JHotDraw 8: Software Specification

Table of Contents

List of Figures

JHotDraw 8: Software Specification

Table of Contents

List of Figures

JHotDraw 8: Software Specification

Werner Randelshofer

Chapter 1. Introduction

Table of Contents

Purpose

This document specifies the requirements and the design of "JHotDraw 8".

Definitions

List of terms:

Action

> An action is a command that can be invoked with a graphical user interface.

Activity

> An activity is a set of actions that relate to a topic of some kind.

Application

> An application is a computer program with a graphical user interface that can be used to perform one or more activities.

Document

> A document is an editable dataset that can be stored in a file.

Document-based Activity

> A document-based activity is an activity with actions that relate to a document.

Document-based Application

> A document-based application is an application that supports document-based activities.

Drawing

> A drawing is a collection of figures presented on a canvas.

Figure

> A figure is a visual representation of some kind (such as a line, a shape, a diagram element or something more complex).

Project

A project is an editable dataset that can be stored in a directory structure.

Project-based Activity

A project-based activity is an activity with actions that relate to a project.

Project-based Application

A project-based application is an application that supports project-based activities.

License

JHotDraw 8 can be licensed under one of the following license models:

- The MIT License

The MIT License

Resources

- The JHotDraw 8 web-site.
- The JHotDraw 8 github-site.

References

Bibliography

[MacHIGuidelines] . Copyright © 2019 Apple, Inc.. *macOS Human Interface Guidelines*. https://developer.apple.com/design/human-interface-guidelines/ .

[WinUXGuidelines] . Copyright © 2016 Microsoft, Inc.. *Windows User Experience Interaction Guidelines*. https://docs.microsoft.com/de-de/windows/desktop/uxguide/ .

Chapter 2. Overall description

Table of Contents

Product Perspective

"JHotDraw 8" is an object oriented framework for developing drawing editors in the Java programming language.

It consists of two independent frameworks, and a number of demos ("samples"), which demonstrate typical uses of the framework.

Frameworks:

- the "Drawing Editor Framework"
- the "Application Framework"

Example programs:

- the "Mini" examples
- the "Grapher" example
- the "Modeler" example
- the "Teddy" example

The "Drawing Editor Framework" can be used to realize drawing editors for computer aided design and for artistic drawings. Drawings can be animated and interactive. It is possible to back a drawing with a data model so that a drawing editor can be used as a user interface for the data model.

The "Application Framework" can be used to create document-based applications that comply to platform-specific user interface guidelines. Such as [MacHIGuidelines] and [WinUXGuidelines].

The "Mini" examples demonstrate small use cases from both frameworks.

The "Grapher" example demonstrate a working drawing editor application. It focuses on many use cases of the application framework and the drawing framework. It exercises the "simple" implementation classes that are provided with the drawing framework.

The "Modeler" example demonstrate a working drawing editor application. It focuses on many use cases of the application framework and the drawing framework. It adds implementation classes that exercise the extensibility of the drawing framework.

The "Teddy" example demonstrates a working text editor. It focuses on the application framework. It exercises the "simple" implementation classes that are provided with the application framework.

