|  |
| --- |
| Sandcastle |
| Software Architecture Documentation 1.1 |
| Architecture Studio |

**Revision History**

Contents

[Table of Figures 6](#_Toc229312537)

[Table of Responsibility Catalogs 6](#_Toc229312538)

[1 Introduction 7](#_Toc229312539)

[1.1 About this document 7](#_Toc229312540)

[1.2 Definitions and Acronyms 7](#_Toc229312541)

[1.3 Templates 7](#_Toc229312542)

[1.3.1 Decomposition Rationale 7](#_Toc229312543)

[1.3.2 Element and Relationship Responsibility Catalog 7](#_Toc229312544)

[1.3.3 Requirements Traceability Matrix 8](#_Toc229312545)

[1.4 References 8](#_Toc229312546)

[2 Architectural Documentation 8](#_Toc229312547)

[2.1 UML 2.0 Discussion 8](#_Toc229312548)

[2.1.1 Limitations of UML 8](#_Toc229312549)

[2.1.2 Ports 9](#_Toc229312550)

[2.2 Development Framework Discussion 9](#_Toc229312551)

[2.3 Context Diagram 10](#_Toc229312552)

[2.4 Component-and-Connector Decomposition Level 1 12](#_Toc229312553)

[2.4.1 Decomposition Rationale 12](#_Toc229312554)

[2.4.2 Element Responsibilities Catalog 13](#_Toc229312555)

[2.4.3 Relationship Responsibilities Catalog 14](#_Toc229312556)

[2.5 Component-and-Connector Decomposition Level 2 15](#_Toc229312557)

[2.5.1 Decomposition Rationale 15](#_Toc229312558)

[2.5.2 Element Responsibilities Catalog 16](#_Toc229312559)

[2.5.3 Relationship Responsibilities Catalog 19](#_Toc229312560)

[2.6 Module Decomposition Level 1 22](#_Toc229312561)

[2.6.1 Decomposition Rationale 22](#_Toc229312562)

[2.6.2 Element Responsibilities Catalog 23](#_Toc229312563)

[2.6.3 Relationship Responsibilities Catalog 23](#_Toc229312564)

[2.7 Module Decomposition Level 2 24](#_Toc229312565)

[2.7.1 Decomposition Rationale 25](#_Toc229312566)

[2.7.2 Element Responsibilities Catalog 26](#_Toc229312567)

[2.7.3 Relationship Responsibilities Catalog 26](#_Toc229312568)

[2.8 Allocation Decomposition Level 1 – Deployment Style 28](#_Toc229312569)

[2.8.1 Decomposition Rationale 28](#_Toc229312570)

[3 Mappings between Perspectives 29](#_Toc229312571)

[4 Domain Model 29](#_Toc229312572)

[5 File Formats 31](#_Toc229312573)

[6 Requirements Traceability Matrix 31](#_Toc229312574)

# Table of Figures

[Figure 1 Context Diagram 11](#_Toc229312575)

[Figure 2 Component-and-Connector Context - Initial context 12](#_Toc229312576)

[Figure 3 Decomposition of Architecture Studio 15](#_Toc229312577)

[Figure 4 Module Context - Initial context 22](#_Toc229312578)

[Figure 5 System - Decomposition of the Architecture Studio packages 25](#_Toc229312579)

[Figure 6 Mapping of artifacts to directories 28](#_Toc229312580)

[Figure 7 Domain Model 30](#_Toc229312581)

# Table of Responsibility Catalogs

[Catalog 1 Elements - Component-and-Connector Level 1 13](#_Toc229312582)

[Catalog 2 Relationships – Component-and-Connector Level 1 14](#_Toc229312583)

[Catalog 3 Elements - Level 2 16](#_Toc229312584)

[Catalog 4 Relationships - Level 2 19](#_Toc229312585)

[Catalog 5 Elements - Module Level 1 23](#_Toc229312586)

[Catalog 6 Relationships - Module Level 1 23](#_Toc229312587)

[Catalog 7 Elements - Module Level 2 26](#_Toc229312588)

[Catalog 8 Relationships - Module Level 2 26](#_Toc229312589)

# 1 Introduction

## About this document

This document contains the software architecture for the Architecture Studio project. It is advised that the reader familiarize themselves with the Architectural Drivers Specification document first before examining the architecture detailed in this document.

## Definitions and Acronyms

All terms and acronyms used in this document have been defined in the Sandcastle Glossary document. Please refer to that document for further information. If you didn’t get the glossary with this document, please contact a Sandcastle team member.

### Element and Relationship Responsibility Catalog

#### Element Catalog

List the drawing containing the elements in Associated Drawings. Use short sentences and bullets to document the responsibilities. See [Lat08] for examples on how to phrase responsibilities.

|  |  |
| --- | --- |
| **Associated Drawings:** [*List related figure(s)*] | **Perspective:** [*Name the perspective of the related figure*] |
| **Element** | **Responsibilities** |
| [*Element A*] | [*Responsibilities of Element A*]  *If the element has ports, use the following table:*   |  |  | | --- | --- | | Port | Description | | [*Name of Port A*] | [*Responsibilities of Port A*] | |

#### Relationship Catalog

See 1.2.2.1 (previous section).

|  |  |
| --- | --- |
| **Associated Drawings:** [*List related figure(s)*] | **Perspective:** [*Name the perspective of the related figure*] |
| **Relationship** | **Responsibilities** |
| Element A[PortX] *interacts with* Element B[PortY] | [*Responsibilities of this relationship*] |

### Requirements Traceability Matrix

See [Lat08] chapter 10 for more details on using this template.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requirement satisfied** | **Design decision** | **Element** | **Relationship** | **Design artifact reference** | **Comments and descriptions** |
|  |  |  |  |  |  |

# Architectural Documentation

## Context Diagram



Figure Context Diagram

## Portability Level 1

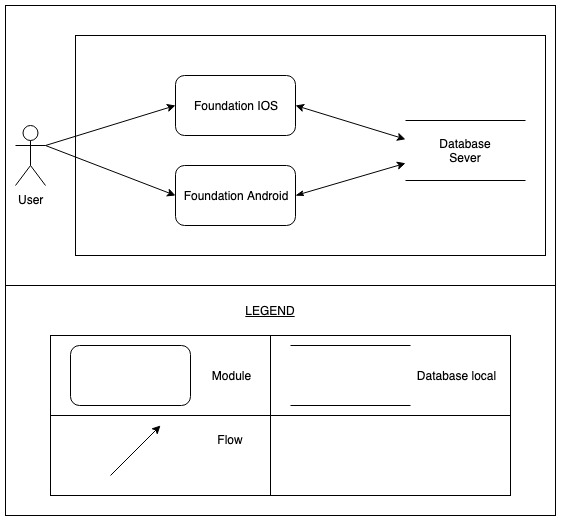


Figure Portability Level 1

### Decomposition Rationale

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 2  **Associated Responsibilities:** Catalog 1, Catalog 2 | **Perspective:** Dynamic, physical |
| This diagram shows one user can use on two foundation, 2 Foundation uses the same database | |

