Software Requirements Specification (SRS) for "Suzu"

TophUwO

November 2023

Contents

Version History

Below is a chronological list of changes, alongside metadata. Version *1.0* marks the first release version of this document.

Date	Version	Contributor(s)	Changes
05/11/2023	0.1	TophUwO	Initial commit
07/11/2023	0.2	TophUwO	ADD base requirements and introductory paragraphs
12/11/2023	0.4	TophUwO	ADD pkgds, ucds and GUI mock-ups; now adhere more strongly to the IEEE 830 specification; clarified sentences and removed redundancies
13/11/2023	0.4.1	TophUwO	ADD more requirements plus FIX some redundant and outdated requirements
14/11/2023	0.4.2	TophUwO	ADD ucd for ProjectExplorer component plus clarify requirements
14/11/2023	0.4.3	TophUwO	ADD links to external references (section *External References*) and Section *Standard Assets*
14/11/2023	0.4.4	TophUwO	REMOVE requirement for additional shape sets; is now an optional requirement
14/11/2023	0.5	TophUwO	ADD ad for activity describing style override retrieval
14/11/2023	0.5.1	TophUwO	ADD requirement for default shapes inside the toolbox
14/11/2023	0.6	TophUwO	ADD appendix A and B (A - OOA cd, B - sd for *export model* uc)
14/11/2023	0.6.1	TophUwO	ADD requirement for languages English and German (other translations are optional features); ADD requirement of code-base language being *English (United States)*
15/11/2023	0.6.5	TophUwO	CHANGE document structure (figures are now all in the appendix); ADD list of required standard shapes

1 Introduction

1.1 Purpose of This Document

This document describes the specifications for the initial release version of an all-new diagram and modeling tool, codenamed and now referred to as *Suzu* within this document, aimed at maximum customizability and ease-of-use. This document was created in accordance with the *IEEE 830* standard, is part of the internal developer documentation, and is only a rough overview of what the system is supposed to do.

1.2 Document Conventions

In this document, technical terms relevant to the implementation are in *italics* while table heads are rendered in **boldface**. See section *Glossary* for an explanation of important technical terms used in the context of the product's domain.

1.3 Stakeholders and Audience

The only stakeholder is the authorizer, developer, and maintainer, TophUwO. This document is also meant for future contributors to the project of the software described within this document.

1.4 Background and Rationale

In modern software development, modeling tools are an integral part of object-oriented analysis and design. While the tools used for the job differ, they have one thing in common: They are heavy and complex applications, with limited support for customization. As a result, they are often tedious to get into and use. Additionally, they often provide obscure features not commonly needed in practice. These solutions become bloated very quickly. To make things worse, some modeling tools also have performance issues when dealing with complex diagrams.
With this application, I intend to provide a flexible and lightweight alternative to common modeling and diagramming tools. It aims to minimize hassle that is introduced by having to learn complex and tedious modeling tool-kits by providing an easy-to-use and self-explanatory interface with sensible features.
Furthermore, the application shall serve as a playground for applying important software development processes and concepts, commonly found in the real world. In no way do I intend to provide a do-all solution that massively improves upon all shortcomings of current established solutions.

1.5 Product Feature Overview

At its core, Suzu is a standalone visual modeling and diagramming tool, specialized but not limited to UML. In its base configuration, it shall provide unified and consistent tools to create all diagrams defined by *UML 2.0*. Furthermore, its base configuration may be extended with additional functionality aimed at creating different diagram types at the user's will.

1.6 Glossary

Below is a list of technical terms used in this document and their meaning in the context of this document. This list may not be exhaustive.