Chapter 3. Requirements

Table of Contents

Application Framework

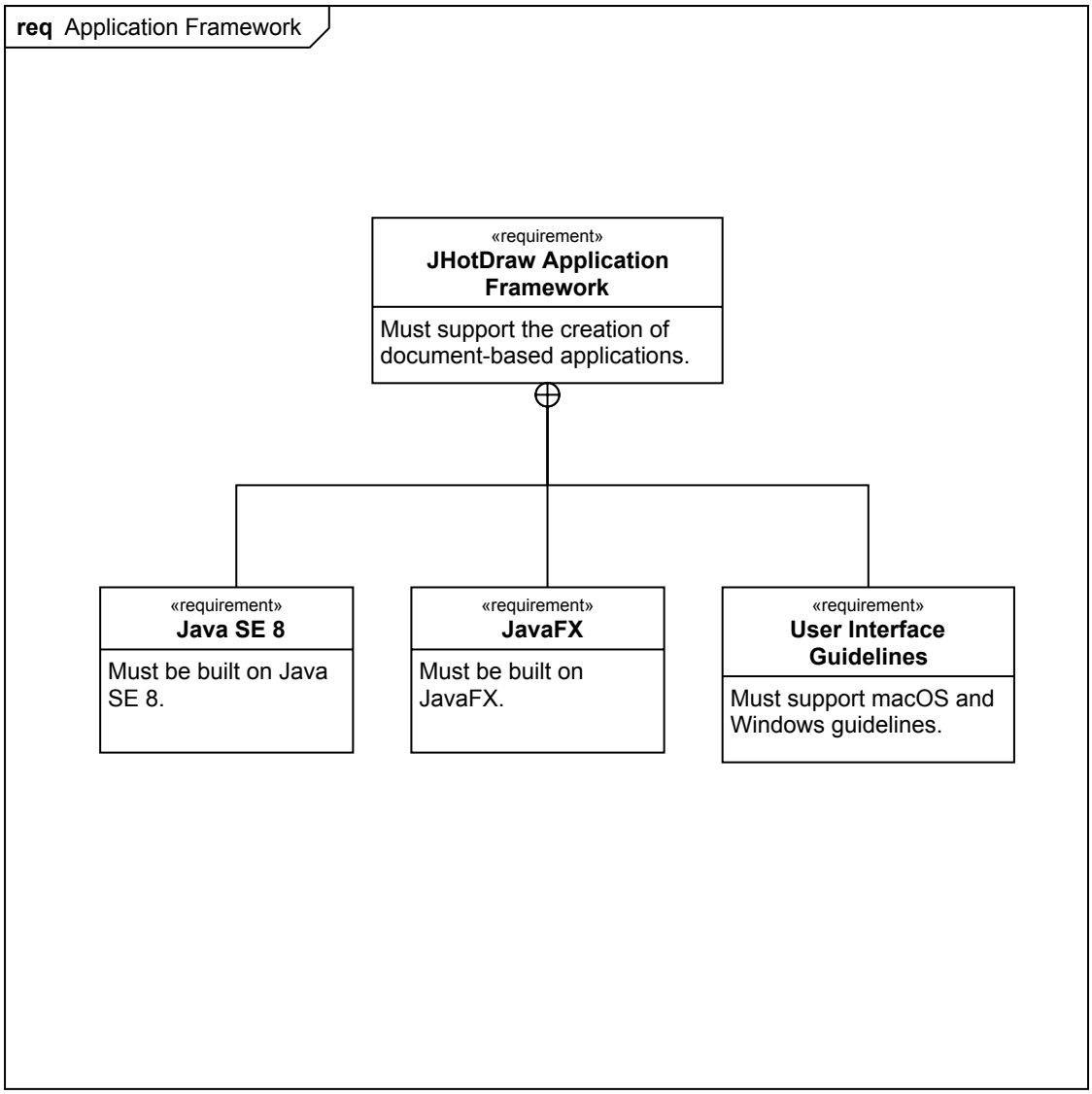

Figure 3.1. SysML Requirements Diagram: Application Framework

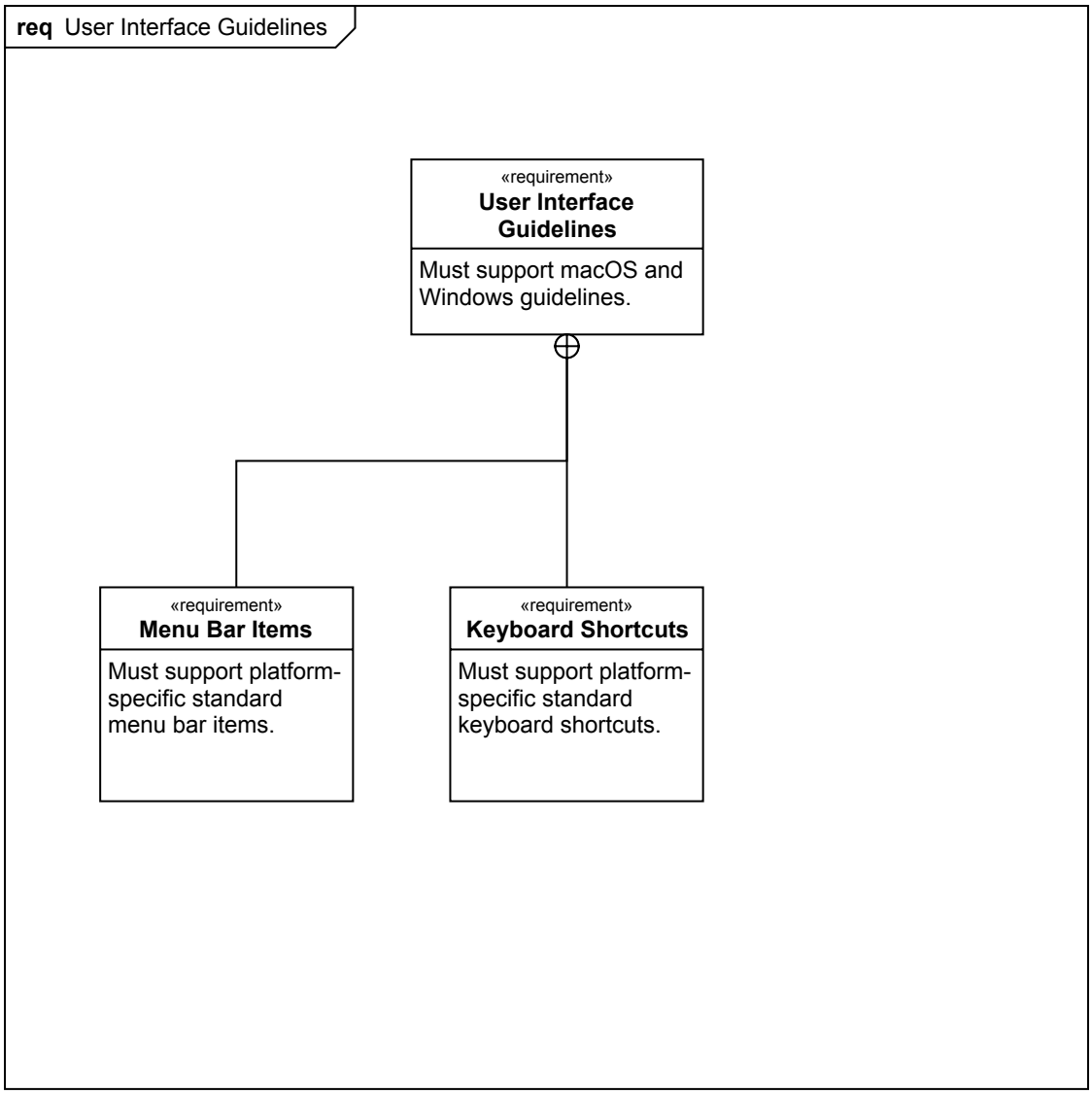

Figure 3.2. SysML Requirements Diagram: User Interface Guidelines

Application Framework

The application framework

- must define a design for document-based applications,
- must provide an API of the design,
- must provide a reference implementation of the API.

User Interface Guidelines

The application framework must support the following user interface guidelines.

- Apple macOS Human Interface Guidelines [MacHIGuidelines].
- Windows User Experience [WinUXGuidelines].

Java SE 8

The application framework must be built on Java SE 8.

Java FX

The application framework must be built on Java FX.

Menu Bar Items

The application framework must support platform-specific standard menu bar items.

macOS-specific Menu Bar Items

The application framework must support the following menu bar items on macOS.

- Application
    - About "AppName"
    - Preferences... **Command+,**
    - Services
    - Hide "AppName" **Option+Command+H**
    - Hide Others **Command+H**
    - Show All
    - Quit "AppName" **Command+Q**

- File
    - New **Command+N**
    - Open... **Command+O**
    - Open Recent
    - Close **Command+W**
    - Close File **Shift+Command+W**
    - Save **Command+S**
    - Duplicate
    - Save As... **Shift+Command+S**
    - Export As...
    - Save All
    - Revert to Saved
    - Print... **Command+P**
    - Page Setup... **Shift+Command+P**
- Edit
    - Undo **Command+Z**
    - Redo **Shift+Command+Z**
    - Cut **Command+X**
    - Copy **Command+C**
    - Paste **Command+V**
    - Paste and Match Style **Option+Shift+Command+V**
    - Delete

    - Select All **Command+A**
    - Find **Command+F**
    - Find Next **Command+G**
    - Emoji & Symbols **Control+Command+Space**
- Format
    - Show Fonts **Command+T**
    - Show Colors **Shift+Command+C**
    - Bold **Command+B**
    - Italic **Command+I**
    - Underline **Command+U**
    - Bigger **Shift+Command+=**
    - Smaller **Shift+Command+-**
    - Copy Style **Option+Command+C**
    - Paste Style **Option+Command+V**
    - Align Left **Command+{**
    - Center **Command+|**
    - Justify
    - Align Right **Command+}**
    - Show Ruler
    - Copy Ruler **Control+Commmand+C**
    - Paste Ruler **Control+Command+V**
- View

    - Show/Hide Toolbar **Option+Command+T**
    - Customize Toolbar
    - Enter Full Screen **Control+Command+F**
- Window
    - Minimize **Command+M**
    - Minimize All **Option+Command+M**
    - Zoom
    - Bring All to Front
    - Arrange in Front
- Help
    - "AppName" **Help**

Windows-specific Menu Bar Items

The application framework must support the following menu bar items on Windows.

- File
    - New **Ctrl+N**
    - Open... **Ctrl+O**
    - Open Recent
    - Close **Ctrl+W**
    - Save **Ctrl+S**
    - Save As... **Shift+Ctrl+S**
    - Export As...

    - Revert to Saved
    - Print... **Ctrl**+**P**
    - Page Setup... **Shift**+**Ctrl**+**P**
    - Exit **Alt**+**F4**
- Edit
    - Undo **Ctrl**+**Z**
    - Redo **Ctrl**+**Y**
    - Cut **Ctrl**+**X**
    - Copy **Ctrl**+**C**
    - Paste **Ctrl**+**V**
    - Paste and Match Style **Alt**+**Shift**+**Ctrl**+**V**
    - Select All **Ctrl**+**A**
    - Delete
    - Find **Ctrl**+**F**
    - Find Next **F3**
    - Replace... **Ctrl**+**H**
    - Go to... **Ctrl**+**G**
- View
    - Show/Hide Toolbar
    - Show/Hide Status bar
    - Zoom in **Ctrl**+**+**
    - Zoom out **Ctrl**+**-**

    - Full Screen **F11**
    - Refresh **F5**
    - Tools...
    - Options...
- Help
    - Help **F1**
    - About

Keyboard shortcuts

The application framework must support platform-specific keyboard shortcuts.

