Beulah Works LLC

User Manual
Version 1.0

UML Sequence Diagram File Generator

April 19, 2019

Isis Curiel, Bruno Hnatusko III, Brayden Mccoy,

Dhyey Patel, Jesse Primiani, Jacob Taylor,

and Syed Arshiyan Ali Zaidi

Approvals:

Title	**Signature**	**Date**
Project Manager	Isis Curiel	04/19/19
Integration Engineer	Dhyey Patel	04/19/19

Revision History

Date	Revision	Description	Author
04-18-2019	0.1	Created This Document	Jesse Primiani
04-18-2019	0.5	Added content to most sections	Jesse Primiani
04-19-2019	0.8	Added the remaining content	Jesse Primiani
04-19-2019	0.9	Added content	Dhyey Patel
04-19-2019	1.0	Final review	Dhyey Patel

Table of Contents

1 The SDMFileGenerator Library

This library provides a relatively easy to use interface for saving a Java object that holds Sequence Diagram information to a file. It provides two extensible adapters for both an input SDM library and an output file creation library, an extensible means of describing the output type, as well as an interface that manages these two adapters and the creation of a file from an input SDM object.

2 Installation

2.1. Adding this Library to a Project

To add this library to your project, simply add the compiled library jar file to your project's dependencies, alongside the input and output libraries that will be used with this library.

In the current version of this library, a re-compiled version of the umltranslator library is used for the input SDM, and it is included in the libs folder as umltranslator-0.0.7b.jar. Add this jar file as a dependency to your project to support the current input model.

For the output adapter, Aspose.Diagram version 19.2 was used to compile this project. This version of Aspose.Diagram should be added to your project via Maven. The VisioMasters folder must be added to the same folder as the your project's working directory, as it contains stencils in a master file needed by Aspose to generate sequence diagrams. Finally, an Aspose.Diagram.lic Aspose license file should be added to the project's working directory.

2.2. This Library's Folder Structure

- The root of this repository contains some information used by Git and Eclipse.
- The root buglist.txt contains a list of all known bugs in this library and the input library.
- The root Aspose.Diagram.lic is a license for Aspose.Diagram that is used by this library.
- The Documentation folder contains, inside subfolders for different file types, all the documentation and UML diagrams used in the development of this library.
- The doc folder contains all the Javadoc files for this library.
- The releases folder contains this library's compiled jar file.
- The libs folder contains the latest compiled version of the umltranslator library, as well as a previous stable version.
- The src folder's subfolder, SDMfileGenerator, contains all the source code for this library.
- The src folder's subfolder, VisioMasters, contains code used in the development of this library that was used to combine and read Visio master files.
- The root VisioMasters folder contains MasterSDM.vssx, which is the masters file that is needed by Aspose to generate a sequence diagram. All other master files in this folder were either used to create this file, or are backups of these files in different file formats.

2.3. Updating the SDM Library

To update Beulahwork's input SDM library, simply replace the umltranslator jar file dependency with a more recent version of that library. It can be done in this library by adding the new and updated jar file to the libs folder, then modifying the .classpath file to reference this new version instead of the old one.

2.4. Updating the Aspose Library

To update Aspose.Diagram, simply instruct Maven to download a different version of that library. It can be done in this library by modifying the contents of the version tag inside the Aspose-Diagram dependency, found within the pom.xml file.

3 Library Usage

3.1. Typical Usage

1) Have a UMLSequenceDiagram object created and available.
    a) This object will be called "diagram" in this example.
2) Wrap this diagram in a newly created input sequence diagram adapter object.
    a) Whenever the content of the "diagram" object is changed, a new adapter object has to be created to include these changes.
    b) The Beulahworks' SDM model is used in this example.
    c) `InputBeulahWorks inputAdapter = new InputBeulahWorks(diagram);`
3) Create the output adapter and output type objects.
    a) Aspose.Diagram and .vsdx respectively in this example.
    b) `OutputAspose outputAdapter = new OutputAspose();`
    c) `OutputTypeAsposeVSDX outputType = new OutputTypeAsposeVSDX();`
4) Create the SDMtoFile object.
    a) `SDMtoFile exportSDM = new SDMtoFile();`
5) Configure the SDMtoFile object to use the desired adapter, type, path, and file name.
    a) `exportSDM.setOutputAdapter(outputAdapter);`
    b) `exportSDM.setOutputType(outputType);`
    c) `exportSDM.setOutputFile("Folder", "Filename");`
6) Call the exportFile method with the input adapter and logging output to create the file.
    a) `exportSDM.exportFile(inputAdapter, System.out);`

3.2. Output Configuration

The SDMtoFile object has a few useful methods for configuring the output file that is generated by this library.

The setOverwrite(boolean overwrite) method can set to true to overwrite an already existing file, or to false to append an integer to the file name, the default is false.

The setOutputFile(String path, String name) method can be used to set the location and file name of the output file, the defaults are the working directory and "SequenceDiagram".

The setOutputType(OutputType<?> type) method can be used to change the file's type and extension by giving it an appropriate output type object, the default is Aspose's VSDX file type.

The setOutputAdapter(OutputAdapter adapter) method allows for different output adapters to be used in generating a file, the default is OutputAspose.

Finally, a few constructors are available that incorporates the functionality of these methods when creating an SDMtoFile object.

