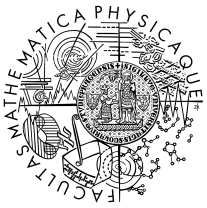



CHARLES UNIVERSITY
Faculty of mathematics
and physics

MASTER THESIS

Bc. Artur Finger

User interface of system ERIAN based on web technologies

Department of Software Engineering

Supervisor of the master thesis:	doc. Mgr. Martin Nečaský, Ph.D.
Study programme:	Master of Computer Science
Study branch:	Software and Data Engineering (Web)

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: User interface of system ERIAN based on web technologies

Author: Bc. Artur Finger

Department: Department of Software Engineering

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: ERIAN is a complex business rule management system developed by company Komix. Part of this system is the Rule Management Interface (RMI) which allows users to create, edit, schedule, test and otherwise manage their business rules. The RMI is implemented as a thick client based on C# and WPF, which has its disadvantages. This thesis provides a prototypical implementation of the RMI as a thin client based on cutting-edge web technologies. This thesis predominantly deals with the choice of the correct technologies for the task, while allowing development and maintainance of different customized versions of the RMI and making sure the prototype handles working with business rules seamlessly even if they are exceptionally large. The resultant RMI prototype is well testable and adds several new functionality features, compared to the original. It lays a good foundation for a complete re-implementation of the RMI as a thin client.

Keywords: BRMS UI ERIAN JS framework SVG

I would like to thank doc. Mgr. Martin Nečaský, PhD. for his time and supervision of this thesis.

Additionally, I would like to thank Ing. Jan Vrána and Ing. Radovan Jupa from Komix for introducing me to ERIAN and allowing me to work on a project that is meaningful.

Contents

Introduction

ERIAN is a Business Rule Management System (BRMS) developed by the company Komix. A BRMS is a software system used to define, deploy, test, execute, monitor and maintain decision logic used by operational systems of an enterprise. Such systems can also be referred to as Automatic Decision Support (ADS) systems. An example of an enterprise which uses ERIAN is the Customs Administration of the Czech Republic, which uses it to check for illegal activity and fraud in customs declaration documents.

From an architectural standpoint, ERIAN has a client-server architecture. The server component is responsible for storing and executing the decision logic. The client component, known as the Rule Management Interface (RMI) *(Figure 1)*, is used by enterprises (customers of Komix) to create, edit, schedule, test and otherwise manage their decision logic.

The decision logic (a.k.a. business rules or just *rules*) includes policies and requirements that are used to determine the tactical actions that take place in the operational systems of the enterprise. In ERIAN's RMI rules are represented visually as sequential diagrams *(Figure 2)* similar to a flowchart diagram.[1] Rules are composed of *elements* that may perform actions (e.g. an assignment to a variable - *Figure 3*) or control flow (e.g. a conditional statement element).

The main drawback of the RMI is that it is implemented as a thick client (using WPF technology and C#). The goal of this thesis is to design, create and test a thin client prototype of the RMI using modern web technologies for the implementation. This approach will allow the customers of Komix to use the RMI in a SaaS fashion via a web browser, thus eliminating the drawbacks of a thick client (i.e. the need for installation, distribution of updates, etc). In addition, the prototype also contains several new functionality features.

This thesis was developed in cooperation with Komix which provided detailed explanation of the GUI and functionality of the current RMI, specified the goals, answered questions during the analysis of the requirements and provided feedback about the emerging prototype.

The work entailed getting acquainted with the current RMI, understanding the goals, reverse-engineering the logical model of rules based on a demonstration of the current RMI (the model was then reviewed by Komix), analyzing the requirements, choosing suitable technologies and designing and implementing a functional prototype.

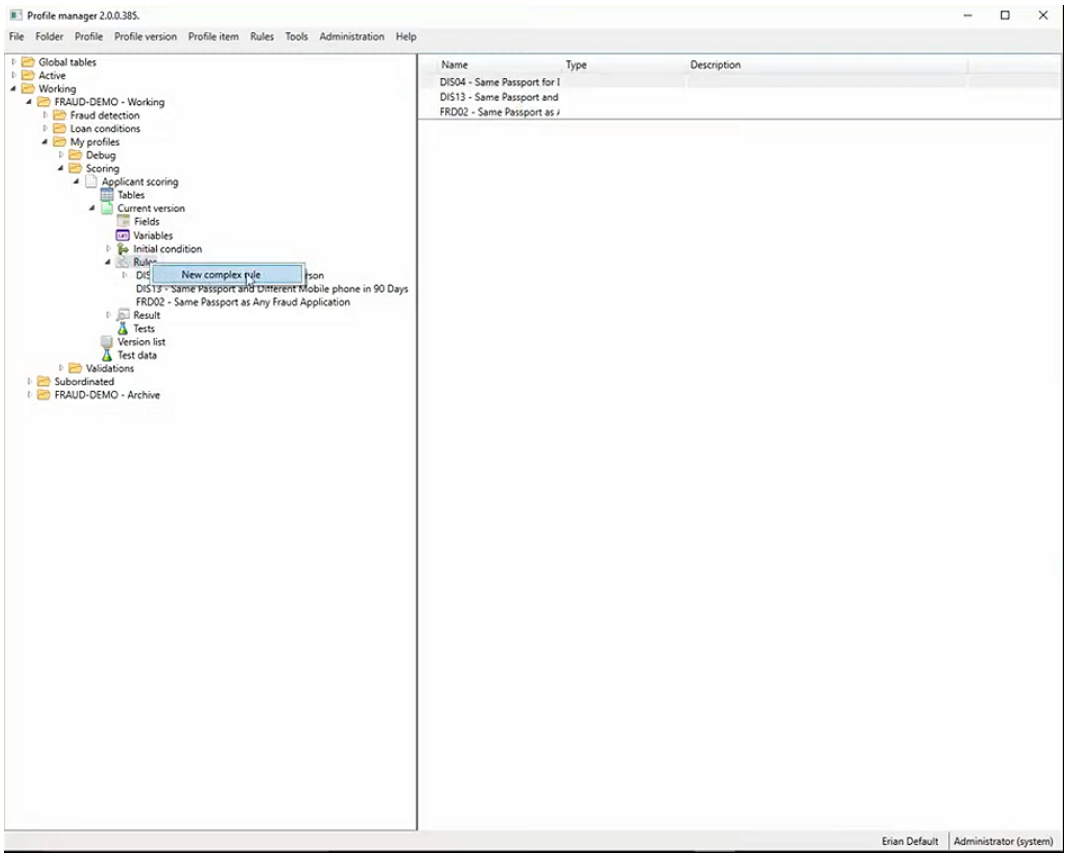


Figure 1: The original Rule Management Interface

This screenshot shows a menu at the top, which allows many action to be performed. These actions include creating, deleting, editing, saving, testing and scheduling rules. On the left there is a folder structure containing rules. On the right, a group of rules is shown. This GUI allows rules to be grouped into so called *profiles*.

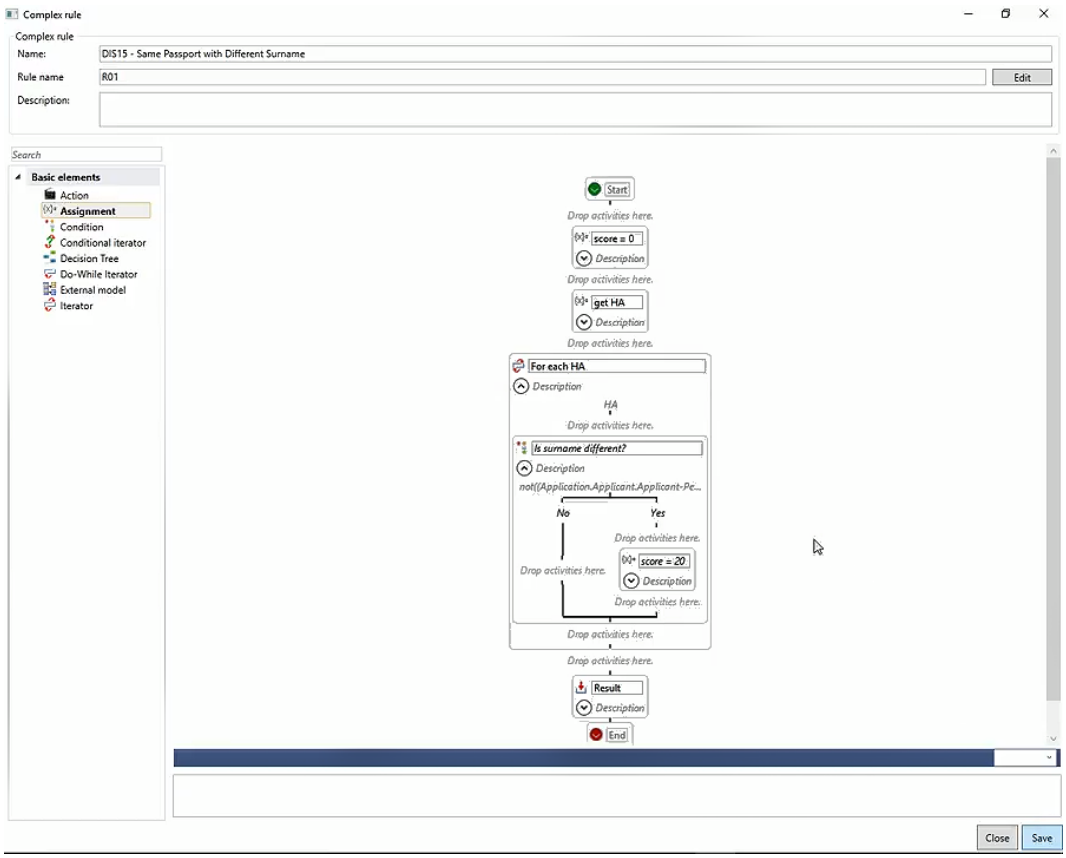


Figure 2: The original Rule Management Interface

This screenshot shows the *rule editor*, which allows adding elements into the rule or into any sub-rule. A rule is a sequence of elements and some of them can contain a sub-rule (e.g. if conditional statement element has two sub-rules and one or the other is executed based on the truth value of the condition). The panel on the left offers types of elements that can be instantiated by dragging them into the rule. Elements of the rule can be collapsed or deleted and they also have a description. After double-clicking an element, a pop-up windows appears and it allows editing of the elements meta-data (e.g. double-clicking a conditional statement element allows the user to edit its condition, etc).

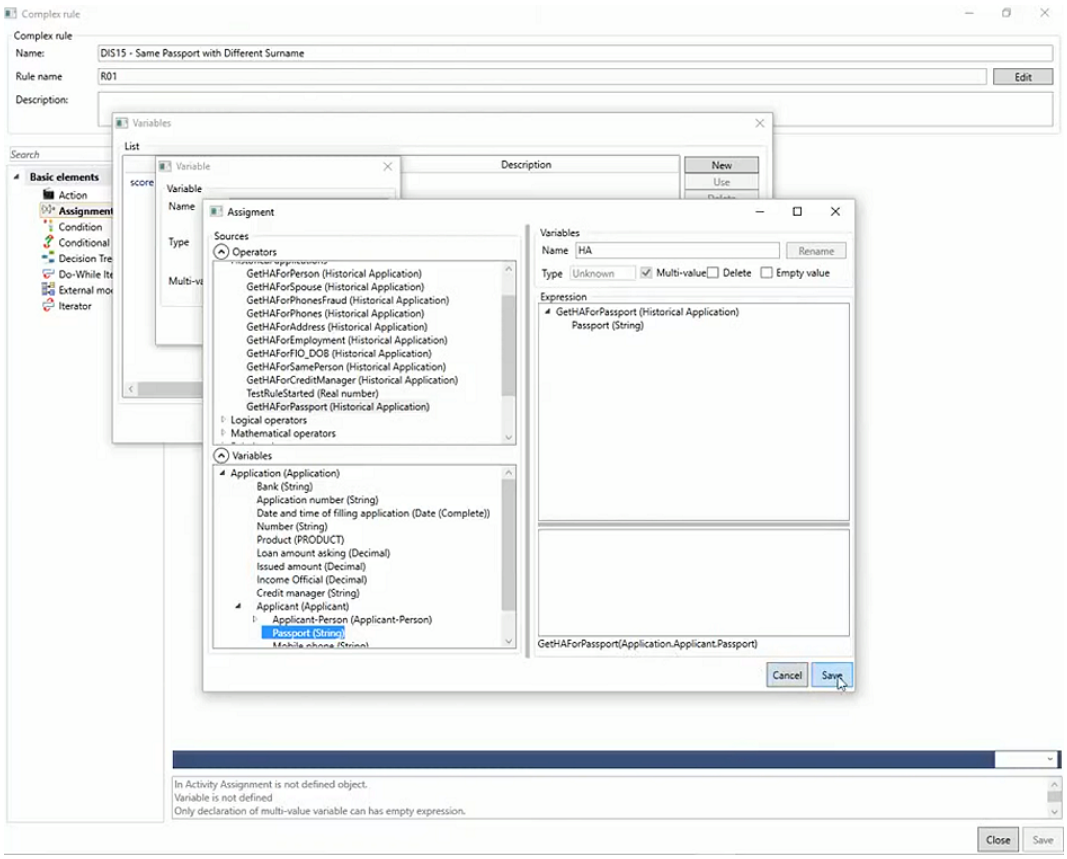


Figure 3: The original Rule Management Interface

This screenshot shows the GUI for creating expressions (in the *Assignment* pop-up window). The expression is built by dragging operations from the upper left box and variables or their attributes from the lower left box into the expression tree on the right. Values and operations are typed and can be multi-valued.

1. Goals

The goal of this thesis is to analyze, design and implement a thin client prototype of the RMI using modern web technologies.

In order to reduce the functional complexity of the RMI, the prototype is only required to implement the most essential component of the RMI, which is the *rule editor*. Other components should be left out and the prototype should be ready for their addition.

The prototype should satisfy the following functional and qualitative requirements:

1. **Internationalization** - It must be possible to change the language of all text displayed in the GUI.

2. **Component-based UI** - A mechanism that allows the definition of UI components must be used to implement the GUI. UI components are reusable and testable parts of the UI, usually written using a combination of pseudo-HTML, CSS and JS.

3. **Customizability** - *ERIAN is currently being used by several enterprise customers. Some of them have made additional feature requests, which has led to the development and support of multiple customized versions of the RMI.*

   It must be possible to write customized versions of the RMI, so that it can be tailored to the specific needs of different customers. It should be possible to add or remove functionality and GUI components, and change GUI styling and layout.

4. **Testability** - The correctness of the functionality of the prototype must be testable.

5. **Functionality of the rule editor**

   - add an element to the rule via drag and drop
   - edit the meta-data of an element (e.g. specify the condition of a conditional statement element)
   - edit mathematical expressions
   - prevent saving elements in an invalid state (e.g. when the condition is empty, the meta-data cannot be saved)
   - select multiple elements
   - delete elements
   - undo and redo
   - collapse and expand sub-rules
   - browse the rule (e.g. by scrolling up and down)
   - copy and paste elements
   - zoom
   - save the graphical representation of a rule as image
   - serialize and deserialize rules in order to save them or send them to the server component

2. Specification of Rules

A rule describes decision logic as a sequence of basic building blocks called *elements* 2.1. Elements in a rule will be executed by the server component in sequential order (visually from top to bottom).

An important part of a rule are *variables* and mathematical *expressions* 2.2. Expressions allow the user to compute a value out of variables and literals using mathematical operators and calls to predefined functions. Variables allow storing of the computed results and they can be created inside the rule or passed to it from an outside context (e.g. in the case of the Customs Administration, the rules that ERIAN's server component executes for them usually get passed a customs declaration document which was submitted for risk analysis).

The whole rule and all the algorithmic information within in it must be serializable and deserializable, so that it can be sent to the server to be saved or executed, or downloaded from the server, so that it can be edited in the RMI.

2.1 Elements

Elements represent individual commands or actions executed by the server component. An element can contain zero or multiple sub-rules (e.g. the *if element* has two sub-rules, one for the true branch and one for the false branch). For simplicity, only the following most important types of elements will be implemented 2.1:

- **Assignment** element - It tells the server component to assign a mathematical *expression* to a variable. The assignment works in two ways:

  If the variable has not been declared yet (in the particular variable scope where the element appears), it gets declared and its type is inferred from the return type of the expression that is assigned to it.

  Otherwise, the expressions is assigned to the already declared variable and the return type of the expression is checked against the declared type of the variable.

  It is possible to assign an expression containing a variable to the same variable, but not inside an element that declares that variable.

- **If** element - It allows flow control via a conditional statement. It has two sub-rules (the true and false branches) and one condition. The condition is a boolean expression and depending on its truth value, either the true branch or the false branch will be executed by the server component.

- **Iteration** element - It declares an iteration variable, which loops through the values of some multi-valued variable. The element has a sub-rule, which gets executed by the server component once for every item contained in the iterated variable. The iteration variable is accessible only inside the sub-rule.

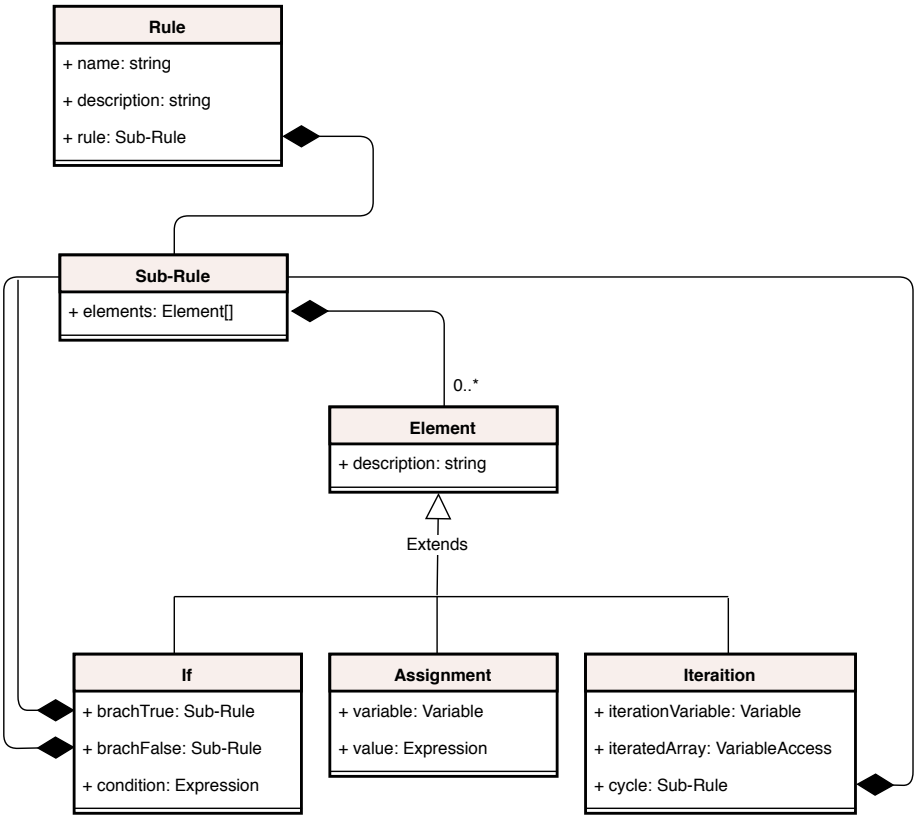


Figure 2.1: Logical model of rules

A rule has a name, a description and one sub-rule. Sub-rules contain an array of elements (order matters). Elements may contain sub-rules.
Expressions and variable accesses are described in figure 2.7, variables are described in figure 2.3.

