**DATAVILIJTM**

**Software Design Description**

****

**Author:** Neil Opena

Professaur Inc. TM

March 2018

Version 1.0

**Abstract:** This document serves as the software design document for **DataViLiJ** (Data Visualization Library in Java), 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.

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

© 2018 Professaur Inc, which is a made-up company. This document is fictitious and was created for the completion of a Software Design Description assignment for the course CSE 219 (Software Design and Development) at SUNY Stony Brook University.

*No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.*

**1. Table of Contents**

Software Design Description 1

Table of Contents2

1. Introduction3
   1. Purpose3
   2. Scope 3
   3. Definitions, acronyms, and abbreviations3
   4. References4
   5. Overview4
2. Package-level Design Viewpoint5
   1. Software Overview5
   2. Java API Usage6
   3. Java API Usage descriptionsFIXME
3. Class-level Design ViewpointFIXME
4. Method-level Design ViewpointFIXME
5. File/Data structures and formatsFIXME
6. Supporting InformationFIXME

**1. 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.

**1.1 Purpose**

The purpose of this document serves as the blueprint for how the DataViLiJ application will be constructed. It specifies the packages, classes, methods, variables, relationships, and interactions necessary to create the fully functioning data visualization desktop application by containing various UML class diagrams. UML Sequence diagrams represent the interactions of the classes for each scenario of a specified use case.

The intended audience for this Software Design Description is the development team, including the instructor, Professor Eugene Stark, the teaching assistants of the course, and myself, the primary software design and developer.

**1.2 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.

**1.3 Definitions, acronyms, and abbreviations**

1. **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.
2. **Clustering:** A type of AI algorithm that learns to assign labels to instances based purely on the spatial distribution of the data points.
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 two parts, one on each side of the line.
4. **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.
5. **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.
6. **IEEE:** Institute of Electrical and Electronic 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.
7. **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.
8. **Software Design Description(SDD):** A written description of a software product, that a software designer writes in order to give a software development team overall guidance to the architecture of the project. This document, for example, is a SDD.
9. **Software Requirements Specification(SRS):** A description of a software system to be developed. It lays out functional and non-functional requirements and may include a set of use cases that describe user interactions that the software must provide.
10. **Unified Modeling Language (UML):** A general-purpose, developmental modeling language to provide a standard way to visualize the design of a system.
11. **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.
12. **User:** Someone who interacts with the DataViLiJ application via its GUI.
13. **User Interface (UI):** See *Graphical User Interface (GUI).*

**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. Bannerjee, Ritwik. *DataViLiJ TM Software Requirements Specification* Professaur Inc., 2018

**1.5 Overview**

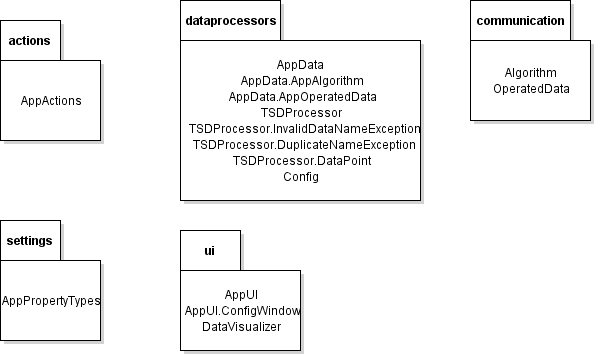
This Software Design Description (SDD) will includes design components that use UML to specify how to build the appropriate technologies for the operational capabilities of DataViLiJ and its UI functionalities and aesthetics, as described in the DataViLiJ Software Requirements Specification. Section 1 of this document includes the Introduction to the document itself and the References used. Section 2 provides the Package-level Design Viewpoint, specifying the packages and frameworks to be designed. Section 3 provides the Class-level Design Viewpoint, using UML Class Diagrams to specify how the classes should be constructed. Section 4 provides the Method-level Design Viewpoint, describing how methods will interact with one another. Section 5 provides deployment information like file and data structures and formats to use. Section 6 provides all the Supporting Information. All the UML Diagrams in this Software Design Description were created using the Violet UML Modeling tool.

**2. Package-level Design Viewpoint**

This design involves the construction of the DataViLiJ application. The Java API by Oracle is heavily relied upon as well the JavaX.XML API. The following sub sections describe how the components of the applications are to be constructed, including how the Java API will be used to build them.

**2.1 DataViLiJ Software Overview**

The DataViLiJ desktop application will be designed with the assistance of the XMLUtilities and ViLiJ frameworks. Fig 2.1. displays the components of the application with classes contained in packages.

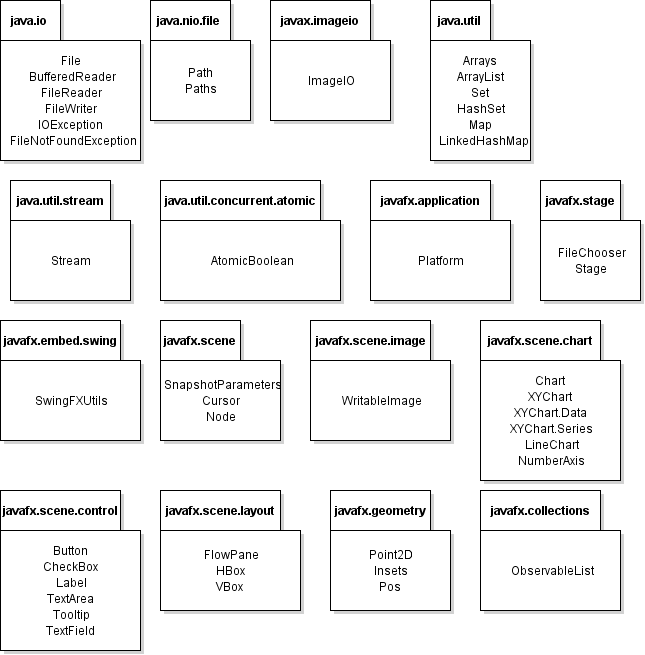
****

**Fig 2.1.** DataViLiJ Package Overview

The actions package contains the class that handles all the primary application actions such as saving and loading. The ui package contains the classes that correspond to the user interaction of the graphical user interface of the application. It also contains the top-level class of the DataViLiJ application itself. The settings package contains the class that provide application specific properties to be applied. The data processors class contains the classes the handle the data aspect of the application. The communication package was created solely to contain the interface for the interaction between the DataViLiJ GUI Module with the Algorithms Module that contain the different algorithms.

**2.2 Java API Usage**

The DataViLiJ application will be programmed using Java, therefore the Java API will be used extensively, the classes of which are specified in Fig 2.2. The java.lang package was not included due to the automatic importation of the package itself by the Java compiler.

****

**Fig 2.2.** Java API Class and Package Usage Overview

**2.3 Java API Usage descriptions**

The tables below (Tables 2.1 – 2.16) describes how the Java API package classes will be used.