Drawing Editor Framework

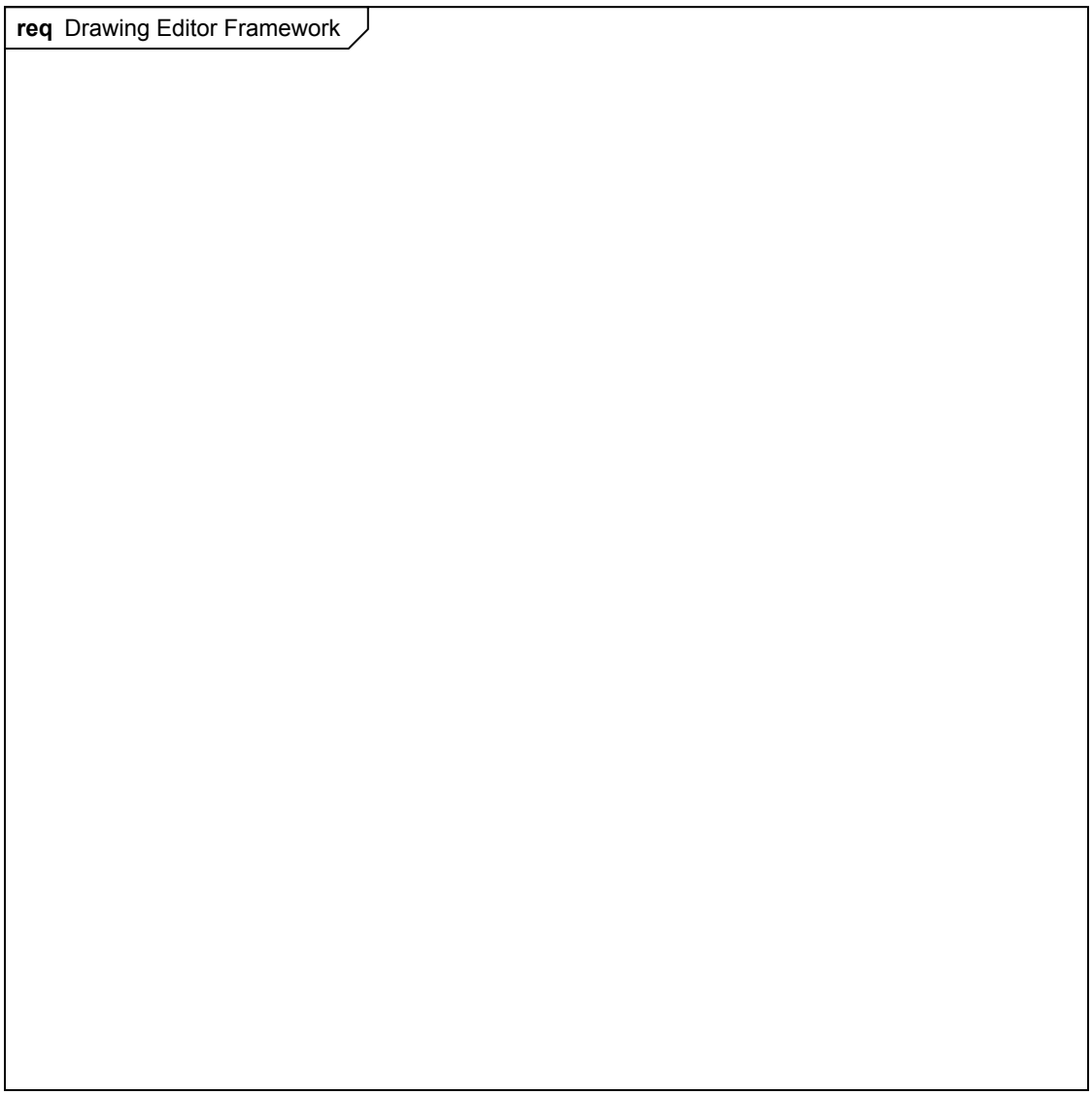

Figure 3.3. SysML Requirements Diagram: Drawing Editor Framework

Drawing Editor Framework

The drawing editor framework

- must define a design for drawing editors,
- must provide an API of the design,
- must provide a reference implementation of the API.

Automatic Layout

The drawing framework must support figures that can be laid out automatically.

For some kinds of drawings, e.g. diagrams, the user is not required to layout all figures manually.

Figures that can be laid out automatically include:

- connection lines between diagram elements
- labels inside a diagram element
- labels attached to a diagram element

Cascading Style Sheets

A drawing must be styleable with cascading style sheets.

Cascading Style Sheets Style Origin

The value of a styled property must origin from one of the 4 style origins defined in JavaFX class StyleOrigin:

INLINE

: The value is set from a style in the „style“ property of a figure.

AUTHOR

: The value is set from an external stylesheet.

USER

: The value is set on a property of a figure.

USER_AGENT

: The value is set from an internal stylesheet.

If a value is defined in more than one style origin, then values must be taken with the following precedence rule:

- INLINE
- AUTHOR
- USER
- USER_AGENT

The framework must support the following user editing functions:

- Add and remove AUTHOR stylesheets.
- Edit USER property values
- Edit the value of the INLINE style property

The developer of a drawing editor must be able to:

- define the USER_AGENT stylesheets.

Layers

A drawing must be organizable into layers.

Locking

The drawing framework must support locking for figures. A locked Figure must not be selectable in a drawing view.

Visibility

The drawing framework must support visibility of figures. An invisible figures must not be drawn.

Z-Order

The drawing framework must support a z-ordering of figures.

Figures with higher z-order must be drawn above figures with lower z-order.

Grouping

The drawing framework must support grouping for figures. Grouped figures must act like they were a single figure.

Constructive Area Geometry

The paths of multiple figures must be combinable into a new figures using constructive area geometry. The combined figures must behave like a single figure.

