DATAVILIJ™

Software Design Description

Author: **Yichun (William) Wu**
Professaur Inc. ™
March, 2018
Version 1.0
Abstraction: **This document describes the software design for DataViLiJ, a significantly enhanced data visualization application.**

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

1. Table of Contents

2. Introduction

Given the increasing importance of data-driven artificial intelligence (AI) in many aspects of computer science, visualizing how AI algorithms work is becoming increasingly important. Java is among the most important programming languages used to implement these algorithms, but it lacks standard data visualization libraries (unlike some other languages such as Python). Moreover, all existing libraries are meant to show us the final output of the data science algorithms. They are not designed for visualizing the changes that happen while the algorithms are running and updating the data. In other words, the visualization libraries do not help us see how these algorithms learn from the data.

DataViLiJ (Data Visualization Library in Java) will be a desktop application that will allow users to select an algorithm (from a set of standard AI algorithms) and dynamically show the user what changes, and how.

2.1 Purpose

The purpose of this document is to specify how the DataViLiJ application should look and operate. This document serves as an agreement among all parties and as a reference for how the data visualization application should ultimately be constructed. Upon reading of this document, one should clearly understand the visual aesthetics and functionalities of application's user interface as well as understand the way AI algorithms are incorporated.

2.2 Scope

The goal of this project is for students and beginning professionals in AI to have a visual understanding of the inner workings of the fundamental algorithms. AI is a vast field, and this project is limited to the visualization of two types of algorithms that "learn" from data. These two types are called **clustering** and **classification**. The design and development of these algorithms is outside the scope of the project, and the assumption is that such algorithms will already be developed independently, and their output will comply with the data format specified in this document. DataViLiJ serves simply as a visualization tool for how those algorithms work. Both clustering and classification are, in theory, not limited to a fixed number of labels for the data, but this project will be limited to at most four labels for clustering algorithms, and exactly two labels for classification algorithms. Further, the design and development of this project will also assume that the data is 2-dimensional. As such, 3D visualization is currently beyond the scope of DataViLiJ.

As for the GUI interactions, touch screen capabilities are not within the scope of this application.

2.3 Definitions, acronyms, and abbreviations

Algorithm: In this document, the term 'algorithm' will be used to denote an AI algorithm that can "learn" from some data and assign each data point a label.

Clustering: A type of AI algorithm that learns to assign labels to instances based purely on the spatial distribution of the data points.

Classification: A type of AI algorithm that learns to assign new labels to instances based on how older instances were labeled. These algorithms calculate geometric objects that divide the x-y plane into parts. E.g., if the geometric object is a circle, the two parts are the inside and the outside of that circle; if the geometric object is a straight-line, then again, there two parts, one on each side of the line.

Framework: An abstraction in which software providing generic functionality for a broad and common need can be selectively refined by additional user-written code, thus enabling the development of specific applications, or even additional frameworks. In an object-oriented environment, a framework consists of interfaces and abstract and concrete classes.

Graphical User Interface (GUI): An interface that allows users to interact with the application through visual indicators and controls. A GUI has a less intense learning curve for the user, compared to text-based command line interfaces. Typical controls and indicators include buttons, menus, check boxes, dialogs, etc.

IEEE: Institute of Electrical and Electronics Engineers, is a professional association founded in 1963. Its objectives are the educational and technical advancement of electrical and electronic engineering, telecommunications, computer engineering and allied disciplines.

Instance: A 2-dimensional data point comprising a x-value and a y-value. An instance always has a name, which serves as its unique identifier, but it may be labeled or unlabeled.

Unified Modeling Language (UML): A general-purpose, developmental modeling language to provide a standard way to visualize the design of a system.

Use Case Diagram: A UML format that represents the user's interaction with the system and shows the relationship between the user and the different use cases in which the user is involved.

User: Someone who interacts with the DataViLiJ application via its GUI.

User Interface (UI): See *Graphical User Interface (GUI)*.

2.4 Reference

DataViLiJ™ Software Requirement Specification - Professaur's Software Requirement Specification for the DataViLiJ.

IEEE Std 830™ -1998 (R2009) - IEEE Recommended Practice for Software Requirement Specification.

2.5 Overview

Imagine you are a traveler who has never tasted a certain exotic fruit X, and upon reaching a far-away tropical island, see this fruit for the first time. You don't know how to pick the tasty ones, of course. But based on your experience with other fruits, you know that the color and softness are good indicators of taste. So, you start tasting randomly, and gradually, you figure out what color and softness you want when you go shopping for this fruit. AI algorithms work in a similar way. If the color and softness are somehow both represented in the [0,1] interval, then every instance of X can become a 2-dimensional data point. If the instance is known to be tasty, then it can have a label called "tasty" (or "somewhat tasty", "not tasty", etc.). Based on such a representation of the fruits, the algorithms can learn. Then, even if you throw a new fruit at the algorithm, it can figure out whether or not it is likely to be tasty. As mentioned in Sec. 1.3, such algorithms are beyond the scope of this project. As far as we are concerned, any such algorithm is iterative (the iteration being the machine's equivalent of a human tasting one fruit at a time and getting a better understanding with more experience).

Now imagine you are a student of AI who wants to understand how these algorithms learn. For that, it would be helpful to see how these algorithms manipulate the data they are learning from. This is very similar to how we gain a better understanding of, say, a sorting algorithm, when we see a running animation of that algorithm manipulating its input array of data. With this motivation in mind, the rest of this section will the whole system.

3. Package-level Design Viewpoint

As mentioned, this design will encompass the DataViLiJ application. In building this application, we will heavily rely on the Java API to provide services. Following are descriptions of the components to be built, as well as how the Java API will be used to build them.

3.1 Software overview

The DataViLiJ application will be designed with minor edits to the ViLiJ, including components, propertymanager, settings, and templates.

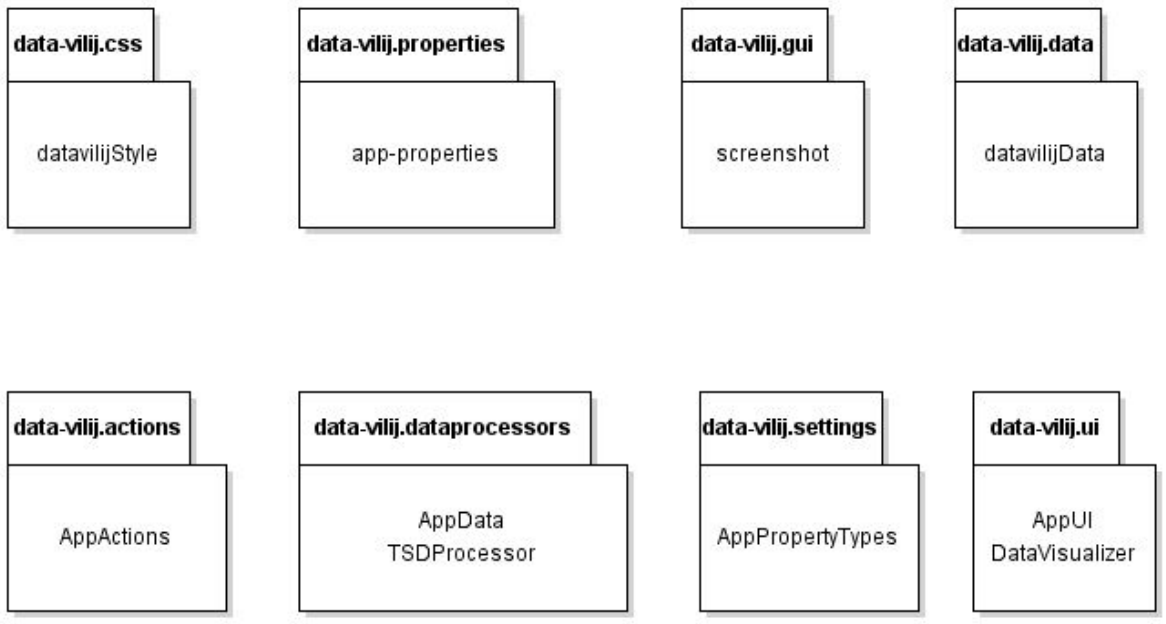

Figure 3.1: DataViLiJ Package Overview

3.2 Java API Usage

The DataViLiJ application will be developed using Java programming languages. As such, this design will make use of the class specified in Figure 3.2.

