DataViLiJ ™

Software Design Description

Author: Britney Echeverria
Stony Brook University Enterprises
March 2018
Version 1.0

Abstract: This document gives a detailed outline for the intended software design for DataViLiJ, a program developed to allow for the visual analysis of data using different algorithms.

Based on IEEE Std 830 - 19998 (R2009) document format

Copyright © 2018 Stony Brook University Enterprises, which is a made up company.

Table of Contents

1 Introduction

This is the software Design Description (SDD) for the DataViLiJ application. The document format is based on the IEEE Standard 1016-2009 recommendation for software design.

1.1 Purpose

This document is meant to lay out the construction of the DataViLiJ application. Through the use of UML diagrams, all packages, classes, instance variables, class variables and method signatures required for the application will be explained thoroughly. The intended audience for this document is for all members of the development team. UML Sequence diagrams will be used to specify object interactions or timed events. After reading this document, one should be able to understand how the application was meant to operate.

1.2 Scope

DataViLiJ will be an application for students and beginner professionals in AI to understand on a visual level the inner mechanisms of algorithms. Tools used for the development of the application include DataViLiJ Framework and Properties Manager, due to the necessity for this application to be modified. This design contains design descriptions for the application. Java is the target language for this software design.

1.3 Definitions, Acronyms and Abbreviations

Algorithm: a process used in this application to analyze data and assign each point a label.
Class Diagram: A UML document format that portrays classes with graphs, describing their instance variables, method headers and relationships with other classes.
Clustering: A kind of AI algorithm which learns to assign labels to instances based on the distribution of the data points.
CSS (Cascading Style Sheet): A stylesheet language written in either HTML or XML which governs the appearance of the application.
Framework: A collection of classes and interfaces that provide a service for building the application.
GUI (Graphical User Interface): Visual controls within a window of a software application that allows the user to interact with the program.
IEEE (Institute of Electrical and Electronics Engineers): A professional association for the advancement of technology.
Java: A programming language that uses a virtual machine layer between the Java application and the hardware to provide program portability.

Sequence Diagram: A UML document format specifying how object method interact with each other.
Software Design Description (SDD): A description of a software application written by the software developer, giving an overview to the architecture of the project, directed towards a software development team.
Software Requirement Specification (SRS): A description of the software system to be developed. It describes the functional and non-functional requirements. Includes a set of use cases that detail user interactions that the software is required to provide.
Stylesheet: A text file employed by HTML pages that control the appearance and 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 specifying 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

DataViLiJ SRS- Software Requirements Specification written by Ritwik Banerjee and Professaur Inc.™ for the DataViLiJ application.

1.5 Overview

This Software Design Description document explains the intended design for the DataViLiJ software application as specified in the DataViLiJ Software Requirements Specification. Section 2 will provide the Package-Level Viewpoint, noting the packages and frameworks to be created. Section 3 will provide the Class-Level Viewpoint by using UML Class Diagrams to explain how classes should be made. Section 4 will provide the Method-Level System Viewpoint, explaining how methods will work with one another. Section 5 provides information on which file structures and formats to use. Section 6 provides a Table of Contents, an Index and references. All UML Diagrams in this SDD document were created with VioletUML.

2 Package-Level Design Viewpoint

As disclosed earlier, this design will cover the entirety of the DataViLiJ application. This application mostly utilizes the Java API for services. In this section, the components and the usage of the Java API will be thoroughly explained.

2.1 DataViLiJ Overview

The DataViLiJ will be designed with minor edits to the DataViLiJ Framework and PropertiesManager. Figure 2.1 specifies all the components to be developed and places all classes in home packages.

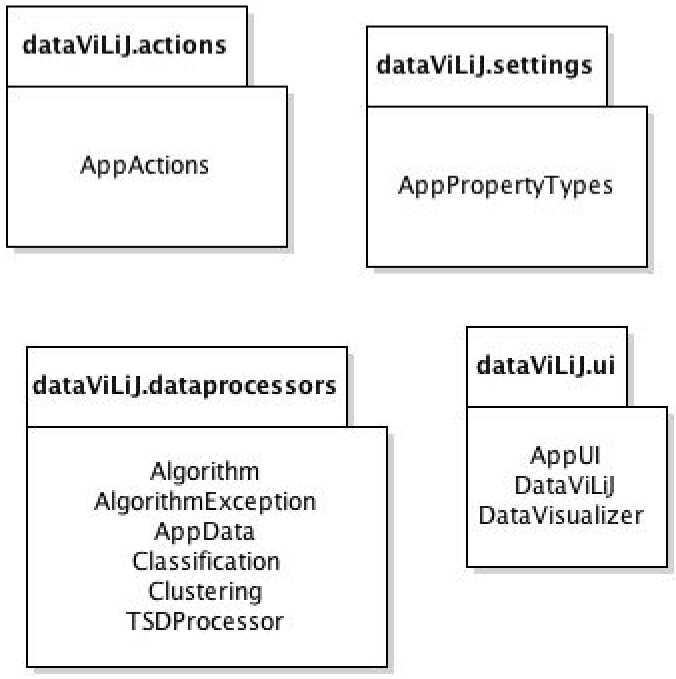


Figure 2.1: DataViLiJ Package Overview
The above figure shows the packages with their respective classes to be used in the DataViLiJ application.

2.2 Java API Usage

The DataViLiJ application will be developed using Java programming. Thus, the design will be composed of the following classes depicted in Figure 2.2

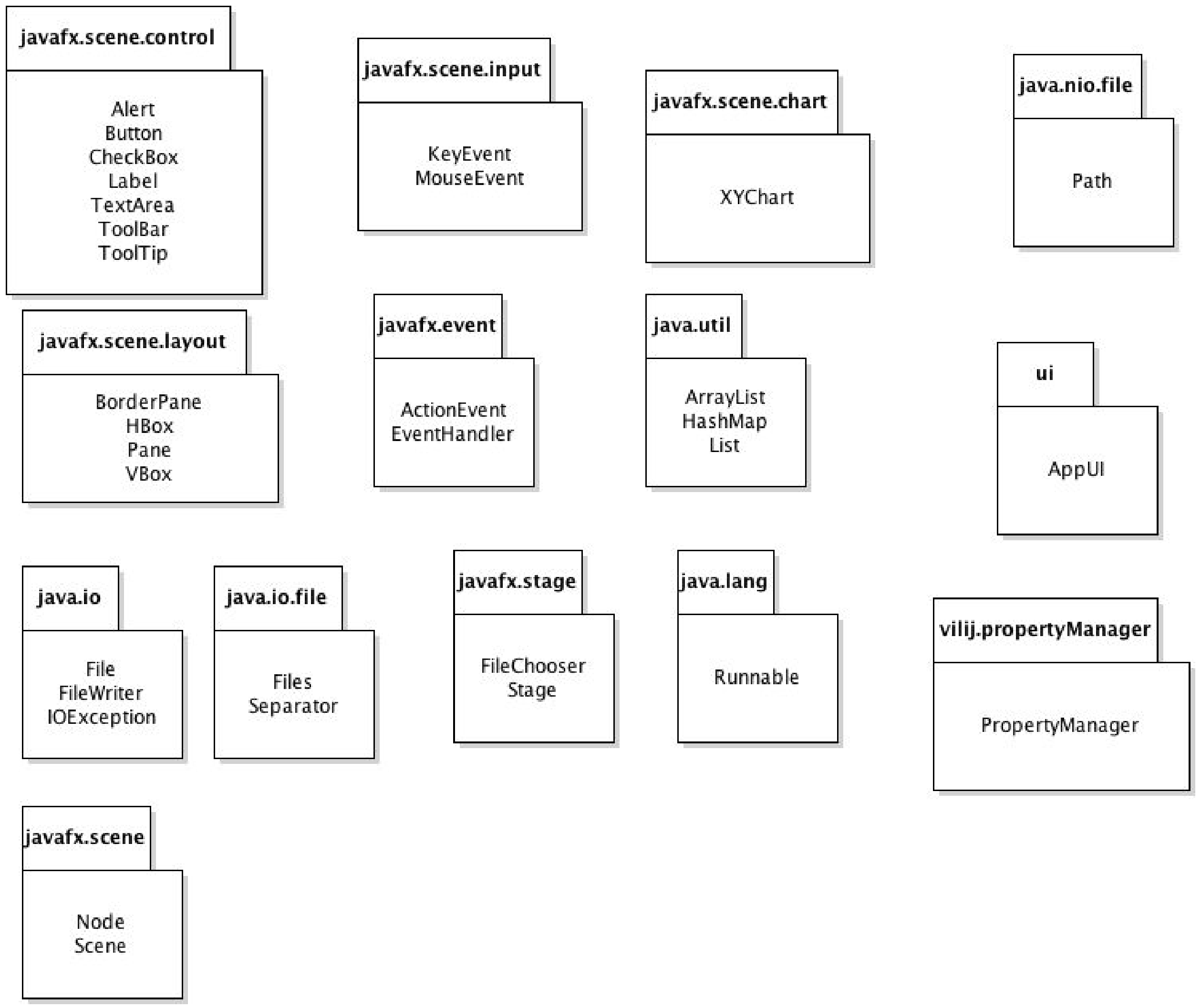


Figure 2.2: Java API Classes and Packages
Above depicts all the major classes and packages to be used in the DataViLiJ application.

