DataViLiJ™

Software Design Description

DataViLiJ™

Author: Rebecca Hassett

Eigen Inc.™

March, 2018

$$\bar{x} = \frac{\int_{-\infty}^{\infty} xP(x,t)dx}{\int_{-\infty}^{\infty} P(x,t)dx} = \int_{-\infty}^{\infty} \psi^*(x,t)x\psi(x,t)dx$$

Abstract: This document describes the design for the software DataViLiJ, a program to help computer science students visualize how clustering and classification AI algorithms affect data.

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

Table of Contents

Table of Figures

1. Introduction

1.1 Purpose

The purpose of this Software Design Description is to reveal the implementation designs and scope of the DataViLiJ application described in the DataViLiJ Software Requirements Specifications document. UML class diagrams will provide a visual depiction of the relationships between various packages, classes, method signatures, variables, and frameworks necessary to construct this application. UML sequence diagrams are included to indicate the order of interactions and messages sent between different objects once the user interacts with the application. The intended audience for this document comprises of instructors, teaching assistants, and software designers. After reading this document, one should clearly understand the design as well as envision the functionalities and aesthetics of the application.

1.2 Product Scope

DataViLiJ is an application that will help computer science students and beginning professionals in artificial intelligence visualize how classification and clustering algorithms “learn” from data. Classification and clustering algorithms, in theory, should not be limited to a fixed number of data labels, but this project will be limited to exactly two labels for classification algorithms, and at most four labels for clustering algorithms. Furthermore, the design and development of this

product assumes data is 2-dimensional because 3D visualization is beyond the scope of DataViLiJ. Touch screen capabilities are beyond the scope of this project as well. This application is constructed using the frameworks vilij, xmlutil, and JavaFX. This document contains descriptions for design choices made in constructing the application as well as any specific modifications made to the vilij and xmlutil frameworks. The target language for this application is Java.

1.3 Definitions, Acronyms, and Abbreviations

1) Algorithm: In this document, an 'algorithm' refers to an AI algorithm that can "learn" from data input by a user and assign these data points labels.
2) Class Diagram: A UML static structure diagram which graphically depicts a system's classes detailing their attributes, methods, and relationships to each other.
3) 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 are two parts, one on each side of the line.
4) Clustering: A type of AI algorithm that learns to assign labels to instances based purely on the spatial distribution of the data points.
5) Cascading Style Sheet (CSS): A stylesheet which formats the layout and appearance of a project written in either XML or HTML.
6) Framework: A collection of interfaces, abstract and concrete classes in an object-oriented environment that are used in the development of applications as well as additional frameworks that provide a common service. Frameworks take advantage of abstraction by providing generic functionality for a broad service that is selectively refined into an application through user-written code.
7) Graphical User Interface (GUI): An interface including visual controls and indicators such as buttons, menus, dialogs, and check boxes within a

window which allows users to interact with an application. Users are quicker at learning how to use a GUI compared to command-line user interfaces.

8) IEEE: Institute of Electrical and Electronics Engineers, a professional association founded in 1963 to advance the technology and education of electrical engineering, computer engineering, telecommunications, and related disciplines.
9) Instance: A 2-dimensional data point that always has a name, may or may not contain a label, and then must have an x-value and a y-value.
10) Java: A high-level programming language designed to be run on any computer system through its use of a virtual machine which handles all interactions between the Java application and the hardware.
11) Sequence Diagram: A UML dynamic structure diagram which graphically depicts the time sequence of object interactions and messages sent between objects to carry out the functionality of an application.
12) Unified Modeling Language (UML): A modeling language used by software developers to visualize, design, and document the components of a software project.
13) Use Case Diagram: A UML format revealing the relationship between a user and various *use cases* in which a user is involved.
14) User: Someone who uses the DataViLiJ GUI to interact with the application's software.
15) XML: eXtensible Markup Language, a markup language used in the processes of transporting data and storing data.

1.4 References

[1] IEEE Software Engineering Standards Committee. "IEEE Standard for Information Technology – Systems Design – Software Design Descriptions." In IEEE STD 1016-2009, pp.1-35, July 20 2009

[2] Banerjee, Ritwik. "*DataV iLiJ* ^{TM} - Software Requirements Specifications." Professaur, Inc., 2018.

1.5 Overview

The Software Design Document will clearly depict the design of the DataViLiJ application as described in the *DataViLiJ* ™ Software Requirements Specifications. Before proceeding to the implementation stage, all parties must agree upon the design choices made concerning the connection between components of the application. The remainder of this document consists of four chapters and supporting information. Chapter 2 provides the Package-Level Design Viewpoint describing the usage of components from the Java API as well as the various packages and frameworks included within the design. Chapter 3 provides the Class-Level Design Viewpoint including UML class diagrams to depict the classes and the relationships between classes necessary to construct this product. Chapter 4 provides the Method-Level Design Viewpoint using UML sequence diagrams to detail how methods will interact and exchange messages with each other in response to user interaction. Chapter 5 provides the file structures, data structures, and formats necessary for the project to be executed and deployed. The document will conclude with supporting information. All UML diagrams are built through the use of VioletUML.

2. Package-Level Design Viewpoint

Frameworks used to construct the DataViLiJ application includes JavaFX as well as the Vilij framework for data visualization and xmlutil for handling the transport and storage of data in the application. This application depends heavily on the Java API. The descriptions below includes components that need to be implemented, information about the frameworks used to build these components, and descriptions about specific components from the Java API used to build this application.

2.1 DataVilij Overview

The DataViLiJ application is constructed using the JavaFX framework, the Vilij framework for data visualization and xmlutil. Figure 2.1 specifies the packages to be developed and places all classes inside of these packages. It does not include framework packages that were not altered. The data-vilij module contains the folders for constructing the user interface. The algorithm module contains the algorithms that will modify the data loaded into or created in the UI. The communication module will communicate between the algorithm module and the data-vilij module. The data-vilij module contains a resource folder with the css file for the gui layout, the data files saved from and loaded to the application, and xml files to prevent hard-coding strings.

Figure 2.1: DataViLiJ Package Overview

2.2 Java API Usage

Java is the programming language used for creating the DataViLiJ application. Figure 2.2 specifies the Java classes used in designing the DataViLiJ application.

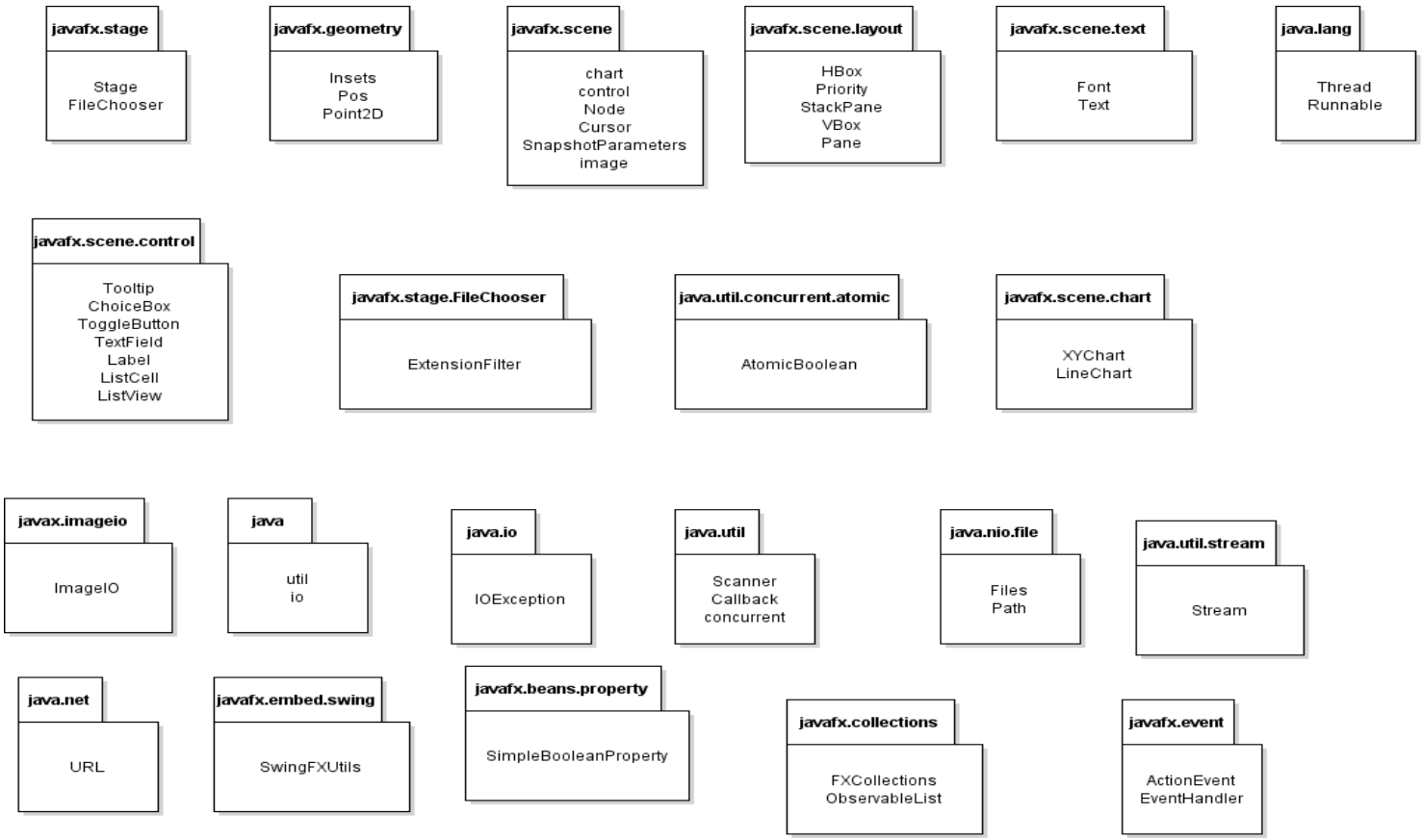


Figure 2.2: Java API Classes and Packages

2.3 Java API Usage Descriptions

Figures 2.3.1-2.3.18 describes how these Java classes will be used in the construction of the DataViLiJ application.

Class/Interface	Description
Stage	For setting the windows of the application.
FileChooser	For loading and saving tab-separated data files containing instances and saving screenshots.

Figure 2.3.1: Usage of classes in javafx.stage

Class/Interface	Description
Insets	For setting the spacing between the title, the text area, and the display button.
Pos	For centering the title, the text area, the display button, and any other controls within the workspace.
Point2D	For creating the 2-dimensional data point with an x-value and a y-value separated by a comma. Needed for constructing an instance.

Figure 2.3.2: Usage of classes in javafx.geometry

Class/Interface	Description
chart	For creating the chart for displaying the data
control	For creating the display button, toolbar buttons, the toolbar, and the text area for the GUI.
Node	For getting the node related to a series and setting it to a line, so that the lines between points in the same series can be set to transparent in the line chart and the series line added with the classification algorithm can be set to green.
Cursor	For setting the cursor icon.
SnapshotParameters	For taking a screenshot of the chart area when it is not empty.
image	For creating a writable image from the screenshot of the chart area, and writing the fx image to an image file to be saved.

Figure 2.3.3: Usage of classes in javafx.scene

Class/Interface	Descriptions
HBox	For setting panes that stretch from left to right. Used for containing the display button.
Priority	For having layout areas grow (or shrink) to increase (or decrease) with other layout areas of ALWAYS that have to grow (or shrink).
StackPane	For creating a layout where nodes are stacked on top of each other. Used for adding the chart to the workspace.
VBox	For layouts stretching from top to bottom. For the left panel of the workspace containing the title, the text area, and the display button.
Pane	For holding the list view that will be used to select the particular algorithm type.

Figure 2.3.4: Usage of classes in javafx.scene.layout

Class/Interface	Descriptions
Font	For selecting the font for text such as the title.
Text	For setting the title of the application.

Figure 2.3.5: Usage of classes in javafx.scene.text

Class/Interface	Descriptions
Tooltip	For showing the name of an instance when hovered over in the chart.
ChoiceBox	For selecting the algorithm type in the workspace.
ToggleButton	For switching between ‘Edit’ and ‘Done’ for newly entered data
TextField	For entering numbers in the run configurations of the algorithms such as the maximum number of iterations and the update iterations.
Label	For labeling the list view that will be used to

	select the specific algorithm to be run.
ListCell	For customizing the cells in the ListView so that each cell will hold an algorithm name as well as a button for the configuration window of that algorithm.
ListView	For selecting the specific algorithm for the algorithm type that is chosen.
ToggleGroup	Toggle between edit and done.

Figure 2.3.6: Usage of classes in javafx.scene.control

Class/Interface	**Descriptions**
ExtensionFilter	For setting the extensions of files saved with instances as .tsd and setting the extensions of images of the chart to .img. For only loading files with extensions of .tsd into the text area.

Figure 2.3.7: Usage of classes in javafx.stage.FileChooser

Class/Interface	**Descriptions**
AtomicBoolean	For creating booleans that are updated automatically and are thread-safe. Used when creating a flag for whether or not processString method had an error.

Figure 2.3.8: Usage of classes in java.util.concurrent.atomic

Class/Interface	**Descriptions**
XYChart	For storing the various series depending on the labels of the instances.
LineChart	For visualizing instances and how AI algorithms learn from these instances.

Figure 2.3.9: Usage of classes in javafx.scene.chart

Class/Interface	**Descriptions**
ImageIO	For writing a javafx writable file to an image

	file to be saved containing the screenshot of the chart area.

Figure 2.3.10: Usage of classes in javax.imageio

Class/Interface	Descriptions
util	For handling operations involving the scanners for reading from files such as closing the scanners after use.
io	For handling io operations with reading from files, writing to files, and handling paths to files.

Figure 2.3.11: Usage of classes in java

Class/Interface	Descriptions
IOException	For catching errors from PrintWriter in case saving to a file is disrupted or cannot be completed.

Figure 2.3.12: Usage of classes in java.io

Class/Interface	Descriptions
Scanner	For reading instances from a file to be loaded to the text area or chart.
Callback	For setting up the cells in the algorithm list view to contain the algorithm name and button for the configuration window.
concurrent	For handling multiple events occurring at same time in different threads.

Figure 2.3.13: Usage of classes in java.util

Class/Interface	Descriptions
Files	For accessing methods that operate on files in the process of loading and saving instances.
Path	For handling the path to files containing instances for loading these files. For setting the

	initial directory of the file chooser.

Figure 2.3.14: Usage of classes in java.nio.file

Class/Interface	Descriptions
Stream	For applying functional programming to a collection of objects such as strings representing an instance of data. Used to store parts of an instance of data in Hash Maps by parsing the strings representing the instance of data.

Figure 2.3.15: Usage of classes in java.util.stream

Class/Interface	Descriptions
URL	A Uniform Resource Locator. Used for accessing the resource folder.

Figure 2.3.16: Usage of classes in java.net

Class/Interface	Descriptions
SwingFXUtils	For writing a writable image to an image file to save the screenshot of the chart when it is not empty and the screenshot button is pressed.

Figure 2.3.17: Usage of classes in javafx.embed.swing

Class/Interface	Descriptions
SimpleBooleanProperty	For wrapping a boolean into an object.

Figure 2.3.18: Usage of classes in javafx.beans.property

Class/Interface	Descriptions
FXCollections	For creating the list of strings of algorithm names for each algorithm type.
ObservableList	For containing the list of strings of algorithm names for each algorithm type.

Figure 2.3.19: Usage of classes in javafx.collections

Class/Interface	Descriptions

ActionEvent	For creating an event when the configuration button is clicked inside of the ListView customized cell.
EventHandler	For handling the event when the configuration button is clicked inside of the ListView customized cell.

