rationally

Software Architecture Specification

by Ronald Kruizinga and Ruben Scheedler

1 Introduction

Within the context of software intensive systems (SIS), knowledge management is inherently a part of their development [1]. In this same context, architects from different disciplines will work on one system. Knowledge about the design of the system and design decisions that are being made while creating it are divided amongst these architects. What makes architecting SIS difficult is this partitioning of knowledge. Architects of SIS are required to collaborate closely and share this knowledge to create a properly working system. *rationally* intends to be a tool that makes the process of knowledge sharing more efficient and effective.

Besides streamlining the knowledge sharing process between architects, *rationally* makes the architectural knowledge (design decisions + design) explicit and visualizes it in a systematic manner. That is, it visualizes it in an architecture decision viewpoint. *rationally* views can thus be used as a work of reference for meetings and it can be iterated over, to capture the design process of architecting the system.
Furthermore, *rationally* allows for knowledge personalization: making explicit which person holds what knowledge by documenting this information.

A part of the architectural knowledge (AK) of the system is thereby being codified. *rationally* is capable of providing a snapshot or overview of certain design (design AK) related to an AD (reasoning AK).
This codification process is also made easier for architects by *rationally*: it was built as an add-in for Microsoft Visio, which is widely used throughout the industry[2] and offers the flexibility that a multidisciplinary team requires, due to the many different approaches to the problem, through its large collection of visual components. With these components, diagrams can be constructed quickly that capture the AK in a visual way.

This architecture document consists of the following sections. We explain what *rationally* is required to do, what decisions were made in the process of creating it, and explain the rationale behind those decisions. In section 3, we will place *rationally* in a context, describing its relations to actors and other related systems. Next, in section 4, we state our vision, we list relevant stakeholders and work out the main use cases of our product (section 4.3). After this, we go into the functional and non-functional technical requirements of *rationally* (section 4.4).
The software architecture of *rationally* is set out in section 5. It entails assumptions that were made, a logical view of our application and finally an overview of the decisions that were made during the creation of *rationally*.
After the software architecture section, we go into verification and validation (section 6) of our application, we describe possible improvements that can be made to *rationally* in the future, and we summarize our work.

2 Glossary

Context Menu - Shapes in Visio can be right-clicked to show a menu with options for that shape. The Visio API offers developers to extend these menus with custom options. Context menus can be unique per shape.
Decision Sheet - The area within Microsoft Visio onto which users can drag shapes.
Enhanced Container- A container shape that offers tailored operations to the user to optimize the processing of information into it. For example: the *Forces* view offers the *"add force"* operation that automatically creates a row with all required cells and force values.
Enhanced Item - A shape that is a direct child of an enhanced container that offers tailored operations to the user to optimize manipulation of it, like the list the operations "add", "delete", "move up" and "move down". These items often consist of several shapes but are offered as atomic entities to user.
Enhanced Shape A shape that is part of an enhanced item that offers tailored operations to the user to optimize manipulation of it. For example, force values change background according to their value and allow users to move up/down the force of which it is a part.
Shape Visible object in Visio that is completely customisable.
Shapesheet Table-like object that stores a shape's properties.
View - work product expressing the architecture of a system from the perspective of specific system concerns. [6]
Viewpoint work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns. [6]

3 System Context

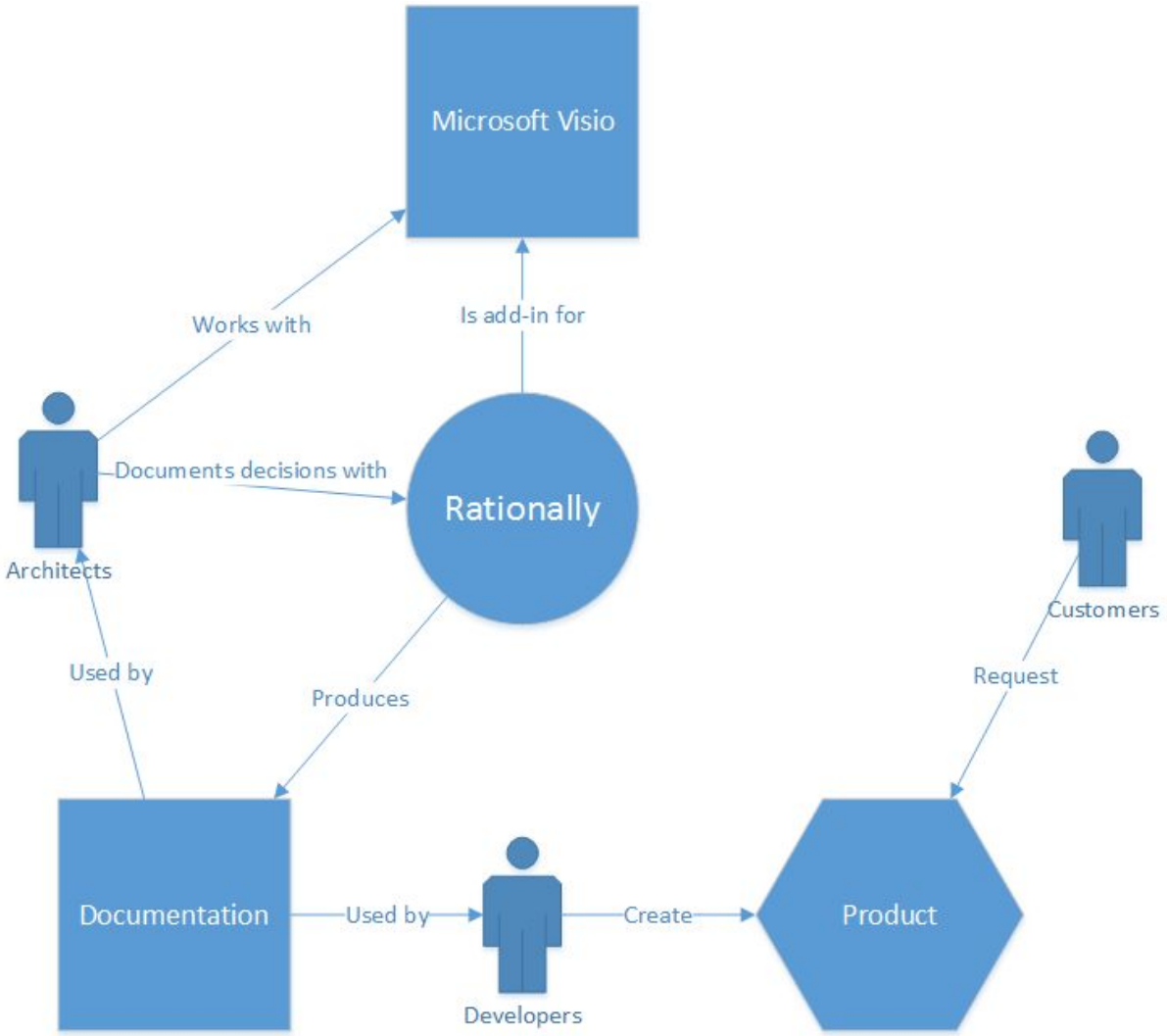


Figure 1: System context diagram for `rationally`.

`rationally` extends the functionality of Microsoft Visio™ in a way that allows an easy documentation of decisions made during the creation and maintenance of software-intensive systems. Software architects are therefore the main actor in the context of our application. They use Visio together with `rationally` during the creation of the architecture of a system.

The created decision documentation has one main purpose: it preserves architectural knowledge. This knowledge/documentation on itself has several purposes. It allows sharing between architects in a format, understandable for architects in different disciplines and can also be used by developers to understand the reasoning behind the design they are to implement in a software product. Finally architects can base new decisions on old ones, relating them to each other, which gives rise to a collection of related decisions that capture the process of creating an architectural design of a system.

4 Requirements

4.1 Architectural Vision

`rationally` intends to be a useful tool for software architects that have adopted or want to adopt the viewpoint-based approach regarding design decisions. It should combine information relevant to several known viewpoints (detail viewpoint, the chronological viewpoint, the forces viewpoint, the decision relationship viewpoint and the stakeholder involvement viewpoint) into one view: a decision view.

`rationally` will offer a sheet that contains areas to enter information relevant to each viewpoint. All these areas come with tailored operations that allow for fast and clear documentation of decisions. Furthermore, relations between views are implemented and, in the future, relations between decisions as a whole will be documentable as well.

4.2 Stakeholders

The stakeholders of our product can be split into two groups. The first group has to do with the production of the decision views. The second group are related to the consumption of the decision views.

4.2.1 Production Stakeholders

System Architects
Software architects require a way of documenting their reasoning behind a certain design. When making design decisions, they consider various viewpoints and they want to be able to document views for all of them in an efficient manner.

Development Company
The Architectural Knowledge that is created using `rationally` can help new architects of a system to understand it faster and better, yielding a product earlier that might even work better. This benefits the company developing systems that uses `rationally`, since it can shorten development time and increase the maintainability of those systems.

4.2.2 Consumption Stakeholders

Developers
Developers do not only benefit from a detailed software design, but can also benefit from knowing the reasoning behind it.

4.3 Use-Cases

- UC0 - Documenting the decision detail viewpoint of a decision.
- UC1 - Manipulate the decision sheet via the wizard
- UC2 - Add components to the decision sheet using drag-n-drop functionality
- UC3 - Manipulate the decision sheet using context menus

UC0 - Documenting the decision detail viewpoint of a decision.

A user wants to document the decision detail viewpoint of a decision, using `rationally`. This entails giving a description for the decision, defining various alternatives and arguments for or against the arguments. The user also marks the states of all the alternatives.

Main actor: Software architect
Stakeholders:
- Software Architect - *Documenting a description of and the alternatives for a certain decision can become very useful. A tempting, but bad, alternative can be marked as so using arguments and better alternatives.*

Flow 0 (sunny day scenario):
1 The user opens Visio.
2 The user creates a *decision sheet*.
3 The user makes a description for the decision, using Visio shapes.
4 The user adds an alternative, using the context menu on the alternatives area.
5 The user fills in a title (“Alternative A”) and a state (“Accepted”).
6 The user adds a description to the alternative, using Visio shapes.
7 The user adds a second alternative, using the context menu on the alternatives area.
8 The user fills in a title (“Alternative B”) and a state (“Discarded”).
9 The user adds a description to the second alternative, using Visio shapes.
10 The user adds arguments for and against the alternatives, in the *arguments* area.

UC1 - Manipulate the decision sheet via the wizard

`rationally` offers various ways of manipulating the decision view. One of these is via a wizard. This use case describes some frequently occurring flows from the user perspective.

Main actor: Architect
Stakeholders:
- Architect - *Although `rationally` offers context menus to add elements to the view, this is not a fast way to add/remove many items at once. The wizard offers a more pleasant more of entering text as well, besides offering faster add and remove options.*

Flow 0 (sunny day scenario):
1 The user opens the wizard, or creates a new decision, and is prompted with the wizard.
2 The user fills in the general information tab (title, author, date, version)

3 The user adds elements on various tabs (e.g. alternatives, forces and stakeholders)
4 The user clicks the “create/update decision” button.
5 The view is updated to represent the wizard’s information.

Flow 1 (rainy day scenario “invalid data”):
1 The user opens the wizard, or creates a new decision, and is prompted with the wizard.
2 The user fills in the general information tab (title, author, date, version)
3.1 The user adds elements on various tabs (e.g. alternatives, forces and stakeholders), but did not fill in all required fields
3.2 The user is prompted with a message stating what fields are filled in incorrectly or not at all.
3.3 The user fixes the incorrectly filled in fields.
4 The user clicks the “create/update decision” button.
5 The view is updated to represent the wizard’s information.

Flow 2 (rainy day scenario “no general information”):
1 The user opens the wizard, or creates a new decision, and is prompted with the wizard.
2.1 The user does not fill in general information tab (title, author, date, version)
3 The user adds elements on various tabs (e.g. alternatives, forces, stakeholders)
4 The user clicks the “create/update decision” button.
5 The view is updated to represent the wizard’s information.

Flow 3 (rainy day scenario “premature close”):
1 The user opens the wizard, or creates a new decision, and is prompted with the wizard.
2 The user fills in the general information tab (title, author, date, version)
3 The user adds elements on various tabs (e.g. alternatives, forces and stakeholders)
3.1 The user closes the wizard, discarding the filled in information.

UC2 - Add components to the decision sheet using drag-n-drop functionality

`rationally` offers a stencil to the user that contains the shapes that are used in the decision view. All enhanced containers are, together with their content shapes, included in `rationally`’s stencil (see requirement 9.3 and 9.4). These shapes can be dragged from the stencil on to the view and are placed in the correct area. This allows architects to quickly add new elements to the view.

Main actor: Architect
Stakeholders:

- Architect - *Drag-n-drop allows for faster adding of elements to the view, benefiting the architect that is creating the document.*

Flow 0 (sunny day scenario):
1 User selects a shape from the `rationally` stencil and drags in onto the view.
2. The shape is added to the appropriate enhanced container.

Flow 1 (rainy day scenario “shape requiring user input”):
1 The user selects a shape from the `rationally` stencil and drags in onto the view.
1.1 The user is prompted with a dialogue, to enter the relevant properties of the selected shape (like a title for an alternative, or a source file for a related document).
1.2 The user fills in the required information.
2 The shape is added to the appropriate enhanced container.
Flow 2 (rainy day scenario “required user input not given”):
1 The user selects a shape from the `rationally` stencil and drags in onto the view.
1.1 The user is prompted with a dialogue, to enter the relevant properties of the selected shape (like a title for an alternative, or a source file for a related document).
1.2 The user does not fill in the required information.
1.3 The shape is not added to the decision sheet.
Flow 3 (rainy day scenario “no container for the selected shape”):
1 The user selects a shape from the `rationally` stencil and drags in onto the view.
1.2 The appropriate container is not present on the decision sheet.
1.3 The shape is not added to the decision sheet.

