Line UML Sequence Diagram (use case 2.16)

Note: Refer to Figure 4.9a to select station from combo box.

If the station is not on the canvas, program will be set to add station mode and upon the user clicking on any line, the station will be added. If the station is already on the canvas, the addStationButton is disabled and user is not allowed to add the station. They can remove the station and add it to another line or drag the station along the line.

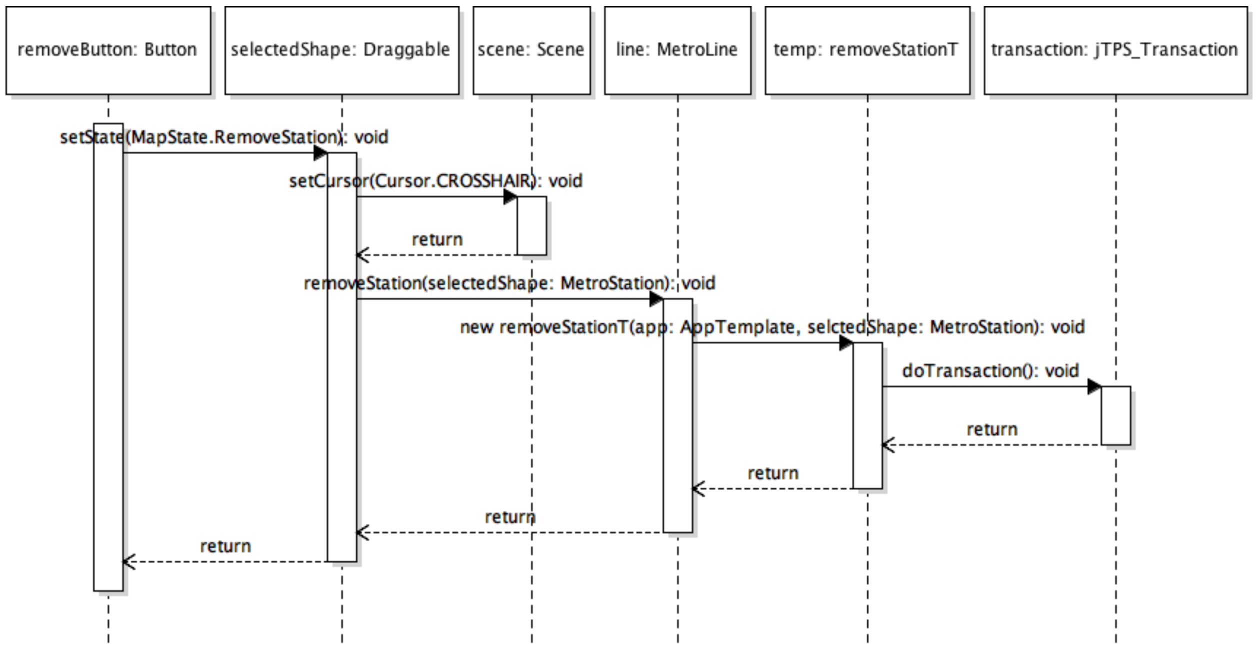

removeButton: Button
selectedShape: Draggable
scene: Scene
line: MetroLine
temp: removeStationT
transaction: jTPS_Transaction
setState(MapState.RemoveStation): void
setCursor(Cursor.CROSSHAIR): void
return
removeStation(selectedShape: MetroStation): void
new removeStationT(app: AppTemplate, selctedShape: MetroStation): void
doTransaction(): void
return
return
return
return

Figure 4.10: Remove stations from Line UML Sequence Diagram (use case 2.17)

Note: Refer to Figure 4.9a to select station from combo box. This will set the state to remove station and the cursor will change. Upon change, the user can click on any station which will remove the station from that line. The station still exists in the program but not on the canvas.