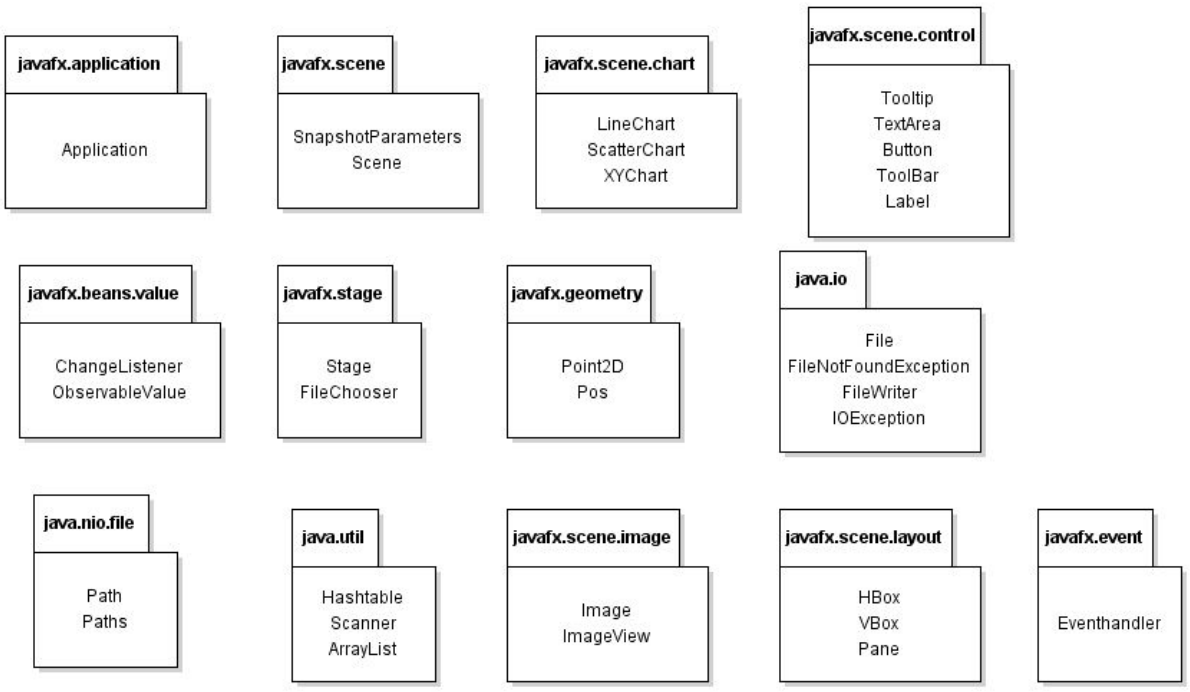

Figure 3.2: Java API classes and packages to be used.

3.3 Java API Usage Description

Table 3.1- 3.13 below summarizes how each of these classes will be used

Class/Interface	Use
Application	**The entry point for launching the application.**

Table 3.1: Uses the classes in the Java API's javafx.application

Class/Interface	Use
SnapshotParameters	**For exporting the chart as image.**
Scene	**For the control over the application window.**

Table 3.2: Uses the classes in the Java API's javafx.scene

Class/Interface	Use
LineChart	**For classification algorithm**
ScatterChart	**For clustering algorithm**
XYChart	**For processing data in the TSDprocessor**

Table 3.3: Uses the classes in the Java API's javafx.scene.chart

Class/Interface	Use
Tooltip	For showing information of a node when it is hovered by mouse.
TextArea	For entering and showing the data.
Button	For loading, saving, creating data, exporting data as image, and exiting the application.
ToolBar	For showing the buttons of this application.
Label	For identifying sections in the application.

Table 3.4: Uses the classes in the Java API's javafx.scene.control

Class/Interface	Use
ChangeListener	For changing the state of a button or the textarea.
ObservableValue	For comparing the changes of values.

Table 3.5: Uses the classes in the Java API's javafx.beans.value

Class/Interface	Use
Stage	For creating a primary window for the application.
FileChooser	For choosing a file from the user's computer.

Table 3.6: Uses the classes in the Java API's javafx.stage

Class/Interface	Use
Point2D	For showing data as points on the chart.
Pos	For organizing the sections of the application.

Table 3.7: Uses the classes in the Java API's javafx.geometry

Class/Interface	Use
File	For loading a file chosen by the user or saving data as a file.
FileNotFoundException	Exception for not finding a file.
FileWriter	For saving the data as a file.
IOException	Exception for writing or reading a file.

Table 3.8: Uses the classes in the Java API's java.io

Class/Interface	Use
Path	For getting the path of a file or saving a file to a specific path.

Table 3.9: Uses the classes in the Java API's java.nio.file

Class/Interface	Use
Hashtable	For processing a group of data.
Scanner	For loading a file.
ArrayList	For storing a list of data.

Table 3.10: Uses the classes in the Java API's java.util

Class/Interface	Use
Image	For saving data as an image.
ImageView	For saving data as an image.

Table 3.11: Uses the classes in the Java API's javafx.scene.image

Class/Interface	Use
HBox	For organizing the sections horizontally.
VBox	For organizing the sections vertically.
Pane	For showing the sections of the application on a window.

Table 3.12: Uses the classes in the Java API's javafx.scene.layout

Class/Interface	Use
Eventhandler	For handling an event by buttons.

Table 3.13: Uses the classes in the Java API's javafx.event

4. Class-Level Design Viewpoint

As we mentioned before, the design will encompass the DataViLiJ application. The following UML Class Diagram reflect this. Note that due to the complexity of the project, we present the class design using a series of diagrams going from the overview diagrams to detailed ones.