UC3 - Manipulate the decision sheet using context menus

`rationally` is a Visio add-in and will likely be used by architects that are already familiar with Visio. One of the traditional ways that Visio offers to manipulate a sheet is by the offering of context menus on shapes. `rationally` has implemented these menus on its shapes as well.

Main actor: Architect
Stakeholders:
- Architect - *This actor might benefits from a familiar way of manipulating the decision sheet.*

Flow 0 (sunny day scenario):
1 The user right clicks an enhanced container or enhanced item, opening a context menu.
2 The user selects a desired operation (add, delete, move up, etc.)
3 The requested desired operation is executed.
Flow 1 (rainy day scenario “shape requiring user input”):
1 The user right clicks an enhanced container or enhanced item, opening a context menu.
2 The user selects a desired operation (add, delete, move up, etc.)
2.1 The user is shown a pop-up, requesting additional information (like a title for an alternative that will be added)
2.2 The user fills in the required information.
3 The requested operation is executed.
Flow 2 (rainy day scenario “required user input not given”):
1 The user right clicks an enhanced container or enhanced item, opening a context menu.
2 The user selects a desired operation (add, delete, move up, etc.)
2.1 The user is shown a pop-up, requesting additional information (like a title for an alternative that will be added)
2.2 The user does not fill in the required information.
2.3 The requested operation is cancelled.

4.4 Functional requirements

In this section we will go into the functional requirements for `rationally`. The decision sheet of `rationally` consists of various views (alternatives, forces, etc.). All these views have some tailored requirements, but also share some general required functionality. This last part is described in *R0 - General view requirements*. After that, we describe the specific requirements for each view on the decision sheet and the requirements for the wizard.

4.4.1 Constraints

1. All enhanced containers are present once or not at all on the decision sheet.
2. Attempts of the user to change the layout of the enhanced views may be overwritten by `rationally`.
3. The user is not allowed to use the grouping functionality offered by Visio on shapes that are managed by `rationally` (part of enhanced containers).
4. The user is only allowed to have one instance of the wizard open at a time per running instance of Microsoft Visio.

R0 - General requirements

All views that offer special functionality (alternatives, stakeholders, but not description) are basically lists. To manipulate lists, several operations are to be offered to the user, in order to manipulate `rationally` views.

R0.1 A list item should always be addable via a context-menu.
R0.2 A list item should always be deletable via a context-menu.
R0.3 It should be possible to move a list item up one place in the list via a context-menu.
R0.4 It should be possible to move a list item down one place in the list via a context-menu.
R0.5 It should <u>not</u> be possible to move up the top list item.
R0.6 It should <u>not</u> be possible to move down the bottom list item.
R0.7 All views must be addable at all times.
R0.8 All views must be deletable at all times.
R0.9 The template should contain all views by default.

R1 - General information of a decision

The user should be able to document some general information regarding the decision. A topic, a version, an author and a date should be asked of the user on creation of a decision. This information is useful in establishing a relation between decisions and creates something to refer at, when discussing a decision. The version helps creating a timeline of the evolution of the decision.

R1.1 The topic of the decision must be shown at the top of the view.
R1.2 The topic field should be optionally deletable.

R1.3 The author of the sheet must be shown at the top of the view.
R1.4 The author field should be optionally deletable.
R1.5 The date of creation of the decision must be shown at the top of the view.
R1.6 The date of creation field should be optionally deletable.
R1.7 The version number of the decision must be shown at the top of the view.
R1.8 The version number must be optionally deletable.

R2 - Description of a decision

One of the important parts of a decision is the description, which is a brief problem statement that conveys the context of a decision. This can possibly be done with flowcharts of the implementation of this decision. Therefore, there must be a component in which the user can provide this information. No explicit functionality is added to the component, since the user must be completely free to decide how to provide the description.

R3 - Alternatives

In the decision-making process, architects usually consider multiple alternative solutions that could solve the problem. Eventually, one of these solutions is chosen. Documenting not only the chosen solution but also the considered solutions is important, because it provides important rationale that is particularly helpful during maintenance and evolution, e.g., whether a certain solution was considered before or whether we need to investigate it. It can also be useful to know why a solution was not selected and whether the same reasons still apply. This way the user can always see which alternatives have already been discussed and which might still provide a new approach.

R3.1 The alternatives section must be deletable at all times.
R3.2 It must be possible to add the alternatives section at all times.
R3.3 The alternatives section should be available by default on the template.
R3.4 The must only be one instance of the alternatives list at a time.
R3.5 All updates to the list of alternatives must update the forces table (R4).
R3.6 Alternatives must be editable using a context menu.
R3.7 The state of an alternative must changeable using a context menu.
R3.8 Attempting to add more than three alternatives should prompt the user with a warning.

R4 - Forces

When making a decision, there are multiple concerns/forces, such as performance or ease-of-use, influencing the decision. Therefore, a forces overview is provided on the sheet, allowing the user to quickly enter or see the benefits of certain decisions in relation to the alternatives.

R4.1 There must only be one instance of the forces table at a time.
R4.2 Every row in the forces table must contain a name, a description and a value for each alternative currently present in the alternatives view.
R4.3 The user must be able to change a value at any time.
R4.4 Changing a value may automatically update the total.

R4.5 Changing a value should automatically update the color of the cell in question and its total cell.
R4.6 Columns must be ordered according to the order of the alternatives.

R5 - Related Documents

No decision is taken in a vacuum, which means that there are often documents or webpages related to the decision, containing information or argumentation. These files and links are therefore also provided on the page in order to easily view and access this information.

R5.1 The list operations described in R0 should work for files as well as links.
R5.2 It must only be possible to enter valid hyperlinks.
R5.3 A link-document also contains an url-component displaying the exact url that was entered.
R5.4 It must only be possible to add existing files to the view.
R5.5 Both document types should be accompanied by a name when added to the view.
R5.6 All properties of a document on the sheet must be editable.
R5.7 Deletion of a document is an atomic operation, except when an url-component is deleted on its own.

R6 - Stakeholders

In the end, knowledge also remains in the heads of the people involved with a decision. It is generally not possible or viable to document all knowledge, which means that it is very useful to personalize knowledge relevant to a decision. This way, knowledge and rationale can easily be traced back to a person.

R6.1 A stakeholder, when added, consists of a name and a role. The role of the stakeholder describes how the stakeholder was involved in the decision-making process.

R7 - Planning

Since `rationally` documents can be used in meetings and for capturing the architecting process of a system, maintaining a planning of things to do and things that have been done is a useful addition to `rationally`.

R7.1 A planning item, when added, consists of a text describing the planning item and a checkbox.
R7.2 The checkbox can be clicked, toggling the state of the planning item (*finished* or *unfinished*).
R7.3 A planning item that is in the *finished* state is struck through.
R7.4 A planning item that is in the unfinished state is not struck through.

R8 - Wizard

The information of the decision sheet should be manipulatable is several ways. One of which, is a wizard. It offers a way of entering text that might be preferable over the way that Visio offers when editing the text of shapes.

R8.1 The wizard should be a separate window.
R8.2 The wizard can be closed at all times.
R8.3 The wizard opens when the user creates a *rationally* document.
R8.4 The wizard can be reopened via a button in the *rationally* ribbon.
R8.5 Only one instance of the wizard can be open at a time.
R8.6 The wizard has a button that, when clicked, updates the decision sheet according to the data present in the wizard.
R8.7 The button described in R8.6 should say “create decision” when the user has just created a *rationally* document. Otherwise, it should be “update decision”.
R8.8 The wizard has a navigation menu with an item for all enhanced containers.
R8.9 Every enhanced container is represented by a page in the wizard, that can be accessed by the corresponding item in the navigation menu.
R8.10 Every page should offer the list operations described in R0, <u>except for reordering.</u>
R8.11 A navigation item should be present in the menu for the general information about a decision.
R8.12 A page should be present in the wizard for the general information about a decision, with fields for its title, its author, its creation date and its version.
R8.13 Every page should validate its content when the user clicks the button from R8.6. The validation should be similar to the one that happens during decision sheet manipulation via context menus of drag-n-drop.

R9 - Drag-n-drop functionality

The information of the decision sheet should be manipulatable is several ways. One of which, is drag-n-drop. This way allows *rationally* users to drag a shape from the *rationally* stencil on the view. This is particularly useful for experienced Visio users, since it is Visio’s conventional way of adding Shapes to the sheet.

R9.1 All enhanced containers can be filled using drag-n-drop shapes.
R9.2 The drag-n-drop shapes can be dropped anywhere on the sheet and are automatically placed in the correct enhanced container.
R9.3 All enhanced containers are included in the stencil of *rationally*.
R9.4 All atomic sub-elements of enhanced containers are included in the stencil of *rationally*.

R10 - Undo/Redo behavior

Visio offers its users undo and redo operations. These operations are available on all actions that a user can perform in Visio. This includes adding and removing shapes, styling shapes, modifying shapesheet properties of a shape and moving shapes. *rationally* should offer similar functionality.

R10.1 All *rationally* operations that appear atomic for the user should be undoable on their own.
R10.2 All *rationally* operations that appear atomic for the user should be redoable on their own.

R10.3 `rationally` undo operations should not interfere with unrelated other Visio operations performed by the user.
R10.4 `rationally` redo operations should not interfere with unrelated other Visio operations performed by the user.
R10.5 Every Visio undo/redo should leave `rationally` in a consistent state.

4.5 Non-functional requirements

Performance
Performance is an important requirement for our product, since it's goal is make the process of documenting decisions easier, which means faster. Below, we list the performance goals of various operations in `rationally`.

Operation	Avg time (ms)
Add an alternative	1000
Remove an alternative	500
Add a force	1000
Remove a force	500
Create a decision with title and general information	5000

Metrics

Besides performance, there are several non-functional requirements for `rationally` that can be measured using metrics. We will list the metrics to be measured with their possible values and an interpretation of those values. [3] We choose metrics that are part of the Microsoft Visual Studio 2015 Metrics Tool, since `rationally` was developed using Visual Studio.

Maintainability Index
Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code.[3] The scale used to rate this metric is logarithmic.

Score	Interpretation
0-9	low maintainability
10-19	moderate maintainability
20-100	good maintainability

We aim for `rationally` to score between **20-100**.

Cyclomatic Complexity
Measures the structural complexity of the code. It is created by calculating the number of different code paths in the flow of the program. [3]

For this metric, we aim for a value between **2-15** per method/module. Microsoft does not offer an interpretation of this metric besides stating that lower values are better than higher values. However, research has shown that a value lower than 10 or even lower than 15 yield successful systems.[4] Note that this metric has no maximum value, because one can theoretically reference an infinite amount of types.

Class Coupling
Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration.

Research has shown that class coupling can be used to predict software failure and is thus an important metric to take into account.[5] We aim for a score lower than **10** for our classes in `rationally`.[5]

4.6 Evolution Requirements

There are multiple factors that could lead to new requirements or can affect how current requirements are met.
The first factor is a change in the environment in which `rationally` is developed and deployed. New versions of Windows, C#, Microsoft Visio or Visual Studio can affect the ease of application development and may include improvements that make it easier to satisfy requirements, for example performance optimizations.
Another factor would be the development of a server, to store `rationally` decisions and easily relate them to each other. This would lead to new requirements, such as being able to synchronize documents, serialize data efficiently and being able to select `rationally` as related documents.

5 Software Architecture

5.1 Assumptions

1. It is assumed that architects rarely discuss more than three alternatives in detail. Therefore, `rationally` supports only up to three alternatives. The layout might break if more alternatives are added.

5.2 Architectural Views

5.2.1 Logical View

5.2.1.1 Package Overview

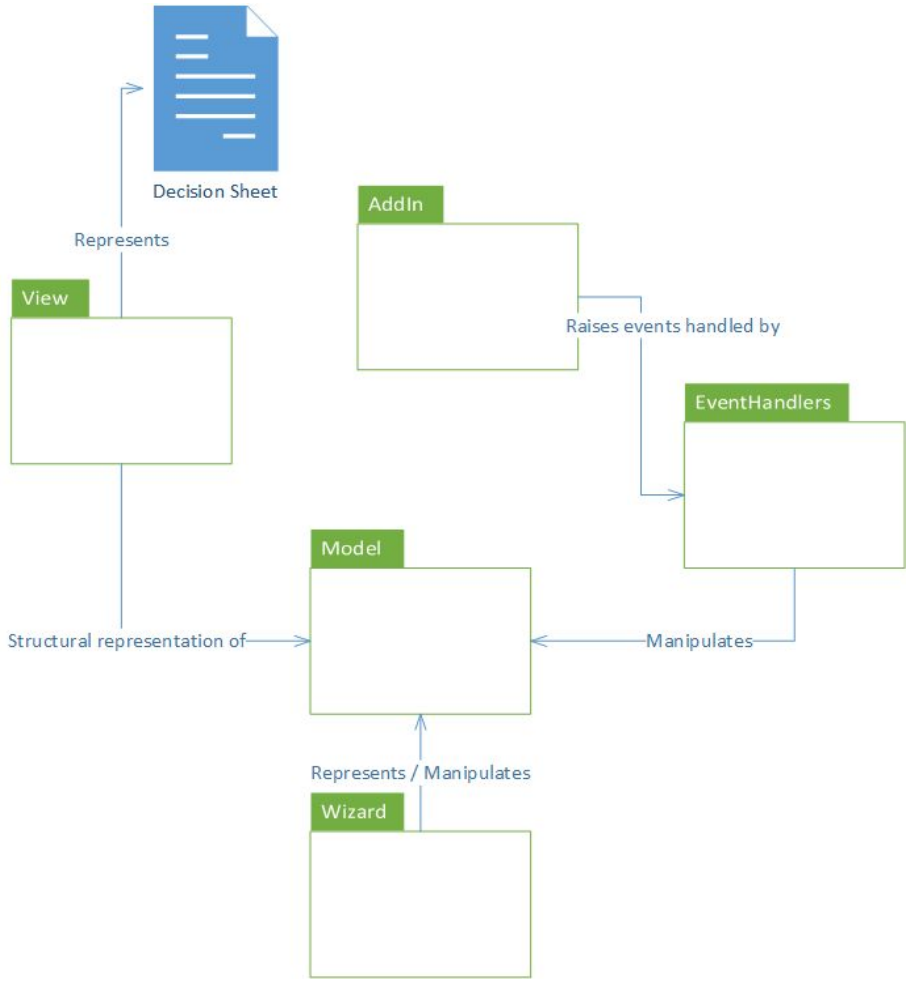


Figure 2: A high-level overview of relations between packages within `rationally`.

