# Homework 2-Part 3: Project Design + Patterns

## Contents



### Project Summary

Group 2 chose to develop a system for defining, populating and visualizing datasets.

For the purpose of this project, a d*ataset* is a structure defined by a number of named, typed attributes and a sequential collection of samples containing a value matching each attribute. The Attributes within a Dataset serve as definitions for bounding the Values within a Sample. In summary, a *Dataset* consists of a name, Dataset Definition and Samples.

A *Dataset Reference* describes the location of a dataset that is available on the server. A reference consists of a name and a URI to the dataset itself. Data definitions provided clients with a mechanism to access datasets and visualizations.

A *Visualization* consists of attributes and samples to define visualizations such as a scatter, series or histogram plot. The attributes serve as to define and bounding the visualization sample data.

**See Appendix A for JSON representations of a dataset, dataset and visualization dataset references and visualization datasets.**

The Visitor pattern (seen in Figure 7 - Data Definition Class Diagram) instances allow us to provide strongly typed but heterogeneous data samples.

Together the two parallel class hierarchies allow us to define a dataset as concrete, but flexible data structure and a series of samples conforming to that structure.

The abstractions provided allow us to work with these datasets with no regard to how their originally provided to the interface.

The following diagrams describe use cases and an overview of patterns utilized within the project:

- Define a dataset.
  - Figure 3 - Data Definition Sequence Diagram
  - Figure 7 - Data Definition Class Diagram
- Append data to an existing dataset.
  - Figure 4 - Add to Dataset Sequence Diagram
  - Figure 8 - Add to Dataset Class Diagram
- Upload a complete dataset.
  - Figure 5 - File Input Sequence Diagram
  - Figure 9 - File Input Class Diagram
- Visualize and Define a dataset
  - Figure 6 - Visualize and Define Dataset Sequence Diagram
  - Figure 10 - Visualize Data Set Class Diagram
- Patterns:
  - Figure 11: Attribute Visitor Pattern
  - Figure 12: Value Visitor Pattern
  - Figure 15 - FileInputHandler Strategy Visitor Pattern
  - Figure 15 - FileInputHandler Strategy
  - Figure 15 - FileInputHandler Strategy
  - Figure 16: Visualization Template Pattern
  - Figure 17: GsonTypeAdapter - Adapter Pattern

## Project Demo URL

URL: https://youtu.be/vaocsKgwadg

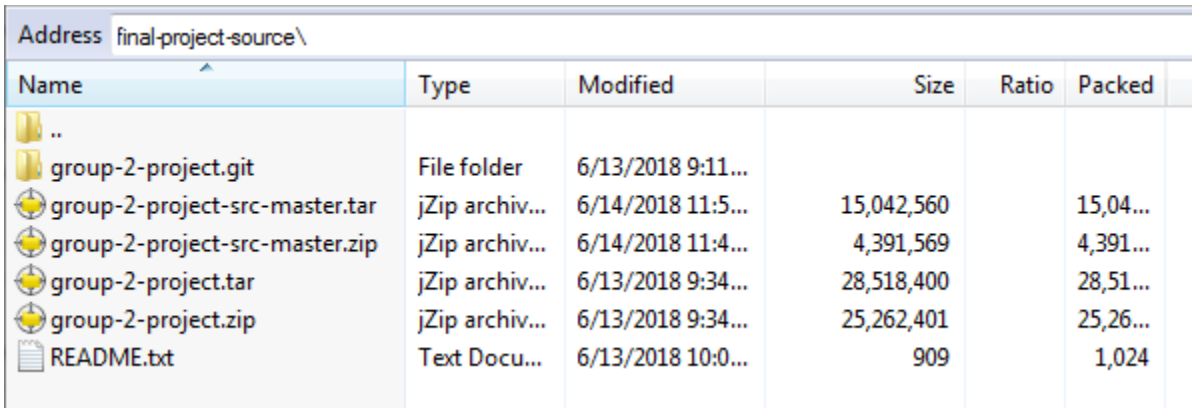
Narrator: Justin Buckley

## Code Delivery Contents

Project Prerequisites:

- Java 10 (major version: 54)
- MongoDB : https://www.mongodb.com/download-center?jmp=tutorials&_ga=2.209797138.604650130.1528941887-1410356226.1528485520#community

Expand the provided final-project-source.tar.gz to access folder final-project-source containing:




- Project Source (Note: both archives contain identical data):
  - group-2-project.git – clonable .git to access the projects source code.
    *Note: The team did send an invite to access the Group 2 GitLab repository earlier within the quarter. Please email the group if you require another invite to the repository to clone. The src-master archives provided so access to the repo is not necessary.*
    To clone the repo:
    - *git clone group-2-project.*git
      OR
      *git clone https://gitlab.cci.drexel.edu/se-577-group-projects/group-2-project.git*
  - group-2-project-src-master – archived downloads of the repos master branch.
- Projects Binaries
  - group-2-project.tar
  - group-2-project.zip
- Readme.txt – details included here for those who do not have access to this document.

## Running the Project

1. Extract or Clone source code “group-2-projects” folder.
2. Change directory to “group-2-projects” folder.
3. Start MongoDB via shell or command line:

a. *"{PATH_TO_MONGODB}\bin\mongod.exe" --dbpath ".\mongoDataTemp"*
   OR
b. Use script *group-2-projects\start_MongoDBServer.bat*

4. Start the project, see group-2-projects\Readme.md
5. Open a browser, navigate to the applications website URL http://localhost:4567.

# 1 Deployment Diagram

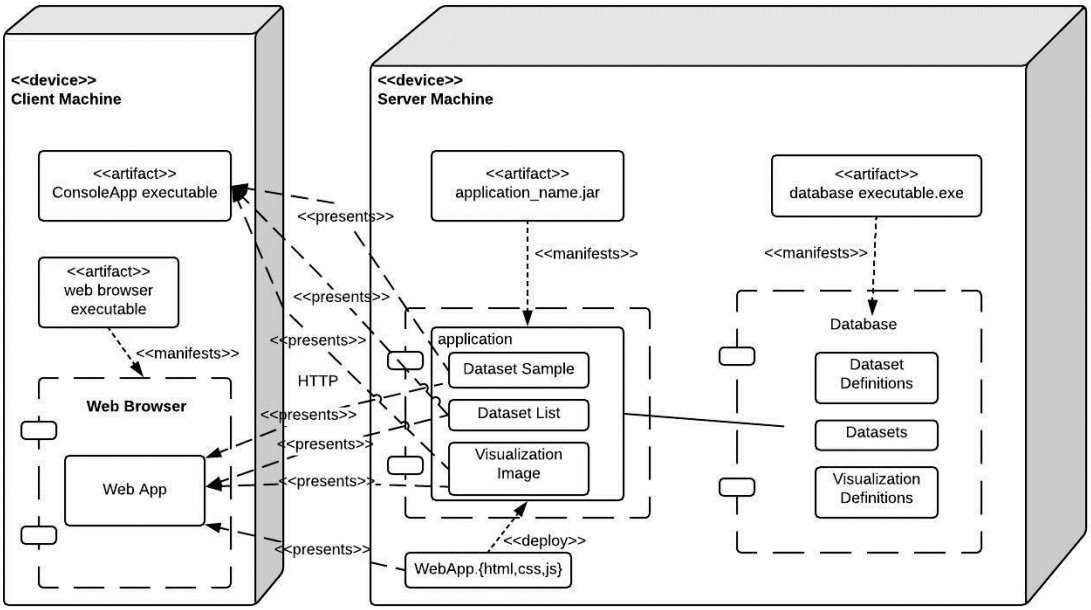



*Figure 1 - Project Deployment Diagram*

Figure 1 depicts the projects deployment. For our use-cases, the client uses the web application. The ConsoleApp executable was included as an example of a future feature to demonstrate the current architectures extensibility. For example, when the client request a command-line interface, the REST API provides easy access the server's service endpoint. Both the command-line interface and the web applications server tier would access the same service endpoints without modification to the server code providing a different user experience with minimal effort.

The server shall execute the Application Server via a JAR assembly. The Application Server utilizes datasets, data samples and visualization images stored within a database. The Database Executable shall deploy a database storage solution.

Additionally the server hosts the web application. The Web application will consist of HTML, CSS and JavaScript files.