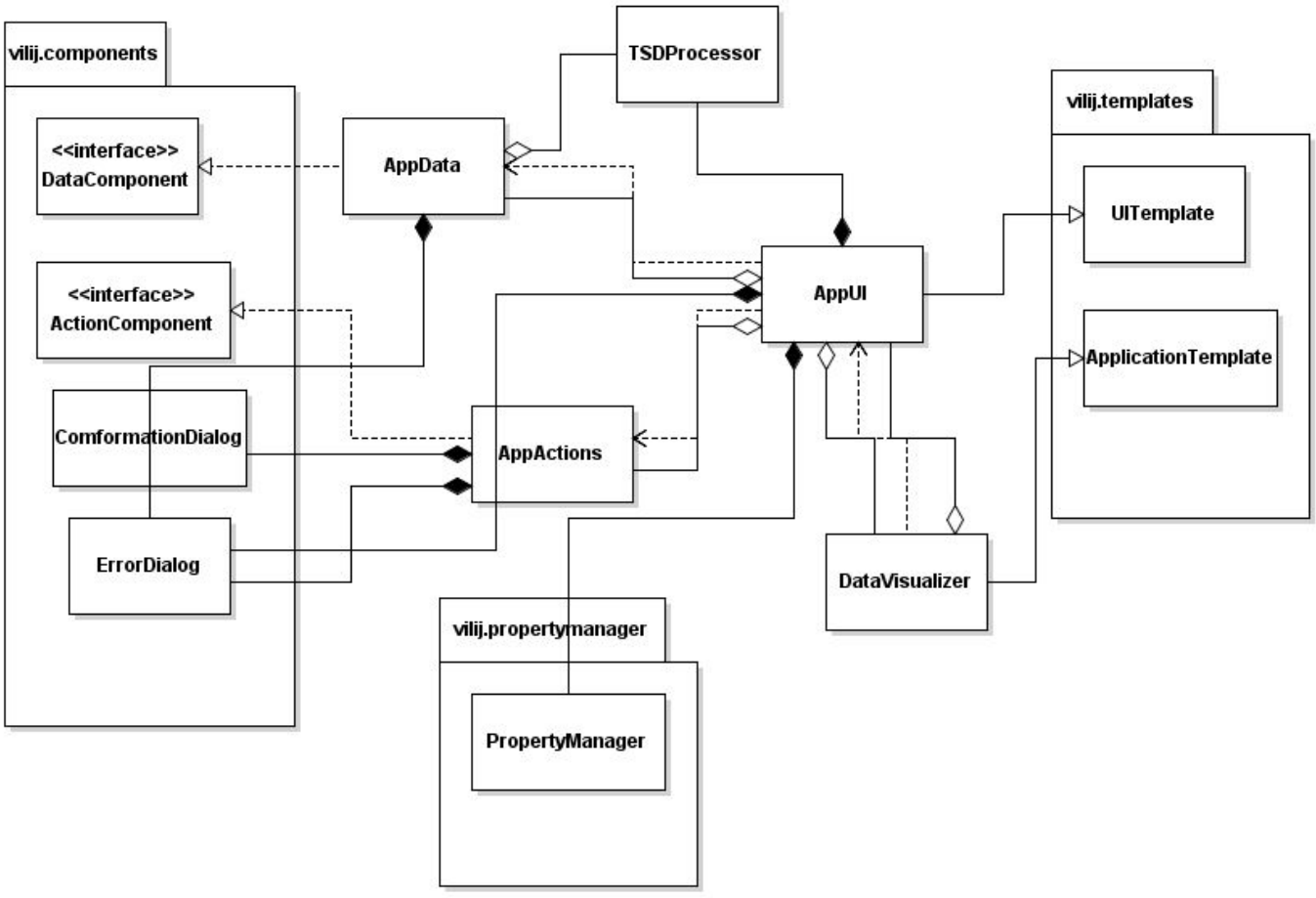

Figure 4.1: DataViLiJ Overview UML Class Diagram

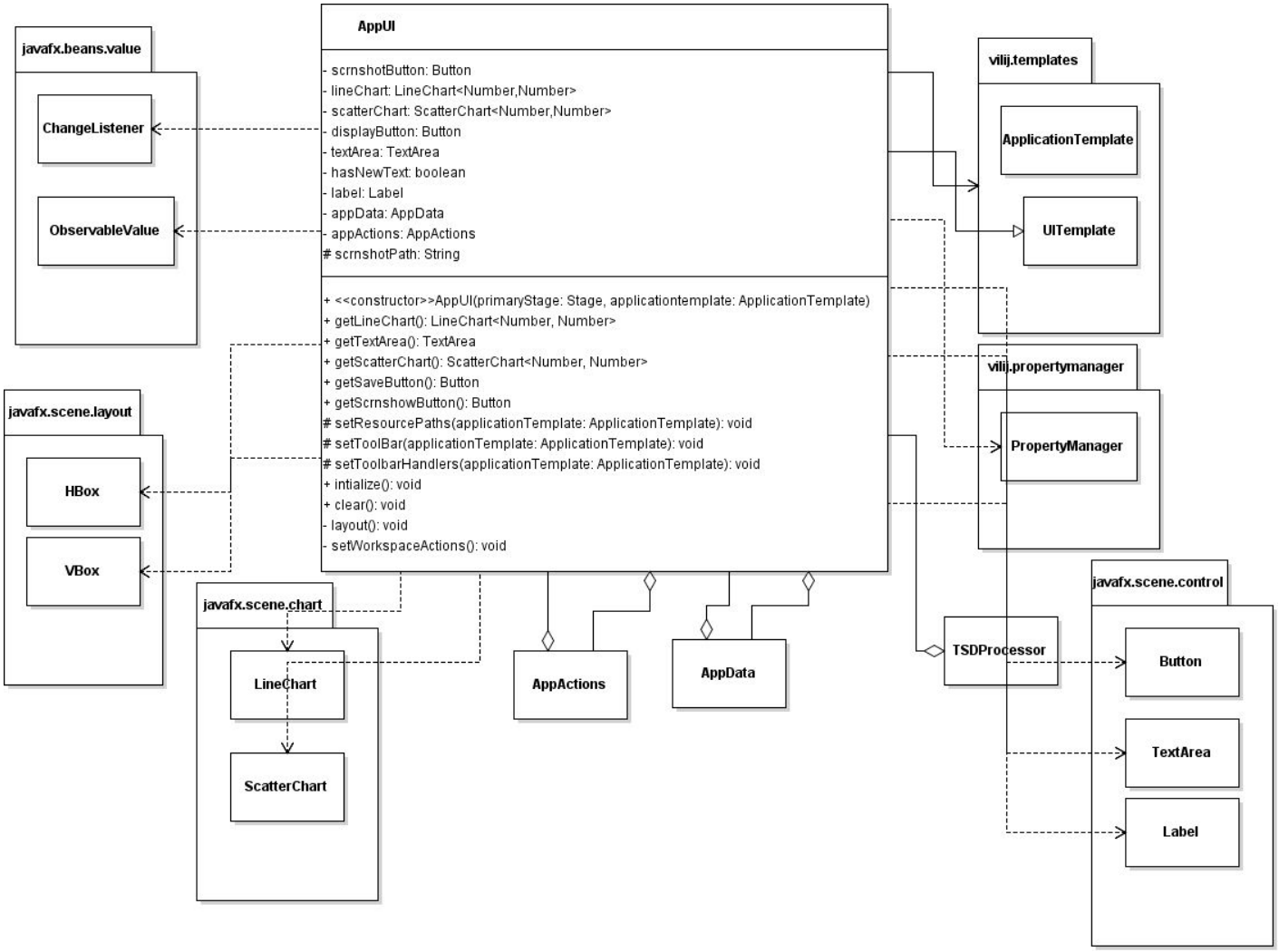

Figure 4.2: Detailed AppUI UML Class Diagram

This is the application's user interface implementation.

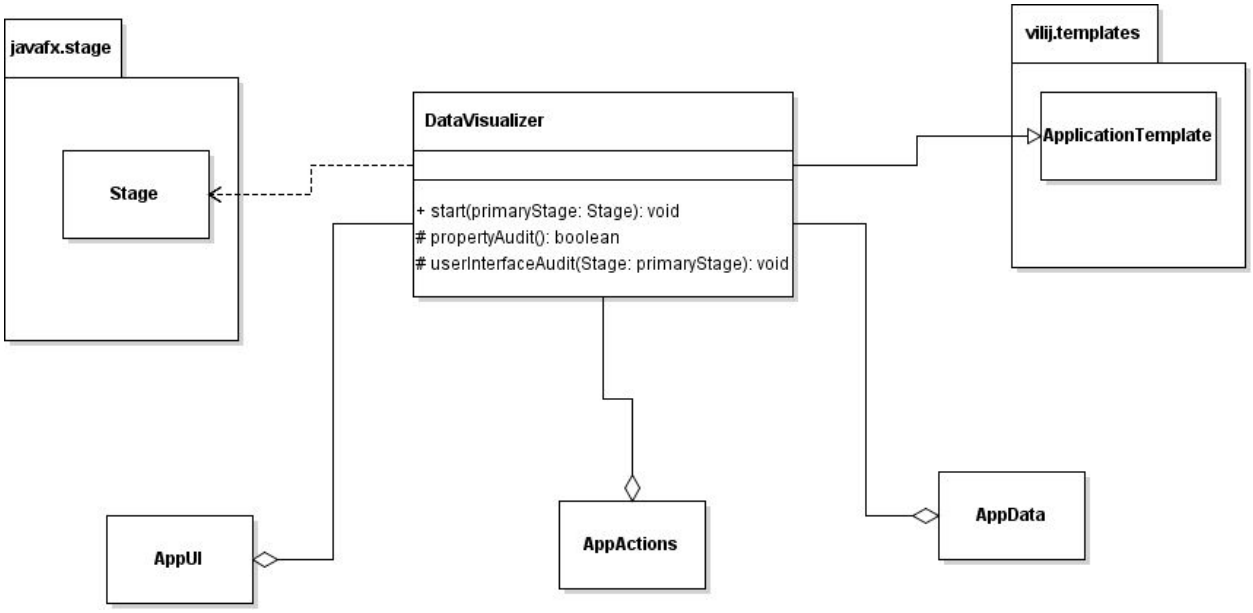

Figure 4.3: Detailed DataVisualizer UML Class Diagram

This class is the main class from which the application is run. The various components here must be concrete implementations of types defined in vilij.component.

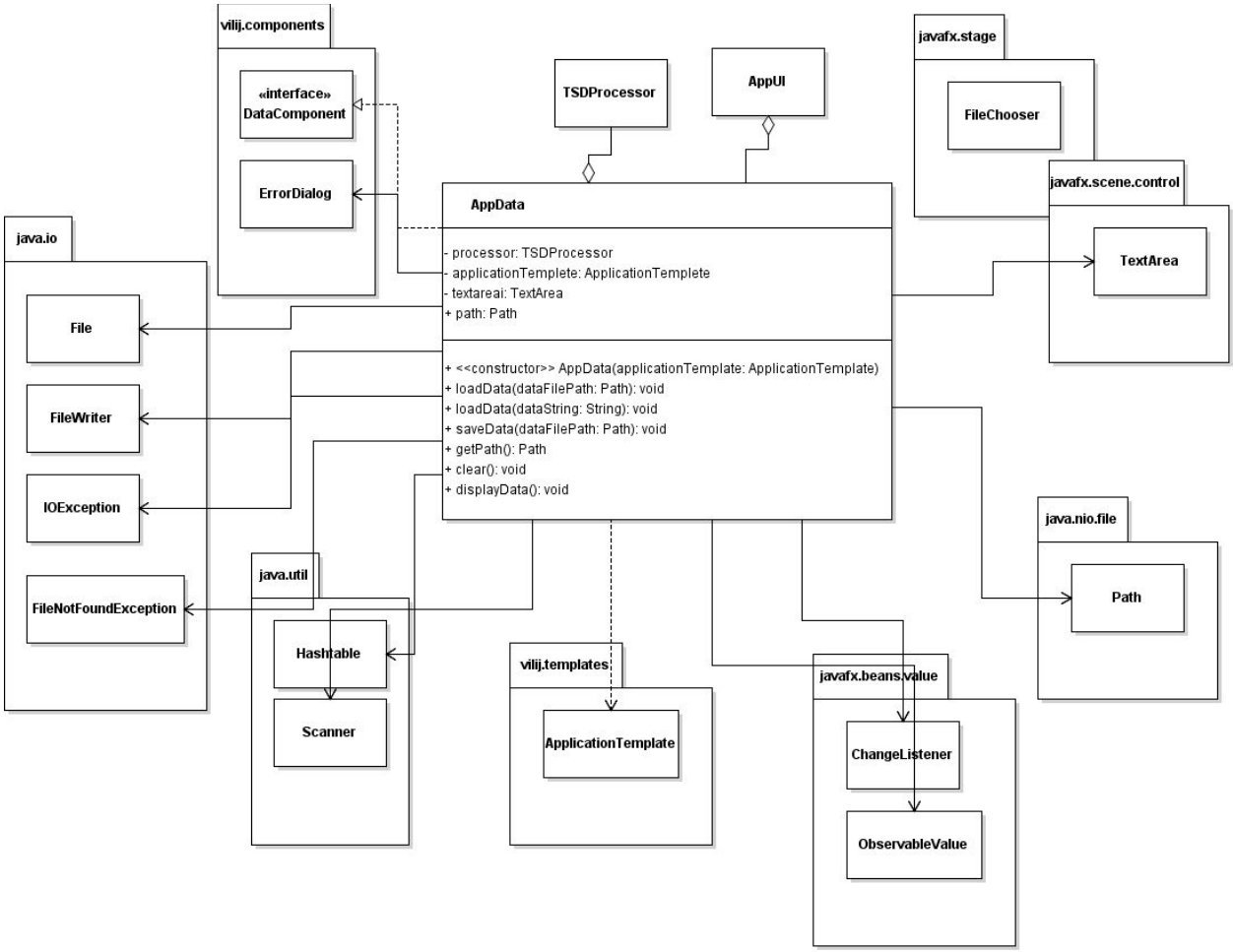

Figure 4.4: Detailed AppData UML Class Diagram

This is the concrete application-specific implementation of the data component devindec by the Vilij framework.

