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13/11/2023	0.4.1	TophUwO	ADD more requirements plus FIX some redundant and outdated requirements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
17/11/2023	0.6.6	TophUwO	ADD description for selection algorithm used by the *PropertyEditor* component
18/11/2023	0.6.7	TophUwO	CHANGE application style requirement to *optional*
19/11/2023	0.7	TophUwO	ADD ucds + OOA pkgd + OOA cd to the appendix
23/11/2023	0.7.1	TophUwO	User now selects elements from the toolbox by left-click; DnD is now an optional feature
25/11/2023	0.7.2	TophUwO	ADD functional requirements for plug-in manager; ADD search function requirement plus optional requirement for *DiagramPreview* view component
25/11/2023	0.7.3	TophUwO	ADD table describing the software quality requirements according to *ISO/IEC 25010:2011*
30/11/2023	0.8	TophUwO	ADD cd for prop hierarchy; CHANGE text order in diagramming description (makes more sense now); REMOVE unused sections; ADD requirement for automatic saving
30/11/2023	0.9	TophUwO	CHANGE explanation of several requirements + minor typo and format fixes
30/11/2023	1.0	TophUwO	**INITIAL RELEASE VERSION**

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 serves as a rough overview of what the system is intended to do and be used for.

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.
Figures such as GUI mock-ups and OOA diagrams can be found in the Appendix A and B, respectively.

1.3 Stakeholders and Audience

The only stakeholder is the authorizer, developer, and maintainer, TophUwO. This document is also meant for future contributors 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, we 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 we 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 the main diagrams defined by *UML 2.0* (required diagrams listed in section *Standard Assets*). 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	-

document	used as a blanket term for a *modeling project*, a *model*, and a *diagram*	If not specified otherwise, this term always refers to all three given terms at once.
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*.
diagram 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)	-
widget	visible UI element that can be interacted with	-
viewport	rectangular area of currently visible contents inside a view widget	-

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**
IEEE 830 specification	`http://www.math.uaa.alaska.edu/~afkjm/cs401/IEEE830.pdf`
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`
URN specification	`https://datatracker.ietf.org/doc/html/rfc2141`
ISO/IEC 25010:2011	`https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`

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 select elements from the toolbox 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. Optionally, the same could be done for the application's look and feel as well.
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. The application is to be used with a mouse and keyboard. Touchscreen support is not planned.

2.6 Developer Limitations

There are little limitations in place. The application should be written in C++ using the Qt (widgets) 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 SRS provided by this document, there will be a *DOXYGEN*-based online developer documentation available. Furthermore, there must be a user documentation, outlining the application's

features. The user documentation must be accessible from the application (via a hyperlink or an embedded manual). The SRS 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 Distribution

The application's main distribution format is compiled and packaged, ready to be installed. The application should also be available as a portable version. The application's source code is not to be distributed alongside the compiled binaries. A release of the application's full source code under a non-copyleft open-source license is thinkable. Details regarding this must be worked out in time.

2.10 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. In the same way, there could be plug-ins that extend the basic shape library.
Internationalization is imperative in today's globalized world. Therefore, the application may provide translations into languages other than the ones listed under section *Internationalization*.
The *StyleEngine* component may be extended to also allow the user to load application styles aside from just diagram styles. These styles should then be able to alter the look and feel of the application's user interface.
The application can be extended to feature another view component, a *DiagramPreview* that displays the current working diagram in a small window, alongside a movable rectangle outline, representing the current viewport in the diagram view. If the rectangle is moved, the diagram view scrolls its contents to reflect the new viewport position.
Optionally, there may be an undo-redo function for most actions, with configurable history length.

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 Diagram Element Customization

The user should be able to provide custom style overrides for all *element classes* and even all elements and element references individually.