Chapter 4. Design

Table of Contents

Principles

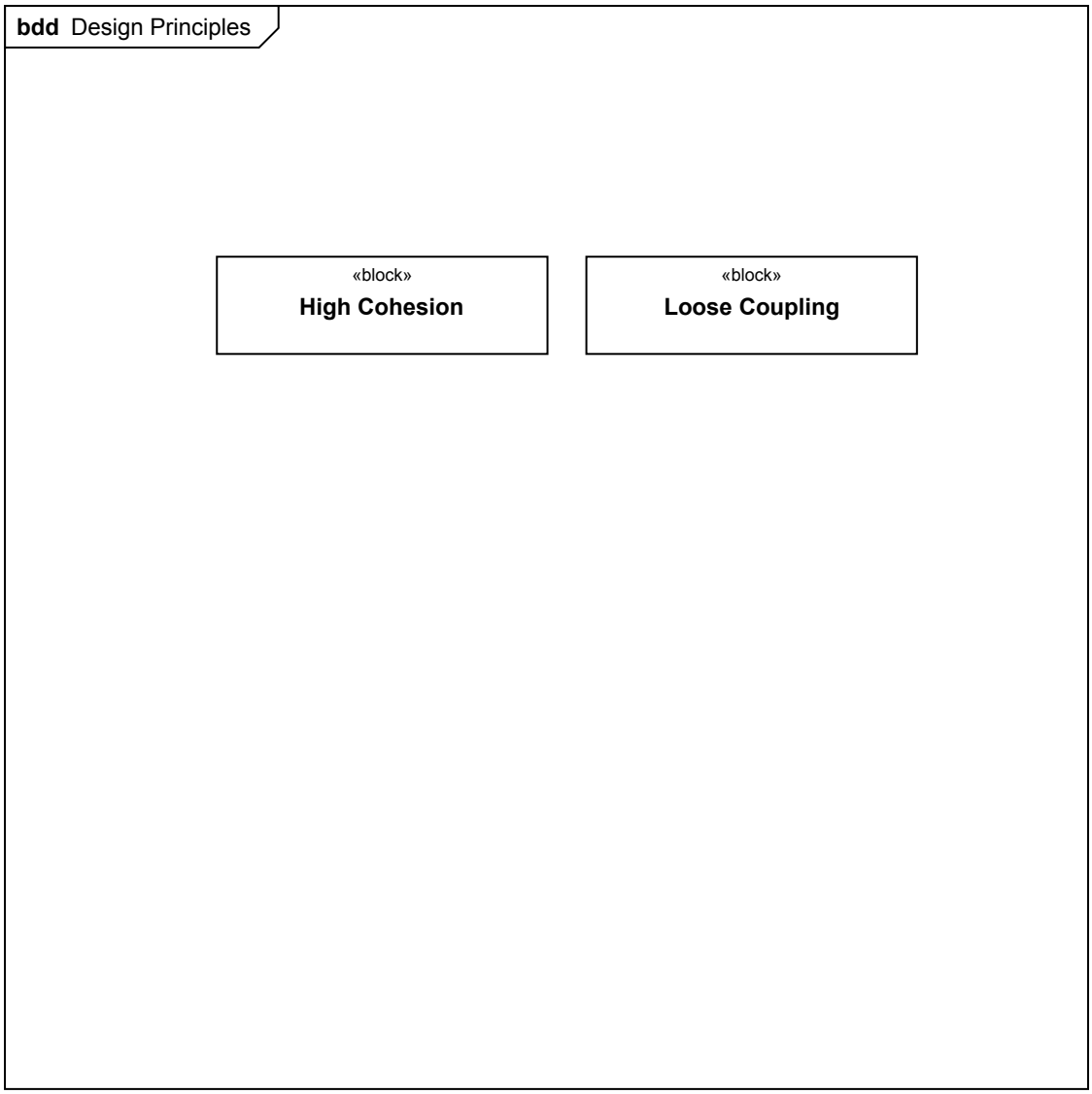

Figure 4.1. SysML Block Definition Diagram: Design Principles

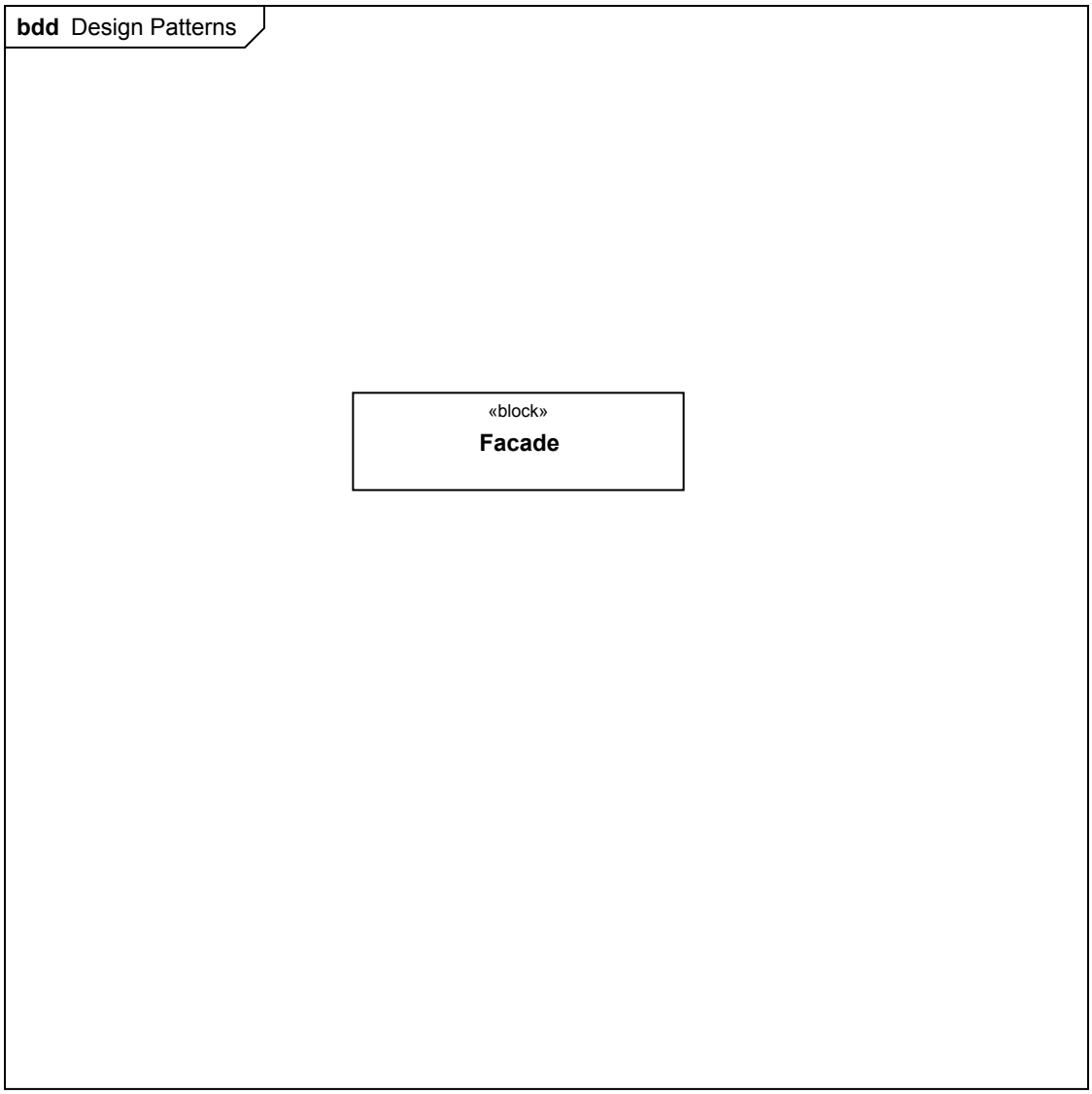

Figure 4.2. SysML Block Definition Diagram: Design Patterns

High Cohesion

Cohesion is the degree to which a class has a single, focused purpose.

We can achieve high cohesion by adhering to the following rules:

- If not all clients of an interface use all methods of an interface, we should split the interface up.

Trade-off

A class with high cohesion is typically less complex than a class with low cohesion.

Loose Coupling

Coupling is the degree to which one class knows about another class.

We can achieve loose coupling by adhering to the following rules:

- A class should hide its internal fields from other classes.

  This can be done by declaring fields private, package protected or protected.

- A class should hide its internal structure from other classes.

  This can be done by hiding the internal structure with a facade.

- A class should hide its internal types from other classes.

  This can be done by declaring internal types private, package protected or protected.

- A class should reference a type only if it does use methods or fields specified by that type.

  This can be done by only importing types that are referenced, and by declaring type parameters instead of concrete types.

- A class should reference concrete classes only if they are part of its internal structure, and are hidden from other classes.

- A class should reference only types from its domain and from the Java API.

- A class should not require that types from another domain must reference other types from its domain.

Trade-off

A set of loosely coupled classes is easier to maintain than a set of tightly coupled classes, because changes typically affect less classes.