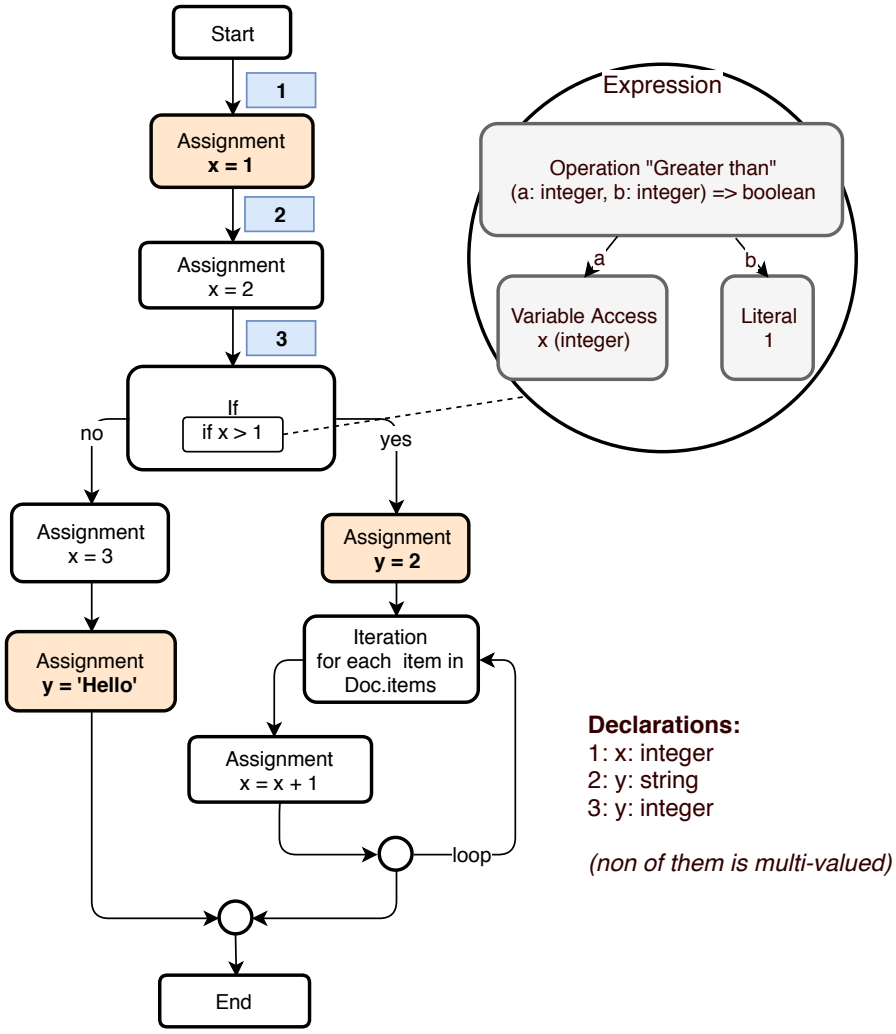


Figure 2.2: Example of a rule

This rule makes a few very simple expression assignments to variables **x** and **y**. It also has one conditional statement element and one iteration element.
The elements where declaration of a variable occurs are highlighted in red. This example demonstrates the fact that multiple variables of the same name can be declared independently inside different variable scopes. In this example, it is no longer possible to declare the variable **y** at positions **1**, **2** or **3**, because this declaration would over-shadow the declarations present inside the sub-rules of the conditional statement element (”if” element).
The condition of the ”if” element is a boolean expression with the comparison operator as its root and a variable access of **x** and an integer literal as its leaves.

2.2 Variables

Variables are used by the server component to hold values. They act as references to their declaration 2.3. They are typed and the type is stored with the declaration.

As already mentioned, there are two types of variables - local and global.

- **Local variables** can be declared and used inside the rule using dedicated elements (e.g. the assignment element). They are block-scoped, i.e. a variable is visible only in the rule where it is declared and only to the elements that follow the declaration. Plus, it is also visible inside of all sub-rules of those elements.

- **Global variables** are passed to the rule by the server component during the execution of the rule and they are accessible by all elements. Declarations of all global variables for a specific rule are downloaded from the server component.

  Since ERIAN is usually used for processing documents (making automated decisions based on an input document), a very important global variable is the one representing the input document.

Since variables may hold a complex type, it is possible to access their attributes via dot notation (e.g. *document.author.name*). The type of a variable is determined during its declaration and it cannot be changed later on.

Variables can also be multi-valued (i.e. representing an array) and it is possible to access their concrete elements using an integer index.

2.3 Types

Rules are statically typed. Variables and operations are typed and only expressions of a correct type can be assigned to a concrete variable or supplied as an argument to an operation.

Types are uniquely identified by their name and they can be complex or simple. Complex types have typed attributes, simple types do not 2.3.

The list of all available types and their type definitions is downloaded from the server component 2.4. However, the implementation of the RMI requires the presence of four basic simple types:

- **boolean** - because the condition of a conditional statement element is boolean
- **integer** - because the index used for array access must be an integer
- **float** - because it allows the use of a float literal as an argument of an operation
- **string** - same reason as float

These types must always appear int the list of downloaded types.

For example, if a document is supplied as a global variable and it is an XML document, the server component generates its type definitions directly based on its XML Schema 2.5.

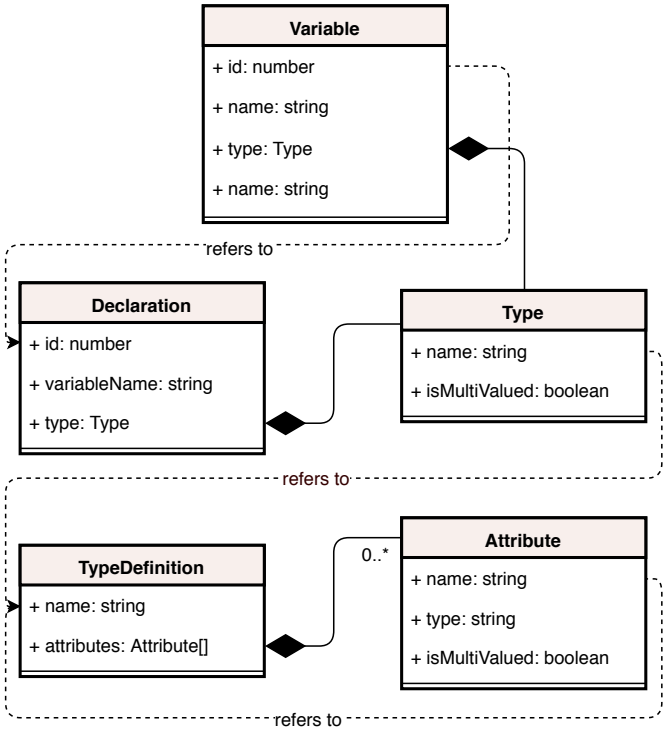


Figure 2.3: Logical model of variables, declarations and type definitions

Type definitions describe complex or simple types. A complex type is recursively defined by its name and typed attributes, whereas a simple type only has a name and no attributes (e.g. number, string, etc). Types are uniquely identified by their names.
Variables refer to their declarations, which store the type. The type is also stored with each variable to make debugging and working with variables easier, but it is always equal to the type of the declaration.
IDs 0-999 are reserved for global variables and 1000 and higher are used by local variables.

```
{ "name": "Doc", "attributes": [
{ "name": "People", "type": "Person", "isMultiValued": true }
]},
{ "name": "Person", "attributes": [
{ "name": "age", "type": "integer" },
{ "name": "passport", "type": "string" }
]},
{ "name": "string" },
{ "name": "number" },
{ "name": "integer" },
{ "name": "float" }
```

Figure 2.4: Example of type definitions

This is an example of how type definitions downloaded from the server component may look in JSON format.

In addition, type conversions are allowed (e.g. for converting integer to float, etc). They are transitive and the list of conversions is again downloaded from the server component 2.6.

2.4 Expressions

Mathematical *expressions* are used for computing values (eg. the value assigned to a variable is an expression and the condition of a conditional statement is a boolean expression).

An expression is a tree with operations as its inner nodes and variable accesses, array accesses or literals as its leaves 2.7 2.8.

The types of operation arguments must be checked and it should not be possible to build and save an invalid expression.

2.4.1 Literals

When creating an expression, the user can specify *literal* values. However, array literals are not supported, even though variables can be multi-valued.

2.4.2 Operations

Operations are described by their name, an ordered list of named and typed arguments (including whether the argument is multi-valued or not) and a return type. The list of available operations is again downloaded from the server component.

For illustration purposes, the list of operations usually entails the following:

- logical (and, or, not) *argument and return types are boolean*
- mathematical $(+, -, *, /, \%, max, min, avg, exp)$ *argument and return types are integer or float*
- relational $(=, >, >=, !=, <, <=,)$
- classic date and time operators
- *and many more...*

2.4.3 Variable Accesses

Variable accesses allow two ways of using a variable in an expression.

The first way is simply using the variable (e.g. $x = y$) and it can be applied to variables of both simple or complex type.

The second way is accessing an attribute of a variable or recursively an attribute of an attribute, etc (e.g. *document.id* or *document.person.name*). This is only possible with variables of a complex type.

2.4.4 Array Accesses

Array accesses are used to access specific elements of a multi-valued variable using an integer index (starting at zero).

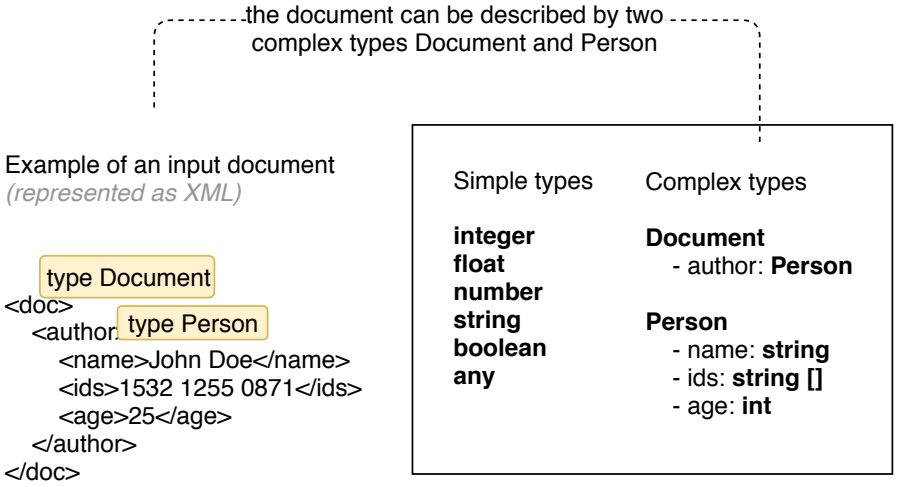


Figure 2.5: Example of types

In this example there are a few simple types and two complex types.
The server component supports the same classic types that are found in most programming languages (integer, string, boolean, etc).
Complex types may be generated by the server component based on the XML Schema of an input document.

```
{ from: "integer", to: "float" },
{ from: "float", to: "number" },
{ from: "number", to: "string" },
{ from: "string", to: "any" }
```

Figure 2.6: Example of type conversions

This is an example of how a list of supported type conversions downloaded from the server component may look in JSON format.
Type conversions are transitive, for instance in this example it is possible to convert an integer variable to string.

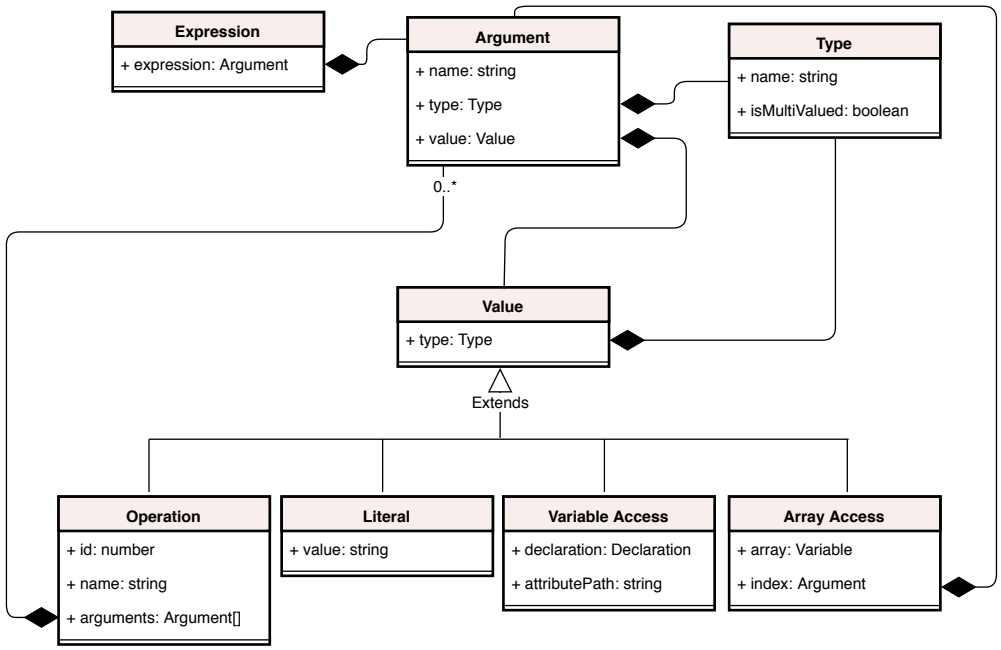


Figure 2.7: Logical model of an expression

Expressions are composed of a tree of operations, with literals, variable accesses and array accesses in its leaves. Each node is typed and the type of the argument is supposed to be the same (or convertible to) the type of the value assigned to it. Otherwise, the expression is invalid.
Variables and declarations are described in figure 2.3.

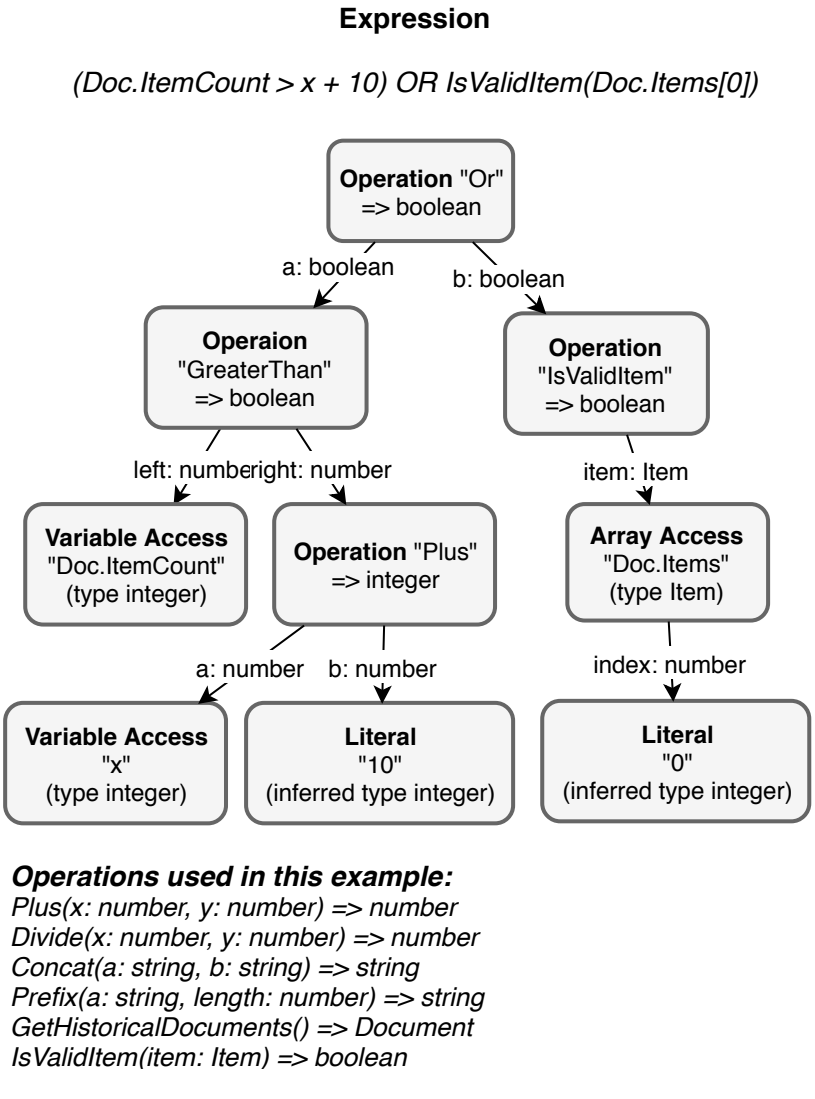


Figure 2.8: Example of an expression

Expressions are composed of a tree of operations, with literals, array accesses and variable accesses in its leaves.

3. Analysis of the Requirements

Functional and quality requirements of the RMI prototype were analyzed in order to understand and describe more precisely what should be implemented.

3.1 Functional Requirements

The functional requirements were put together based on observing and trying out the functionality of the current RMI, as well as leading discussions with Komix employees (more specifically with two member of the team working on the RMI), which were accompanied by developing several pre-prototypes.

The difference between the functionality of the current RMI and the new functional requirements is that they are focused on keeping the rule valid at all times.

In the current RMI it is possible to create an invalid rule by not finishing expressions, mismatching argument types, etc. There is also a lack of auto-completion and filtering.

The resulting functional requirements are described by a use-case diagram 3.1, which is elaborated in the following scenarios:

1. **Save a rule**

Starting situation	A rule is open.
Normal scenario	The user clicks *Save* or leaves the page. The rule gets sent to the server component where it is saved.
Final situation	The rule is saved on the server component and the user is notified about success or failure.

2. **Load a rule**

Starting situation	A user sees a list of available saved rules (loaded from the server component).
Normal scenario	The user selects a rule from the list.
Final situation	The selected rule is open in the rule editor.

3. **Export a rule as an image**

Starting situation	A rule is open.
Normal scenario	The user clicks *Export*. The rule is rendered, which is downloaded to the user's computer.
Final situation	The user's folder with downloaded files contains a vector graphic image of the whole rule (not just the part currently visible on the screen), at 100% zoom. Collapsed elements remain collapsed in the downloaded image.

4. **View and edit the details of an assignment element**

Starting situation	A rule is open.
Normal scenario	The user clicks a specific assignment element inside the rule to open it. A dialog appears where the user can edit the description text of the element, select a variable and edit the expression that is assigned to it. The user inputs the name of the variable and creates / edits the expression.
What can go wrong	If the user inputs a variable name that overshadows an existing variable used further in the rule, the user is alerted an the details will not save until the error is corrected. Ditto when the variable name is left empty. Ditto when the type of the expression differs from the type of the variable (when using an existing variable, visible in the scope of this element). Ditto when there is any validation error with the expression (*scenario 5*).
Final situation	The details of the assignment element are saved. If the variable name is not among the variables visible in the scope of this element, a new variable of this name gets declared and its type is inferred from the type of the expression.