3.3 Plug-in 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, override resources/assets, add new resources/assets, 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 the respective 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.
Elements are selected from the toolbox by clicking on them or by the means of a hot-key. To add them to the diagram, the user has to left-click the diagram view. The selected element will be inserted and positioned in a way that the upper-left corner of the element corresponds to the mouse position (relative to (0, 0) in diagram coordinates) at the time of the left-click. Optionally, drag-and-drop behavior should be implemented to accomplish the same task with analogous properties.
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 *ID*, *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. The *ID* field should contain an identifier akin to a *Uniform Resource Name (URN)*. This identifier must be unique across the document. Duplicate identifiers must be rejected. If the user wishes to do so (via settings), identifiers should be automatically generated from the *name* properties of all parents of the current element. For example, if there is an element with name *dia* and this diagram has a child with name *class1*, then the URN of the element with name *class1* should be *dia:class1*. Consequently, if *class1* now has a child called *ident42*, *ident42*'s identifier would be *dia:class1:ident42*. To prevent namespace pollution, the project explorer should support an operation that removes all unused (i.e. non-referenced) diagram elements from the either the currently selected diagram document, or all open documents (as long as they are not read-only).
In the *PropertyEditor* window, its content depends on the currently selected element/item. If there is a diagram element (or diagram element reference) is in focus inside the *DiagramEditor* window, its properties should be shown. If not, the properties of the working diagram inside the currently visible tab should be shown. If there are none, the properties for the current model should be shown, or, if there is no model, the window should allow the editing of the current modeling project properties. If no modeling project is open, the property window should show nothing. Items can also be selected inside the *ProjectExplorer* window, but selections inside the *DiagramEditor* window should take precedence.
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. Diagram elements themselves can be made invisible individually, which makes all children and connected edges invisible as well.
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. The title of each tab should be the current working diagram's *name*, should be closable. If a tab is closed, the working diagram must be saved if required. 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.
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. There should be a way to hide the grid, e.g. by the means of a toolbar action. 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 and edges (corner points of boxes, line points, etc.) to a grid of a fixed unit size which can be modified in the grid settings dialog. 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.
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.

The diagram view should feature a text-based search function that allows the user to search for a diagram element (reference) using its URN or name. If an element satisfying the search term is found, the diagram view will scroll to the current position of that element. If more than one elements were found, navigation buttons will appear that allow the user to traverse all search results bidirectionally. The search function can be implemented as a widget or a non-modal (but always foregrounded) dialog.
Additionally, there should be a way to set an automatic saving interval. The minimum automatic saving interval should be 30 seconds, the highest should be one hour. Between that, the saving interval can be set in one-minute steps.
When the application is closed, if there are pending changes, the user should be asked whether or not to close the application. The user should have the choice between closing and saving, closing and discarding, and aborting.

5.1.2 Diagram Element Customization

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 by the application. The component responsible for managing the styling of diagram elements is the *StyleEngine*.

5.1.3 Plug-in 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.
To manage plug-ins, the user has access through a special dialog available in the settings which allows the addition and removal as well as reloading of plug-in instances. Plug-in instances are a folder inside the plug-in's root directory.
The way the application adds resources and/or extends its functionality is through callback functions that "ask" the plug-in for widgets and component instances. These are then validated and added to the application's feature pool. The application will then take ownership of the resources.

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
- (regular) n-polygon
- star

- line
- poly-line
- bézier curve
- path (composed of *lines*, *poly-lines*, and *bézier curves*)
- text
- image (common image formats such as JPG, PNG, SVG, etc.)

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. When using GPU-accelerated rendering, Qt's built-in *QOpenGLWidget* should be used where supported. Rendering using other technologies (i.e. *Direct2D*, *Vulkan*, etc.) are completely optional features.

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. Language packs should be distributed as Qt **.qm* files generated from Qt's own in-ecosystem internationalization tool, *Qt Linguist*.

5.2.3 Software Quality Requirements

Below is a list of all *ISO/IEC 25010:2011* software quality requirements alongside an integer value, the level of importance, in the interval [1, 5] where *5* denotes very high importance and *1* denotes low importance relative to level *5*.