2.3 Java API Usage descriptions

Tables 2.1 - 2.7 indicate and define the classes from Figure 2.2.

Class/Interface	Use
Alert	For informing the user in instances of errors or changes.
Button	For adding/removing stations, images, labels and elements. Used for

	composing the toolbar.
CheckBox	To enable/disable the textarea, changing it from read-only and editable version.
Label	For identifying sections of the primary window (Data File, Data Visualization and Read-Only).
TextArea	For taking the user input to be loaded and processed.
Toolbar	To encapsulate the buttons that interact with the application
ToolTip	For showing information when a node is hovered over by mouse

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

Class/Interface	Use
BorderPane	Lays out the workspace
HBox	Lays out panes from left to right
Pane	For the canvas canter of the workspace
VBox	Lays out panes from top to bottom

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

Class/Interface	Use
File	For finding the path to a particular external file or saving a file with a particular path
FileWriter	A convenience class for writing character files
IOException	Catches errors from incorrect saving/loading of a file

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

Class/Interface	Use
Node	For the purpose of generics and the calling of shapes

Scene	Used in the control over the application window

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

Class/Interface	Use
ActionEvent	Gets information about action events
EventHandler	Specifying the type of response to occur when an action occurs

Table 2.5: Uses the class in the Java API’s javafx.event

Class/Interface	Use
FileChooser	Choose the file to load, save or choose an image from.
Stage	Contains the application display

Table 2.6: Uses the class in the Java API’s javafx.stage

Class/Interface	Use
ArrayList	For storing objects, such as Integers
HashMap	For storing pairs of names and values
List	A general term for types of lists for storing objects, such as Integers, Doubles, etc.

Table 2.7: Uses the class in the Java API’s java.util

Class/Interface	Use
Runnable	Used to differentiate threads from the main thread

Table 2.8: Uses the classes in java.lang

3 Class-level Design Viewpoint

This design will include the DataViLiJ application. The UML Class Diagrams shown below will reflect this. Due to the complexity of the design, the class designs will be shown in segments, allowing for details to be explained.

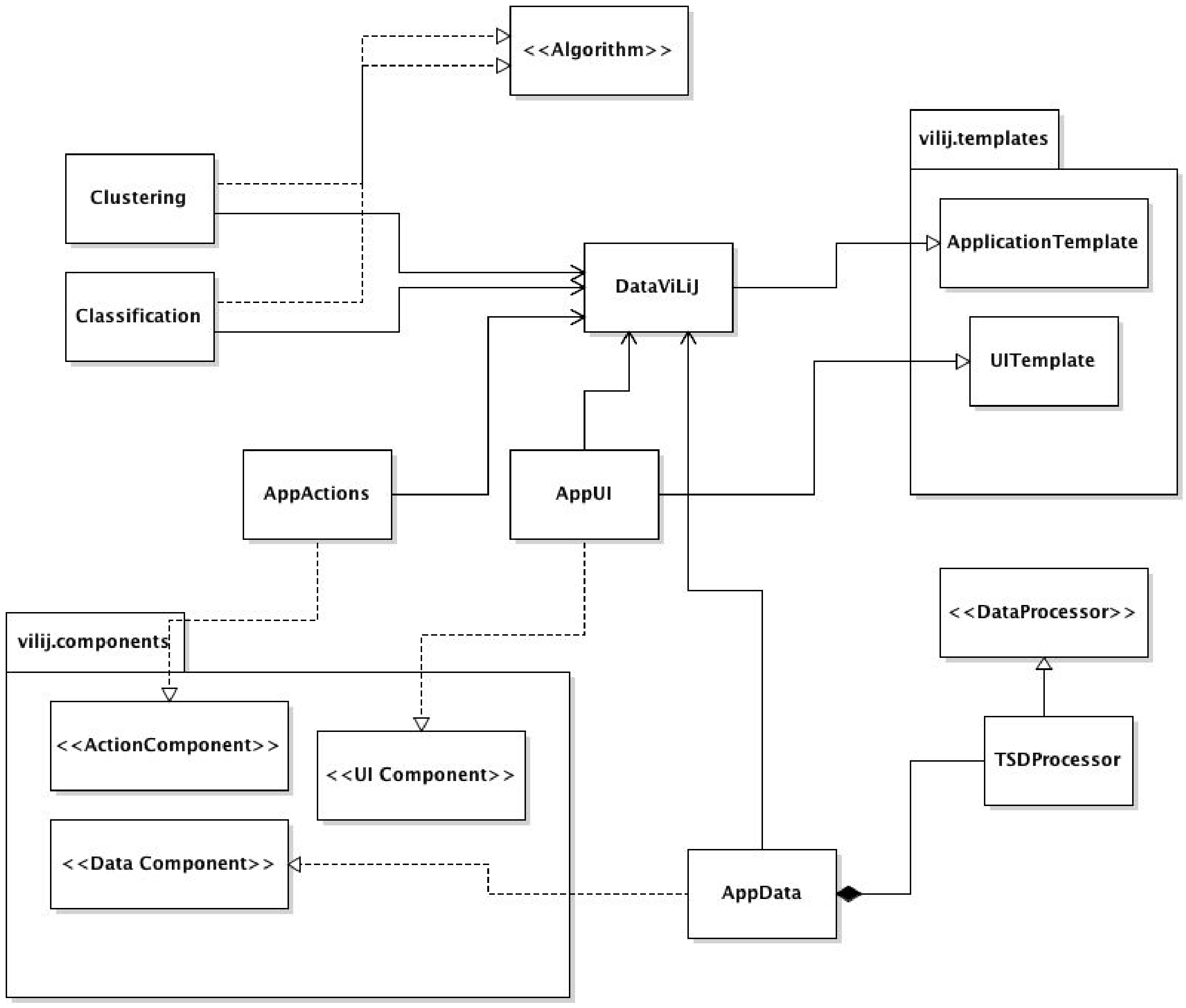


Figure 3.1: Class Structure Level Overview of the DataViLiJ Application
Details the basic outline of the DataViLiJ application's classes. Note that not all of the classes used in the application are indicated in this diagram. These classes will be explored in later diagrams, as this particular diagram is for the purpose of a general understanding of the application.