Figure 2.3.20: Usage of classes in javafx.event

Class/Interface	**Descriptions**
Thread	For creating a thread that handles communication tasks from the user interface by the data-vilij module and creating another thread that handles communication tasks by the algorithms module.
Runnable	For creating a runnable object to be passed to the Thread constructor so the application implements Runnable instead of extending Thread.

Figure 2.3.21: Usage of classes in java.lang

3. Class-Level Design Viewpoint

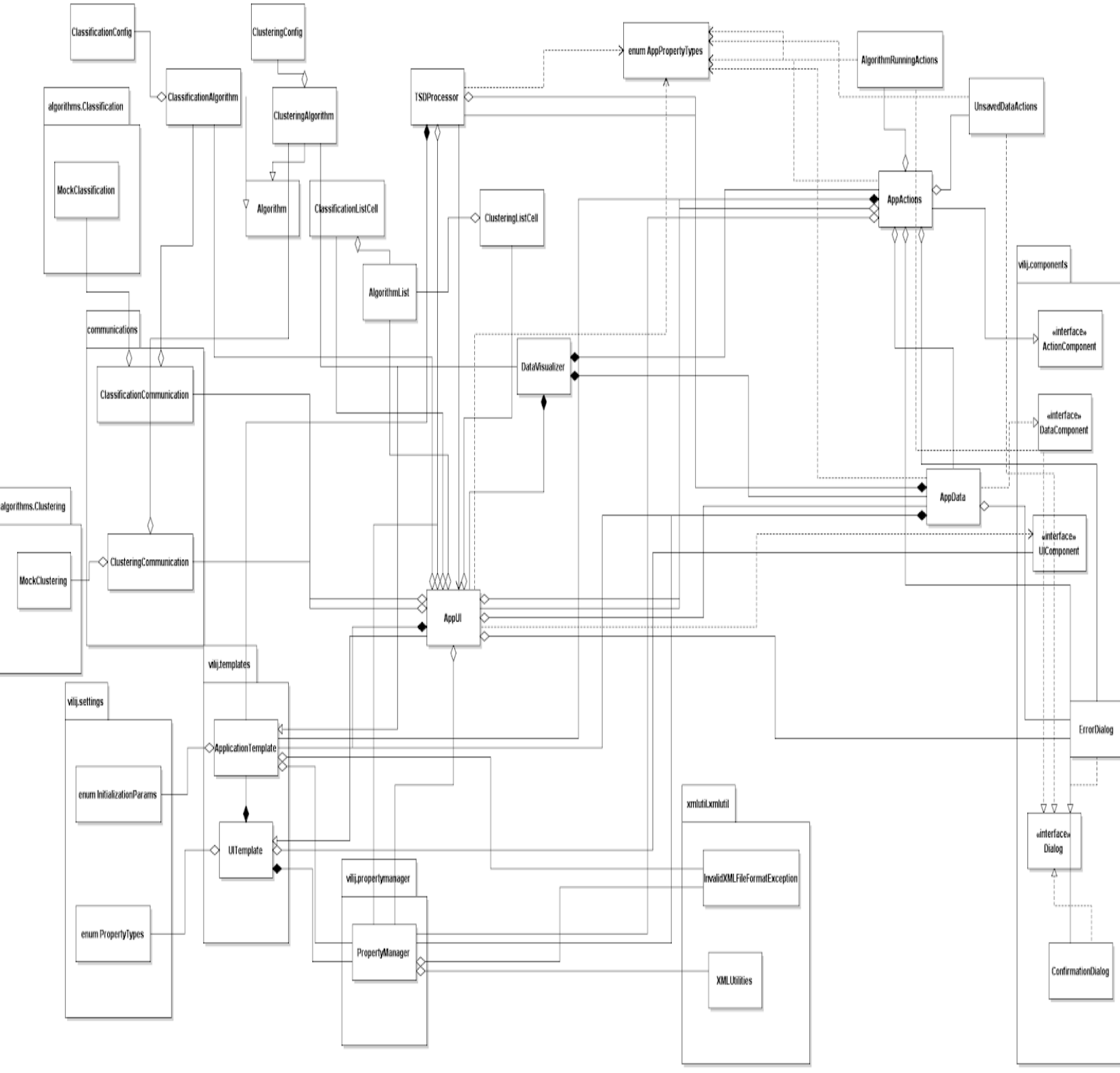


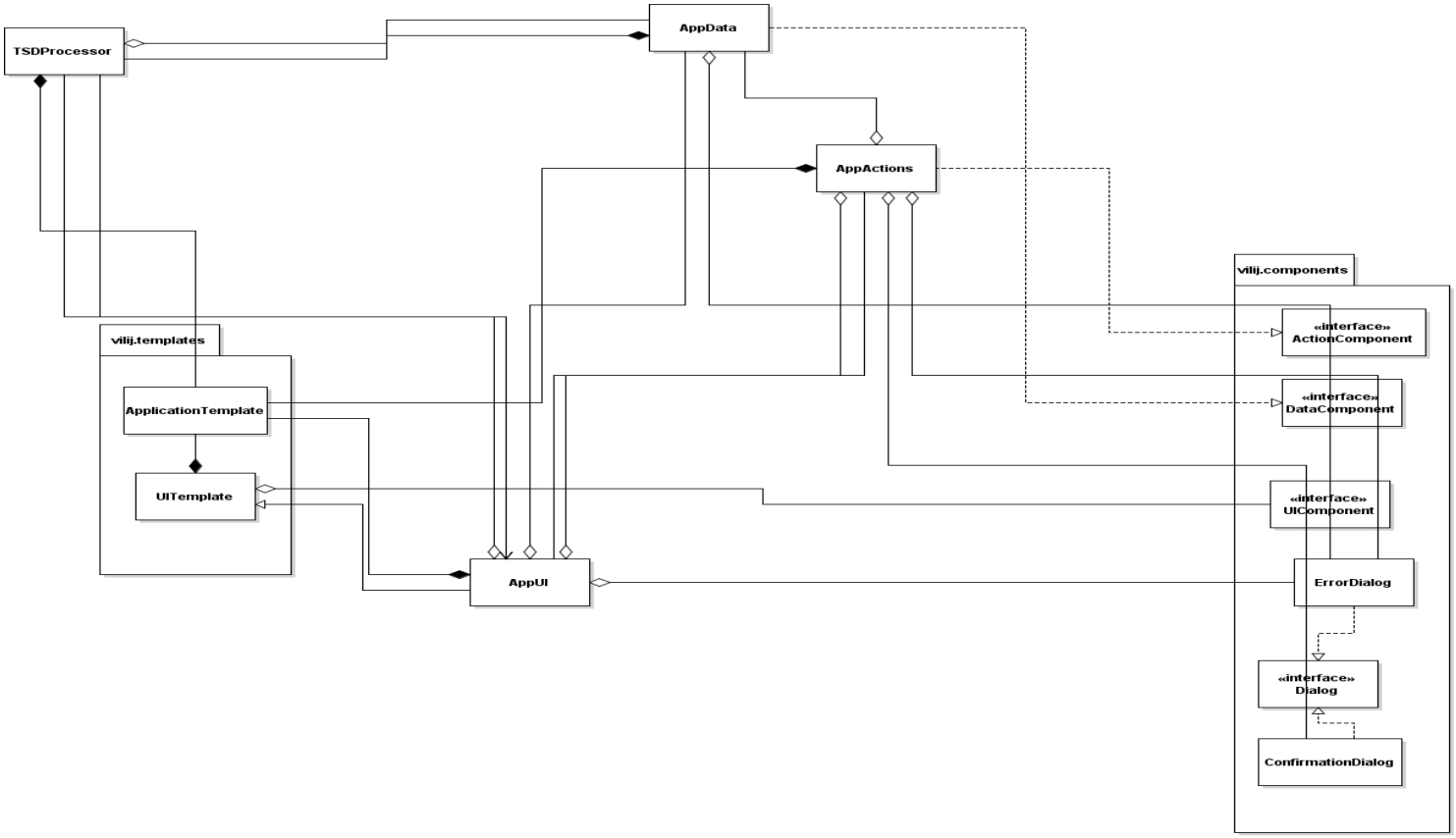
Figure 3.1.1: Overview Class Diagram: Relationship between classes, more detail later classes

Figure 3.1.2: System Overview: Description of subsystems' responsibilities and relationships to each other (Zooming in on major components)-

AppUI handles the layout of the application. AppData handles the processing of data such as loading and saving data. It communicates between AppActions and TSDProcessor because any data actions need to be verified and broken down into their specific components such as data points, labels, and names. AppActions handles actions in the gui such as creating new data, exiting the application, taking screenshots, saving and loading data. It must communicate with AppUI to get the data from text areas or to get the chart images. AppActions communicates with AppData. TSDProcessor verifies the data format. It communicates with AppData which is composed of TSDProcessor. The ApplicationTemplate handles interactions between components such as the DataComponent (AppData), the UIComponent (AppUI), and the ActionComponent (AppActions). AppUI, AppActions, AppData, and TSDProcessor are all composed of the ApplicationTemplate. AppActions implements the vilij framework interface ActionComponent. AppData implements the interface DataComponent. AppUI is a subclass and inherits fields and methods from UITemplate. ErrorDialog in vilij is an aggregate of AppUI and AppActions which contain instances of this object. ErrorDialog implements Dialog interface in vilij framework. ConfirmationDialog in vilij framework is an aggregate of AppActions and implements Dialog. Algorithms classes are for constructing algorithms, AppUI contains instances of these classes. Communication classes send information between algorithms module and the AppUI GUI. vilij.settings is used by vilij.templates. Vilij.propertymanager is used by AppData, AppActions, AppUI and TSDProcessor to access data in other classes. Xmlutil is used by PropertyManager classes and template classes in the vilij framework. Further descriptions for each class in subsequent diagrams.

Example: AppUI communicates to AppData through

((AppData)applicationTemplate.getDataComponent.resetExtraLines());



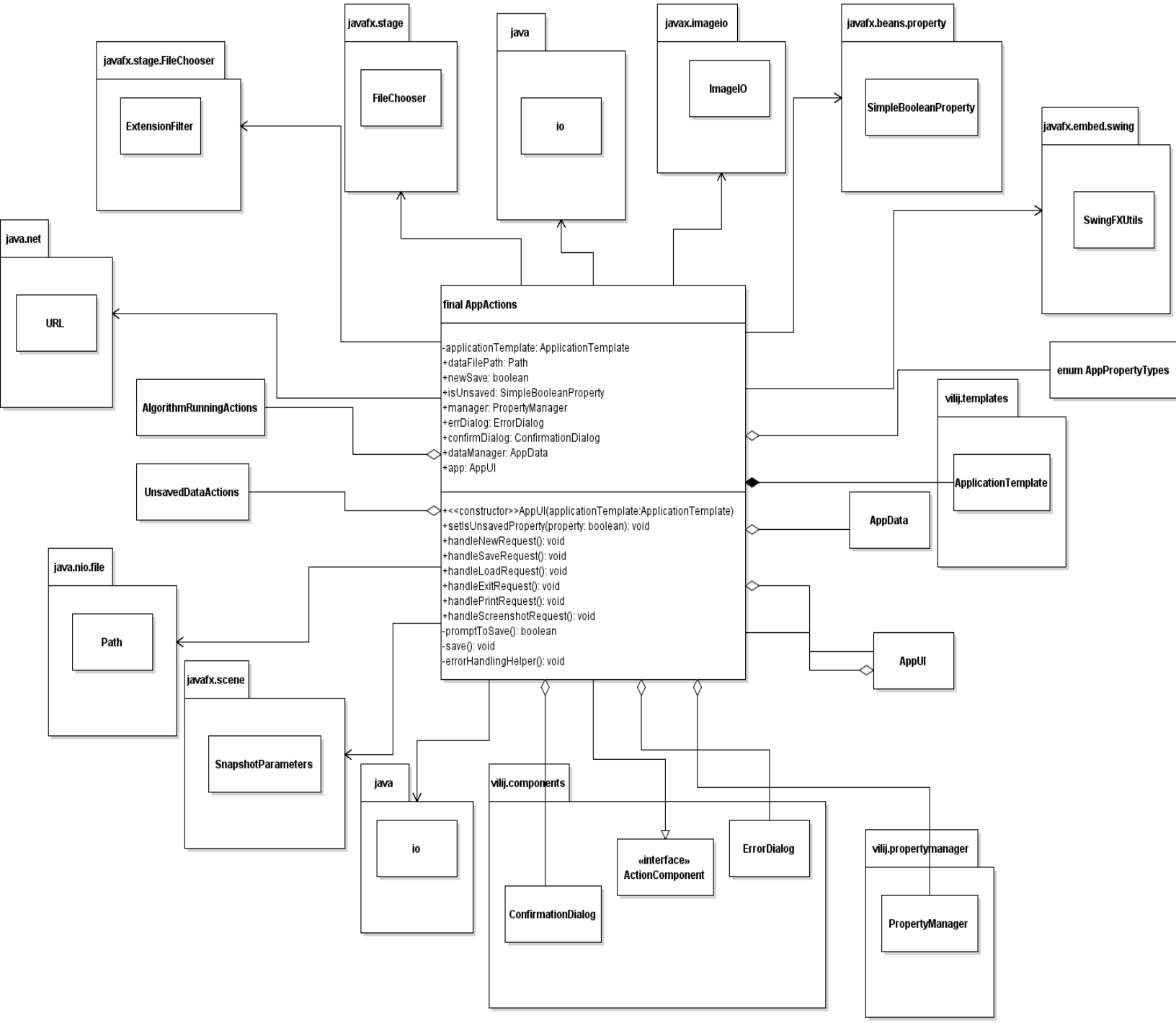


Figure 3.2: Detailed AppActions UML Class Diagram

These classes handle actions inside the UI such as loading data, saving data, saving images, and handling new data.

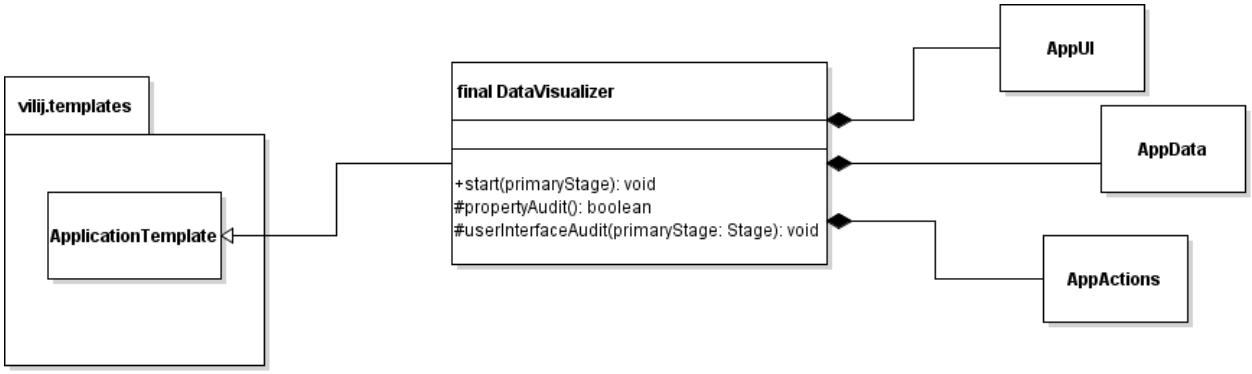


Figure: 3.3: Detailed DataVisualizer UML Class Diagram

This class contains the start method for the application. It is the starting point of the application.