## 2 Component Diagram

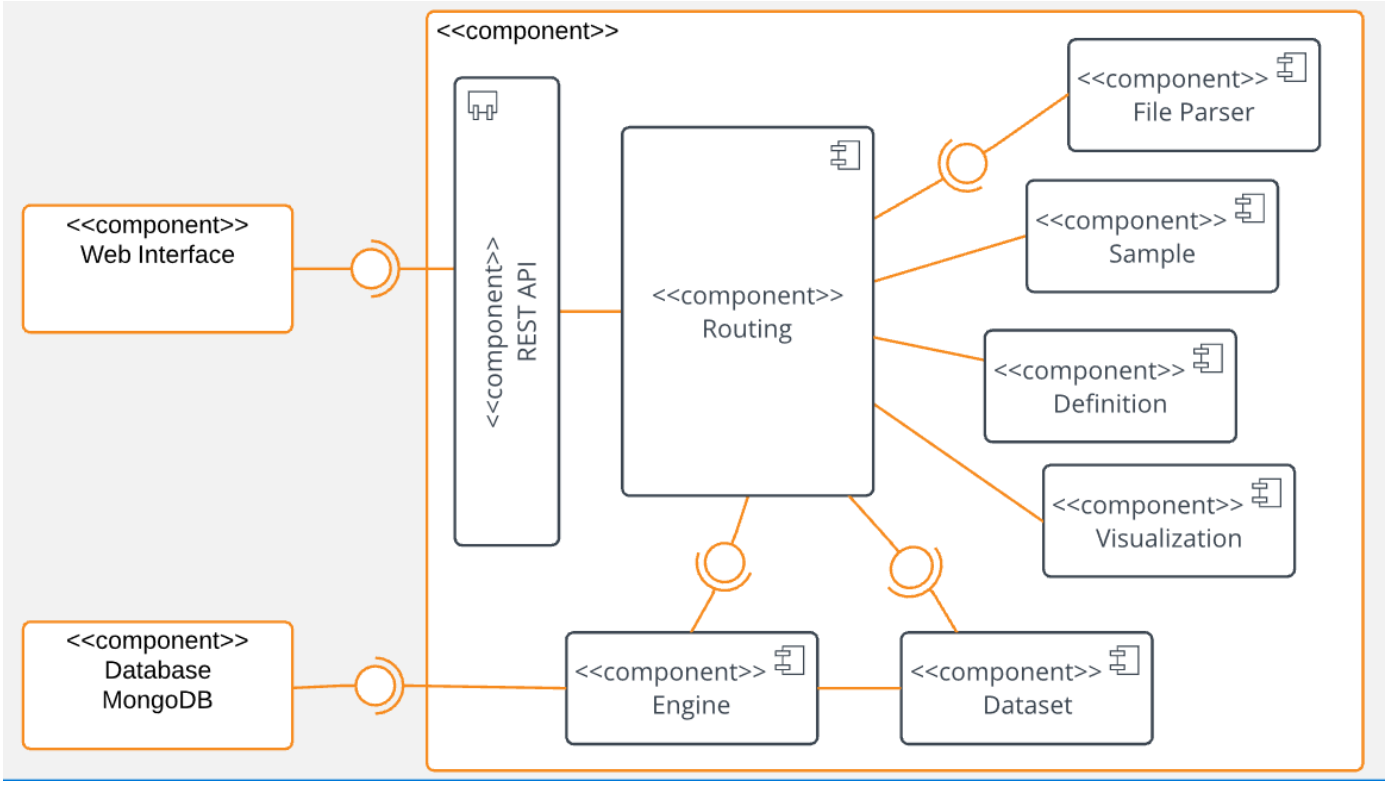



*Figure 2 - Component Diagram*

Figure 2 describes the projects components. The left represents the client facing component including the REST API endpoint and Static Web Hosting.

The middle describes the processing components:

- File Parser – Determines file type to provide appropriate strategy for parsing.
- Routing – Depending on the request, business logics applied to satisfy use cases. Functionality includes modifying an existing Dataset, preparation of a Dataset for visualization and the creation of new or modified Datasets and data definitions.
- Engine – Generalized interface to an underlying data storage solution. For example, MongoDB is an option we are investigating.
- Visualization – Determines and executes strategy for visualization of datasets.
- REST API – Endpoint utilized by the client to access the application dataset services.

The right side represents the storage tier of the system. The storage tiers abstracted from the processing components via a common interface.

# 3 Sequence Diagrams

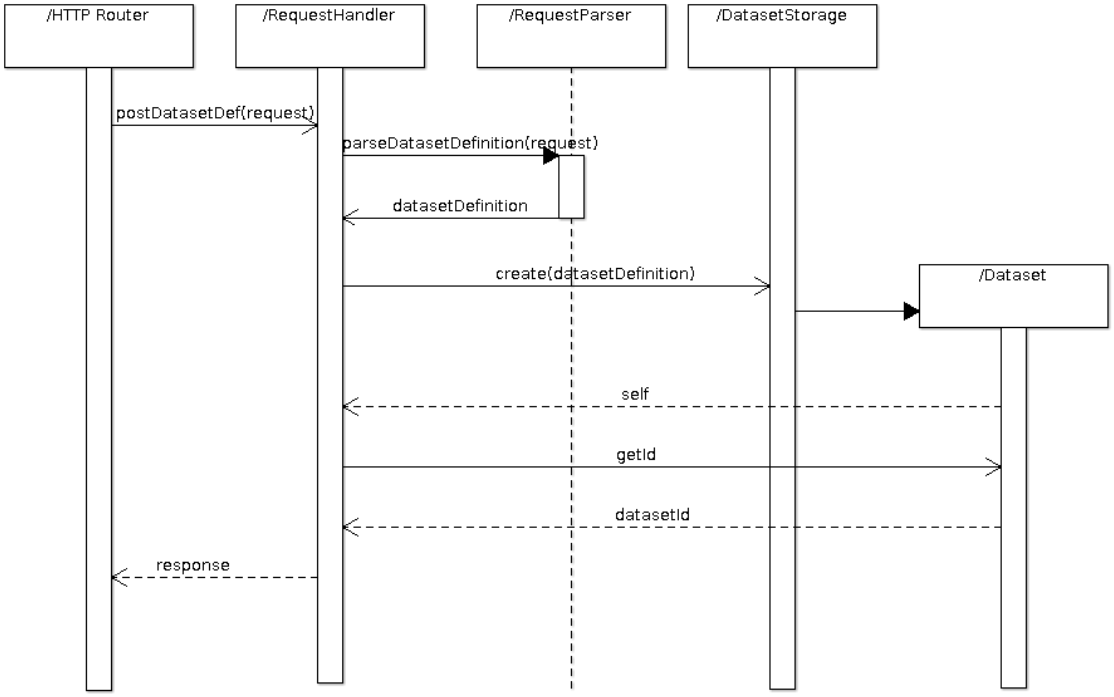



*Figure 3 - Data Definition Sequence Diagram*

Data definition consists of a post HTTP request method containing a dataset definition. The data definitions parsed and relayed to the data storage where its Datasets created then stored. The new Datasets identifiers provided to the request handler, which then requests and ID from the Dataset. The Dataset IDs relayed back via a HTTP response containing the now available dataset.

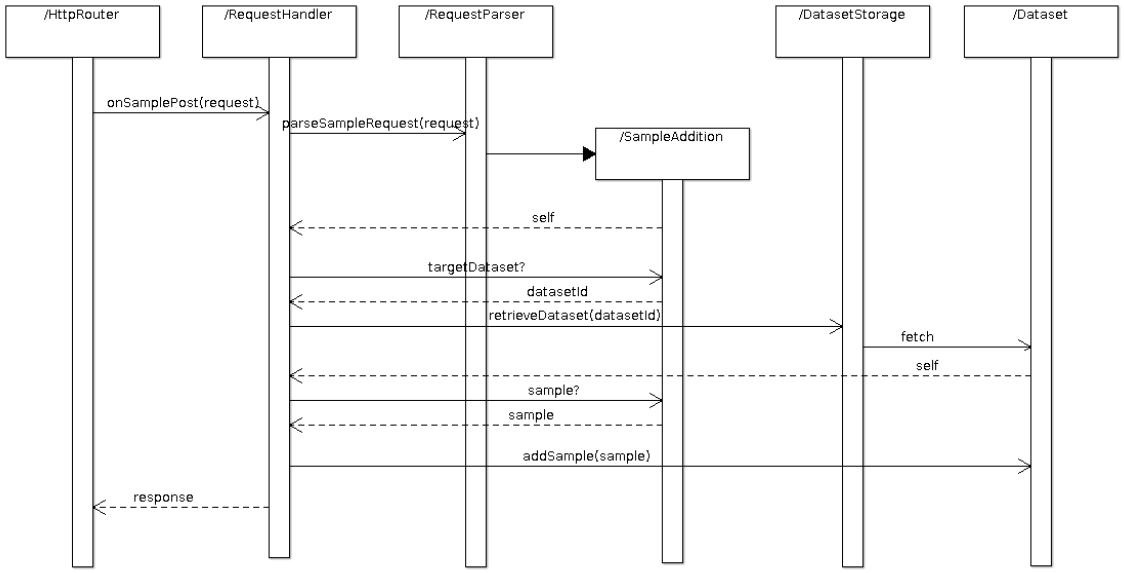



*Figure 4 - Add to Dataset Sequence Diagram*

Adding to an existing Dataset consists of a post HTTP request method containing new sample and the ID of the targeted data set. The request handler resolved the appropriate parser then extracts the new data and existing target Dataset ID. The targeted Datasets

extracted from storage. The targeted Datasets updated with the new data samples. The modified target additions to the dataset are replicated in the storage. A response relays back via a HTTP response a successful outcome.