### Element Responsibilities Catalog

**Catalog 1 Elements - Dynamic**

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 2 | **Perspective:** Dynamic, physical |
| **Element** | **Responsibilities** |
| Database Sever | Where data is stored |
| Foundation IOS | It is foundation |
| Foundation Android | It is foundation |

### Relationship Responsibilities Catalog

**Catalog 2 Relationships**

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 2 | **Perspective:** Dynamic, physical |
| **Relationship** | **Responsibilities** |
| Flow | This relationship will be used to connect the elements together. This symbol indicates an asset A will transfer the data to the B element. Element B will receive information data |
| Data flow | This relationship will be used to connect the elements together. This symbol indicates an asset A will transfer the data to the B element. Element B will receive information about the data and returns |

## Component-and-Connector Decomposition Level 1

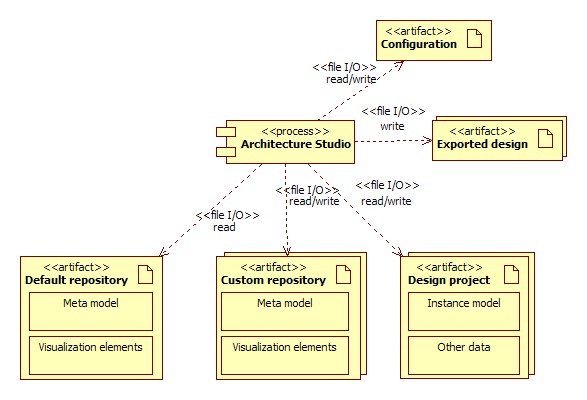


Figure Component-and-Connector Context - Initial context

### Decomposition Rationale

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 2  **Associated Responsibilities:** Catalog 1, Catalog 2 | **Perspective:** Component-and-Connector |
| This is the top level decomposition of the system, and shows interactions of the system with various files and directories that are used and/or created by the system. The *Design project* directory is created by the system to store all information related to the current project the user is working on. This concept is similar to the one used by Eclipse, in which a “.project” file contains properties of a project. If the user creates diagrams for their project, they are saved within the project directory structure.  The *Default repository* directory contains the diagram element types which the user can use when creating his diagrams. Each element type includes the visualization for the element type (the actual shape that is drawn when the user creates an instance of the element type), and the properties of the element type. The *Configuration* file contains application level settings. The *Exported design* file is created when the user chooses to export their current project in a particular format.  Most of the structural decisions conveyed in this decomposition are dictated by decisions defined in the architectural drivers. The *Default* repository is separate from *Architecture Studio* in order to easily add and modify the default element types at development time without affecting the rest of the application. The *Custom repository* stores the custom element types created by the user. The format of the *Custom repository* is the same the *Default repository.* There are two separate repositories for the custom element types and default element types so that the custom element types can be stored with the *Design Project* that references those element types. This makes it easier to for the user to give their design with the associated custom element types to other users or move it to another computer.  The architectural drivers also specify that the application will provide the ability to export a project to an external format as a report or overview document. The *Configuration* file is present to allow the user to change his application preferences and thus enhancing modifiability of the application at run-time. | |

### Element Responsibilities Catalog

Catalog Elements - Component-and-Connector Level 1

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 2 | **Perspective:** Component-and-Connector |
| **Element** | **Responsibilities** |
| Architecture Studio | * This element is a process * Be able to interact with the user * Be able to open, manipulate and save to files in the Design project directory. * Be able to read, process, and change the configuration file. * Be able to read and use default meta model elements and visualization elements. * Be able to read/write and use custom meta model elements and visualization elements. * Be able to export currently loaded design to an external format. It will not be able to open or manipulate this external file format. |
| Design project | This is a directory that will be created for each project the user creates and saves using the tool. The choice of using a directory to hold project information, as opposed to a single file, has been taken because of the different storage formats of different parts of the project.  This directory contains all information specific to the project, which is stored as an instance model which includes:   1. Architectural drivers instance model elements 2. Design drawings instance model elements containing:    1. Coordinates & dimensions of visualization elements    2. Properties of the instance model (such as name, rationale, etc.)    3. Information about groups of visualization elements 3. Information on mappings and assignments 4. Information on decompositions 5. Information on to-do’s   Other data stored in this directory is related to preferences or other entities that are not contained in the meta model:   1. (Possibly) Information on the state of project windows, e.g. which views were open when the project was last saved. |
| Configuration | This file contains settings of the application that can be changed after compilation.  Some settings that are considered for storing in this file are:   1. Location and status (collapsed or visible) of the various Eclipse views in the main window. This configuration is at the application level and is applied to the application regardless of which project is open. |
| Default repository | This directory structure holds the default meta model elements and their associated visualization elements. This information will be used to provide the user with a palette of shapes to draw their designs with. |
| Custom repository | This directory structure holds custom meta model elements and their associated visualization elements. |
| Exported design | This file will contain partial information that is contained in the Design project in a common file format. The system cannot import or read such a file. |

### Relationship Responsibilities Catalog

Catalog Relationships – Component-and-Connector Level 1

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 2 | **Perspective:** Component-and-Connector |
| **Relationship** | **Responsibilities** |
| Architecture Studio *reads/writes to* Design project | This relationship denotes read and write operations to the Design project directory. The system will ensure that files in this directory are saved in the appropriate format. The system will read information from this directory and present information to the user. |
| Architecture Studio *reads/writes to* Configuration | This relationship denotes read and write operations to the configuration file. The system will read this configuration file on startup. The system will write to this file when the user saves settings. |
| Architecture Studio *reads from* Default repository | This relationship denotes read operations on this directory structure. |
| Architecture Studio *writes to* Exported design | This relationship denotes a write operation; the system will create this file. |
| Architecture Studio *reads/writes to* Custom repository | This relationship denotes read and write operations on this directory structure. Information is written to this directory when the user creates or modifies a custom element or relationship. This directory is not available at the application level and is specific to a project. |

## Component-and-Connector Decomposition Level 2

Figure 3 shows the decomposition of the Architecture Studio element in the previous decomposition level.