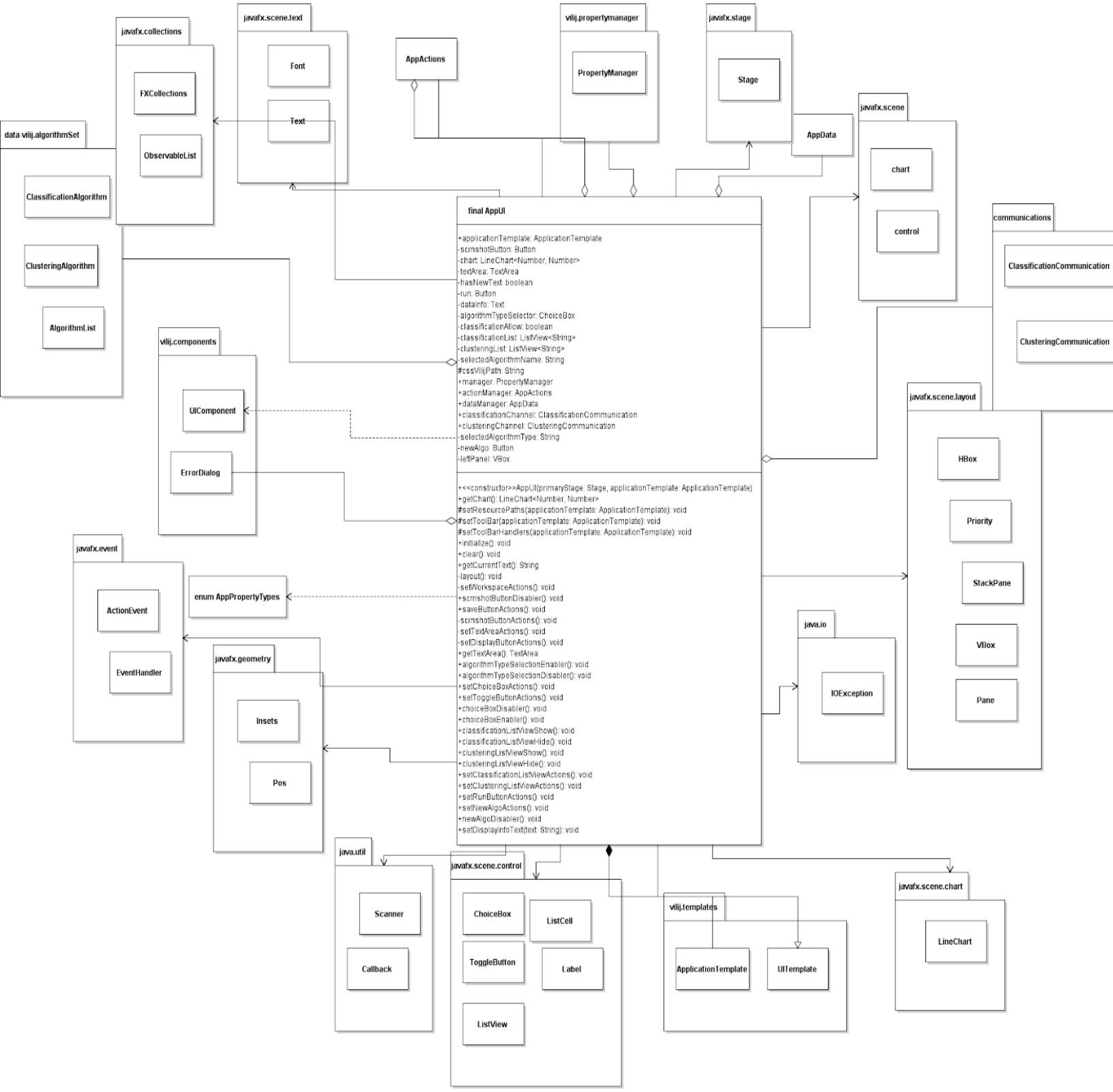


Figure 3.4.1: Detailed AppUI UML Class Diagram This class handles the layout of the UI such as the placement of the chart, the text areas, etc. It also is where communication with the communication interface occurs.

Figure 3.4.2: Closer Look at AppUI: Methods added for handling algorithm controls

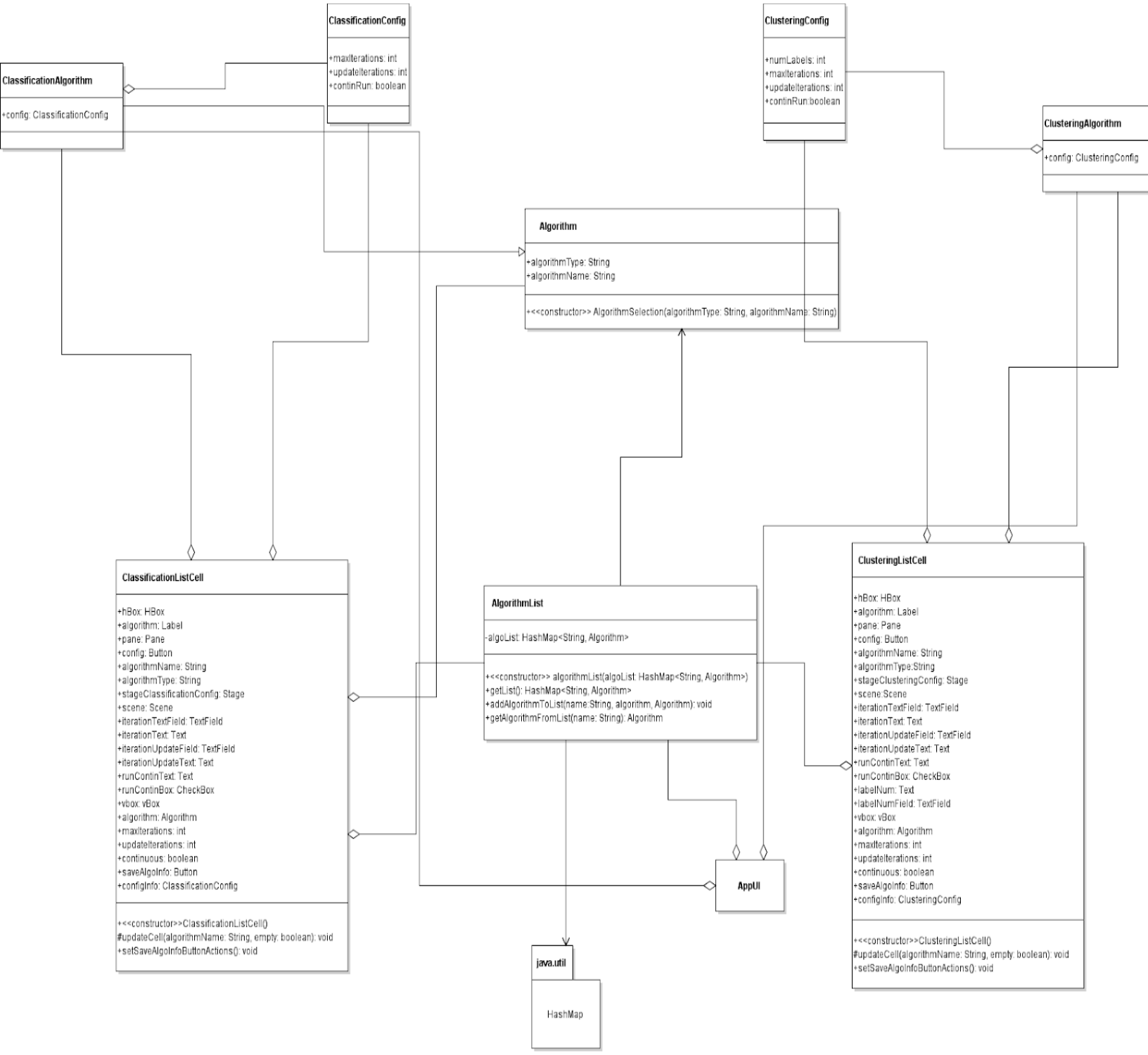


Figure 3.5: Detailed data-vilij.algorithmSet Classes and ListCell UML Class Diagrams

These classes are used for constructing the algorithms either as a classification algorithm or a clustering algorithm. The algorithm list is for storing the algorithms that have already been constructed with configuration settings. The custom cell classes are for adding configuration buttons inside the cells for each type of algorithm. Config objects store information added into the dialog about each algorithm during runtime.

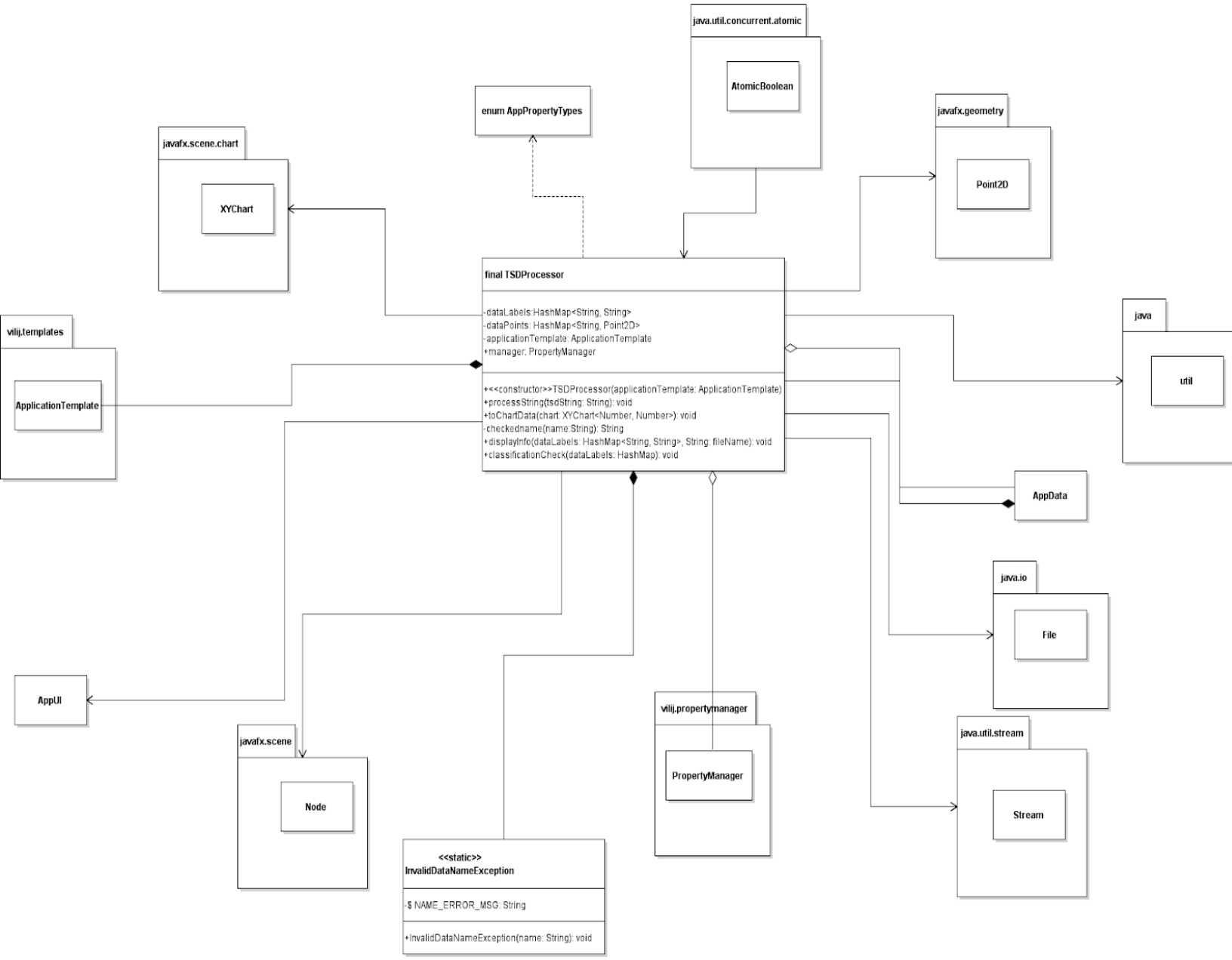


Figure 3.6: Detailed TSDProcessor UML Class Diagram

This class processes the data to determine whether or not the data is valid. This class also is used for checking whether a classification algorithm can be run on the data. It is also used for setting up the information to be displayed about the data, and to prepare the data to appear on the chart.

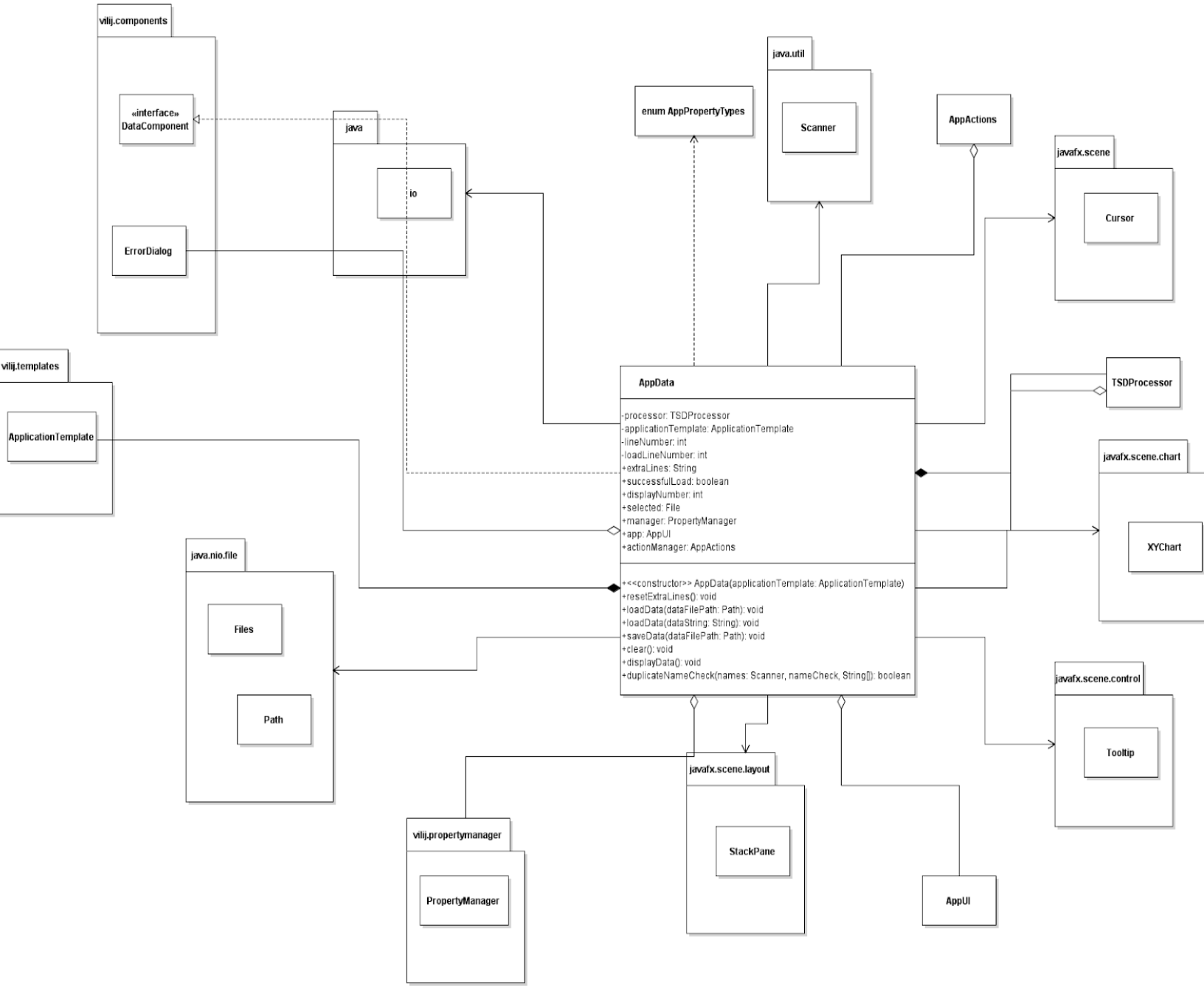


Figure 3.7: Detailed AppData UML Class Diagram

This class communicates between the AppActions class and the TSDProcessor class when loading, saving, and displaying data. This class makes sure validity of data is checked before actions are performed. It is used for reading and writing the data from files.