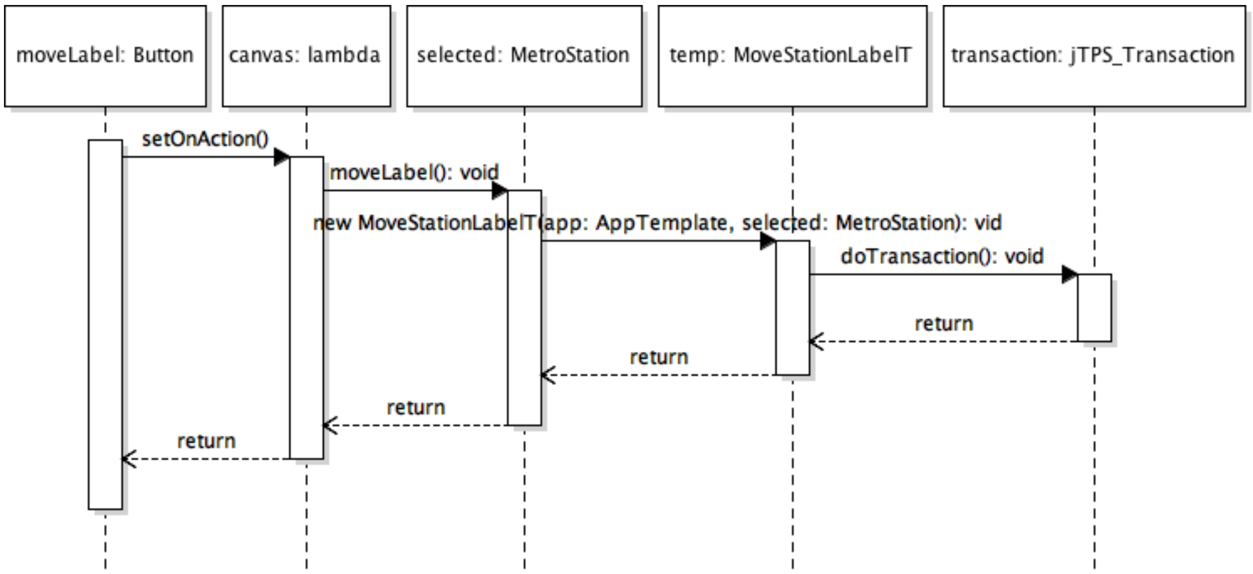


Figure 4.11: Move Station Label UML Sequence Diagram (use case 2.23)

Note: Refer to Figure 4.9a to select station from combo box. Refer to 4.6a to select station by clicking on it. This action will move the station label to the designated 4 locations, top, left, bottom or right of the station.

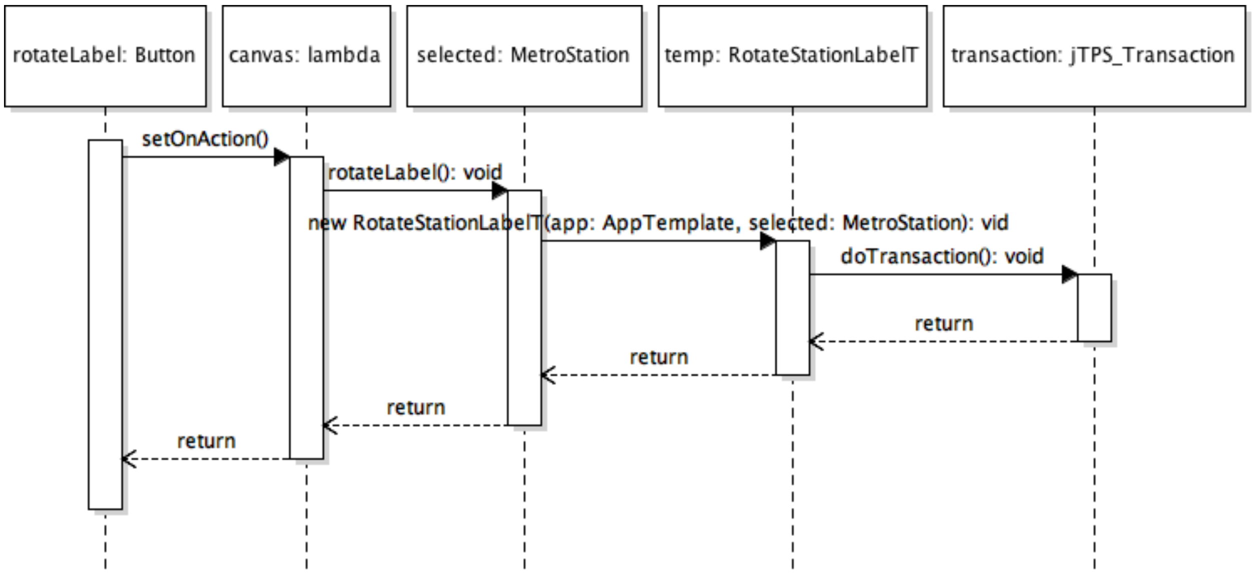


Figure 4.12: Rotate Station Label UML Sequence Diagram (use case 2.27)

Note: Refer to Figure 4.9a to select station from combo box. Refer to 4.6a to select station by clicking on it. This will rotate the label by 90 degrees. Similar concept is applied to the metro line text.