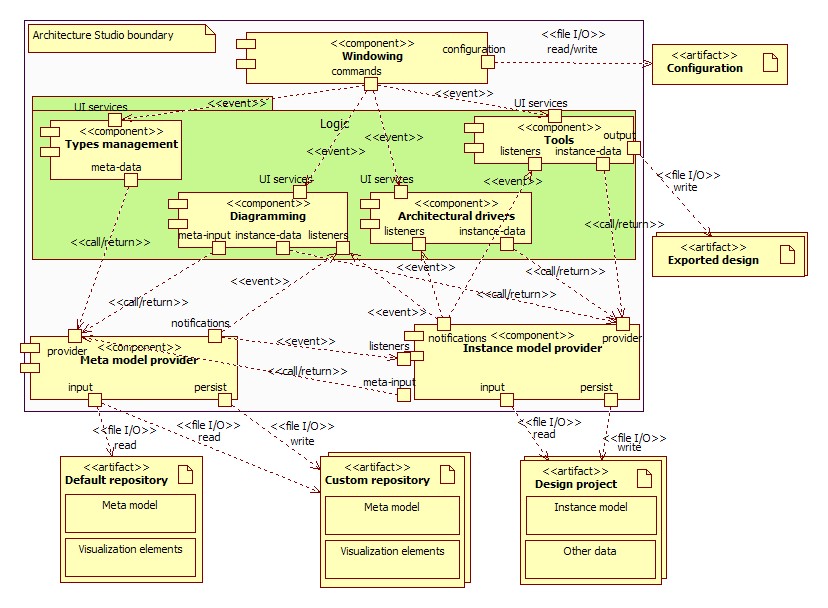


Figure Decomposition of Architecture Studio

### Decomposition Rationale

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 3  **Associated Responsibilities:** Catalog 3, Catalog 4 | **Perspective:** Component-and-Connector |
| This decomposition shows the logical components that make up the system. *Architecture Studio* is decomposed into components at this level by functionality. This increases the coherence within each component and decreases the coupling between components making it easier to further decompose each of these components.   * The *Windowing* component is responsible for managing the application window and the various panes within that window. The *Windowing* component includes subcomponents responsible for the menu bar, toolbar, status bar, panes, and size and location of the views and editors (but not the views and editors themselves). The functionality in this component is intended to be generic to any RCP-based desktop application. * Components in the *Logic* logical grouping are responsible for implementing specific application functionality. For example, the Diagramming component is responsible for the functionality required to draw the design diagrams. Subcomponents responsible for the views, editors, and dialogs required to support the diagramming functionality is included in the diagramming component. * The *Meta model provider* component is responsible for reading information about the element types from the *Custom repository* and *Default repository* and providing that information to other components. * The *Instance model provider* component is responsible for reading information about saved data from the *Design project* directory and providing that information to other components. It is also responsible for notifying components about changes in the instance model. | |

### Element Responsibilities Catalog

Catalog Elements - Level 2

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 3 | **Perspective:** Component-and-Connector |
| **Element** | **Responsibilities** |
| Windowing | This component is responsible for managing the application window and the various panes within that window. It is also responsible for initializing the application according to the application configuration. Subsequent changes to the configuration are also managed by this component.  The component in some cases passes commands to other components based on user actions. An example would be a user clicks a button in the toolbar that is related to a diagramming feature. This component will send a command to the Diagramming component, telling it to create the appropriate UI to interact with the user.   |  |  | | --- | --- | | Port | Description | | configuration | The component provides data to be written to the configuration file, and requires data that is read from the configuration file. | | commands | The component provides UI related commands to other components. The component requires other components to subscribe to these command events. | |
| Meta model provider | This component is responsible for reading information on meta models (both default and custom) and providing in memory representations of those models to other components.   |  |  | | --- | --- | | Port | Description | | input | The component reads in data about the meta model. | | provider | The component provides information about meta model elements through this interface. It can also take as input information about meta model elements and make changes in them. The component then uses the notifications port to publish events notifying other components of changes in meta model elements. | | notifications | The component provides notifications via events to registered components about changes in the meta model. | | persist | The component writes changes made to the meta model. Note that the application is only allowed to modify custom meta model elements. Default meta model elements cannot be changed. | |
| Instance model provider | This component is responsible for reading information on the instance model, which is represented by files in the Design project directory. It is responsible for providing other components access to the instance model, allowing them to examine, create, and modify instance model elements.   |  |  | | --- | --- | | Port | Description | | input | The component reads in data about the instance model. | | notifications | The component provides notifications via events to registered components about changes in the instance model. | | provider | The component provides information about instance model elements through this interface. It can also take as input information about instance model elements and make changes in them. The component then uses the notifications port to publish events notifying other components of changes in instance model elements. | | persist | The component writes changes made to the instance model. Changes are written to files in the Design project directory. | | listeners | The component requires event notifications about changes in the meta model, to be able to update its instance model (both in memory representations and saved information in the Design project) to match the meta model. | | meta-input | The component requires information on the meta model to be able to instantiate instance model elements. | |
| Diagramming | This component is responsible for handling all operations related to drawing diagrams on a canvas. It acquires instance model elements and uses them and the associated visualization elements to provide users with the ability to create diagrams and set the various properties of elements, relationships, views and other entities in a diagram.  Specifically, this component is also responsible for creating diagramming canvases and other related dialogs and Eclipse views to aid the user in drawing and manipulating design diagrams.   |  |  | | --- | --- | | Port | Description | | instance-data | The component requires instance model elements and uses them to create (and populate with data) design diagrams and supporting dialogs. The component also provides instance model elements to the Instance model provider component via this port, so that they can be saved. | | listeners | The component requires event notifications about changes in the instance model so that it can the appropriate operations (e.g. update the UI). It also receives notifications about changes in the meta model elements, so that it is able to update the palette when a meta model element changes. | | UI services | The component requires commands that tell it what UI elements to create to interact with the user. | | meta-input | The component requires meta model elements to create palettes. Palettes are part of diagram canvases that provide a list of available shapes that can be added to a diagram. | |
| Architectural drivers | This component is responsible for handling all operations related to architectural drivers that are saved in a Design project. It acquires instance models of architectural drivers and allows users to create, modify, and assign them to other instance model elements (such as elements, relationships, or groups).  This component is also responsible for creating dialogs and windows for manipulating architectural drivers.   |  |  | | --- | --- | | Port | Description | | instance-data | The component requires instance model elements and uses them to represent architectural drivers on screen. The component also provides instance model elements to the Instance model provider component via this port, so that they can be saved. | | listeners | The component requires event notifications about changes in the instance model so that it can the appropriate operations (e.g. update the UI). | | UI services | The component requires commands that tell it what UI elements to create to interact with the user. | |
| Types management | This component is responsible for providing tools to the user so that they can create their own (custom) visualization elements and use them in their design diagrams. It provides the user with a canvas in which they can create custom visualization elements (or modify existing ones), and save them for later use.  Additionally, this component is responsible for creating other UI elements such as dialogs that support the complete flow of managing custom visualization elements.   |  |  | | --- | --- | | Port | Description | | meta-data | The component requires visualization elements and meta model elements so that it can provide the user with a list of existing default visualization elements. The component provides representations of custom visualization elements and their associated meta model elements. This information can either represent new meta model elements or changes in existing ones. | | UI services | The component requires commands that tell it what UI elements to create to interact with the user. | |
| Tools | This component represents any extra functionality that is not part of the core drawing and documenting features of the system. Specifically, this component is responsible for:   * The export feature, which exports the Design project (architectural drivers, designs, and mappings) to an external format * Generating warnings on screen based on the state of architectural drivers and design diagrams in the project * Allowing the user to maintain a to-do list   Note that this component does not modify the instance model. It only uses the information present in the instance model to provide additional features.   |  |  | | --- | --- | | Port | Description | | model data | The component requires instance model elements and uses them for the various features (export, warnings, and to-do’s) that it supports. | | output | The component uses the instance model and saves them in an external format. | | UI services | The component requires commands that tell it what UI elements to create to interact with the user. | | listeners | The component requires event notifications about changes in the instance model so that it can the appropriate operations (e.g. update list of warnings). | |
| Default repository | See Catalog 1 for details. |
| Custom repository |
| Exported design |
| Design project |
| Configuration |