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **File** | For determining the path to an external file provided by the user, or for saving to a selected file using a provided file path |
| **BufferedReader** | For reading the input stream from a FileReader |
| **FileReader** | For reading lines from a specified file by the user |
| **FileWriter** | For saving and writing the data from the text area to a specified file |
| **IOException** | For throwing and catching the errors whenever any I/O operation is performed. Specifically, when errors occur during the reading or writing of a file. |
| **FileNotFoundException** | For handling error when user specified file by the user is non- existent |

**Table 2.1.** java.io class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Path** | For reading/writing data to a specific location |
| **Paths** | For obtaining the absolute path the application is directed to. Used for setting the paths of the FileChooser. |

**Table 2.2.** java.nio.file class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **ImageIO** | For saving a WritableImage to a specific file provided by the user |

**Table 2.3.** javax.imageio class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Arrays** | For turning a stream into a list for data checking by the TSDProcessor |
| **ArrayList** | For storing the displayed and hidden data lines of the text area when a file is opened containing a large amount of data lines |
| **Set** | For storing the distinct values of labels for the data lines |
| **HashSet** | For the actual instantiation of a Set object |
| **Map** | For storing the key value pairs of each data point, with its name as the key and its label and data point values as the values themselves |
| **LinkedHashMap** | For the actual instantiation of a Map object, that stores the data in the order of insertion |

**Table 2.4.** java.util class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Stream** | For translating the lines read by the BufferedReader into a stream that is comprehensible for iteration and data manipulation |

**Table 2.5.** java.util.stream class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Atomic Boolean** | For creating Boolean objects that operate atomically, and specifically used for checking if an input string is valid |

**Table 2.6.** java.util.concurrent.atomic class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Platform** | For exiting the application |

**Table 2.7.** javafx.application class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **FileChooser** | For obtaining a specific file through its path that a user specified to for saving and loading data. It is also used to constraint the file extensions displayed when a file chooser window is displayed. |
| **Stage** | For the top-level container of the application |

**Table 2.8.** javafx.stage class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **SwingFXUtils** | For use by ImageIO to save a screenshot into a file |

**Table 2.9**. javafx.embed.swing class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **SnapshotParameters** | For the parameters used by the chart to obtain a snapshot of the current display |
| **Cursor** | For changing the displayed style of the cursor of the application |
| **Node** | For adding an event listener to the average value series line |

**Table 2.10.** javafx.scene class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **WritableImage** | For saving the screenshot of the chart display in an object that can be saved in a file |

**Table 2.11.** javafx.scene.image class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Chart** | For getting chart object to get its specific properties and to use for obtaining a screenshot given a specific data set to display |
| **XYChart** | For displaying the data points into a visual chart with axis |
| **XYChart.Data** | For obtaining the data points belonging to a specific series of a specific data set |
| **XYChart.Series** | For obtaining the specific series or labels of a specific data set |
| **LineChart** | For displaying the data points on the XYChart as nodes with the added functionality of displaying additional lines if specified |
| **NumberAxis** | For instantiating the chart with the default axis values that automatically ranges the chart itself |

**Table 2.12.** javafx.scene.chart class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Button** | For use by all of the different toolbar functionalities. Also for use by the user to display the text area data or the file specific data to the chart in the application |
| **CheckBox** | For use by enabling the read me functionality of the text area, preventing any other input |
| **Label** | For identifying the multiple user interface nodes such as the text area and chart titles, and check box. |
| **TextArea** | For letting the user input data lines, and view the lines of their specified file |
| **Tooltip** | For showing information such as the title of the toolbar buttons or the specific data from the data points in the chart when the mouse is hovered over them. |
| **TextField** | For allowing the user to input data into the ConfigWindow |

**Table 2.13.** javafx.scene.control class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **FlowPane** | For instantiating the layout of the workspace of the application |
| **HBox** | For instantiating the layout where horizontal display is necessary such as the checkbox control and the label corresponding to it |
| **VBox** | For instantiating the layout of the overall input area including the text area and the controls such as multiple buttons corresponding to it |

**Table 2.14.** javafx.scene.layout class usage

|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **Point2D** | For the storing of data point values and for the display within the chart |
| **Insets** | For setting the margins around the overall input area |
| **Pos** | For aligning the nodes in their respective containers |

**Table 2.15.** javafx.geometry class usage

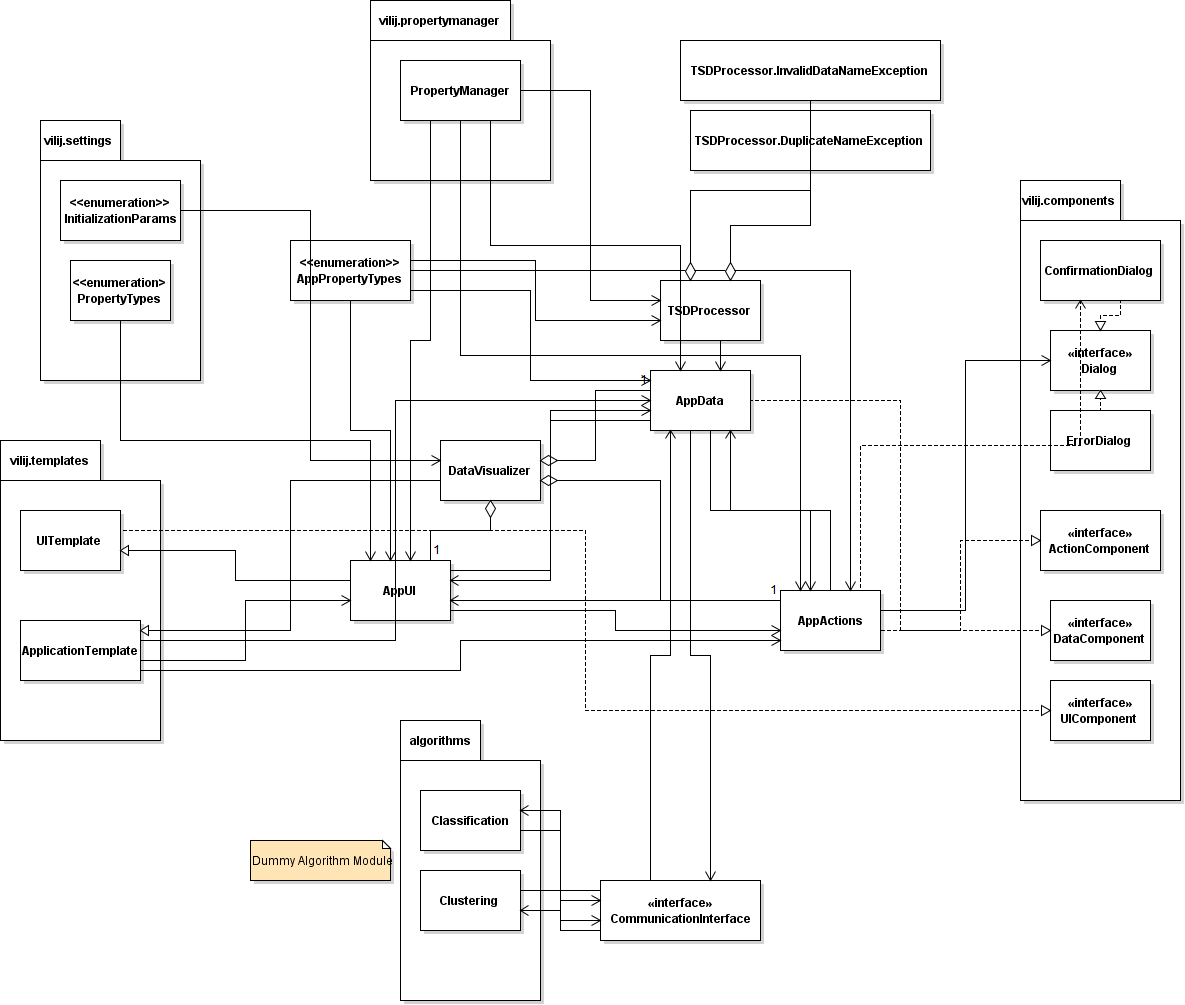
|  |  |
| --- | --- |
| **Class / Interface** | **Usage** |
| **ObservableList** | For obtaining the list of the data nodes in a corresponding series of the chart |