3.3. Logging & Exceptions

When SDMtoFile's exportFile method is called, and the sequence diagram's input adapter is passed into its first parameter, an optional second parameter for a PrintStream is available. If this parameter is set, logging information regarding the output file's export status will be printed to it as the method runs.

All methods in this library may throw an SDMException whenever a precondition is invalid. This SDMException will contain a String stating what went wrong, which can be retrieved by calling the standard Exception error message method on it. For example, ex.getMessage() for an SDMException called ex.

4 Library Maintenance

4.1. Adding More Output Types

More output types can be added through extending the OutputType class. For the Aspose library, the class that needs to be extended is OutputType<Integer>, since that library's file type information is stored as an Integer. Other libraries may use different objects or data structures to store file type information. Once a class that extends OutputType<...> is created, the getExtension() method must be defined to return the filename's extension string for the file type that this new class represents. The getType() method must then be defined to return the output library's required type object that represents the file type that this new class represents.

4.2. Creating or Modifying an Input Adapter Implementation

If the input model is updated, or a new library is added that handles SDM objects, then a new implementation of the interface class InputAdapter can be added that handles this new model. Simply implement all the methods in InputAdapter to handle reading the SDM information from this new input library, then use it as an adapter for this new or updated input model, and it will automatically work with any output adapter and the SDMtoFile class.

4.3. Creating or Modifying an Output Adapter Implementation

If the output library is updated, or a new library is added that exports SDM objects to a file, then a new implementation of the interface class OutputAdapter can be added that handles this new model. Simply implement all the methods in OutputAdapter to handle saving the SDM information for this new output library, then use it as an adapter for this new or updated output model, and it will automatically work with any input adapter and the SDMtoFile class. However, in the case of a new library being added, new extensions to the OutputType class must be created to contain the file type information used by this new output library.

4.4. Modifying the Input Adapter Interface

The input adapter interface would rarely have to be modified, as UML sequence diagrams rarely update or change their format. However, some aspect of a UML sequence diagram may not have been added to this library. In that case, this interface would have to be modified to incorporate that aspect.

To add the functionality required to support the soon to be added part of a sequence diagram, first the method(s) needed to retrieve information from the input model must be added to the interface class InputAdapter. Next, all implementations of this class have to incorporate these new methods, either by implementing their functionality or returning some default value if that is not currently possible. Finally, the exportFile method in the SDMtoFile class must be modified to make use of these new methods. The output adapter interface will have to be modified with methods to handle this new part as well.

4.5. Modifying the Output Adapter Interface

The output adapter interface would rarely have to be modified, as UML sequence diagrams rarely update or change their format. However, some aspect of a UML sequence diagram may not have been added to this library. In that case, this interface would have to be modified to incorporate that aspect.

To add the functionality required to support the soon to be added part of a sequence diagram, first the method(s) needed to add information to the output library must be added to the interface class OutputAdapter. Next, all implementations of this class have to incorporate these new methods, either by implementing their functionality or returning some default value if that is not currently possible. Finally, the exportFile method in the SDMtoFile class must be modified to make use of these new methods. The input adapter interface will have to be modified with methods to handle this new part as well.

4.6. Modifying or Adding to the Visio Masters file

The GenerateMasters Java file in the VisioMasters package included with this library contains Java source code using Aspose that can be used to read stencil information and create a Visio Master file. First, use Microsoft Visio to create or extract the required stencils, then save them to a .vssx masters file. Afterwards, the previously mentioned source code can be modified to read in these new stencils from this newly created masters file, read in any previously used stencils such as those contained in MasterSDM.vssx, then write a new .vssx file containing all the required stencils. This Java file can also be modified to simply print out all the stencils contained in a master file if that information in required.

4.7. Example Modification: Diagram Placement

A currently unused method, known as finalize(), is part of the output adapter. This method is called after all the parts of the output diagram are added to the output library's internal data structure object, but before the diagram is saved to a file. One of the uses of this method is to either implement, or call a method that implements, code to handle the placement of the diagram's objects. If the output library contains methods that handle diagram object placement, which is quite likely, then this method should be implemented to handle the placement of these objects.

5 Library Testing

5.1. Adding, Running, and Maintaining Unit Tests

All the Unit Tests for this project are contained in the UnitTests sub-package of this library. Each class being tested has its own unit test file, and each method being tested has one test method for it, with commented subsections denoting the test cases for that method. If a new class is added, then a new unit test file should be added to test it. If a new method is added to a class, then a new test method should be added to the unit test file for that class. If new test cases need to be added to a method, then a new commented try block should be appended to the related test method. Unit Tests are run via the standard way that JUnit runs in Java or your IDE.

5.2. Adding, Running, and Maintaining System Tests

All the System Tests for this project are contained in the SystemTests sub-package of this library. Currently, each system test represents a specific requirement found in the SRS, and is named in the form R_*_*. New system tests can be added that do not follow this naming scheme or test any specific requirements however. Each system test file contains a main method that can be used to run that test. Currently implemented system tests can also be run using a command line interface created by the main method of the TestingInterface Java file.

5.3. Adding, Running, and Maintaining Performance Tests

All the Performance Tests for this project are contained in the PerformanceTests Java file in the SystemTests sub-package of this library. New methods can be added to this file if more performance tests need to be done. A main method is contained in this Java file that runs and measures the time taken for each performance test, in terms of both CPU and Wall-Clock time. Currently implemented performance tests can also be run using a command line interface created by the main method of the TestingInterface Java file.