### Relationship Responsibilities Catalog

Catalog Relationships - Level 2

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 3 | **Perspective:** Component-and-Connector |
| **Relationship** | **Responsibilities** |
| **Windowing**[configuration] *reads/writes to* **Configuration** | The Windowing component reads and writes preferences to the configuration file. |
| **Windowing**[commands] *sends event to* **Types management**[UI services] | The Windowing component publishes commands related to the UI that are subscribed to by the Types management component. These commands tell the Types management component which drawing canvases or dialogs to create based on interaction with the user. |
| **Windowing**[commands] *sends event to* **Diagramming**[UI services] | The Windowing component publishes commands related to the UI that are subscribed to by the Diagramming component. These commands tell the Diagramming component which diagram canvases or dialogs to create based on interaction with the user. |
| **Windowing**[commands] *sends event to* **Architectural drivers**[UI services] | The Windowing component publishes commands related to the UI that are subscribed to by the Architectural drivers component. These commands tell the Architectural drivers component which dialogs to create based on interaction with the user. |
| **Windowing**[commands] *sends event to* **Tools**[UI services] | The Windowing component publishes commands related to the UI that are subscribed to by the Tools component. These commands tell the Tools component which dialogs to create based on interaction with the user. |
| **Meta model provider**[input] *reads from* **Default repository** | The Meta model provider component reads information on meta model elements and related visualization elements from the Default repository to be able to provide in-memory representations of default meta model elements. |
| **Meta model provider**[input] *reads from* **Custom repository** | The Meta model provider component reads information on meta model elements and related visualization elements from the Custom repository to be able to provide in-memory representations of custom meta model elements. |
| **Meta model provider**[persist] *writes to* **Custom repository** | The Meta model provider component writes the meta model elements created by the user to the Custom repository. These changes include new elements or modifications in existing ones. |
| **Instance model provider**[input] *reads from* **Design project** | The Instance model provider component reads information on the instance model from the Design project to be able to provide in-memory representations of instance model elements. |
| **Instance model provider**[persist] *writes to* **Design project** | The Instance model provider component writes the instance model to files in the Design project. These changes include new elements or modifications in existing ones. |
| **Instance model provider**[meta-input] *calls interface in* **Meta model provider**[provider] | The Instance model provider component gets information about meta model elements to be able to create instance model elements. |
| **Meta model provider**[notifications] *sends events to* **Instance model provider**[listeners] | The Meta model provider component publishes information about changes in the meta model to the Instance model provider component. |
| **Types management**[meta-data] *calls interface in* **Meta model provider**[provider] | The Types management component takes as input the list of default visualization elements and their associated meta model elements. Additionally, it passes information about changes made in the meta model to the Meta model provider. |
| **Diagramming** [meta-input] *calls interface in* **Meta model provider**[output] | The Diagramming component requires a list of meta model elements, which are provided by the Meta model provider. |
| **Diagramming** [instance-data] *calls interface in* **Instance model provider**[provider] | The Diagramming component requires instance model elements related to design drawings. It also provides instance model elements to be saved by the Instance model provider component. |
| **Instance model provider**[notifications] *sends events to* **Diagramming**[listeners] | The Instance model provider component publishes information about changes in the instance model to the Diagramming component. These are changes in instance model elements related to design drawings. |
| **Meta model provider**[notifications] *sends events to* **Diagramming**[listeners] | The Meta model provider component publishes information about changes in the meta model to the Diagramming component. These changes are used to keep the palette of shapes updated. |
| **Architectural drivers**[instance-data] *calls interface in* **Instance model provider**[provider] | The Architectural driver component requires instance model elements related to architectural drivers. It also provides instance model elements to be saved by the Instance model provider component. |
| **Instance model provider**[notifications] *sends events to* **Architectural drivers**[listeners] | The Instance model provider component publishes information about changes in the instance model to the Architectural drivers component. These are changes in instance model elements related to architectural drivers. |
| **Tools**[instance-data] *calls interface in* **Instance model provider**[provider] | The Tools component requires instance model elements to use it for the features that it supports. The Tools component does not provide any instance model elements to the Instance model provider because it does not make changes to the instance model. |
| **Instance model provider**[notifications] *sends events to* **Tools**[listeners] | The Instance model provider component publishes information about changes in the instance model to the Tools component. These changes are used to update, for instance, the list of warnings on screen. |
| **Tools**[output] *writes to* **Exported design** | The Tools component writes to disk (Exported design) all relevant information related to the instance model in an external format. |

## Module Decomposition Level 1

Figure 4 shows the first level of decomposition in the module perspective.

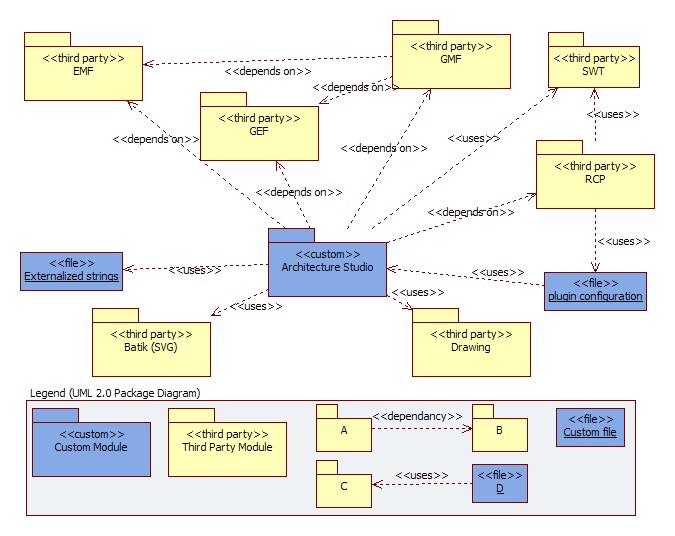


Figure Module Context - Initial context

### Decomposition Rationale

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 4  **Associated Responsibilities:** Catalog 5, Catalog 6 | **Perspective:** Module |
| This decomposition contains the initial partitioning of the system dictated by the use of the Rich Client Platform (RCP) and other third party libraries including Eclipse Modeling Framework (EMF), Graphical Editing Framework (GEF), and Graphical Modeling Framework (GMF). RCP provides infrastructure facilities for Architecture Studio, enabling it to use Eclipse as a foundation and allowing plug-ins to be written for the application. RCP uses a plugin configuration to relate the application code with services provided by RCP.  GMF uses both EMF and GEF to provide a unified set of API’s for applications to use and create modeling environments, such as the one provided in the Architecture Studio for crafting design diagrams. The application itself will also use EMF and GEF on its own.  Furthermore, Architecture Studio will also use an SVG library to manipulate SVG files that represent visualization elements. It will use a drawing library to allow users to create their own custom elements. It will also make use of SWT classes to draw complex dialogs and windows that are not supported by RCP. | |