Requirement	Description	Priority
functional suitability	This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.	**5**
performance efficiency	This characteristic represents the performance relative to the amount of resources used under stated conditions.	**3**
compatibility	Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment.	**1**
usability	Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.	**3**
reliability	Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.	**4**
security	Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.	**2**

maintainability	This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.	**5**
portability	Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.	**2**

A GUI Prototypes

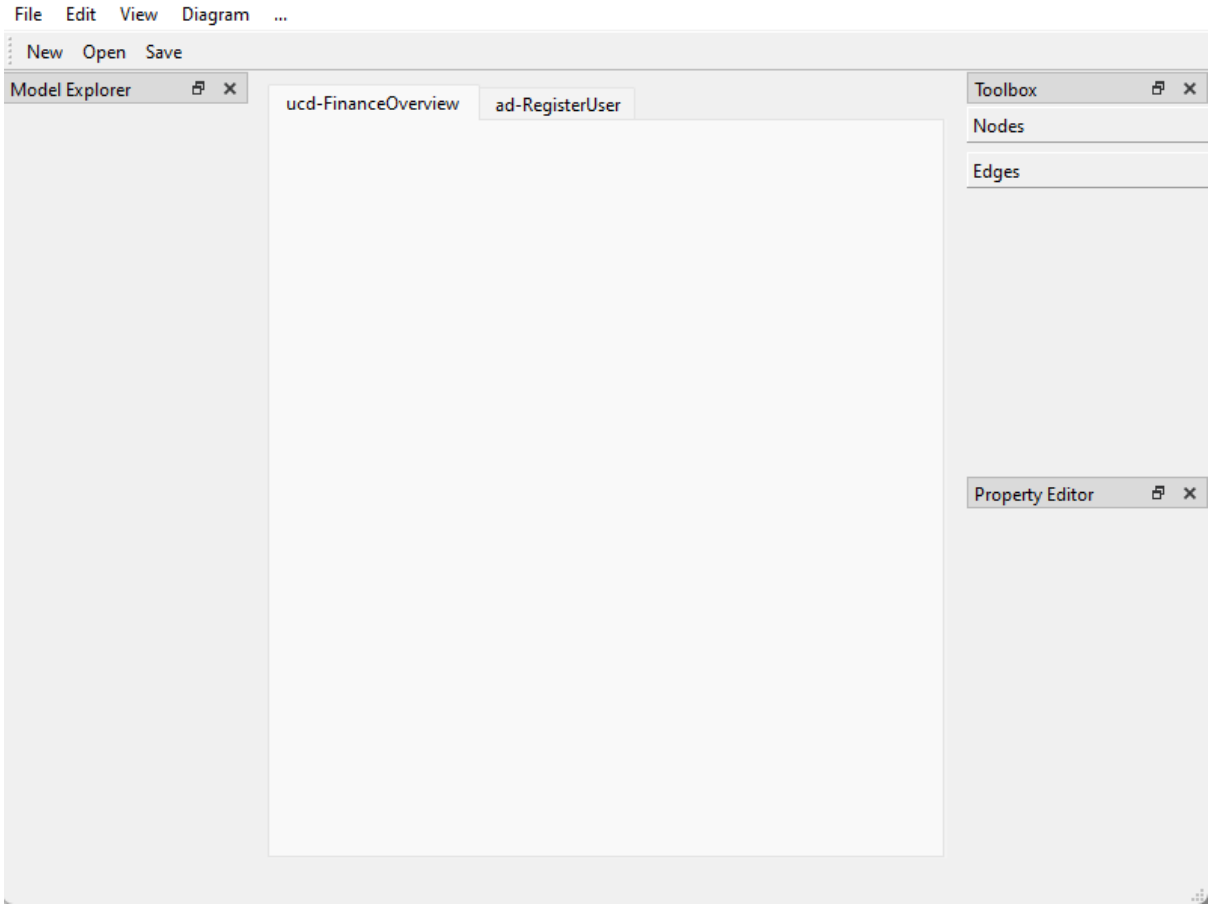

Figure 1: GUI prototype of the main window, showing the application in action

B OOA Diagrams

B.1 OOA package diagram