Facade Pattern

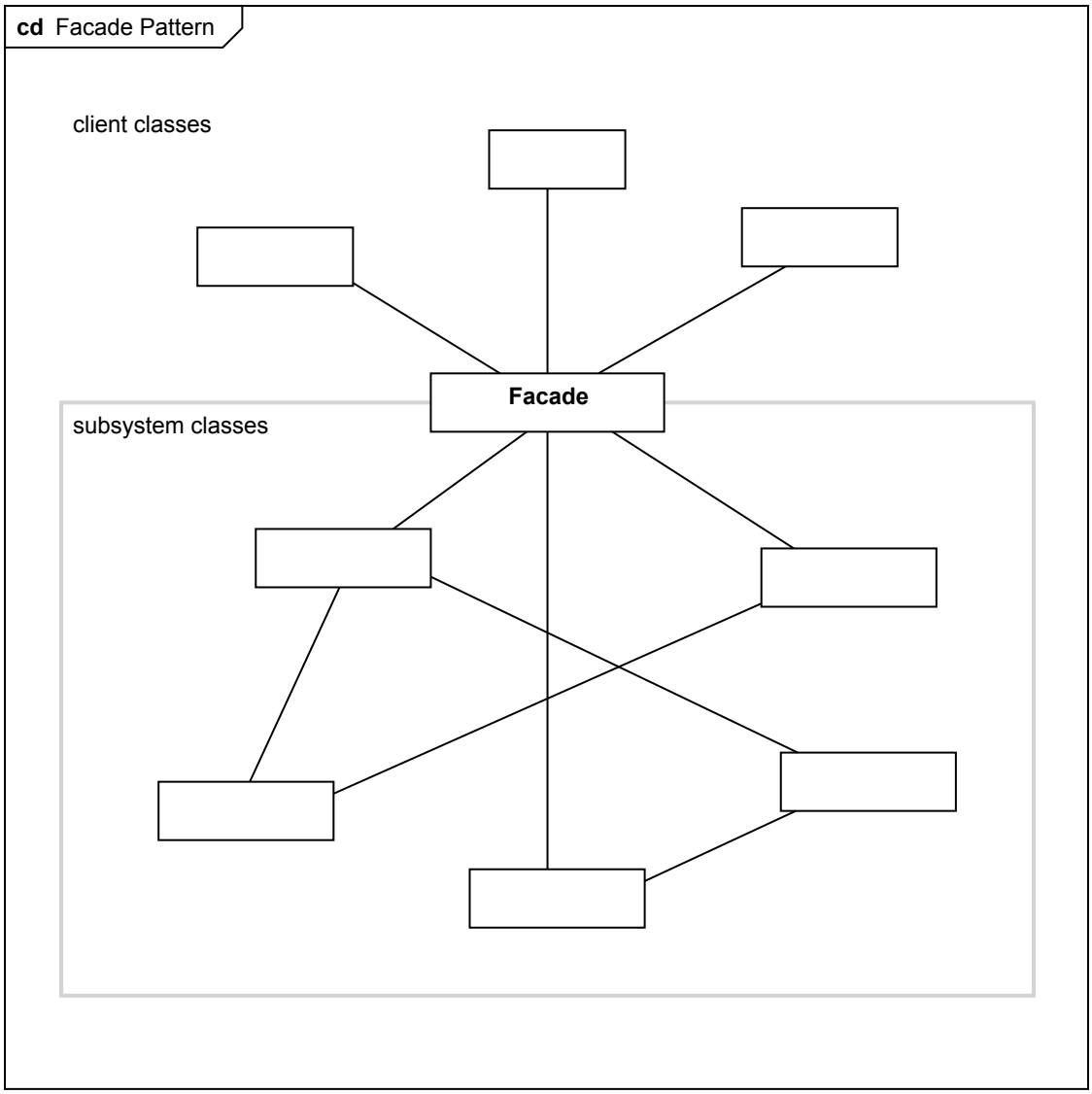

Figure 4.3. UML Class Diagram: Facade Pattern

A facade provides a unified interface to a set of interfaces in a subsystem.

Trade-off

A facade allows to conveniently pass a subsystem around as a single object.

A facade allows to hide the structure of the subsystem.

A facade allows to make the subsystem easier to use.

A facade adds a level of indirection that may increase the program complexity.

A facade adds a level of indirection that may impose a performance hit.

Application Framework

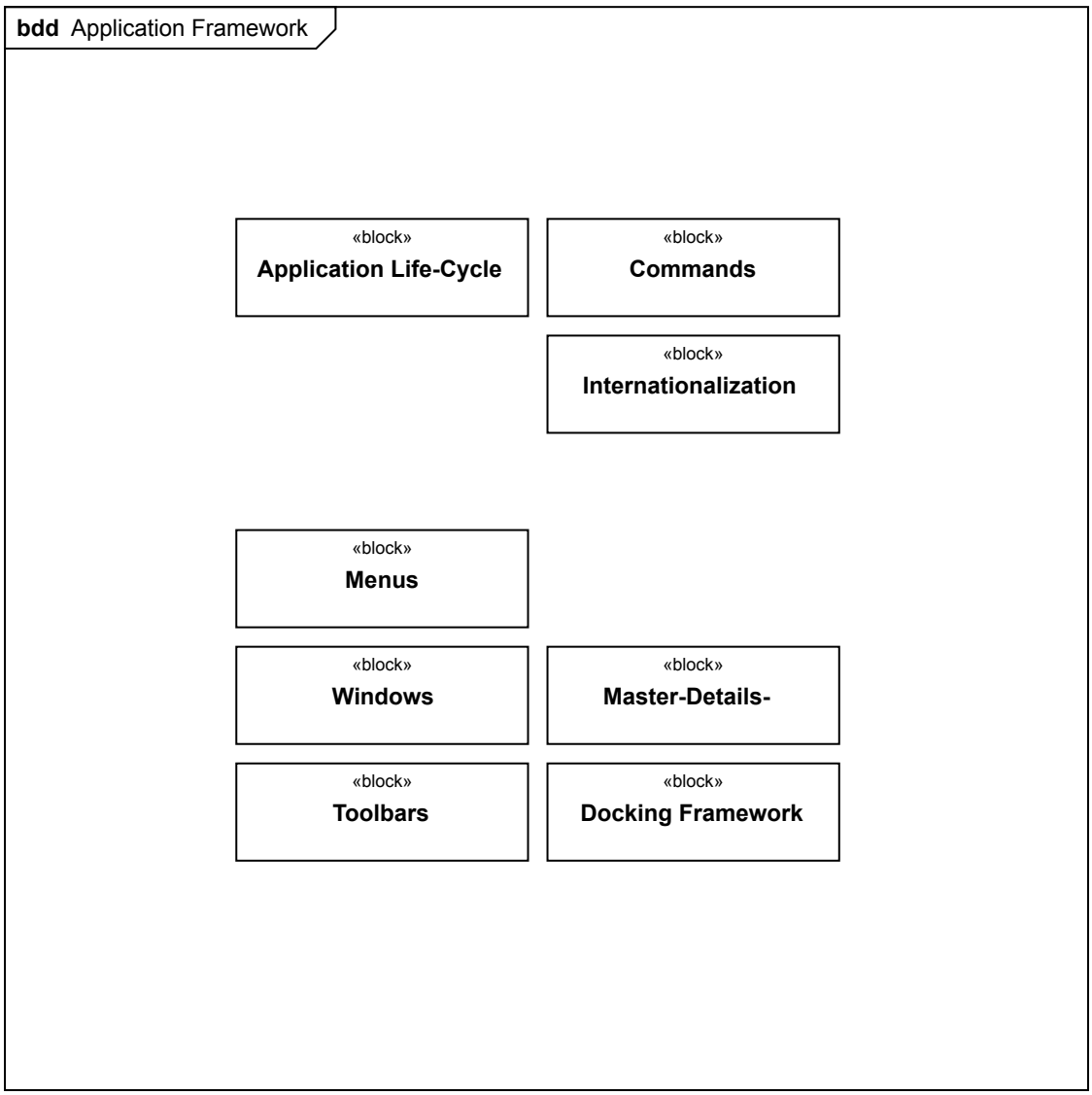

Figure 4.4. SysML Block Definition Diagram: Application Framework

Master-Details-Inspector Layout

The master-details-inspector layout is a layout that consists of the following three panes:

- The master pane in the left area of the window

  A master pane is a control that displays a collection of objects from which one can be selected.