`rationally` consists of various packages that interact with each other. The AddIn package contains the main class of `rationally`. It registers event handlers via the Visio API and maps custom event handlers to them, that are defined in the *EventHandlers* package. The classes in this package are responsible for handling events for specific shapes.

The event handlers make changes to the model, encapsulated in the *Model* package. The model represents the data present on the decision sheet. All this sheet-data is spread out over various shapes on the decision sheet, that are embedded into other shapes. The relations between model data and shapes are maintained in the *View* package. The classes in this package are used in maintaining a tree that captures the composition structure of

shapes on the decision sheet (Visio's perspective of the data) and connects it to our model, which is an interpretation of the decision's sheet state.

Finally, there is package containing all classes relevant to the wizard. This package interacts with the model package in two ways. It retrieves data from it to show a correct state of the decision sheet to the user when showing the wizard and it manipulates the model according to the changes that the user is requesting in the wizard.

5.2.1.2 Class Diagrams

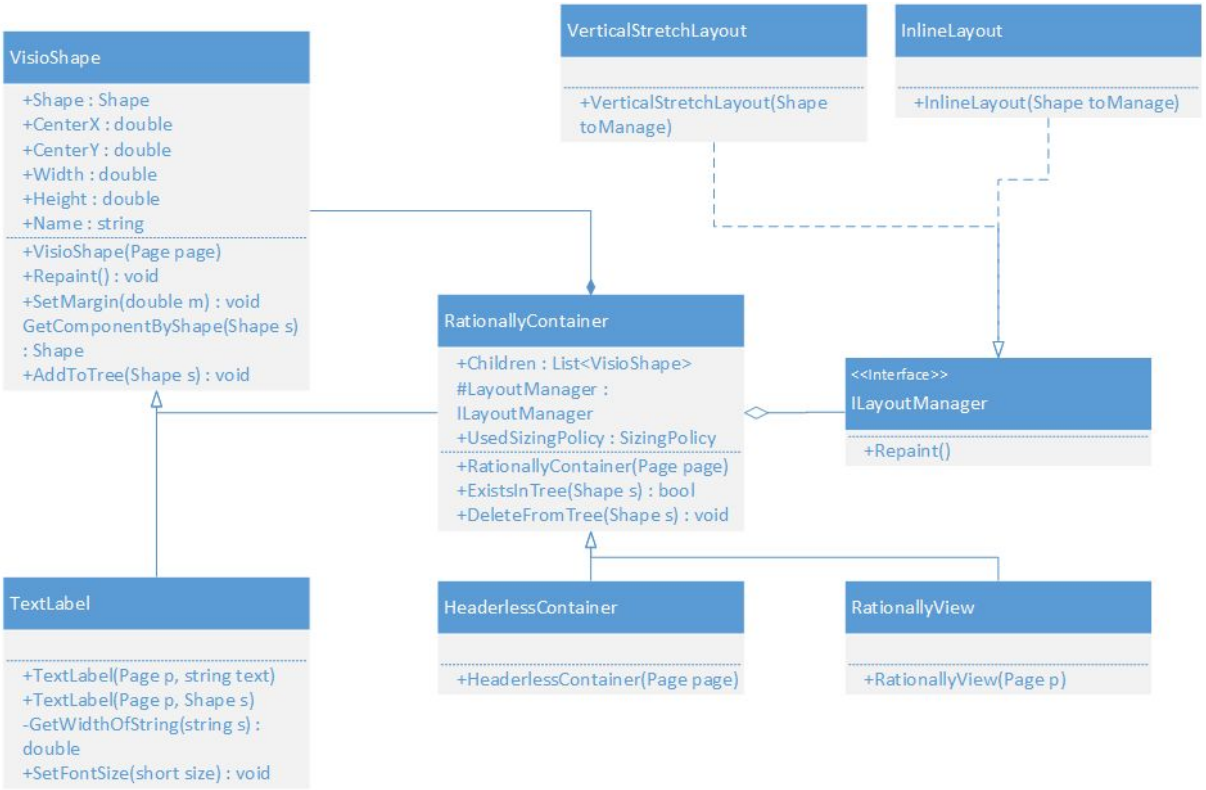


Figure 3: Class diagram of important components of the view package.

In this first class diagram, the lowest level of view classes are shown. VisioShape is a wrapper classes for the *Shape* class that the Visio API offers. This class may not be instantiated or inherited from, so we created a wrapper class for it. *RationallyContainer* forms a composite pattern together with *VisioShape*. All `rationally` Shapes inherit (indirectly) from one of these classes. Containers also have a *LayoutManager* property. This is an object that is responsible for the layout of the content of a (container)shape and is typically used during a *Repaint* procedure. *TextLabel* forms a base class for shapes that only contain text. Visio does not offer such a shape itself, so functionality like automatic size-scaling according to text size had to be implemented here manually. *RationallyView* forms the root node of the view tree that is maintained in `rationally`.

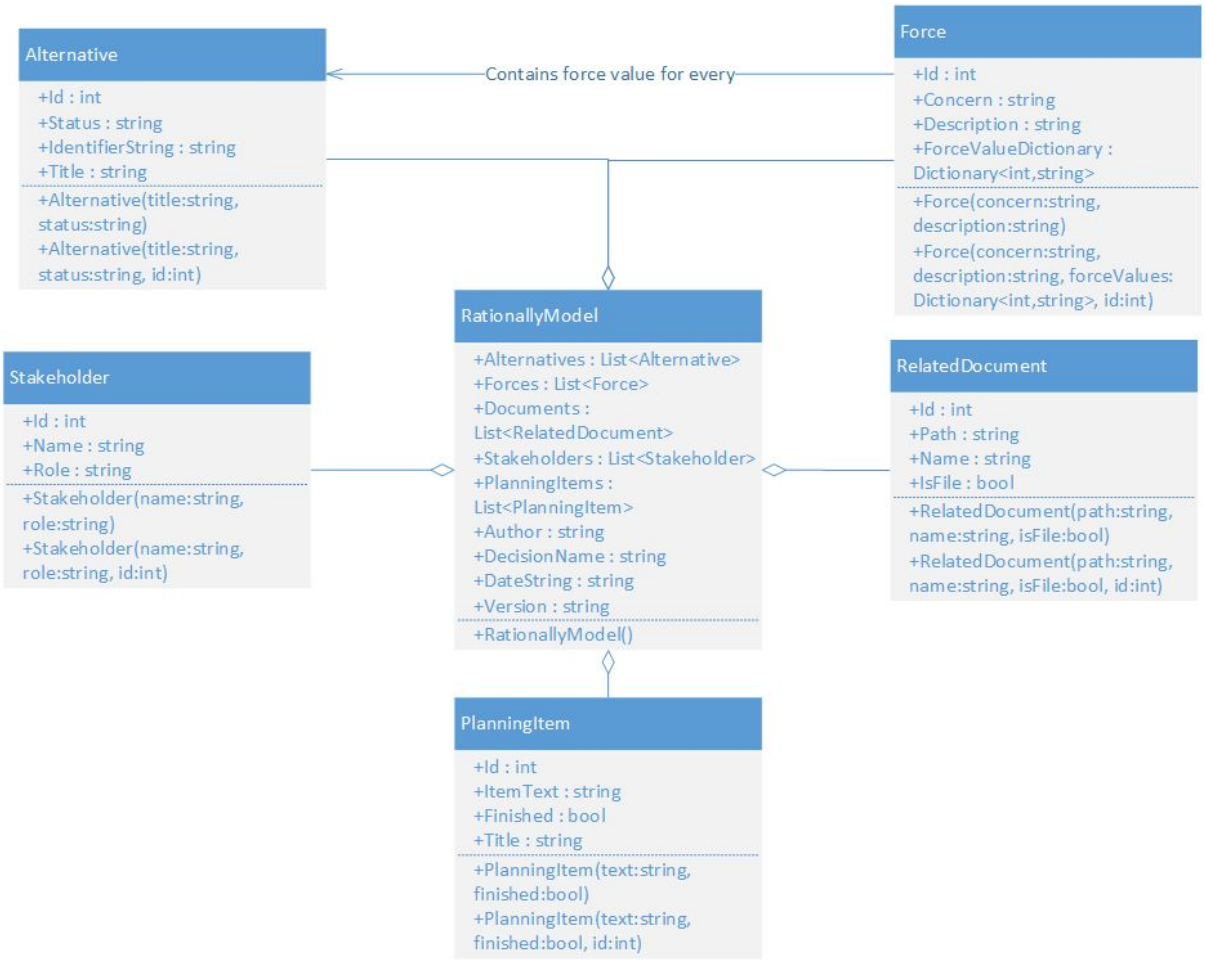


Figure 4: overview of the classes in the model of `rationally`.

The model of `rationally` contains the general information that is shown on the top right of the decision sheet and the data present in all enhanced containers on the decision sheet. Alternatives and Forces have a special relationship, since every alternative is represented in each force by a force value, which contains a score of an alternative on a force.

One of the challenges that comes with `rationally` is synchronizing this model with what is shown on the decision sheet (stored in shapes, behind the Visio API). In a typical MVC structured application, the view adapts according to what the model contains. In `rationally`, the view itself can be modified by the user, which requires the model to change according to the view. In the future, it might be possible to combine model and view by making the view tree components responsible for managing their own data. An example of that can be found in the following class diagram.

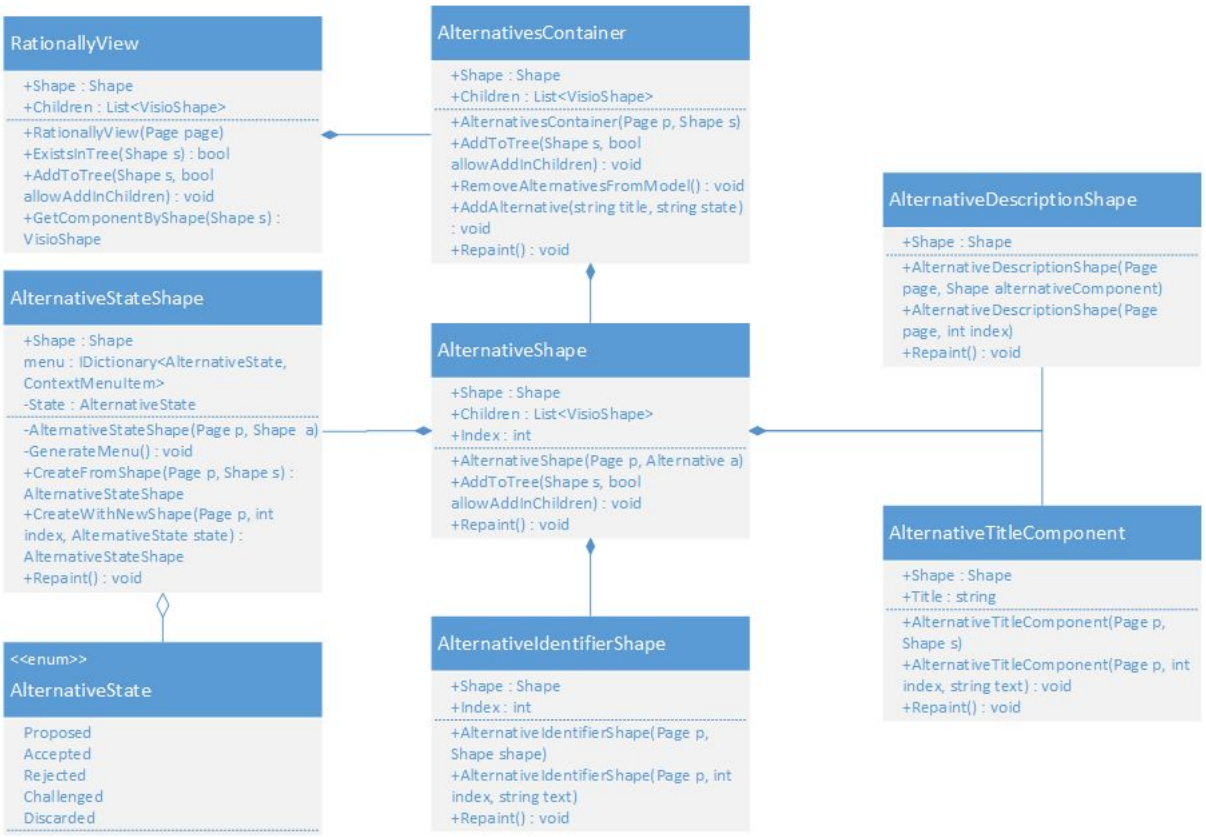


Figure 5: Overview of the classes representing alternatives on the decision sheet.

In Figure 5, most classes that are responsible for representing alternatives on the decision sheet are shown. Their base classes have been left out for compactness. Instead of including the base classes of the shown classes in Figure 5, we included some of their properties that are relevant in the context of this diagram, namely the *Shape* property and the *Repaint* method.

The *AlternativesContainer* is one of `rationally`'s enhanced containers. It contains *AlternativeShape* objects in its *Children* list property. *AlternativeShape* on its turn, is a container itself, containing one of each child types in its *Children* property.