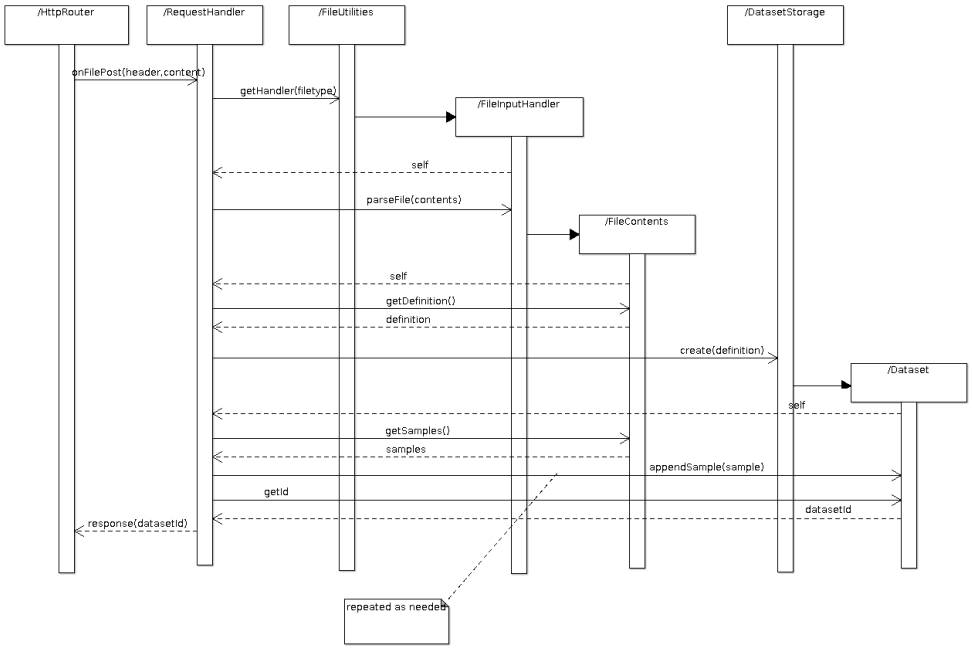



*Figure 5 - File Input Sequence Diagram*

File input consists of a post HTTP request method containing a data file. Request handler request the appropriate file parser from *FileUtilities* given file type. The files parsed extracting the *FileContents*. File contents decompose into a data definition. A *Dataset*s created. Data samples extraction from the *FileContents* begins. Data sample extraction continues building a collection of samples until the files completely parsed. The Datasets stored. A response relays back via a HTTP response a successful outcome and the newly created and available Dataset ID.

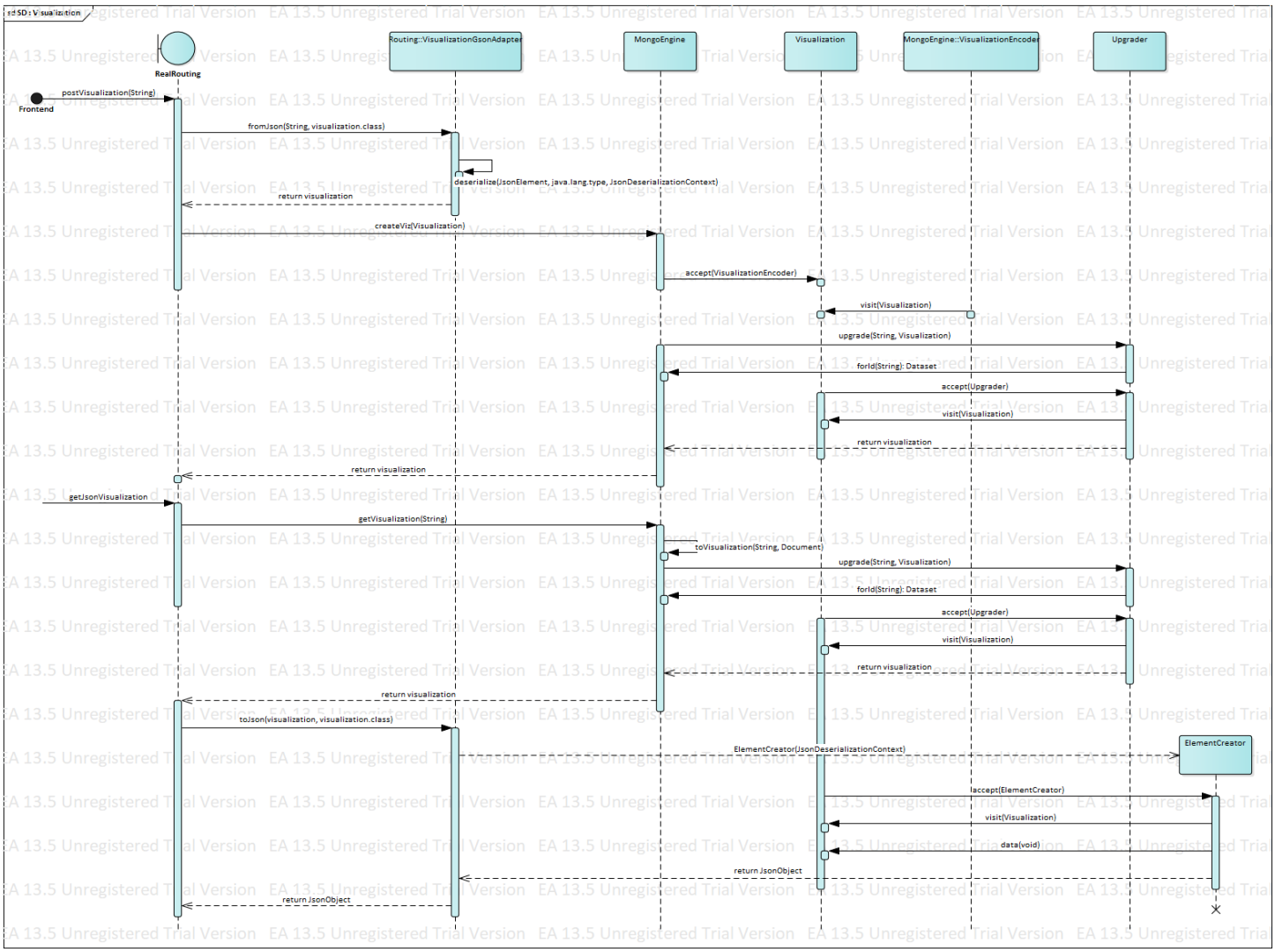



*Figure 6 - Visualize and Define Dataset Sequence Diagram*

Data visualization post HTTP request contains the target dataset ID, attributes to be used and the type of visualization, which are all encapsulated into a JSON object. The gsonadapter converts the JSON object into a visualization object, which is then further converted into the visualization object based on the desired type. This object is then sent to the database endpoint MongoEngine, which takes care of storing the visualization definition into the database and returns the visualization which contains the visualization ID as well.

Data Visualization get HTTP request contains a visualization ID, which is received by the REST endpoint RealRouting. The requested visualization definition is extracted from the database endpoint MongoEngine and returned. The data samples and the required attributes are then extracted based on the type of visualization, which is handled by the template method pattern implemented in Visualization. Image is used to return complete images in cases where JavaScript may not be available in the host system. The Upgrader class implements the visitor for visualization and performs operations based on the visualization types.