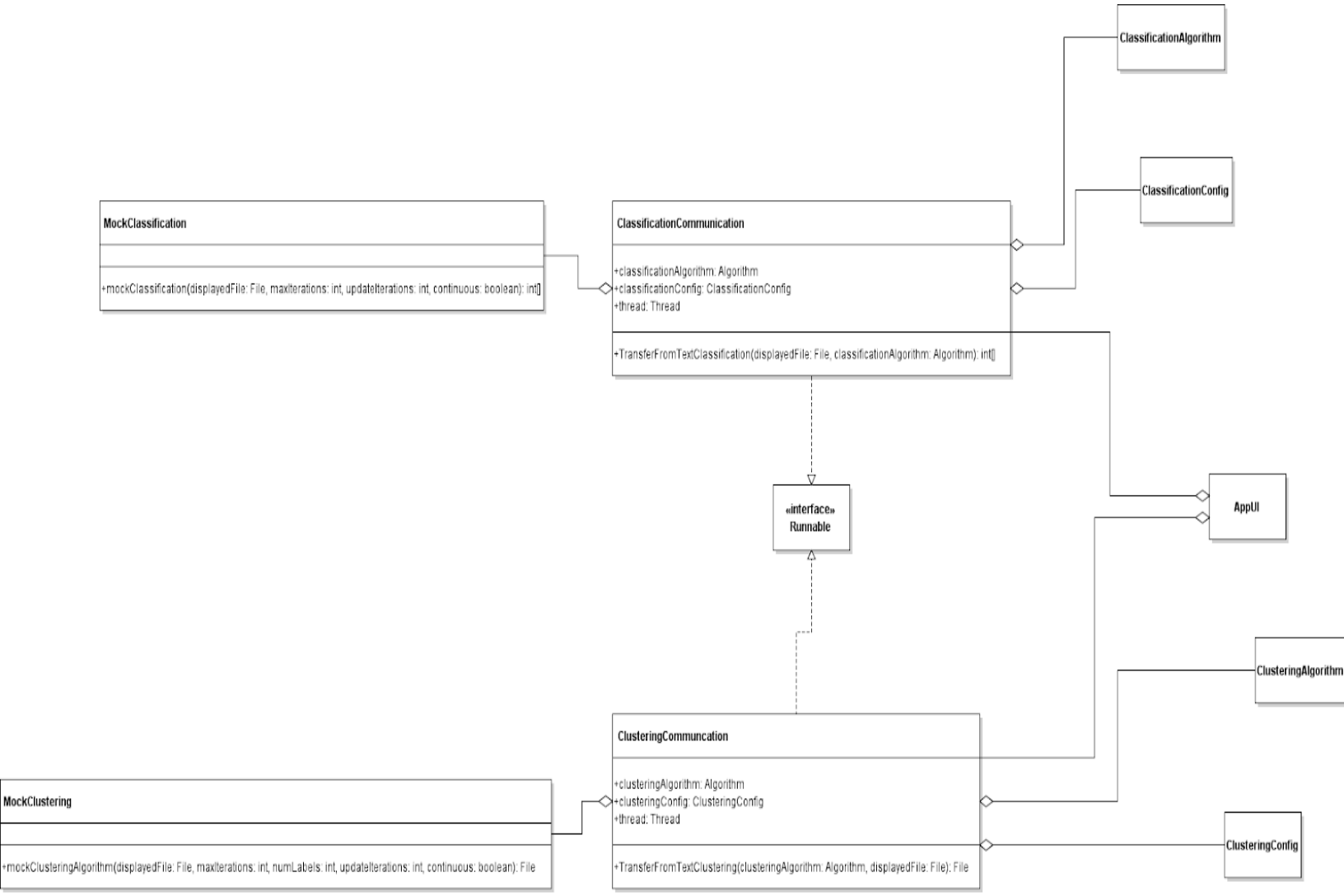


Figure 3.8: Detailed Communication Interface UML Class Diagram

These classes handle the communication between the UI and the algorithms by giving the algorithms necessary information such as the file of data as well as information in the configuration settings such as the maximum number of iterations, the number of iterations before updating, and whether or not the algorithms run continuously.

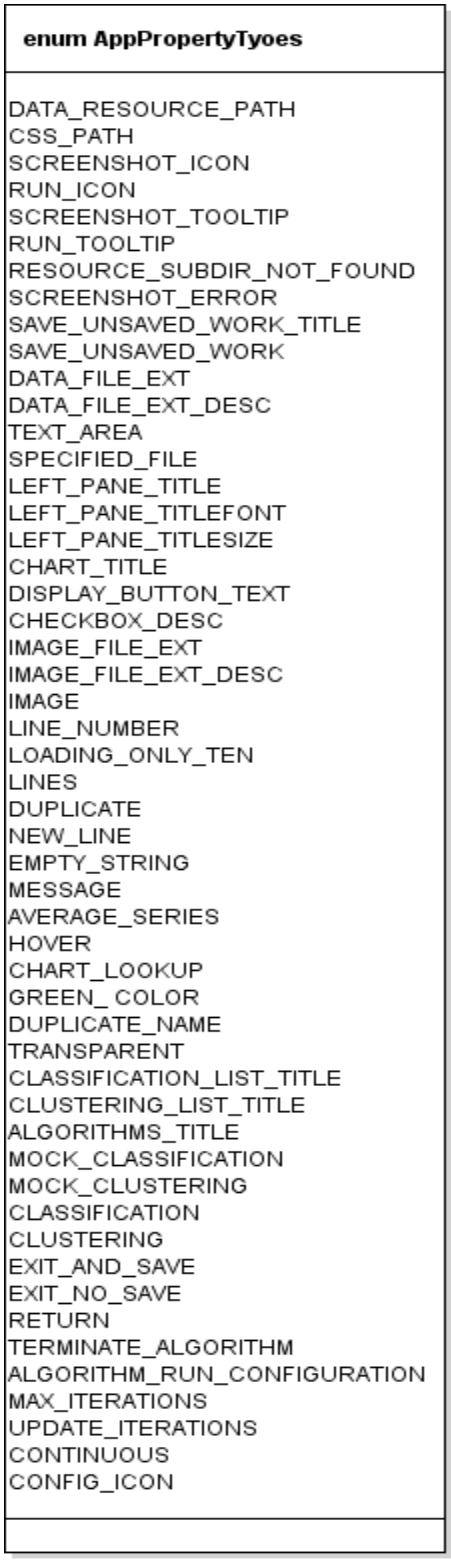


Figure 3.9: AppPropertyTypes Enum

Class contains titles for buttons and controls in application to prevent hard-coding strings.

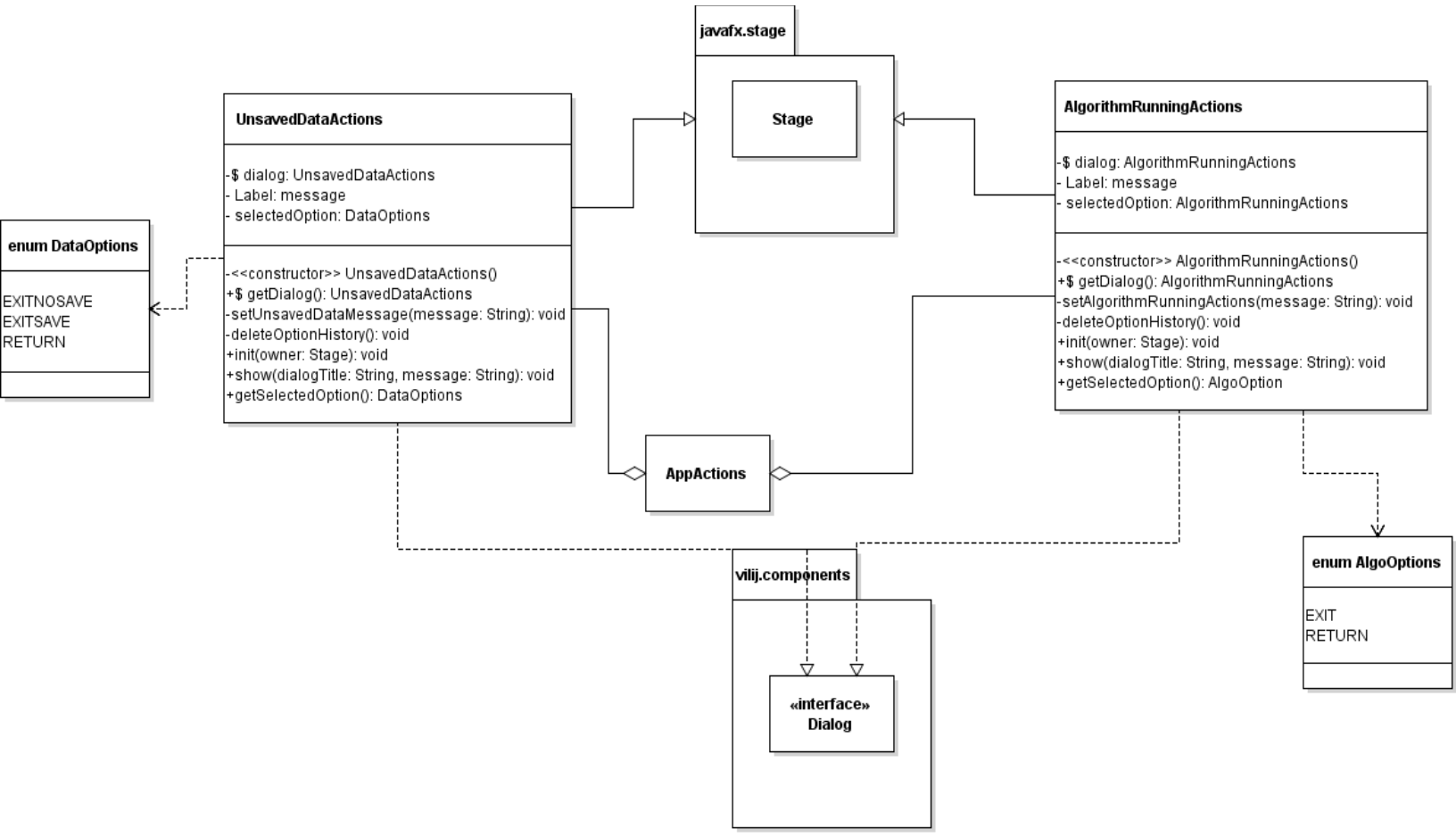


Figure 3.10: UnsavedDataActions and AlgorithmRunningActions Dialogs

These classes are the dialogs that pop up when a user tries to exit the program with unsaved data or the user tries to exit the program while an algorithm is running. AppActions contains instances of these classes in its handleExitRequest() method.

4. Method-Level Design Viewpoint

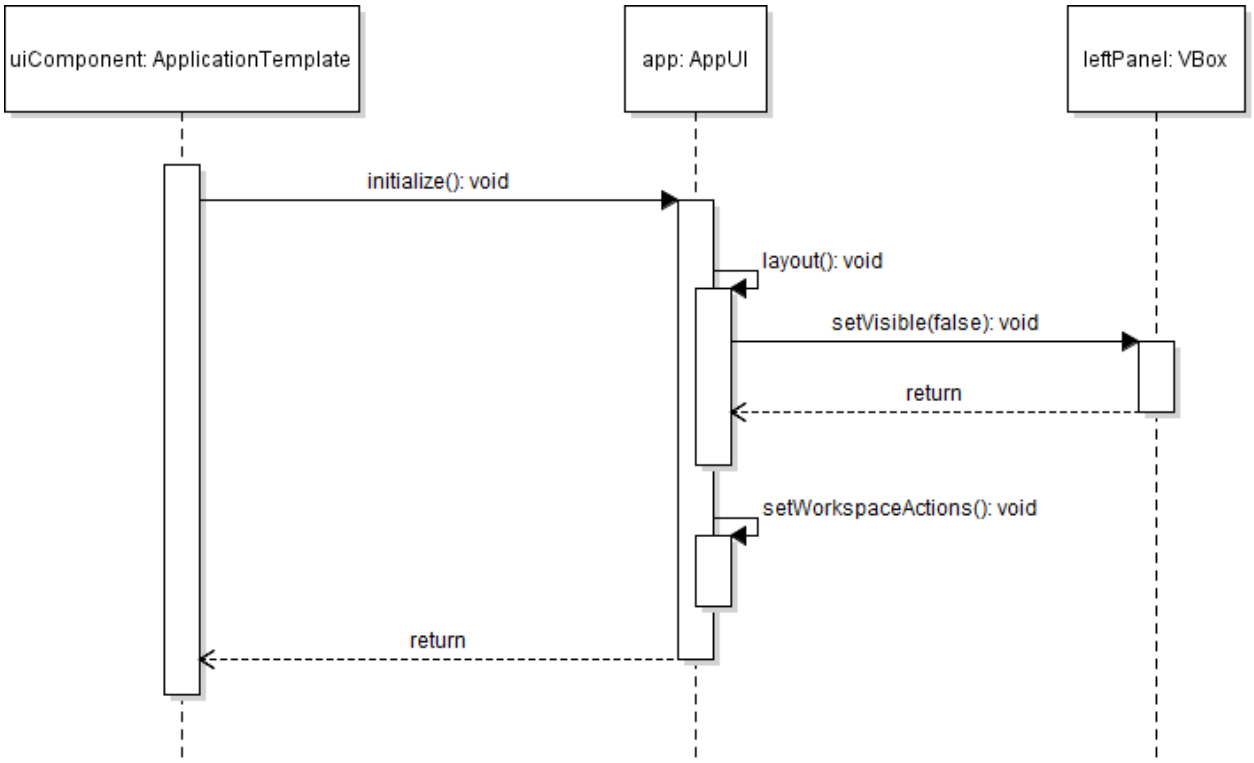


Figure 4.1: Start Application UML Sequence Diagram (Use Case 1)
Upon starting the application, primary window appears containing the toolbar and the empty JavaFX chart. Everything in the left panel is set invisible so that only the chart and the toolbar appears.