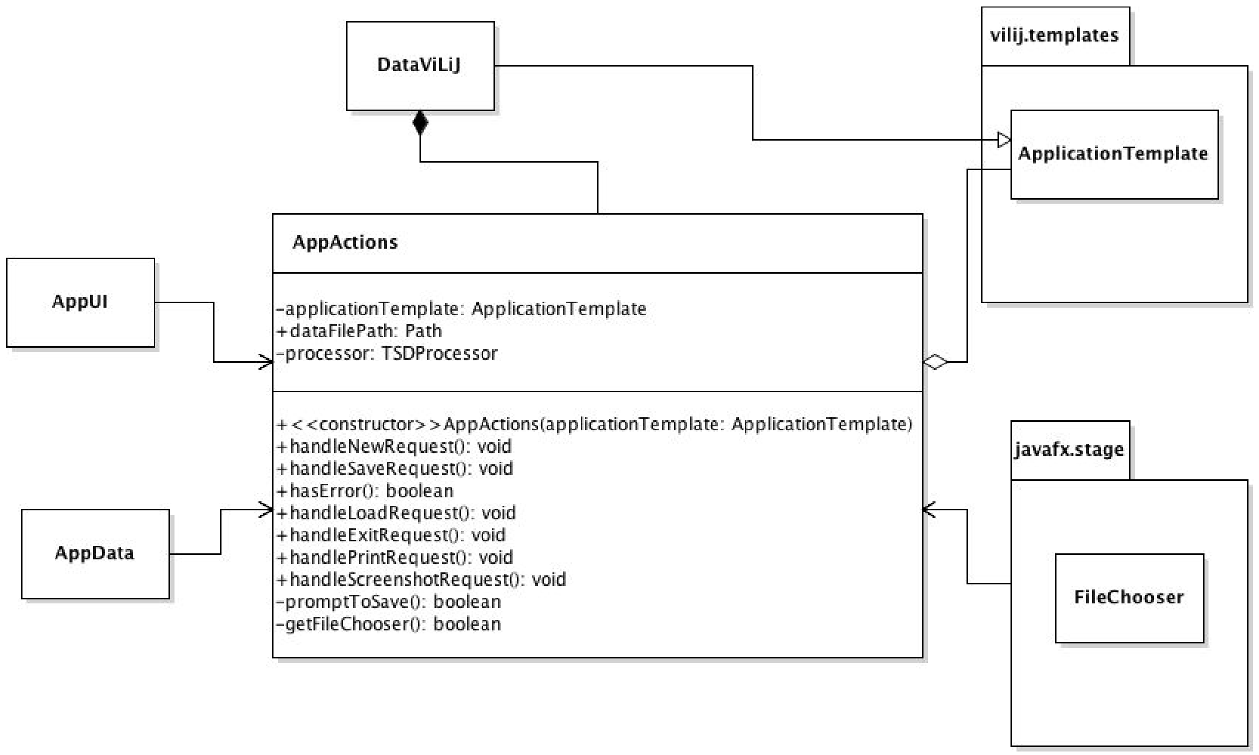

DataViLiJ
vilij.templates
ApplicationTemplate
AppActions
AppUI
-applicationTemplate: ApplicationTemplate
+dataFilePath: Path
-processor: TSDProcessor
+<<constructor>>AppActions(applicationTemplate: ApplicationTemplate)
+handleNewRequest(): void
+handleSaveRequest(): void
+hasError(): boolean
+handleLoadRequest(): void
+handleExitRequest(): void
+handlePrintRequest(): void
+handleScreenshotRequest(): void
-promptToSave(): boolean
-getFileChooser(): boolean
AppData
javafx.stage
FileChooser

Figure 3.3 Overview of AppActions Class

The AppActions class works together with other classes in order to perform its functionalities. Its major responsibility would be handling the user's interactions with the buttons. This must be done through lambda expressions and action listeners.

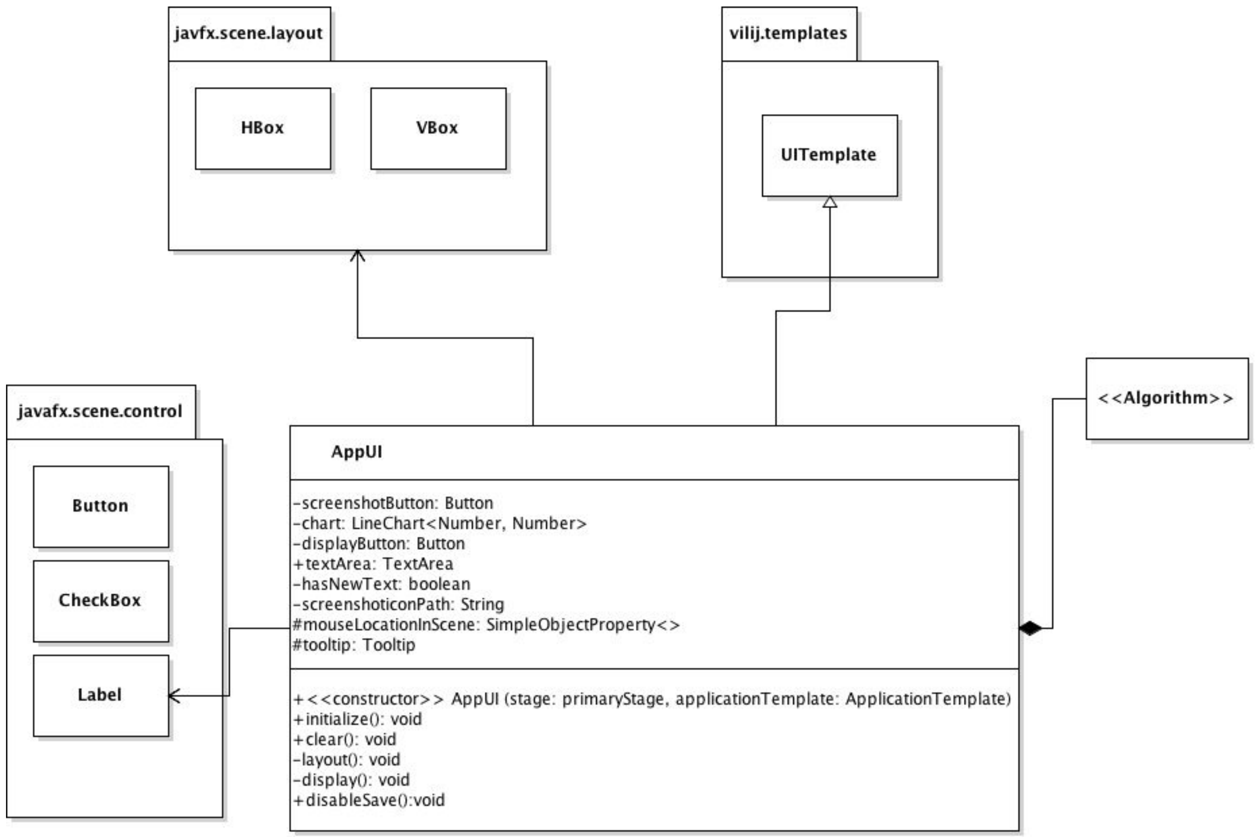


Figure 3.4 Overview of AppUI Class
This figure depicts the AppUI class and its relationship to other classes. The main functionality of this class is to set up the display and layout of the application.

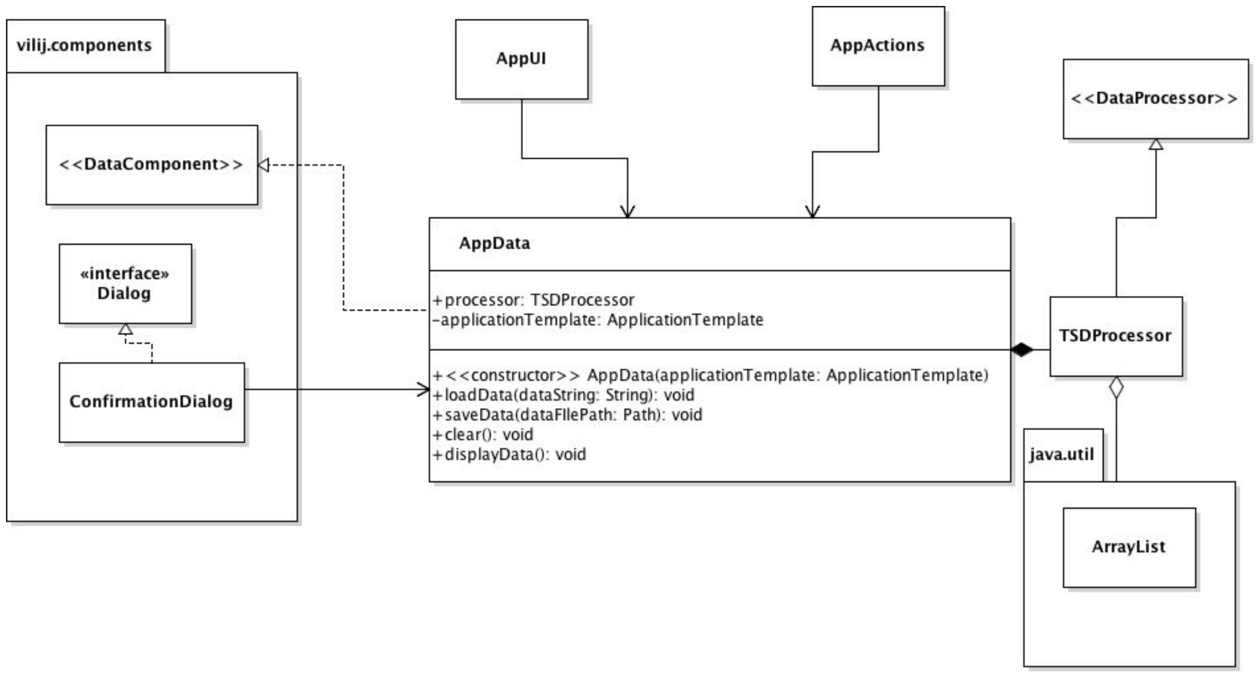


Figure 3.5 Overview of AppData Class
A more detailed look at the AppData class. This class works closely with the TSDProcessor in order to load and assist saving data.