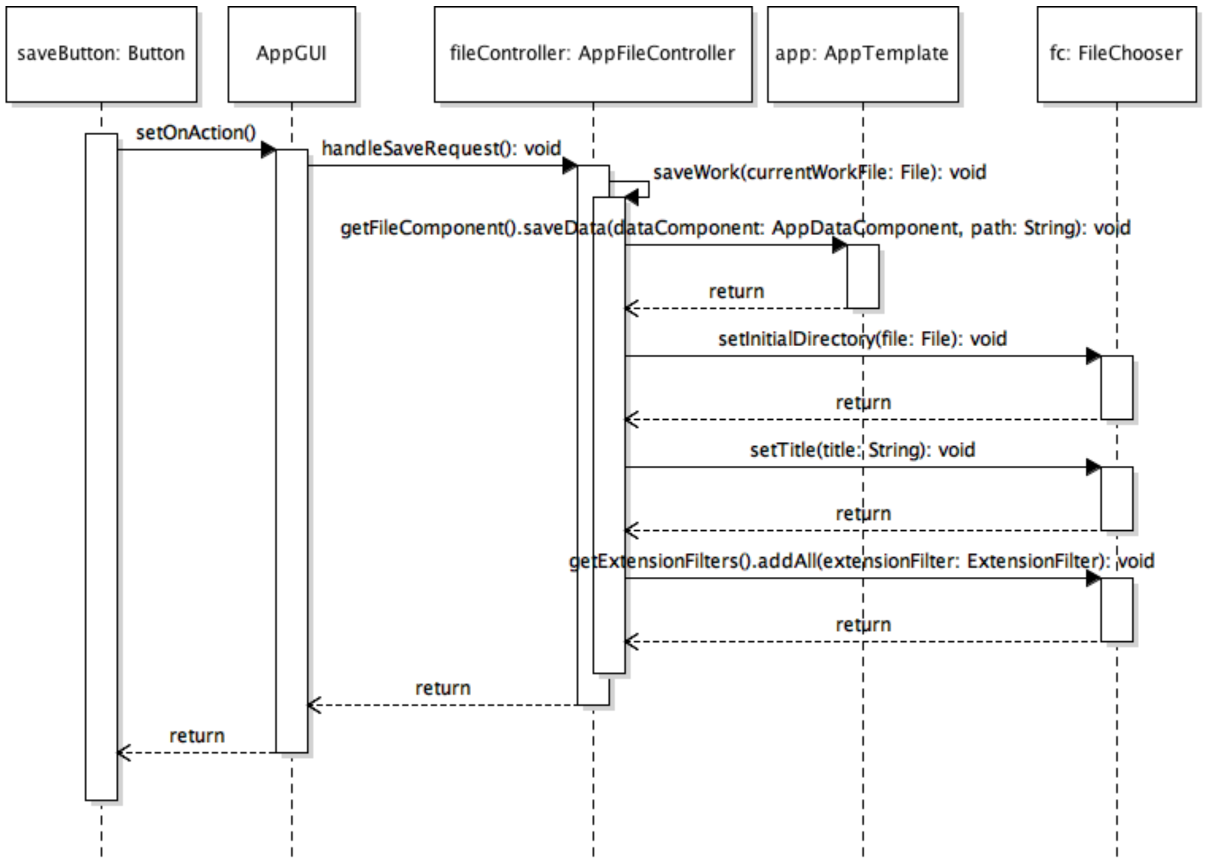

saveButton: Button
AppGUI
fileController: AppFileController
app: AppTemplate
fc: FileChooser
setOnAction()
handleSaveRequest(): void
saveWork(currentWorkFile: File): void
getFileComponent().saveData(dataComponent: AppDataComponent, path: String): void
return
setInitialDirectory(file: File): void
return
setTitle(title: String): void
return
getExtensionFilters().addAll(extensionFilter: ExtensionFilter): void
return
return
return

Figure 4.13: Save Map UML Sequence Diagram (use case 2.6)

Note: This will save the file. If the file already exists, the file will just save. If the file was not once saved, program will prompt user to enter name for the file. In the saveWork method, time is recorded such that this file will be visible when the welcome dialog is opened.