Term	**Explanation**	**Comment(s)**
Qt	cross-platform application framework	-
GPL	popular license for open-source development	-
polyline	line composed of a sequence of points that are joined by straight lines	-
layer	document structure, associating diagram elements with a global Z-position within the current view aside from relative Z-positions within the layer they are associated with	-

toolbox	tabbed view, grouping elements in categories, allows dragging of elements into the diagram editor	-
modeling project	collection of models	-
model	collection of diagrams	-
diagram	structured view of a composition of *diagram elements*	-
diagram element	a meaningful unit inside a diagram, e.g. *class*, *association*, *activity*, etc.	Each diagram element belongs to exactly one *element class*.
element class	type of an element, each class implements an UML language token	-
DND	drag-and-drop	-
HIDPI	high-DPI scaling, allowing apps to look similarly independently of the screen's PPI	-
content rectangle	rectangular area describing the smallest bounding rectangle enclosing a diagram element's contents, normalized to origin (0, 0)	-

1.7 External References

Below is a non-exhaustive list of useful resources that are likely to be frequently consulted throughout the conceptualization and implementation of the software described by this document.

Resource	**URL**
UML 2.0 specification	`https://www.omg.org/spec/UML/2.0/`
Qt 6 documentation	`https://doc.qt.io/qt-6/`
SVG specification	`https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Introduction`

2 General Description

2.1 Perspective

The software described by this document is a standalone application, supposed to be used by myself for everyday modeling tasks. A release is not planned as of now, but is thinkable if the application turns out worthwhile.

2.2 Product Features

The application should be able to create UML 2.0 diagrams by providing a *toolbox* holding all UML elements and a diagram view, represented by the *DiagramEditor* component. The user can drag elements from the toolbox into the diagram view in order to add them to the diagram. Furthermore, there should be a tree-view, represented by the *ProjectExplorer* component, displaying the current document structure as a tree showing parent-child relationships. The last view in the main window should be a property view, allowing easy and quick viewing and editing of the currently-selected diagram element's properties. It should be structured in a table-like form.
Diagram elements should have connection points for connections (*association*, *collaboration*, etc.) to dock onto. The number and position of these connection should be configurable for each diagram element individually.
Another main feature is a rich customization system that allows styling of all elements of the diagram. The same can be done to configure the look and feel of the application itself.
The third and last main component is a plug-in system that allows users to create plug-ins for the application itself, extending functionality which can then be accessed through menus and dialogs.

2.3 Application Users

The application is to be used by software engineers, scholars in that field, and software engineering students. It can be assumed that the users of the application are familiar with UML and modern modeling concepts.

2.4 User Classes and Characteristics

The application does not feature any other user class than the default one, encompassing all users of the application in the same way. Therefore, there is no user hierarchy, permission system, or any other component typically linked to user accounts.

2.5 Operating Environment

The application does not directly interact with hardware, but uses interfaces provided by the host platform. Similarly, there is no direct communication with any other software product.

2.6 Developer Limitations

There are little limitations in place. The application should be written in C++ using the Qt application development framework. Qt should be version 6.6 or later. Third-party frameworks can be used if they are not licensed under a strong copyleft license like *GPL*. Inside the Qt ecosystem, all APIs that are not solely available under a strong copyleft license can be used freely. Plug-ins extending the functionality of the application should be written in C++ and added into the application via shared libraries and a global plug-in path. Another type of plug-ins, called *resource plug-ins*, can be used to override the default resources used by the application (i.e. icons, styles, dialogs, etc.). Persistent storage for application settings should be JSON documents.
The language used in the code-base must be *English (United States)*.

2.7 User Documentation

Aside from the developer documentation provided by this document, there will be an online documentation available. The documentation must be accessible from the application (via a hyperlink or an embedded manual). The developer documentation represented by this document can also be accessed by users. The format of the user documentation is HTML. A printable PDF edition is not planned.

2.8 Assumptions

Qt is cross-platform. The application should be compilable and runnable on all major desktop operating systems. Releases and optimization for mobile devices are not required.