# 4 Class Diagrams

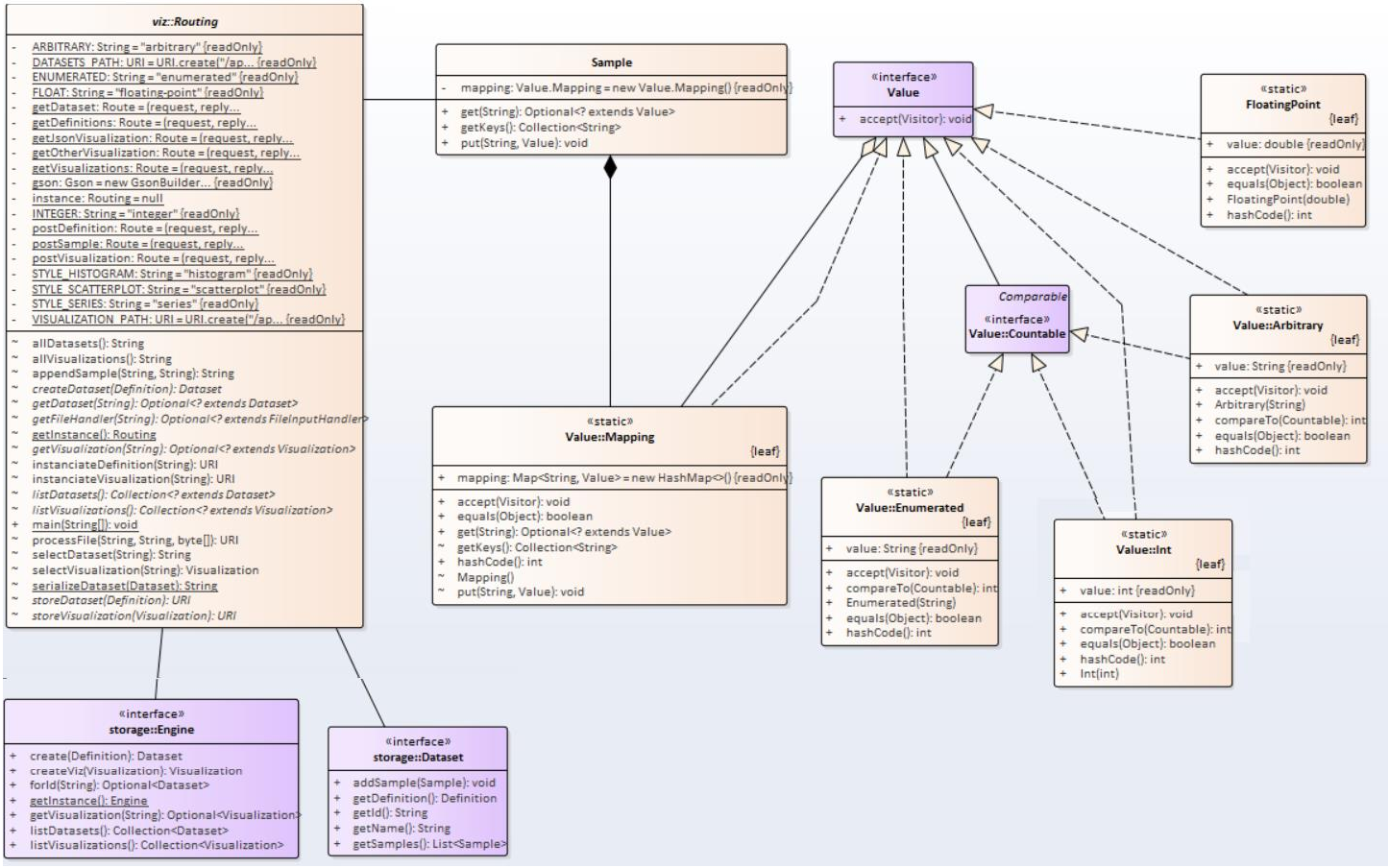



*Figure 7 - Data Definition Class Diagram*

Data definition post HTTP request containing a dataset definitions received by the REST endpoint *Routing* then parsed and identified by the *Sample*. The dataset Definitions modeled using the Visitor pattern. Once extracted the data definitions relayed back though the to the *Routing* and stored as a Dataset. The data definition identifier is relayed back via a HTTP response.

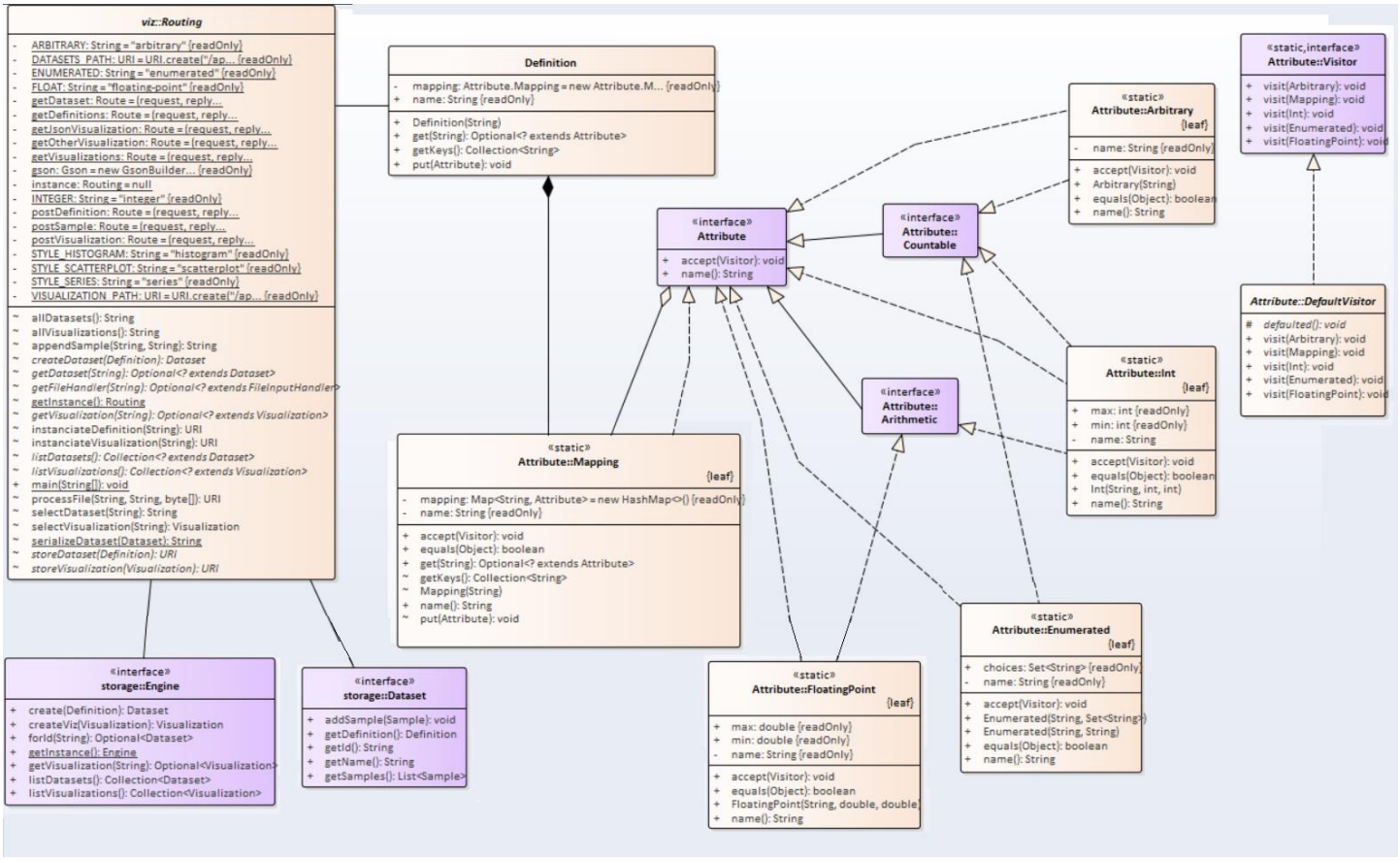



*Figure 8 - Add to Dataset Class Diagram*

Data definition post HTTP request containing a sample data and the ID of the targeted Datasets are received by the REST endpoint *Routing* then parsed and identified by the *Definition*. The d*efinition re*solves the appropriate parser then extracts the new data and existing target Dataset ID. The targeted Datasets are extracted from storage. The sample data is modeled using the Visitor pattern and is appended to the target dataset. The modified target additions to the dataset are replicated in the storage. A response relays back via a HTTP response for successful outcome.