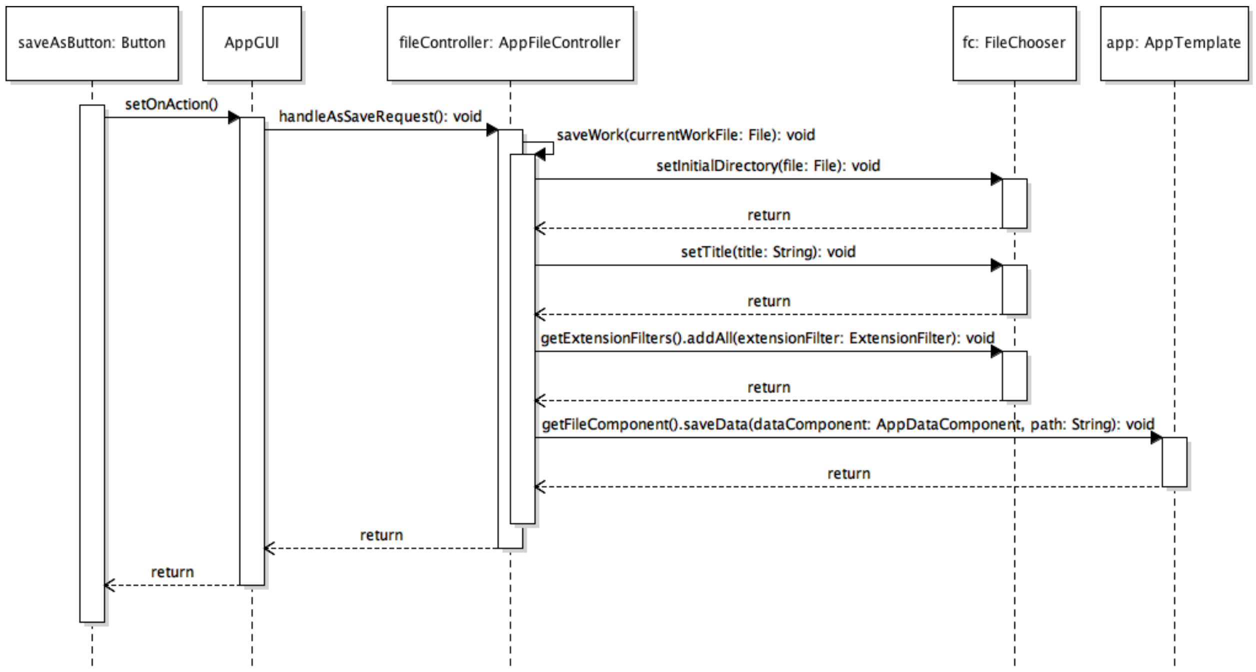

saveAsButton: Button
AppGUI
fileController: AppFileController
fc: FileChooser
app: AppTemplate
setOnAction()
handleAsSaveRequest(): void
saveWork(currentWorkFile: File): void
setInitialDirectory(file: File): void
return
setTitle(title: String): void
return
getExtensionFilters().addAll(extensionFilter: ExtensionFilter): void
return
getFileComponent().saveData(dataComponent: AppDataComponent, path: String): void
return
return
return

Figure 4.14: Save as Map UML Sequence Diagram (use case 2.7)

Note: This is a similar feature as the save button (figure 4.13) but it will not check if the file was already saved.

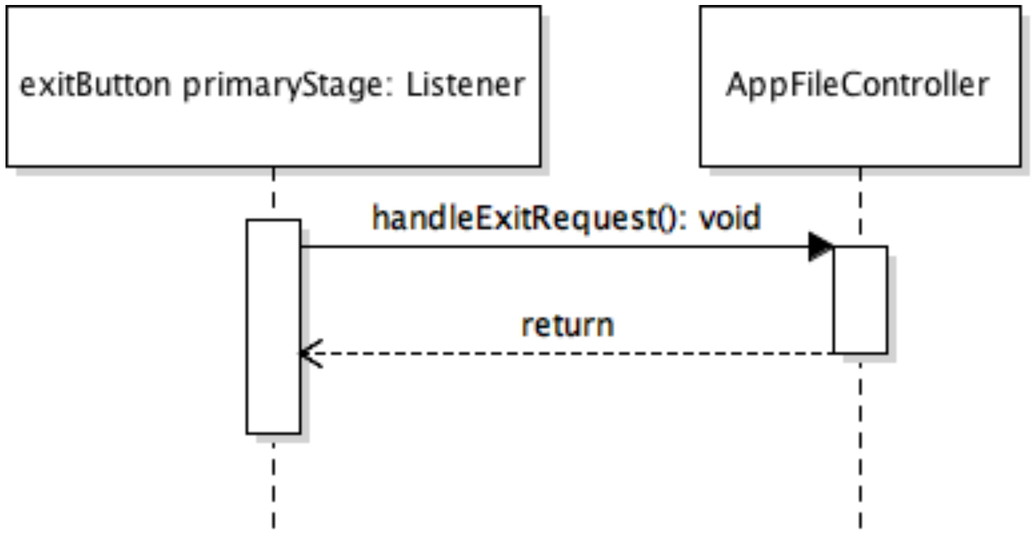


Figure 4.15: Exit Application UML Sequence Diagram (use case 2.39)

Note: This action will close the whole program. This only works for the main UI. If file is not saved, program will ask user to save, otherwise, all edits will be deleted and program will close.