The *AlternativeStateShape* is different from the other alternative child shapes. Instead of (public) constructors, this class uses factory methods. This is to avoid invoking functionality in the constructor before the object has been fully initialized. As hinted to in the previous section, this class is the first one to be responsible for managing its own data. This is done in the *State* property. It hides mapping from and to string values and is responsible for updating its own context menu. Ideally, all view classes will be refactored to this structure, making the model obsolete, which would remove complexity such as the described synchronization in the previous section from the project.

All enhanced containers of `rationally` are designed in a similar structure such as the one described in the class diagram above: an enhanced container class that is a direct child of *RationallyView*, a class representing one list item of the enhanced container and child classes representing components of the list item.

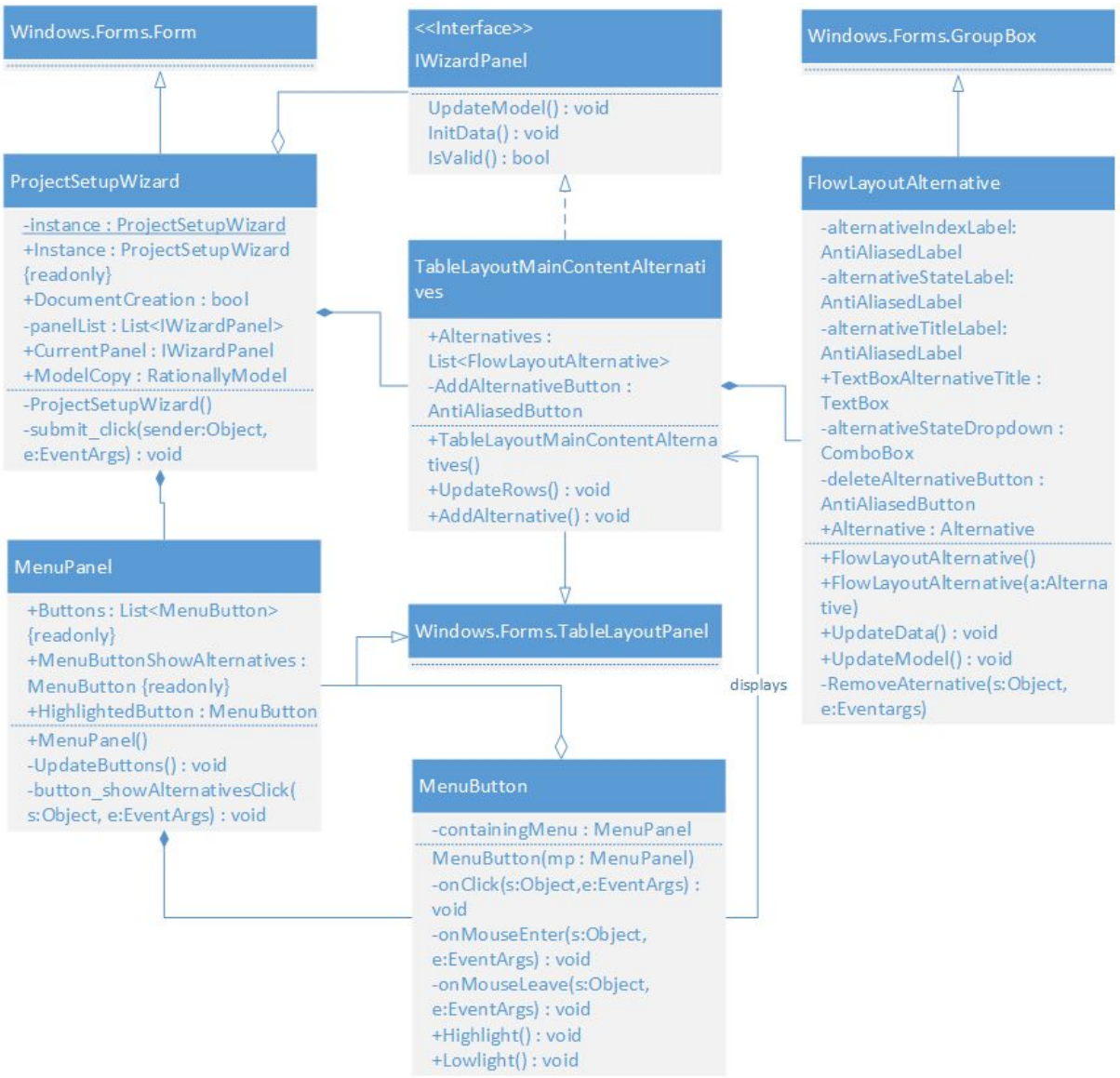


Figure 6: Class diagram of the of the part of the wizard dealing with alternatives.

This final class diagram shows the required classes for managing the content of an enhanced container in the wizard. *ProjectSetupWizard* represents the whole wizard. The menu is managed by *MenuPanel,* containing a *MenuButton* for each enhanced container. A content area connected to each menu item is implemented in a class implementing the *IWizardPanel*. This interface defines some mandatory operations for content panels of the wizard. One content area lets the user manage the content of one enhanced container. In this class diagram, we took the alternatives as an example.

TableLayoutMainContentAlternatives is such a content panel. Just like *AlternativesContainer*, it offers a list of items (represented by *FlowLayoutAlternative*) to the user and some list operations.
The synchronization between wizard and model takes place via *IWizardPanel* interface, which allows the wizard to extract data from a content panel and place it into the model.

5.2.1.3 Sequence Diagrams

There a few objects that often occur in the following sequence diagrams. We will give a brief explanation of them here.

Application
The Application object is the technical representation of Visio, for add-ins. The API available for developers is implemented foremostly in this object. We treat is as a black-box for all actions that the user performs in the visio sheet. For example, when a user deletes a shape from the sheet, we treat this as an action on the Application object. This not only simplifies our sequence diagrams, but is also makes sense since the Application object raises the events that we listen to.

Registry
For most events, `rationally` maintains a registry of event handlers. Each handler is responsible for dealing with the same event, but with a different origin or action. A markerevent can be raised by dozens of different shapes and can be of various types: “add”, “change”, “delete” etc. The logic for deciding which handler to chose is left out of sequence diagrams in this document to improve readability.

Handler
A Handler object in a sequence diagram in this document is a certain eventhandler that is subscribed to a specific event in a registry as described above. We choose to not always include the full name of the event handler to keep the diagrams compact in size.

Wizard
The wizard object represents the wizard and its components. In `rationally`, it is not just one object, but a composition of several types of classes that all interact with each other to maintain a model of the data in the wizard. In most sequence diagrams, this complexity is hidden to focus more on the flow of events happening outside the wizard.

SD0 - Creating a decision with general information

The sequence diagram for this use case has been split out into three parts. The first describes what happens when a user opens Visio and creates a `rationally` decision sheet. The second describes the key initialization operation of showing the wizard. The third shows what happens after the user has entered some general information in the wizard and want to create a decision sheet with that information.

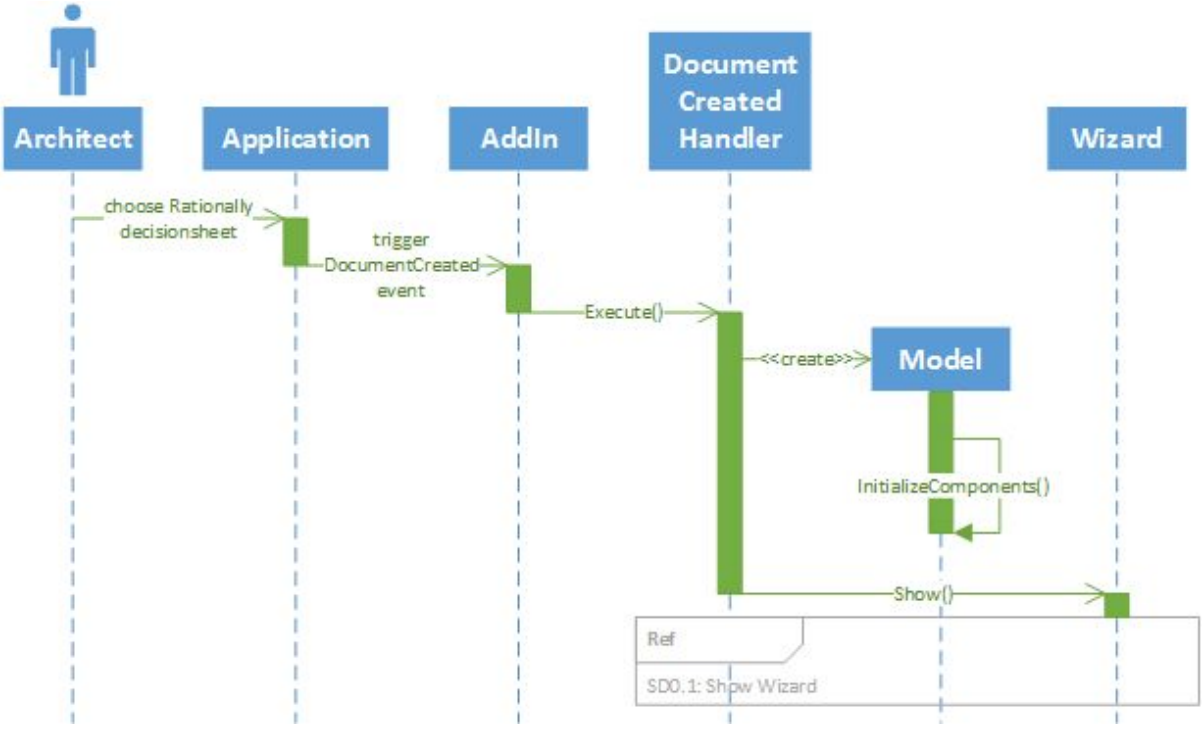


Figure 7: sequence diagram for creating a decision sheet.

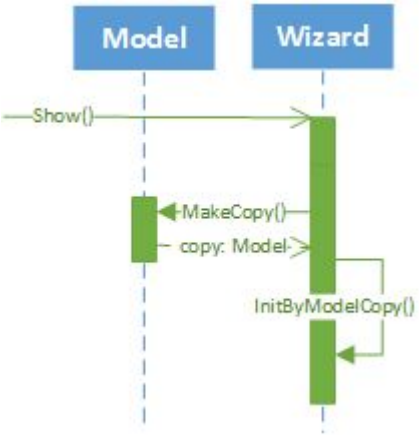


Figure 8: sequence diagram of showing the wizard.

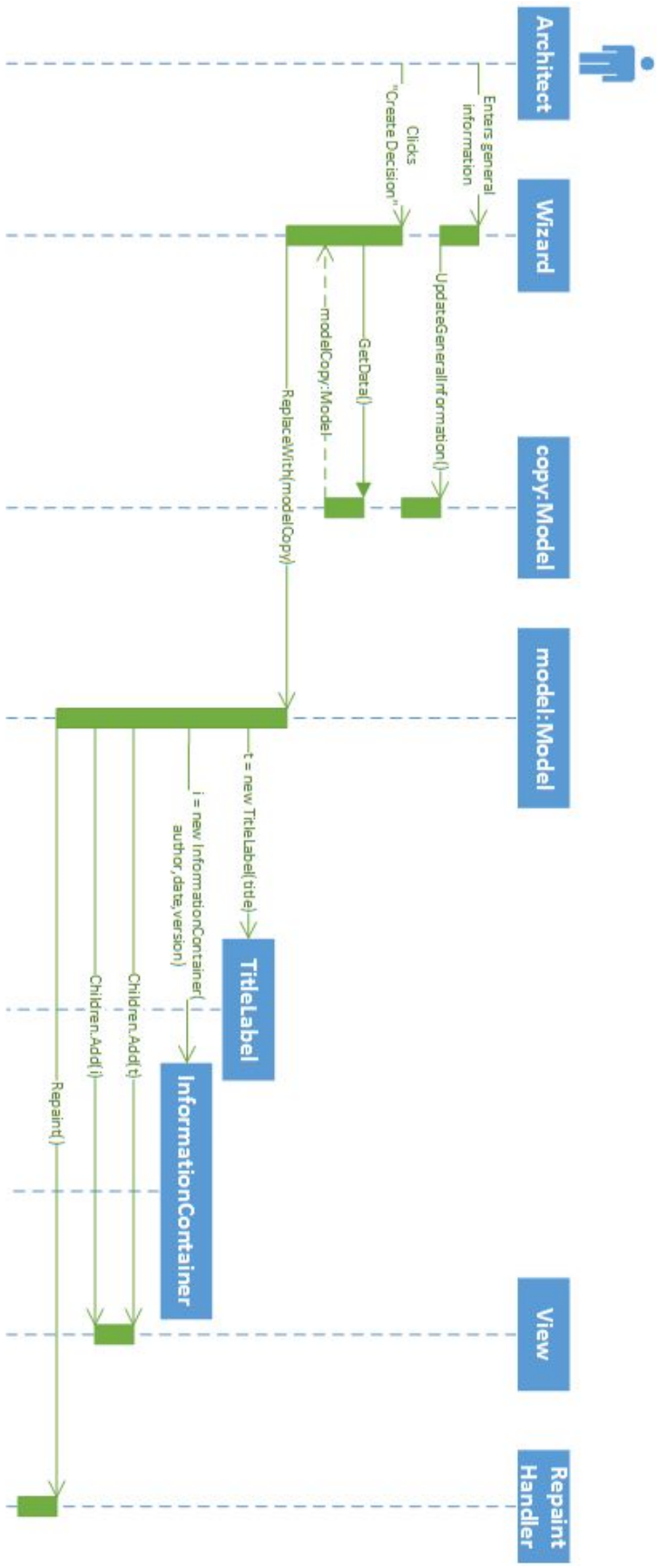


Figure 9: Sequence diagram showing the flow following entering general information in the wizard.