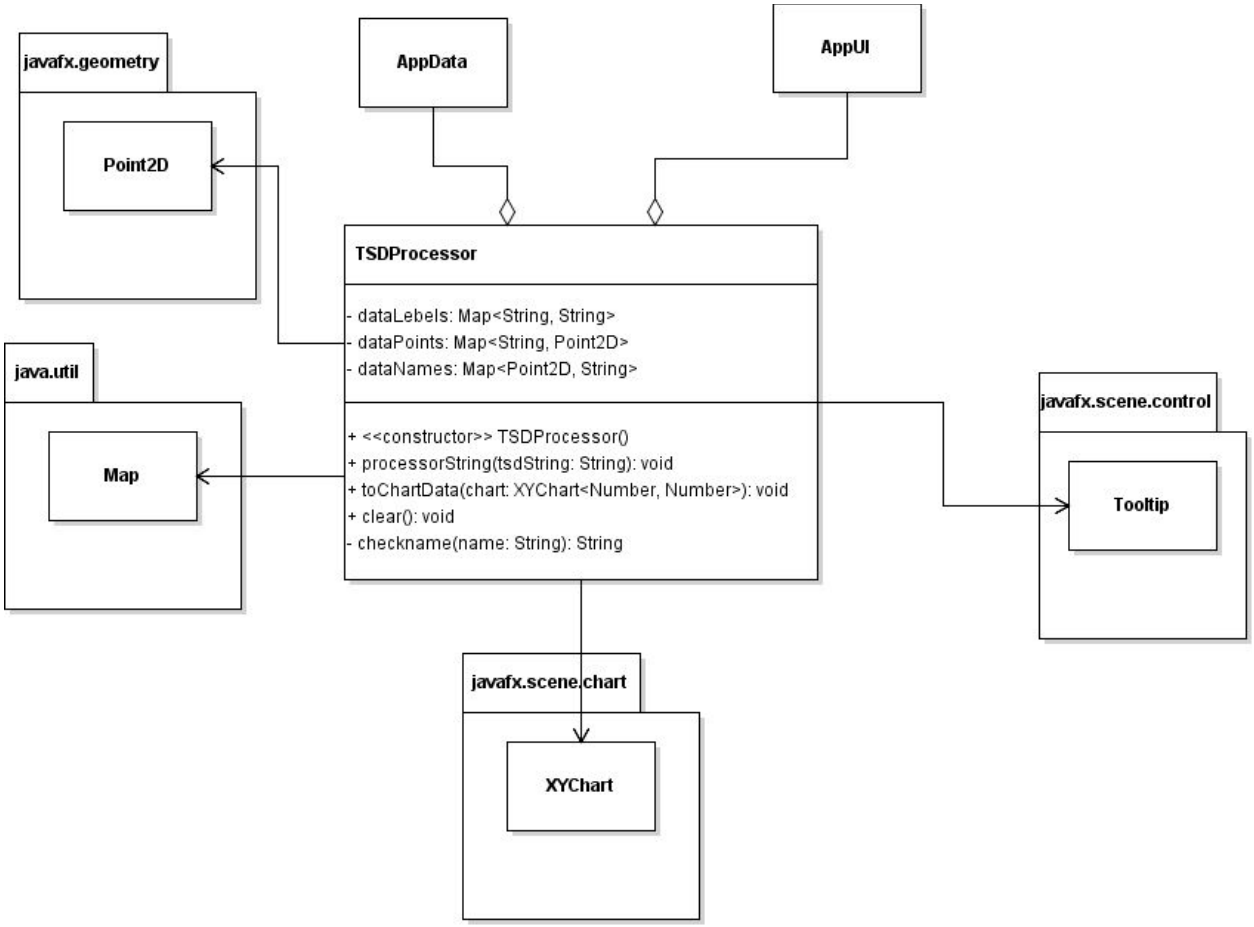

Figure 4.5: Detailed TSDProcessor UML Class Diagram

The data file used by this data visualization application follow a tab-separated format, where each data point is named, labeled, and has a specific location in the 2-dimensional X-Y plane. This class handles the parsing and processing of such data. It also handles exporting the data to a 2-D plot.

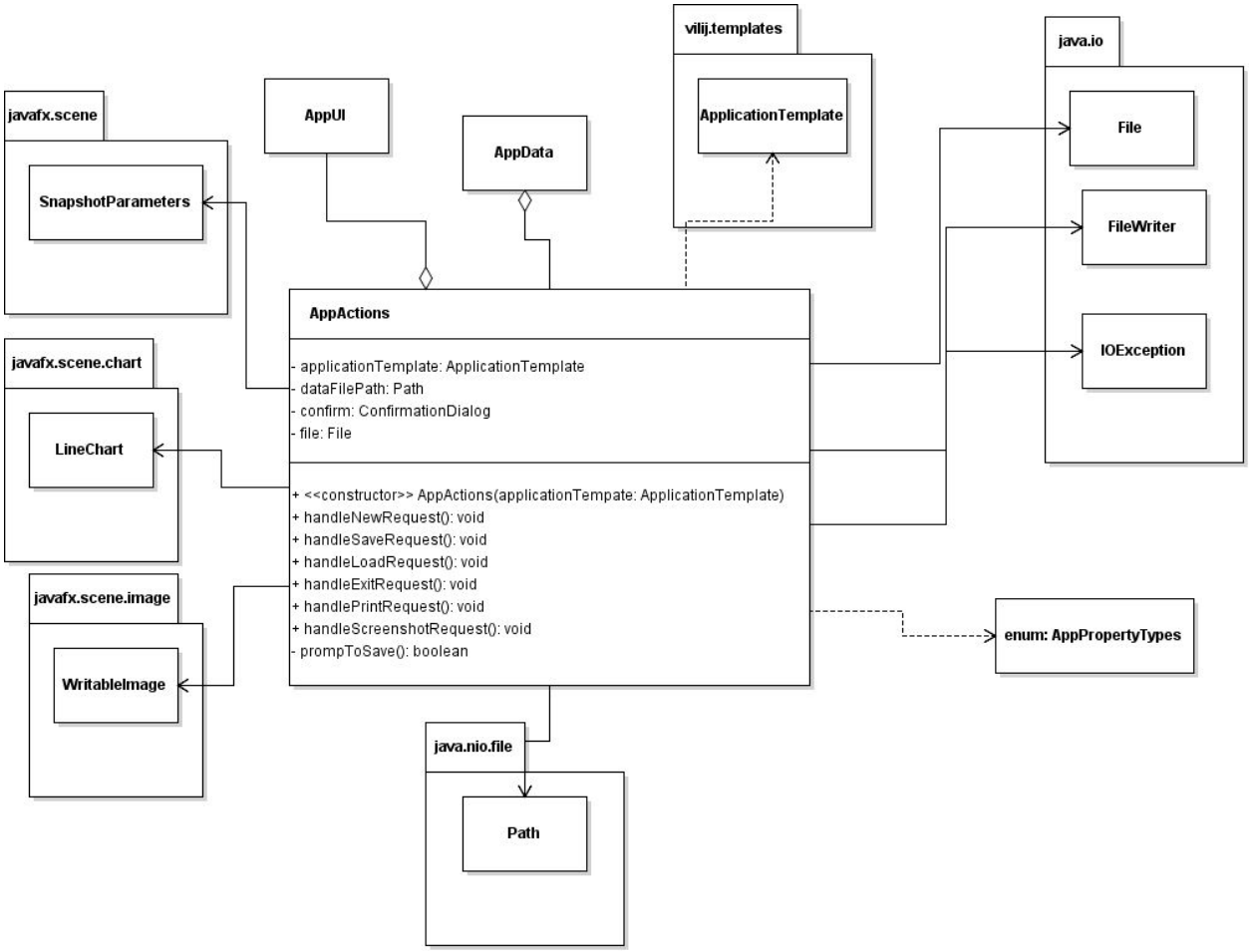

Figure 4.6: Detailed AppActions UML Class Diagram

This is the concrete implementation of the action handlers required by the application.

5. 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.