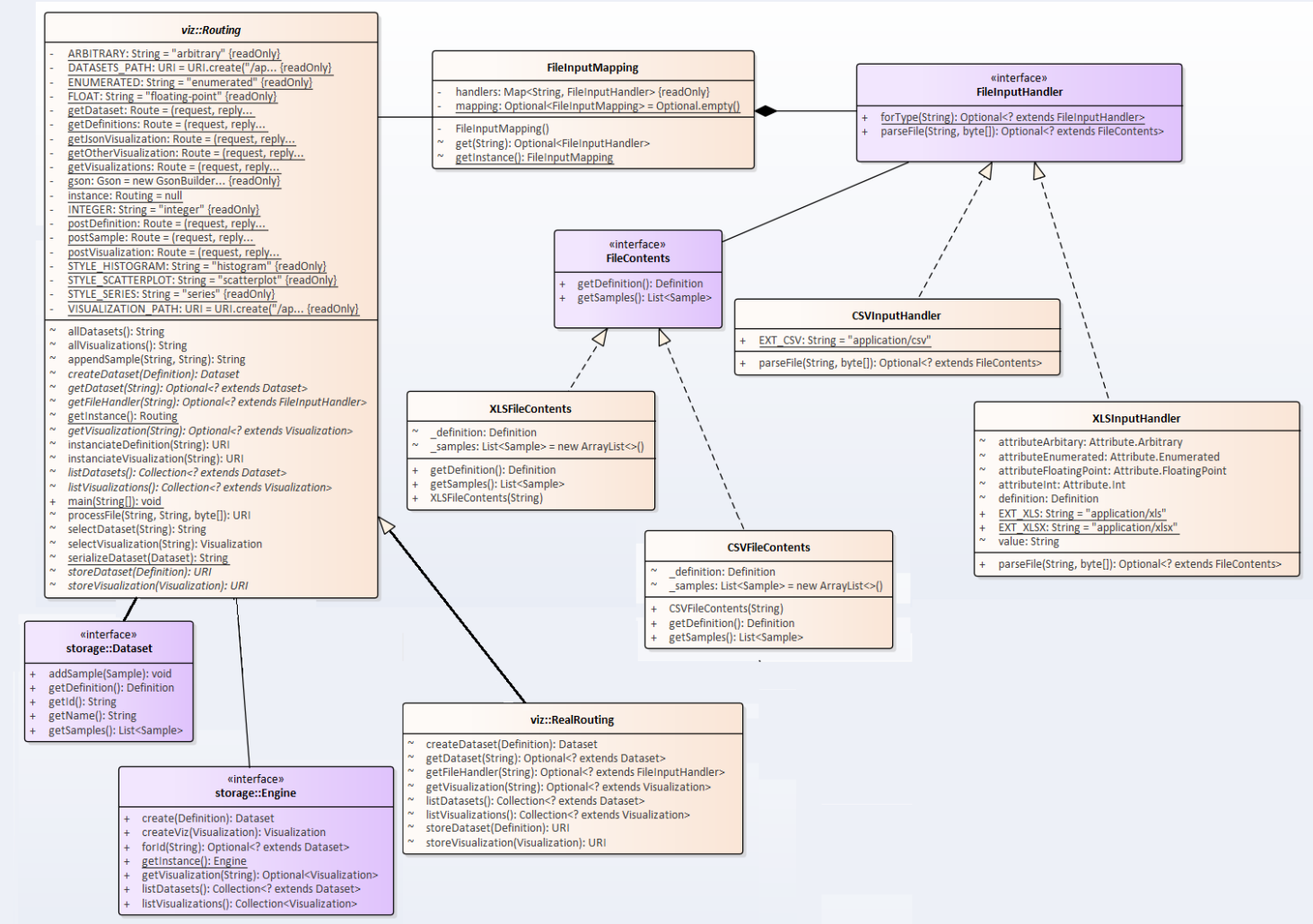



*Figure 9 - File Input Class Diagram*

Data definition post HTTP request containing a data file received by the REST endpoint *Routing* then parsed and identified by the *FileInputHandler*. The *FileInputHandler* utilizes the *FileInputMapping* to identify the correct strategy to parse the file. A *Dataset i*s created. The Datasets is stored via the *Engine*. A response relays back via a HTTP response of successful outcome.

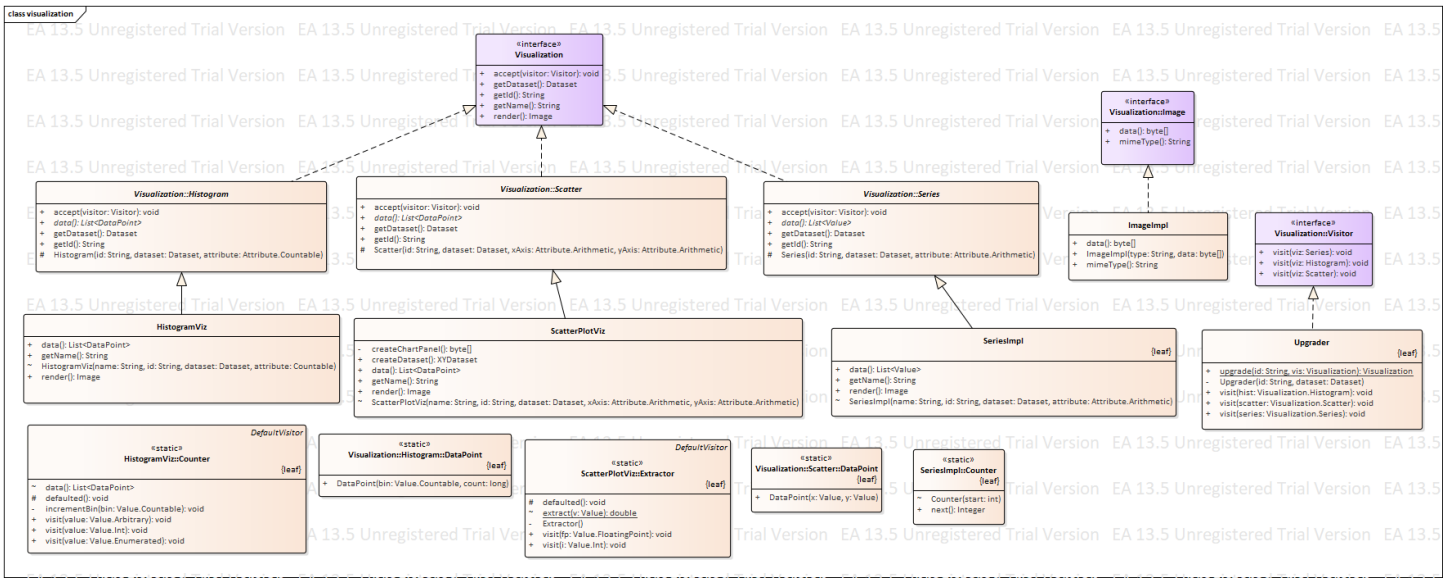



*Figure 10 - Visualize Data Set Class Diagram*

Data Visualization get HTTP request contains a visualization ID, which is received by the REST endpoint RealRouting. The requested visualization definitions extracted from the database endpoint and returned. The data samples and the required attributes are then extracted based on the type of visualization, which is handled by the template method pattern implemented in Visualization. Image is used to return complete images in cases where JavaScript may not be available in the host system. The Upgrader class implements the visitor for visualization and performs operations based on the visualization types.

# 5 Patterns

## 5.1 Visitor Pattern

The visitor pattern is used with the Value.Mapping class, the Attribute.Mapping class, translating back to JSON in the routing class and the samplevalidator class. The purpose of this pattern is to allow a new method or type to be added to the overall object structure without changing the existing elements that object currently operates on. Therefore, the visitor pattern was chosen for all areas to allow for future additions of different types of values with minimal modification. As an aside, there is some preliminary support for nested attributes/values in the Attribute.Mapping and Value.Mapping classes. For the purposes of the project, it was decided that utilizing these to introduce the composite pattern would complicate things unnecessarily. The team decided to leverage the Visitor pattern just to provide double dispatch in these areas. The {Attribute,Value}.Mapping classes are actually used to provide most of the implementation of the Definition and Sample classes.