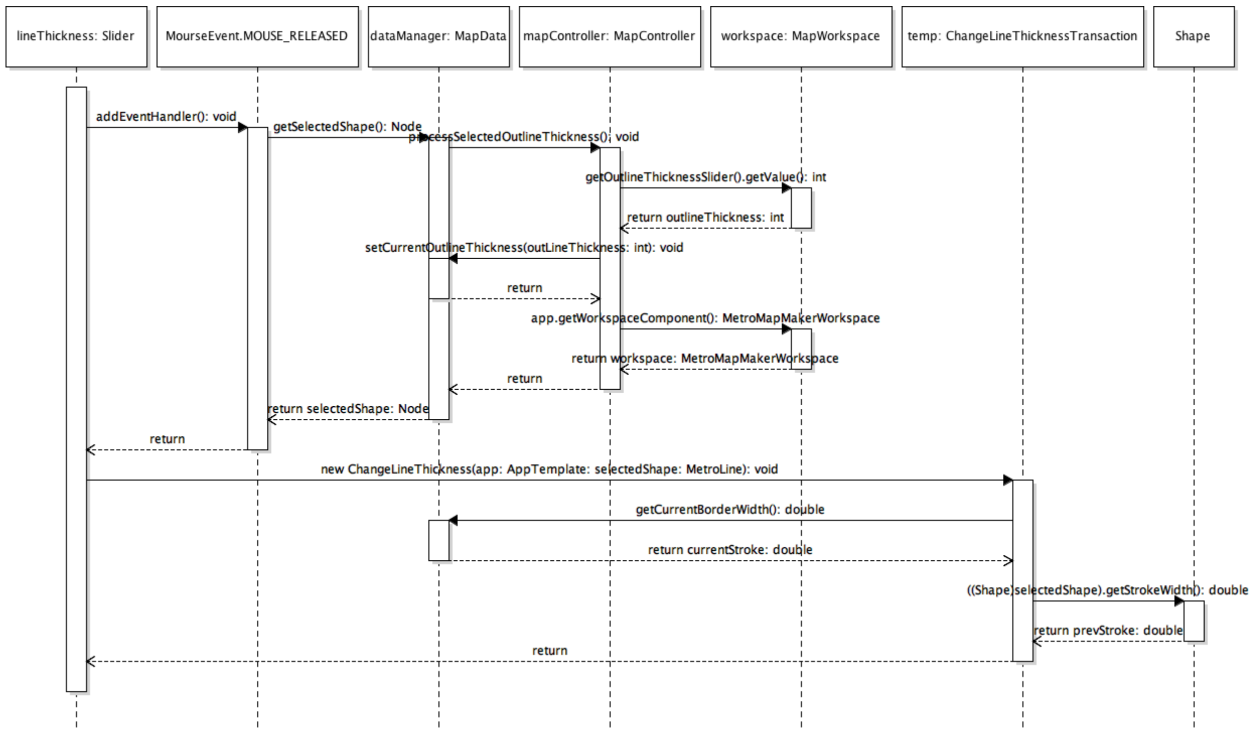

lineThickness: Slider
MourseEvent.MOUSE_RELEASED
dataManager: MapData
mapController: MapController
workspace: MapWorkspace
temp: ChangeLineThicknessTransaction
Shape
addEventHandler(): void
getSelectedShape(): Node
processSelectedOutlineThickness(): void
getOutlineThicknessSlider().getValue(): int
return outlineThickness: int
setCurrentOutlineThickness(outLineThickness: int): void
return
app.getWorkspaceComponent(): MetroMapMakerWorkspace
return workspace: MetroMapMakerWorkspace
return
return selectedShape: Node
return
new ChangeLineThickness(app: AppTemplate: selectedShape: MetroLine): void
getCurrentBorderWidth(): double
return currentStroke: double
((Shape)selectedShape).getStrokeWidth(): double
return prevStroke: double
return

Figure 4.16: Change Line Thickness UML Sequence Diagram (use case 2.19)

Note: Refer to Figure 4.6a and 4.6b to select node. This action is for when the user drags the slider. The line thickness will change. When the slider is released, the value is recorded.