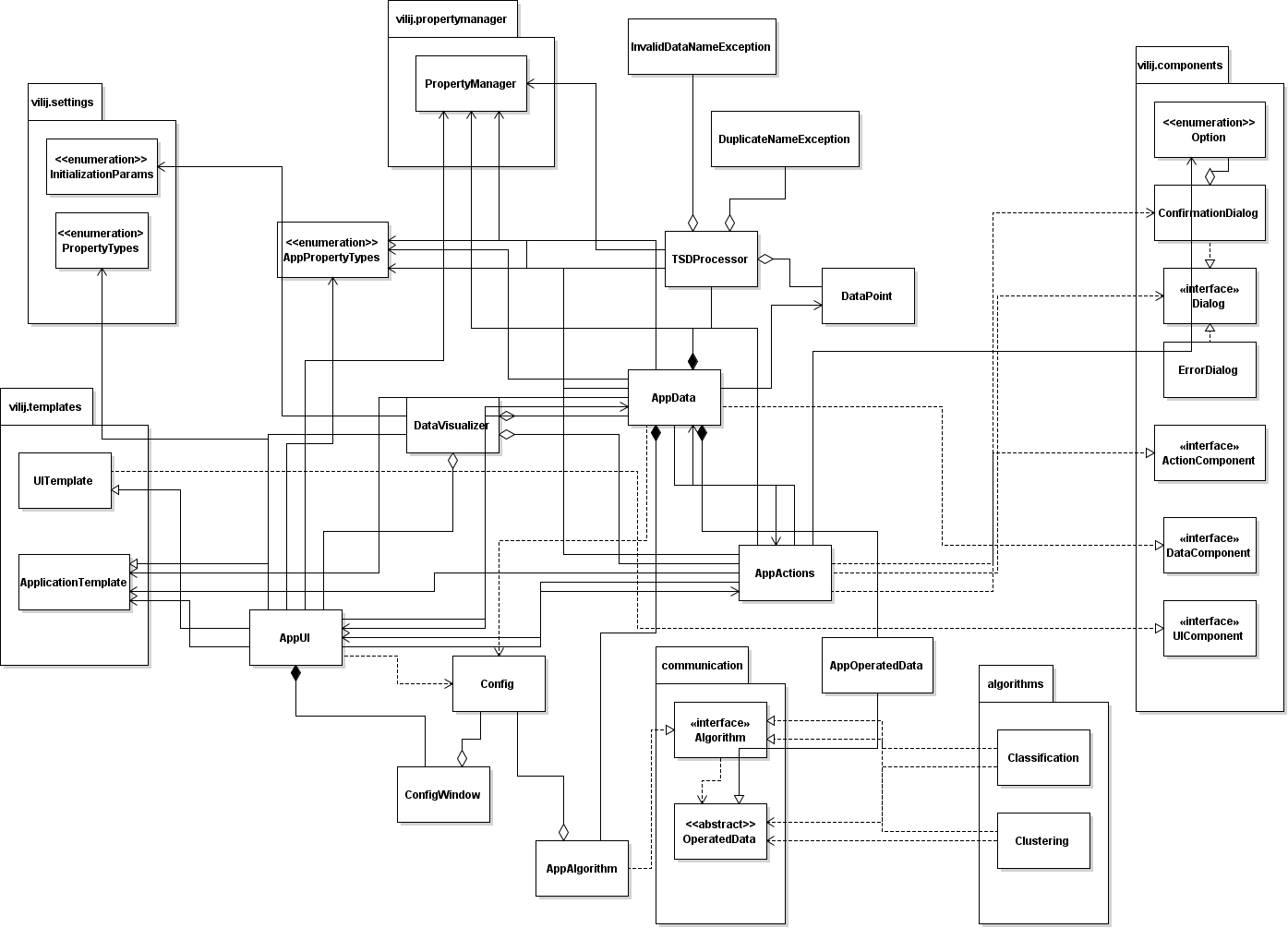
**Table 2.16.** javafx.collections class usage

**3. Class-level Design Viewpoint**

UML Class Diagrams below display the overall design of the DataViLiJ application. The class diagrams of the classes themselves are displayed more in depth following the more general class diagram of the application as shown below. Fig 3.1 displays the overall class interaction of the DataViLiJ classes.

**Note:** Below represents the Data Visualization Module’s GUI component. It represents the interaction between the classes of the DataViLiJ application and the ViLiJ framework. The classes themselves are not placed in their respective packages for clarity of what the actual module is comprised of. For a detailed description of how the DataViLiJ classes were placed in packages refer to Fig 2.1. For the detailed diagrams of each specific class, go to the UML Class Diagram corresponding to it below.