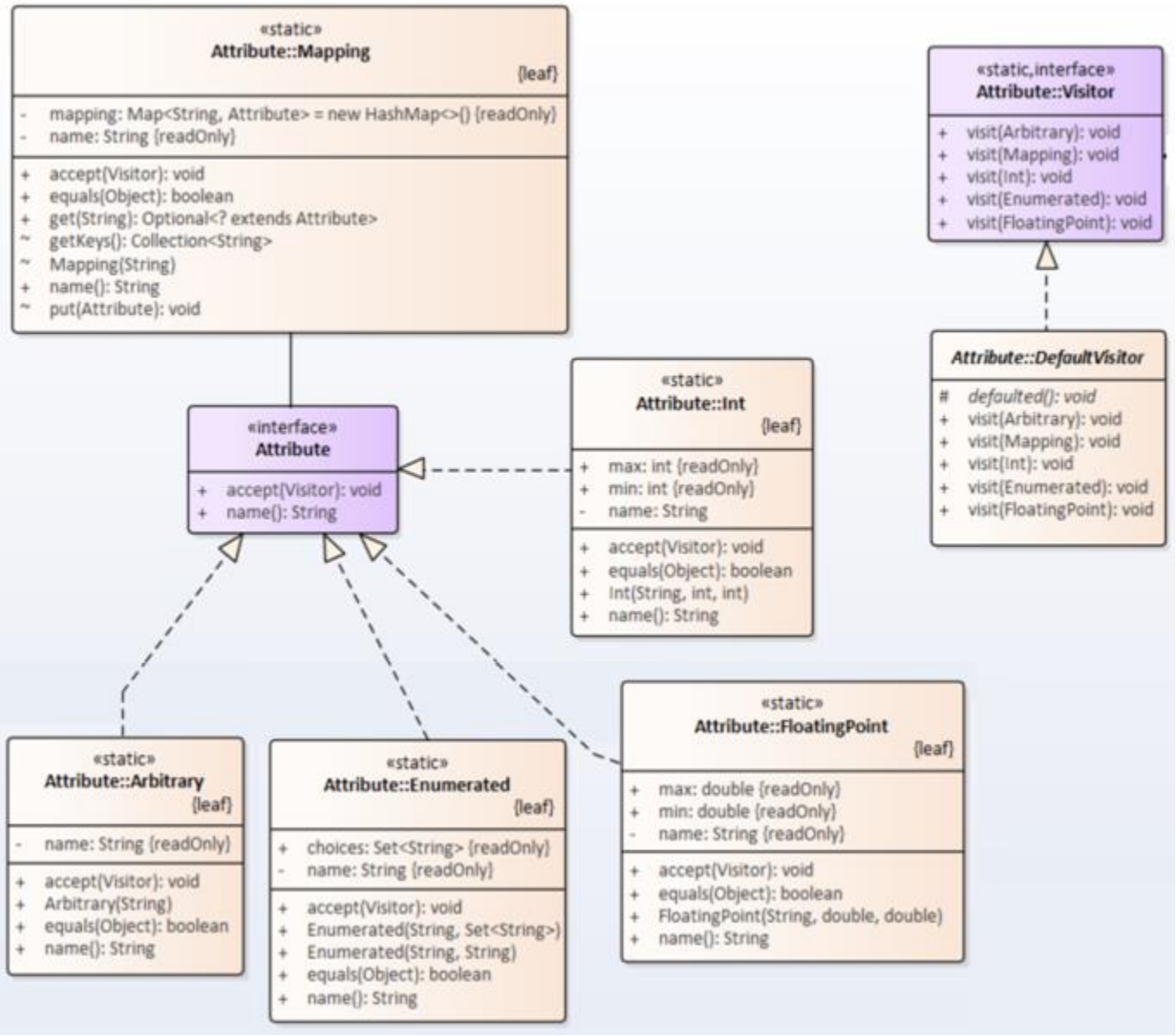



*Figure 11: Attribute Visitor Pattern*

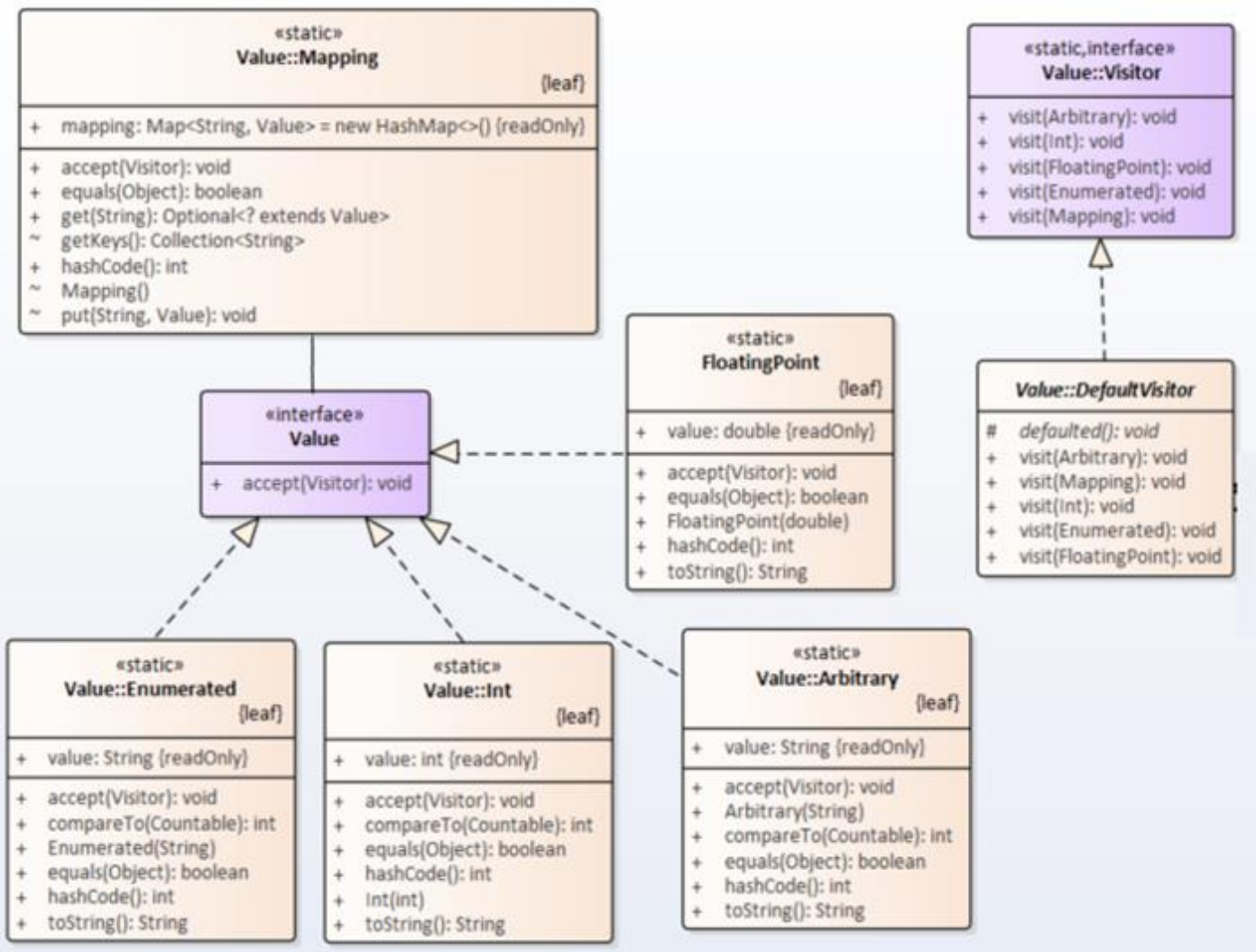



*Figure 12: Value Visitor Pattern*

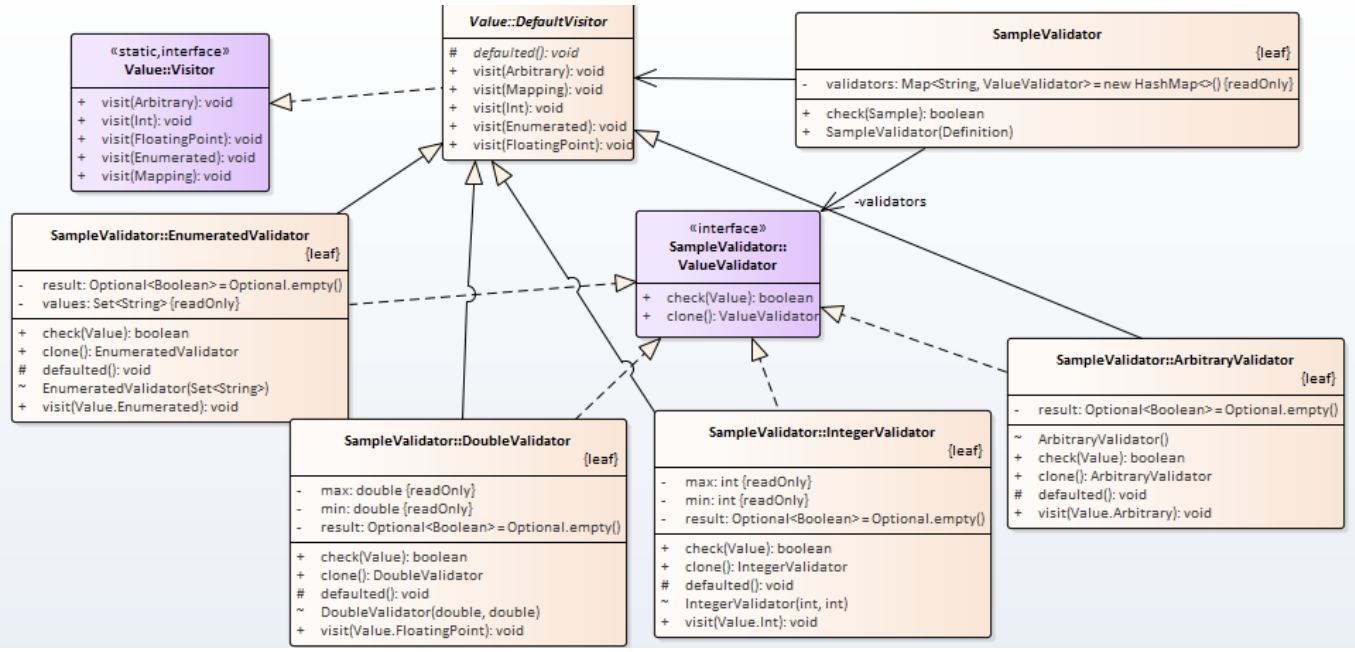



*Figure 13: Sample Validator Visitor Pattern*

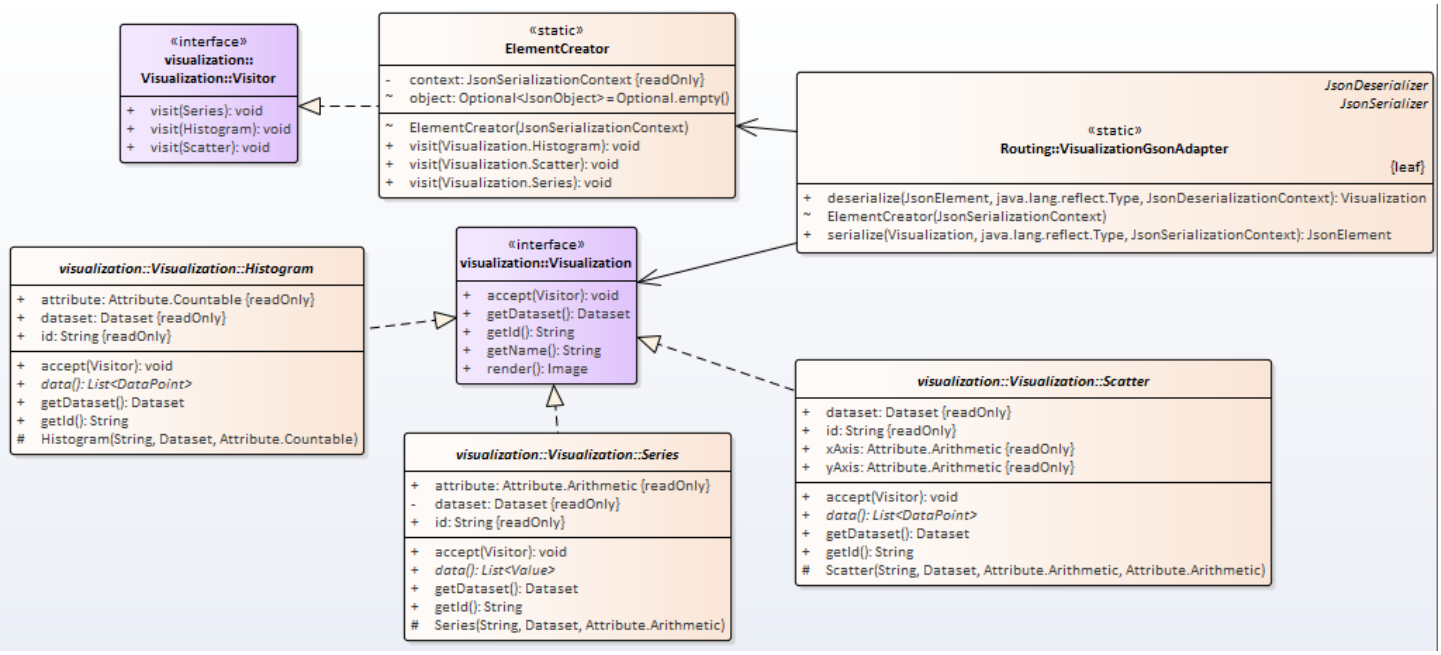



*Figure 14: VisualizationGsonAdapter Visitor Pattern*

## 5.2 Strategy Pattern

The strategy patterns used to define a family of algorithms that can be used when parsing an input file. This pattern allows the algorithms to vary independently from the clients that implement them. The File input handler is used as an interface for different types of files to be parsed. The request handler is considered the pattern context that defines the different rules for what class will be implemented depending on the request. This pattern allows the easy addition of file types.

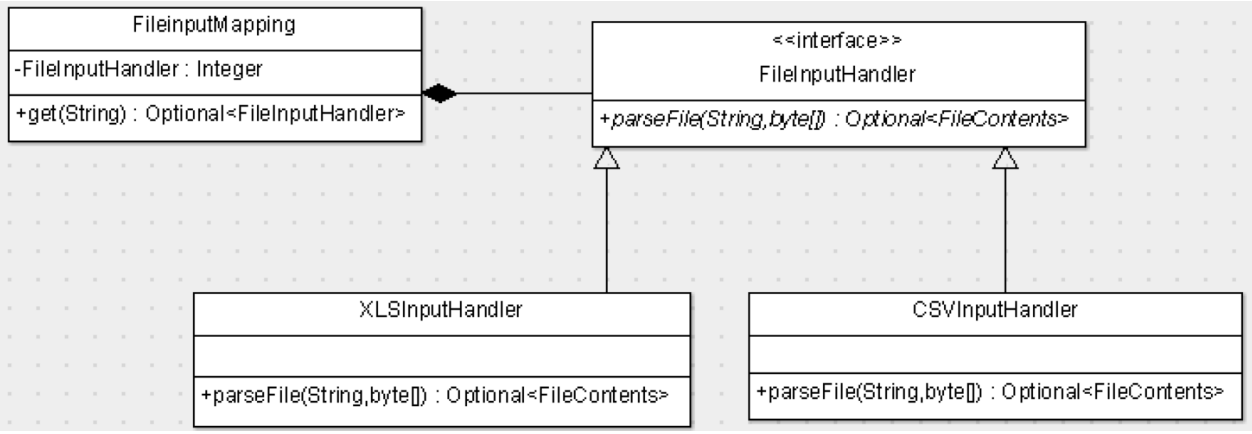



*Figure 15 - FileInputHandler Strategy Pattern*