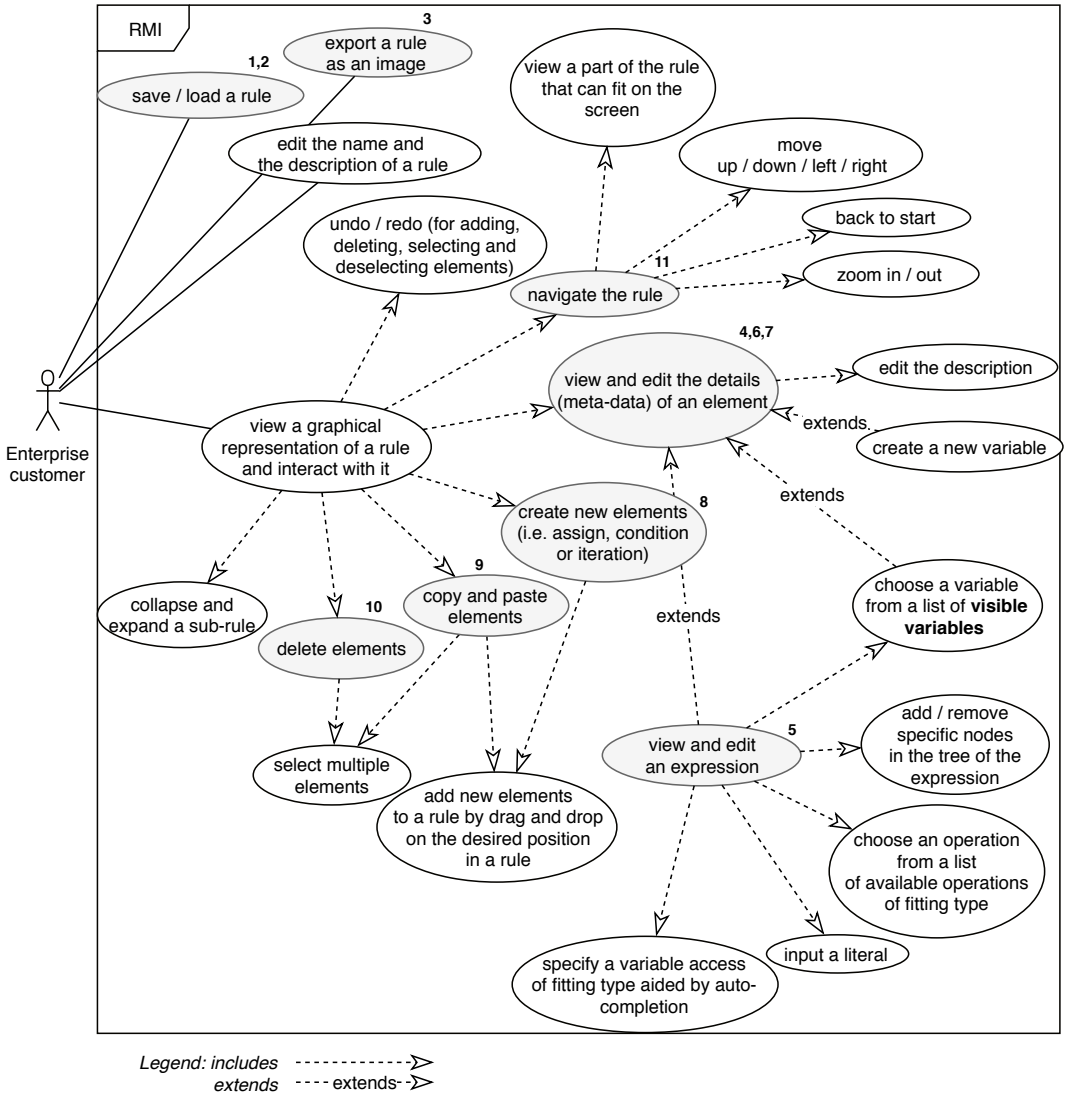


Figure 3.1: Functional requirements

This use-case diagram was put together based on observing the current RMI and having discussions with Komix.

The numbered use cases are elaborated into use case scenarios.

5. Edit an expression

Starting situation	A dialog containing an expression is open (e.g. the dialog of an assignment element or a conditional statement element).
Normal scenario	The user builds the expression by filling empty arguments shown in the expression tree. The user inputs literals via a text input. For each empty argument, only the operations and variable accesses of corresponding type are shown (i.e. only those of the same type or type convertible to the expected type of the argument). The user can see a list of all such operations and variable accesses and he / she can also filter them by typing in their name / attribute path. Only the variables visible in the scope of the element that owns this expression are listed among available variables. Also, the user can delete any node of the expression tree in order to change the expression.
What can go wrong	If the user forgets to fill some empty arguments, he / she gets alerted and the expression will not save until the errors are fixed. Ditto when the type of the expression does not correspond to the expected type (e.g. the type of a conditional statement has to be boolean). Ditto when the user specifies a literal as an argument, but the type of the literal does not correspond (i.e. the literal cannot be parsed as a type that is convertible to the expected type of the argument).
Final situation	The valid expression gets saved.
Explanation	The ability to filter operations / variable accesses allows the user to be efficient even in a situation where there are many operations or variables and attributes. On the other hand, it is important to show the list of all available operations and variable accesses, because the user likely does not know all of them by heart.

6. View and edit the details of a conditional statement element

Starting situation	A rule is open.
Normal scenario	The user clicks a specific conditional statement element inside the rule to open it. A dialog appears where the user can edit the description text of the element, select a variable and edit the condition of the conditional statement. The user edits the condition (*scenario 5*).
What can go wrong	If the type of the condition that the user has built is not boolean, the user gets alerted and the dialog will not close until the error is fixed.
Final situation	The details of the conditional statement element are saved.

7. View and edit the details of an iteration element

Starting situation	A rule is open.
Normal scenario	The user clicks a specific iteration element inside the rule to open it. A dialog appears where the user can edit the description text of the element, select the iterated variable access and create a new iteration variable that will loop through the values of the iterated variable. The list of all multi-valued variable accesses visible in the scope of this element is shown and it can be filtered by typing the attribute path. The user selects a variable from this list. The user also input the name of the newly created iteration variable.
What can go wrong	If no multi-valued variable is selected, the user is notified and the details will not save until the error is fixed. Ditto when the name of the iteration variable is the same as a name of any variable visible in the scope of this element.
Final situation	The details of the iteration element are saved. A new variable (the iteration variable) is declared and its type is the same as the type of the iterated variable, only not multi-valued.

8. Add a new element to a rule

Starting situation	A rule is open. A list of all available element types is visible (i.e. assign, condition, iteration).
Normal scenario	The user drags an element type from the list and drops it somewhere in the rule in a designated area. Each of these areas corresponds to a position between some two elements in some sub-rule or a position at the start / end of a sub-rule.
What can go wrong	If the user drops the element outside any designated area, nothing happens.
Final situation	A new empty element of the chosen type is created and inserted into the rule at the position corresponding to the area the element was dropped on. An element editing dialog opens immediately to allow the user to input the details of the element (*scenarios 4, 6, 7*)

9. **Copy and paste**

Starting situation	A rule is open.
Normal scenario	The user selects multiple elements, presses *Ctrl + C*, then *Ctrl + V* and selects where to drop the pasted elements.
What can go wrong	If the pasted elements declare a variable that collides with an already declared variable in the target scope, or over-shadow another variable declaration, or they use** a variable that is not declared in the target scope, then the elements in question will be invalidated. They and all their ancestor elements* are highlighted in red and the rule cannot be saved until the invalid elements have been dealt with.
Final situation	The copied elements are inserted into the rule on the specified position in the same order as they appeared in the rule originally. Elements are copied including all their sub-rules.
Definitions:	(*) A *parent element* is an element that contains a sub-rule, which contains the invalid element. *Ancestor elements* are defined recursively as the parent element, its parent, etc. (**) A variable declaration is being used by another element either when it assigns a value to that variable (e.g. in an assignment element) or when it uses that variable (e.g. via a variable access) inside an expression (e.g. in the condition of a conditional statement).

10. **Remove selected elements from the rule**

Starting situation	A rule is open.
Normal scenario	The user selects multiple elements and presses the *Delete* key.
What can go wrong	If a declaration is deleted, but some elements that use** this declaration remain in the rule, the elements get invalidated. They and all their ancestor elements* are highlighted in red and the rule cannot be saved until the invalid elements have been dealt with.
Final situation	The selected elements are removed from the rule.

11. **Navigate / browse a rule**

Starting situation	A rule is open.
Normal scenario	The user navigates the rule by moving it up / down / left / right or zooming, e.g. by pressing the *Arrow* keys.
Final situation	The graphical representation of the rule is moved or resized according to the user's request. All rules and sub-rules and their elements that fit inside the screen are displayed.
Explanation	It is important that the browsing of a rule happens in a single window and that the **structure of the rule is fully visible** (i.e. sub-rules, elements of sub-rules, their sub-rules, etc). This functionality is required by the stakeholders. *Therefore, it is, for example, insufficient to display a rule without its sub-rules and then navigate it by opening the sub-rules in a new tab, and perhaps using breadcrumbs to keep track of all the open sub-rules. (This technique, by the way, would decrease the demands on rendering and also it might easily decrease the cognitive overhead the user has to deal with when working with a complex rule, by allowing the user to focus on each sub-rule separately.)*

3.2 Quality Requirements

The quality requirements were put together based on analyzing the goals and the functionality requirements, and prototyping.

1. **Performance:**

   (a) A rule containing up to 1000 elements and / or 100 levels deep nesting of sub-rules shall render in under 1s during browsing at 100% zoom.*

   Explanation: Some enterprise customers have created rules that contain up to 1000 elements and up to 60 nested sub-rules (60 levels deep nesting). The rule editor must therefore support rules of this size and render them reasonably fast, even though customers should be discouraged from creating such large rules, since they are very hard to work with.

One second seems to be a generally accepted time limit during which *"the user's flow of thought stays uninterrupted"*[57].

(b) A rule containing up to 1000 elements and / or 100 levels deep nesting of sub-rules shall render in under 3s when it is being exported as an image.*

(*) Measured on a regular computer (1920x1080 screen resolution, 8GB RAM, Intel i7 CPU) in Google Chrome browser.

2. **Architecture:**

(a) It shall be possible to translate all text visible to the user to a new language simply by translating a dictionary file.

(b) A mechanism that allows the definition of UI components shall be used to implement the GUI.

(c) There shall be only one web application for all enterprise customers at once.

Explanation: From the point of view of creating customized versions for different customers, this means the following:

The application shall be loaded in two steps. First, a generic part of the application (common to all customers) is loaded. Then, after the successful authentication of the user, the customer-specific part of the application is loaded and integrated into the previously loaded part 3.2. Afterwards, the user can start utilizing the application.

As a result, the architecture shall include a mechanism that allows this kind of 2-step configuration-based loading.

Note that customers are companies and users are their employees. Therefore there will be just several customized versions (max 5).

Side notes: *Implementing authentication is not a part of this thesis and neither is the creation of a customized version of the RMI prototype. This thesis only deals with finding the technical solution that allows the creation of customized versions of the RMI under the described conditions.*

3. **Tooling:**

(a) Unit tests shall be demonstrated.

Explanation: The prototype shall include a unit test runner and a testing library, and several examples of unit tests including the tests of UI components.

(b) End-to-end tests shall be demonstrated.

Explanation: *End-to-end tests are tests that simulate user's behavior in an open web browser (using web browser drivers) and check the visible changes that occur in the application as the result of that behavior.*

The prototype shall include an end-to-end test runner and a testing library, and several examples of end-to-end tests. The tests shall run on the latest version of Google Chrome.

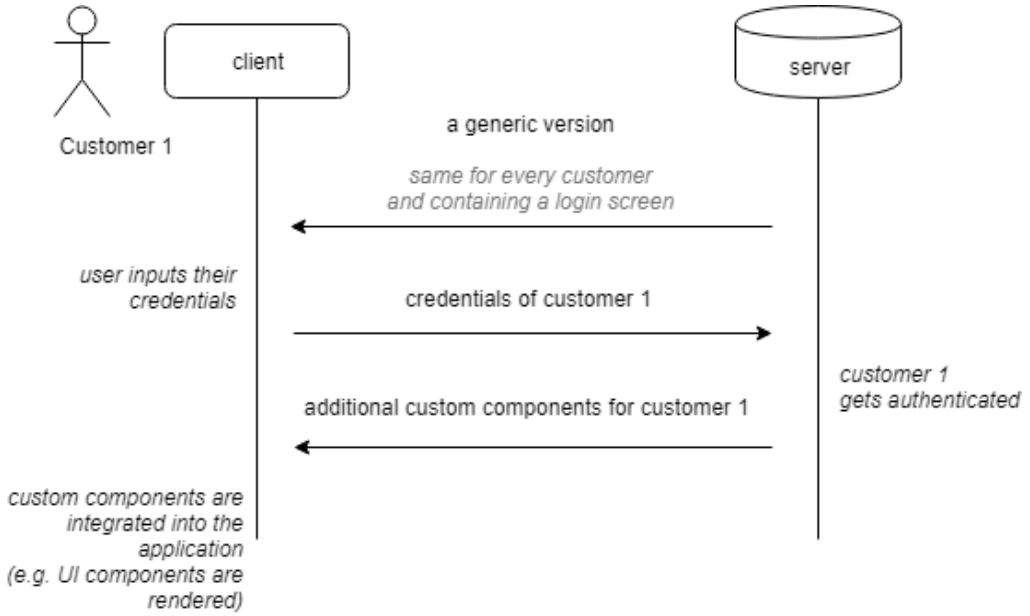


Figure 3.2: Architectural requirement: single application

There is only one application for all enterprise customers. After successful authentication the customer receives the rest of the application (the components specific to this customer) and can start using the application.

4. Design

The RMI prototype is designed to satisfy the functional and quality requirements and to adhere to software engineering best practices. The design is mainly concerned with choosing the right technologies for development, finding a way of rendering rules which is optimized for speed and designing a customization mechanism.

4.1 Preliminaries

- **ES5** - EcmaScript 5 is the old classic version of JavaScript. The specification of verions 5 was finished in 2009 and is now supported by all modern browsers. The features of the language are obsolete and the syntax is limiting. It is intended for small projects only.

- **ES8** - EcmaScript 8 is the most modern version of JavaScript. It offers very useful language constructs that were missing in ES5 such as classes, constructors, block-scoped variables, constants and modules.

  However, due to the lack of support on older versions of browsers, ES8 is usually not used directly. Instead, the standard approach is to transpile ES8 code to ES5 before deploying it. The de-facto standard transpilation tool is Babel.

- **ES8 module** - An ES8 module is a file containing ES8 code, which *exports* some classes, functions or variables, etc. Other ES8 modules can *import* what is exported and use it. *(This is a big improvement over ES5, where code is executed in a way analogous to concatenating all files together.)*

- **TypeScript** - TypeScript is a statically typed superset of ES8. It prevents type-related errors in code and adds several useful constructs to ES8 including enums, interfaces, generics and access modifiers for class attributes. These features make TS suitable for large projects.

  TypeScript is compiled to JavaScript (again usually to ES5) using the TypeScript compiler.

- **TS typings** - TS typings are a statically typed description of the API of a JS library (using TypeScript). Whenever a TS application wants to use a JS library, it is preferred that the library has typings and many libraries do.

  Typings can come with the installation of a library or they can be often found in the de-facto standard *DefinitelyTyped* repository for typings.

  When typings for a library cannot be found anywhere, it is possible to write custom typings for it or just bypass the need for typings completely by using type *any*. Bypassing typings obviously means the loss of intellisense and the other advantages of static typing.

- **Babel** - Babel is a transpilation tool, which can translate one flavor of JavaScript to another (e.g. ES8 to ES5, ES6 to ES5, etc).

  Additionally, Babel can be configured via plugins to extend its transformational capabilities.

- **Webpack** - Webpack is a module bundling tool. It searches the dependency tree of ES8 modules and bundles all the required modules into a single file.

  Webpack can be heavily customized with plugins and it is usually used in conjunction with Babel and the TypeScript compiler.

  Modern *front-end frameworks* (e.g. Angular, React and Vue) use Webpack in order to be able to leverage ES8.

- **Webpack code-splitting feature** - Code splitting[23] is a feature of Webpack, which allows code to be bundled into multiple bundles instead of just one. One of the bundles is the main bundle and it can download the other bundles during runtime via a Promise.

- **Promises** - A JS Promise is a class which represents the eventual completion or failure of an asynchronous operation, and its resulting value.

- **Front-end framework** - A front-end framework (a.k.a. a JavaScript framework) is a set of tools that facilitate the development of single-page applications by solving common problems. It is a mixture of libraries, executables, configurations, boiler-plate code best practices.

  Most of them provide a template-based mechanism for defining UI components using a mixture of JS, CSS, HTML and custom syntax.

  Some frameworks force the developer to use a predefined architectural pattern such as model-view-controller or dependency injection. Some frameworks allow the programmer to use ES8 or TS by providing a pre-configured transpilation pipeline (e.g. Webpack and Babel or TypeScript compiler). It is common that frameworks come with a pre-configured test runner and a testing library (if the code uses transpilation, the testing pipeline must also be set up to use transpilation).

  It is also worth noting that popular JS frameworks have an additional ecosystem of plugins and styling libraries with pre-defined UI components.

  Examples of the most popular modern JS frameworks are React, Angular and Vue.

  Frameworks make setup and development of large projects easy. Of course this comes at the cost of having to learn them and accepting any limitations they might impose on the project. Nevertheless, a strong advantage of frameworks is that by imposing structure on the project they make it much easier for other developers to understand the code (if they are familiar with the framework).

- **Selenium** - Selenium is a browser automation tool, which is most widely used for end-to-end testing of web applications. Selenium runs a real web browser of choice (e.g. Chrome, Firefox, etc.) and simulates DOM events.

- **Angular** - Angular is a popular JS framework created and maintained by Google. It uses TS and comes with pre-configured transpilation and testing tools. Its API is extensive.

  (For a more in-depth description of Angular see Section 4.4.1)

- **React** - React is a popular JS framework created and maintained by Facebook. By some statistics[64] is is the most popular JS framework of 2017 world-wide. It uses ES8 by default, but can be configured to use TS. Its API is small compared to Angular, but it has many plugins to balance that out.

  (For a more in-depth description of React see Section 4.4.1)

- **Front-end routing** - Front-end routing is a technique used in SPAs in order to structure the UI into multiple pages or views, which can be bookmarked (basically, it is an imitation of classic web browsing but done inside an SPA and without page loads). This is done by manipulating the URL and browser history.

  Most modern JS frameworks offer an API for fronted-routing either directly or via a plugin.

4.2 Architecture

From the high level point of view the architecture of the RMI has the following modules 4.1:

- **Client** - The Client module is responsible for exchanging data with the server component. It can send and receive serialized rules and download the list of all saved rules, available operations, type definitions and type conversion rules.

- **Rule** - The Rule module contains all the rule logic including the logic of elements, variables, declarations, expressions and type conversions. It implements algorithms for finding visible variables in a block scope, checking variable over-shadowing, etc. Everything in this module can be serialized and de-serialized in order to be sent to the server component (or received).