- The details pane in the center area of the window

  A details pane is a control that displays the selected object from the master pane. The object can be structured from which one or multiple elements can be selected.

- The inspector pane in the right area of the window

  An inspector pane is a control that displays properties of the current selection in the details pane.

The master-details-inspector layout is used by many applications, and thus is well known and understood by most users.

We can implement this layout JavaFX by following these steps:

- For each component (master pane, detail pane, inspector pane) we create a FXML file for the layout, and a JavaFX controller class: MasterPaneController, DetailsPaneController, InspectorPaneController.

- We add a List control (or Table or Tree control) to the MasterPaneController.

- We expose the currently selected item in the master pane with an ObjectProperty in the MasterPaneController.

- We expose the currently shown object in the details pane with an ObjectProperty in the DetailsPaneController.

- We expose the current selection in the details pane with an ObservableList in the DetailsPaneController.

- We expose the currently inspected selection in the inspector pane with an ObservableListProperty in the InspectorPaneController.

- We bind the exposed properties together.

We can alter this layout in various ways:

- We can omit the master pane, which leaves us with a details-inspector layout.

- We can omit the inspector pane, which leaves us with a master-details layout.
- We can enable multiple selection in the master pane, which turns the object that is displayed in the details pane into a collection.
- We can show properties of other objects in the inspector pane.
- The window can contain more than one inspector pane.

Drawing Editor Framework

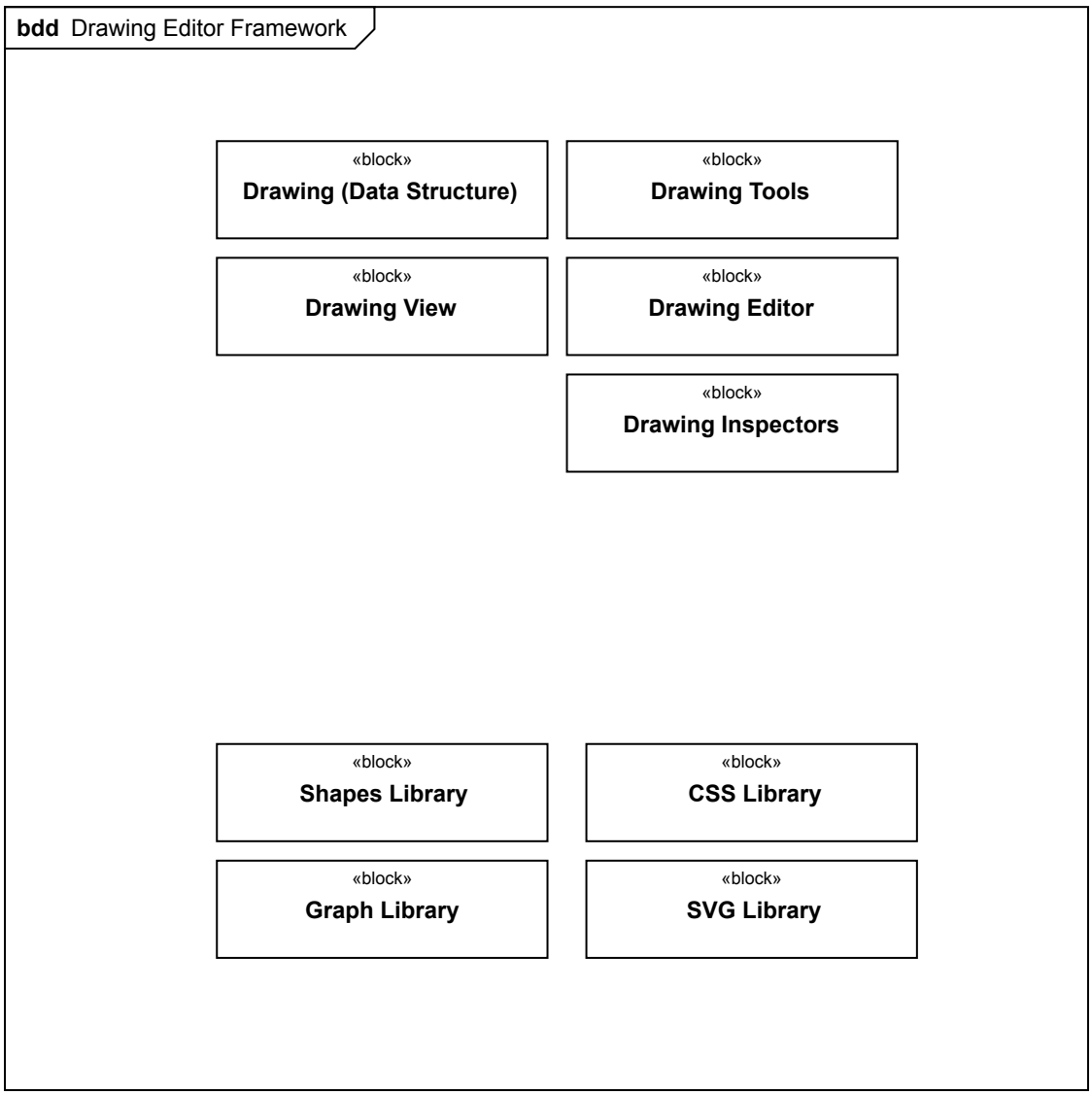

Figure 4.5. SysML Block Definition Diagram: Drawing Editor Framework

Layers

We can support layers in a drawing by treating layers as a special kind of figures.

Since we want to represent a drawing as a tree structure, we can require the following:

- A `Layer` can only have of a parent of type `Drawing`.

- Any other type of `Figure` can only have of a parent that is not of type `Drawing`.

Trade-off

The trade-off with this design is that we always must add at least one layer to a drawing, even if we don't need layers in that specific drawing.

Drawing Data Structure

The data structure of a drawing can be represented as a tree using the "Composite" design pattern.

TreeNode<Figure>

> TreeNode is the interface type that realizes the "Composite" design pattern.

Figure

> Figure is the interface type for all elements in a drawing, including the drawing itself.
>
> Figure extends the TreeNode<Figure> interface.

Trade-off

This design implements the "Composite" design pattern in a two interfaces. This allows to separate the concern of representing a drawing element (by Figure) from the concern of representing a tree structure (by TreeNode).

Layout Dependencies

The layout of a figure can depend on the layout of other figures.

In order to layout a drawing with such figures, we must first layout the figures that do not depend on other figures. Then we can continue to layout figures that depend on figures that we have already laid out until no more figures are left.

The layout dependencies can be thought of as a directed graph. By sorting this directed graph topologically, we can get the sequence in which we must layout the figures.