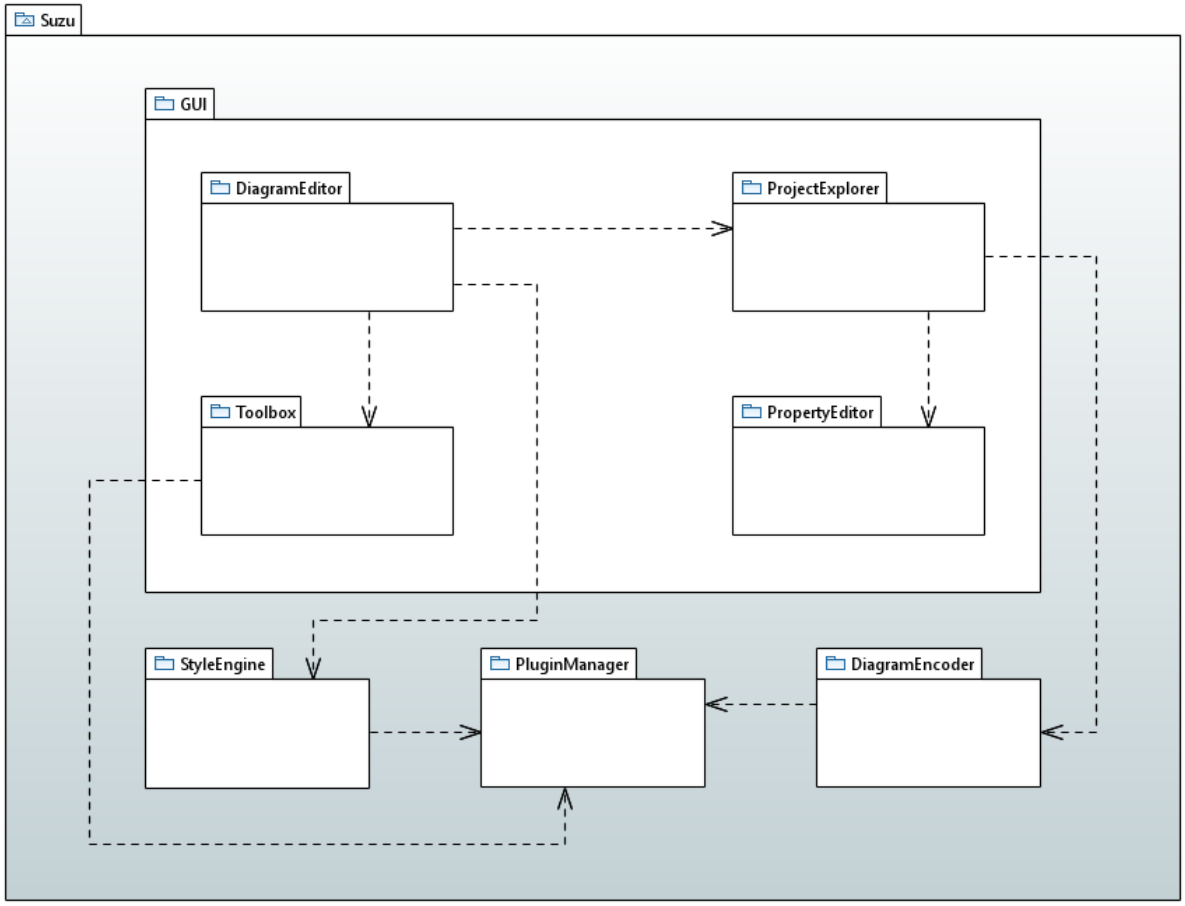

Figure 2: Top-level architecture diagram (as a Package Diagram) of Suzu

B.2 Use-Case Diagrams

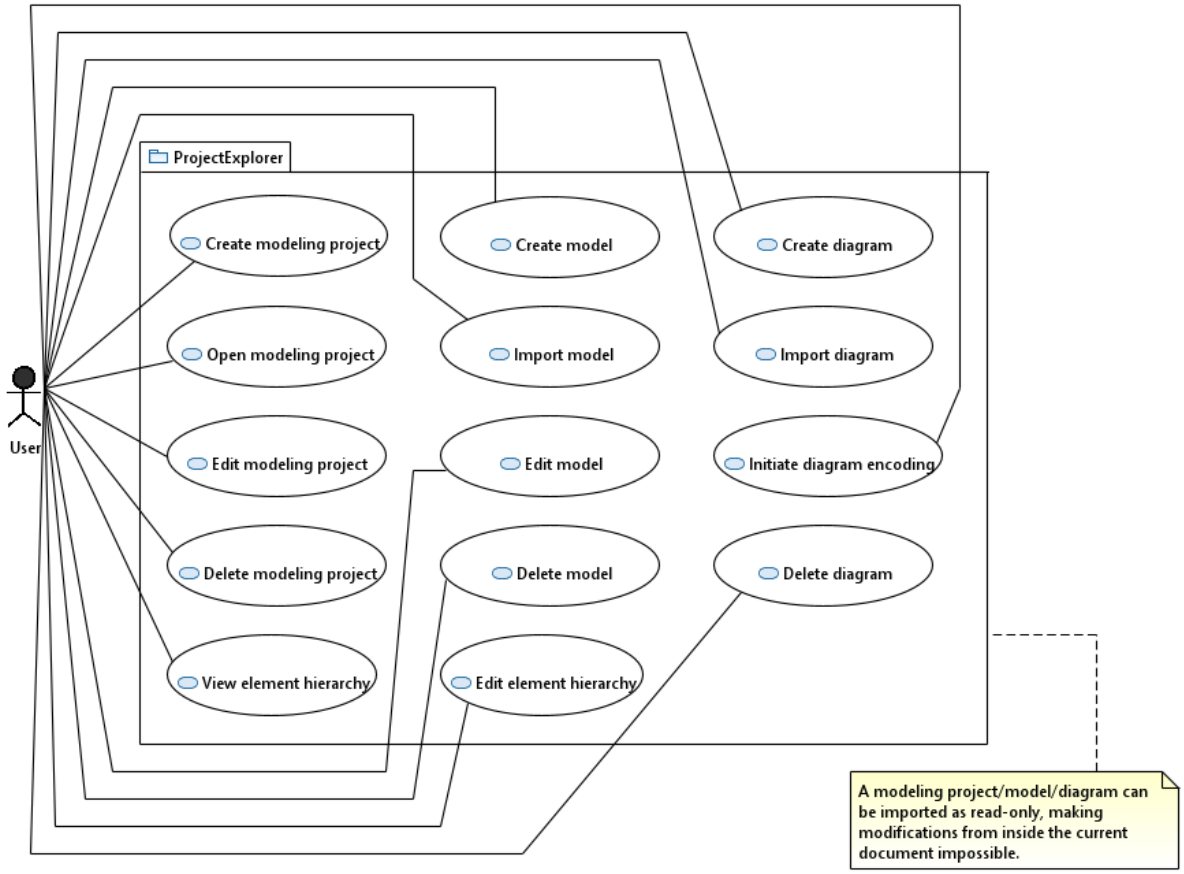

Figure 3: ucd of the *ProjectExplorer* component

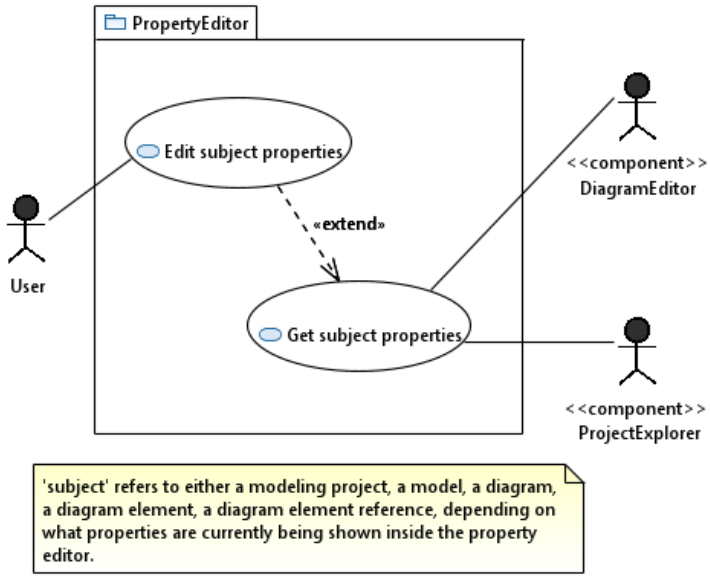

Figure 4: ucd of the *PropertyEditor* component