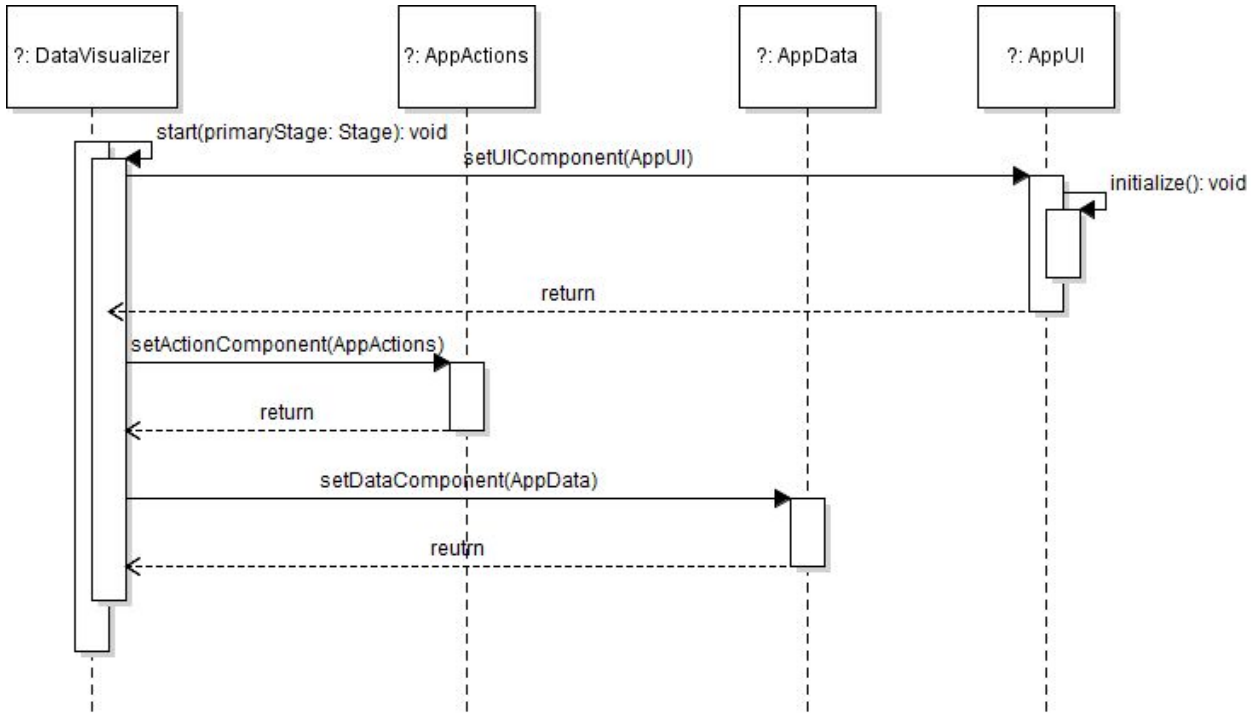

Figure 5.1: Start Application Use (Use Case 1)

Upon starting the application, the user shall see the main screen, which serves as the primary window of the GUI. Initially, it should only contain the toolbar and an empty area for the data display (i.e., a JavaFX chart or plot). The window size and decorations are to be finalized by the UI designer. The designer should also select appropriate icons and tooltips for each toolbar button.

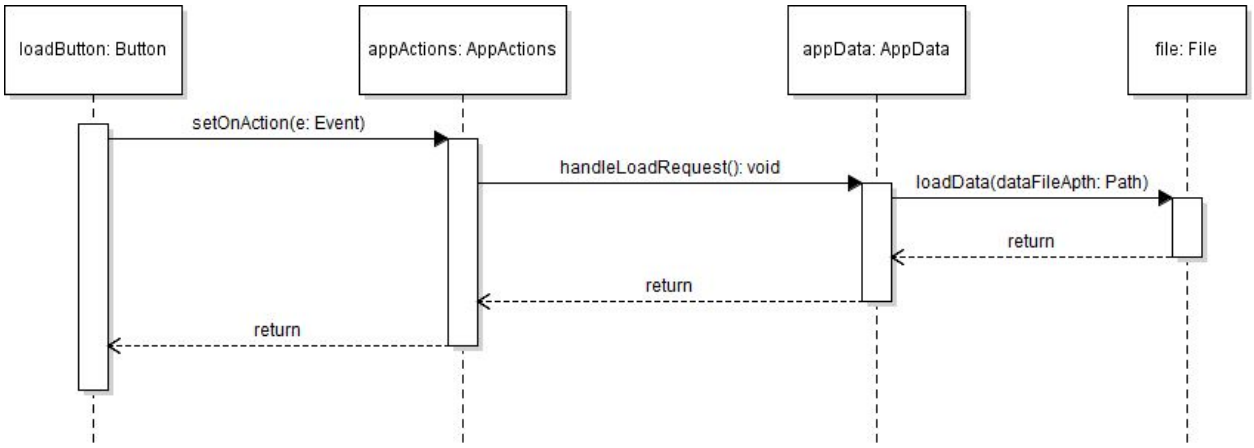

Figure 5.2: Load Data(Use Case 2)

The user shall be able to load valid data from a file in the TSD format by clicking the Load button in the toolbar. If the data is valid, then the top 10 lines will be shown in a read-only text box. Additionally, the following information about the data will be displayed: (a) the number of instances, (b) the number of labels, (c) the label names, and (d) the source of the data. If the data is invalid, then an appropriate error dialog shall be shown to the user.

Once the data is loaded, the GUI shall also allow the user to select the type of algorithm to be run. At this point, it should be noted that instances need not always have a label. The null label name is reserved for this purpose. This label must not be considered a usual label. Further, the type of algorithm is often decided by the type of data. Classification algorithms require two non-null labels. Therefore, the user shall not be able to choose 'classification' if the loaded data does not meet this criterion.

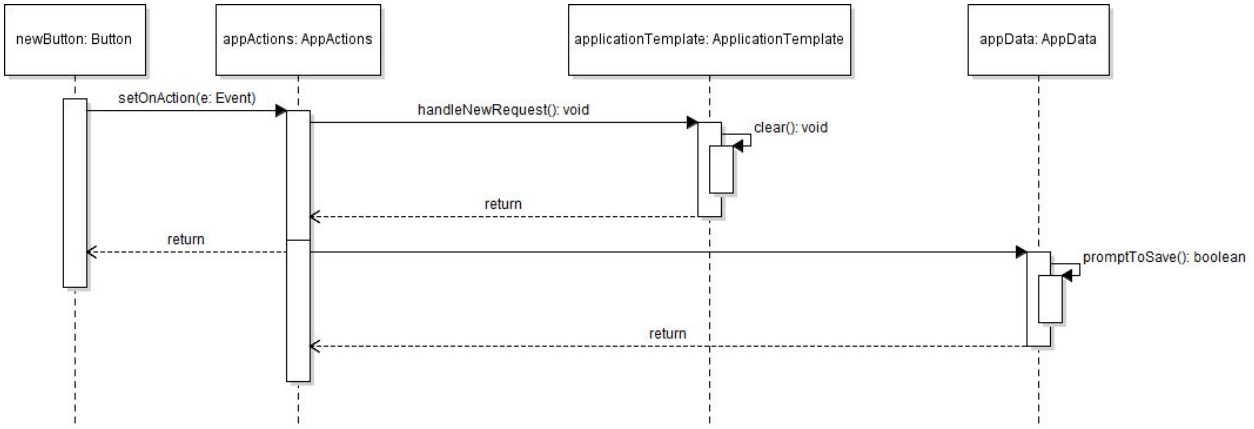

Figure 5.3: Create New Data(Use Case 3)

The user shall be able to create new data by clicking on the New button in the toolbar. This use case is similar to the previous, except that the text area will be active to allow the user to type in data in the TSD format. Here, the GUI primary window must provide the user with toggle that tells the program when the creation of new data is finished (e.g., a button that toggles between "Done" and "Edit"). If the data is valid, information about the data

must appear just like in Use Case 2. Similarly, the user must also then be able to choose an algorithm type. Additionally, once the user indicates that data creation is finished, the text area must become disabled (and in this example, the control change from "Done" to "Edit").

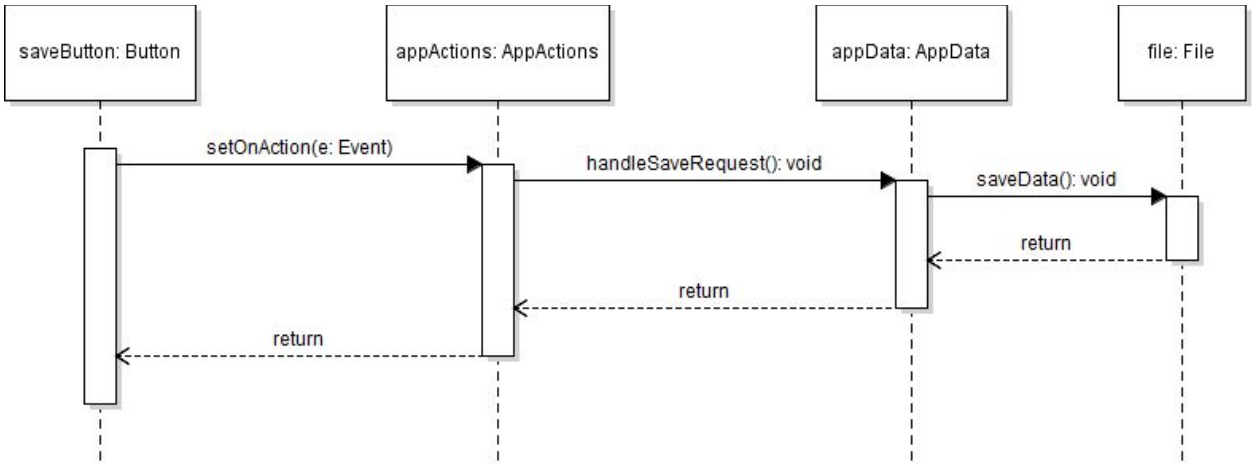

Figure 5.4: Save Data(Use Case 4)

If the user chose to create new data, then he or she shall be able to save it by clicking the Save button in the toolbar. This button must remain disabled while there is no data to be saved, which can happen when (a) there is no created data, or (b) there is no change in the created data since it was saved the last time. The data must be saved in a TSD format (e.g., by restricting the extension in the FileChooser).