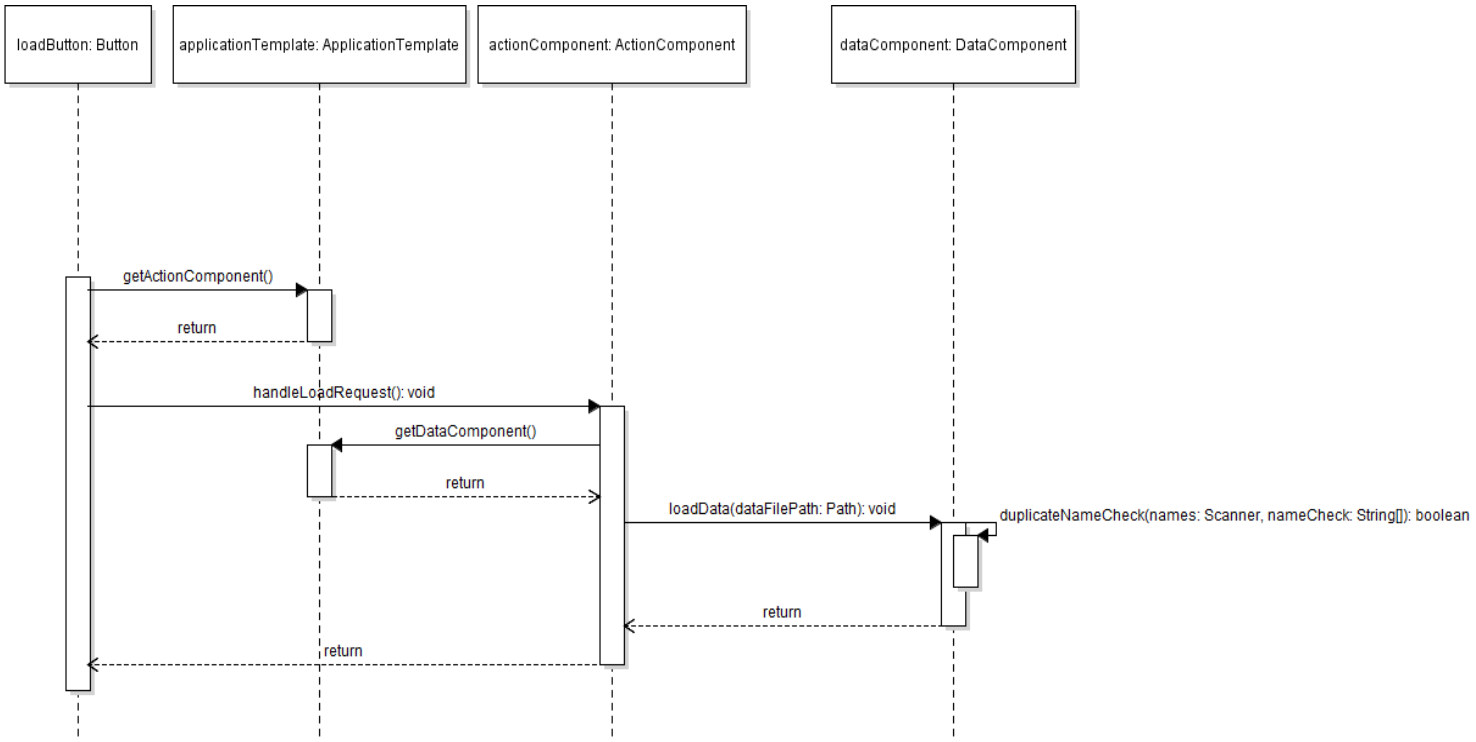


Figure 4.2.1: Load Data UML Sequence Diagram (Use Case 2) [Accessing loadData]

Every time the load button is pressed, the data is checked for validity. If data is valid, information is displayed about the data. The data must then be checked to determine whether or not a classification algorithm can be performed on the data. The algorithm choice box is then enabled. If classification is not allowed and the user chooses classification, an error message will pop up explaining why classification cannot be used for this set of data. The text area is disabled after loading the data.

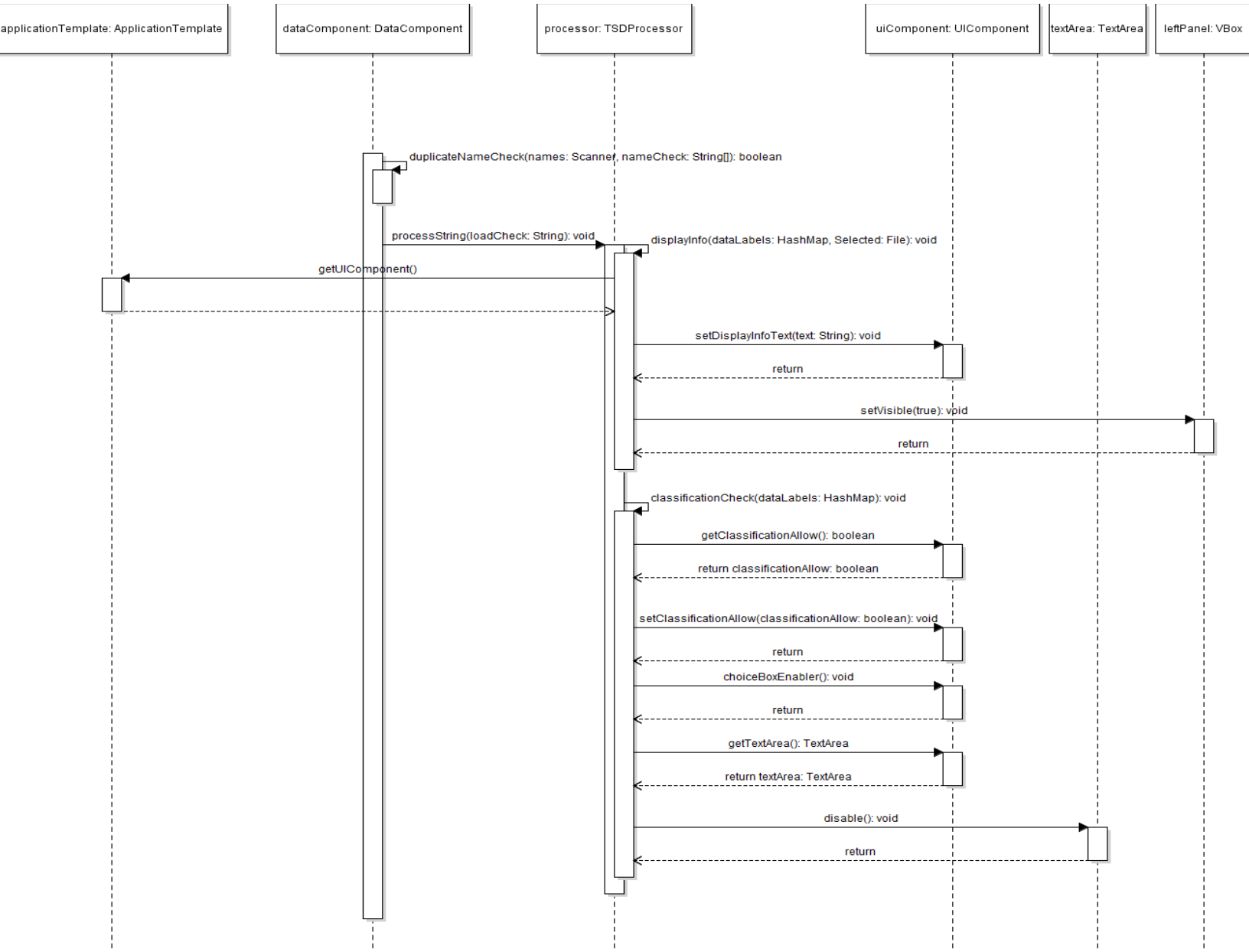


Figure 4.2.2: Load Data UML Sequence Diagram (Use Case 2)

Upon calling the loadData function, data validity is checked. If data is valid, information about the data is displayed, the data is checked for whether or not classification algorithms can be run on it, and the text area is disabled when the choice box for algorithm type is enabled.

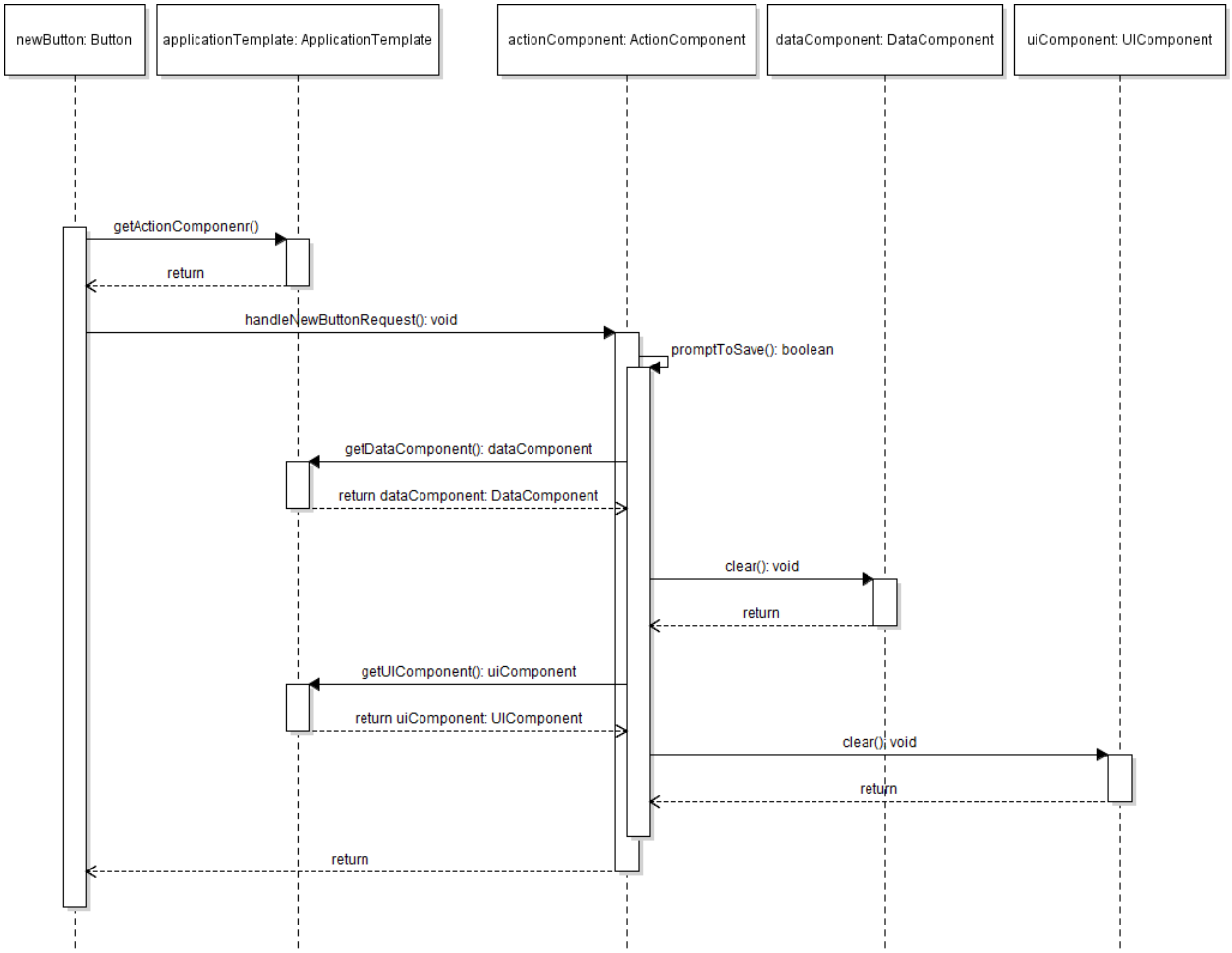


Figure 4.3: Create New Data UML Sequence Diagram (Use Case 3). Part of Use Case 3 Involving Starting a New Data Input.

When the new button is pressed, the user will be prompted to save any unsaved data. The data is then cleared from the text area, the chart, and the data component.

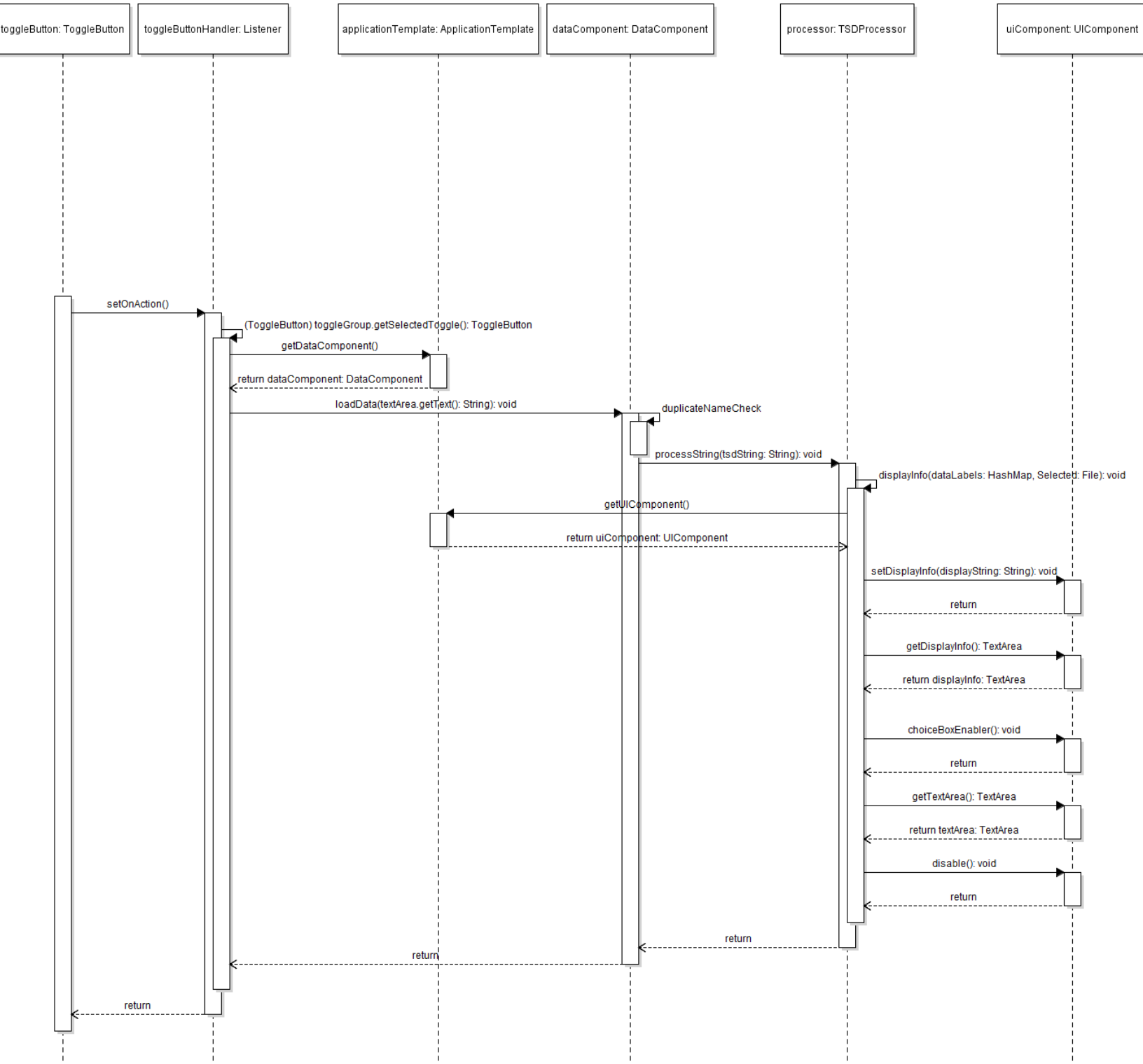


Figure 4.4: Setting New Data With Toggle. (Use Case 3) [case where toggled to done.]
If the toggle button is switched from edit to done, the data validity will be checked, data will be displayed, choice box will be enabled, and the text area will be disabled.