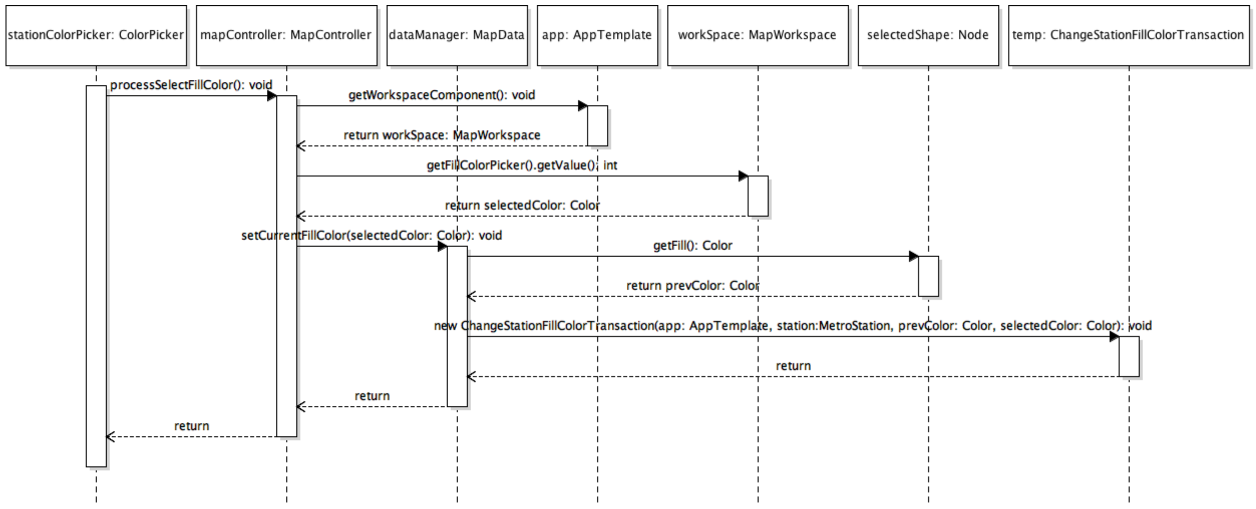
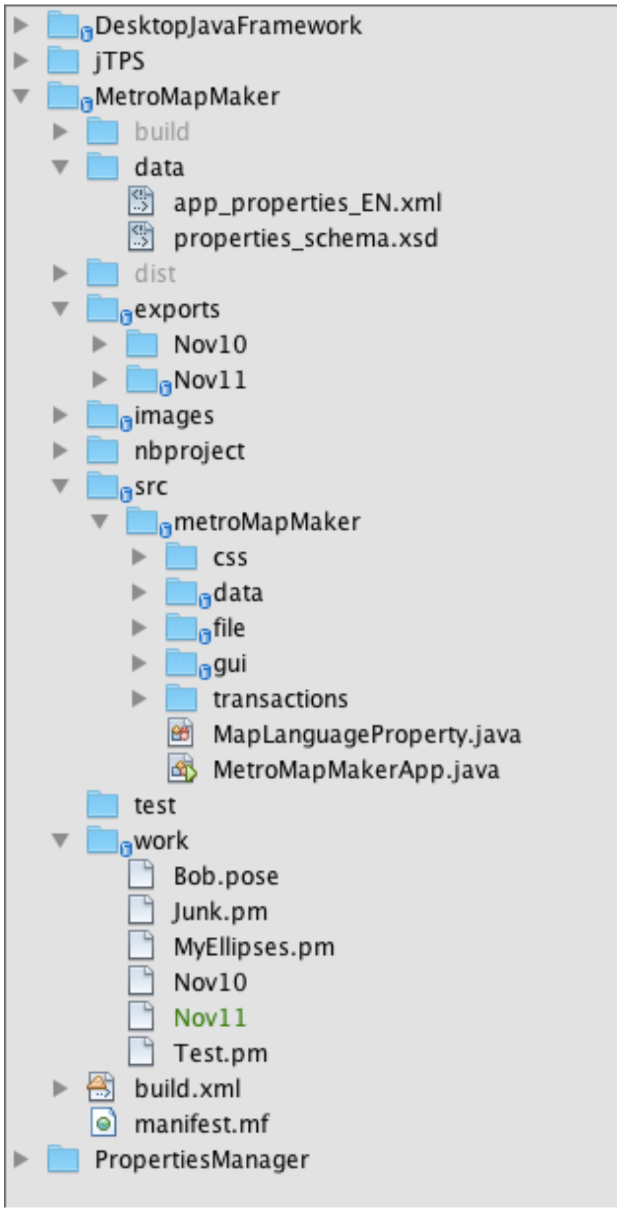

stationColorPicker: ColorPicker
mapController: MapController
dataManager: MapData
app: AppTemplate
workSpace: MapWorkspace
selectedShape: Node
temp: ChangeStationFillColorTransaction
processSelectFillColor(): void
getWorkspaceComponent(): void
return workSpace: MapWorkspace
getFillColorPicker().getValue(): int
return selectedColor: Color
setCurrentFillColor(selectedColor: Color): void
getFill(): Color
return prevColor: Color
new ChangeStationFillColorTransaction(app: AppTemplate, station:MetroStation, prevColor: Color, selectedColor: Color): void
return
return
return

Figure 4.17: Change Station Fill Color UML Sequence Diagram (use case 2.25)

Note: This changes the station color and the text the station is associated with. The user selects the color in the colorpicker provided.

5 File Structure and Formats

Note that the DesktopFamework, PropertiesManager, jTPS, and JavaX.JSON will be provided inside DesktopJavaFramework.jar, PropertiesManager.jar, jTPS.jar, and JavaX.json-1.0.4.jar, which are all Java Archive file that will encapsulate the entire framework. This should be imported into the necessary project for the Metro Map Maker application and will be included in the deployment of a single, executable JAR file titled MetroMapMaker.jar. Note that all necessary data and art files must accompany this program. 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.



```
▶ DesktopJavaFramework
▶ jTPS
▼ MetroMapMaker
    ▶ build
    ▼ data
        app_properties_EN.xml
        properties_schema.xsd
    ▶ dist
    ▼ exports
        ▶ Nov10
        ▶ Nov11
    ▶ images
    ▶ nbproject
    ▼ src
        ▼ metroMapMaker
            ▶ css
            ▶ data
            ▶ file
            ▶ gui
            ▶ transactions
              MapLanguageProperty.java
              MetroMapMakerApp.java
      test
    ▼ work
        Bob.pose
        Junk.pm
        MyEllipses.pm
        Nov10
        Nov11
        Test.pm
    ▶ build.xml
      manifest.mf
▶ PropertiesManager
```