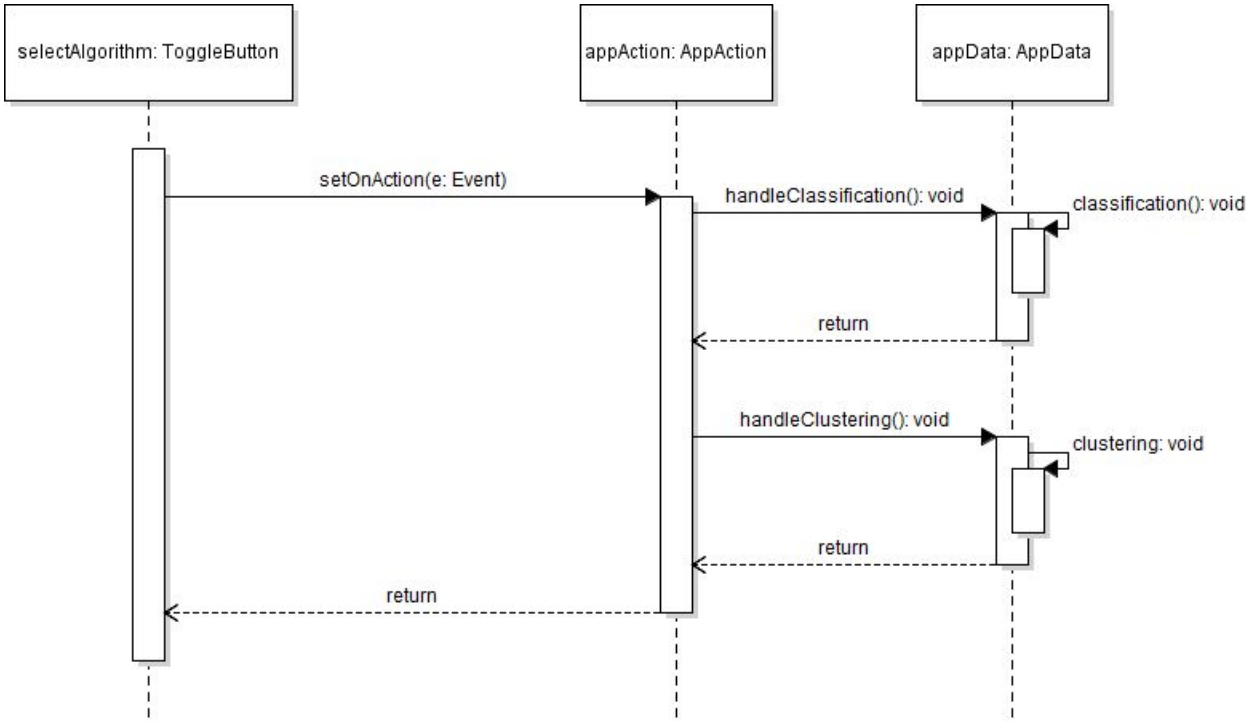

Figure 5.5: Select Algorithm Type(Use Case 5)

Once there is valid data in the application, due either to the loading of valid data or to the creation of new data, the user shall be able to select one of two following types of algorithm: (a) classification, or (b) clustering. However, based on the limited scope of this project, classification can only be done if there are exactly two labels. Therefore, if this condition is not met, the user must not be able to make this choice. This may be implemented by, say, disabling the 'classification' option. Upon selecting the algorithm type, the primary window shall display a list of algorithms of that type to the user. From this list, the user must be able to select the actual algorithm to run.

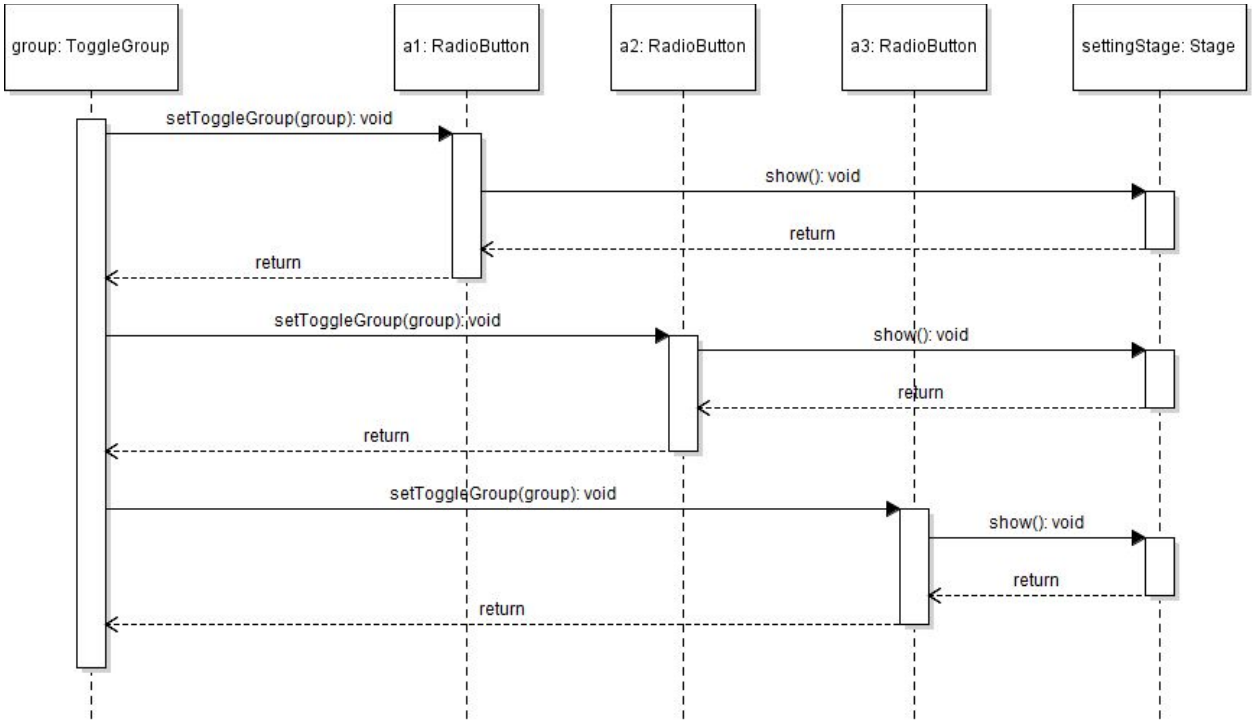

Figure 5.6: Select Algorithm(Use Case 6)

Upon selecting the algorithm type, the user shall be able to select an actual algorithm from that category. Note that there is a configuration/settings icon next to each choice of the algorithm. This is the control for the algorithm's run configuration, described next in Use Case 7. The actual running of the selected algorithm with the data happens after the user sends the signal to run the algorithm through the appropriate GUI control.

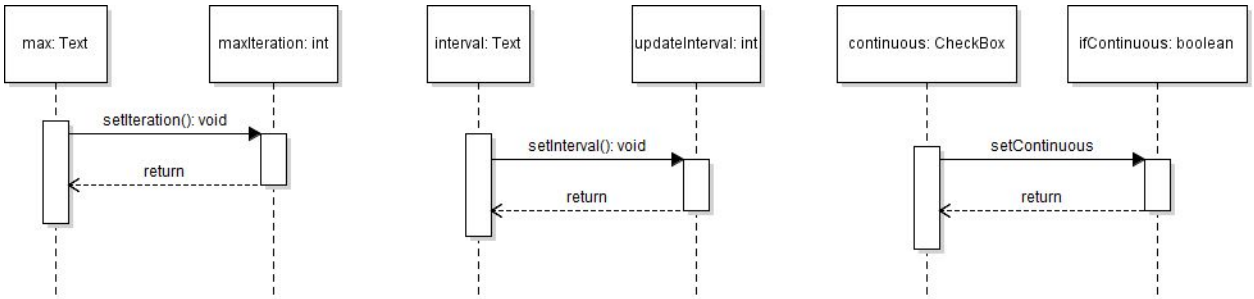

Figure 5.7: Select Algorithm Running Configuration (Use Case 7)

Every algorithm shall be made available with a 'run configuration'. This configuration must be filled in by the user when the algorithm is run the first time. Then onward, this configuration shall be used as the default setting for that particular algorithm. The configuration settings shall be provided by the user on a separate Configuration Window. The minimal configurations that must be provided by the user are (a) the number of

iterations of the algorithm, (b) the interval, specified in terms of the number of iterations, at which the data will be dynamically updated in the primary window's display area (e.g., every '5' iterations), and (c) whether or not the algorithm should automatically run till its end or wait for the user to indicate that it should continue. The third option is a valuable tool for the user since if the algorithm runs till the very end, the user gets to see an animation of how the data changes throughout the algorithm's run time. The disadvantage, however, is that the user cannot export any intermediate image. On the other hand, if the user tells the algorithm to dynamically update the display but wait for the user's indication before continuing, then the overall process is much slower, but the user can spend more time studying the changes and export even those charts that show the intermediate state of the data during the algorithm's run.