4 Method-level Design Viewpoint

The following figures in this section pertain to the methods within the classes that compose DataViLiJ. Each will explain how the ten use cases from the DataViLiJ SRS is meant to be implemented.

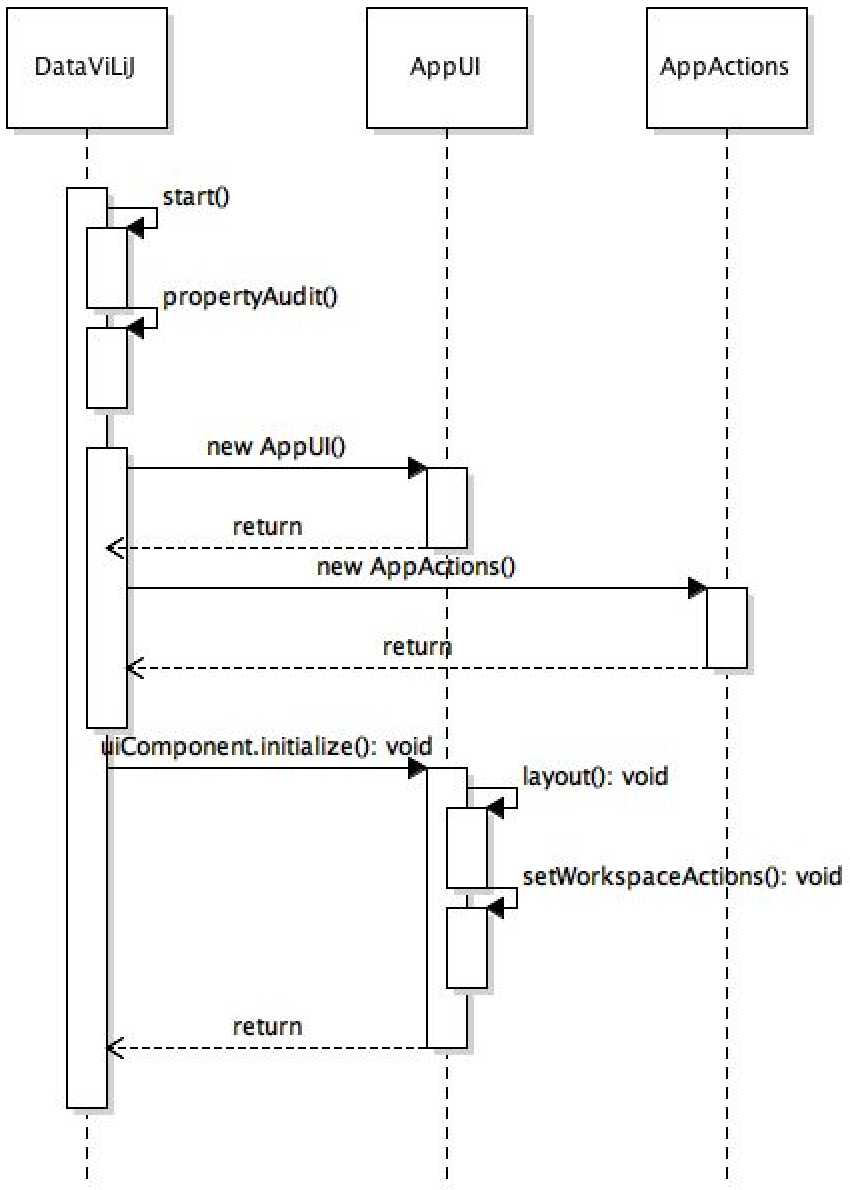


Figure 4.1: Overview of Application Start
The start of the application will prompt the layout of the primary window to be set up, as well as set up the user interface buttons by setting up the workspace.

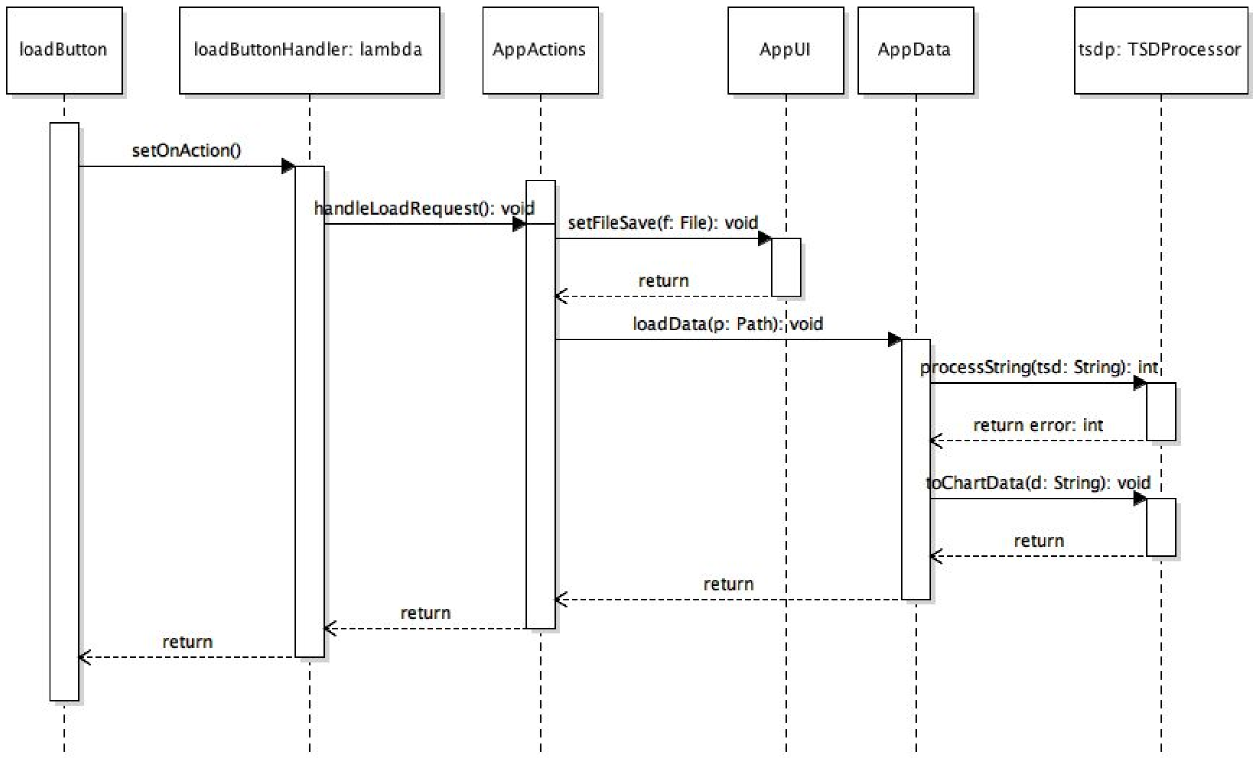

loadButton
loadButtonHandler: lambda
AppActions
AppUI
AppData
tsdp: TSDProcessor
setOnAction()
handleLoadRequest(): void
setFileSave(f: File): void
return
loadData(p: Path): void
processString(tsd: String): int
return error: int
toChartData(d: String): void
return
return
return
return

Figure 4.2: Overview of Load Button
The above figure depicts the functionality of the load button. If there is no save path currently available, a path will be set. Then the data is loaded with the path. The string text is processed and loaded into the chart.