### 5.3 Template Method Pattern

The template method pattern is used in the Routing class to provide a behavioral skeleton for routing operations. The contract provides structure for easily changing implementations depending on supporting class availability. Each implementation's refinement of its subclass processes and details can occur in isolation without alteration to the overall structure of the base implementation, which has easily allowed us to enable backend testing with mock interfaces during development.

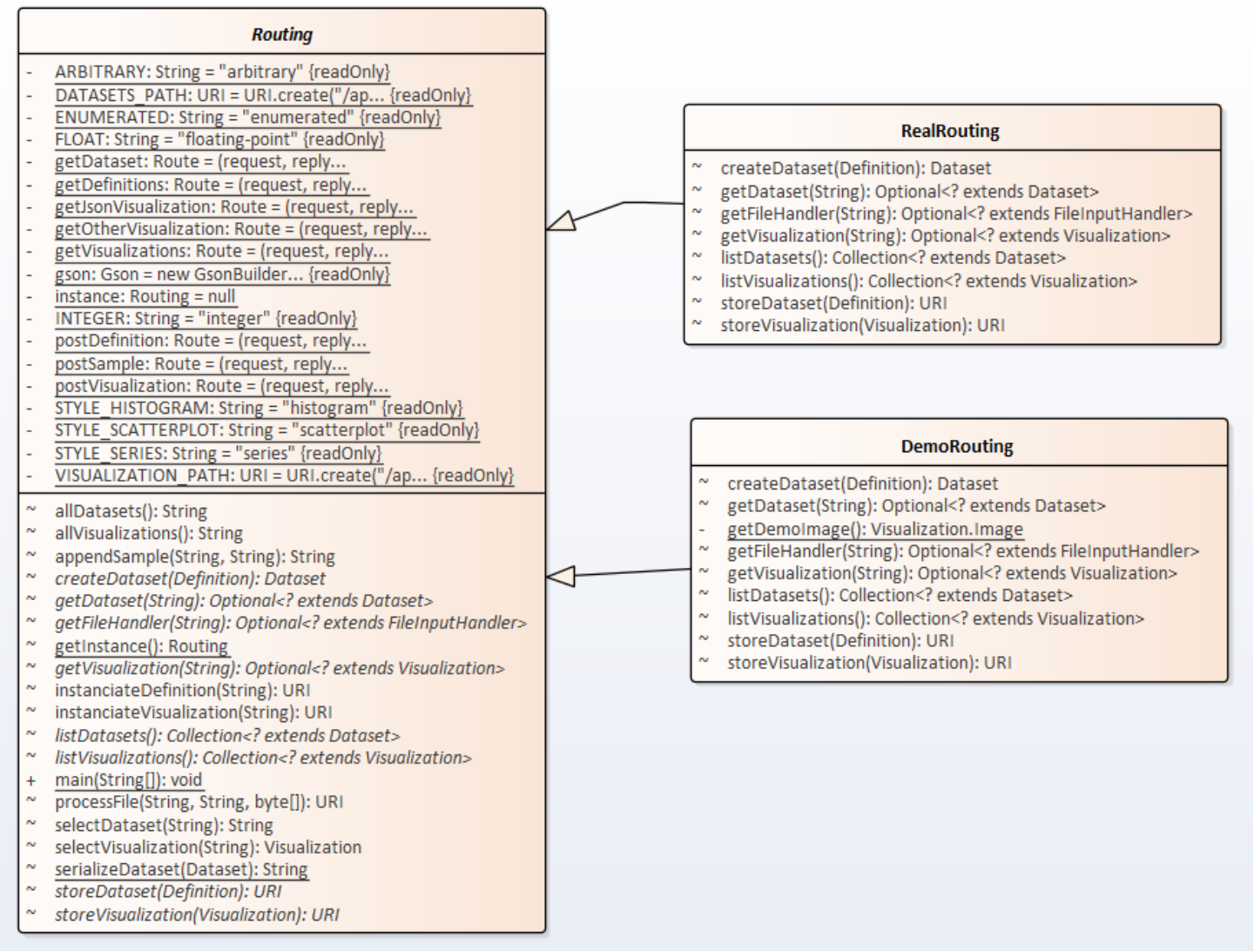



*Figure 16: Visualization Template Pattern*

### 5.4 Adapter Pattern

The adapter patterns used in the GSON builder to define UTF-8 encoded text JSON serialization and deserialization adapters for the various dataset objects and sub-objects to be stored within the permeant storage. The approach provides a wrapper for objects that do not conform to the expectations for the GSON converter client. The adapter pattern enables this project to augment or expand support for the transformation of objects to and from the UTF-8 encoded text JSON.