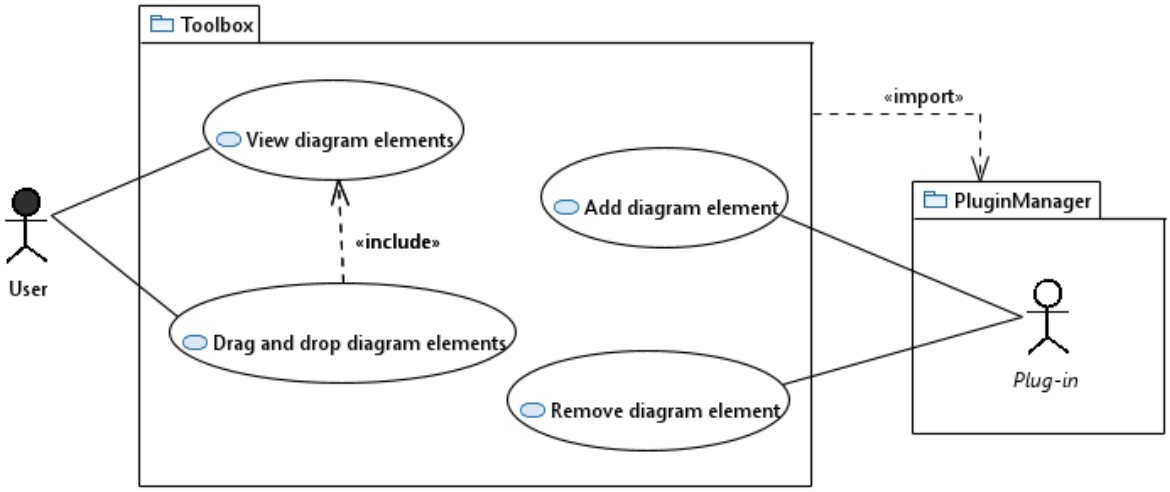

Figure 5: ucd of the *Toolbox* component

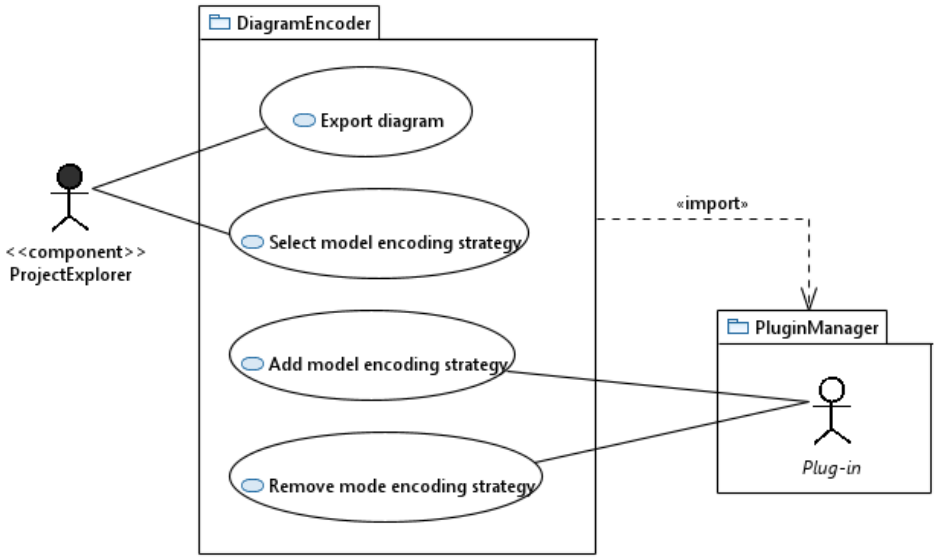

Figure 6: ucd of the *DiagramEncoder* component

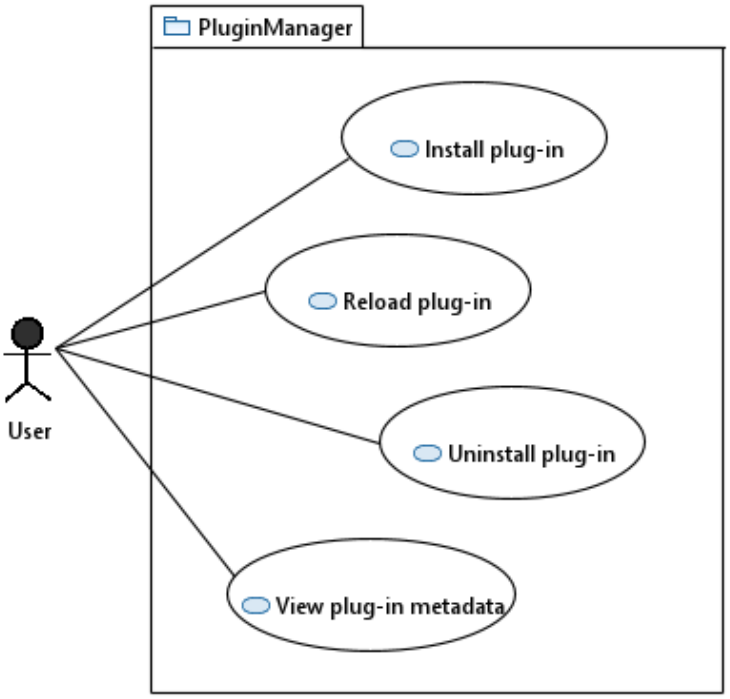

Figure 7: ucd of the *PluginManager* component

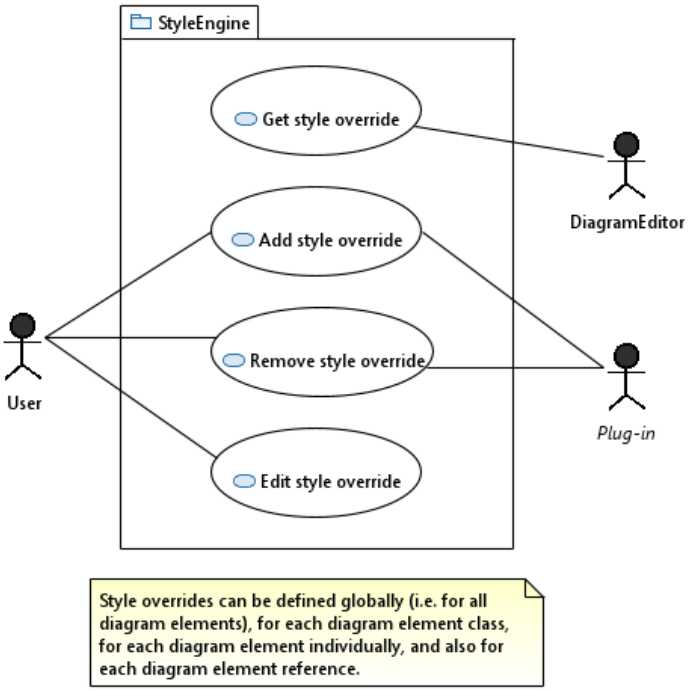