- **Diagram** - The Diagram module is responsible for rendering the visual representation of a rule (or at least the part of it that can fit on the screen). It is also responsible for most of the interactivity including zoom, moving the rule in different directions, selection of elements, deletion, copying and pasting.

  In the future, if there is a requirement to completely change the visual representation of rules, it is sufficient to re-implement only this module and everything will work correctly.

- **GUI** - The GUI module is responsible for rendering an interactive GUI using UI components. The most important submodule is the Rule Editor, which includes components for editing the meta-data of concrete elements (e.g. their expressions, variables, etc).

- **Internationalization** - The Internationalization module manages translations and allows the other modules to display text in a chosen language. It is implemented by the *react-i18next* library.

  This module satisfies the requirement for internationalization of the prototype.

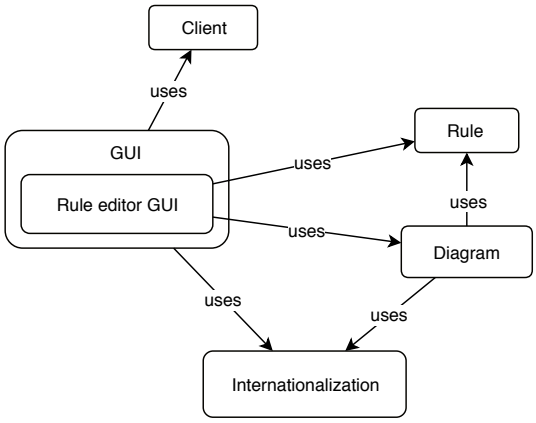


Figure 4.1: High level architecture of the RMI prototype

Legend: This is a usage view of the modules of the RMI prototype. A module uses another module if and only if its correctness depends on the correctness of the other module.

4.3 Choice of Programming Language

TypeScript is the optimal choice of programming language. It is a popular statically typed alternative to JS.

There are many potential alternatives to JS and all of them work by being transpiled to JS (mostly to ES5). However when choosing one, it is essential that it is statically typed.

Static typing is crucial, because it dramatically improves productivity thanks to intellisense. Additionally, it prevents type-related errors and it automatically servers as code documentation (because it describes the types of method arguments, attributes of interfaces, etc). For these reasons, statically typed languages are very beneficial for larger projects and the RMI definitely falls into that category.

There seem to be at least fourteen statically typed alternatives to JS[59]. The three most popular ones are the ones developed, used and maintained by some of the most notorious software companies:

- **TypeScript** - TS was created by Microsoft in 2012. It is a boosted version of ES8 with even more powerful language features (enums, access modifiers, interfaces, etc).

  There are IDEs that directly support it (e.g. Visual Studio Code) and offer intellisense and other static analysis tools for better coding experience.

  TS is probably much more popular that the alternatives, because it is the language used in Angular. It is reasonable to assume that JS programmers know Angular at least marginally, since it is one of the two most

popular JS frameworks, and therefore they should be acquainted with TS as well.

At Komix, an overwhelming majority of JS teams (including the ERAIN team) use Angular and therefore know TS. This is a very good reason to pick TS over Flow or Dart.

Otherwise, from the technical point of view, TS, Flow and Dart are comparable. They all have similar syntax and offer a similar variety of language constructs.

- **Flow** - Flow was created by Facebook in 2014. It is similar to TS in that it is also a boosted version of ES8[58] and is supported in Visual Studio Code (via a plugin).
- **Dart** - Dart was created by Google in 2011. It syntax also seems to be a superset of ES8 and again, it is supported in Visual Studio Code (via a plugin).

4.4 Choice of JS Framework

In order to satisfy the requirement for a component based UI, five of the most popular JS frameworks were considered and **React was chosen as the most suitable option**. This chapter provides detailed reasoning for this choice, which was based on the following criteria: current popularity, expected future popularity (if a framework does not have a good support team, it's development might stop abruptly), ease of development (IDE support - code highlighting, completion and static analysis), TypeScript support (this requires a CLI tool capable of transpilation) and the ability to satisfy requirement 2c (the framework must support loading of components in runtime).

JS frameworks facilitate the development of large JS applications. They solve common problems one of which is providing a mechanism for the definition of UI components using templates. Additionally, they bring many other benefits, such as pre-configured transpilation pipelines (which allow the programmer to leverage ES8 or TS), test runners, etc. Plus, by imposing structure on the code, they make it easier to understand to other programmers. All these are good reasons to use a JS framework instead of just a simple component definition library.

There are at least fifteen JS frameworks to choose from. Searching Google for *"top frontend frameworks"* and filtering out the most frequent names from the first thirty most relevant articles narrows the list down to six frameworks: React, Angular, Vue, Ember, Knockout and Meteor.

Meteor is actually not a front-end framework. It is a full-stack framework that uses React, Angular or Blaze for front-end.

Blaze was developed as part of Meteor and used to be the only front-end option for Meteor, but then the authors realized that *"React and Angular are more developed and have larger communities"*[60]. Therefore, Blaze can be dismissed in favor of React and Angular.

4.4.1 Choosing between React, Angular, Vue, Ember and Knockout

React, Angular, Vue, Ember and Knockout all offer a mechanism for defining UI components, are open-source, licensed under the MIT license, can be used with TS and offer the appropriate tooling and configurations for it. This makes all of them reasonable choices.

Filtering by Activity

The selection can be narrowed down by observing the activity on these projects (i.e. how often are the frameworks updated with bug fixes and new features).

The activity statistics during the last month 4.2 and last year 4.3 reveal that Angular, React and Ember have continuous support by at least three larger contributors and the activity is reasonably stable throughout the whole year. Vue and Knockout, on the other hand, have only a single contributor (the author of the framework) and the activity is sporadic throughout the year.

Indeed, these findings correspond nicely with the fact that Angular, React and Ember all have dedicated teams of people working on them, whereas Vue and Knockout do not.

This makes Angular, React and Ember preferable to Vue and Knockout. First of all, it is because they have a more stable support and second, if the author of Vue or Knockout decides to stop supporting the framework, then likely there is nobody to take over and that is a problem.

Choosing between React, Angular and Ember

React, Angular and Ember all have the same UI component philosophy and they all come with sufficient tooling. They are all reasonable choices.

UI components in these frameworks work in the following way. Components are composed of sub-components, they receive input parameters from their parent component and they can notify the parent component about custom events. It is also possible to return data to the parent component via these custom events. The lowest level of sub-components are classic HTML tags. 4.4

- **Angular** - Angular[2,3,4,5,6,7,8] was created in 2016 by Google as a successor of AngularJS. The latest version is Angular 6, which came out in 2018.

  - **Tools** - Angular comes with the following tools: a CLI for project initialization and simpler generation of new components, pre-configured testing environment for unit and end-to-end tests, string extractor for internationalized text.

    Syntax highlighting and static analysis for UI components is supported in Visual Studio Code IDE. A browser inspector extension for debugging UI components is available in Google Chrome.

    A myriad of plugins is available as well as several component styling libraries (Material-UI, Semantic-UI, Bootstrap, etc).

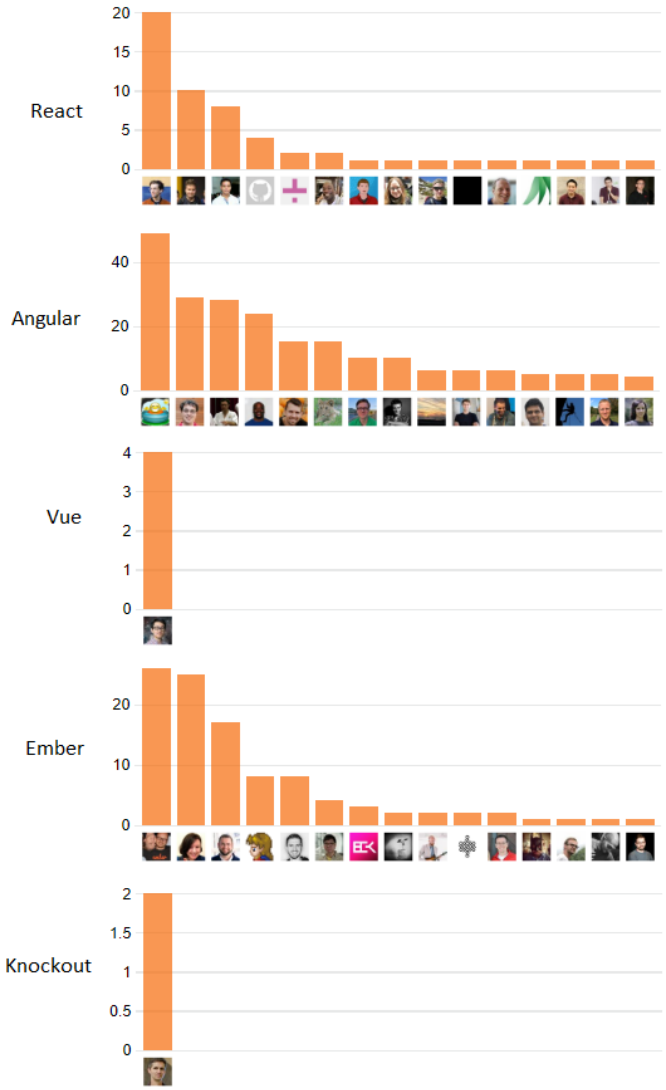


Figure 4.2: Frameworks - recent activity

These charts show the number of commits with improvements and bug fixes made to the five chosen frameworks during June 2018 (most recent at the time of writing).
React, Angular and Ember have between three to five significant contributors.
Vue and Knockout however, only have contributions from the single author of the framework. This is not an optimal situation, since if anything happens to the author or the author decides to stop supporting the framework, it is likely that there will be no more improvements and bug fixes and the framework will deteriorate.
Charts were taken from Github insights of each project.

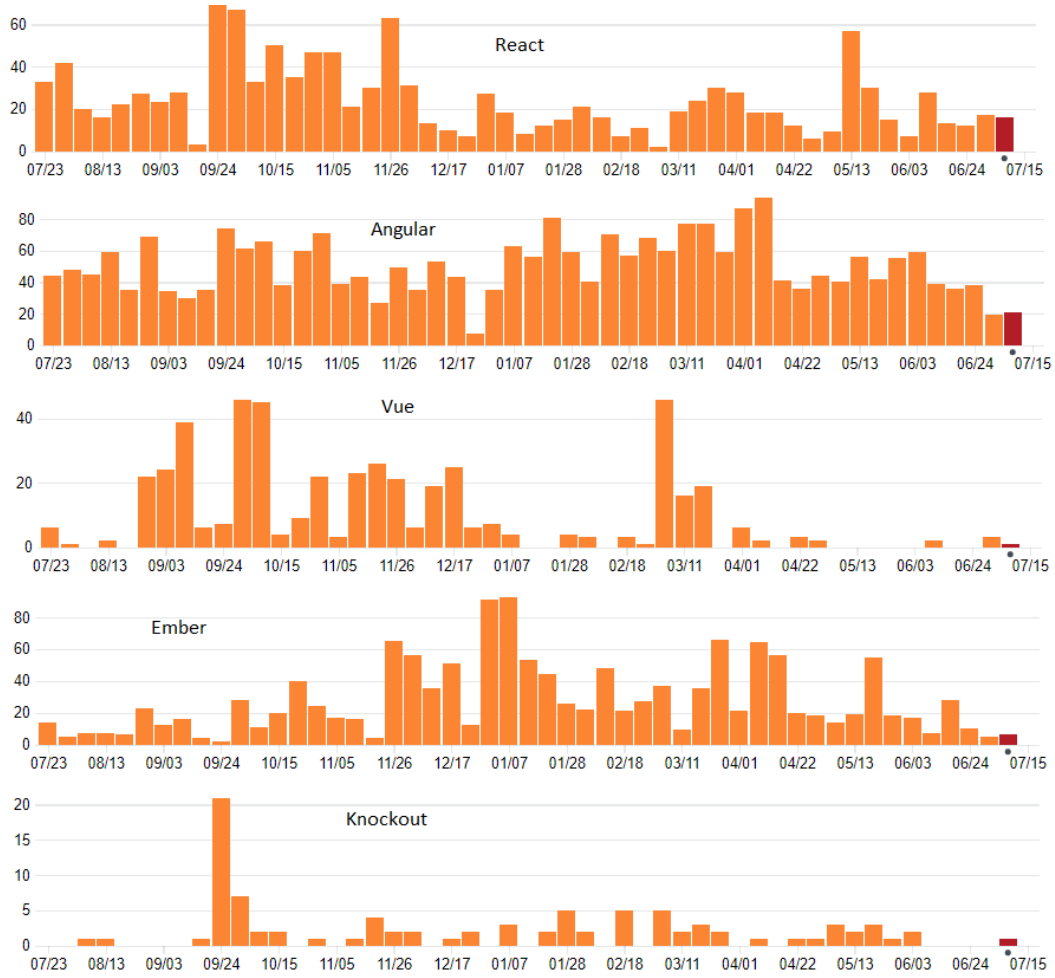


Figure 4.3: Frameworks - activity during the last year

These charts show the number of commits with improvements and bug fixes made to the five chosen frameworks from June 2017 through June 2018 (chunked by weeks).
React and Angular show stable activity.
Vue and Knockout show sporadic activity that seems to deteriorate during the last three months.
Charts were taken from Github insights of each project.

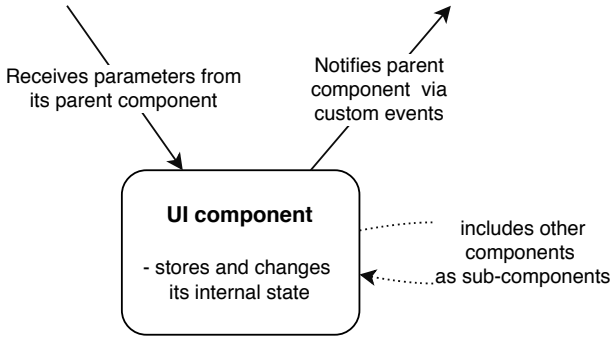


Figure 4.4: Architecture of UI Components in React, Angular and Ember

A component contains sub-components. It passes input parameters to them and can listen to their custom events.
In React parameters are called *Props*, in Angular they are *Input bindings*.
In React custom events are implemented by passing a callback function to the sub-component via props.
In Angular there is special API for custom events.

- **API** - Code is split into modules that contains UI components and services 4.7. Dependency injection is used for services and UI components.

  The API also offers front-end routing, an HTTP client and internationalization.

- **UI components** - Components are defined as TS classes with a template and styles supplied in an annotation. 4.5 4.6

  The template syntax offers structural directives (e.g. for, if, switch), one-way and two-way data binding (for display and input of values) and binding of event handlers (both for browser events and custom events).

  Components offer life-cycle hooks (i.e. it is possible to execute handler code when a component is mounted, unmounted, gets new input parameters, etc).

- **Notes**: *Angular is complicated and difficult to learn. Reading articles and guides about Angular is sometimes a nuisance because people tend to confuse Angular with AngularJS.*

- **React** - React[13,14,15] was created in 2013 by Facebook. The latest version is React 16, which came out in 2018.

  React on its own is just a UI component library. However, there are official plugins (e.g. React Router) and a CLI (Create React App) that can be added to React to make it comparable to Angular with respect to both tooling and API. The following text assumes the use of these add-ons.

  - **Tools** - React comes with the following tools: a CLI for project initialization and a pre-configured testing environment for unit tests.

    Syntax highlighting and static analysis for UI components is supported in Visual Studio Code IDE. A browser inspector extension for debugging UI components is available in Google Chrome.

    A myriad of plugins is available as well as several component styling libraries (Material-UI, Semantic-UI, Bootstrap, Antd, etc).

  - **API** - There are plugins for frontend-routing and internationalization.

  - **UI components** - Components are defined as ES8 classes with special JSX syntax for templates, which is a combination of JS and HTML and it gets transpiled to JS. 4.8

    JSX must be returned from the *render()* method, but otherwise it can be used anywhere (e.g. stored in a variable), which is useful. The template syntax offers one-way data binding and binding of event handlers.

    Components have *state* and when it changes, the component gets re-rendered. They also offer life-cycle hooks.

- **Ember** - Ember[30] was created in 2011. The latest version is Ember 3, which came out in 2018.

```
@Component({
    selector: 'app-hello',
    template: `
        <div>Hello {{name}}</div>                    1
        <button (click)="sayHi"></button>
    `,                                               2
    style: `div {
        color: red;                                  3
    }`,
    providers:  [ MessageService ]                   4
})
export class HelloComponent {
    @Input name: string;

    constructor(public messages: MessageService) {
        messages.setName(this.name);
    }

    sayHi(): void {
        alert(this.messages.getHiMessage());
    }
}

@Component({
    selector: 'app-welcome',
    template: `
        <div>
            <h1>Welcome</h1>
            <app-hello [name]="Joe"/>                5
        </div>                                       6
    `
})
export class WelcomeComponent {}
```

Figure 4.5: Angular API - Example UI Components

There are two components app-hello and app-welcome, the latter uses the former. Both have their template and style defined inside an annotation (a.k.a. a *decorator*), which is special Angular syntax and it gets transpiled to JS. Decorators are used heavily in Angular.
(1) One-way data binding, which displays the name.
(2) The click event handler is registered.
(3) Style is specified in the decorator and it applies only locally to this component.
(4) The component specifies that a MessageService service should be injected into it via DI.
(5) The app-welcome component uses the app-hello component in its template.
(6) The app-welcome component passes a name as an input parameter to the app-hello component.
(The example continues in figure 4.6.)

```
@Injectable()    1
export class MessageService {
    private name = '';
    setName(name: string): void {
        this.name = name;
    }
    getHiMessage(): string {
        return 'Hi';
    }
}

@NgModule({
    declarations: [ HelloComponent, WelcomeComponent ], 2
    providers: [ MessageService ], 3
    // imports: [],
    // exports: []
})
export class HelloModule { }
```

Figure 4.6: Angular API - Example Service and a Module

There is an Angular service and a module with all its components and services registered in its decorator.
(1) The message service is normal TS class with the *@Injectable* decorator, which signifies that it is a service and can be injected into components and other services via DI.
(2) All components used in a module are registered in its decorator.
(3) All services used in a module are registered in its decorator.
Angular uses the list of component and services to make Dependency Injection work.