### Element Responsibilities Catalog

Catalog Elements - Module Level 1

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 4 | **Perspective:** Module |
| **Element** | **Responsibilities** |
| Architecture Studio | This module will contain majority of the custom code written by the team and generated using EMF and GMF. This includes code related to models, controllers, views and parts required for an RCP application. |
| RCP | This module will provide the infrastructure facilities for the application. It will contain bootstrap code that will load the application. |
| GEF | This module will contain services provided by GEF, which will allow engineers to write views and controllers for diagrams in the application. |
| EMF | This module will contain dependencies for EMF generated code and models. |
| GMF | This module will include code that will provide an additional layer over EMF and GEF, allowing custom code to better leverage both of these frameworks. |
| Batik (SVG) | This module will provide services allowing the application to read, write, and render SVG files. |
| SWT | This module will provide services allowing the application to create complex dialogs and windows. |
| Drawing | This module will provide classes for creating freeform drawing canvases to allow users to create custom visualization elements. |
| plugin configuration | These files will be used to register our custom application and other relevant parts with RCP. |
| Externalized strings | This file will contain string messages that are displayed onscreen. These are stored in an external file to allow changing without compiling and facilitating internationalization. |

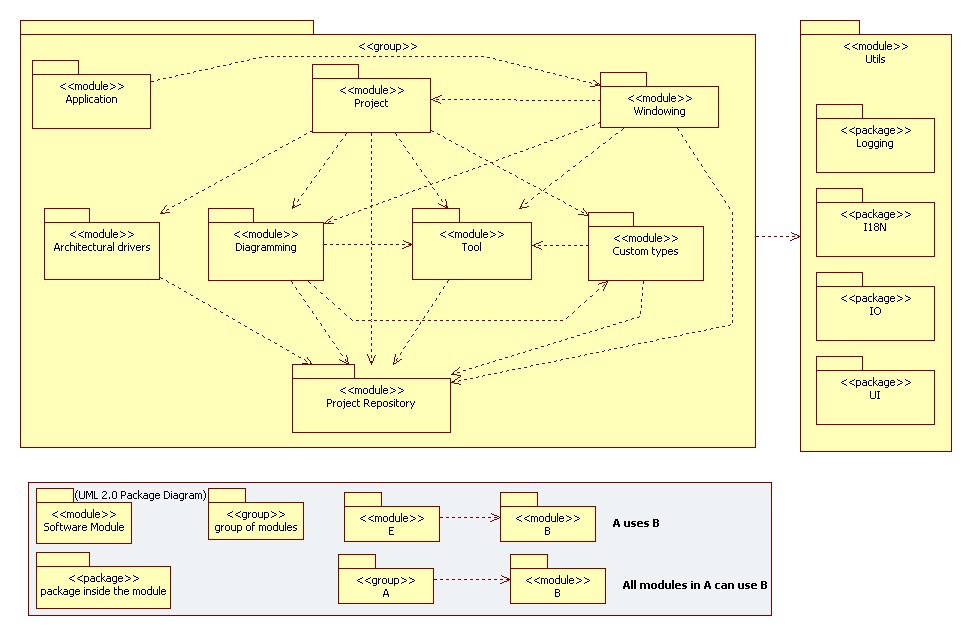
### Relationship Responsibilities Catalog

Catalog Relationships - Module Level 1

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 4 | **Perspective:** Module |
| **Relationship** | **Responsibilities** |
| Architecture Studio *depends on* RCP | Code in the custom module will implement and inherit from code in the RCP packages, e.g. WorkbenchAdvisor, ISelectionListeners, and IPlatformRunnable to name a few. |
| Architecture Studio *depends on* GEF | Code in the custom module will implement and inherit from code in the GEF module, e.g. AbstractEditPart, EditPartFactory and GraphicalEditor to name a few. |
| Architecture Studio *depends on* EMF | Code in the custom module will implement and inherit from code in the EMF module, mostly notably the Ecore meta model classes. Additional classes will be used to manipulate models. |
| Architecture Studio *depends on* GMF | Code in the custom module will make use of code generated by the GMF modeling tool to use EMF and GEF together. |
| Architecture Studio *uses* Batik (SVG) | Code in the custom module will use this library to save and load SVG files and to render SVG files onscreen outside of GMF canvases. |
| Architecture Studio *uses* Drawing | Code in the custom module will use this library to provide users with drawing canvases on which they can draw custom elements. |
| Architecture Studio *uses* SWT | Code in the custom module will use SWT classes to create complex dialogs and windows. |
| Architecture Studio *uses* Externalized strings | Code in the custom module will refer to these string variables, and will know how to load them at a particular point in the application’s execution. |
| GMF *depends on* GEF | GMF generated code will use services provided by GEF to create views for its EMF models. |
| GMF *depends on* EMF | GMF generated code will use EMF defined models for our modeling language (different architectural elements and relationships etc.) and use them to display diagrams to the user via GEF. |
| RCP *uses* SWT | RCP uses SWT classes to provide prebuilt commonly used dialogs and windows. |
| RCP *uses* Plugin configuration | RCP bootstrap code will use this file to identify and locate files related to our custom RCP application. |
| Plugin configuration *uses* Architecture Studio | The plugin configuration (plugin.xml and other files) will reference specific classes in the custom code which will allow the RCP infrastructure to use these classes at runtime. |

## Module Decomposition Level 2

The figure below shows the decomposition of the *Architecture Studio* module from the previous decomposition level. *Note because there is only type of relationships (uses) in this view, we have not used stereotypes for relationships to enhance the readability of the diagram.*

Figure System - Decomposition of the Architecture Studio packages

### Decomposition Rationale

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 5  **Associated Responsibilities:** Catalog 7, Catalog 8 | **Perspective:** Module |
| This decomposition shows major module structures in the Architecture Studio module and their dependencies between them. The Application modules dependency on all feature modules is a result of how the initialization mechanism is designed in RCP. By *feature modules* we are referring to the Custom types, Diagramming, Architectural drivers, Tool modules.  All feature modules register to events posted by the Windowing component, thus they need to reference it.  All feature modules need access to the instance model, and some need access to the meta model. To enhance semantic coherence of the design, this common service of accessing the model has been abstracted into the Project Repository module.  Separating features into their own modules contributes to the understandability of the architecture, and allows us to identify dependencies between modules by using knowledge about features they will be implementing.  All modules will make use of the three helper modules:   * Internationalization (I18N) * Utils * Logging * UI   These three modules ensure that this logic is not spread out through the whole system. This group is only a logic grouping to make the understanding of the diagram easier. Otherwise all three modules would have connection to every other module. | |