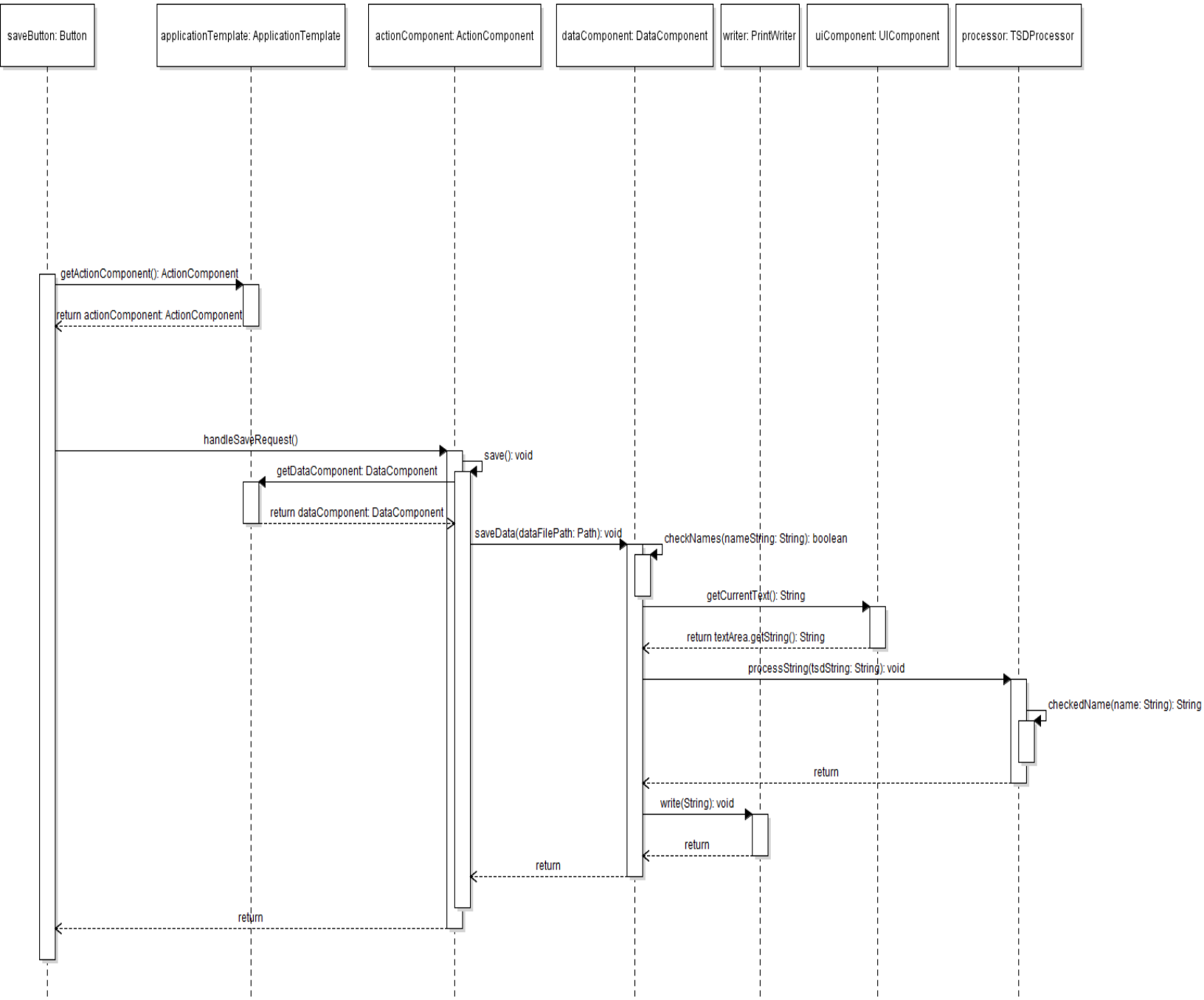


Figure 4.5: Save Data to Prior Saved File Once Save Button is Clicked. (Use Case 4)
If the save button is pressed, the data will be checked for validity and the data will be written to a file.

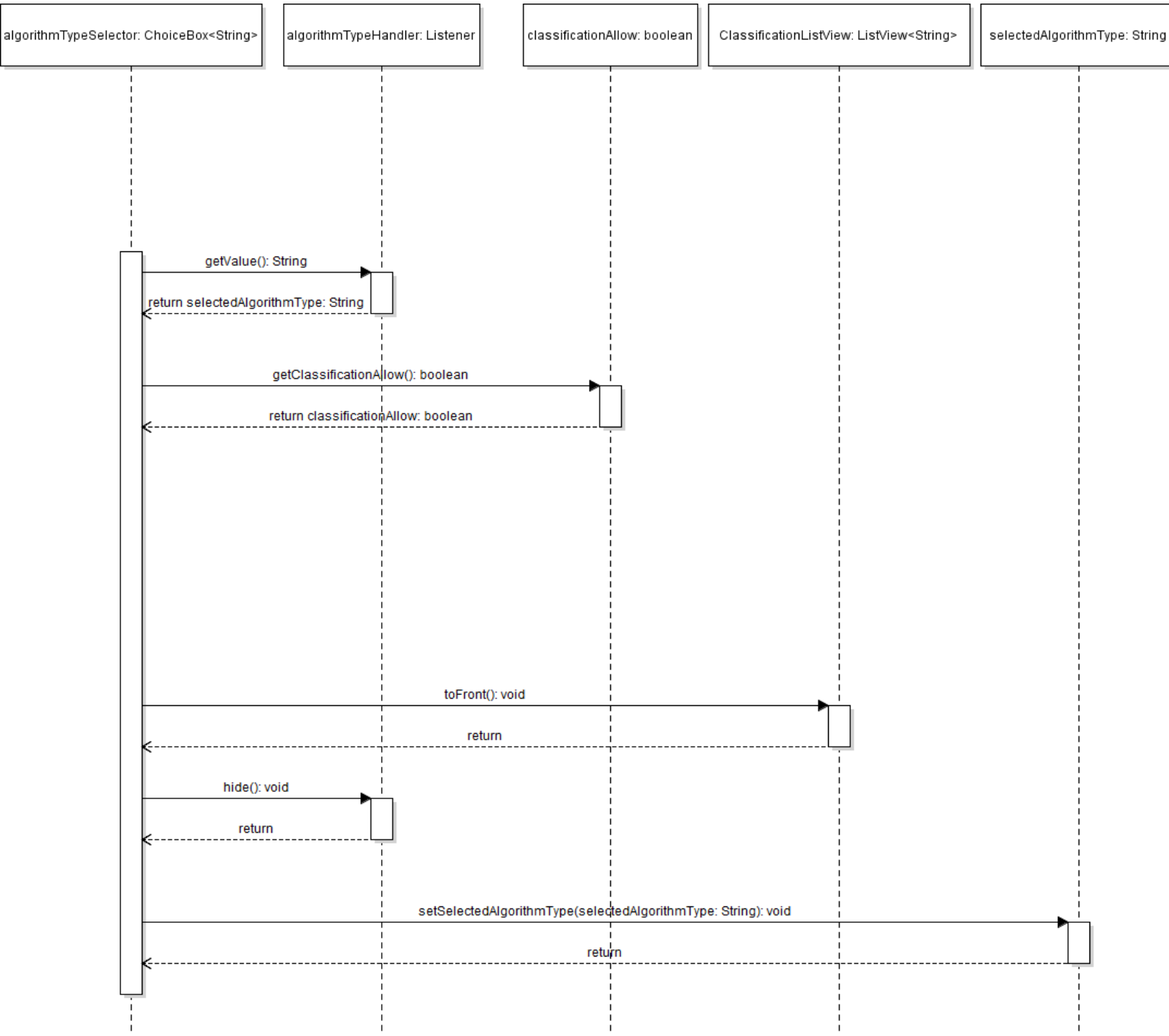


Figure 4.6: Select Classification for Algorithm Type (Use Case 5)

If classification is chosen from the choice box, the application will check whether or not classification is allowed for the current set of data. If classification is allowed, then the algorithm type will be saved, the choice box will be hidden, and the classification list view will be sent to the front of the StackPane.

Figure 4.7: Algorithm Selection (Use Case 6) Algorithm name is selected from the list view.

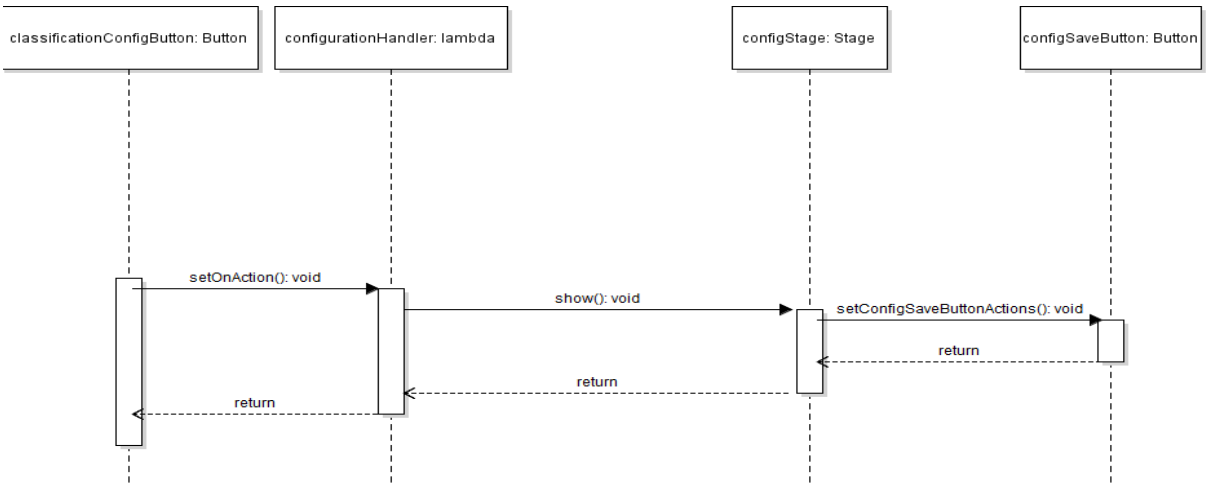


Figure 4.8: Algorithm Run Configuration (Use Case 7) The Configuration Window Appears with text fields for entering maximum iterations and update iterations as well as a check box for

choosing whether or not to continuous run after clicking the configuration button. The settings are saved for this algorithm once the save settings button is clicked shown in the next diagram.

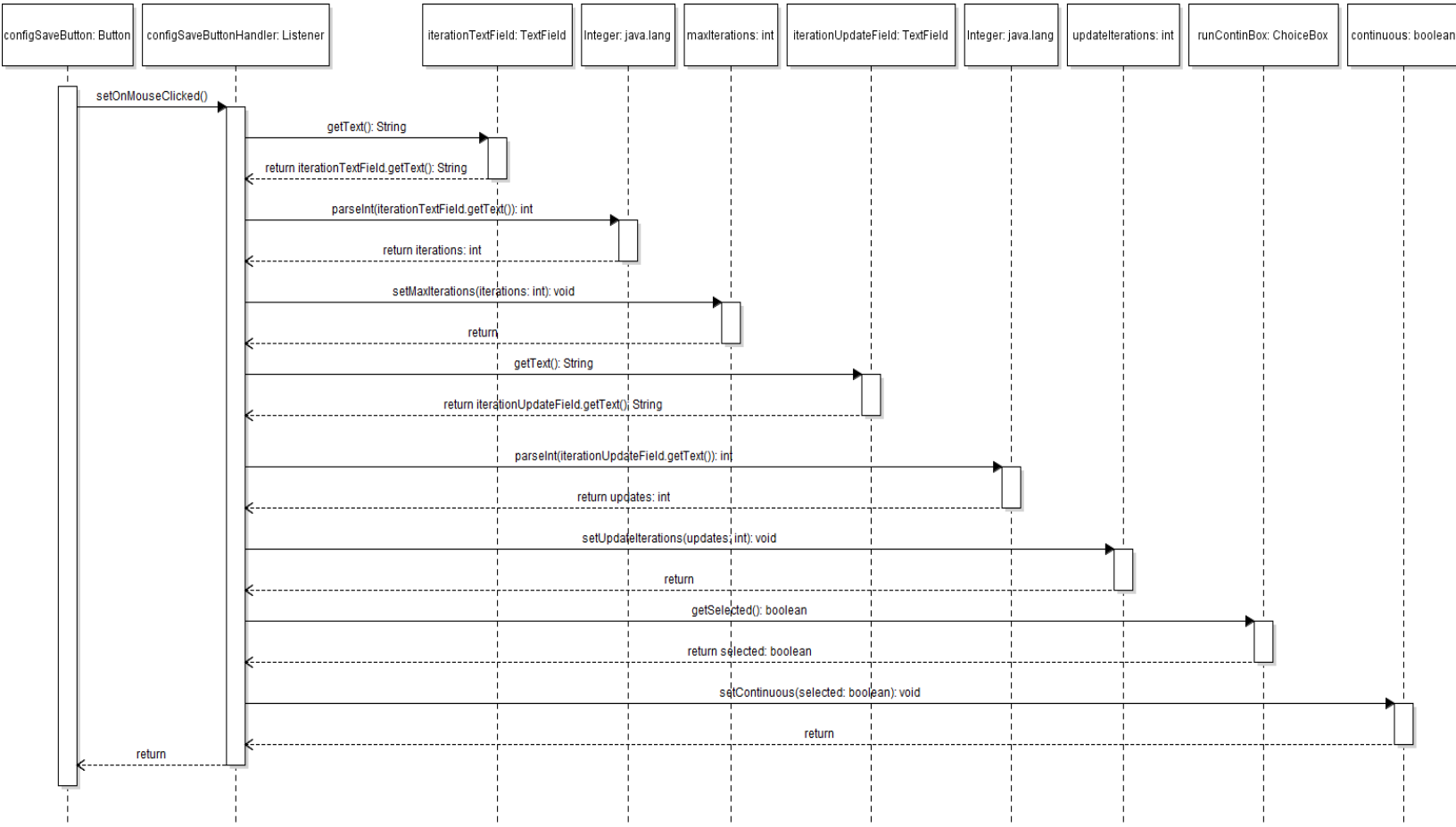


Figure 4.9.1: Algorithm Run Configuration Saving Settings (Use Case 7)

If the configuration icon is selected and settings are inputted. The settings are saved to an algorithm once the configuration save button is pressed.

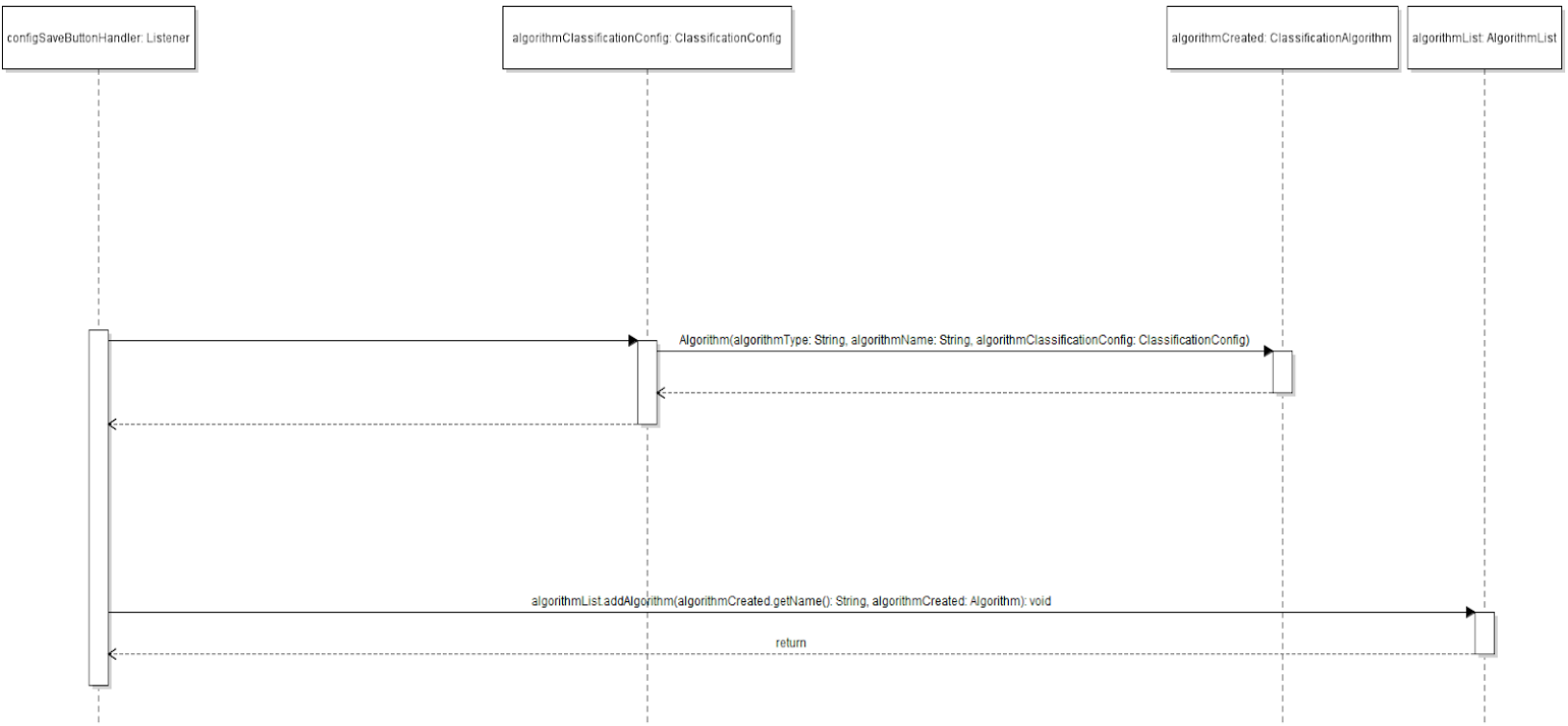


Figure 4.9.2: Algorithm Run Configuration Saving Settings Extension [Save configuration to chosen algorithm] (Use Case 7)

The settings saved in the configuration for classification algorithms object will then be saved inside an algorithm of the chosen type and name. The algorithm is stored in an algorithm list which is a HashMap. The algorithm is found from the list using its key which is its name.