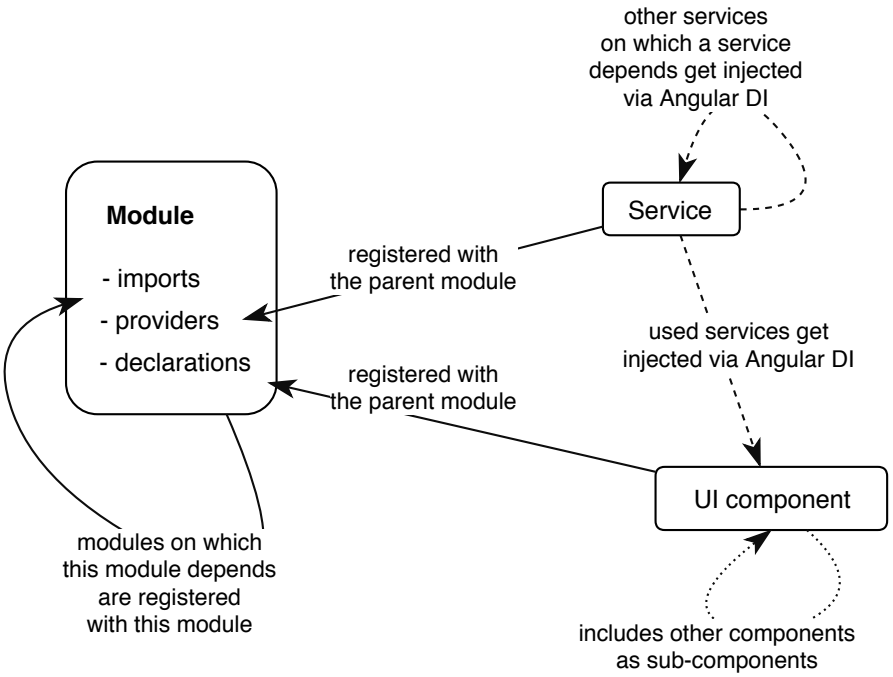


Figure 4.7: Angular API - Components, Modules and Services

Angular has rather strict rules about the architecture of the application. The application is split into modules. Modules can be purely logical or they can be a collection of UI components.
In the latter case, all UI components that are used in a module must be registered with it.
Services implement logic and data manipulation. Components are supposed to just display data gathered from services or pass user input to services for processing. Again, all services used in a modules must be registered with it. Angular instantiates all services of a module as singletons and injects them via DI into the components that use them.

```
interface IProps {
    name: string;
}
interface IState {
    wasGreeted: boolean;
}
class Hello extends React.Component<IProps, IState> {
    constructor(props: any) {
        super(props);
        this.state = {
            wasGreeted: false
        };
        this.sayHi = this.sayHi.bind(this);
    }

    sayHi(): void {
        if (!this.state.wasGreeted) {
            alert('Hi ' + this.props.name);
            this.setState({ wasGreeted: true });
        }
    }

    render() {
        return (
            <div>
                <h2 style={{ color: 'red' }}>Hello</h2>
                <button onClick={this.sayHi}>Say hi</button>
            </div>
        );
    }
}

class Welcome extends React.Component<IProps, IState> {
    render() {
        const name = "John";
        return (
            <div>
                <h1>Welcome</h1>
                <Hello name={name}/>
            </div>
        );
    }
}
```

Figure 4.8: Example of React Component Syntax

There are two components *Hello* and *Welcome*, the latter uses the former as a sub-component.
(1) The Hello component declares that it must receive a name as input (via *props*).
(2) The Hello component stores information about greeting as its state.
(3) The Hello component initializes its state.
(4) The Hello component returns its template in the *render()* method.
(5) Styling is passed via props and it applies only locally, only to the component in question.
(6) The click event handler is registered.
(7) The Welcome component uses the Hello component in its template.
(8) The Welcome component passes a name as a prop to the Hello component.

– **Tools** - Ember comes with the following tools: a CLI for project initialization and a pre-configured testing environment for unit tests and end-to-end tests.

  Syntax highlighting and static analysis for UI components is supported in Visual Studio Code IDE. A browser inspector extension for debugging UI components is available in Google Chrome.

  A myriad of plugins is available as well as several component styling libraries (Material-UI, Semantic-UI, etc).

– **API** - Code is divided into UI components, *models* for storing data and *controllers* for adding functionality to models. Everything works in conjunction with front-end routing.

– **UI components** - Components are defined as ES5 objects.

  The template syntax offers structural directives, one-way data binding and binding of event handlers.

  Components offer life-cycle hooks.

There are two advantages of Angular and React over Ember. First, Angular and React both have paid dedicated teams maintaining them, whereas Ember has only an unpaid steering committee composed of volunteers. Second, Komix only uses Angular and React organization-wide. From this perspective, it is a good idea to stick to these two frameworks, especially when they are reasonably suitable for the task.

Angular is actually preferred, because the ERAIN team specifically is trained in Angular and not React. However, as described in section 4.6, React is much more suitable for the implementation of the customization mechanism, which is one of the quality requirements. This makes React the optimal choice.

4.5 Rule Browsing, Rendering and Interaction

Several libraries and options were considered for the implementation of the Diagram module, which is the module responsible for the interactive visual representation of a rule.

A low level SVG library was chosen. SVG makes the solution testable. A low level library is preferred over a high level diagram library among other things because it is more versatile and easier to learn by other programmers.

The requirement for fast rendering of large rules will be handled by a combination of using a custom layout algorithm and rendering only a part of the rule (only the part that can actually fit into the browser window).

4.5.1 Visual Representation

Rules will be visually represented as flowchart diagrams.

Since rules represent business logic *algorithms*, this representation makes perfect sense. In fact, the visual representation of rules in the current RMI resembles a flowchart diagram to a large degree 4.9.

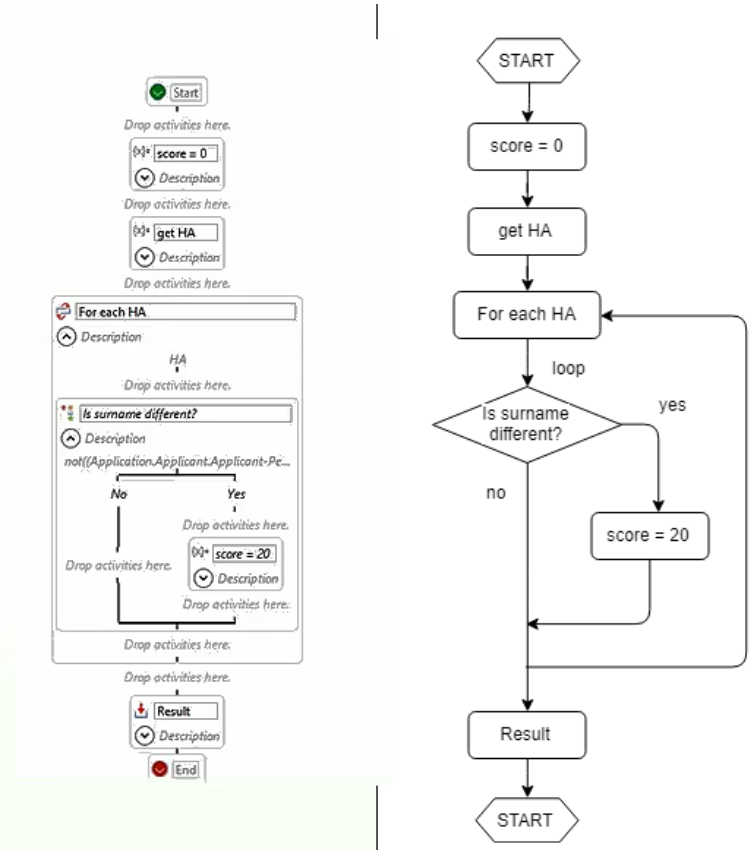


Figure 4.9: Visual representation

On the left is the visual representation of rules in the current RMI, it is basically a combination of a flowchart diagram and nested windows.
On the right is the same rule redrawn as a classic flowchart diagram with arrows and without the nested windows. This is the representation that will be used in the prototype (i.e. nodes and arrows with labels).

4.5.2 Choice of the Underlying Technology

There are three possible ways of drawing a flowchart diagram in a web browser - using HTML, the canvas element or SVG. Out of these three, SVG is definitely the best choice.

Even though diagram libraries based on HTML do exist, HTML was never intended for graphics and it is impossible to use it for drawing custom shapes. This is why all serious diagram libraries leverage SVG or the canvas element.

The canvas element allows drawing of arbitrary custom shapes, however it lacks many of the perks of SVG.

SVG elements are part of the DOM, therefore they can listen to DOM events and crucially, the testing methods developed for HTML can be applied to SVG as well (e.g. end-to-end testing via Selenium). In addition, it is possible to apply geometric transformations to SVG via CSS, which could easily be leveraged to implement zoom and panning.

Even though the canvas element may be faster performing, SVG is clearly a better choice for an interactive diagram.

4.5.3 Layout Solution

The visual layout of elements and sub-rules of a rule will be computed by a custom algorithm, because the available libraries are too generic and too slow.

There seem to be no libraries for calculating the layout of a flowchart diagram. The next closest solution to the layout problem is to use a generic mathematical graph layout library and use a *layered algorithm*.

Graph layout libraries take a set of nodes and edges as input and calculate their positions. They usually offer a variety of different layout algorithms and the algorithm that produces a result best resembling a flowchart diagram is usually called a *layered algorithm*. This algorithm chunks the graph into layers by traversing it in a breath-first manner and returns a layout where all nodes from the first layer are at the top, below them is the second layer and so on.

Dagre[51] and ElkJS[52] are two such libraries that offer a layered algorithm and there do not seem to be many others. However, the problem with Dagre and ElkJS is that they do not take into account edge priority. This results in inconsistent layouts of rules with conditional statement elements where the *true* and *false* branches sometimes swap sides, which is a problem.

Also, it takes ElkJS up to two seconds to calculate the layout of a graph with only a hundred nodes and a hundred edges, which is too slow.

It is reasonable to assume that other potential graph layout libraries also suffer from the lack of edge priority. Therefore it is best to calculate the layout using a custom algorithm.

Custom Layout Algorithm

The custom layout algorithm calculates the layout of elements based on a grid. Elements of a rule or sub-rules are placed into subsequent squares in the grid going from top to bottom. 4.10

Sub-rules are placed right below the elements that they belong to. If there are multiple sub-rules, then they are placed next to each other horizontally from left to right. If there is only a single sub-rule, one additional column is reserved on the right for a loop-back edge. On row right below the sub-rules of an element is reserved so that their edges can be reconnected.

This algorithm is independent of concrete element types, it only depends on the knowledge of rules, sub-rules and abstract elements. Therefore, it is possible to add new element types to the RMI without changing this algorithm.

The resulting elements positions are used to render a flowchart diagram representation of a rule. 4.11

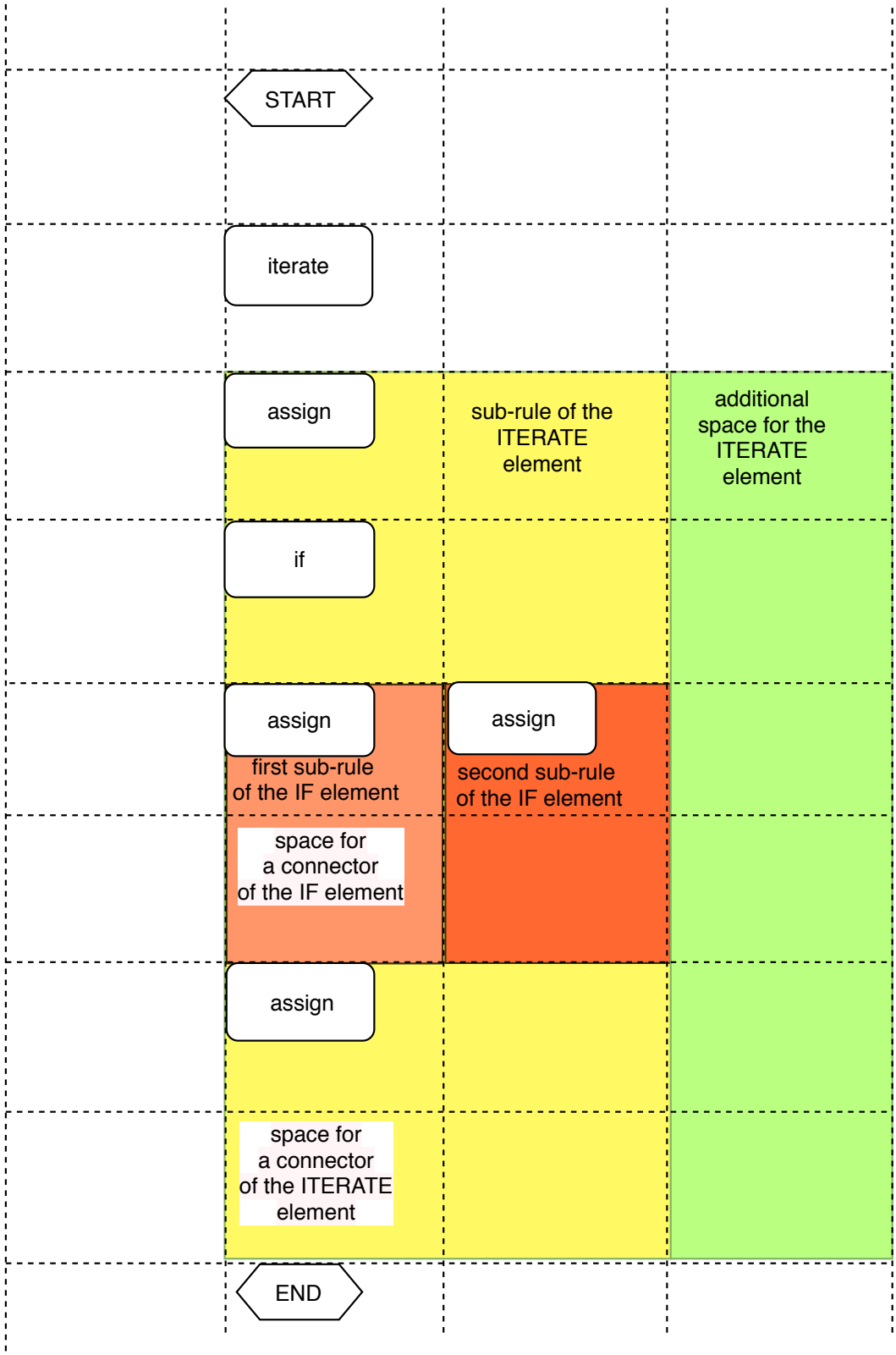


Figure 4.10: Custom layout algorithm

The custom layout algorithm positions elements into squares of a grid. The positions of elements in sub-rules are calculated recursively. Multiple sub-rules are laid out next to each other and space is reserved for connector spaces and loop-back edges.

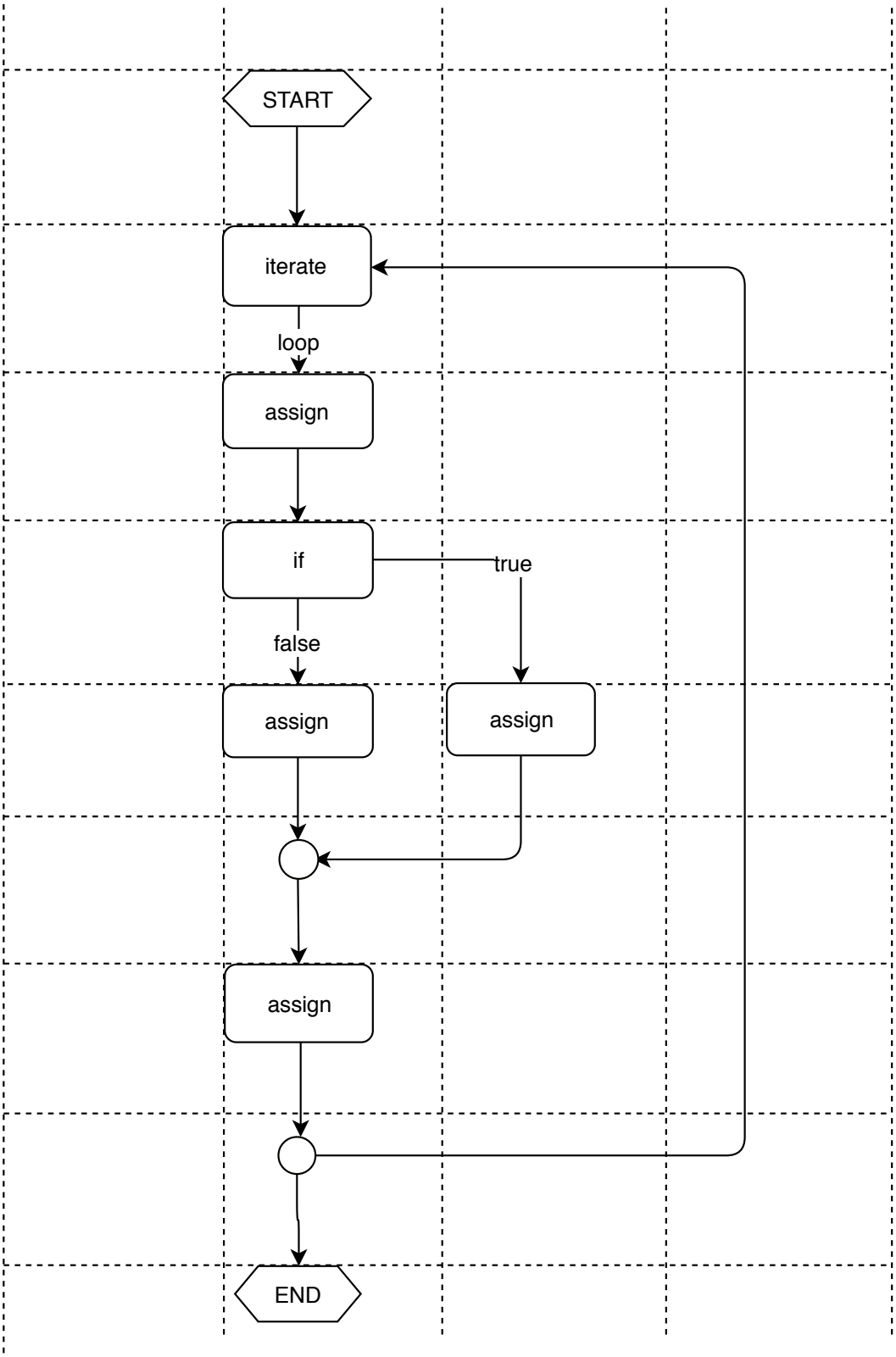


Figure 4.11: Rule rendering concept

This is a concept showing how the custom layout algorithm can be leveraged produce a visual flowchart representation of a rule. This rendering depends on the knowledge of concrete element types unlike the layout algorithm.
At the top, sub-rules are connected to their parent element by labeled arrows. At the bottom, their arrows are merged into a connector spot. The reserved row next to the sub-rule of an iteration element is used to render the loop-back arrow.

4.5.4 Choice of a Diagram Library

It would be useful to use a diagram library for the implementation of the flowchart diagram. Ideally, it would save time and testing, and it would make the prototype more future-proof thanks to the extra functionality that libraries offer. However, the two most promising libraries (JointJS and MxGraph) have several issues, which make them a liability. In the end, using pure SVG turns out to be more optimal.

In order to avoid potential problems with browser incompatibilities the Snap.svg[53] library will be used. It is a simple wrapper of SVG functionality and it was chosen because it is popular (however, any other simple SVG library could be used just as well).

There are quite a few diagram libraries of different kinds. Twelve of those successful enough to be mentioned in an article[32,33,34,35,36] were considered. JointJS was the most often mentioned free library.

JointJs

JointJS[47] would be a good candidate. It has a great documentation and demos, it is very popular (6K downloads monthly) and it has a large community mostly active on Google Groups.

To describe it briefly, it is SVG-based, it supports serialization to/from JSON, allows the programmer to listen to events, gives access to the SVG properties of any vertex or edge, supports translation and zoom, has automatic routing of edges (i.e. finding the most visually pleasing path for an edge), has algorithms for automatic layout of vertices, all vertices are draggable, clickable, etc.