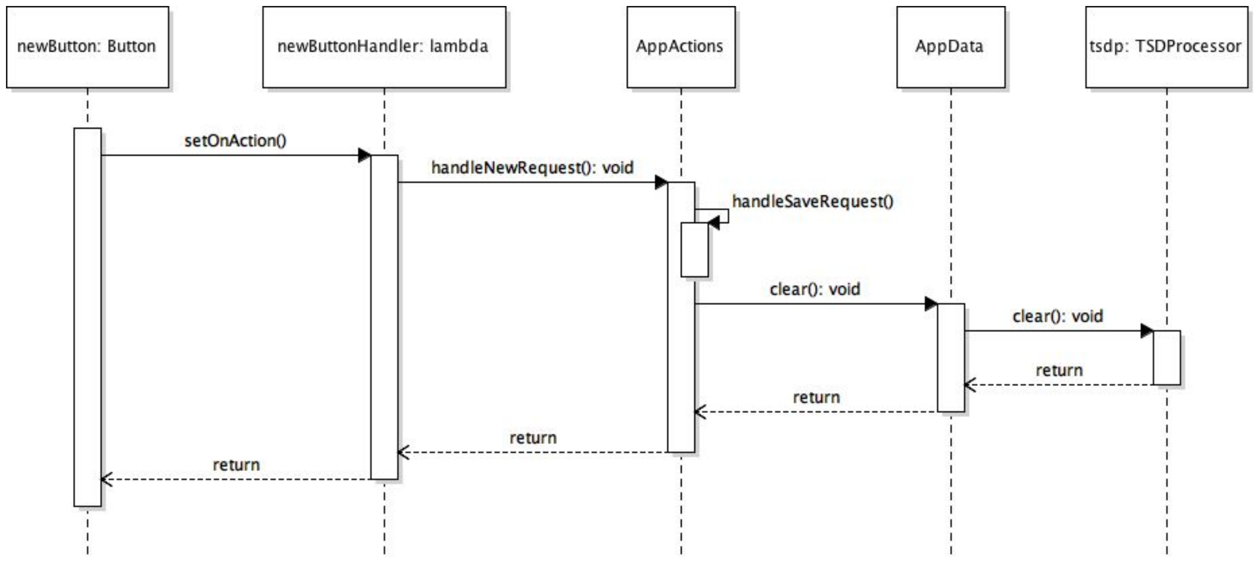


Figure 4.3: Overview of New Button
AppActions will process the new request. This in turn prompts the save request if there has been new data added to the text area. After this, the text area and chart are cleared.

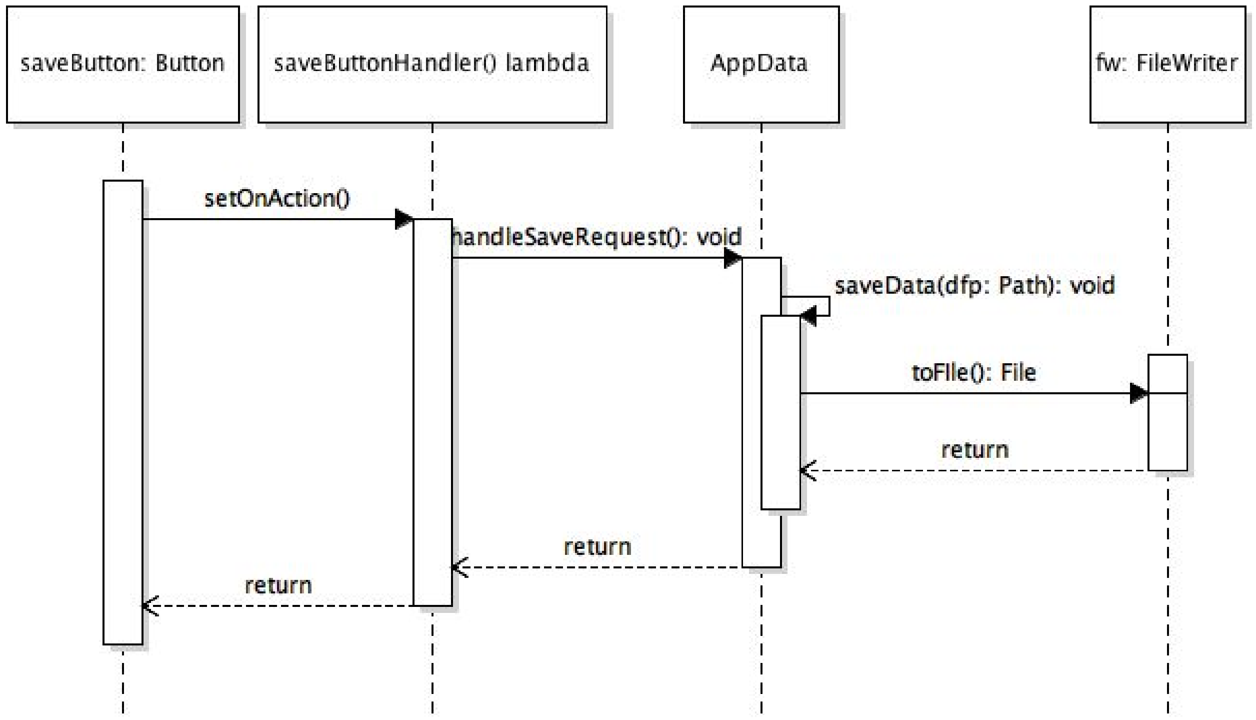


Figure 4.4: Overview of Save Button
If the save button is clicked and the save request has been handled, AppData will save the data by retrieving the data file path to the file.

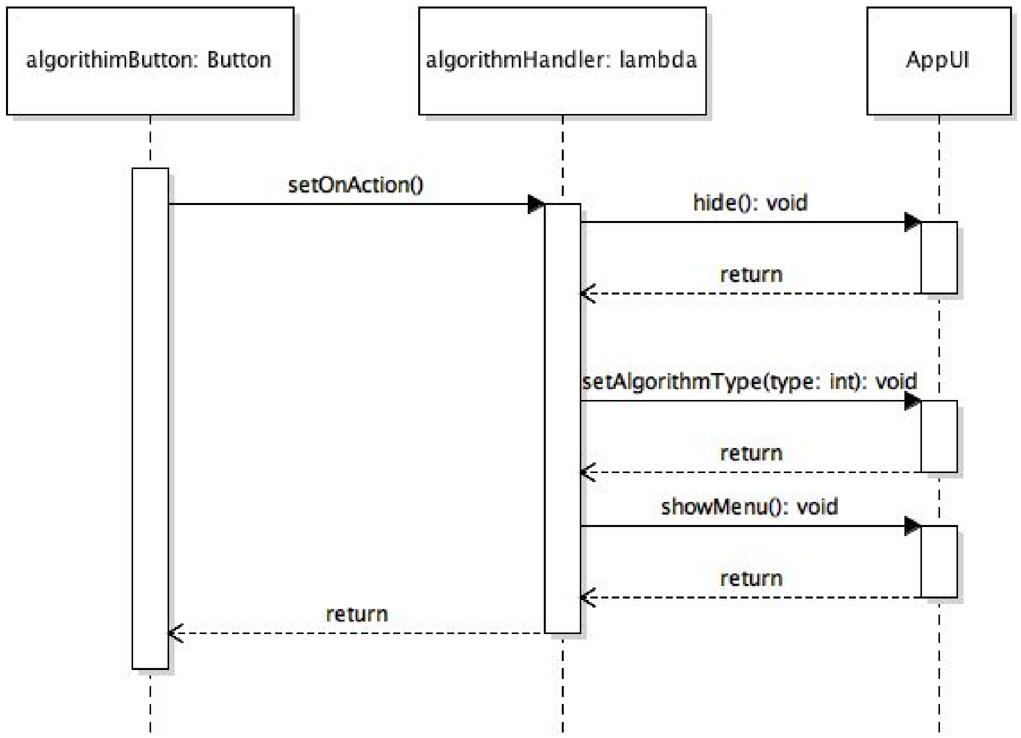


Figure 4.5: Overview of Setting Algorithm Type
If the null/Classification/Clustering button is clicked, the algorithm type will be switched. The types of algorithms will be displayed on a menu.