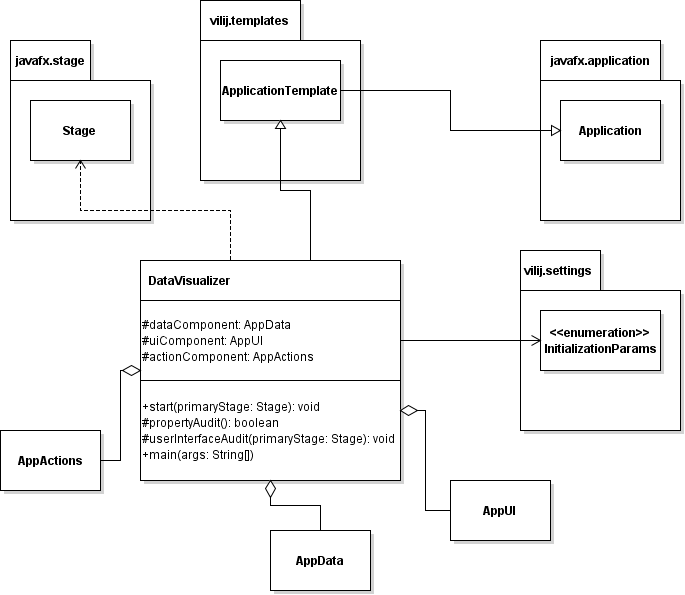


****

**Fig 3.1.** DataViLiJ application Overview UML Class Diagram

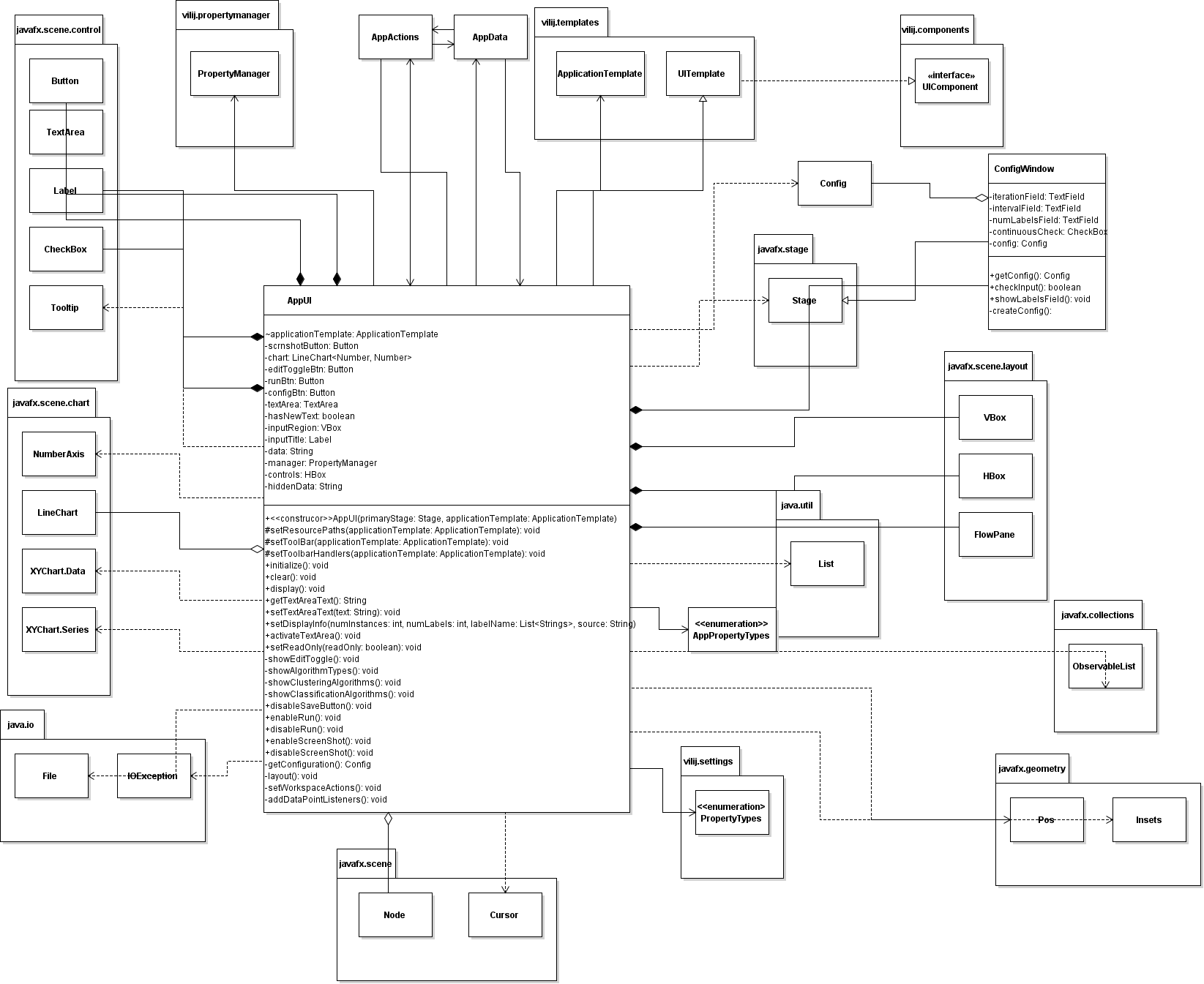
Below shows more specific UML Class Diagrams of the Data Visualization GUI Module.

**Note**: Many of the classes inherit behavior and attributes from their corresponding parent classes in the ViLiJ framework, however, the members of the parent classes are not displayed for clarity since it does not add any value to the reader and will simply take up space. Only members of the specific classes within the DataViLiJ module are presented in the UML diagrams below. The packages within DataViLiJ are also not shown except the communication package to distinguish the proper communication between the GUI component and the Algorithm module.



**Fig 3.2.** Detailed UML Class Diagram of DataVisualizer.java

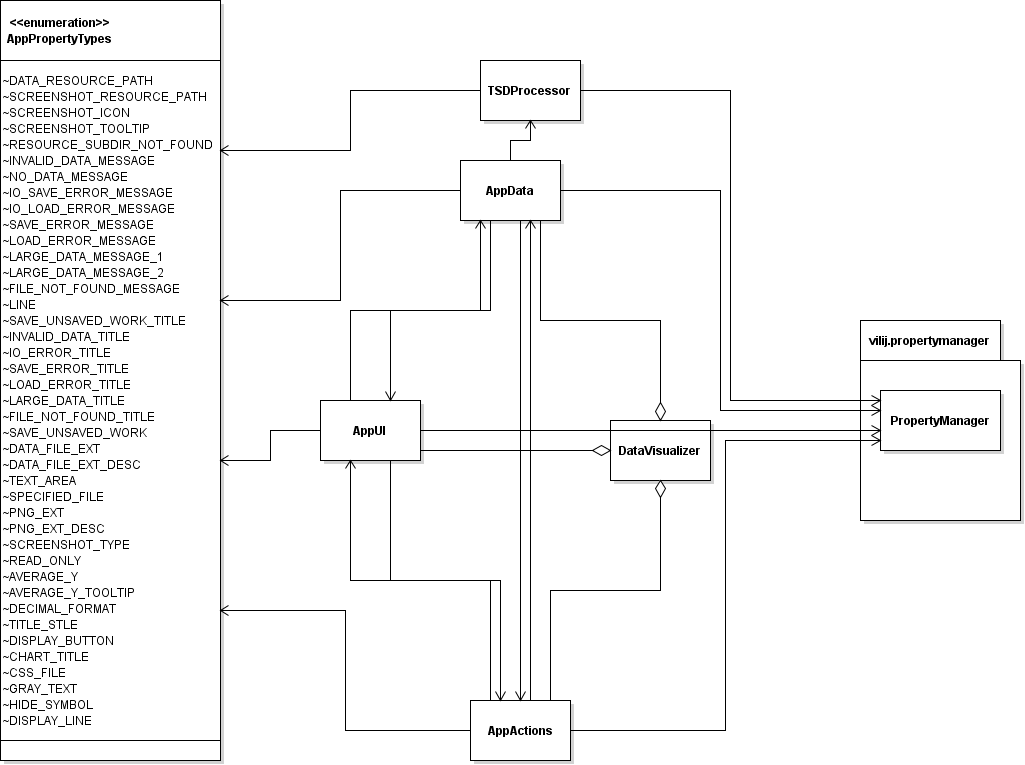
**Note:** This class is the container class of the whole application which overrides the start method itself. Initialization of the program itself, including the other classes of the module begins upon the call to this class’s start method.