The fact that it spans 16K lines of code, plus it uses powerful libraries - Backbone, jQuery, Lodash and Dagre - confirms that it has a lot of useful functionality.

However, there are five big issues with JointJS:

- Even though it has 4K lines of typings, they are so incomplete and bugged that even the first example among official demos fails to compile with a Typescript compiler.

  Even though this could be solved by writing a custom typings file, it is not a good thing.

- It should be possible to create custom SVG shapes using JointJS, but the documentation is lacking completely.

  Based on some unofficial examples it seems to be possible at least partially, but the code is very complicated.

- Listening to all DOM events that happen on the SVG elements is not possible. JointJS gives access only to a subset of the events.

- The learning curve is steep - it takes a few days to understand JointJs and learn its API. This is not optimal, code should be easily understandable to other programmers if possible.

This learning curve is a result of the fact that it is a large library and it also uses the Backbone library[20] and mixes its API into the API of JointJS. Therefore, more advanced code also requires the knowledge of Backbone.

The negative aspects of JointJS outweigh its positives, especially its learning curve is a problem. This is the reason why a low level SVG library was chosen instead.

MxGraph

MxGraph[42] is diagram library with functionality comparable to JointJS. The popular diagram editing application Draw.io[50] is written using MxGraph, which is a proof of its capabilities. However, it has substantial problems with bugs and bad documentation, which renders it unsuitable for use.

At the time of writing there was a bug which prevented the installation of the library altogether[44]. It took the authors several months, to fix it, but the fixed version (21. 3. 2018) prevented the library from being imported and used in code, which is not a big improvement.

This inability of the authors to publish a version of the library that can be installed and used without first hacking it is definitely not a good sign.

In addition, the API[43] and documentation is approximately half as good as that of JointJS and examples of code are missing.

An Overview of Other Dismissed Libraries

The following are the other ten libraries that were found to be unfitting either because they are paid, are not based on SVG or do not support interactivity (events):

- **GoJS**[41] - paid (22K CZK)
- **Rappid**[40] - paid (40K CZK)
- **Mindfusion JavaScript Diagram Library**[39] - paid (10K CZK)
- **JsPlumb**[45] - paid (70K CZK)
- **Vis.js Network**[49] - it uses the canvas element, not SVG
- **Mermaid** - completely static, no events
- **FlowChart** - completely static, no events
- **JS Sequence Diagrams** - completely static, no events, only supports UML sequence diagrams
- **Dracula Graph Library**[37] - virtually no documentation at all, no events
- **Js-graph.it**[38] - based on pure HTML, doesn't have almost any functionality

It should be noted that diagram libraries should not be mistaken for graph theory libraries, which are similar, but they focus on visualization of large amounts data. In literature (articles), diagram and graph theory libraries often get thrown into the same category.

Such libraries usually display vertices as small dots, offer automated layout algorithms (e.g. force-directed layout), etc., but they do not focus on the interactivity that is needed for this prototype. Examples of such libraries are Cytoscape[46] and SigmaJS.

Graph theory libraries were not considered, because they do not fit the purpose.

4.5.5 Optimizing Rendering for Speed

A simple solution for rendering a rule would be to take an empty SVG element of a fixed size and render all of the elements and edges of the rule in it. This way all the SVG sub-elements would be added to the DOM, but only the ones with coordinates inside the parent SVG element would be visible. This solution is clean and it makes it very easy to implement exporting as an image, browsing and zooming (browsing and zooming could be implemented very simply via CSS geometric transformations).

However, even though only a lightweight SVG library is used, this solution is still too slow to satisfy the performance requirements when browsing large rules. A more sophisticated and complicated approach has to be taken.

Benchmark: *The Snap.svg library takes on average 1 second to render 1K relatively complex elements (composed of an SVG rectangle, shadow, text and an arrow) 4.12. If the graphical representation of a rule becomes more detailed or the application is run on a slower machine, the library will not be able to satisfy the quality requirement for rendering under 1 second.* *(The reproducible benchmark can be found in Attachment 3. This benchmark was measured on the same computer as specified in quality requirements.)*

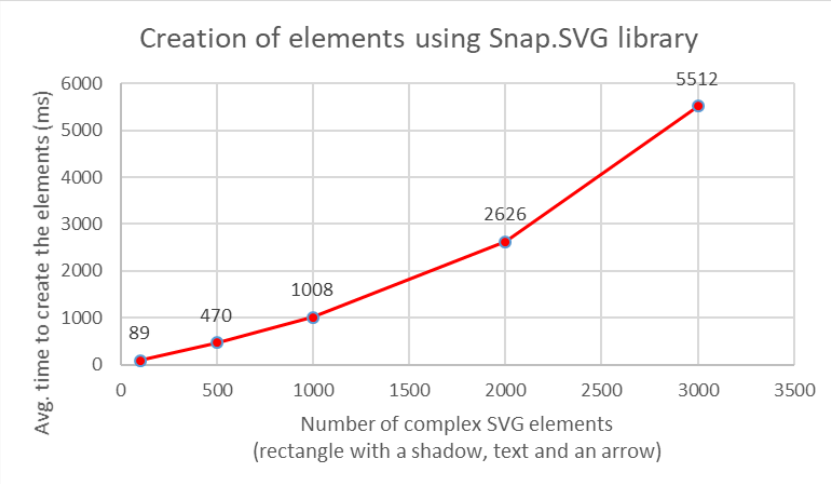


Figure 4.12: Speed of Snap.svg library

The speed of Snap.svg library is measured when creating visual representations of business rule elements (this equates approximately to one rectangle, some text, a shadow and a an arrow). The time complexity function is linear and becomes steeper with extreme number of nodes, which can probably be attributed to DOM behavior, but it is irrelevant to this application. This behavior is probably caused by DOM. The library does not meet the quality requirement for rendering 1000 elemens in under 1 s.

The solution is to use a view-port object 4.13 representing a cut-out of the area of a rule which is currently visible on the screen, and only rendering the elements and edges that are inside that area or cross that area.

The layout still has to be calculated for the whole rule in order to find out which elements fit into the view-port. Luckily though, calculating the layout using the custom algorithm is very fast.

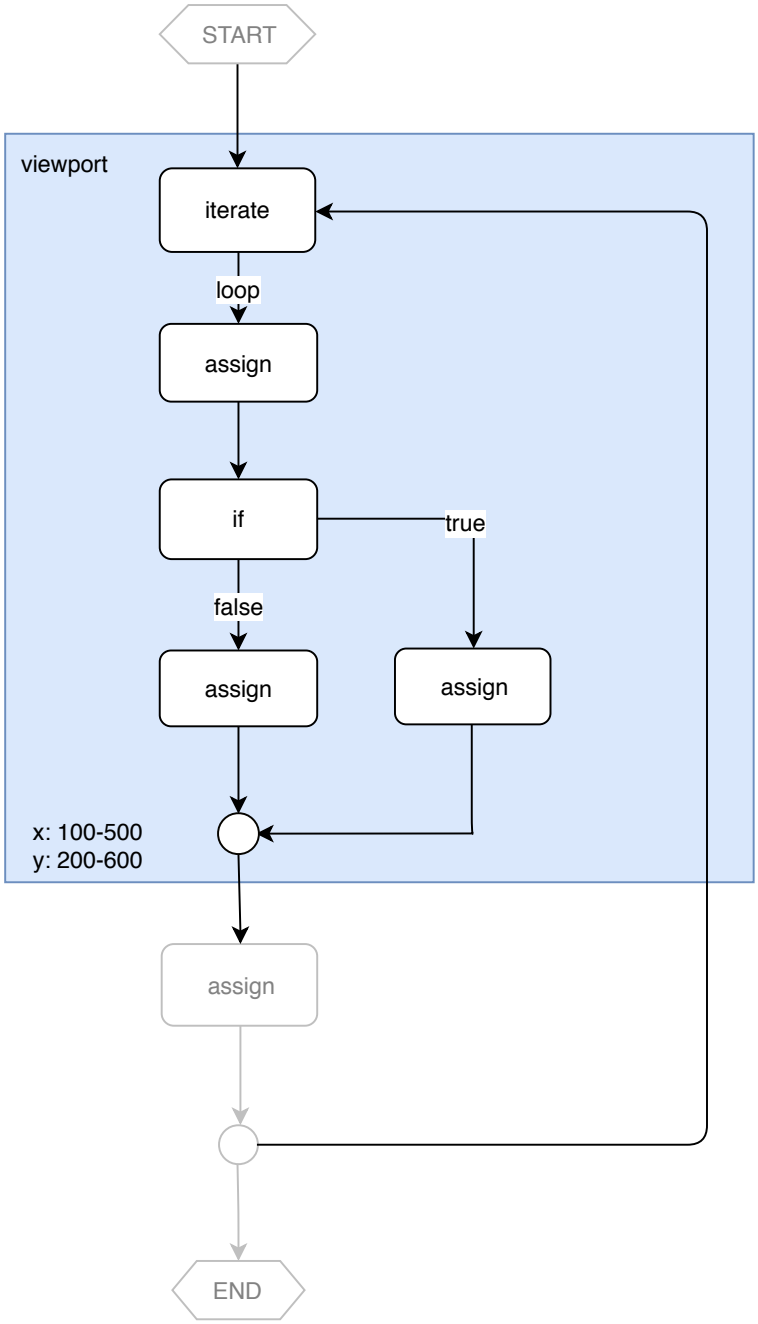


Figure 4.13: Partial rendering for browsing of a rule

A view-port object specifies the ranges of coordinates visible on the screen. The layout of the whole rule is computed and only the elements inside the view-port and the edges crossing the view-port are actually rendered.

Benchmark: *Calculating the layout of a rule consisting of 1K elements takes on average 2 ms. 4.14* *(The reproducible benchmark can be found in Attachment 3. This benchmark was measured on the same computer as specified in quality requirements.)*

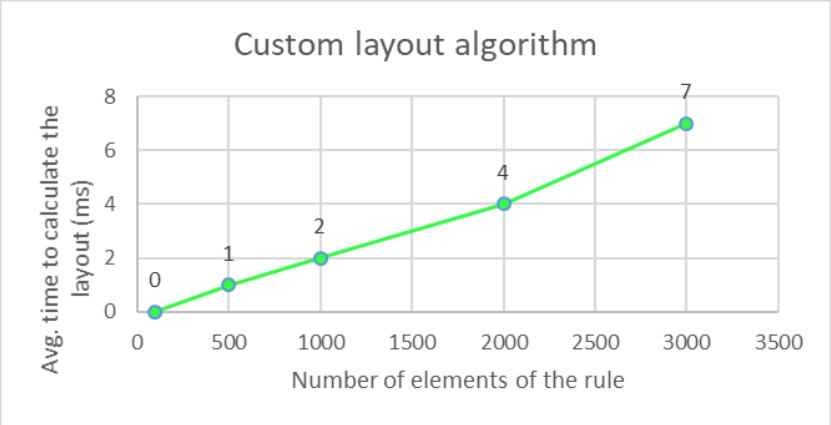


Figure 4.14: Speed of the layout algorithm

The time complexity function is linear and the layout is calculated extremely fast.

The consequences of this partial rendering approach are that a rule has to be re-rendered each time it is moved or zoomed. Also browsing and zooming can no longer be implemented via CSS transformations, because there are no hidden elements outside the visible area of the SVG element.

4.6 Customization Mechanism

To satisfy the requirement for a 2-step configuration-based loading of customized versions of the RMI, solutions in React and Angular were considered. Even though some sort of a customization mechanism can be designed in both cases, React is much more suitable for this task.

The configuration-based loading consists of two steps - loading of customizations and their integration into the already existing code.

4.6.1 Loading of Customizations

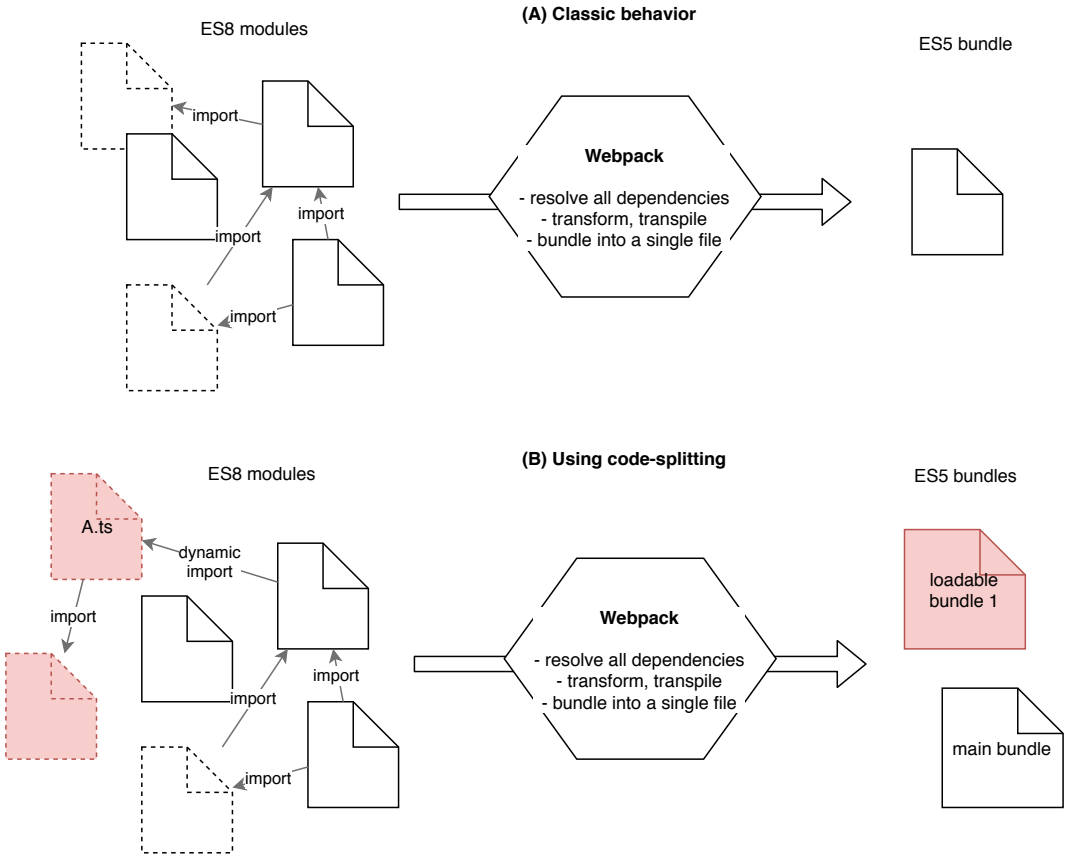
Both Angular and React deploy their code by using Webpack to bundle all the code after transpilation into a single ES5 file.

Luckily, Webpack offers a code-splitting feature (enabled in both Angular and React), which enables bundling the code into multiple ES5 files. One of them is the main file and the others are lazily loaded as Promises. 4.15

Loading of code-splits can be successfully used to load customizations (custom pieces of code - classes, GUI components, methods, etc).

4.6.2 Integration of Customizations into Existing Code

The customizations (i.e. the code that implements custom functionality of a concrete customer) can be either pure TS code or it can be UI components written




```
// dynamic import syntax
const loadedCodeSplit = await import ('./A.ts');
```

Figure 4.15: Webpack bundling

(A) The normal operation of Webpack is that it finds all the files needed by the application by following the dependency tree, applies transformations to them (such as transpiling ES8 to ES5) and bundles the result into a single JS file.
(B) Webpack also offers code-splitting, which bundles the code in the same way, but into multiple code splits. Code-splitting is based on **dynamic imports**, which are module imports that are executed in runtime, e.g. inside of a function call (as opposend to classic imports, which are found at the top of a file). *In this example a TypeScript modules A.ts is imported dynamically, therefore it and all its dependencies get bundled into a separate loadable bundle.*
Dynamic imports are transpiled to ES5 code, which when executed loads the loadable bundle via AJAX as a Promise.

in the syntax of the used JS framework.

Pure TS code can be integrated into existing code easily (e.g. a class can be instantiated, a function can be called, etc).

However, integrating UI components into existing code (i.e. adding them as sub-components of already existing components) may or may not be possible depending on the concrete JS framework and how it operates. In order for the integration to be possible, it is necessary that the chosen JS framework supports instantiating a *previously unknown* UI component in runtime. *This shall be termed 'dynamic component instantiation' for the purposes of this thesis for lack of an official term in literature.*

Definition: *Dynamic component instantiation* is the act of instantiating a UI component whose type is not known by the application during startup and it only becomes known later on in runtime.

Note: Dynamic component instantiation should not be confused with classic component instantiation, which is the instantiation of a previously ***known*** *type of component at runtime. This is a very common ability of probably any modern JS framework including Angular, React and Vue.*

Dynamic Component Instantiation in React

In React dynamic component instantiation is supported thanks to two details of its component syntax. The first is that JSX allows tag names to be supplied as variables 4.16. The second is that templates are returned from the *render()* method, which makes it possible to conditionally change parts of the template.

This makes the integration of customizations in React simple.

This behavior of JSX is a side-effect of the fact that JSX tags work by getting transpiled into JS function calls 4.17.

Dynamic Component Instantiation in Angular

Angular is more complex than React. All Angular UI components must be contained in Angular modules and data and logic should be handled by Angular services, which are injected (via Angular DI) into the UI components that depend on them.

Angular requires that all services and UI components inside a module must be declared when the module is instantiated. This makes dynamic component instantiation impossible.

Note: Confusingly enough, Angular offers a so called *ComponentFactoryResolver* whose purpose is *"to add components dynamically"*[62]. However, in fact it cannot be used for dynamic component instantiation because it only works for previously known component types. (It requires that all the dynamically instantiated components are registered with a parent module).

4.6.3 The React Solution

A full customization mechanism can be implemented in React using a Customizations class, Webpack code-splitting and dynamic component instantiation.

```
interface IProps {}
interface IState {}
class Hello extends React.Component<IProps, IState> {
    constructor(props: any) {
        super(props);
        this.state = {};
    }

    render() {
        return (<div>Hello</div>);
    }
}

class NormalInstantiation extends React.Component<IProps, IState> {
    constructor(props: any) {
        super(props);
        this.state = {};
    }

    render() {
        return (<Hello/>);
    }
}

class DynamicInstantiation extends React.Component<IProps, IState> {
    constructor(props: any) {
        super(props);
        this.state = {};
    }

    render() {
        const ComponentClass = Hello; 1
        return (<ComponentClass />); 2
    }
}
```

Figure 4.16: Dynamic component instantiation in React

It is possible to pass tag name as a variable.