Type-Safe Property Map

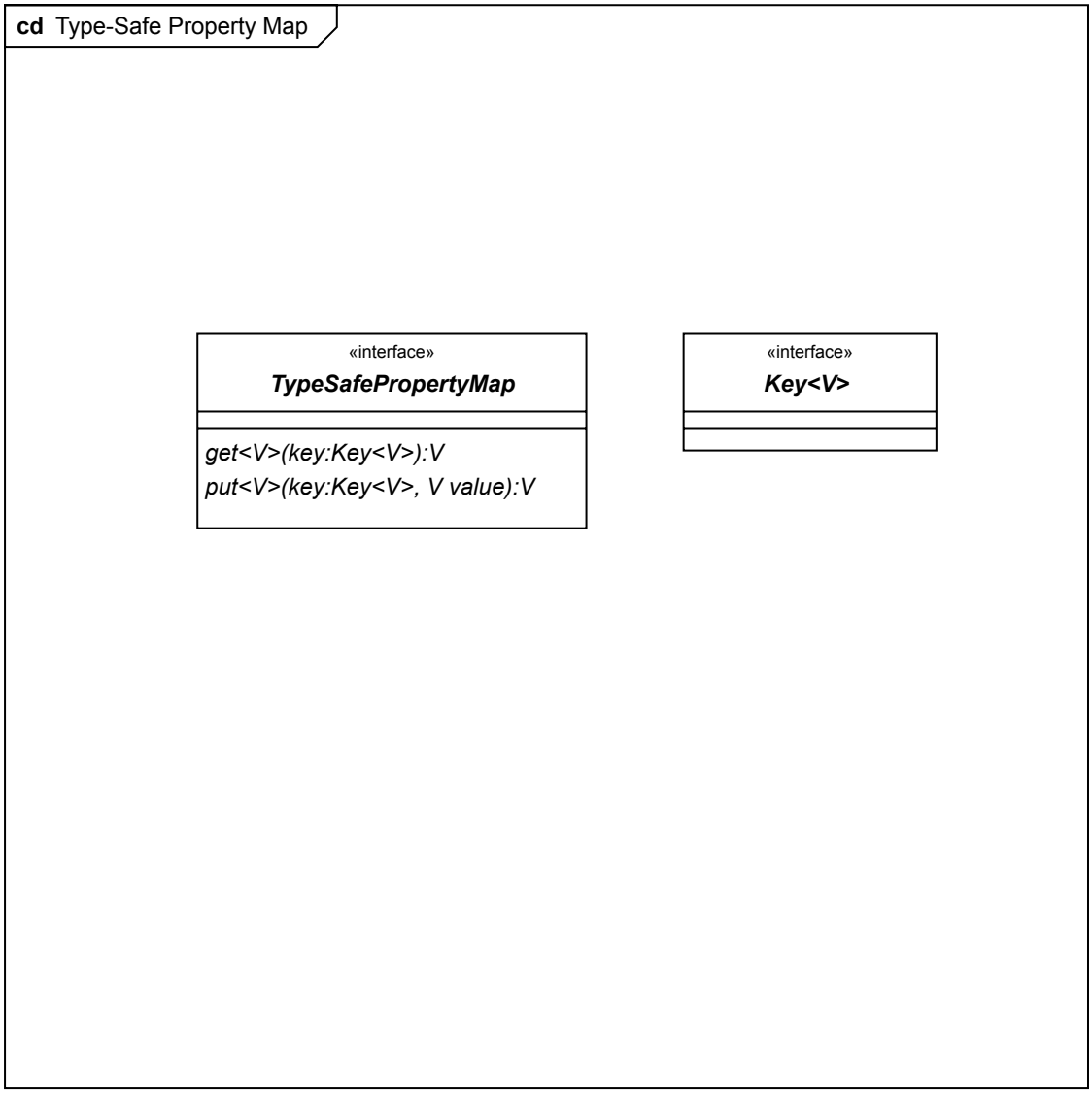

Figure 4.6. UML Class Diagram: Type-Safe Property Map

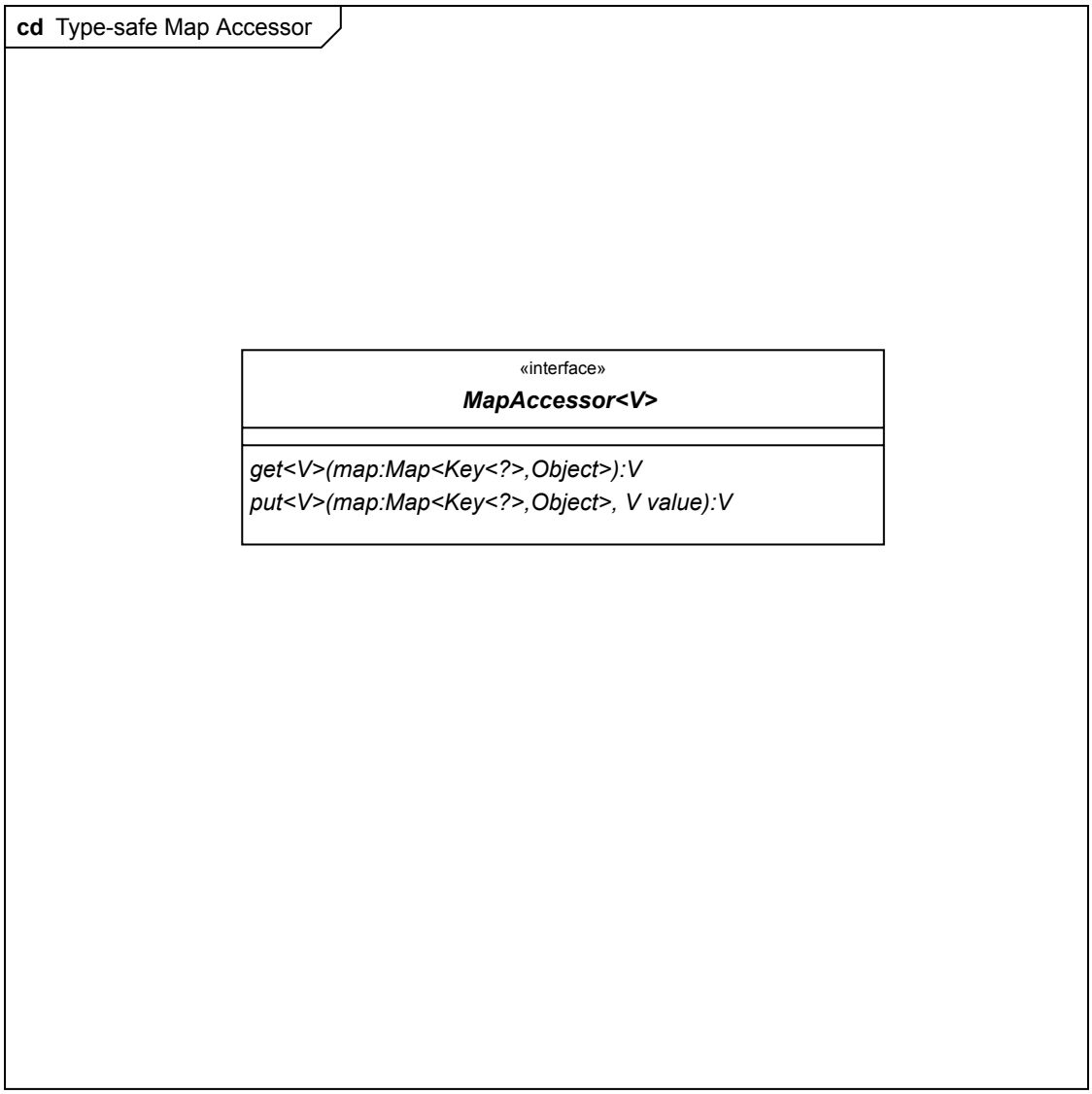

Figure 4.7. UML Class Diagram: Type-Safe Map Accessor

The figures in a drawing can support a large number of properties. We do not want to implement an accessor method for each property but we still want to be able to access a property value in a type-safe way.