What becomes clear from the first sequence diagram is that *rationally* interacts with Visio's API via events. *rationally* subscribes handlers to events and Visio invokes these handlers when the events occur. In this case, when an architect selects the *rationally* template, a DocumentCreatedEvent is raised. Our handler for this event creates an empty Model object and shows the wizard to the user.

The wizard's purpose is to represent the decision sheet and allow the user to modify its content in a faster way than Visio's traditional way. When showing the wizard, we make a copy of the Model (which holds the data of the view) and lay out the data of that copy in the wizard. All changes that are made in the wizard (see SD0.2, entering general information) are made in the model copy as well. When the user is done, the original model is replaced by the updated model copy and the view is repainted according to it.

SD1 - Changing the state of an alternative

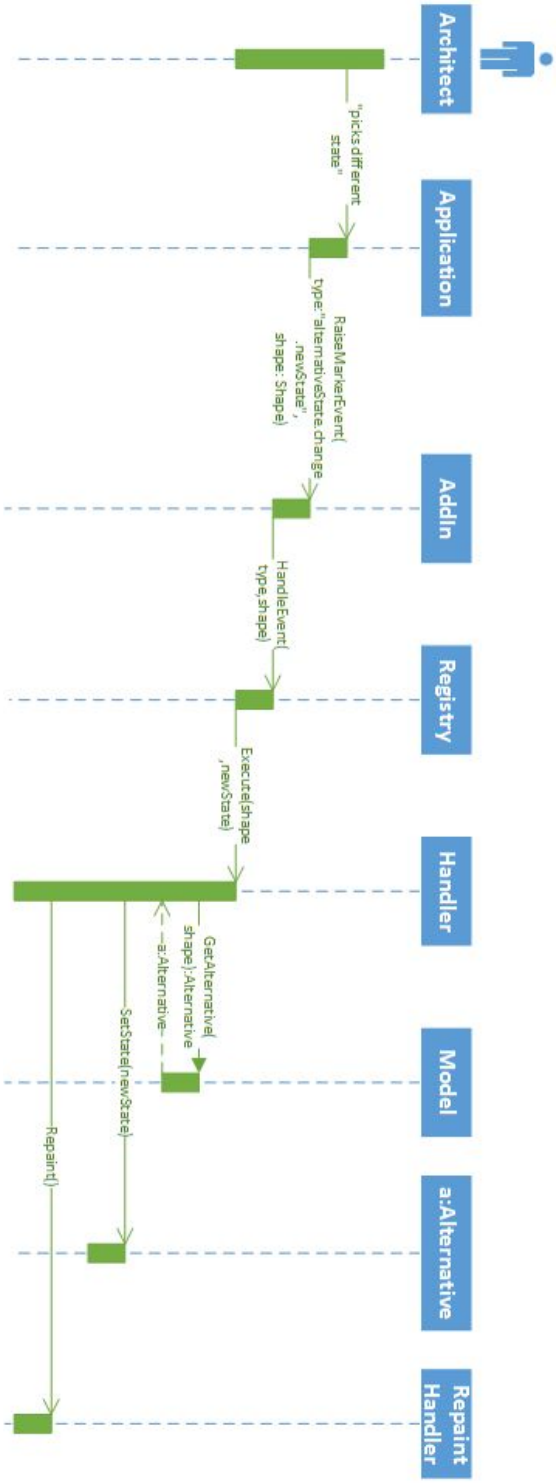


Figure 10: Sequence diagram for changing an alternative's state.

The architect selects one of the available states via the context menu offered by a state-component. A MarkerEvent is raised by Visio and handled by our AddIn object. The appropriate handler locates the relevant alternative (for example, by finding the shape in the view tree, and see what alternative is connected to the alternative container shape), and

changing the state of it. Now that the model is updated, a repaint can be performed to bring the view up-to-date with the model. Note that only the alternatives area needs to be repainted.

SD2 - Creating an alternative

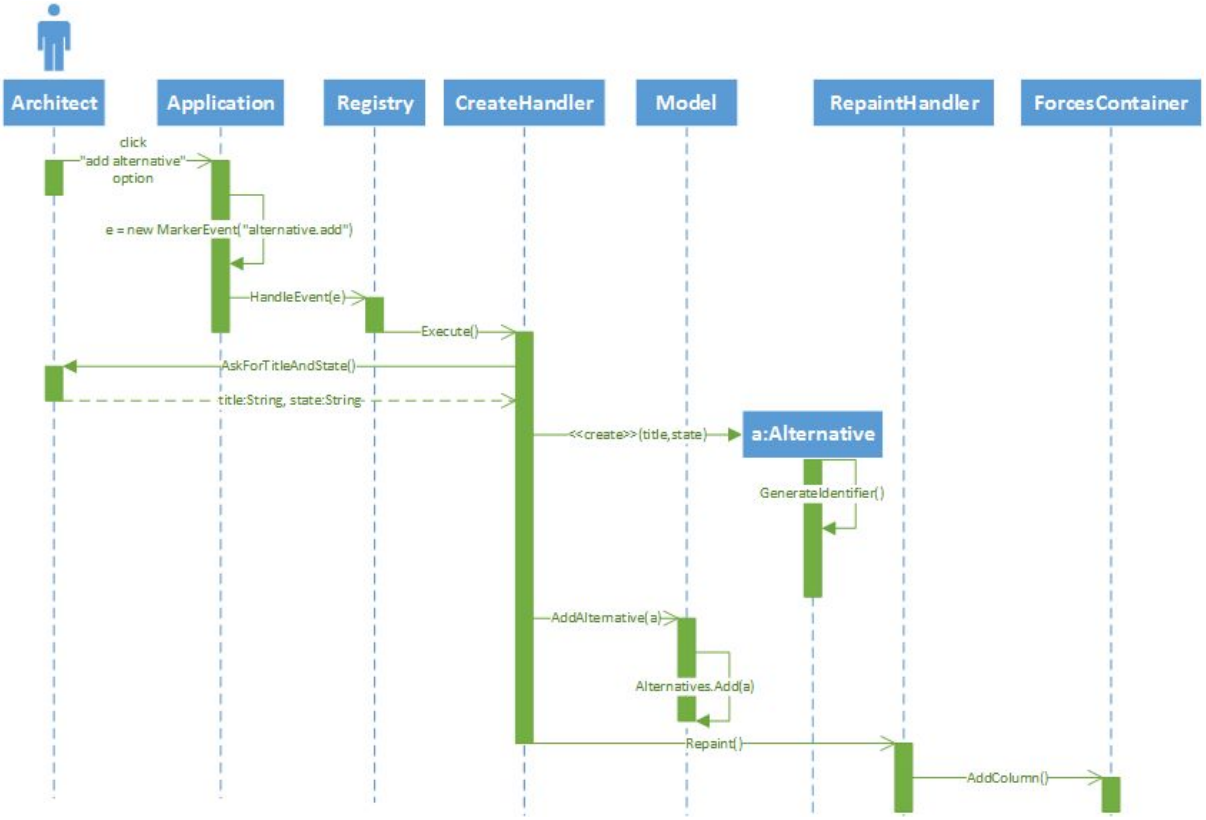


Figure 11: Sequence diagram for adding an alternative.

In figure 11, a simplified version of the process of creating an alternative is displayed. Note that the especially the *Application* related messages are simplified and moreover not part of `rationally`, but rather part of Visio itself. In this flow, the user is asked to chose a title and state for the alternative and these are used in the creation of an *Alternative* object (*a*), which is then added to the model. Note that forces are dependent on alternatives, because each alternative has a column in a force row. Therefore, the repaint operation should add a column to the forces table when an alternative is created.

5.2.2 Decision Detail Views

In this section, we will describe what decisions were made during the process of developing `rationally`. We will do so according to the decision detail viewpoint as described by Van Heesch et al. [7]. We give a description of the decisions and the alternatives that were considered.

Software design decisions

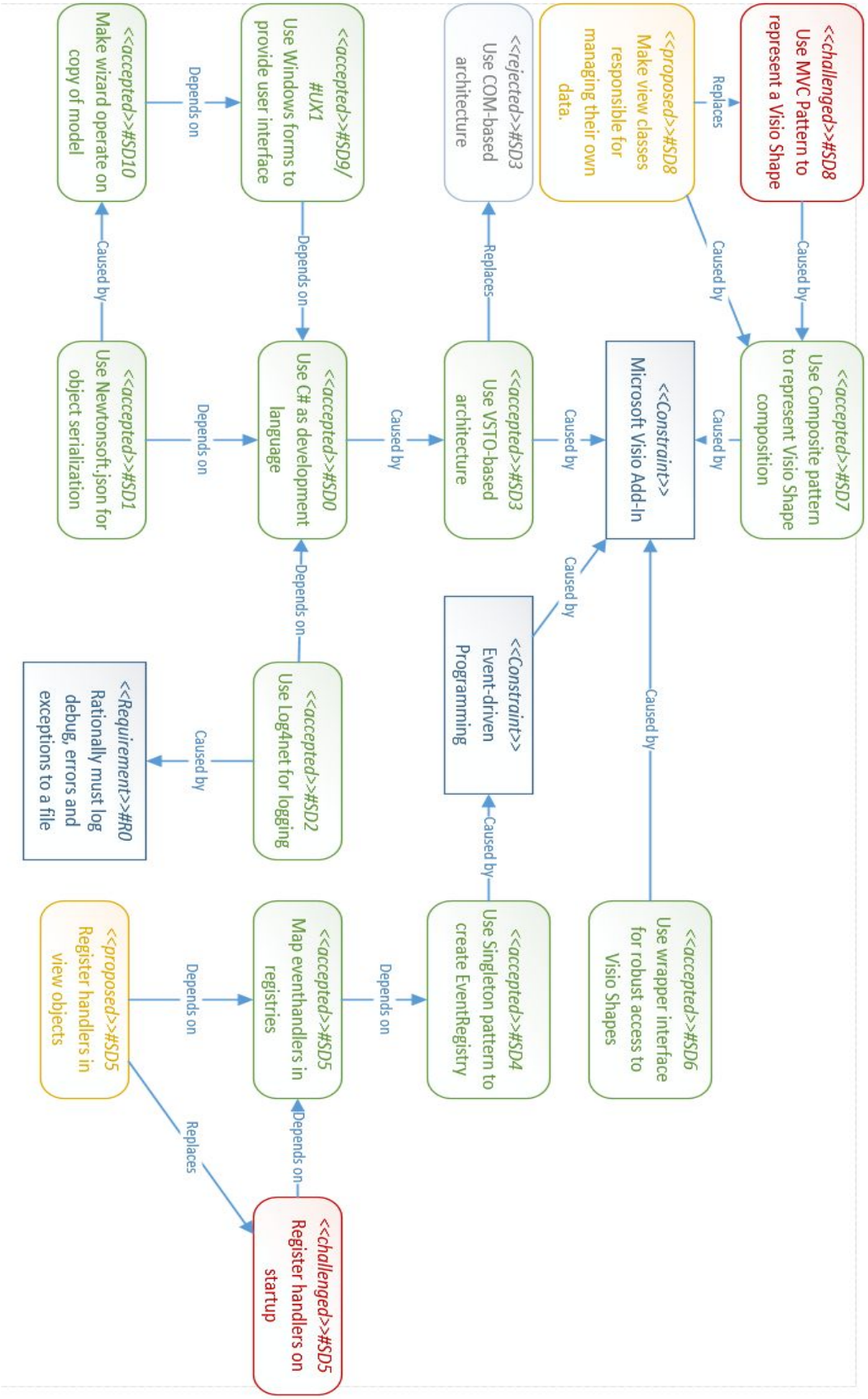


Figure 12: Decision relationship view for software design decisions.

Name	SD0 - Choosing a development language
Description: In order to develop our add-in, we need to choose a language in which we will be developing the add-in.	
Considered Alternatives: **C#:** In order to write an add-in for Microsoft Visio, C# is the only up-to-date language that allows us to develop the add-in.	

Name	SD1 - Serializing objects
Related Requirements: R8.6	
Description: In order to communicate with a server or in order to make a deep copy of an object, we need a way to serialize objects.	
Considered Alternatives: **Newtonsoft Json.NET:** Json.Net is easy to install using the NuGet manager. It allows for easy serialization using just a few methods. It is the most commonly used package to serialize objects.	

Name	SD2 - Logging actions and events
Description: To keep track of what the application is doing, we need a way to log actions, events and debug statements.	
Considered Alternatives: **Log4Net:** Log4Net is very easy to setup and allows us to start logging in just a few statements. It supports good exception logging tools, which make it easy to trace errors. **NLog:** NLog is also very easy to setup, again allowing us to start logging in a few statements. It is updated quite often and therefore bugs are quickly fixed.	
Decision: Both options are easy to use and satisfy our requirements. We have previous experience with Log4Net and therefore decided to use Log4Net.	

Name	SD3 - Choosing an add-in architecture
Description: We need to decide upon an architecture using which we will develop our add-in.	
Considered Alternatives: **COM-based:** COM-based add-ins are low-level add-in that provide a small layer to interact with visio properties. **VSTO-based:** VSTO-based add-ins also make use of COM architecture, but in addition, provide functionality which makes it easy to design more complex visio components, such as the ribbon.	
Decision: VSTO-based architecture offers a lot more functionality and is therefore the way to go.	

Name	SD4 - Handling events
Description: Visio functionality forces us to constrain ourself to event-driven programming, so we need a way to handle events.	
Considered Alternatives: **Event Registry:** Create a registry that stores event handlers and can easily map an event to the right handler. This handler is Singleton, to make it accessible from everywhere in the application.	

Name	SD5 - Registering event handlers
Description: In order to utilise our event registry, we need a way to register event handlers.	
Considered Alternatives: **Register handler on object initialisation:** We can register a new eventhandler when an object is initialized. This eventhandler is then coupled to the right object, which saves us from finding the relevant objects. **Register handlers on application start:** It is possible to register all event handlers at the start of the application and have the handlers be responsible for interacting with the right objects. This approach is easy to implement, but is state-sensitive.	
Decision: At first we initialised the event handlers on the start of the application, as the consequences and complications at that point were yet unclear. However, since that approach does not benefit the maintainability of the application, it is now a challenged approach and registering on object initialisation is now the proposed solution.	