### Element Responsibilities Catalog

Catalog Elements – Module Level 2

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 5 | **Perspective:** Module |
| **Element** | **Responsibilities** |
| Application | This module is responsible for providing the scaffolding for the system. It is responsible for providing code that reads configuration, and starts up the system. |
| Project | The Project module is responsible for the project management, which includes creating new projects, saving and loading a project. It is also responsible for exporting the design of the project to an external file format suitable for printing, such as DOCX, PDF, or HTML. |
| Windowing | This module is responsible for maintaining the main UI window of the application and allowing other UI components to register to events published by it. |
| Custom types | This module is responsible for providing features that allow the user to create their own custom types, and save and modify them. |
| Diagramming | This module is responsible for providing features that allow the user to create design diagrams. |
| Architectural drivers | This module is responsible for providing features that allow the user to manage architectural drivers. |
| Tool | This module contains different functionality and is responsible for the following things:   * Monitor the state of information in the project and providing warnings to the user regarding good architectural practices. * Allow the user to manage a to-do list * Trace all the assigned architectural drivers. |
| Project Repository | This module is responsible for using the meta model and providing a layer of services over them, allowing other modules to use the meta model and instance model. It also provides the ability to save and load meta model elements and instance model elements. This module is also used to create new meta model of the element and relationship types. |
| I18N | This module contains all the externalized strings for internationalization. These strings are stored in configuration files which can be changed without recompiling the code. This helps to correct spelling mistakes or just rename labels and other fields. |
| Utils | This module contains all the utile components which are used throughout the application. IO will be part of this module, which handles all the interaction with the operation system. Create new directories and read and write files. Separating this functionality in one module should make it easier to support other operating systems. |
| Logging | This module is the logging component for the whole application |

### Relationship Responsibilities Catalog

Catalog Relationships - Module Level 2

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 5 | **Perspective:** Module |
| **Relationship** | **Responsibilities** |
| Application *uses* Windowing | The Application module references classes in the Windowing module which are responsible for creating the main UI windows. |
| Windowing *uses* Project | The Windowing module references classes in the Project module which are responsible for creating a new project. |
| Windowing *uses* Diagramming | The Windowing module references classes in the Diagramming module to inform them about action the user performed. |
| Windowing *uses* Tool | The Windowing module references classes in the Tool module to inform them about action the user performed. |
| Project *uses* Custom types | The Project module references classes in the Custom types that are responsible for creating for creating UI. |
| Project *uses* Diagramming | The Project module references classes in the Diagramming module that are responsible for creating UI. It also references classes in the Diagramming module to get the generated images of the diagrams. |
| Project *uses* Architectural drivers | The Project module references classes in the Architectural drivers module that are responsible for creating UI. |
| Project *uses* Tool | The Project module references classes in the Tool module that are responsible for creating UI. |
| Custom types *uses* Project Repository | The Custom types module uses meta model providers in the Project Repository module. |
| Custom types *uses* Tool | The Custom types module references classes in the Tool module that provide the decorations for relationship types. |
| Diagramming *uses* Project Repository | The Diagramming module uses instance model providers and meta model providers in the Project Repository module. |
| Diagramming *uses* Tool | The Diagramming module references classes in the Tool module that provide decoration for relationship types. |
| Diagramming uses Custom types | The Diagramming references a dialog in custom types that it can be called in the diagramming. This is a weak reference since it just opens a dialog in the Custom types module. |
| Architectural drivers *uses* Project Repository | The Architectural drivers module uses instance model providers in the Project Repository module. |
| Tool uses Project Repository | The Tool module uses instance model providers in the Project Repository module. |
| Group *uses* I18N | All elements within the group will use the I18N package. |
| Group *uses* Utils | All elements within the group will use the IOUtils package. |
| Group *uses* Logging | All elements within the group will use the Logging package. |
| Group *uses* UI | All elements within the group will use the UI package. |
| Windowing *uses* Project Repository | The Windowing module references the Exceptions in the Project Repository since they can be end up in the Windowing module |
| Project *uses* Project Repository | The Project module references classes in the Project Repository to manage a new project and get information to generate the exported document. |

## Allocation Decomposition Level 1 – Deployment Style

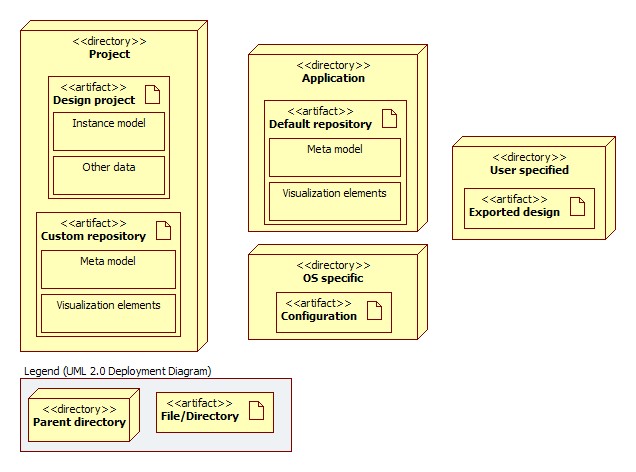


Figure Mapping of artifacts to directories

### Decomposition Rationale

|  |  |
| --- | --- |
| **Associated Drawings:** Figure 6  **Associated Responsibilities:** N/A | **Perspective:** Allocation |
| This decomposition shows the mapping of artifacts defined in the level 1 connector-and-component decomposition (Figure 2) to directories. The purpose of this decomposition is to mainly show which artifacts are at which directory level.  The Design project and Custom repository are at the ‘project’ level, meaning that there is one of each for every project. The implication of this is that custom types are not sharable between projects.  The Default repository is stored with the application, typically in the directory in which the application is installed. This implies that all default types are available to all projects.  The Configuration is also available at the application level, but depending on the operating system, its location may vary (e.g. in Windows Vista, configuration files cannot be in Program Files where an application is installed, they must be in a user profile directory).  The Exported design is obviously created in a directory of the user’s choosing.  *Due to the simplicity of this decomposition, an element responsibilities catalog is not provided.* | |

## Allocation Decomposition Level 1 – Eclipse Project

This Allocation view reflects the project structure. This ensures that if a project needs to be changed it won’t affect the whole application. The project name is the same as the base package name in each project.