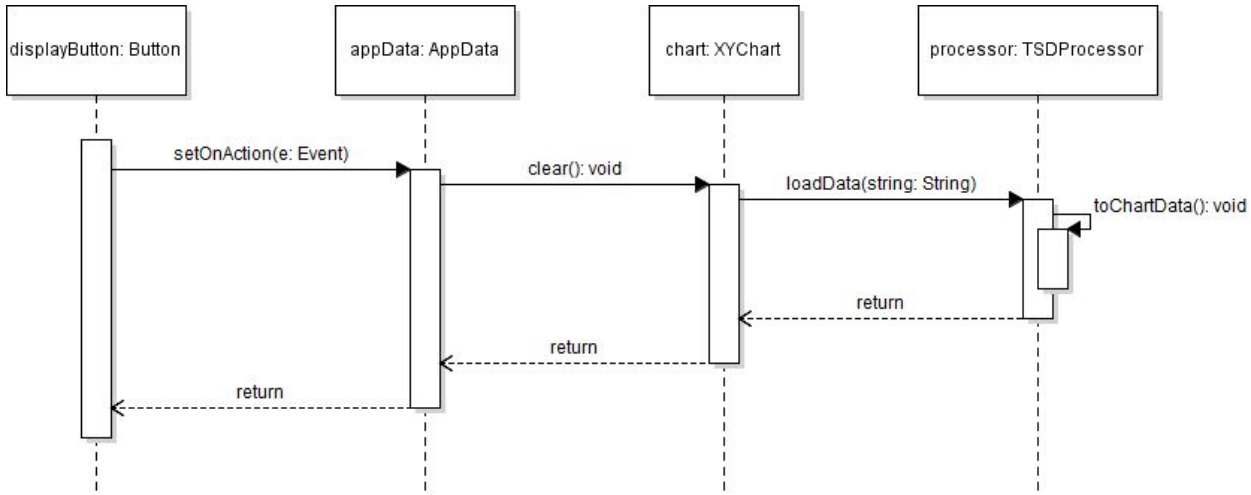

Figure 5.8: Running an Algorithm(Use Case 8)

If an algorithm is selected and its run configuration has been provided, then the user shall be able to run this algorithm on the available data. If the run configuration has disabled continuous run (i.e., the user must tell the algorithm to resume after every update interval), then the GUI control must make this obvious (e.g., by toggling between enable/disable of the run button). If the run configuration has enabled continuous run, then the GUI control to run the algorithm must remain disabled until the current run is finished.

Note that even though these algorithms may often manipulate the data, this must not be reflected in the original data that was provided to them. That is, the user must be able to retain the original data for, say, future runs of this or another algorithm.

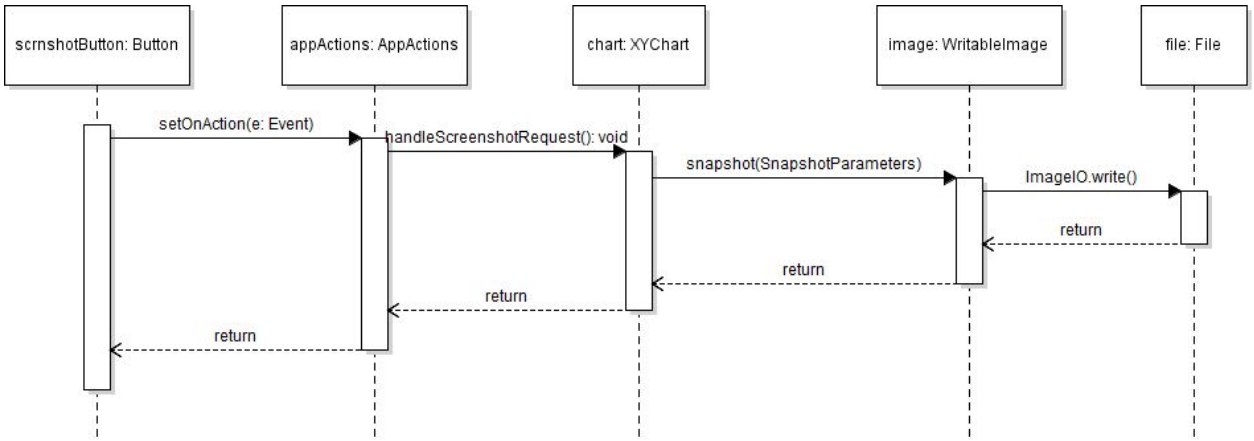

Figure 5.9: Export Data Visualization as Image(Use Case 9)

The screenshot/camera icon on the toolbar shall remain disabled as long as the chart/plot display is empty. However, since this application allows algorithms to run in a manner where it stops every few iterations (i.e., when the 'continuous run' is disabled from the run configuration), it shall be possible for the user to export the display during these pauses of the algorithm. While the algorithm is running (i.e., it is not paused), the export option must remain disabled irrespective of the run configuration's choice of the continuous run parameter.

The export action shall only export the visualization as an image, and not anything else in the GUI.

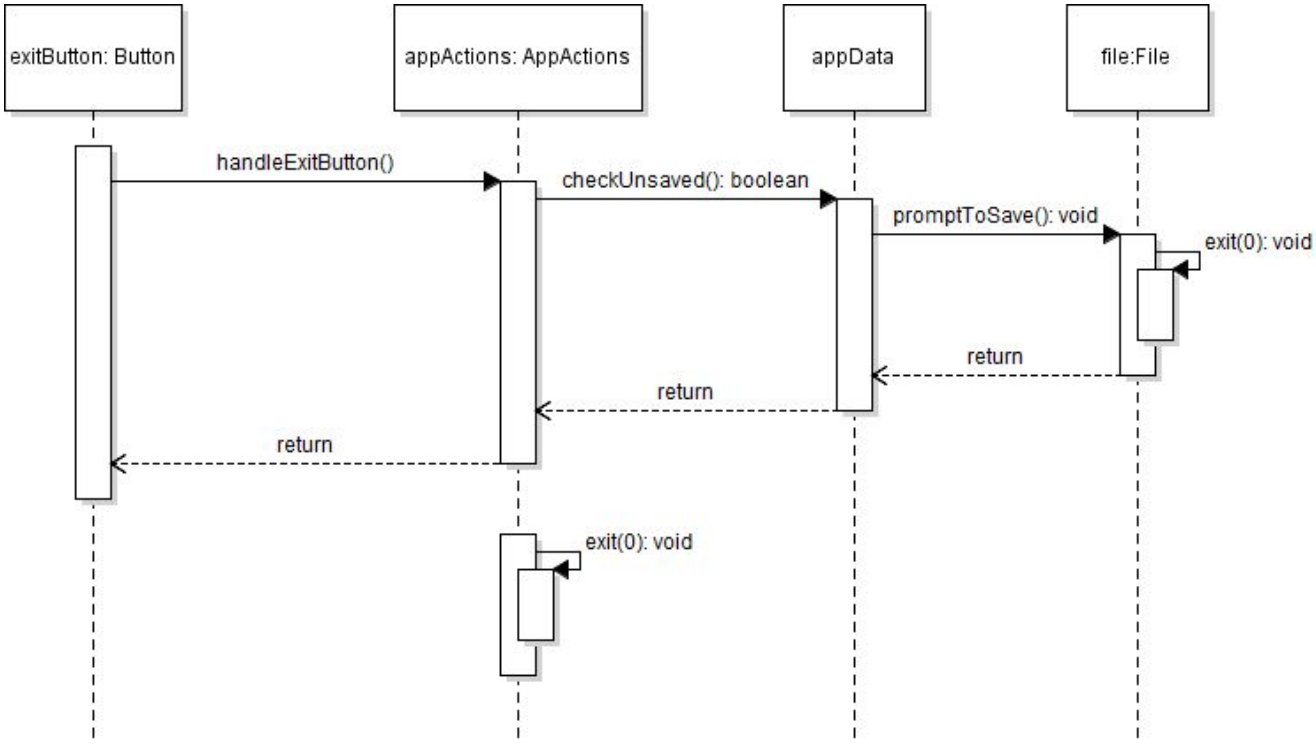

Figure 5.10: Exit Application(Use Case 10)

The user shall be able to exit the DataViLiJ application at any time, including while an algorithm is running. However, if there is any unsaved data or an algorithm has not finished running, an appropriate warning dialog shall be provided to the user. Note that there are two different types of exceptional conditions here depending on whether or not there is (a) unsaved data, and (b) an algorithm currently running.