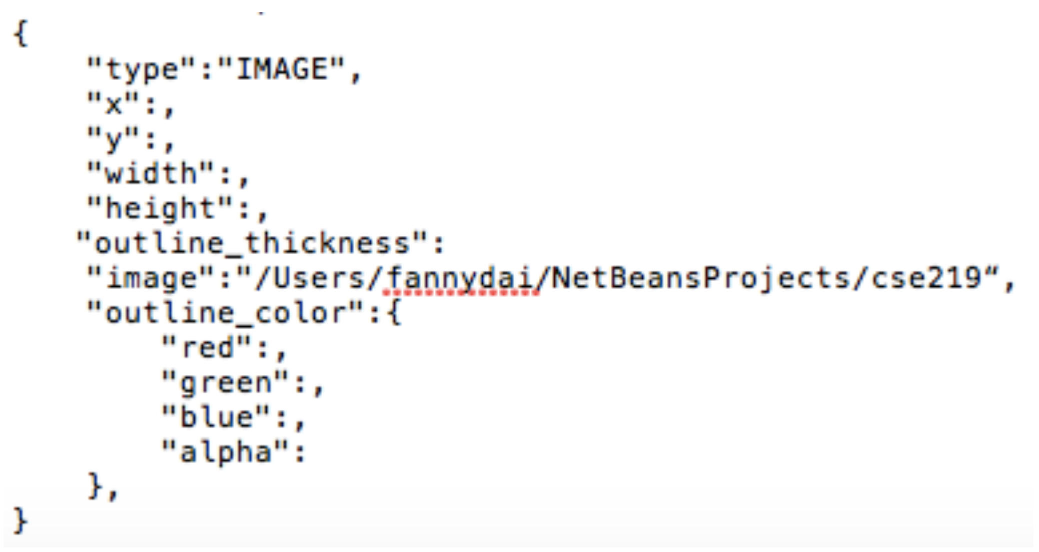
Figure 5.1: MetroMapMaker program and File structure

```
 <property name="APP_TITLE"        value="Metro Map Maker"></property>
 <property name="WELCOME_TITLE"           value="Welcome to the Metro Map Maker"></property>
<property name="APP_LOGO"          value="metroMapMakerLogo.png"></property>
<property name="APP_PATH_CSS"      value="css/"></property>
<property name="APP_CSS"           value="Map_style.css"></property>

 <!-- PREFERRED GEOMETRY -->
 <property name="PREF_WIDTH"         value="1000"></property>
 <property name="PREF_HEIGHT"        value="600"></property>
 <property name="START_MAXIMIZED"    value="true"></property>

 <!-- PROGRAM ICON FILES -->
 <property name="NEW_ICON"         value="New.png"></property>
 <property name="LOAD_ICON"        value="Load.png"></property>
 <property name="SAVE_ICON"        value="Save.png"></property>
 <property name="SAVEAS_ICON"      value="SaveAs.png"></property>
 <property name="EXPORT_ICON"          value="Export.png"></property>
 <property name="UNDO_ICON"        value="Undo.png"></property>
 <property name="REDO_ICON"        value="Redo.png"></property>
 <property name="ABOUT_ICON"       value="About.png"></property>

 <!-- TOOLTIPS FOR BUTTONS -->
 <property name="NEW_TOOLTIP"        value="Create a New Map"></property>
 <property name="LOAD_TOOLTIP"       value="Load an Existing Map"></property>
 <property name="SAVE_TOOLTIP"       value="Save this Map"></property>
 <property name="SAVEAS_TOOLTIP"             value="Save As this Map"></property>
 <property name="EXPORT_TOOLTIP"             value="Export this Map"></property>
 <property name="UNDO_TOOLTIP"       value="Undo Transaction"></property>
 <property name="REDO_TOOLTIP"       value="Redo Transaction"></property>
 <property name="ABOUT_TOOLTIP"              value="Information"></property>
```

Figure 5.2: app_properties_EN.xml format
This is not the complete file but it is a peek of how it should look. This file contains the name of the program, the logo file name, the icons for each button, the tooltips and error messages.

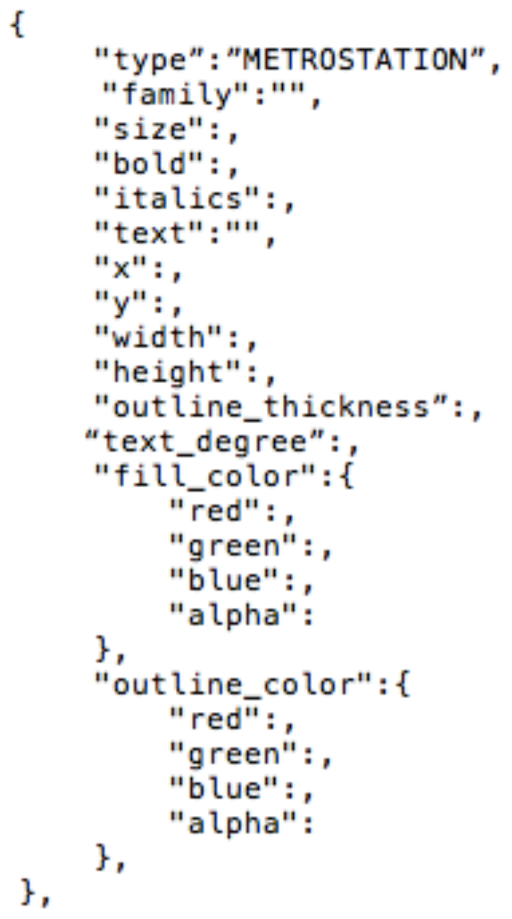
```
{
    "type":"TEXT",
    "x":,
    "y":,
    "family":"",
    "size":,
    "bold":,
    "italics":,
    "text":"",
    "outline_thickness":
    "fill_color":{
        "red":,
        "green":,
        "blue":,
        "alpha":
    },
     "outline_color":{
        "red":,
        "green":,
        "blue":,
        "alpha":
    },
},
```