Name	SD6 - Interacting with the Visio API
Description: The Visio ShapeSheet treats every shape in the same way. Shapes with no actions offer the same interface as Shapes with many actions. The same is true for containers, that offer the same interface as non-container shapes. Not only do all shapes offer the same interface, all rows in their ShapeSheet hide their internal unit. This means that the API offers no type safety. The problem with this lack of type safety and uncertainty of the actual available interface, is that the developer needs a lot of additional logic (e.g. existence checks, type checks) to interact with the ShapeSheet.	
Considered Alternatives: **Direct interaction with the Visio API:** One possible way to go is to not do anything about the api and perform all relevant type and existence checks whenever an interaction occurs. This is the simplest way, but leads to a lot of unmaintainable and duplicate code. **Create a wrapper layer around the visio API:** We implement a wrapper around the API that is responsible for type and existence checking. It offers more type safety and similar constraints as the actual API by validating values and fields before updating the ShapeSheet.	
Decision: A wrapper layer is way more maintainable and easier to extend than direct interaction. It is also less error-prone than doing it by hand.	

Name	SD7 - Representing Visio shape composition
Description: In order to easily interact with shapes, we need a way to represent them in C#.	
Considered Alternatives: **Use a composite pattern to represent Visio shape composition:** Visio represents all shapes the same way, with certain properties specifying whether they are containers, or contained in a different shape. Using a composite pattern is a very close way to represent that structure in the C#.	

Name	SD8 - Representing Visio shapes
Description: We need a way to represent the properties of a Visio shape using C# classes.	
Considered Alternatives: **MVC pattern:** The classic separation of concerns. Separate model and view classes, with event handlers representing the controllers. This way is easy to implement, but not easy to maintain. **View classes are responsible for managing their own data:** Make view classes responsible for registering their own event handlers (#SD5) , which allows the class to also manipulate the relevant parts of the model easily. This is slightly harder to implement, but in time is really beneficial to the maintenance and extendibility of the application.	
Decision: At first we used to MVC pattern to represent Visio shapes. However, since that approach does not benefit the maintainability of the application, it is now a challenged approach and making the view classes responsible is now the proposed solution.	

Name	SD9 - Providing an user interface
Description: Related to (#UX0). We need a tool to provide an user interface to users, consisting of multiple windows and popups.	
Considered Alternatives: **Windows forms:** Windows forms is the default way of developing windows and forms for the user to use. It is very flexible, has a WYSIWYG editor and integrates well in Visio.	

Name	SD10 - Ensuring wizard operations are atomic
Related Requirements: R8.6	
Description: In order ensure the robustness of our wizard, we need to ensure that operations are atomic	
Considered Alternatives: **Operate on a copy of the model:** In order to ensure changes to the model caused by the wizard are atomic, we perform all operations on a deep copy of the model (#SD1), with which we replace the old model when wizard changes are submitted.	

Name	SD11 - Rebuilding the view tree
Description: When an existing Visio file is opened or a switch between two open documents occurs, a tree needs to be constructed representing the state of the decision sheet, with all its shapes. To do this, we depend on the interface that Visio provides for this. It offers the *Shapes* property, implemented in the *Page* object, that returns a flat list of all shapes (of type *Shape*) present on the sheet. We want a more realistic representation of the decision sheet, in which shapes can be embedded into other shapes, giving rise to a composite structure, that can be represented using the tree data structure. We want our solution to score high on several quality attributes: complexity, flexibility (support for different tree composition), extensibility (support for new components) and implementability.	
Considered Alternatives: **A0 - Do not sort the list** This is our current implementation. It requires very little complexity during the creation and modification of shapes, but it yields a lot of logic and switching during the rebuilding of the tree. **A1 - Sort the list using layer and order properties.** We assign every class a layer value, representing the layer of the tree that the class occurs. We also assign an order property to every shape, that defines the order of sibling shapes within one parent shape. This solution allows us to sort the list in a way this satisfies two properties: The parent of a shape is always placed before that shape in the shape-list. Sibling shapes are placed next to each other, in the correct order, in the shape-list.	

These properties yield some advantages. The first one is that we can only have to traverse the list once, because we don't have to search for the child shapes, for every shape. This means an improvement in time complexity of the problem: O(n^2) => O(n).

The other advantage is that we solve a problem that arises very frequently in the rebuild procedure, namely that we try to add a child shape to the three, while the parent shape is not yet placed in the tree. We now solve this by placing a stub shape in the tree and replacing it with the parent when we add that parent to the tree.

A2 - Sort the list using the shape's name property.
We currently have use a composite pattern (#SD7) to represent Visio shapes, in a way that a certain type of shape always occurs in the same level of the tree. That is, *RationallyView* is always the root node of the tree (layer 0) and an *AlternativeShape* always occurs in layer 2 (*RationallyView* > *AlternativesContainer* > *AlternativeShape*).

A3 - Sort the list using dynamic layer and order properties
We assign every shape a layer value, which is simply the layer value of it's parent plus the order property can also be defined on creation of a shape, by the parent shape.

A3 - Sort the list using a timestamp property and an order property
We assign every shape a timestamp on it's creation, or use an existing property of the shape interface that does this for us. Given that a parent shape will always be defined before it's child shapes, this gives a way of moving parents in front of children in the shape list. The order property is implemented similar to the other alternatives.

A5 - Do not sort, and rebuild direct children recursively
We start by locating the root node and initiate a rebuild procedure on it. The procedure will locate the direct children in the flat list and add these as its children (thereby building a part of the tree we desire to end up with). After that, we initiate the same procedure on our just found children.

Besides this, we still need an *order* property to sort the siblings.

Decision:
At this point, we are still using A0 as our approach to this problem. However, it is not sufficient with respect to flexibility and maintainability and so an alternative solution must be searched for.

Name	SD12 - Managing component layout

Description:
Visio shows a page,on which shapes can be positioned and customized, to the user. The shapes are also ordered in a composite-fashioned way. Visio offers various sets of simple shapes, that is, they are single-layer singular objects. Creating and maintaining a

hierarchical structure is left to the user, and so is the positioning of the shapes. Only aligning tools are offered.
Our project, on the other hand, offers a set of relatively complex multi-component multi-layered shapes. The problem that occurs is that we have to programmatically structure and position shapes. Because this is generally the responsibility of the user, the visio API does not offer any layout managing.

Considered Alternatives:

Make our components responsible for layout managing their content: We implement the layout managing in each component that requires it, and tailor that code to the specific component.

Create layout managers: We create generic layout managers that can manage the layout of any component. Each manager implements its own type of layout managing, for example by stacking all elements vertically, or placing them in a horizontal line. By using a composition of these generic managers, more complex layouts can be defined.

Decision:
Making the components responsible is a quick solution. Because every component is responsible for its own layout management, it allows for very tailored code. Using component-specific constants, a layout is easily created. However, the constants are magic numbers which are not flexible at all. The code tends to become complex with a lot of long mathematical additions and subtractions of constants and factors.This solution neither offers any reusability, and is not easy to extend.

On the other hand, the layout managers require quite some work to create. They should be incredibly flexible, making their creation relatively hard. Once they are created, however, they bring extensibility and usability, in the sense that you only need to define a certain manager, instead of writing the layout code for every component. Therefore we decide to go with the managers.

Name	SD13 - Handling Custom Styling of `rationally` Components

Description:
The `rationally` add-in generates (groups of) shapes for the user to make the process of decision documentation faster and more structured. Which shapes are displayed on the view to the user depends on the model that is maintained in the add-in. A model change leads to a repaint of the view, to make the view up to date.

A repaint of the model generally only considers the state of the model and nothing else. In Visio, it is possible for the user to style components, even the ones generated by the add-in. The problem that would occur is that the style added by the user to a component generated by `rationally` is discarded when a repaint takes place, because the representation of the model data (the styled shape) is not part of the model.

Considered Alternatives:

Disable styling on `rationally` components:
Components that are generated by the add-in can not be styled in any way. No styling can be lost this way.

Include the style of components in the mode:
Besides the data of model components, we store the styling of the component that represents that data. We would fire events at every styling action and update the model accordingly.

Maintain a tree of components generated by our add-in:
Parallel to the model, we maintain a state of the view (i.e. the page the user is working on) in which we store the components generated by `rationally`. This state maps model components to view components (shapes), so that model updates can modify a view component instead of overwrite it.

Decision:
Disabling styling has many advantages for the developers. It is flexible, easy to extend, and easy to implement. However, it completely disables all styling options on `rationally` components, that will make up a large part of the view. Styling is too important to disable on these components, so this solution is rejected.

The second alternative, including the styling, does not affect any functionality but requires a lot of work. If every styling option is to be supported on components, all of them have to be included in the model and should come with event firing and handling.

The third and final available alternative, maintaining a tree of components, also leaves all functionality intact. However, this solution is easier to implement, because only a structure containing the shapes is required instead of separate items for each styling property. The styling is encapsulated in the shape and all that is required is to reuse the shape on repainting. Because it offers the best functionality to the user and does not require excessive amounts of work, we decided to implement this solution.

User experience decisions

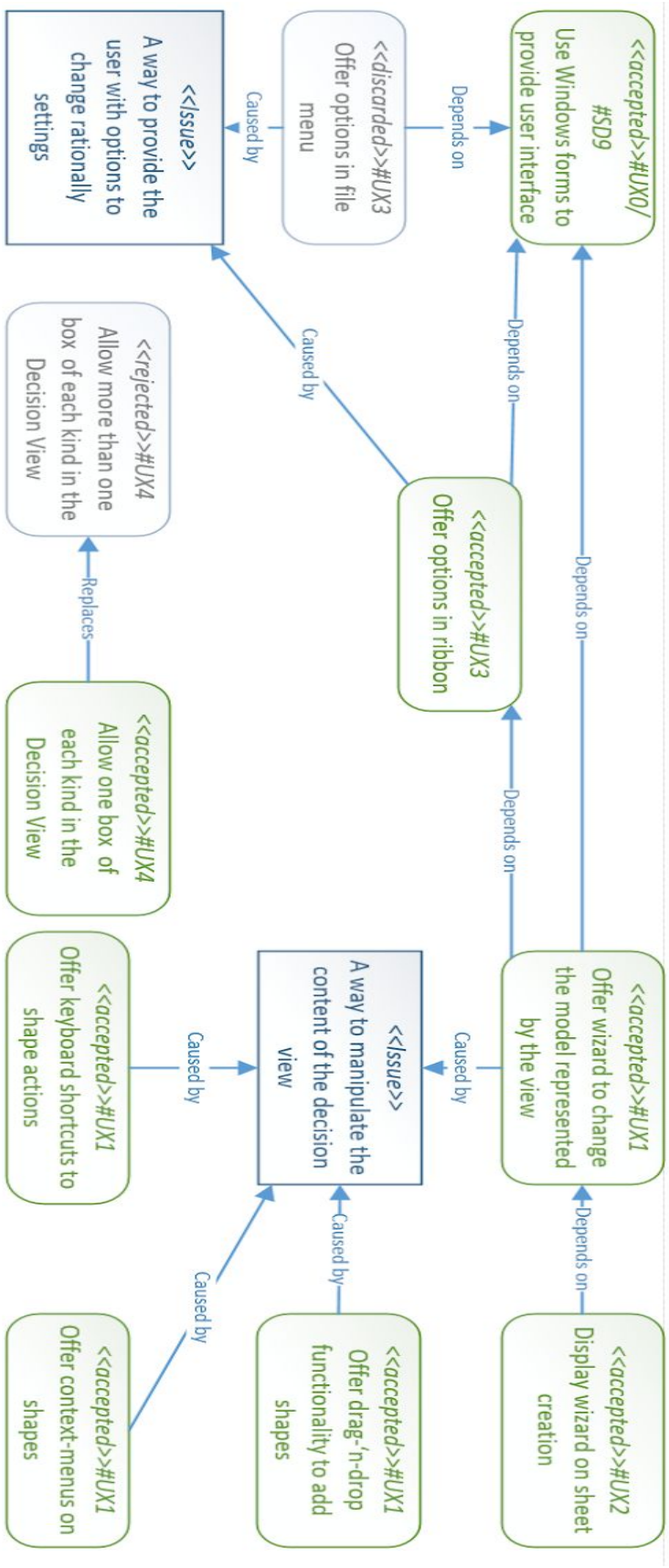


Figure 13: Decision relationship view for user experience decisions.

Name	UX0 - Providing an user interface
Related Requirements: R8	
Description: Related to (#SD9). We need a tool to provide a user interface to users, consisting of multiple windows and popups.	
Considered Alternatives: **Windows forms:** Windows forms is the default way of developing windows and forms for the user to use. It provides a clear interface, responsive functionality and many different options.	

Name	UX1 - Manipulating the content of the decision view
Related Requirements: R8, R9, R10	
Description: In order for the user to use our application, they need a convenient way to modify the data presented in the decision view.	
Considered Alternatives: **Wizard:** A wizard, as presented in (#SD9), provides a convenient and intuitive way to manage the content of the decision view. It allows for quick and simple changes to the content. **Context-menus:** Context-menus offer context sensitive operations to the users, such as reordering items in a list. They are easy to access using a simple right click and due to this are intuitive to use, especially since Visio by default uses context-menus a lot. **Drag-'n-drop:** A key Visio functionality is to easily add shapes by simply dragging them onto the sheet. This can easily be extended to `rationally` by providing the user with many enhanced shapes that can be dragged onto the sheet, adding them to the right containers if relevant. **Keyboard shortcuts:** Operations such as undo and redo are staple in any application. They are often utilised using keyboard shortcuts and thus support for them must be included.	
Decision: All four options provide a different but very useful functionality to the user. Therefore, they will all be implemented.	