**Fig 3.3.** Detailed UML Class Diagram of AppUI.java

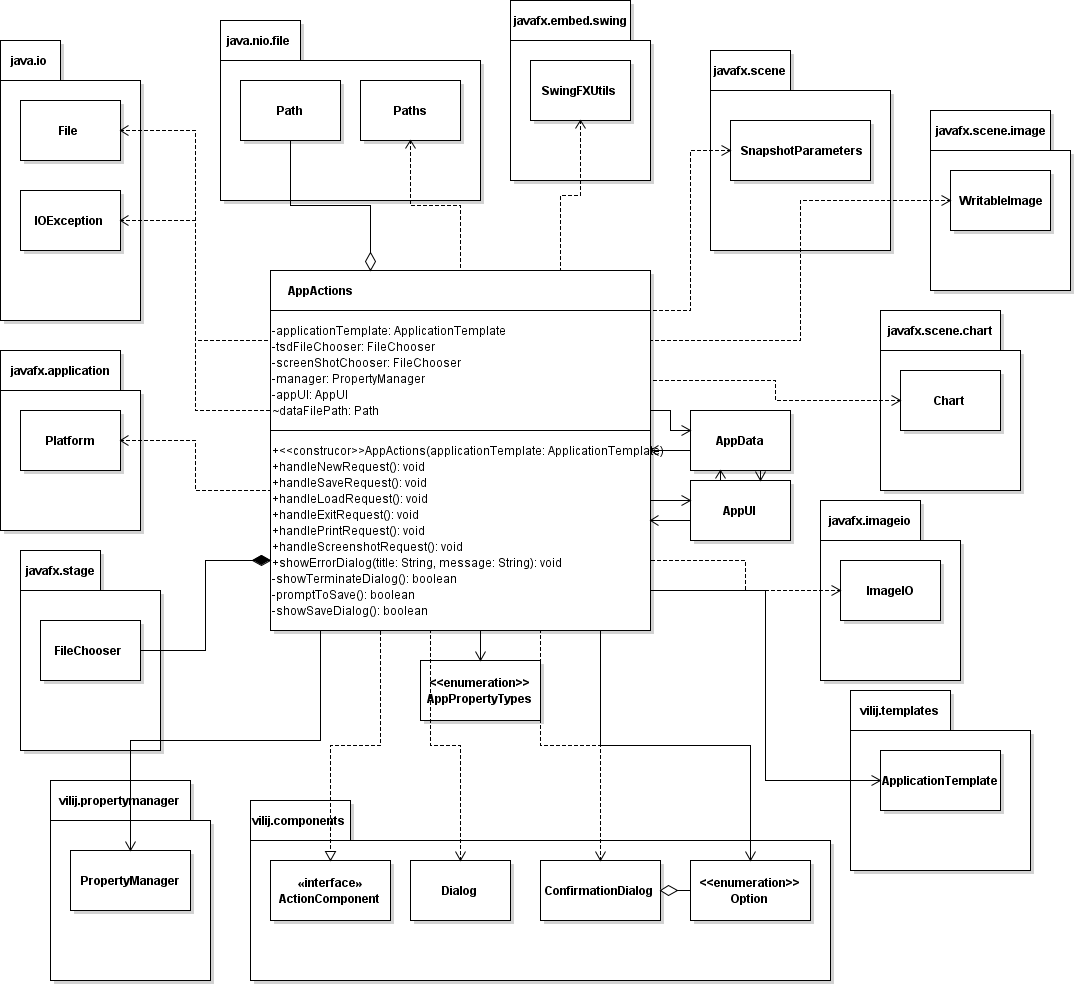
**Note:** This class is responsible for creating and displaying the Graphical User Interface of the application. It handles the relative location and overall aesthetic of each component of the user interface including the controls, the chart, and the top-level display of the application itself and its windows. This class also is responsible for setting the event handlers of each control listed in the UML Class Diagram above.

AppUI.java also contains an inner class that allow configurations of the algorithms to be set.

****

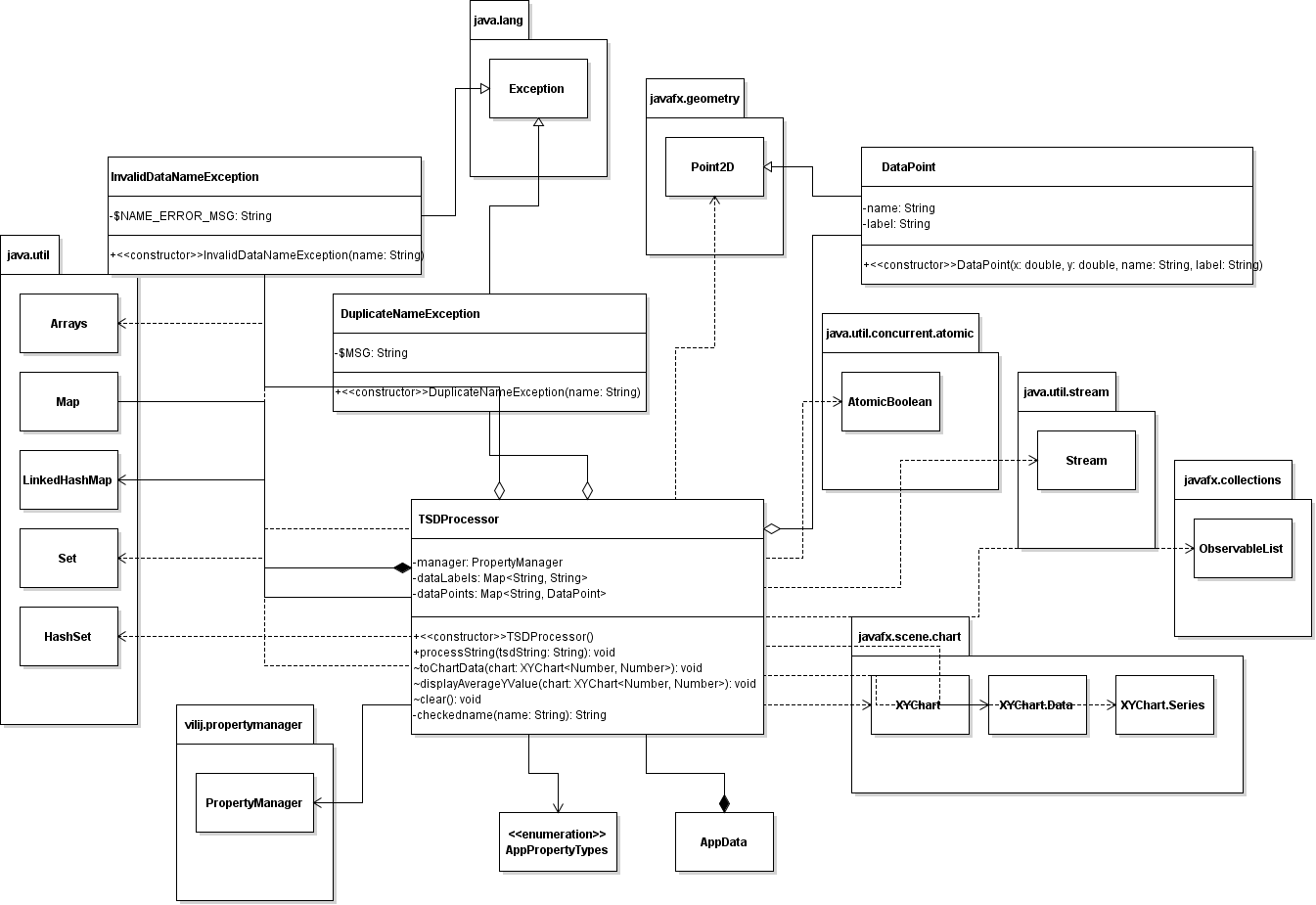
**Fig 3.4.** Detailed UML Class Diagram of AppPropertyTypes.java