Figure 8: ucd of the *StyleEngine* component

B.3 OOA Class Diagrams

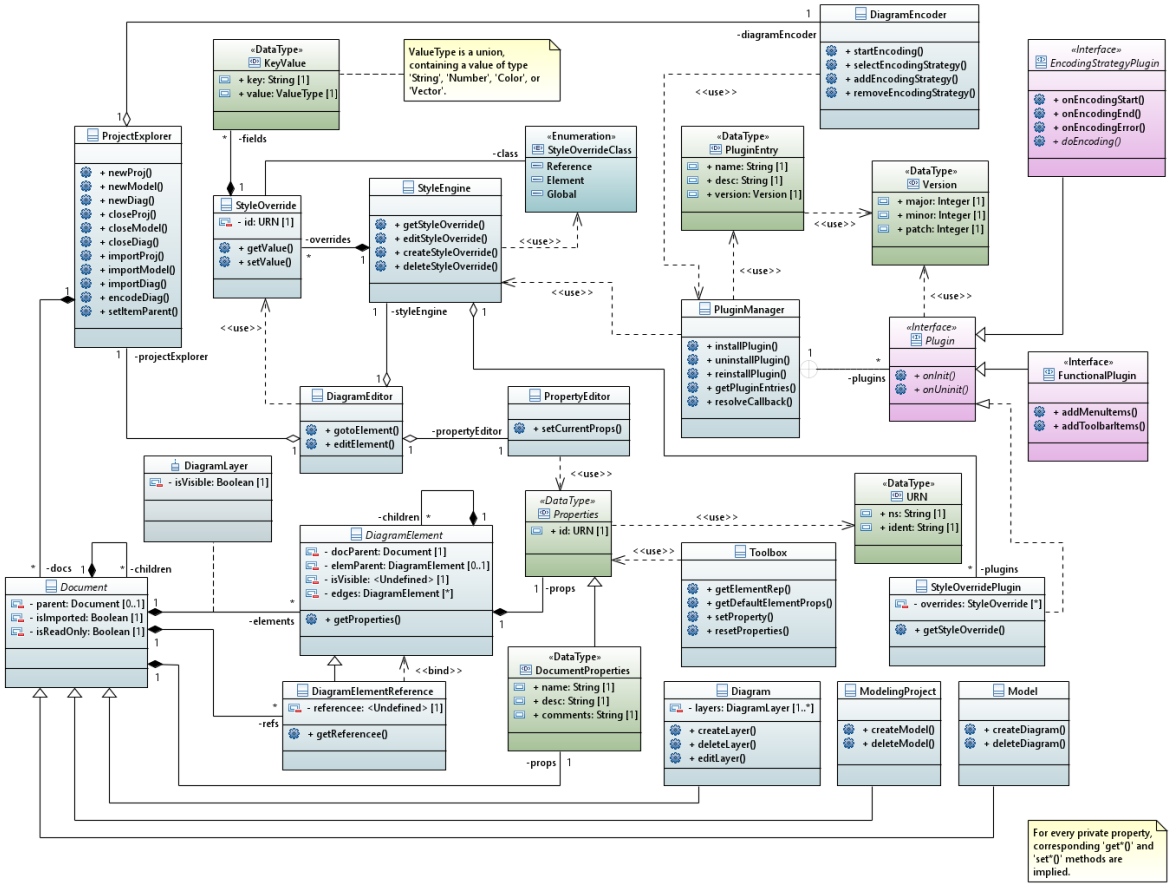

Figure 9: OOA class diagram of Suzu's components

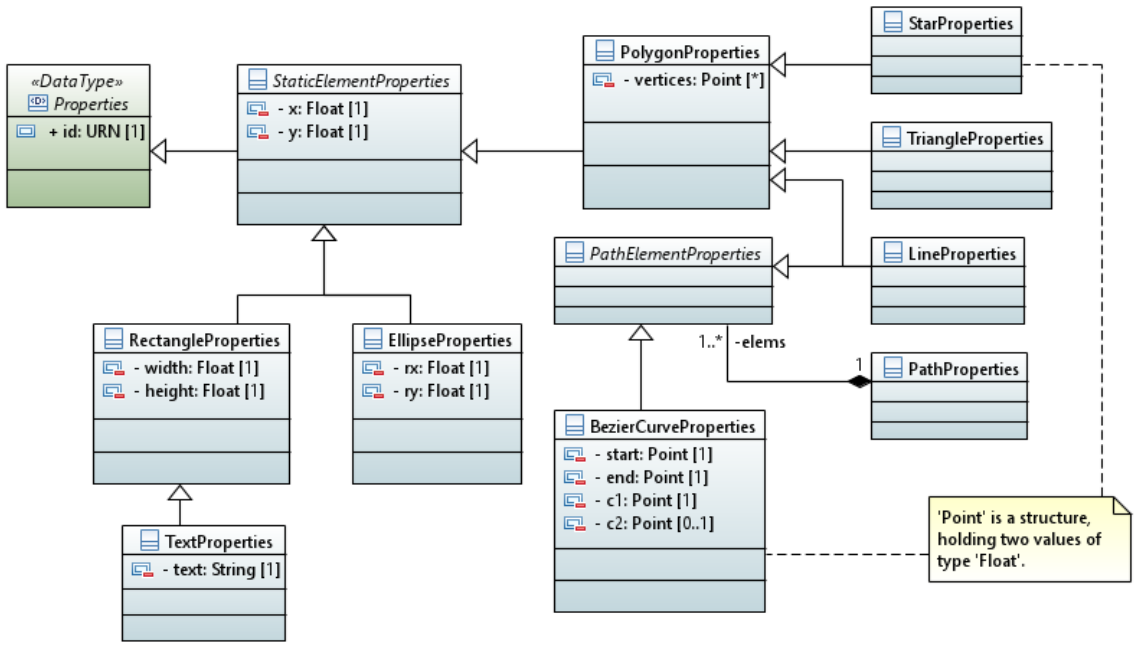

Figure 10: cd depicting static shape (see *Standard Assets*) property hierarchy