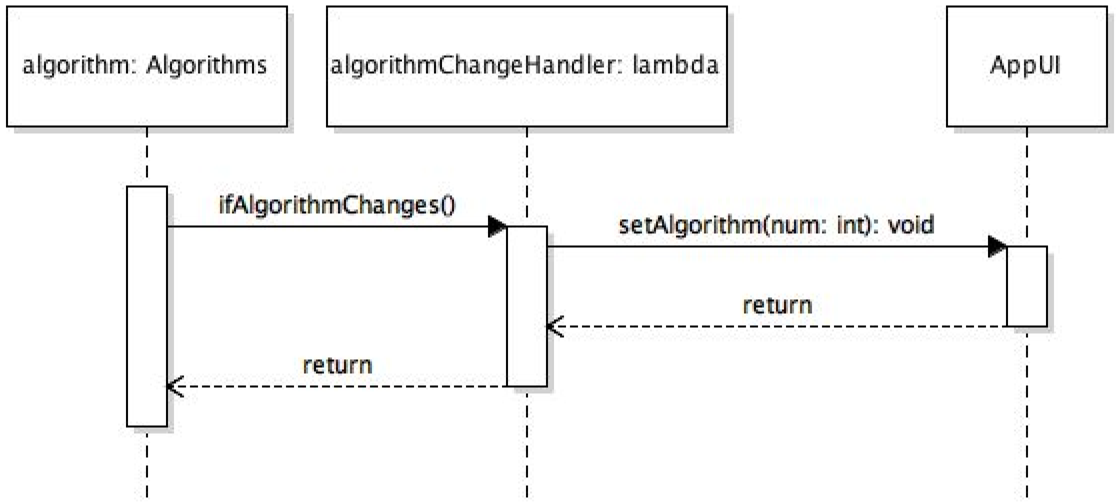


Figure 4.6: Overview of Setting Algorithm
Algorithms has 3 different CheckBoxes that correspond to integer values 0, 1, and 2. These will be responsible between setting the algorithm to null, classification or clustering. When a button is changed, the AppUI will be updated.

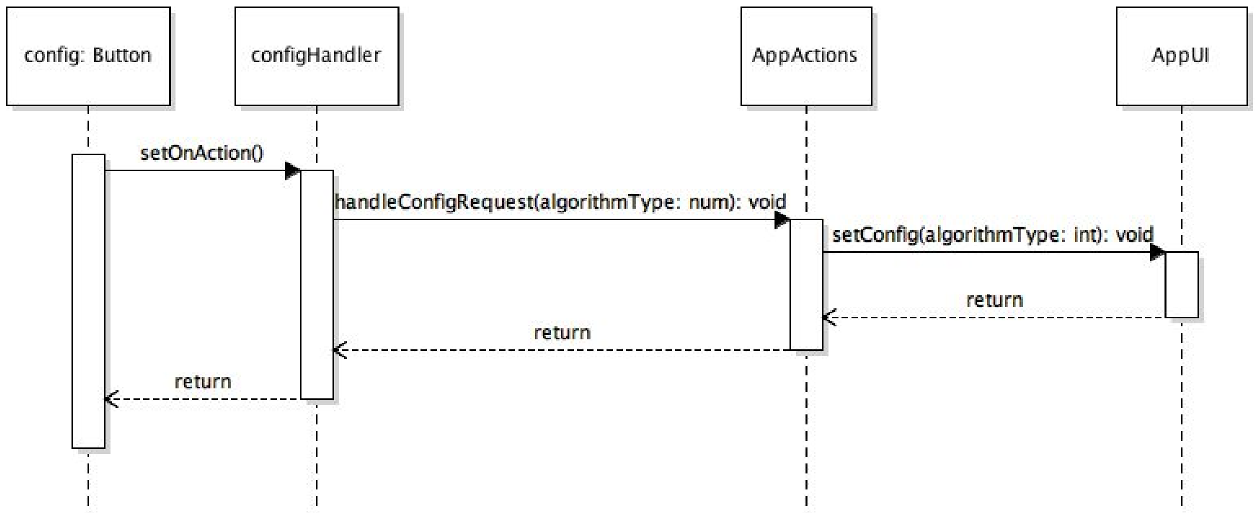

config: Button
configHandler
AppActions
AppUI
setOnAction()
handleConfigRequest(algorithmType: num): void
setConfig(algorithmType: int): void
return
return
return

Figure 4.7: Overview of Select Algorithm Run Configuration
The above figure shows the functionality of the configuration buttons, which set the configuration of the run based on the number of the algorithm passed.

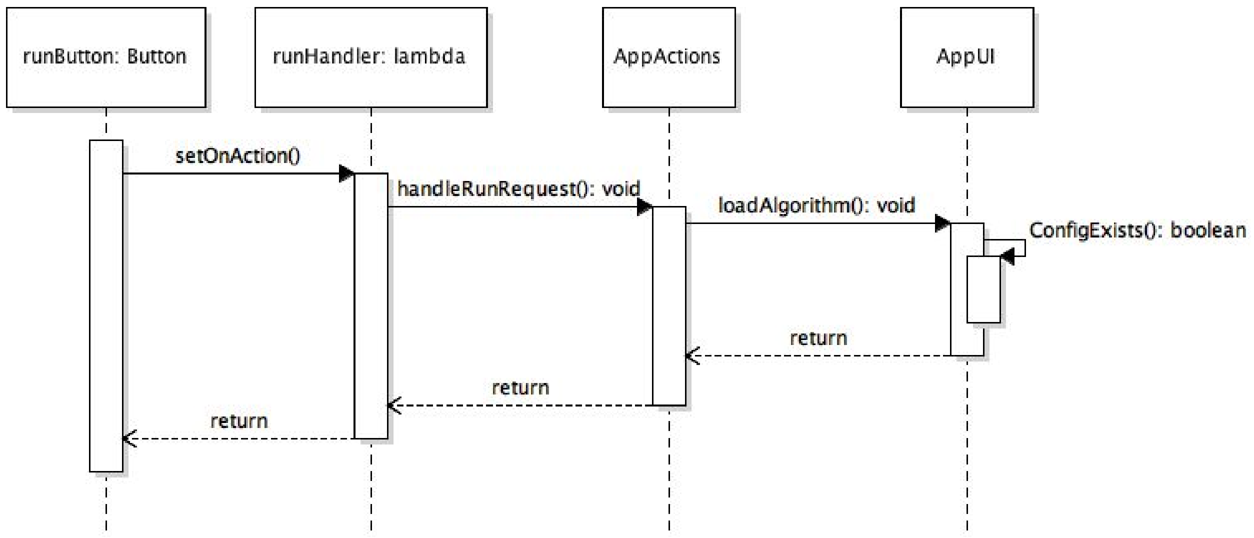


Figure 4.8: Overview of Run Button
When the run button is clicked, AppActions handles the run request. Once the type of algorithm is loaded and selected, the AppUI will run a check on if the configuration is valid.