**Note:** This enumerated class contains the constants that refer to the application specific properties.



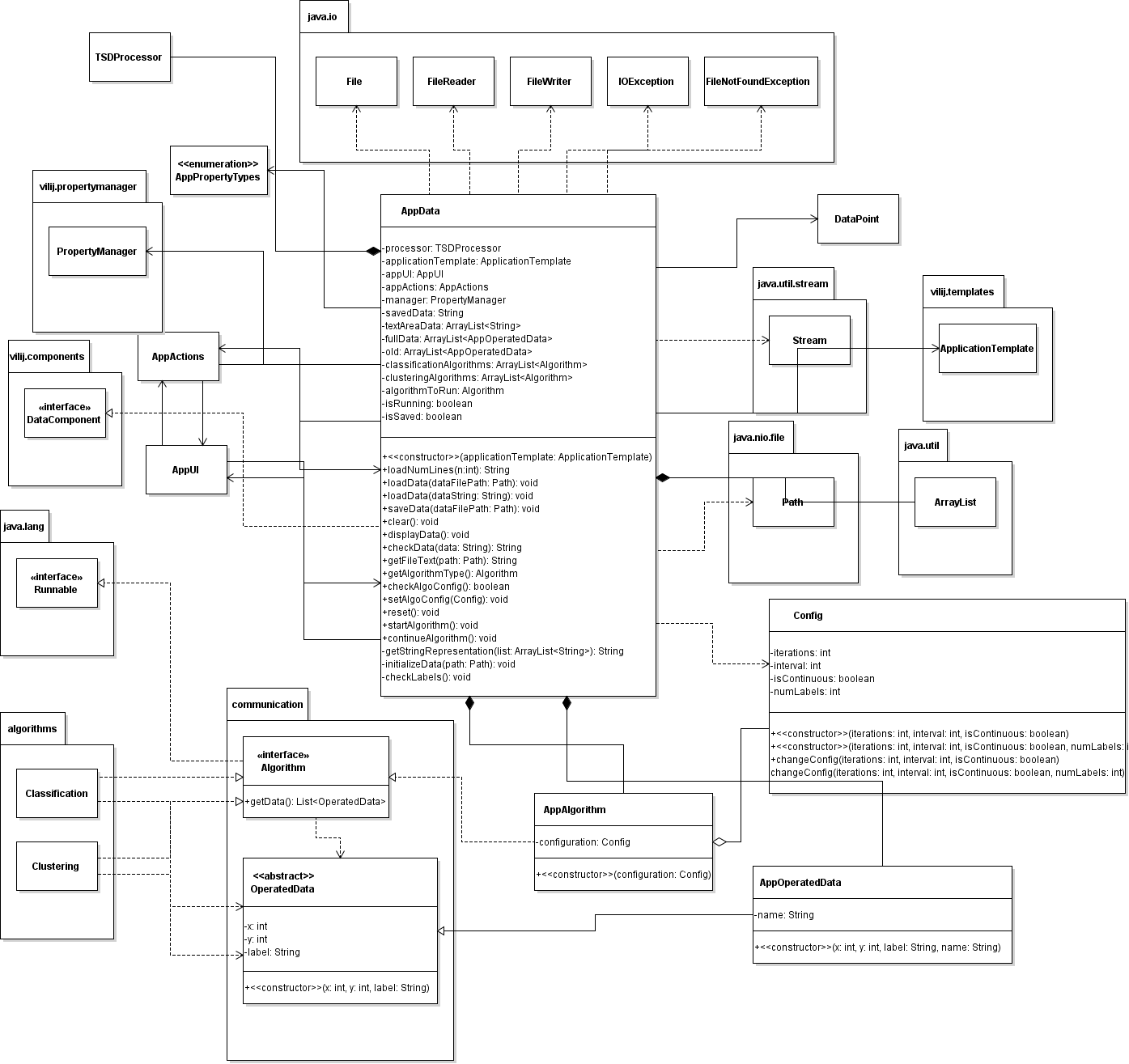
**Fig 3.5.** Detailed UML Class Diagram of AppActions.java

**Note:** This class is responsible for handling the actions that are possible in the application. Most of which correspond to the actions corresponding to the tool bar buttons of the primary window. This class is also the primary class that shows the Error Dialogs whenever an error occurs.



**Fig 3.6.** Detailed UML Class Diagram of TSDProcessor.java

**Note:** This class is the application specific class that parses the data of any given TSD file or any entered data in the text area and checks it validity to the constraints of the DataViLiJ module.



**Fig 3.5.** Detailed UML Class Diagram of communication package and AppData.java

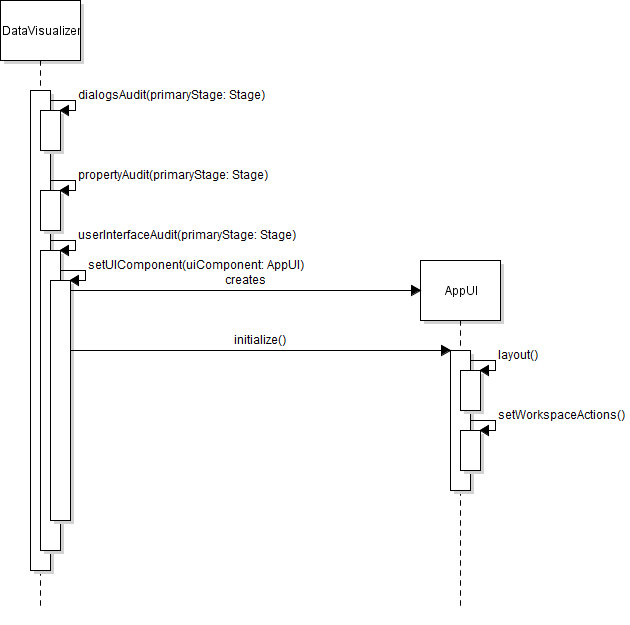
**Note:** This diagram displays the primary interaction between the GUI module, primarily AppData.java, the Algorithm module, and the communication package. The classes within the communication package are abstract, which provide the specification of behavior that does not need to be application dependent.

AppData.java is responsible for manipulating the TSD data provided by the other app components.

**4. Method-level Design Viewpoint**

This section deals with the dynamic processes that occur whenever the user interacts with DataViLiJ. It is different from the static UML Class Diagrams above. Instead it specifies the method calls using UML Sequence Diagrams.