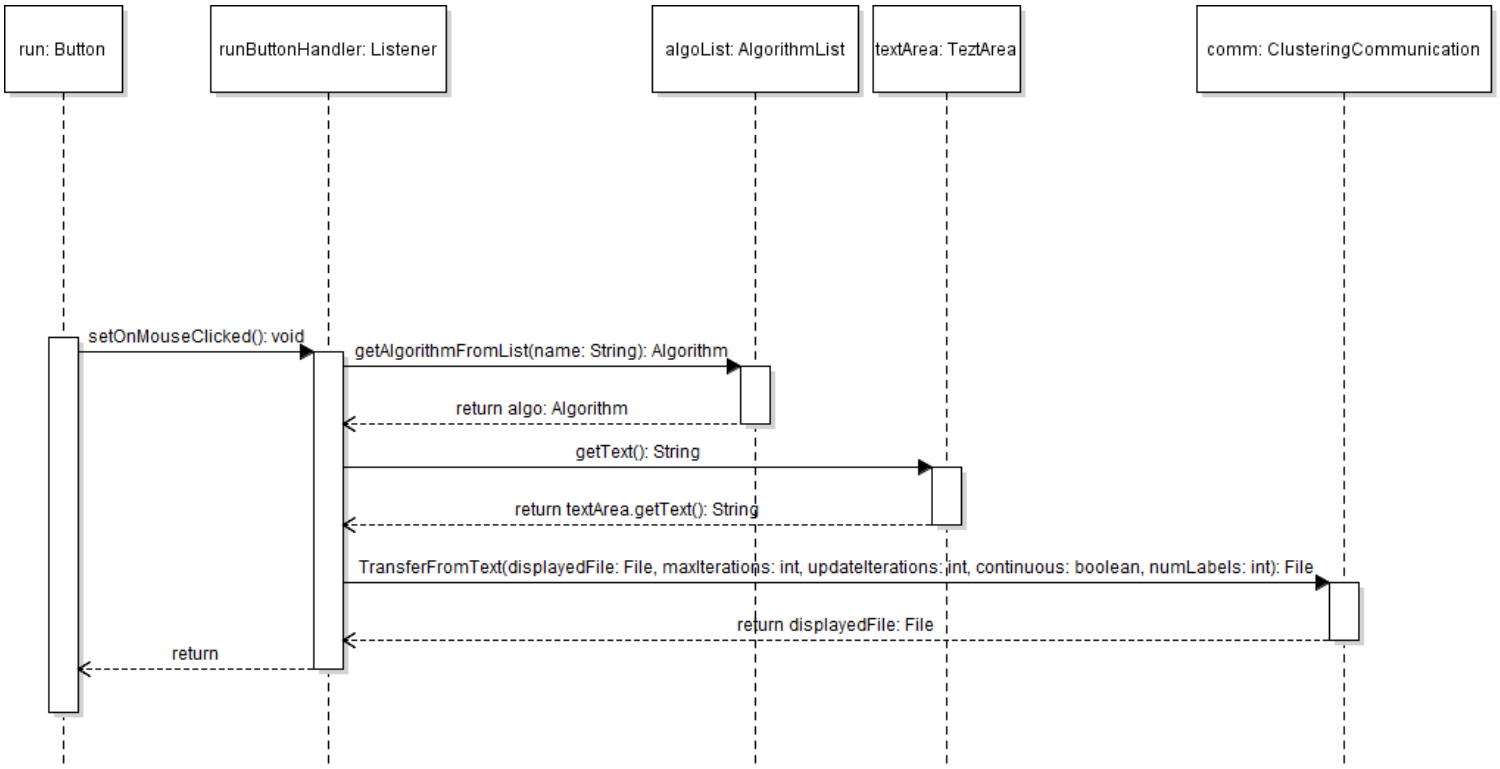


Figure 4.10.1: Algorithm Running (Use Case 8) [Reaching Communication Interface]

When the run button is clicked, the algorithm is searched for in the list. If the algorithm is not found in the list, a message will pop up explaining that the user must enter configuration information before running the algorithm on the data. The data is then taken from the text area, stored in a file, and sent to the communication interface with the algorithm object. The data in file format is then sent to the algorithm along with information in the algorithm configuration object.

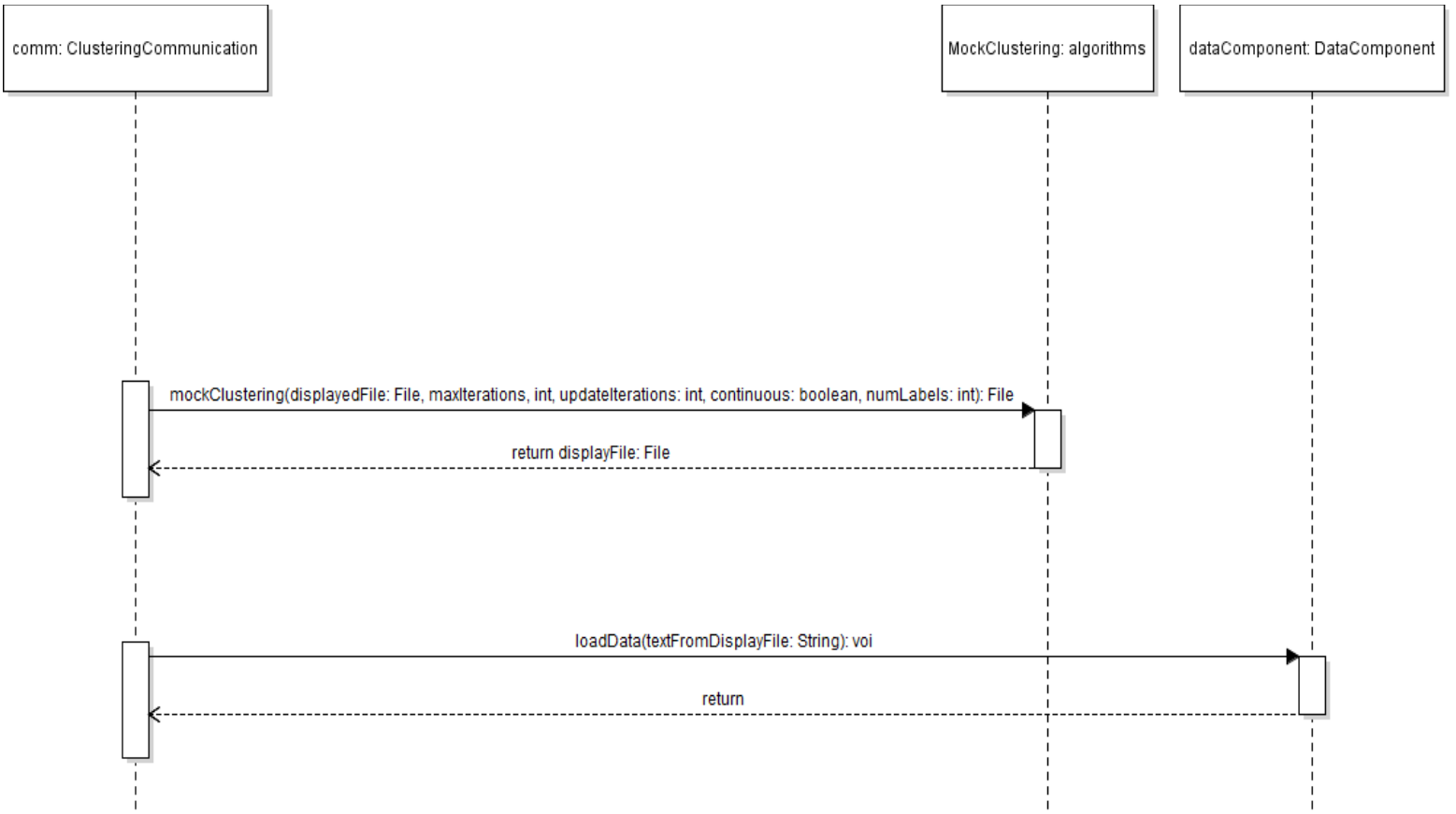


Figure 4.10.2: Algorithm Running (Use Case 8)

Transferring the file from the communication interface to the algorithm module. For clustering algorithms, the tsd file is given to the algorithm as well as the max iterations, the update iterations, the number of labels, and whether or not the algorithm runs continuously. The file labels are changed by the algorithms and returned to the gui in string form to be displayed on the chart without altering the text area. If the algorithm does not run continuously, then the max iterations will be decremented by the communication module when the update iterations are up. The gui component will keep returning to the communication module with the file until the maximum iterations are zero. The same occurs for classification algorithms except that number of labels are not needed and an int array is returned with coordinates of the line that are then displayed on the chart. When maximum iterations is decremented to zero, the algorithm will stop running completely and it will return the values it has computed.

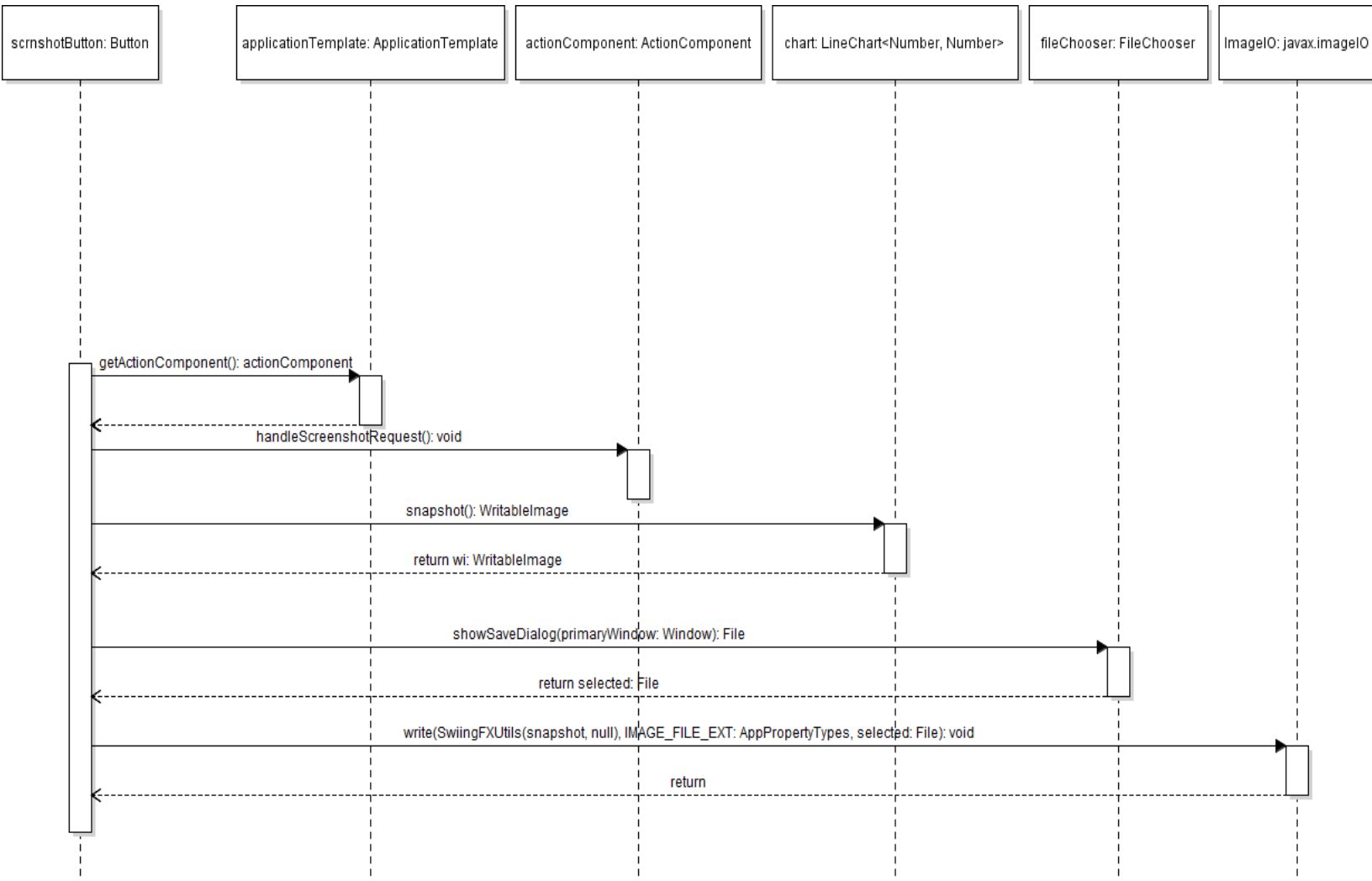


Figure 4.11: Export As Image (Use Case 9)

Once the screenshot button is pressed (which means there must be data in the chart), then a snapshot is taken of the chart area and written to a file to be saved.

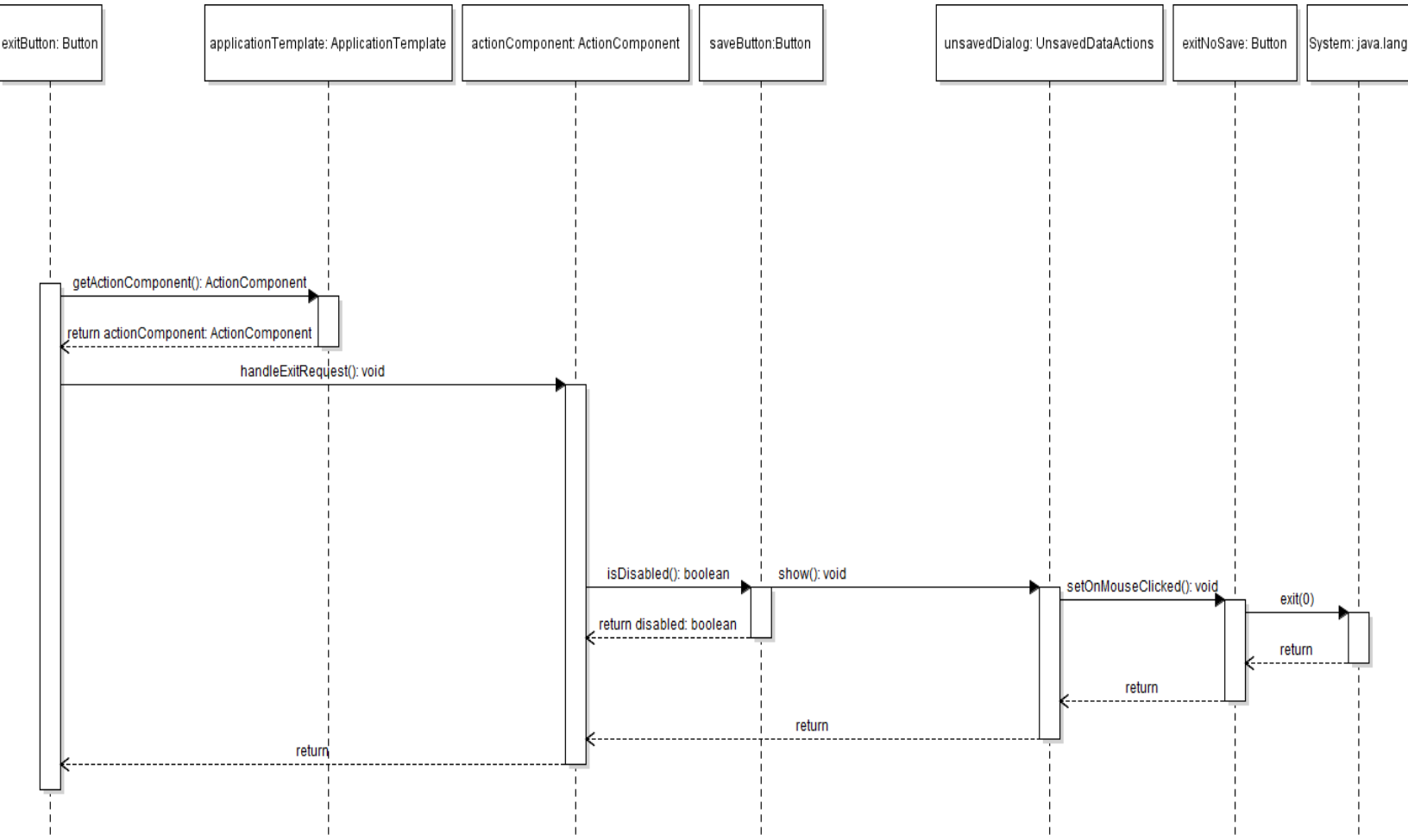


Figure 4.12: Exit Application (Use Case 10)

The exit button can be clicked at any point during the running of the application. If there is unsaved data in the text area, a prompt will ask the user whether they want to return to the application, save the data and exit, or exit without saving. Application also provides a prompt if algorithm is running to determine whether or not user wants to return to application or terminate algorithm and exit.