2.9 Optional Features

The main optional feature is code generation from class diagrams. This should include a validity check of the working diagram. The languages that the application should be able to generate code for is C++ and Java. Other code generators can be supplied via plug-ins.
Another optional feature is the inclusion of more diagram elements used by other UML 2.0 diagrams. These diagram elements may be either shipped with the application itself, or provided using plug-ins.
Internationalization is imperative in today's globalized world. Therefore, the application may provide translations into languages other than the ones listed under section *Internationalization*.

3 Core System Features

3.1 Modeling and Diagramming

The user is able to create and edit *models* comprising one or more *diagrams* where each *diagram* contains an arbitrary number of *diagram elements*. Optionally, *models* and *diagrams* can be encompassed by an overarching *modeling project*. The diagram editor supports *DND* and zooming behavior. Diagrams can be exported in a variety of image formats, both raster- and vector-based. The options for model exporting (referred to as *model encoding strategies*) can be extended by the means of plug-ins.

3.2 Styling of Diagrams and the Application

The user should be able to provide custom style overrides for all *element classes* and even all elements individually. Furthermore, the user should be able to provide custom application styles either by creating them or by loading them via plug-ins.

3.3 Extension Framework

The application should feature a rich plug-in framework, designed to make developing new functionality for the core application as simple as possible. Plug-ins can extend the UI, the functionality, can override resources, can add new resources, and can add model encoding strategies.

4 External Interface Requirements

4.1 User Interfaces Overview

Generally, the application is a graphical UI. It uses dialogs for user input and uses the mouse and the keyboard extensively (for shortcuts, etc.). Data is submitted through submit buttons present in all dialogs. Standard dialogs like the *File Open dialog* or the *Choose Color dialog* as well as *Message Boxes* will be native dialogs provided by either the host OS or the respective Qt implementation. The main window features a menubar, utilizing classic drop-down menus where menu items have icons where sensible. Additionally, below the menubar, one or more toolbars can be placed to allow quick access to commonly-used actions. These toolbars change state depending on what view widget currently has focus.
On the left side of the main window, the application will display the visualization of the *ModelExplorer* component, allowing the user to view the current model and diagram hierarchy. On the right side of the main window, there will be the toolbox, and below the toolbox will be the *PropertyEditor* component. The central widget is the *DiagramEditor* component. The default window layout described here should be changeable, savable and restorable.
Regarding *HIDPI*, the application must support high-DPI scaling.

4.2 Hardware and Software Interfaces

The application does not implement any device drivers itself. Therefore, all interaction with hardware is done through host OS-provided interfaces. Similarly, the application does not interact with any third-party software directly. The application framework used for building the application is Qt 6.6 or newer.

5 Specific Requirements

5.1 Functional Requirements

5.1.1 Modeling and Diagramming

The user must be able to create a new *modeling project*, a new *model*, or a new *diagram* from a dialog. A modeling project can contain multiple models while a model contains multiple diagrams. Models and diagrams, however, can also be created *standalone*, that is, without a modeling project or a model as a parent. Diagram elements are by default global in the context of the modeling project, but can also be made private to a specific model or even a specific diagram. A modeling project, a model, and a diagram should have properties like *name*, *description*, *version*, and *comments*. Deleting an item from the model/modeling project/diagram should delete all its children. Furthermore, all references to that item must be removed as well.
Diagrams should have support for multiple layers that can be either *visible* or *invisible*. These layers can be encoded individually or as one file (default). Working diagrams are saved to the disk using XML. They can be imported into either the current diagram (to extend the current working diagram), or into a new diagram view.
Diagrams can be exported in a variety of different formats. The central output format should be *Scalable Vector Graphics (SVG)* and *Portable Network Graphics (PNG)*; other formats are optional and can be provided via plug-ins.
The diagram view should be a tabbed window, allowing the editing of multiple diagrams at the same time. There should be copy-paste behavior across working diagrams. A diagram element can be in multiple diagrams at the same time, provided these diagrams and their diagram elements are part of the same model or are part of an imported model. Each diagram element inside a diagram should be able to be styled and transformed independently from their identical elements in other models.
In the diagram window, if an item is moved, the position and orientation of the its connections should be adjusted accordingly. Connections should be able to be broken up into *poly-lines* by dragging any point along the connection line given it does not overlap with an arrow element of the line. Item dragging should be done by the mouse. It must be possible to clip all nodes (corner points of boxes, line points, etc.) to the grid. This should allow straight and angled lines. Most diagram elements must support resizing; they should be growing indefinitely but not shrink smaller than their smallest *content rectangle*. If an action is performed that would increase/decrease the content rectangle, the diagram element should be resized accordingly.
The diagram view should feature a dynamic grid that resizes according to the current zoom factor. If the grid gets too large, it should automatically switch to a smaller, more fine-grained grid, and vice-versa. There should be a way to customize the appearance of the grid. A way to do that should be provided through a global and diagram-item-specific settings dialog that should be accessible via right-click or the menu bar.
Application settings are JSON documents and can be saved and loaded from those documents using a dialog in the application settings. There must be a way to restore factory defaults even if the default configuration was changed without a backup.