Name	UX2 - Provide functionality on file creation
Related Requirements: R8	
Description: When a new decision view is created, it would be useful to provide the user with some suggestions as to how to use the add-in.	
Considered Alternatives: **Showing the wizard on creation:** The wizard provides a good flow for the user to set up their decision view. It would therefore be very useful for the user to be presented with the wizard when the view is created.	

Name	UX3 - Provide the user with access to the wizard and add-in options
Related Requirements: R8	
Description: At any point during use, the user may wish to access the wizard or change settings of the `rationally` add-in.	
Considered Alternatives: **Utilizing the file menu:** Visio related settings are also found in the file menu, which means that using this file menu is intuitive for the user. **Utilizing the ribbon:** The ribbon is immediately accessible from the Visio sheet. It allows us to present all `rationally` related settings together, without settings from other add-ins. Visual studio also provides a WYSIWYG editor for the ribbon.	
Decision: The ribbon provides more flexibility to the developers and more overview to the user.	

Name	UX4 - Handling multiple of the same containers
Related Requirements: R0.1, R0.7, R0.9	
Description: The user can add a second instance of a container to the sheet, such as the “Related Documents” container. We need a way to handle that event.	
Considered Alternatives: **Allowing a single instance of a container:** Automatically deleting a second container means that we can always work with the assumption that there is at most a single container. This means the code is easy to maintain and extend. **Allowing multiple instances of a container:** Allowing multiple containers means more flexibility for the user, but also makes the code highly complicated, negatively impacting the robustness of the system.	
Decision: The robustness and maintenance are more important than a slight increase in flexibility, in this case, so we only allow a single container.	

Development process decisions

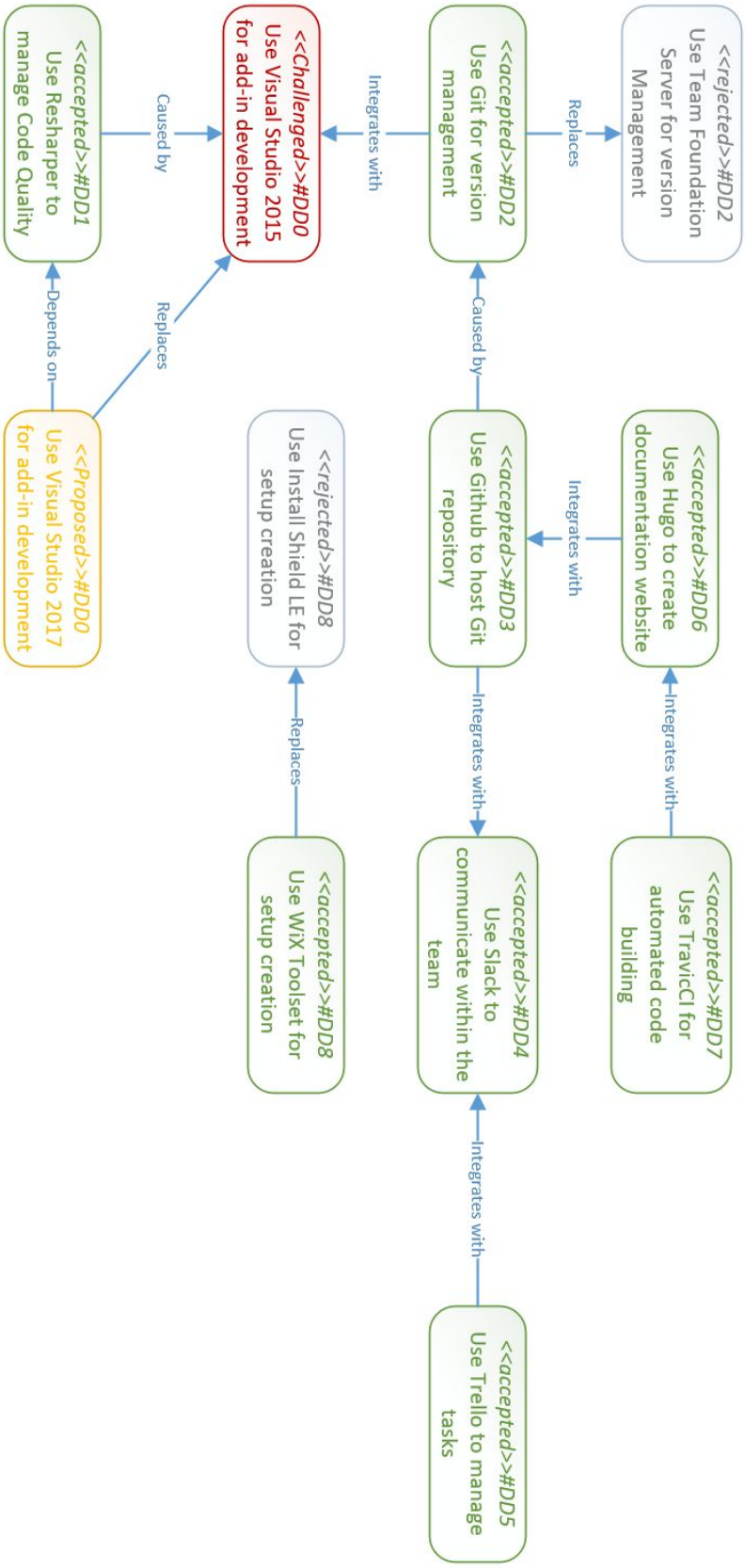


Figure 14: Decision relationship view for development design decisions.

Name	DD0 - Choosing an IDE
Description: In order to develop in C# (#SD0), we need an IDE with widespread support for the language. We want the IDE to have good debugging tools, integration with Microsoft Visio and integration with a version management tool. It also needs to have support for a tool to enforce code style.	
Considered Alternatives: **Visual Studio 2015:** Visual studio is the official IDE for C# so our choice for the product was quickly made. VS 2015 has integration with Git and Team Foundation Server, allows for live debugging of Visio add-ins and has the Resharper Add-in to enforce code style. (#DD1) **Visual Studio 2017:** Visual studio 2017 has improved code optimizations and is faster than 2015. It also has better support for C# 6.0. However, at the start of the project, it was not yet released. Porting the code to a newer version is therefore a risk and proper support for Resharper is not there yet.	
Decision: Since Visual Studio 2015 still satisfies our requirements and porting our code to Visual Studio 2017 is a risk, we decide to use and keep using Visual Studio 2015.	

Name	DD1 - Enforcing code style
Description: To be able to maintain our code over the years, a proper code style goes a long way. Since multiple developers work on the project, a tool to enforce this code style is required.	
Considered Alternatives: **Resharper:** Resharper is one of the most commonly used tools for maintaining code style and quality. It has good integration with Visual Studio (#DD0) and allows settings to be configured using Git or TFS.	

Name	DD2 - Version management
Description: In order to work on the project with a team of developers, we need a tool to manage code versions, that is capable of handling conflicts in changed code within a file and offers an online repository to store the code. It also needs to be able to integrate with Visual Studio.	
Considered Alternatives: **Git:** Git allows us to work in the same file simultaneously, has integration with Visual Studio and offers many different hosts for our repository. **Team Foundation Server:** TFS does not allow us to work in the same file and by doing this, ensures that conflicts can never occur. It is integrated by default in Visual Studio and a TFS server is easy to host.	
Decision: Since we often need to work in the same file, Git is the way to go.	

Name	DD3 - Hosting Git
Description: We need a place to host our Git repository.	
Considered Alternatives: **GitHub:** Github is one of the most used and most stable repository hosts around. It also allows us to document and track code issues.	

Name	DD4 - Communicate within the team
Description: We need a tool or application to communicate within the team and share files.	
Considered Alternatives: **Slack:** Slack is a commonly used communication application for development teams that does everything we need it to. It also integrates with other tools like Github (#DD3), Trello (#DD5) and TravisCI (#DD7).	

Name	DD5 - Manage tasks and deadlines
Description: We need a tool or application to manage deadlines, keep track of issues and track user stories.	
Considered Alternatives: **Trello:** From previous projects we have a lot of experience with using this tool. It also satisfies all requirements we have and integrates well with Slack (#DD4).	

Name	DD6 - Hosting documentation on a website
Description: We need a tool or application to easily manage and host the documentation for `rationally` on a website.	
Considered Alternatives: **Hugo:** Hugo integrates well with Github and allows us to easily document our application and automatically build it using TravisCI (#DD7).	

Name	DD7 - Automatically building the codebase
Description: We need a tool that automatically builds new commits to the website repository and updates the website.	
Considered Alternatives: **TravisCI:** TravisCI integrates very well with Github and Hugo (#DD6) and satisfies all our requirements.	

Name	DD8 - Exporting the codebase to an installer
Description: In order to deploy our add-in to an installer, we need a tool that can export our code to an msi.	
Considered Alternatives: **WiX Toolset:** Wix is a free toolset, that is easy to setup using a visual studio project. The only thing to configure inside this project is an xml, which is human readable and therefore easy to understand.	

6 Verification and Validation

6.1 Performance Tests

Unfortunately, `rationally` can only modify shapes via the Visio API, which creates a lot of overhead. From the perspective of our add-in, creating and deleting shapes are atomic operations that can't be improved, performance wise. Creating a single shape programmatically takes roughly 1s[1]. Deleting a shape takes roughly half that time. However, a different approach to adding shapes that relies on stencils instead of creating shapes programmatically might yield some performance improvements.

Create and delete operations in our add-in are mostly operations on composite shapes, dealing with multiple shapes at once. This means that creating an alternative (composed out of 5 shapes) will on average take 5 seconds. Most of this time is taken by processing that the API performs, namely 5 times the creation process of a shape.

All of our enhanced items are composite shapes and thus take multiple seconds to create on a decision sheet. This means that user will have to wait several seconds for every add-operation he performs in `rationally` (independent of using the wizard, drag and drop or context menus). The deletion of said components takes half that time. Changes being made to a shape (changing the text, styling, position, etc) take almost no time (<< 1s) so those operations perform well.

6.2 Metric Results

In the table below, we list some descriptive statistics with regard to the metric results that were obtained for the three metrics described in section 4.5: Maintainability Index (MI), Cyclomatic Complexity (CY) and Class Coupling (CC). The data in this table, together with some method-specific metric results, are discussed in the following sections.

	MI	CY	CC
mean	79.78717	2.460285	5.172098
median	84	1	4
mode	90	1	0
standard deviation	16.98677	3.950038	5.592211
minimum	29	1	0
maximum	100	44	32

Table x: Descriptive statistics regarding `rationally`.

[1] On a computer with an Intel i7 4720HQ CPU @ 2.6Ghz

6.2.1 Maintainability Index

We calculated the MI for all members in all classes of the `rationally` project. We aimed for our members to score between 20-100. All members in the project scored this high. However, some members scored relatively low. An overview of those lesser scoring members (a score lower than 30) is shown below.

Namespace	Class	Method	MI Score
Rationally.Visio	RationallyAddIn	Application_CellChangedEvent (Cell) : void	29
Rationally.Visio.Forms.WizardComponents	TableLayoutMainContentGeneral	Init() : void	29
Rationally.Visio.View	RationallyView	AddToTree(Shape, bool) : void	29

The maintainability index is relatively low for these methods for several reasons. First of all, cyclomatic complexity plays a role in the calculation of this metric and as is discussed in the next section, the CY metric value for these methods is quite bad. Secondly, the amount of operands and operators plays a role in calculating the value of this metric. *Application_CellChangedEvent* and *AddToTree* contain a lot of logic, which consists of operators and operands. The third method, *Init*, initializes a view component of the wizard and performs many assignments to properties, hence the lower score on this metric.

6.2.2 Cyclomatic Complexity

In the table below, the methods in `rationally` that scored higher than 15 on the Cyclomatic Complexity metric are listed. They did not meet the requirement of scoring between 2-15. The rows that are highlighted have an exceptionally high (bad) score.

Project	Namespace	Method	Cyclomatic Complexity Score
Rationally.Visio	**RationallyAddIn**	**Application_CellChangedEvent(Cell) : void**	**44**
Rationally.Visio	RationallyAddIn	Application_QueryCancelSelectionDelete(Selection) : bool	18
Rationally.Visio	RationallyAddIn	Application_ShapeAddedEvent(Shape) : void	26

Rationally.Visio.EventHandlers.MarkerEventHandlers	**MoveUpAlternativeHandler**	**Execute(Shape, string) : void**	**34**
Rationally.Visio.EventHandlers.MarkerEventHandlers	**MoveDownAlternativeHandler**	**Execute(Shape, string) : void**	**34**
Rationally.Visio.EventHandlers.MarkerEventHandlers	AddForceHandler	Execute(Shape, string) : void	17
Rationally.Visio.View	TextLabel	Repaint() : void	19
Rationally.Visio.View	**RationallyView**	**AddToTree(Shape, bool) : void**	**38**
Rationally.Visio.View.Forces	ForceHeaderRow	Repaint() : void	26
Rationally.Visio.View.Forces	ForceTotalsRow	Repaint() : void	27
Rationally.Visio.View.Forces	ForceContainer	Repaint() : void	27
Rationally.Visio.View.Forces	ForcesContainer	ForcesContainer(Page, Shape)	23
Rationally.Visio.View.Forces	ForcesContainer	Repaint() : void	20
Rationally.Visio.View.Forces	ForcesContainer	AddToTree(Shape, bool) : void	20
Rationally.Visio.View.Forces	ForceTotalComponent	Repaint() : void	18
Rationally.Visio.View.Documents	RelatedDocumentContainer	RelatedDocumentContainer(Page, Shape)	18

The two most interesting results in this table are the methods *RationallyAddIn.Application_CellChangedEvent* and *RationallyView.AddToTree.* These methods score 44 and 38 respectively on the Cyclomatic Complexity metric, but they are also two of the three methods that came close to failing the requirement set for the MI metric.