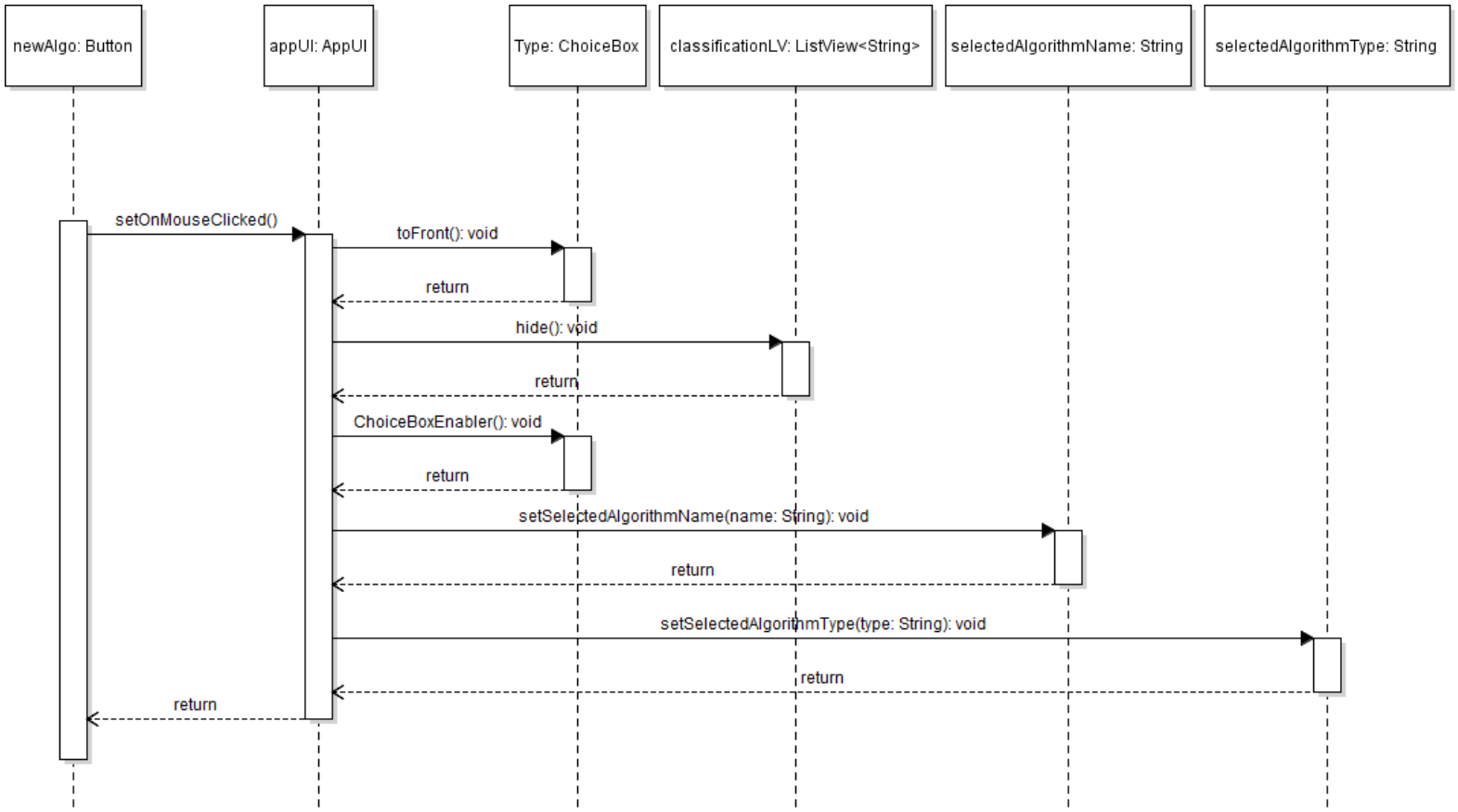


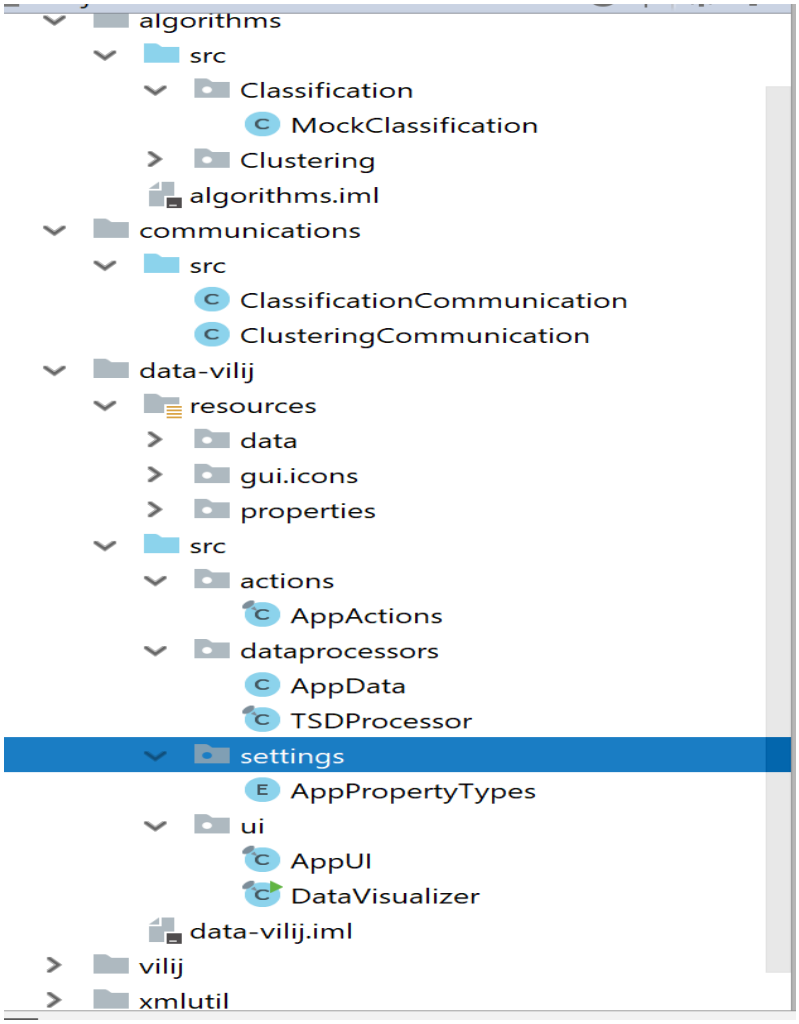
Figure 4.13: Selecting A New Algorithm:

If new algorithm button is enabled, such as when another algorithm is NOT running, then the algorithm type choice box will be sent to the front of the stack pane and enabled if the button is clicked. Also, the selected algorithm type and name will be set to the empty string.
Without this option, middle use cases would not be able to be performed more than once.

5. File Structures, Data Structures, and Formats

5.1 File Structures

The Vilij framework and xml utilities are provided inside of the vilij package and the xmlutil package. These files are all written in the Java target language. Java 1.8 update 162 is used for the project SDK, and project language level 8: Lambdas, type annotations, etc. must be selected in the Project Structure. This project can be deployed on operating systems including Windows 10, Mac OS X 10.11 or higher, and Ubuntu 16.04 LTS or higher. These packages must be included in a data-vilij project and will be deployed with the data-vilij package all inside of a data-vilijFinal package. All necessary icon files for the toolbar buttons must be included. Figure 5.1.1 depicts the file structure that must be used for launching the application. The application can be executed by running the DataVisualizer java file inside of the ui package which is inside of the src package.



The data-vilij resource folder contains a data folder for the tsd-format data files, a gui folder with an icon folder containing the gui icons for the screenshot button and the run button. The properties folder contains the xml files such as the app_properties.xml file which contains titles for different buttons, error message titles, error messages, resource paths, and application specific parameters.

Figure 5.1.1: Data-ViLiJ File Structure

Image of File Structure of project in IntelliJ IDEA

5.2 Data Structures

The processString() method inside the TSDProcessor class converts a string representing an instance into a data structure that stores the constituent parts of an instance. The string is parsed to get the string before the first tab which is the name of the instance, and to get the string before the second tab, which is the label of the instance. The name string is checked to make sure that it starts with @ symbol. The name and label strings are stored in a HashMap called dataLabels which stores the name string as the key and the label string as the value. The remaining portion of the string before the newline character is then split at the comma into two separate strings stored in a list. If the string does not contain a double followed by a comma followed by a double, then an error will be thrown. The first and second string are then both converted from a string to a double to get their double values. These values are then constructed into a 2D point using a Point2D object where the first string contained the x-value and the second string contained the y-value. The points are contained in a HashMap called dataPoints where the key is the name string and the value is Point2D. An instance of data is stored as a data structure consisting of two HashMaps. One HashMap called dataLabels containing the name string as the key and the label string as the value, and another HashMap called dataPoints containing the name string as the key and the Point2D as the value.

5.3 Formats

This section describes the format of files such as the app_properties.xml file, the data-vilij.css file, and the tsd files data files to be loaded into and saved by the application. Figure 5.3.1 depicts how the xml files should be formatted. Figure 5.3.1 does not show the entire app_properties.xml file. Figure 5.3.1 shows a section of the app_properties.xml file. This file contains the names of resource files, the names of resource folders, the names of icon files, the names of tooltip buttons, error messages, message titles, and application specific parameters. Figure 5.3.2 depicts how the css files should be formatted. Figure 5.3.2 does not show the entire data-vilij.css file. Figure 5.3.2 shows a section of the data-vilij.css file. This file is used to format the layout and appearance of the application. It is used for clearing the chart area when empty, for altering the tick marks and axes of the chart, for setting the series lines to certain colors or transparent when necessary, and for setting the size of the text area. Figure 5.3.3 depicts a file that could be loaded into the application. This file must be a tab-separated data file with the appropriate extension. Each instance in this file must end with a new line character. Each instance is tab separated with a label name starting with @, tab, either labeled or unlabeled (null is allowed), and coordinates including an x-value and y-value separated by a comma. Files with invalid data format will not be loaded into the Data-ViLiJ application. Data files will be limited to 2,000 instances, and only one algorithm will be run on a data file at a time.

```
<properties>
    <property_list>
        <!-- RESOURCE FILES AND FOLDERS -->
        <property name="DATA_RESOURCE_PATH" value="data"/>
        <property name="CSS_PATH" value="data-vilij.css"/>
        <property name="RAINBOW_PATH" value="rainbow.png"/>

        <!-- 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."/>
        <property name="SCREENSHOT_ERROR" value="Screenshot Error:"/>

        <!-- 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 name="LEFT_PANE_TITLE" value="Data File"/>
properties
```

Figure 5.3.1: Format of app_properties.xml

Document contains application specific parameters and messages.

```
.chart-vertical-grid-lines {
    -fx-stroke: transparent;
}
.chart-horizontal-grid-lines {
    -fx-stroke: transparent;
}
.axis-tick-mark {
    -fx-stroke-width: 3;
}
.axis-label {
    -fx-text-fill: #4682b4;
}
.axis {
    -fx-font-family: Tahoma;
    -fx-tick-length: 20;
    -fx-minor-tick-length: 10;
    -fx-font-size: 1.4em;
    -fx-tick-label-fill: #914800;
}
.axis-tick-mark {
    -fx-stroke: #637040;
    -fx-stroke-width: 3;
}
.axis-minor-tick-mark {
    -fx-stroke: #859656;
}
.chart-plot-background {
    -fx-border-width: 4px;
    -fx-border-insets: -2px;
    -fx-border-color: black;
    -fx-border-style: solid;
}
.default-color0.chart-series-line { -fx-stroke: transparent; }
```

Figure 5.3.2: Format of data-vilij.css

Css file for styling layout of application

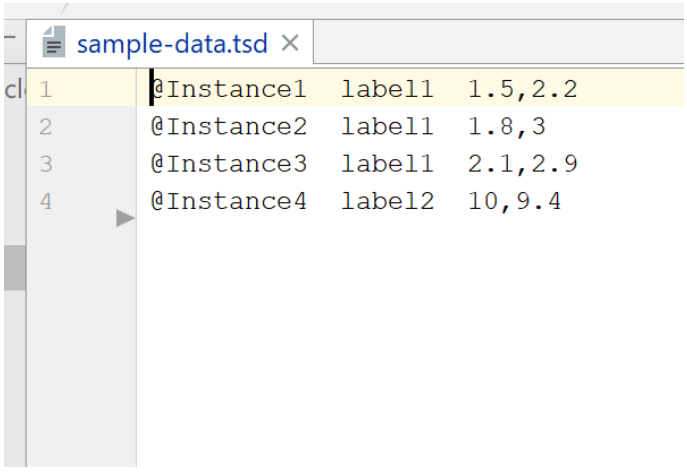


Figure 5.3.3: Format of sample-data.tsd

TSD file containing data instances

6. Supporting Information

Appendix A: Tab-Separated Data (TSD) Format

Input data to this software and any data saved by the user through use of the software must follow the specified tab-separated data format described below. 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

B.1 Classification Algorithms

These algorithms categorize data into one of two known categories. The categories are the labels provided in the input of the data. The algorithm learns by drawing a straight line through the x-y 2-dimensional plane that separates the two labels. This separation is not always possible, but the algorithm will still make an attempt to separate the labels anyways. The output of a classification algorithm is a straight line updated iteratively by the algorithm 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
- the actual (x, y) position of any instance in its input.

The run configuration of a classification algorithm includes the maximum number of iterations (it may stop before reaching this upper limit), the update interval, and the continuous run flag.

B.2 Clustering Algorithms

Clustering algorithms determine patterns based on how data is distributed in space. These algorithms do not need any labels from the data. These algorithms will simply ignore the data labels. These algorithms must know the total number of labels before they start running. The run configuration of clustering algorithms includes the maximum number of iterations (it may stop before reaching this upper limit), the update interval, the number of labels, and the continuous run flag. The output of a clustering algorithm are the input instances, but with its own labels updated iteratively, and shown as part of the dynamically updating visualization in the GUI. The actual position of any instance will not change.