5.1.2 Styling of Diagram Elements and Application

Styling has to be accomplished through user dialogs, where everything, from global styles (for all diagram element classes), to individual element classes, to individual element instances, down to individual element references can be freely customized. These styles are referred to as *style overrides*. The main customization options should be *text color*, *background color*, *line color*, and *line style*. These attributes may not be significant for all diagram element classes, and some classes may add additional attributes. Styling of element classes can be done individually by right-clicking a diagram element inside the *Project Explorer* or a reference inside the *Diagram View*. Additionally, styles can be exported to CSS files which can then be distributed and imported into other units of the application. The component responsible for managing the styling of diagram elements is the *DiagramStyler*.

5.1.3 Extension Framework

Functional extensions are provided by the means of native shared libraries, implementing new functionality using the C++ programming language. Every plug-in should have a *config.json* file in the root directory of the plug-in. This file holds additional information as to what this plug-in represents and

where resources used by the plug-in are stored. Furthermore, the configuration holds meta-information on the plug-in itself, like *name*, *version*, *author(s)*, etc. The *version* attribute contains version information in the *major.minor.patch* schema. There should be a way to compare versions component by component.
The way application resources are exposed to plug-ins is through a special interface that exposes certain UI components or functional components of the application. The implementation of this interface must be supplied by the application; its API specification, however, should be defined by the plug-in framework.

5.1.4 Standard Assets

Aside from built-in resources like icons and other resources required by the application, the application should be shipped by default with all elements necessary to create *UML 2.0 Package Diagrams*, *UML 2.0 Use-Case Diagrams*, *UML 2.0 Sequence Diagram*, *UML 2.0 Activity Diagrams*, and *UML 2.0 Class Diagrams*. The toolbox should group the shown diagram elements according to which diagram they normally belong to. Aside from meaningful diagram elements, the toolbox should also provide access to standard shapes and lines, as well as miscellaneous items such as *text fields*. These items, just like diagram elements, can be styled in the same way and to the same degree.
The standard shapes that are to be implemented are:

- rectangle
- triangle
- ellipse
- n-polygon
- line
- poly-line
- bézier curve
- path
- text
- image

More shapes are optional features. These can be added through plug-ins.

5.2 Non-functional Requirements

5.2.1 Performance

There should be no noticeable lag when moving diagram elements. Zooming should be performed fluidly as well. If an action is expected to take longer than 0.5 seconds, a progress bar indicating progress should be shown. Where possible, GPU-accelerated rendering should be used.

5.2.2 Internationalization

The application's default language is *English (United States)*. Additionally, the application must provide a way to change the GUI's language to *German (Germany)*. Translations into languages other than English or German are optional features. There should be a way to install new languages.

A GUI Prototypes

B OOA Diagrams

B.1 Use-Case Diagrams

B.2 Sequence Diagrams

B.3 Activity Diagrams

B.4 OOA Class Diagram