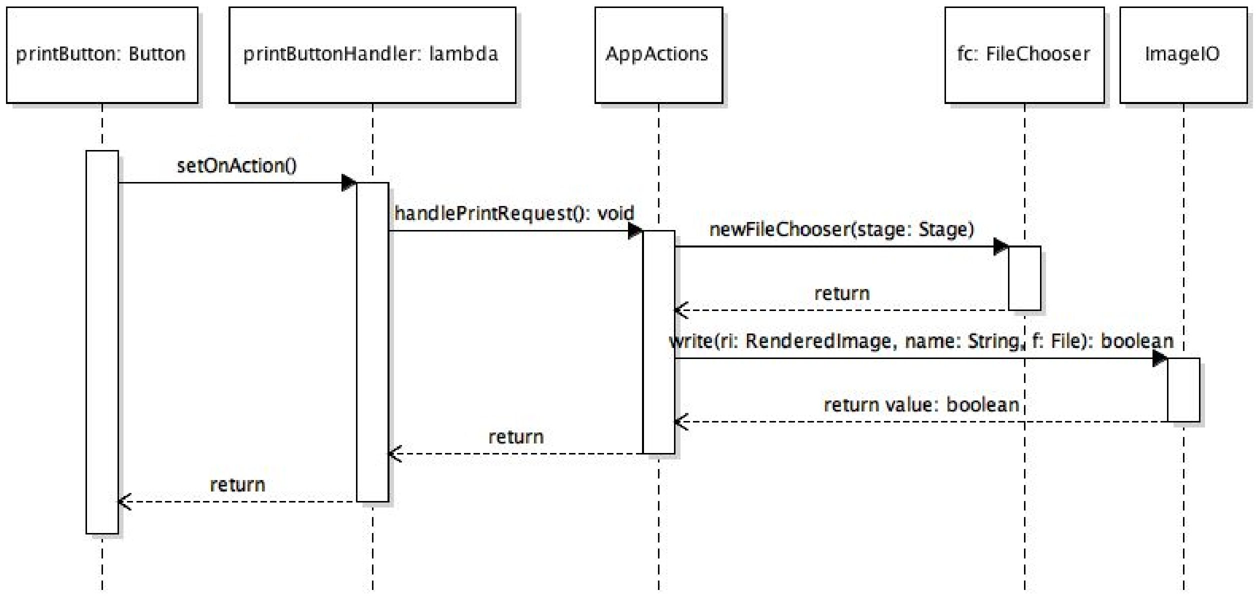

printButton: Button
printButtonHandler: lambda
AppActions
fc: FileChooser
ImageIO
setOnAction()
handlePrintRequest(): void
newFileChooser(stage: Stage)
return
write(ri: RenderedImage, name: String, f: File): boolean
return value: boolean
return
return

Figure 4.9: Export Data Visualization as Image
The above figure depicts the functionality of the print mechanism. After the print request has been handled, AppActions creates an instance of FileChooser to print.

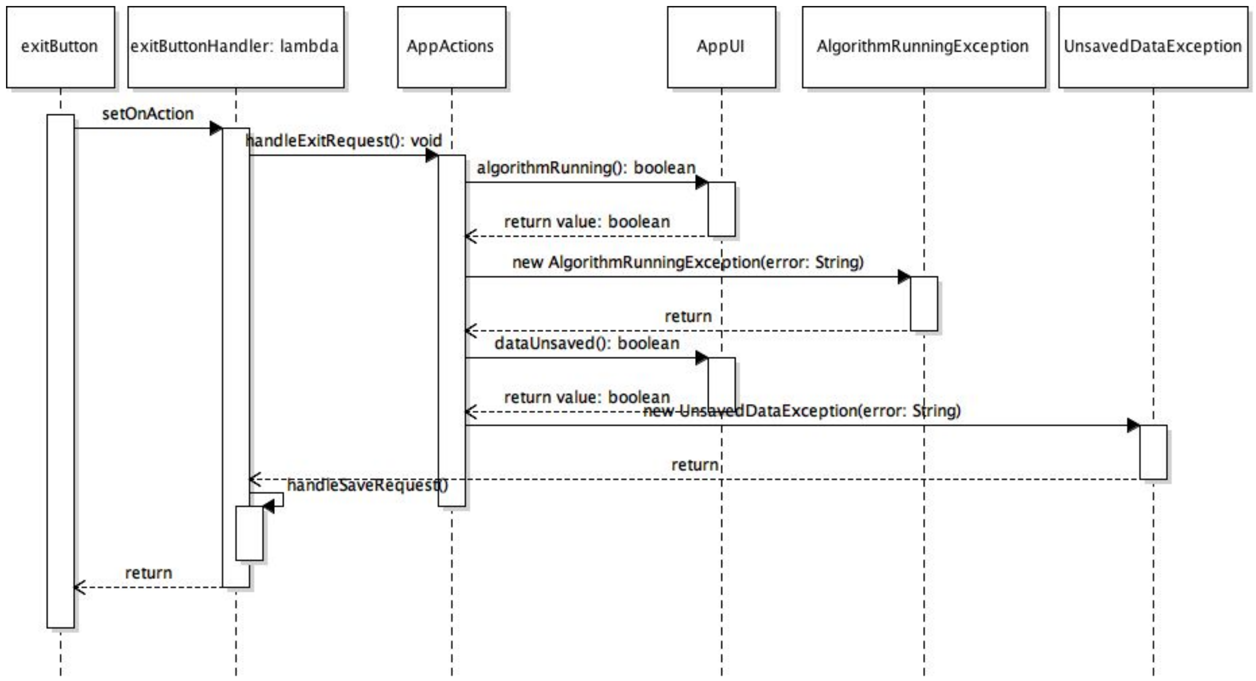
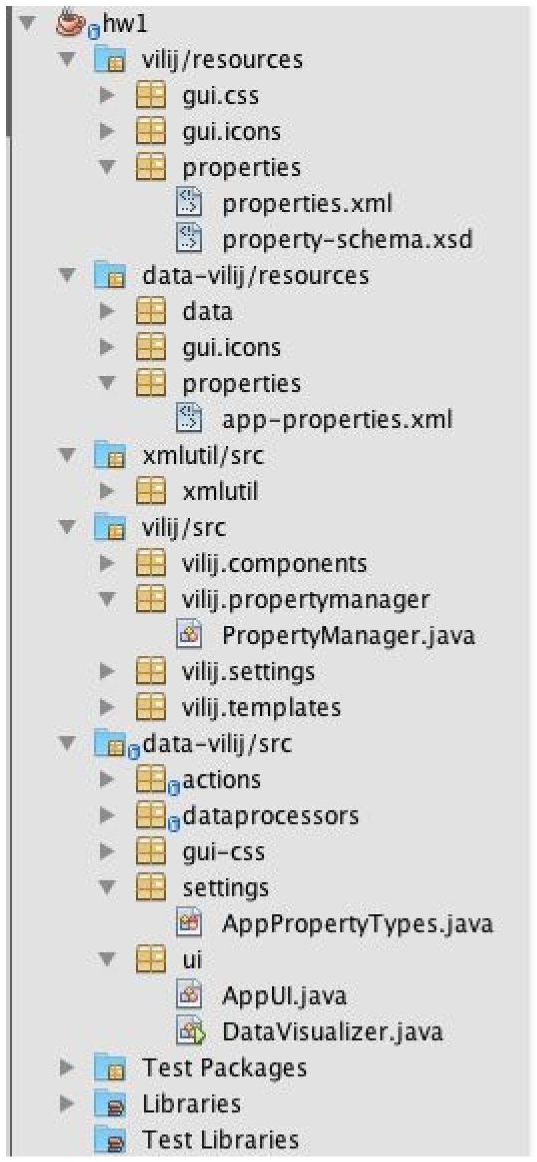

exitButton
exitButtonHandler: lambda
AppActions
AppUI
AlgorithmRunningException
UnsavedDataException
setOnAction
handleExitRequest(): void
algorithmRunning(): boolean
return value: boolean
new AlgorithmRunningException(error: String)
return
dataUnsaved(): boolean
return value: boolean
new UnsavedDataException(error: String)
return
handleSaveRequest()
return

Figure 4.10
The exit button will cause an exception to be thrown if an algorithm is still running. Likewise, an exception will be thrown if the data hasn't been saved and a save prompt will be shown.