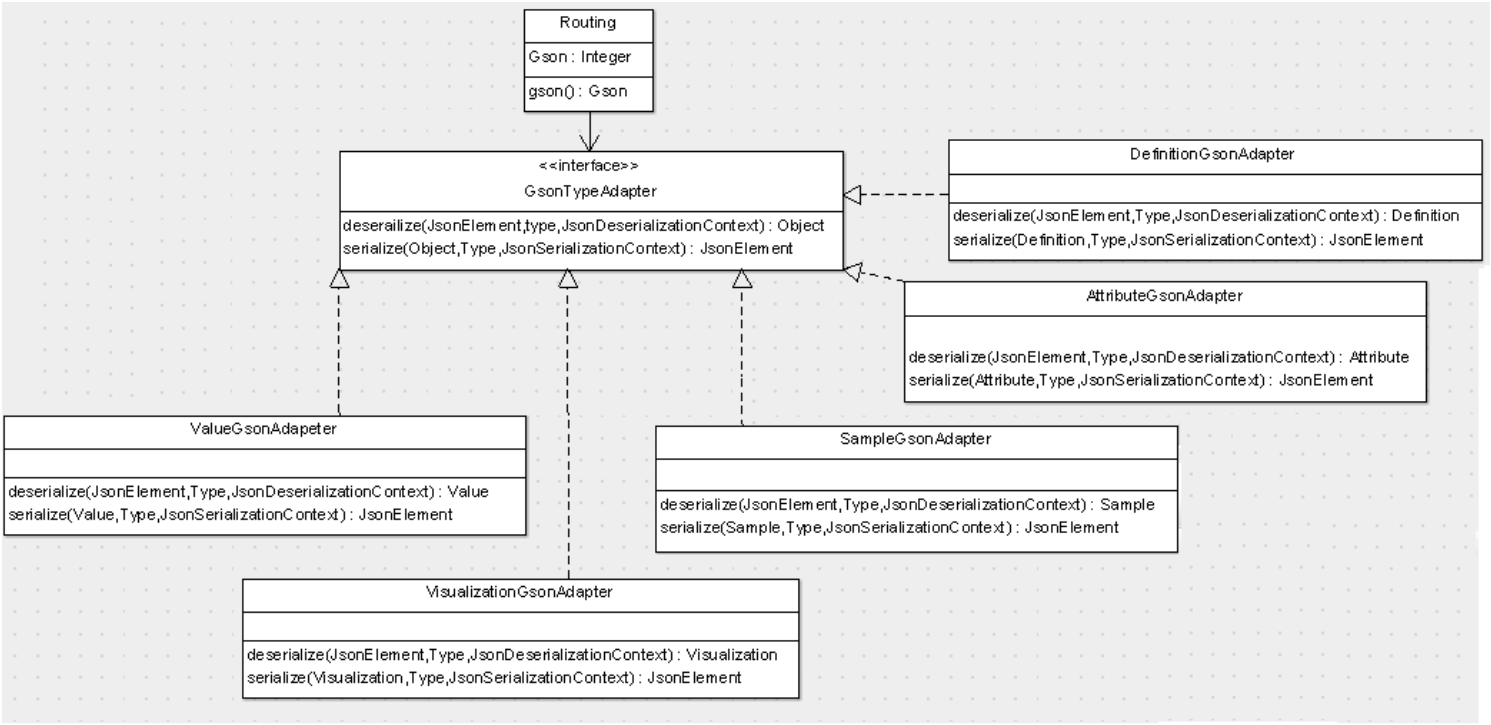



*Figure 17: GsonTypeAdapter - Adapter Pattern*

# 6 Appendix A JSON Data

| | |
| --- | --- |
| Data Definition Reference<br><br>Utilized by the client to locate existing Datasets. | {<br>name: "Demo Dataset",<br>location: "/api/datasets/any-old-id"<br>} |
| Dataset Structure<br><br>A named set of Samples defined by Attributes. | {<br>name: "Demo Dataset",<br>attributes: [],<br>samples: []<br>} |
| Attributes Example<br><br>Attributes within Dataset Definitions and Visualizations bound data. Attribute strategies include classes *Attribute.FloatingPoint*, *Attribute.Int*, *Attribute.Arbitrary* and *Attribute.Enumerated*. (see Figure 11: Attribute Visitor Pattern) | attributes:<br>{ name: "color", type: "enumerated", values: ["Red", "Blue", "Green"] },<br>{ name: "temperature", type: "floating-point", bounds: {max: 30, min: -5} },<br>{ name: "comment", type: "arbitrary"},<br>{ name: "capacity", type: "integer", bounds: {max: 500, min: 10} }<br>] |
| Samples Example<br><br>Values within Dataset Samples and Visualizations describe data. Value strategies include *Value.FloatingPoint*, *Value.Int*, *Value.Arbitrary* and *Value.Enumerated*. (see Figure 12: Value Visitor Pattern) | samples: [<br>{<br>capacity: {<br>type: "integer", value: 100},<br>color: {type: "enumerated", value: "Green"},<br>comment: {type: "arbitrary", value: "Arbitrary string 1"},<br>temperature: {type: "floating-point", value: 25}<br>},<br>{<br>capacity: {type: "integer", value: 100},<br>color: {type: "enumerated", value: "Green"},<br>comment: {type: "arbitrary", value: " Arbitrary string 2"},<br>temperature: {type: "floating-point", value: 25}<br>},<br>{<br>capacity: {type: "integer", value: 100},<br>color: {type: "enumerated", value: "Green"},<br>comment: {type: "arbitrary", value: " Arbitrary string 3"},<br>temperature: {type: "floating-point", value: 25}<br>} ]<br>} |
| Data Visualization Reference Example<br><br>Utilized by the client to locate existing Visualizations. | data : [<br>{<br>location : "/api/visualizations/histogram"<br>name : "Bear population of certain colors"<br>}<br>] |
| Histogram Data Visualization Example<br><br>JSON sterilization of class Visualization.Histogram (see Figure 10)<br><br>Displayed on “Visualize a Dataset” webpage using javascript. | data : {<br>attributes: [],<br>data [<br>{<br>bin: {type: "enumerated", value: "Blue" },<br>count: 7<br>},<br>bin:{type: "enumerated", value: "Green"} }, |

| | |
| --- | --- |
| | count:4<br>},<br>bin:{type: "enumerated", value: "Red"} },<br>count:17<br>},<br>],<br>dataset:"/api/datasets/histogram"<br>name:"Bear population of certain colors"<br>style:"histogram"<br>} |
| Series Data Visualization Example<br><br>JSON sterilization of class Visualization.Series (see Figure 10)<br><br>Displayed on “Visualize a Dataset” webpage using javascript. | data : {<br>attributes: [<br>{ name: "color", type: "enumerated", values: ["Red", "Blue", "Green"] },<br>{ name: "temperature", type: "floating-point", bounds: {max: 30, min: -5} }<br>],<br>data : [<br>{ type : "floating-point", value : 15 },<br>{ type : "floating-point", value : 17 },<br>{ type : "floating-point", value : 11 }<br>],<br>dataset : "/api/datasets/series",<br>name : "Temperature inside a comfortable room",<br>style : "series"<br>} |
| Scatterplot Data Visualization Example<br><br>JSON sterilization of class Visualization.Scatterplot (see Figure 10)<br><br>Displayed on “Visualize a Dataset” webpage using javascript. | data : {<br>attributes: [<br>{ name: "commanded-volume", type: "integer", bounds: {max: 0, min: 11}<br>],<br>data : [<br>{ x : {type: "integer", value: 5}<br>y : {type: "integer", value: 5} },<br>{ x : {type: "integer", value: 10}<br>y : {type: "integer", value: 10} },<br>{ x : {type: "integer", value: 11}<br>y : {type: "integer", value: 20} }<br>],<br>dataset : "/api/datasets/scatter",<br>name : "Alexa volume after given commands",<br>style : "scatterplot"<br>} |