We will have the following projects:

|  |  |
| --- | --- |
| Project and Package name | Modules (Decomposition Level 2) |
| architecturestudio | Application, Project, Windowing |
| architecturestudio.architecturaldriver | Architectural drivers |
| architecturestudio.customtype | Custom types |
| architecturestudio.diagramming | Diagramming |
| architecturestudio.tool | Tool |
| architecturestudio.project  architecturestudio.project.edit | Project Repository |
| architecturestudio.util | Logging, IO, I18N |

# Mappings between Perspectives

This section details the mappings between the different components and modules in the component-and-connector and module views respectively. Components from level 2 decompositions in the component-and-connecter perspective are mapped onto modules from the level 2 decomposition in the module perspective.

|  |  |  |
| --- | --- | --- |
| Component | Modules | Comments |
| Windowing | Application, Windowing, Project |  |
| Types management | Custom types |  |
| Diagramming | Diagramming |  |
| Architectural drivers | Architectural drivers |  |
| Tools | Tool |  |
| Meta model provider | Project Repository |  |
| Instance model provider | Project Repository |  |

# Domain Model

Figure 7 represents the domain model of the primary set of datum that is used by the application. It shows the various types of data and their relationships with each other. This will be used as a reference for creating the abstract model in EMF and mapping it to the graphical concrete syntax in GMF.

Please review the Architectural Drivers Specification document before examining the domain model as it assumes knowledge of the various types of data that the system is required to store. Classes shown in the blue overlay in the domain model are related to design diagrams (they contain data about elements and relationships that are displayed in a diagram). Classes in the green overlay are related to architectural drivers (they contain data about architectural drivers that the user enters into the system).

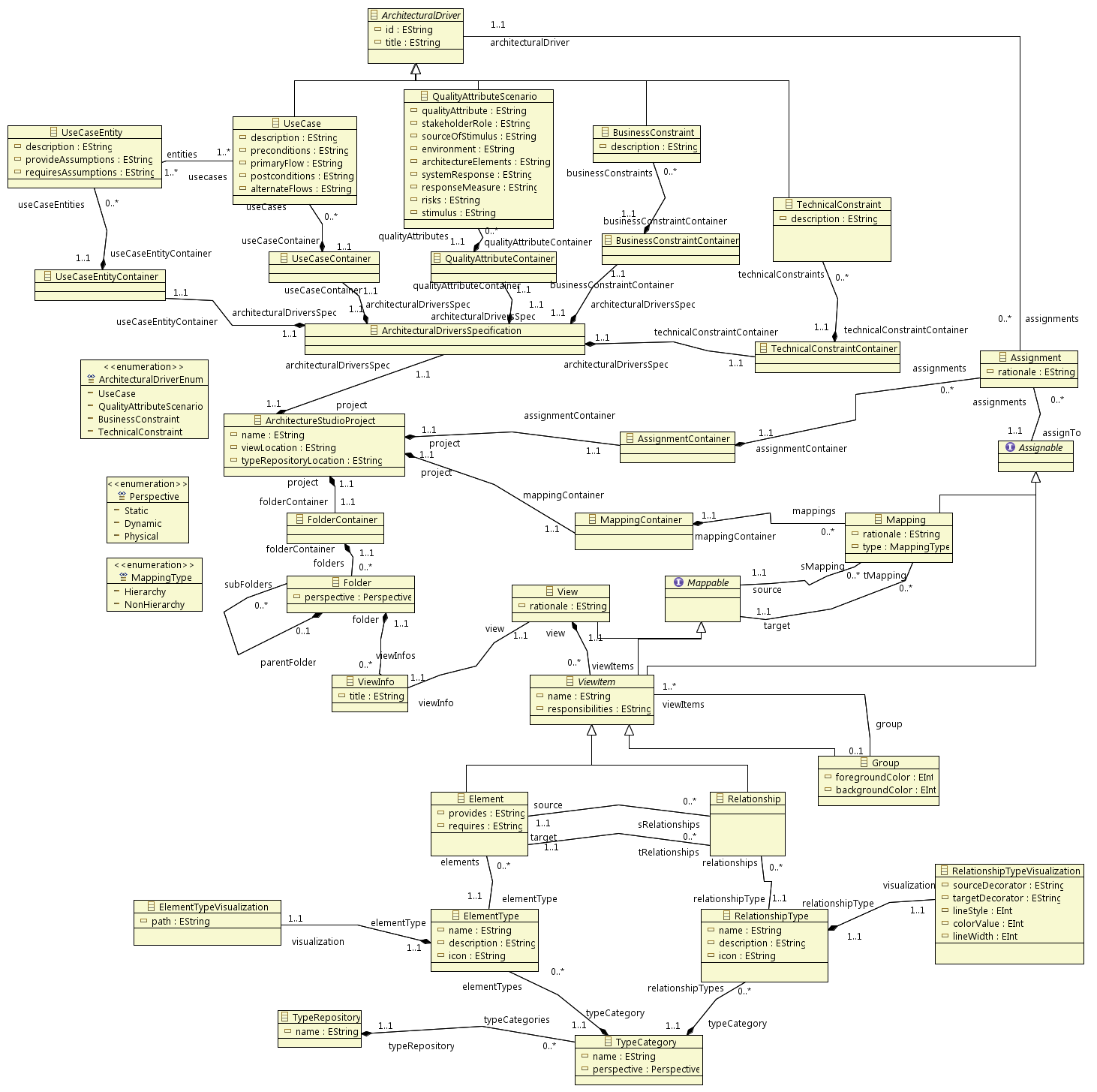


Figure Domain Model

A View[[1]](#footnote-1) is the container of all diagram entities. A diagram entity is basically an element or relationship. Additionally, elements and relationships can be grouped together. Mappings can be created between views, elements, relationships and groups of elements and relationships. A mapping can have different types. Architecture drivers have four types, as shown in the domain model. Any architecture driver can be assigned to an element, a relationship, a group or a mapping.

An element can be connected with an arbitrary number of relationships. A relationship on the other hand can only have one source and one target (like a mapping). Elements and relationships have a *type* attribute, which is used to define how the element or relationship looks like when it is displayed on screen.

# File Formats

This section contains information on the format and structures of various files and directories that will be used by the system.

|  |  |  |
| --- | --- | --- |
| Element (file/directory) | Drawing(s) | Description |
| Design project | Figure 2 | This directory will contain XML file(s) that represent the instance model of the project. Other data that is not part of the instance model will be stored in an XML format as well. |
| Configuration | Figure 2 | For the configuration file we use the properties support from Java which supports access to key=value pairs. |
| Default repository | Figure 2 | **Directory structure**  The following properties of the directory structure and its files are to be determined:   * Effect of the directory structure on categorization of shapes. Perhaps each category of shapes can be stored in a separate directory. * Location and format of file containing information on categories of shapes.   **Visualization elements**  The storage format of visualization elements will be SVG, as that prevents shapes from degrading when scaled. It is also suitable for saving custom visualization elements.  **Meta model**  The storage format for meta model elements will be XML. |
| Custom repository | Figure 2 | Files in this directory follow the same structure and format as in the *Default repository* directory. |
| Exported design | Figure 2 | This will be a format external to the system, most probably either .PDF or .DOCX. Exact layout and formatting of sections in this file is to be determined. |