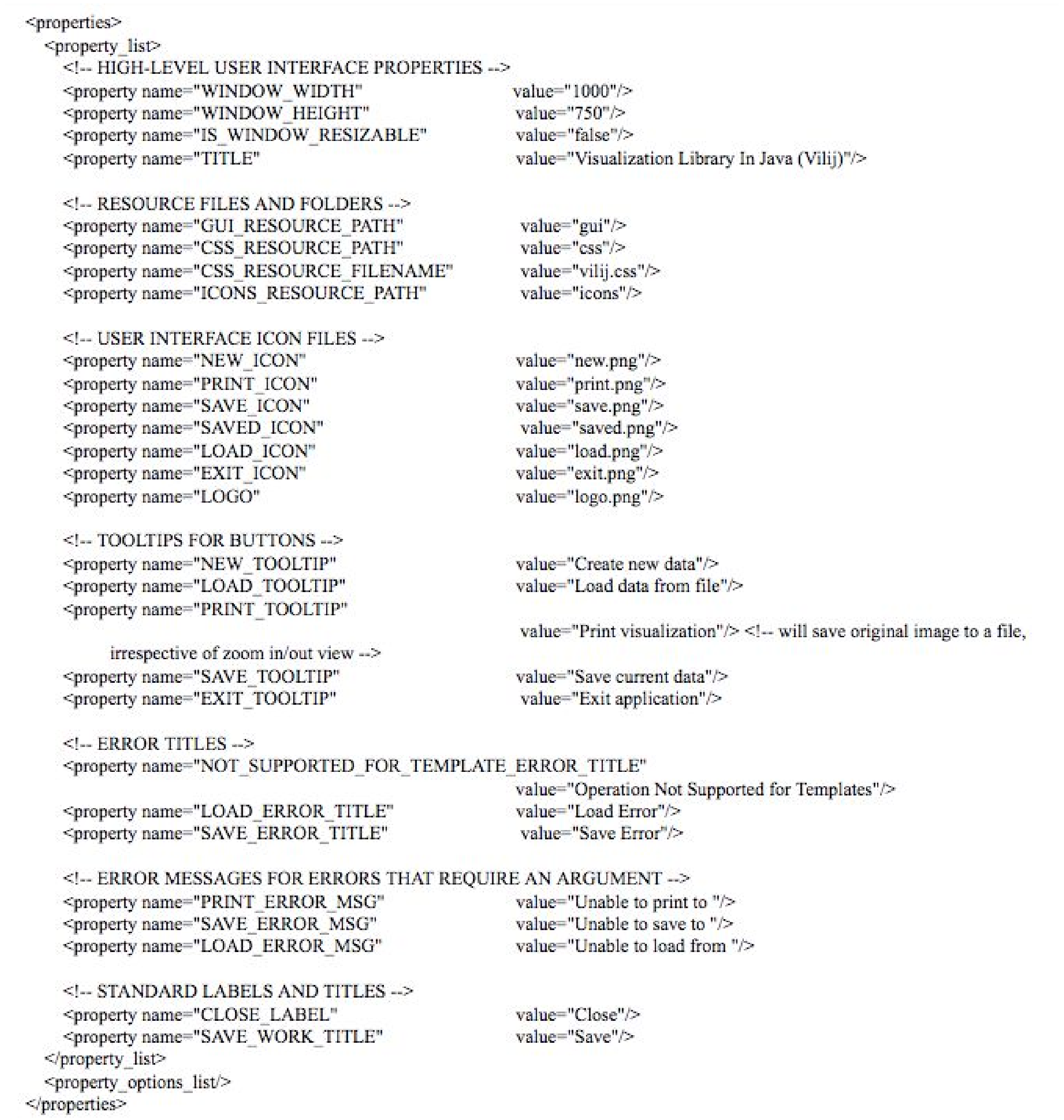
5 File/Data structures and Formats

Note that the DataViLiJFramework and PropertiesManager will be provided in the DataViLiJFramework.jar and PropertiesManager.jar Java Archive files. This should be imported into the project for the DataViLiJ application and included in the deployment of one executable JAR file, DataViLiJ.jar. Note that all necessary data and art files must be included in the program. Figure 5.1 specifies the required file structure the application should be using. All necessary images should go in the image directory



```
hw1
    vilij/resources
        gui.css
        gui.icons
        properties
            properties.xml
            property-schema.xsd
    data-vilij/resources
        data
        gui.icons
        properties
            app-properties.xml
    xmlutil/src
        xmlutil
    vilij/src
        vilij.components
        vilij.propertymanager
            PropertyManager.java
        vilij.settings
        vilij.templates
    data-vilij/src
        actions
        dataprocessors
        gui-css
        settings
            AppPropertyTypes.java
        ui
            AppUI.java
            DataVisualizer.java
    Test Packages
    Libraries
    Test Libraries
```

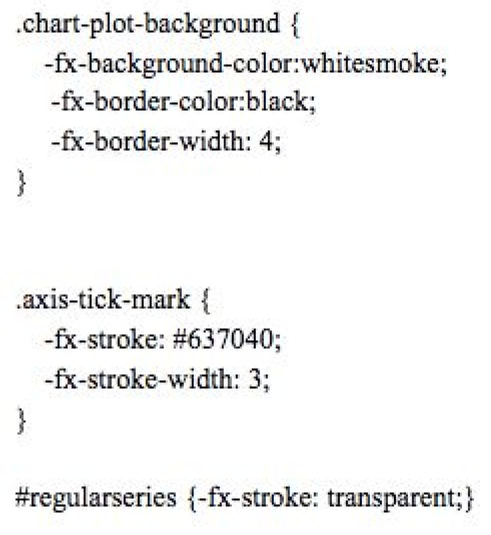
Figure 5.1: DataViLiJ program and File Structure
This shows the expected structure of the packages and class files to be used in the DataViLiJ application. Not all classes are shown in order to fit the image to page length.



```
<properties>
  <property_list>
    <!-- HIGH-LEVEL USER INTERFACE PROPERTIES -->
    <property name="WINDOW_WIDTH"                value="1000"/>
    <property name="WINDOW_HEIGHT"               value="750"/>
    <property name="IS_WINDOW_RESIZABLE"         value="false"/>
    <property name="TITLE"                       value="Visualization Library In Java (Vilij)"/>

    <!-- RESOURCE FILES AND FOLDERS -->
    <property name="GUI_RESOURCE_PATH"           value="gui"/>
    <property name="CSS_RESOURCE_PATH"           value="css"/>
    <property name="CSS_RESOURCE_FILENAME"       value="vilij.css"/>
    <property name="ICONS_RESOURCE_PATH"         value="icons"/>

    <!-- USER INTERFACE ICON FILES -->
    <property name="NEW_ICON"                    value="new.png"/>
    <property name="PRINT_ICON"                  value="print.png"/>
    <property name="SAVE_ICON"                   value="save.png"/>
    <property name="SAVED_ICON"                  value="saved.png"/>
    <property name="LOAD_ICON"                   value="load.png"/>
    <property name="EXIT_ICON"                   value="exit.png"/>
    <property name="LOGO"                        value="logo.png"/>

    <!-- TOOLTIPS FOR BUTTONS -->
    <property name="NEW_TOOLTIP"                 value="Create new data"/>
    <property name="LOAD_TOOLTIP"                value="Load data from file"/>
    <property name="PRINT_TOOLTIP"
                                                 value="Print visualization"/> <!-- will save original image to a file,
        irrespective of zoom in/out view -->
    <property name="SAVE_TOOLTIP"                value="Save current data"/>
    <property name="EXIT_TOOLTIP"                value="Exit application"/>

    <!-- ERROR TITLES -->
    <property name="NOT_SUPPORTED_FOR_TEMPLATE_ERROR_TITLE"
                                                 value="Operation Not Supported for Templates"/>
    <property name="LOAD_ERROR_TITLE"            value="Load Error"/>
    <property name="SAVE_ERROR_TITLE"            value="Save Error"/>

    <!-- ERROR MESSAGES FOR ERRORS THAT REQUIRE AN ARGUMENT -->
    <property name="PRINT_ERROR_MSG"             value="Unable to print to "/>
    <property name="SAVE_ERROR_MSG"              value="Unable to save to "/>
    <property name="LOAD_ERROR_MSG"              value="Unable to load from "/>

    <!-- STANDARD LABELS AND TITLES -->
    <property name="CLOSE_LABEL"                 value="Close"/>
    <property name="SAVE_WORK_TITLE"             value="Save"/>
  </property_list>
  <property_options_list/>
</properties>
```

Figure 5.2: properties.xml format

This is how an xml file should be formatted. Not all xml files used within DataViLiJ are depicted in this figure. The file contains the interface properties, resource files/folders, icon files, tooltips for buttons, error titles/messages and standard labels/titles.



```
.chart-plot-background {
   -fx-background-color:whitesmoke;
    -fx-border-color:black;
    -fx-border-width: 4;
}


.axis-tick-mark {
   -fx-stroke: #637040;
   -fx-stroke-width: 3;
}

#regularseries {-fx-stroke: transparent;}
```

Figure 5.3: .css format of chart object
Above is the expected format of css files in the DataViLiJ application. Note that this is not all of the required css code for the program. This figure is meant to only demonstrate how the format should appear.

6 Supporting Information

Since this document is a reference for those implementing the code, a table of contents will be provided at the beginning and end of this SDD to help find important sections. For any additional information needed on the DataViLiJ application, utilize the DataViLiJ SRS.

Table of Contents