The first mentioned method is responsible for handling the event of a ShapeSheet-cell changing value. For almost all other event handlers, we created a registry object that redirects each different type of a certain event to a specific handler, embedding the logic for different cases in different classes. For the *CellChanged* event, however, this has not been done yet. The large amount of unextracted logic explains the bad score of this method.

The second mentioned method (*AddToTree*) is part of the class that represents the root node of the view tree. The method is currently responsible for evaluating an incoming *Shape* and deciding where and as what the shape needs to be placed into the view tree. Again, the large amount of unextraced logic is a problem with this method.

6.2.3 Class Coupling

As described in the requirements section, we aimed for all our methods to score a value of 10 or lower on the class coupling metric. Unfortunately, a lot of methods did not pass this requirement. 152 methods scored 11 or higher on the class coupling method and some of them event scored 20 or higher. This last category is listed in the table below.

Namespace	Class	Method	Class Coupling Score
Rationally.Visio	RationallyAddIn	Application_CellChangedEvent(Cell) : void	32
Rationally.Visio.View.Information	InformationContainer	InformationContainer(Page, Shape)	27
Rationally.Visio	RationallyAddIn	RegisterMarkerEventHandlers() : void	25
Rationally.Visio.View.Forces	ForceContainer	ForceContainer(Page, Shape)	25
Rationally.Visio	RationallyAddIn	Application_QueryCancelSelectionDelete(Selection) : bool	24
Rationally.Visio	RationallyAddIn	RationallyAddIn_Startup(object, EventArgs) : void	24
Rationally.Visio.View.Forces	ForceContainer	Repaint() : void	24
Rationally.Visio.View.Forces	ForcesContainer	ForcesContainer(Page, Shape)	24
Rationally.Visio.EventHandlers	DeleteAlternativeEventHandler	Execute(RationallyModel, Shape) : void	24

.DeleteEventHandlers			
Rationally.Visio.Forms.AlternativeStateConfiguration	TableLayoutAlternativeStates	Save() : void	24
Rationally.Visio.View.Forces	ForceHeaderRow	Repaint() : void	23
Rationally.Visio.View.Forces	ForceTotalsRow	Repaint() : void	23
Rationally.Visio.View.Alternatives	AlternativeShape	AlternativeShape(Page, Shape)	23
Rationally.Visio.View.Documents	RelatedDocumentContainer	RelatedDocumentContainer(Page, Shape)	23
Rationally.Visio	RationallyAddIn	Application_ShapeAddedEvent(Shape) : void	22
Rationally.Visio.EventHandlers.MarkerEventHandlers	MoveUpAlternativeHandler	Execute(Shape, string) : void	22
Rationally.Visio.EventHandlers.MarkerEventHandlers	MoveDownAlternativeHandler	Execute(Shape, string) : void	22
Rationally.Visio.View	RationallyView	AddToTree(Shape, bool) : void	22
Rationally.Visio.Forms	ProjectSetupWizard	submit_Click(object, EventArgs) : void	22
Rationally.Visio.EventHandlers.DeleteEventHandlers	DeletePlanningItemEventHandler	Execute(RationallyModel, Shape) : void	21
Rationally.Visio.EventHandlers.DeleteEventHandlers	DeleteStakeholderEventHandler	Execute(RationallyModel, Shape) : void	21
Rationally.Visio	DeleteForceEventHandler	Execute(RationallyModel,	21

.EventHandlers .DeleteEventHandlers		Shape) : void	
Rationally.Visio .EventHandlers .DeleteEventHandlers	DeleteRelatedDocumentEventHandler	Execute(RationallyModel, Shape) : void	21
Rationally.Visio .EventHandlers .MarkerEventHandlers	EditRelatedFileHandler	Execute(Shape, string) : void	20
Rationally.Visio .Forms	ProjectSetupWizard	InitializeComponent() : void	20
Rationally.Visio .View.Alternatives	AlternativesContainer	AlternativesContainer(Page, Shape)	20
Rationally.Visio .View.Documents	RelatedDocumentsContainer	RelatedDocumentsContainer(Page, Shape)	20
Rationally.Visio .View.Planning	PlanningContainer	PlanningContainer(Page, Shape)	20
Rationally.Visio .View.Planning	PlanningItemComponent	PlanningItemComponent(Page, Shape)	20
Rationally.Visio .View.Stakeholders	StakeholdersContainer	StakeholdersContainer(Page, Shape)	20

After taking a closer look at the methods in the table above, a pattern emerges. Most methods fall in 1 of 2 categories: container constructors and delete handlers. Both types of methods deal with a complex operation on a composition structure: either creating one or deleting one. Secondly, they have to perform this operation on the model of our program and the view of our program. Because most child components are different types and most containers contain around 4 childs (of different types) this quickly adds up to the score on the class coupling method. Since the child classes only exist within their parent container, this coupling does not have many potential side effects to other classes in the future.

6.2.4 Recommendations

The performance of *rationally* is sufficiently good, but not great. However, there is a clear way to improve the performance of *rationally*. Therefore, to optimise performance we suggest to:

1. Optimise delete/create operations.

2. Avoid manually creating shapes and better reuse/configure stencils

As mentioned in section 6.2.2, there are two methods that are clear candidates for refactoring. The first being, *Application_CellChangedEvent*. The method can be restructured to the same registry structure that other event handlers use in `rationally`.

To optimize maintainability, we suggest to also refactor the *AddToTree* method. Its logic should be extracted to possible classes representing subelements of the tree. This would improve overall coherence in a way that parent A (child of root) is responsible for validating its own children and adding itself to the tree instead of the root class being responsible for that.

To go even further in trying to achieve better coherence, we recommend merging the view and the model that are now being maintained in `rationally`. As described in section 5.2.1.2, the class AlternativeState, is responsible for its own state, instead of it being stored in the model. A good way to reduce complexity and improve coherence would be to further implement this throughout all view classes, making the model obsolete, as well as the (complex) synchronization between view and model.

`rationally`'s event handlers perform a lot of list operations on the view, mostly consisting of searches in that tree. The combined model-view classes allow for tailored properties that encapsulate this logic. The RationallyView class can have properties for all enhanced containers that directly return those containers. That means that event handlers do not have to search the tree for these containers (and check if they even exist), but can just use the properties. These properties also reduce the amount of casting that is now often required in event handlers, since the properties can return the correct type.

Another way to reduce complexity in the code is described in SD11 and has to do with finding a more clever way to rebuild the View tree. It should be possible to make stub components (shapes that function as parent node for other shapes in the view tree while the actual parent is not yet included in the view tree) obsolete, reducing the amount of classes and logic in the project.

6.2.5 Stability

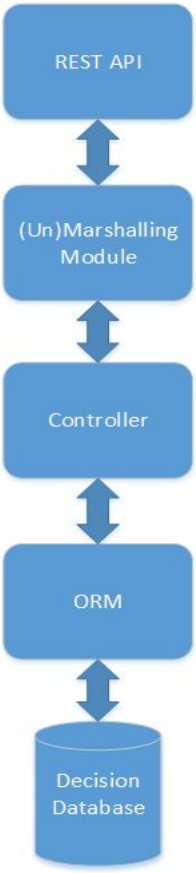
During the whole process of developing `rationally`, stability has been an underlying requirement. Evaluating all the functionalities that `rationally` offers in its current state, it can be concluded that exceptions rarely occur: less than 1 exception per session. All typical flows (CRUD operations on enhanced containers) do not yield errors, nor do undo/redo operations of those flows. The stability of `rationally` can thus be marked as good.

7 Future Improvements

rationally could be subject to several types of improvements in the future. Since *rationally*'s aimed application is being used in companies as a productive tool to produce documentation on decisions, we believe a server would be good addition to store decisions on. Since its aimed users work in the embedded system industry, we suggest an increase in flexibility as well and finally since *rationally* might be subject to additions and changes we suggest to improve the maintainability, stability and performance of *rationally*. In the following sections we will elaborate on these improvements.

7.1 Server

rationally would benefit a lot from having a central server on which decisions can be stored. This section proposes a high-level architecture for such a server, it will describe benefits of it and finally some foreseeable technical implications of it.



7.1.1 Architecture

In this section, we will briefly describe a possible stack for a *rationally* server. We propose a lightweight application, that will offer a REST API that the *rationally* add-in can communicate with. We propose REST because it uses HTTP as it was intended and because it is very modular and extensible.

The communication between add-in and server should still be investigated later since it is a complex issue. Maybe entire Visio files will be sent, but maybe a leaner way of messaging can be found. What will probably be needed is an (un)marshalling module that will be responsible for deserializing data (JSON/XML) to objects and the other way around.

The API would likely offer several operations that would each be encapsulated in a controller module. These modules will use the (un)marshalling module on one side and persist objects to/read data from the database using an ORM. We propose to use an ORM since it provides an abstraction for communicating with a database.

Figure 15: Proposed stack for a *rationally* server.

7.1.2 Benefits

A server would allow for different kind of benefits. The first would be version control and backing up, allowing the user to control the made changes and possibly roll them back if needed. Since decision making and documenting can be an iterative process, this would be a very useful feature. This would also allow the user to view the changes made to the decision, thereby offering a chronological view.

In addition, a server could store a lot of information about possible stakeholders, related decisions and documents. The list of possible stakeholders could even be collected from the employees database of the company the user works for, allowing for closer integration of `rationally` in the workflow. Storing related documents and decisions would make them selectable from within the `rationally` add-in as related information. This way, a decision relationship view can be made, showing how different decisions influence and affect each other. It would also allow for the user to search for a certain decision or decisions related to a subject.

7.1.3 Implications

A server would not only bring benefits to the `rationally` project, but also some extra complexity. Every operation that `rationally` offers to manipulate a decision sheet will also at some point have to be communicated to the server. The model also needs to be serializable in order to send it to and receive it from the server. New functionality will be required to set up communication with a specific server from within `rationally` and finally security will start to play a role: not only for communication but also with regard to (temporary) files that will be stored on a user's computer and not only on the server.

7.1.4 Discussion

Although a basic idea for a server has been described in this section, some questions still need to be answered. The communication between server and add-in need a protocol: what needs to send, when should it be send and what security forms are required.

On top of that, the synchronization is still not thought through. Will it be possible for multiple users to work on the same decision at a time? This would bring a lot of complexity to the project and requires clear constraints and conflict handling.

Besides the communication, the functionality of the server is not clearly defined yet. What operations should be offered to the users? We proposed searching and reusing components but maybe there are more possibilities.

To summarize, the server has a lot of potential when it comes to offering new features but it requires a lot of complex questions to be answered first. Even the basic functionality of it will come with a high workload to develop, but it will yield a very big addition to the `rationally` project.

7.2 Flexibility

`rationally` would also benefit a lot from an increase in flexibility. Currently, more than 3 alternatives at the same time or grouping shapes together can break the layout provided by `rationally.` In additional, multiple instances of the same container are not allowed.
In the future, these would be aspects of the add-in that can be changed in order to improve the flexibility of the application and would make it more convenient to use.

7.3 Refactoring

Finally, as described in section 6.2.4, there are quite a few refactorings still to be done. Doing these refactorings would improve the stability, performance and maintenance of the application and should therefore be a priority for the continued development of `rationally.`

8 Conclusions

This report provided a detailed description of *rationally*. It included an overview of the requirements for *rationally*, use cases for it and decisions that were made in the process of developing it. Furthermore, an overview of *rationally*'s architecture was provided, requirements were validated using metrics and future improvements for the application were suggested.

rationally is an easy to use tool for architects in various disciplines and it provides a way for architects to document decisions in a complete yet efficient manner: *rationally* combined multiple architectural views into one decision view and offers several ways to efficiently document decisions like a wizard and drag and drop functionality. It also integrates well in Visio in a way that all the original functions of Visio can still be used properly, including undo and redo operations. *rationally* is thus a complete, flexible and useful tool for architects.

Although the performance of *rationally* is not great, it is sufficient with respect to our requirements. The results regarding metric scores were good for the maintainability index, cyclomatic complexity and class coupling. There were, however, certain areas that scored lower. These areas are the handling of generic frequently occurring events and building the view tree. The results are thus predominantly good.

Even though our verification and validation yielded good results, there is still room from improvement in *rationally*. The two forementioned worse scoring areas can be refactored, the model and view of the application can be combined and after that event handling can be moved to event-related model components. Lastly, a server on which decisions can be stored on would be a good addition to the project.

All in all, *rationally* may not be production ready but it is a good example of how decision sharing can be improved. Throughout developing it, we gained a better understanding in what a decision consists of, how decisions are shared and how they can be optimally documented.

We believe that *rationally* could be a useful tool for the embedded system industry in a way that it could improve decision sharing and documentation, but we also suggest to empirically validate *rationally* as part of a case study to validate its value for the industry.

References

[1] Peng Liang, Anton Jansen, and Paris Avgeriou. Collaborative Software Architecting through Knowledge Sharing. Collaborative Software Engineering, pages 343–367, 2010.

[2] “Microsoft Visio Market Share, Customers and Competitors” - HG Data - https://discovery.hgdata.com/product/microsoft-visio [accessed: 04-jul-2017]

[3] “Code Metrics Values | Microsoft Docs” - Microsoft - https://docs.microsoft.com/nl-nl/visualstudio/code-quality/code-metrics-values [accessed: 05-jul-2017]

[4] A.H. Watson, T.J. McCabe, “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric” 1996

[5] Shatnawi, R. (2010). A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems (IEEE Transactions on Software Engineering, Vol. 36, No. 2).

[6] ISO/IEC/IEEE 42010:2011 - Systems and software engineering - Architecture description"

[7] U. van Heesch, P. Avgeriou, and R. Hilliard, “A documentation framework for architecture decisions,” The Journal of Systems and Software, 2011.