A type-safe property map provides a type-safe mapping from property keys to property values. Where the type of a property value is specified by the type parameter `V` of the property key `Key<V>`.

The type parameter `<V>` can be used to create type-safe map accessors, such as the following:

- `<V> V get(Key<V> key).`
- `<V> V put(Key<V> key, V value).`

- `<V> void set(Key<V> key, V value).`

Style Attributes Inspector

The “Style Attributes Inspector” is a control that allows to edit style attributes of the currently selected figures in a drawing view.

The inspector has the following components:

- a text area
- a vertical toolbar
- a horizontal toolbar

The text area shows the style attributes of the selected figures in the current drawing view. The style attributes are formatted as a CSS rule-set, consisting of a selector and a block of declarations.

The text area is editable.

The vertical toolbar is displayed along the left border of the text area. It contains a push button next to each declaration in the text area. The push buttons can be used to open a pop-up dialog for editing a declared value.

The horizontal toolbar contains a settings menu, and an apply button and a select button.

The settings menu can be used to select different display options for the text area.

The apply button can be used to apply the CSS rule-set to the drawing.

The select button can be used to evaluate the CSS

Graph Library

Graph Library

This design specifies a graph library that is tailored for the drawing editor framework.

Graph in Graph Theory

In graph theory a graph is an ordered pair $G = (V, E)$.

Where

- V is a set of vertices $V = (v_1, v_2, ...)$.
- E is a set or bag of edges $E = (e_1, e_2, ...)$.
- An edge $e \in E$ is an ordered or unordered pair of vertices $e = (v_i, v_j), v_i \in V, v_j \in V$.

Directed Graph in Graph Theory

In graph theory a directed graph is a specialization of a graph, in that it only supports edges that are ordered pairs of vertices. These edges are named 'arrows'.

A directed graph is an ordered pair $G = (V, A)$.

Where

- V is a set of vertices $V = (v_1, v_2, ...)$.
- E is a set or bag of arrows $A = (a_1, a_2, ...)$.
- An arrow $a \in A$ is an ordered pair of vertices $a = (v_i, v_j), v_i \in V, v_j \in V$.

Graph Facade APIs

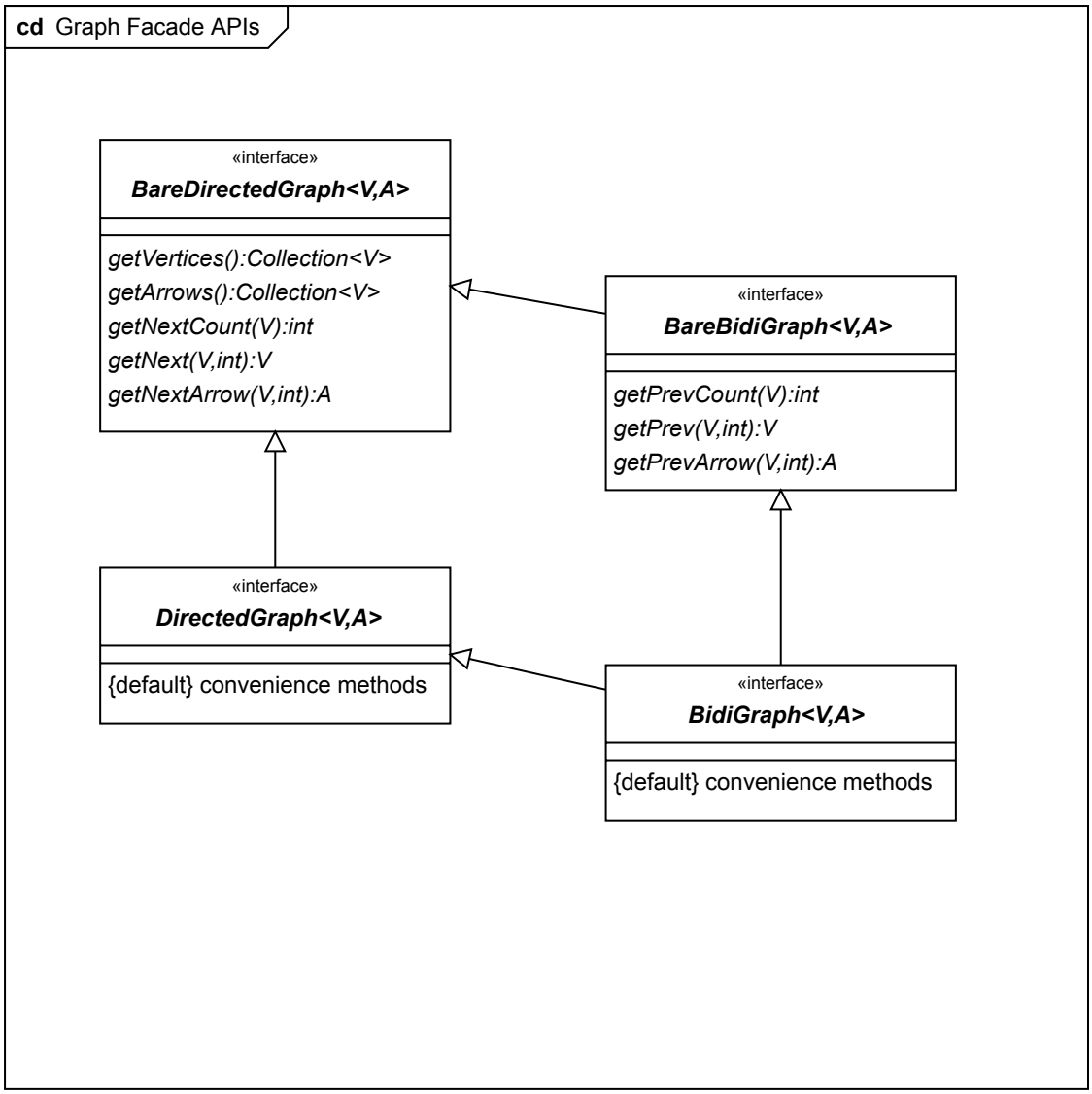

Figure 4.8. UML Class Diagram: Graph Facade APIs

We introduce a number of facade APIs for graphs.

We will use these facade APIs for passing a graph around as a single object, and as a means for hiding its implementation from clients.

The facade APIs are defined in the following interfaces.:

BareDirectedGraph

> Provides a minimal read-only API for directed graphs. The API includes accessor-methods to the collections of vertices and arrows, and methods for following arrows in forward direction.
>
> A vertex V in the graph must have a unique identity or value.

An arrow A in the graph is not required to have a unique identity or value.

The API does not make any assumptions about the type or structure of an arrow A. Users of the API are free to ignore arrow objects (and just specify Void for A), or to use A for storing edge weights or other data associated to arrows.

DirectedGraph

Extends BareDirectedGraph with a set of convenience methods.

BareBidiGraph

Extends BareDirectedGraph with a set of methods for following arrows in backward direction.

BidiGraph

Extends BareBidGraph and DirectedGraph with a set of convenience methods.

To keep the API definitions small, we deliberately only define interfaces with read-only APIs. We implement APIs for writing directly in classes that support writing.

Trade-off

- All trade-offs described in the section called “Facade Pattern” apply.

Table of Contents