For example, this code written with JSX:

```
  class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

can be compiled to this code that does not use JSX:

```
  class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello
${this.props.toWhat}`);
  }
}

ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
);
```

Figure 4.17: An example of transpiled JSX

This example was taken from the official React docs[61].

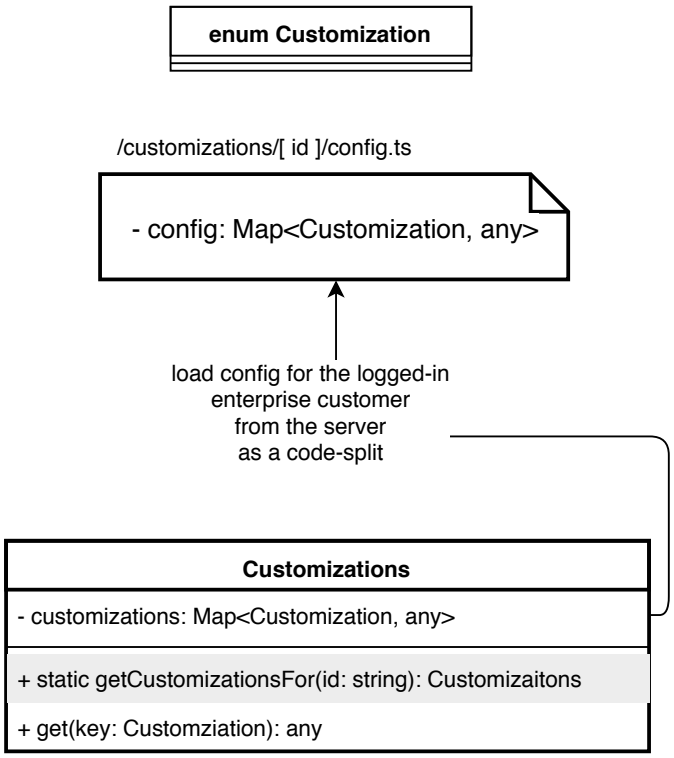


Figure 4.18: Customizations class - the loading phase

The Customizations class holds a map of all customizations of the currently logged in user. The map of customizations gets downloaded as a code-split after the user authenticates.

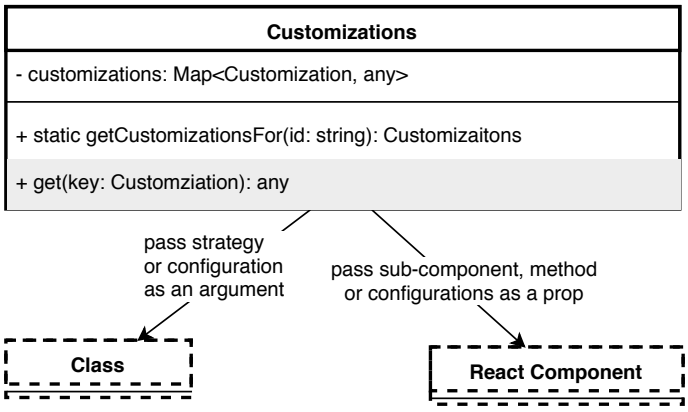


Figure 4.19: Customizations class - the integration phase

The existing code (Classes and React components) gets customizations form the Customizations class and uses them (integrates them).

After the user gets authenticated, the Customizations class loads their customizations via a code-split 4.18. Afterwards, the existing code gets the customizations from the Customizations class and integrates them into the places where they belong 4.19.

Note: This solution is inspired by the ***feature toggles***[21] *design pattern, which is used in A/B testing. However, unlike toggled features, the customizations are not meant to be temporary.*

Implementation and usage of this customization mechanism can be found in attachment 1.

4.6.4 The Angular Solution (Suboptimal)

Even though dynamic instantiation of Angular components is not possible, there is an alternative solution, but it is unnecessarily complicated, not flexible and has other drawbackss.

The Technical Idea

Angular offers a mechanism for lazy loading routed modules called Lazy Loaded Feature Modules.

Lazy Loaded Feature Modules[24] are a feature of Angular that is meant to be used in SPAs that contain multiple views that are navigated via front-end routing (changing URL in the browser). In such a scenario, it becomes reasonable to postpone the loading of some views until they are actually needed (until they are routed to), in order to speed up the initial loading of the application. In this case the view is a complex UI component (consisting of sub-components and services) and it is called a *feature module* in Angular documentation.

The router contains a list of routes and their corresponding feature modules, which are instantiated and rendered when the route is gets routed to. The routed modules may be configured to be loaded lazily or not.

In order to implement loading of *previously unknown* modules, it is necessary to clear and re-set the list of front-end routes dynamically in runtime. It should be noted that this is a very uncommon way of working with front-end routes.

This way the integration of customizations in Angular can technically be implemented, but it is not a good practice.

The Solution

Lazy loaded feature modules combined with re-configuration of routes can be leveraged to implement a solution for customized versions in the following way 4.20:

1. Implement customizations as feature modules.
2. Map the customizations to unique routes.
3. After the user authenticates, download the list of routes and customizations via a code-split. Then reconfigure the router to use them, thus integrating the customizations into the existing code.

The Drawbacks

This solution has several drawbacks:

1. It uses routing for something it is not intended for. This makes the architecture confusing and hard to understand to other programmers.

2. It forces the prototype to use routing, even though it may not need it. (Each customized part of the UI has to be accessed via a route.)

3. It does not work well for small customizations of large modules. It results in a lot of file duplication, because the whole hierarchy of parent components has to be duplicated due to changed imports 4.21.

4. There is also a considerable problem with sharing services (and therefore sharing data) among lazily loaded modules.

   By default lazy-loaded feature modules cannot share services. (If they declare the same service, each will have one dedicated instance).

   The only way to overcome this behavior is to register the services in a parent module common to both of the lazy-loaded modules.[63] However, this is discouraged for two reasons:

   (a) it is analogous to writing code dependent on a global variable, which is a very bad practice

   (b) the module is no longer self-sufficient (it depends on an external service), which is a very bad practice

An example implementation of this customization mechanism can be found in attachment 2.

4.7 Testing Tools

As already mentioned, React comes with a zero-configuration test runner, which supports TS. It is called *Jest* and same as React, it is developed and maintained by a paid dedicated team at Facebook. The API, developer experience and speed of Jest are good enough that there is no need to look for an alternative.

Enzyme shall be used as the UI component testing library and Cypress for end-to-end testing.

4.7.1 Jest

Jest is not only a test runner. It also includes a testing library with an API very similar to Jasmine (a popular testing library).

The API covers everything that is expected from a testing library. This includes:

- tests and test suites
- assertions of identity and deep equality (i.e. equality of JS objects and arrays)

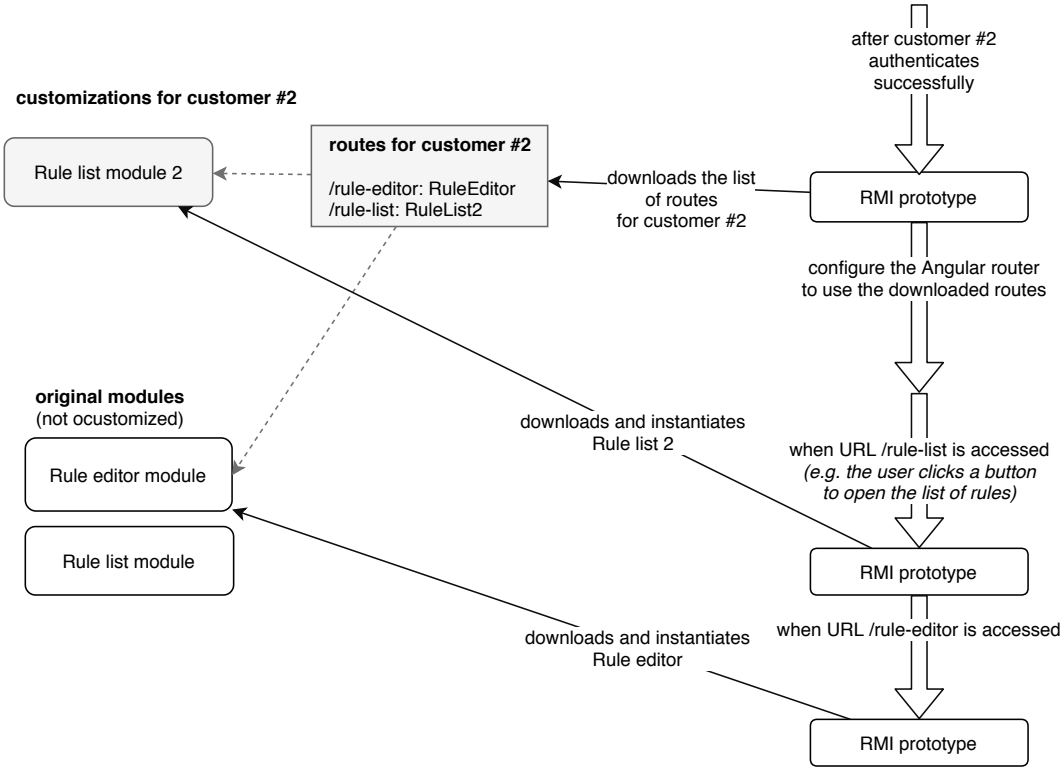


Figure 4.20: A possible customization mechanism in Angular

This example shows how the proposed customization mechanism works for customer #2. The RMI prototype contains the Rule Editor module and the Rule List module. Customer #2 needs some special functionality from the Rule List module, therefore a custom version of that module was created for them. A new route leading to that module was also added so that the module can be lazily loaded when it is routed to. This way customer #2 now receives the customized Rule List 2 module, whereas the other customers receive the original Rule List module.

It is important to note that a side-effect of this mechanism is that a new module and a route has to be created per each customization. (Unless multiple customizations fall inside the same module.) This is one of the several drawbacks of this solution.

- assertions for thrown exceptions
- asynchronously executed assertions for testing asynchronous code
- setup and tear-down functions
- a mocking API

The mocking API consists of *spies* (or spy functions), which can mock a method of an instantiated JS class and observe how many times it was called, with what arguments, etc. A spy can also be used to return a different value from the mocked method.

Jest achieves high execution speed by running tests in parallel.

Jest has several features that improve developer experience. It has a watch mode, which watches changes to files and reruns only those tests that have changed or their dependencies have changed. It has easy to understand error messages when a test fails (e.g. it shows the difference of two compared JS objects, etc). It includes a test coverage tool. It can run tests selectively and it runs failing tests first. Additionally, it has a plugin for integration with Visual Studio Code IDE.

One thing is that may be seen as a disadvantage is that Jest is Node.js based, i.e. it does not run the tests inside a real web browser (such as Google Chrome). This architecture allows Jest to run faster than it would using a real web browser, but it comes at the cost of not being able to test browser-specific features.

4.7.2 TS Testing with Jest

All testing of TS classes and functions can be done with Jest. There is not need to install any additional assertion libraries.

(More details about Jest API can be found in attachment 5.)

4.7.3 UI Component Testing with Jest

Enzyme library is used for testing React UI components.

There are several libraries for testing React components, but Enzyme is recommended by the authors of React. Enzyme is developed and maintained by Airbnb. It instantiates React components and offers a simple API for their manipulation, traversal and the simulation of user events.

Enzyme offers three ways of instantiating React components:

1. Shallow instantiation creates and instance of the component and keeps track of sub-components and their props. However, it does not instantiate the sub-components. This allows unit testing of React components, because the tests do not depend on the functionality of sub-components.

2. Full instantiation instantiates a component with all sub-components recursively.

3. Full DOM rendering is the same as full instantiation, but it allows the components to interact with the DOM.

   In case of Jest the real DOM is not available, because Jest does not run in a web browser, therefore a mock DOM is used.

(More details about Enzyme API can be found in attachment 5.)

4.7.4 End-to-end testing

Cypress was chosen as the preferred end-to-end testing solution. Its main strength is good developer experience.

Cypress

Cypress is a test runner, a page traversal and manipulation library and a testing library all in one. It requires no configuration. Cypress was developed with the following goals in mind: to optimize developer experience, to make end-to-end TDD possible, to allow easy testing of SPAs and to make Selenium obsolete. Additionally, it has perfect documentation.

Cypress allows easy testing of SPAs by automatic waiting for asynchronous events. This is an improvement over Selenium, where the programmer must use waiting statements in code.

Unlike Selenium, Cypress runs inside the browser, which makes it faster[65] and it also allows the programmer to access the JS context (e.g. to put mock values into local storage before the test runs, etc).

It runs end-to-end tests in two modes - development mode and execution mode.

In the execution mode Cypress runs headlessly. This mode should be used for running existing tests.

In the development mode Cypress runs in an open browser, this allows the developer to see exactly what the test is doing. While the browser is being automated, Cypress displays the progress of the test broken down into individual commands. When the test fails, it is immediately visible which command caused the problem. In addition, Cypress remembers the state during each command, therefore it is possible to step over individual commands back and forth, which is extremely useful for debugging tests 4.22. Additionally, Cypress error messages are very easy to understand (this is one of the goals of Cypress). It also has a watch mode, which automatically re-runs changed test files. The development mode should be used for writing and debugging new tests.

Cypress compared to other end-to-end test runners

There are several other end-to-end test runners for JS. Going through the twenty most relevant Google search results for the query *"JS end-to-end tester"* yields eight popular alternatives. Four of them are Selenium-based: WebDriver.io, CodeceptJs, Nightwatch, Protractor. The other four are not: TestCafe, CasperJS, Puppeteer and Cypress.

Cypress makes Selenium-based test runners obsolete thanks to its automatic waiting feature and the ability to access JS context. It also claims to run faster[65], but no benchmarks can be found to support that claim.

What separates Cypress from the non-Selenium-based alternatives is mostly its development mode GUI with its great developer experience - especially the *step over* feature.

There are two downsides to Cypress. First, it can currently only run on Google Chrome, but that is fine with respect to the requirements of this prototype.

Second, it cannot run tests in parallel, but this feature is currently under active development.