6. File/ Data Structures and Formats

Note the Vilij framework including components(ActionComponent, ConfirmationDialog, DataComponent, Dialog, ErrorDialog, UIComponent) , propertymanager(PropertyManager), settings(InitializationParams, PropertyTypes), and templates((ApplicationTemplate, UITemplate) will be provided inside the Vilij package. This should be imported into the necessary project for the DataViLiJ application and will be included in the deployment of a single, executable JAR file titled DataVisualization.jar. Note that all necessary and art file must accompany this program. Figure 6.1 specifies the necessary file structure the launched application should use. Note that all necessary images should of course go in the image directory.

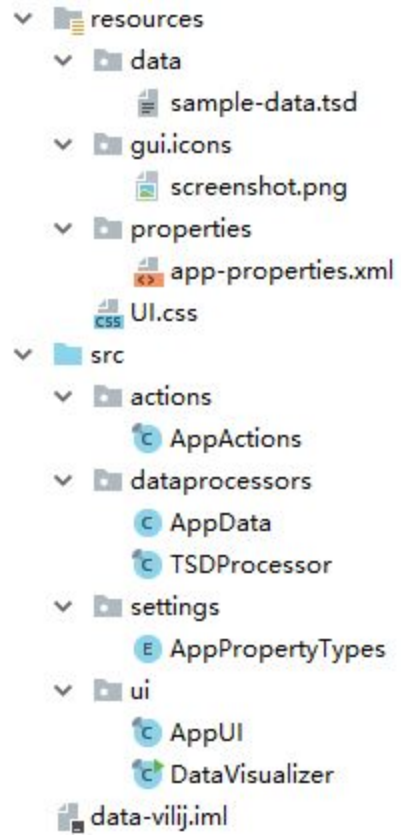

Figure 6.1: DataViLiJ application and file structure.

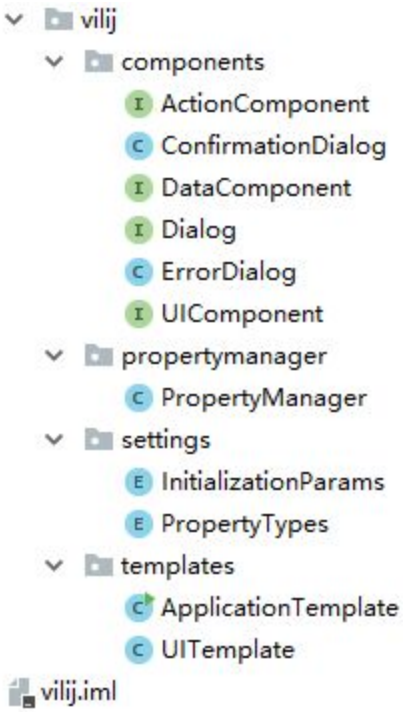

FIgure 6.2: Vilij Framework structure.

```xml
<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 6.3 & 6.4: properties.xml format

```
<properties>
    <property_list>
        <!-- RESOURCE FILES AND FOLDERS -->
        <property name="DATA_RESOURCE_PATH" value="data"/>

        <!-- USER INTERFACE ICON FILES -->
        <property name="SCREENSHOT_ICON" value="screenshot.png"/>

        <!-- TOOLTIPS FOR BUTTONS -->
        <property name="SCREENSHOT_TOOLTIP" value="Screenshot"/> <!-- will print current view of image to a file -->

        <!-- WARNING MESSAGES -->
        <property name="EXIT_WHILE_RUNNING_WARNING"
                value="An algorithm is running. If you exit now, all unsaved changes will be lost. Are you sure?"/>

        <!-- ERROR MESSAGES -->
        <property name="RESOURCE_SUBDIR_NOT_FOUND" value="Directory not found under resources."/>

        <!-- APPLICATION-SPECIFIC MESSAGE TITLES -->
        <property name="SAVE_UNSAVED_WORK_TITLE" value="Save Current Work"/>

        <!-- APPLICATION-SPECIFIC MESSAGES -->
        <property name="SAVE_UNSAVED_WORK" value="Would you like to save current work?"/>
```

```
        <!-- APPLICATION-SPECIFIC PARAMETERS -->
        <property name="DATA_FILE_EXT" value=".tsd"/>
        <property name="DATA_FILE_EXT_DESC" value="Tab-Separated Data File"/>
        <property name="TEXT_AREA" value="text area"/>
        <property name="SPECIFIED_FILE" value=" specified file"/>

    </property_list>
    <property_options_list/>
</properties>
```

Figure 6.5&6.6: app-properties.xml format

```java
public interface UIComponent {

    /**
     * Accessor method to get the application's window, which is the primary stage within which the full application GUI
     * is placed.
     *
     * @return the application's primary stage
     */
    Stage getPrimaryWindow();

    /**
     * Accessor method to get the scene graph of the application's main window.
     *
     * @return the application's scene graph
     */
    Scene getPrimaryScene();

    /** @return the application title */
    String getTitle();

    /** Method to enforce the initialization of the user interface styles for any Vilij application. */
    void initialize();

    /** Method to clear any existing visualization on the graphical user interface front-end. */
    void clear();
}
```

Figure 6.7: UIComponent format

```java
public interface ActionComponent {

    void handleNewRequest();

    void handleSaveRequest();

    void handleLoadRequest();

    void handleExitRequest();

    void handlePrintRequest();
```

Figure 6.8: ActionComponent format

```
public interface DataComponent {

    void loadData(Path dataFilePath);

    void saveData(Path dataFilePath);

    void clear();
}
```

FIgure 6.9: DataComponent

```
.chart-series-line {
    -fx-stroke: transparent;
}
.chart-plot-background {
    -fx-border-color: black;
    -fx-border-width: 5px;
}
.axis{
    -fx-tick-length: 20;
    -fx-minor-tick-length: 10;
}
.axis-tick-mark{
    -fx-stroke: black;
    -fx-stroke-width: 3;
}
```

Figure 6.10: UI.css format

7. Supporting Information

This document should serve as a reference for the designers and developers in the future stages of the development process. Since this product involves the use of a specific data format and specialized algorithms, we provide the necessary information in this section in the form of three appendixes.

Appendix A: Tab-Separated Data (TSD) Format

The data provided as input to this software as well as any data saved by the user as a part of its usage must adhere to the tab-separated data format specified in this appendix. The specification are as follows:

1. A file in this format must have the ".tsd" extension.

2. Each line (including the last line) of such a file must end with '\n' as the newline character.

3. Each line must consist of exactly three components separated by '\t'. The individual components are

- **a. Instance name, which must start with '@'**
- **b. Label name, which may be null.**
- **c. Spatial location in the x-y plane as a pair of comma-separated numeric values. The values must be no more specific than 2 decimal places, and there must not be any whitespace between them.**

4. There must not be any empty line before the end of file.

5. There must not be any duplicate instance names. It is possible for two separate instances to have the same spatial location, however.

Appendix B: Characteristics of Learning Algorithms

As mentioned before, the algorithms and their implementation, or an understanding of their internal workings, are not within the scope of this software. For the purpose of design and development, it suffices to understand some basic characteristics of these algorithms. There are provided in this appendix.

B.1 Classification Algorithms

These algorithms 'classify' or categorize data into one of two known categories. The categories are the labels provided in the input data. The algorithm learns by trying to draw a straight line through the x-y 2-dimensional plane that separates the two labels as best as it can. Of course, this separation may not always be possible, but the algorithm tries nevertheless. An overly simplistic example where an algorithm is trying to separate two genders based on hair length (x-axis) and count (y-axis) is shown in Fig. 5 (picture courtesy dataaspirant.com). The output of a classification algorithm is, thus, a straight line. This straight line is what is updated iteratively by the algorithm, and must be shown as part of

the dynamically updating visualization in the GUI. Moreover, a classification algorithm will NOT change the label of any instance provided as part of its input, and the actual (x, y) position of any instance in its input.

B.2 Clustering Algorithms

Clustering algorithms figure out patterns simply based on how data is distributed in space. These algorithms do not need any labels from the input data. Even if the input data has labeled instances, these algorithms will simply ignore them. They do, however, need to know the total number of labels ahead of time (i.e., before they start running). Therefore, their run configuration will comprise the maximum number of iterations (it may, just like classification algorithms, stop before reaching this upper limit), the update interval, the number of labels, and the continuous run flag. The output of a clustering algorithm is, thus, the input instances, but with its own labels. The instance labels get updated iteratively by the algorithm, and must be shown as part of the dynamically updating visualization in the GUI.

Appendix C: Mock Algorithms for Testing

For testing, it suffices to create simple mock algorithms.

For classification algorithm testing, simply writing a test code that creates a random line every iteration is sufficient. To plot such a line in the GUI output, this algorithm can, for example, return a triple (a, b, c) of integers to represent the straight-line ax + by + c = 0, where a, b, and c are randomly generated using standard Java classes like java.util.Random.

For clustering algorithm testing, simply writing a test code that takes as input the number of labels (say, n) along with the data, and randomly assigns one of the labels {1, 2, …, n} to each instance.

7.2 Appendixes

N/A