# Requirements Traceability Matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requirement satisfied** | **Design decision** | **Element** | **Relationship** | **Design artifact reference** | **Comments and descriptions** |
| QAS1 – Freedom when drawing shapes | This quality attribute is related to the layout and labeling of the user interface, and as such does not have an influence on the software architecture. | | | | |
| QAS2 –Intuitiveness of user interface (UI) of perspectives and their views | This quality attribute is related to the layout and labeling of the user interface, and as such does not have an influence on the software architecture. | | | | |
| QAS3 – Keyboard shortcuts | To use RCP | RCP | Architecture Studio *depends on* RCP | Figure 4 | RCP provides built in support for assigning keyboard shortcuts to commands |
| QAS4 – Quick installation procedure | To use RCP | RCP | Architecture Studio *depends on* RCP | Figure 4 | The RCP development environment provides tools to generate installation files for a project. |
| QAS5 – Portability to non-Windows platforms | To use RCP | RCP | Architecture Studio *depends on* RCP | Figure 4 | The RCP development environment allows us to generate installation files for a project for multiple platforms via a wizard. |
| QAS6 – Ability to add features | This quality attribute scenario is refined by QAS7 and QAS8. | | | | |
| QAS7 – Ability to add new types of default shapes or relationships | * To store meta model properties and visualization elements on disk * *type* property in Element and Relationship class in the domain model * To implement custom element/relationship type requirement | Default repository artifact, Custom models component | Meta model provider *reads from* Default repository, Custom models *reads/writes to* Custom repository | Figure 3, Figure 7 | Storing information about elements and relationships outside the application allows us to add new types without recompiling the application. Having a *type* attribute for elements and relationships at the meta model level allows us to assign different visualization elements and other properties to meta model elements without having to change code.  Having to implement the custom element/relationship type requirement also helps with this because we have to ensure that the meta model allows easy addition/removal of element and relationship types. |
| QAS8 - Ability to add new properties to elements | To use EMF and the structure of the domain model | EMF | Architecture Studio *depends on* EMF | Figure 4 | Meta models will be stored as XML files, and it is possible to modify Meta model element classes to use a specific attribute in the XML node to hold arbitrary properties. |
| UC01 - Capture functional requirement | To use SWT to create dialogs for managing input. | Windowing, Architectural drivers, Design project | Windowing *calls interface in* Architectural drivers. Architectural drivers *calls interface in* Instance model provider, Instance model provider *writes to* Design project | Figure 3 | This set of requirements is implemented using RCP (and SWT) to build the UI and using EMF to manage storage of architectural drivers. |
| UC02 - Capture quality attribute |
| UC03 - Capture technical constraint |
| UC04 - Capture business constraint |
| UC05 - Create/Remove a view | To use GMF | Windowing, Diagramming,  Meta model provider | Windowing *calls interface in* Diagramming. | Figure 3 | Using GMF allows us to create fully featured GEF diagram canvases which are integrated with EMF for managing storage. |
| UC06 - Draw a design in a view | To use GMF | Diagramming,  Meta model provider, Instance model provider | Diagramming *calls interface in* Meta model provider. Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 |
| UC07 - Define new element or relationship type | To use GMF and SVG | Windowing, Types management,  Meta model provider, Custom repository, Batik (SVG) | Windowing *calls interface in* Types management. Types management *calls* *interface in* Meta model provider.  Meta model provider *reads/writes to* Custom repository.  Meta model provider *reads from* Default repository.  Architecture Studio *uses* Batik (SVG) | Figure 3, Figure 4 | Using SVG to store visualization elements allows creating element types at runtime and loading them into GEF diagrams. Information about how to extend the meta model (using a *type* attribute) is discussed above. |
| UC08 - Edit element or relationship type |
| UC09 - Remove element or relationship type |
| UC10 - Reset toolbox |
| UC11 - Manage element and relationship catalogs | To use GMF and SWT | Diagramming,  Instance model provider | Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 | GMF facilitates managing attribute values (such as rationale) of diagram elements. SWT allows us to create rich dialogs to get input from the user. |
| UC12 - Capture design rationale for view |
| UC13 - Capture design rationale for an element, relationship or group |
| UC14 - Assign/Remove an architectural driver to/from an element or relationship or a group | To use GMF and SWT | Diagramming,  Instance model provider | Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 | Using GMF diagramming canvases and SWT will allow us to create dialogs that interact with elements inside the diagram canvas. |
| UC15 - View/Remove an element or relationship to which an architectural driver is assigned |
| UC16 - Decompose an element | To use GMF | Windowing, Diagramming,  Meta model provider | Windowing *calls interface in* Diagramming. | Figure 3 | Similar to UC05. |
| UC17 - Create a mapping between different views | To use GMF and SWT | Diagramming,  Instance model provider | Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 | Parts of this feature will be implemented in SWT dialogs with some in GMF diagramming canvases. |
| UC18 - Create a mapping for groups |
| UC19 - Delete a mapping |
| UC20 - Capture design rationale for a mapping | To use GMF and SWT | Diagramming,  Instance model provider | Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 | Similar to UC12. |
| UC21 - Assign/Remove architectural drivers to/from a mapping of elements | To use GMF and SWT | Diagramming,  Instance model provider | Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 | Using GMF diagramming canvases and SWT will allow us to create dialogs that interact with elements inside the diagram canvas. |
| UC22 - Use UML to draw detail design | To use GMF | Diagramming,  Meta model provider, Instance model provider | Diagramming *calls interface in* Meta model provider. Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 | Similar to UC06. |
| UC23 - Warn user about unmet design obligations | To use RCP | Tools, Instance model provider | Tools *calls interface in* instance model provider. | Figure 3 | Warnings will be displayed using an RCP view. |
| UC24 - silence warning | This use case has been removed since the ADS version 1.2 | | | | |
| UC25 - Manage tasks for the project | To use RCP | Tools, Instance model provider | Tools *calls interface in* instance model provider. | Figure 3 | To-do’s will be displayed using an RCP view and stored in the instance model. |
| UC26 - Manage a project | To use RCP and GMF | Instance model provider, Design project | Instance model provider *reads/writes to* Design project | Figure 3 | GMF generated RCP applications provide functions for saving and loading the project. |
| UC27 - Export a project | To use GMF | Windowing, Tools, Instance model provider,  Exported design | Windowing *calls interface in* Tools. Tools *calls interface in* Instance model provider. Tools *writes to* Exported design. | Figure 3 |  |
| UC28 - Group elements and relationships | To use GMF | Diagramming,  Meta model provider, Instance model provider | Diagramming *calls interface in* Meta model provider. Diagramming *calls interface in* Instance model provider. Instance model provider *sends events to* Diagramming. | Figure 3 | Similar to UC06. |
| UC29 – Ungroup elements and relationships |
| UC30 – View decomposition of an element | To use GMF | Windowing, Diagramming,  Meta model provider | Windowing *calls interface in* Diagramming. | Figure 3 | Similar to UC05. |

1. This represents an ACDM view, not an Eclipse view. [↑](#footnote-ref-1)