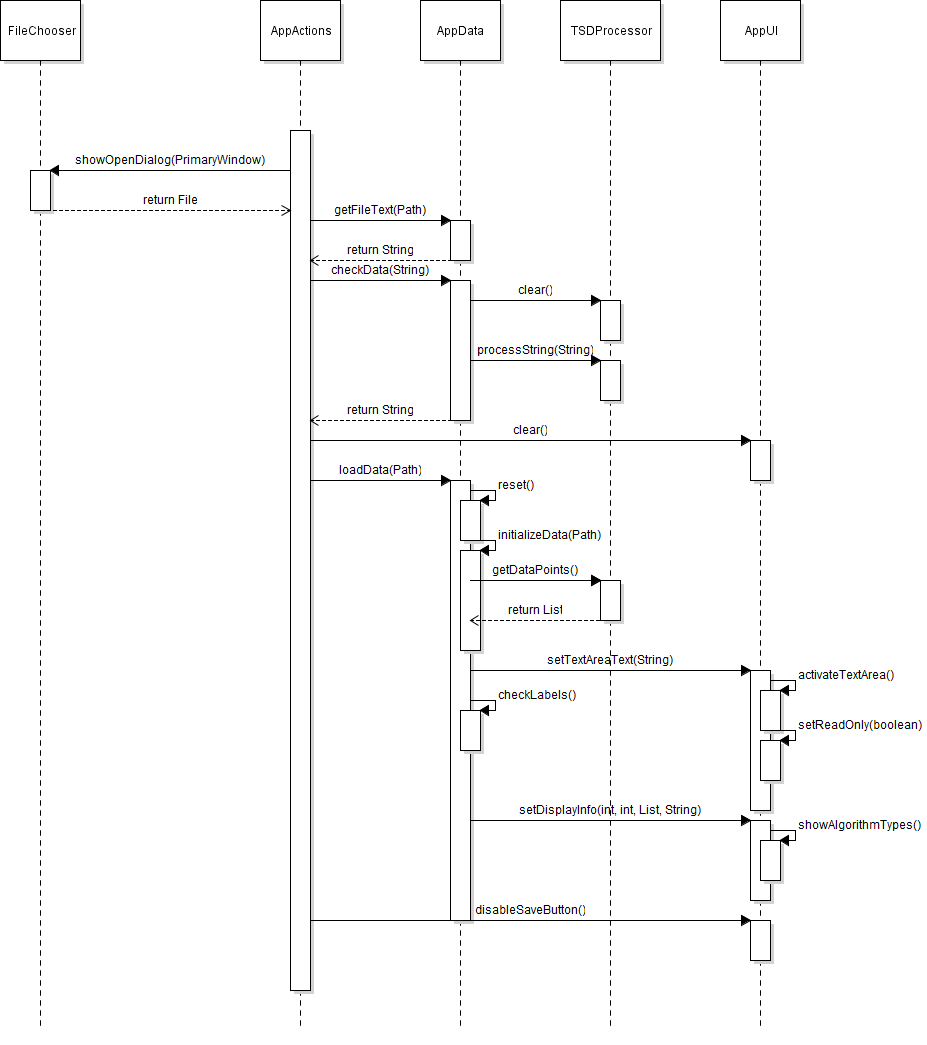
Each diagram corresponds to a use case specified by the Software Requirements Specification document.

Use Case 1: Start Application Use

**Fig 4.1.** Sequence Diagram of Application Starting

**Note:** Whenever DataViLiJ is run, DataVisualizer.java performs the necessary method calls to instantiate the whole application, including creating the user interface component.

The layout and setWorkspaceActions methods allow the primaryWindow to be instantiated including the buttons within it.

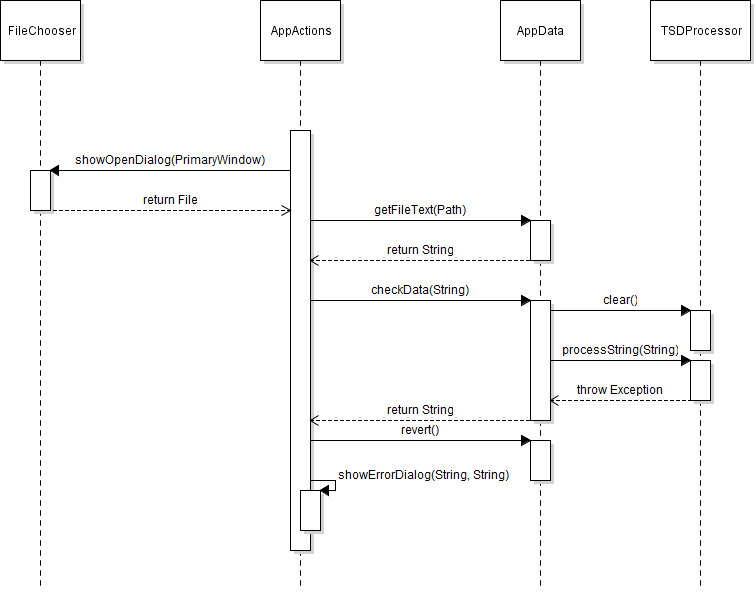
Use Case 2a: Load Data

**Fig 4.2a.** Sequence Diagram of Loading Valid TSD Data

**Note:** This case involves method calls that occur when the user loads valid data from a file in the TSD format by clicking the Load button in the toolbar.

The text area is modified such that only 10 lines are displayed. The information about the data is also displayed to the user, including the number of instances, the number of labels, the label names, and the source of the TSD data.

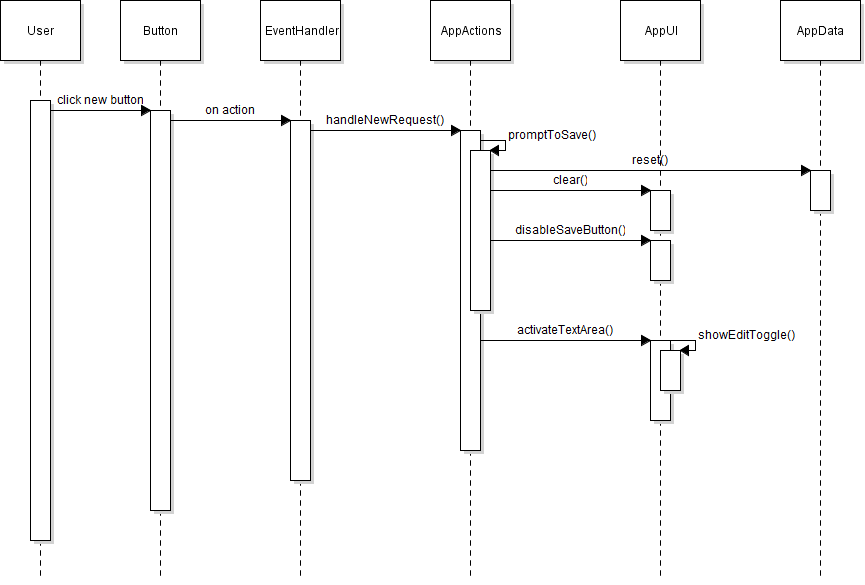
Additionally, the labels of the TSD file are parsed such that the appropriate algorithm types are shown.