Figure 5.3: .json format of text object.



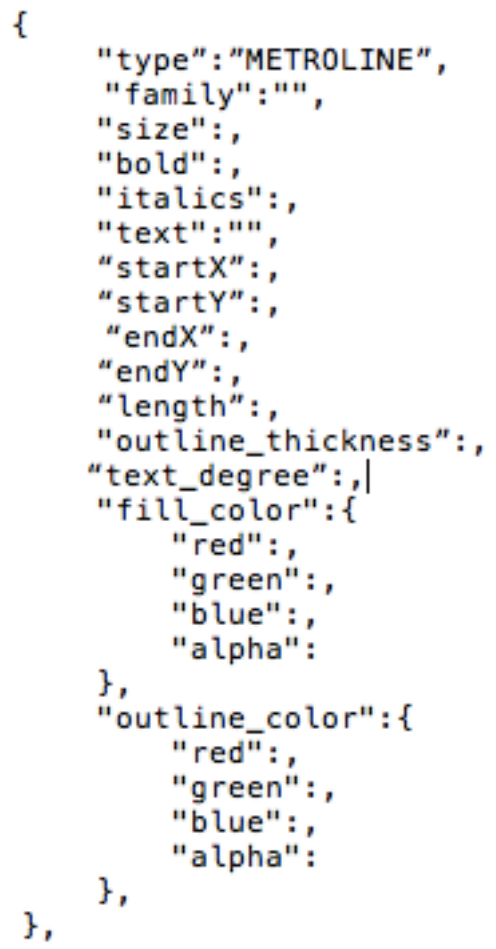
```
{
    "type":"IMAGE",
    "x":,
    "y":,
    "width":,
    "height":,
   "outline_thickness":
    "image":"/Users/fannydai/NetBeansProjects/cse219",
    "outline_color":{
        "red":,
        "green":,
        "blue":,
        "alpha":
    },
}
```

Figure 5.4: .json format of image object



```
 {
     "type”:”METROSTATION”,
      "family":"",
     "size":,
     "bold":,
     "italics":,
     "text":"",
     "x":,
     "y":,
     "width":,
     "height":,
     "outline_thickness”:,
    “text_degree”:,
     "fill_color":{
         "red":,
         "green":,
         "blue":,
         "alpha":
     },
     "outline_color":{
         "red":,
         "green":,
         "blue":,
         "alpha":
     },
 },
```

Figure 5.5: .json format of station object



```
{
     "type”:”METROLINE”,
      "family":"",
     "size":,
     "bold":,
     "italics":,
     "text":"",
     “startX”:,
     “startY”:,
      “endX”:,
     “endY”:,
     “length”:,
     "outline_thickness”:,
    “text_degree”:,
     "fill_color":{
          "red":,
          "green":,
          "blue":,
          "alpha":
     },
     "outline_color":{
          "red":,
          "green":,
          "blue":,
          "alpha":
     },
 },
```

Figure 5.6: .json format of line object

type: They type of object. 4 categories: text, image, metrostation or metroline.

width: The width of the image or ellipse of the station. (applies only to station and image)

height: The height of the image or ellipse of the station. (applies only to station and image)

image: The path of the image. (applies only to image)

startX: The start x coordinate of the line. (applies only to metro line)

startY: The start y coordinate of the line. (applies only to metro line)

endX: The end x coordinate of the line. (applies only to metro line)

endY: The end y coordinate of the line. (applies only to metro line)

length: The length of the line. (applies only to metro line)

text_degree: The rotation of the text bind to station or line. (applies only to metro line and station)

x: The x coordinate on the canvas of this object. The most left position. (does not apply to station and line)

y: The y coordinate on the canvas of this object. The most right position. (does not apply to station and line)

family: The font family of the text (applies to text, the label bind to station and line)

size: The font size of the text (applies to text, the label bind to station and line)

bold: Whether the text is bold or not. (applies to text, the label bind to station and line)

italics: Whether the text is italics or not. (applies to text, the label bind to station and line)

fillcolor: The color of the shape. (applies to station and text)

outline_thickness: The outline thickness of the shape. (Applies to all)

outlinecolor: The outline color of the shape. (Applies to all)

Note: Not all objects have the same variables.

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

Debugging Enterprises