Use Case 2b: Load Data

**Fig 4.2b.** Sequence Diagram of Loading Invalid TSD data

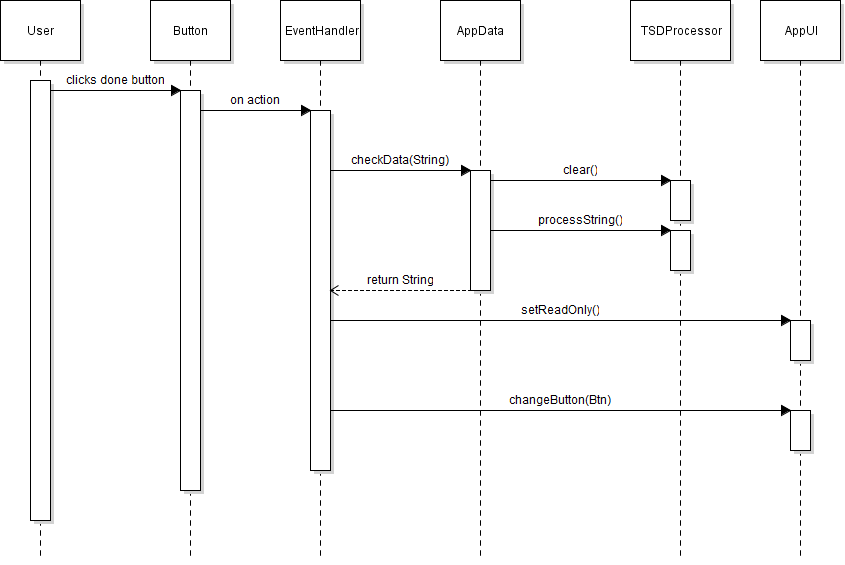
**Note**: This case involves the method calls that occur when the file does not conform to the Tab Separated Data format.

The data is not shown, and instead an error dialog is used to notify the user.

Use Case 3.1: Create New Data

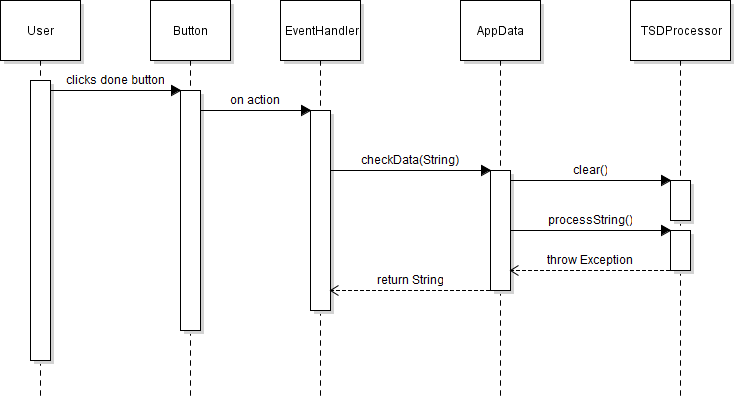
**Fig 4.3.1.** Sequence Diagram of Creating New Save File

**Note:** These are the method calls where the user is presented with a dialog to create new data. Once the save file is created, the text area is activated and the following sub cases are followed.

Use Case 3.2a: Create New Data

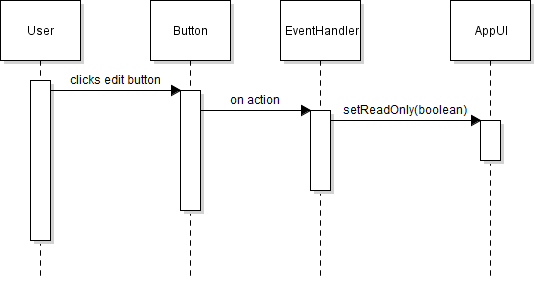
**Fig 4.3.2a** Sequence Diagram when Input Data is Valid

**Note:** When the user clicks the done button after editing, the data is parsed and checked for validity by TSDProcessor.java. Since the data is valid, the sequence follows directly case 2a. The text area is disabled and the toggle button changes.

Use Case 3.2b: Create New Data

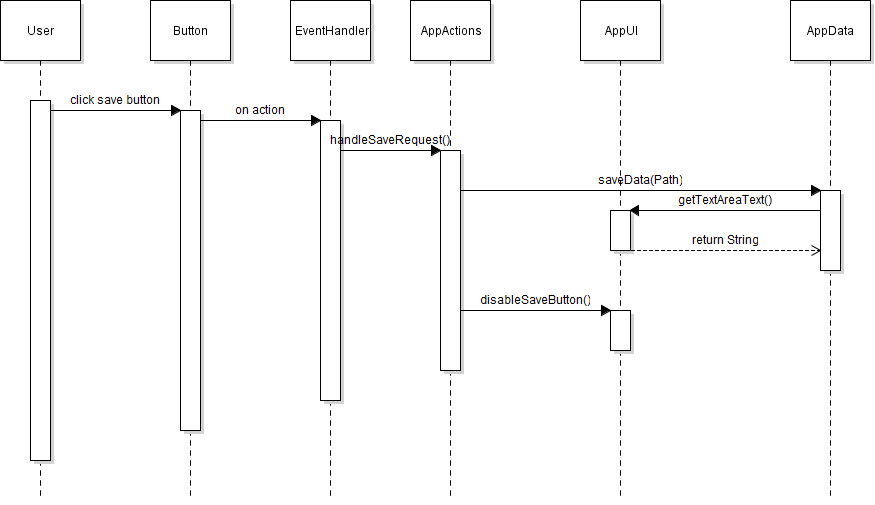
**Fig 4.3.2b.** Sequence Diagram when Input Data is Invalid

**Note:** When the user clicks the done button after editing and the data is invalid, the sequence follows case 2b.

Use Case 3.3: Create New Data

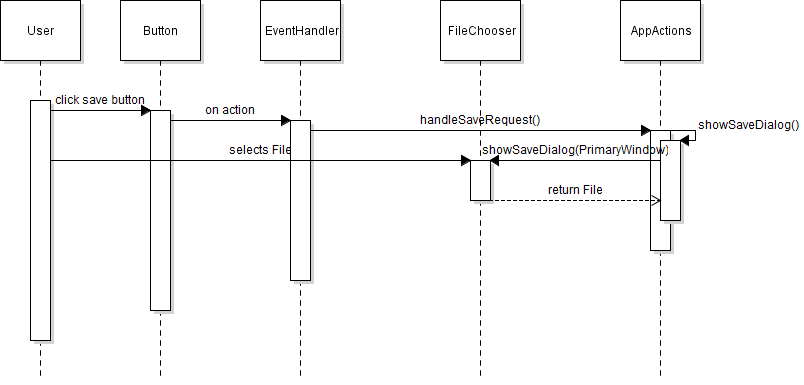
**Fig 4.3.3.** Sequence Diagram when Edit Button is Clicked

**Note**: The method calls above simply show how the edit button allows the text area to be modifiable by the user.

****Use Case 4a: Save Data

**Fig 4.4a.** Sequence Diagram when Save File Exists

**Note**: The sequence shows what occurs when the user clicks the save button and a save file already exists. The save button is disabled upon saving and remains that way until new data is created. The file application also restricts the format of the file by only allowing .tsd files to be saved.

Use Case 4b: Save Data

**Fig 4.4b.** Sequence Diagram when Save File Does Not Exist

**Note**: This sequence shows when the save file does not exist yet. The user selects a file to save the data from the application, and the sequence directly follows case 4a.

Use Case 5: Select Algorithm Type