(More details about Cypress API can be found in attachment 5.)

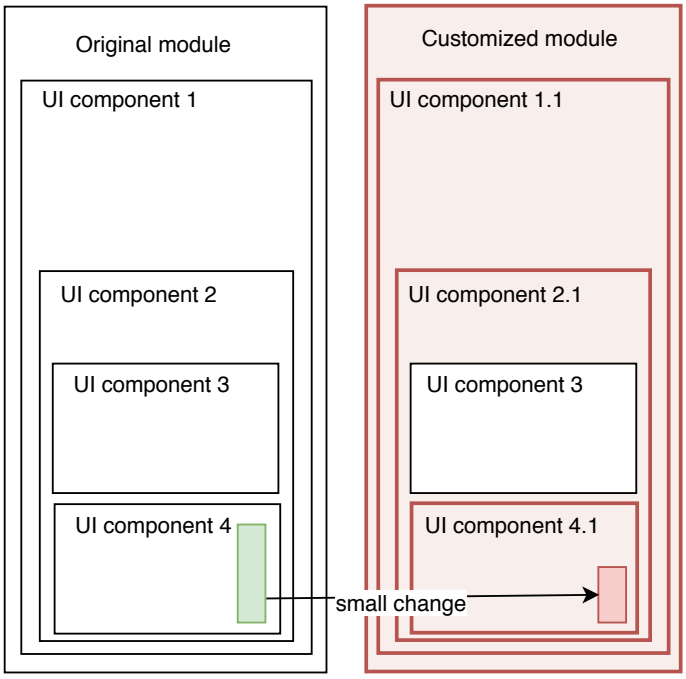


Figure 4.21: The impact of a small change inside a large module in Angular

This problem arises when a small change needs to be made to a part of a complex module composed of multiple levels of sub-modules. When such a change is made, a new version of every parent module has to be created. The code of component 4 is changed, therefore component 2 must change its imports to import the customized component 4.1 instead of 4. And this chain of changing imports propagates up to the root component and the parent module, in which all theses components are registered.

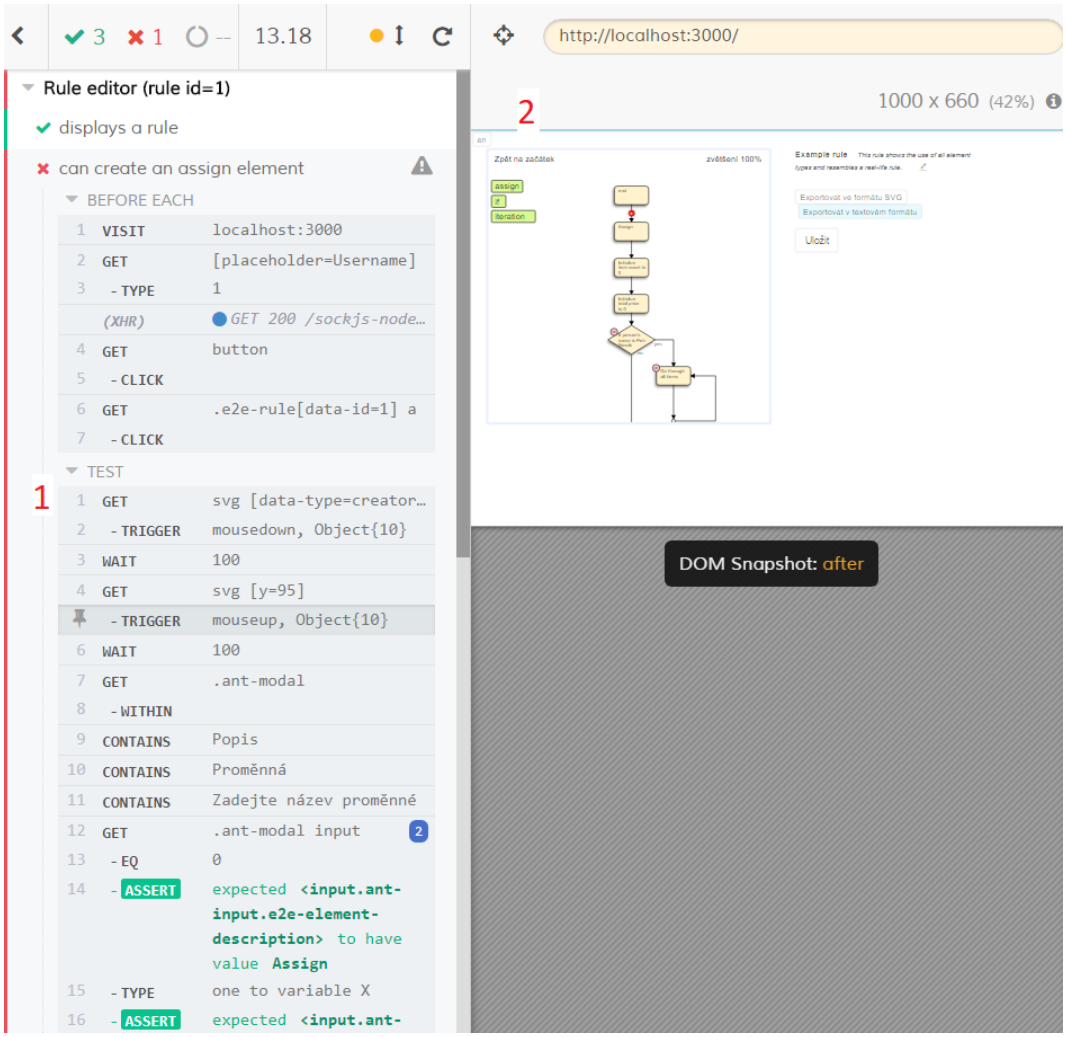


Figure 4.22: Cypress - development mode GUI

The development mode GUI of Cypress shows on the left a list of all test cases of a specific test suite and on the right a real-time display of what is happening to the application.
(1) The code of the test case is broken down into individual actions. It is possible to click on an action and see the *before* and *after* state of the application. This is extremely useful for the development and debugging of end-to-end tests.
(2) Changes to the application are visible as they are being executed by the test runner or as they are being stepped-through.
In conclusion, Cypress offers very good developer experience.

5. Implementation

This chapter provides the architectural description and implementation details of each module of the RMI prototype.

5.1 Client Module

The Client module is responsible for communication with the server component of ERIAN.

Due to lack of access to the actual server component of ERIAN, the Client module is actually a mock module which only imitates communication with the server component. To integrate this prototype with ERAIN, the Client class should be extended with an implementation that actually communicates with the server component.

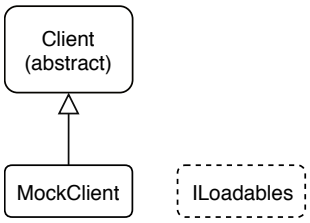


Figure 5.1: Client module

This is an architectural module usage view of TS classes and UI components.
Legend:
"Uses" arrow means that the correctness of the source depends on the correctness of the target. "Produces" arrow means that the source creates instances of the target. Empty-headed arrow represents inheritance. Bold borders signify a sub-module, which is described in more detail in a different figure. Dashed border signifies data objects (classes with no functionality), only important data objects are shown. Grey color signifies classes belonging to a different module than that which is being described. Sharp corners signify a React UI component, only important UI components are shown (rounded corners signify a TS class).

- **Client** - it is an abstract class, which defines abstract methods for loading data from the server component of ERIAN and saving data there.

  It also automatically checks the presence of hard-wired types (as per specification) and converts the loaded data to a more useful format (type converter and type definition map).

  To integrate the RMI prototype to work with the real server component of ERIAN, all that is needed is to supply an implementation inheriting from the client class.

- **ILoadables** - it is a data object, which holds all the loaded data needed when manipulating elements - list of operations, a map of type definitions and a type converter.

- **MockClient** - it is the current implementation of the Client class. It does not use the server component, instead it save everything to local storage and loads it from there.

  Also, at first load it generates some useful example data (for testing, benchmarking and demonstration purposes).

5.2 Diagram Module

The Diagram module is responsible for rendering a rule. It triggers its re-rendering and validation when necessary and adds interactivity (via keyboard shortcuts and mouse gestures).

The rendering of SVG shapes uses Snap.SVG library for convenience.

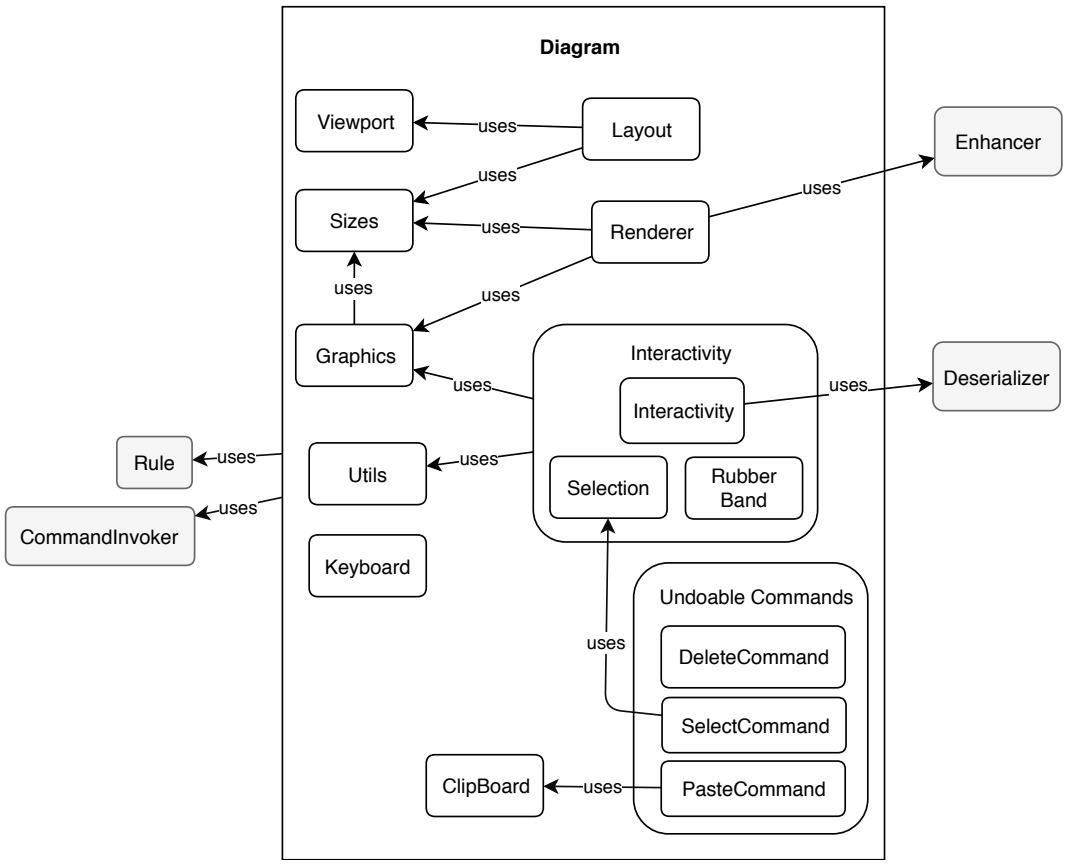


Figure 5.2: Diagram module

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **ClipBoard** - holds a set of copied elements.
- **Graphics** - it is responsible for rendering of all SVG shapes and text.
- **Interactivity**
  - **Interactivity** - it is responsible for several interactive features such as collapsing of sub-rules, creating new elements via drag and drop, pasting and displaying drop spaces.
  - **RubberBand** - it is an interactivity feature. A rubber band that stretches when mouse is dragged and selects the elements inside when the mouse is released.
  - **Selection** - it is an observable set of currently selected elements.
- **Keyboard** - it is responsible for keeping track of pressed keyboard combinations (e.g. Ctrl + C). It allows listening to these keyboard combinations. It can listen to key presses on any HTML element, but in the case

of the Diagram module it is used for listening to the main SVG element which contains the rendered rule.

- **Layout** - computes positions in pixels for all elements of a rule and returns only the ones that fit inside the view-port.

  The computation of the layout is independent of element types, so the code does not have to be changed after adding a new element type.

- **Renderer** - takes a set of elements and their positions are renders them (shape and text) onto a given SVG element (without any interactivity - no event listeners are registered).

  The Renderer has a simple implementation, it just calls the *renderSvg()* method of the enhancers of the given elements. This design minimizes the impact factor of adding new types of elements.

- **Sizes** - it holds sizes. All rendered SVG elements and text get their dimensions from the Sizes class. Zooming in and out is implemented by changing the magnification of the Sizes class. It also holds the sizes for grid cells used for computing the layout of a rule.

- **Undoable Commands** - these are commands (pieces of functionality) that are executed by the CommandInvoker module and therefore can be undone or re-done.

  - **DeleteCommand** - it performs the deletion of selected elements.
  - **PasteCommand** - it performs the pasting of copied elements (from the ClipBoard).
  - **SelectCommand** - it performs selection of elements.

- **ViewPort** - it represents a two-dimensional sliding window. it has a position, width and height all in pixels.

  Its dimensions are intended to be set to be the same as the dimensions of the root SVG element on the screen.

  It is used by the Layout class to determine which elements should be rendered and which do not need to be rendered, because they would not fit on the screen. Browsing is implemented by moving the position of the viewport and re-rendering the rule.

5.2.1 The Rendering Algorithm

The rule is rendered by calculating its layout by the Layout class. Then all elements that fit into the view-port have their SVG representation rendered by the Renderer class. Finally interactivity is added to the rendered rule by the Interactivity class.

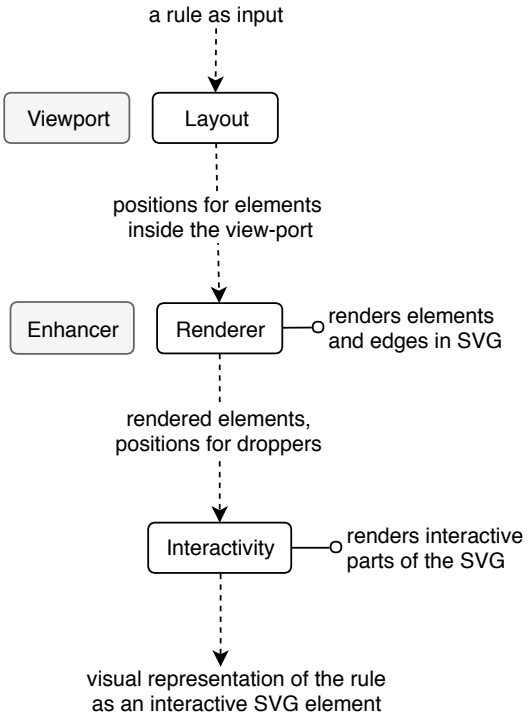


Figure 5.3: Steps of the rendering algorithm

Legend: Dashed arrows show how steps are executed after each other and what data is passed as input to the next step. Modules that are important for the implementation of each step are grey.

5.2.2 Command Invoker Module

The CommandInvoker module is a straight forward implementation of the well-known design pattern. It stores all executed commands and can travel back and forward undoing or re-doing them.

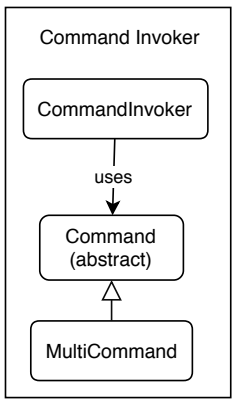


Figure 5.4: Command invoker module

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **Command** - it is an abstract class representing an arbitrary action. It contains one method to execute the action and one to undo it.
- **CommandInvoker** - executes commands and keeps track of them. Allows undo and redo.

- **MultiCommand** - represents several commands grouped together so that they can be undone or re-done all at once.

  It is useful in situations where a command is composed of multiple smaller commands (e.g. delete and unselect).

5.3 Enhancers

Each different type of element (e.g. assignment, conditional statement, etc.) has a different GUI popup for editing its data and a different SVG representation.

Enhancers are used in order to separate the GUI and SVG rendering logic from the Rule module, while still allowing the use of dynamic polymorphism for rendering the GUI popups and SVG representation.

They also minimize the impact factor of adding new types of elements.

Enhancers have two important methods:

- **renderGuiPopup()** - returns the UI component that allows editing of this type of element

- **renderSvg()** - renders an SVG representation of the given type of element, including arrows.

  The method takes the result of layout computation as input 5.6 and returns the rendered element along with all necessary data needed by the interactivity module 5.7. It also uses *element.graphicalMetadata* and *element.errors*.

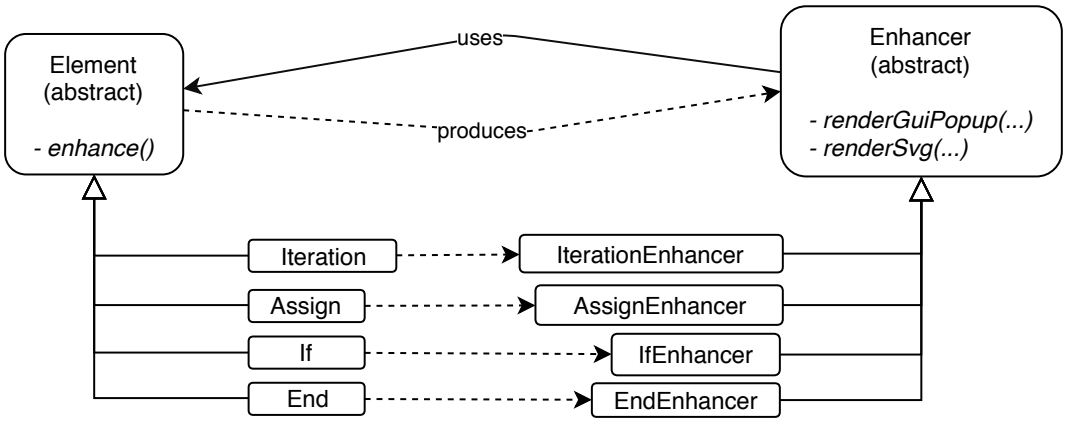


Figure 5.5: Element enhancers

Elements produce their corresponding enhancers by calling the *enhance()* method.
Legend: This is a usage view of the classes. A module uses another module if and only if its correctness depends on the correctness of the other module.

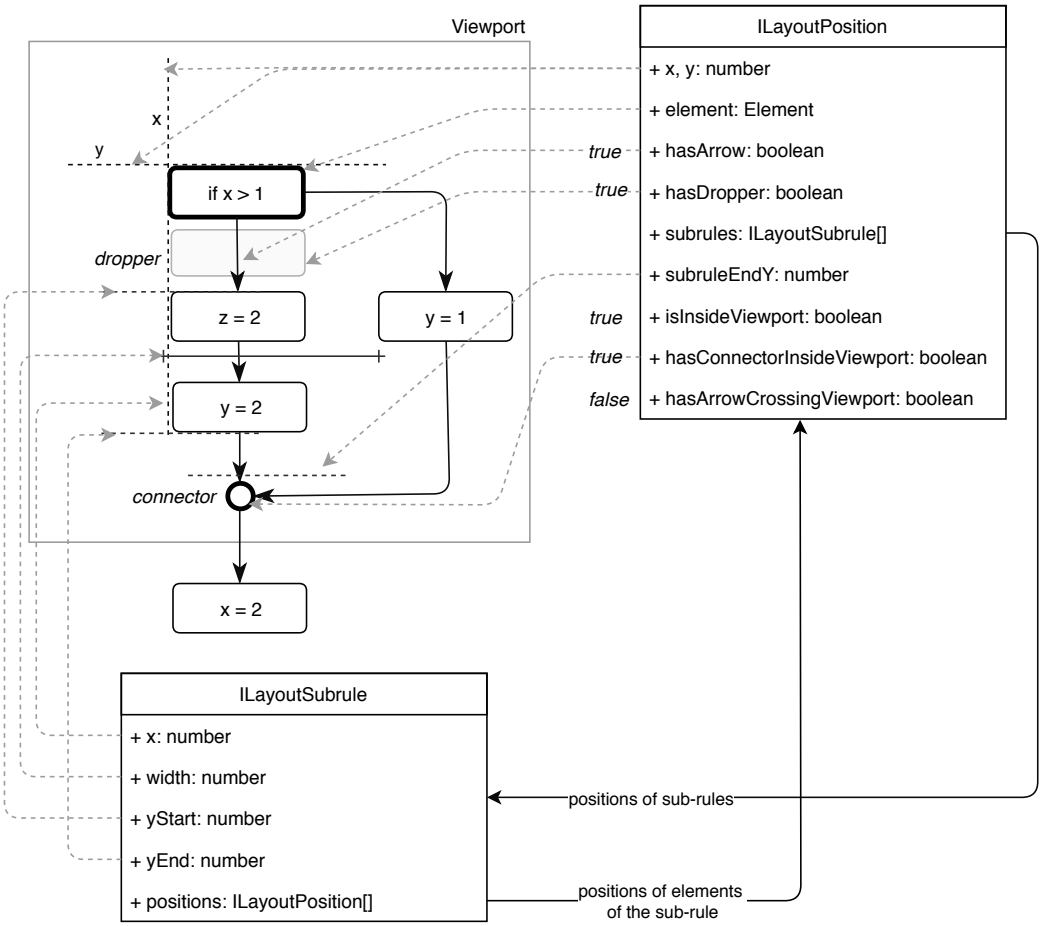


Figure 5.6: Input of the Enhancer.renderSVG method

The conditional statement element is inside the view-port and has a dropper where new elements can be dropped after interactivity is added to the SVG. It renders five arrows - two which connect the element to the beginning of each of its sub-rules, two which connect the end of each sub-rule to a connector and one coming out of the connector.
*The UML classes represent data objects passed as input to the **renderSVG** method. The rest of the diagram is an example of a part of a rendered rule, which illustrates the meaning of each attribute.*

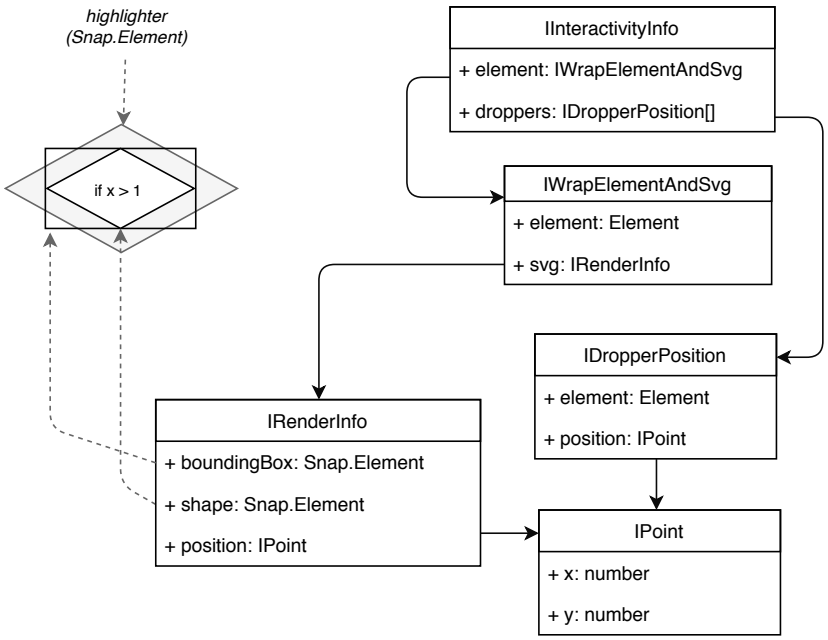


Figure 5.7: Output of the Enhancer.renderSVG method

Dropper positions are used by the interactivity module to create doppers (places where new or copied elements can be dropped and added to the rule).
Bounding boxes are used by the interactivity module to implement selection. An element gets selected only if its bounding box fix inside the selection rubber band.
*The UML classes represent data objects passed as output from the **renderSVG** method. The rest of the diagram is an example of a rendered element shape with a highlighter and a bounding box.*

5.4 GUI Modules

GUI is composed of React UI components. *App* is the root component, which includes all other components and gets mounted onto the HTML page. More complex UI components are called *modules*.

Most UI components use the *antd* component library, which provides nicely styled UI components and the *react-i18next* library, which provides internationalization of UI and non-UI components. *(More details about the libraries and React can be found in attachment 4.)*

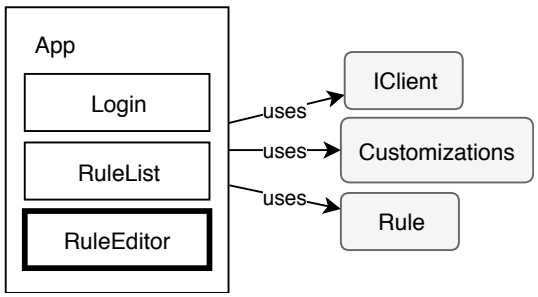


Figure 5.8: GUI value input module

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **App** - loads customizations according to the identity of the logged-in user and passes the Customizations object to all sub-components that contain

code customizations. *(The customization mechanism is implemented and used exactly as described in chapter Design 4.6.3.)*

- **Login** - simulates the login process for the purposes of demonstrating the customization mechanism (implementation of authentication is not part of the RMI prototype).
- **RuleList** - shows a list of rules currently available at the server component and allows the user to open any of them for editing.

5.4.1 GUI Rule Editor Module

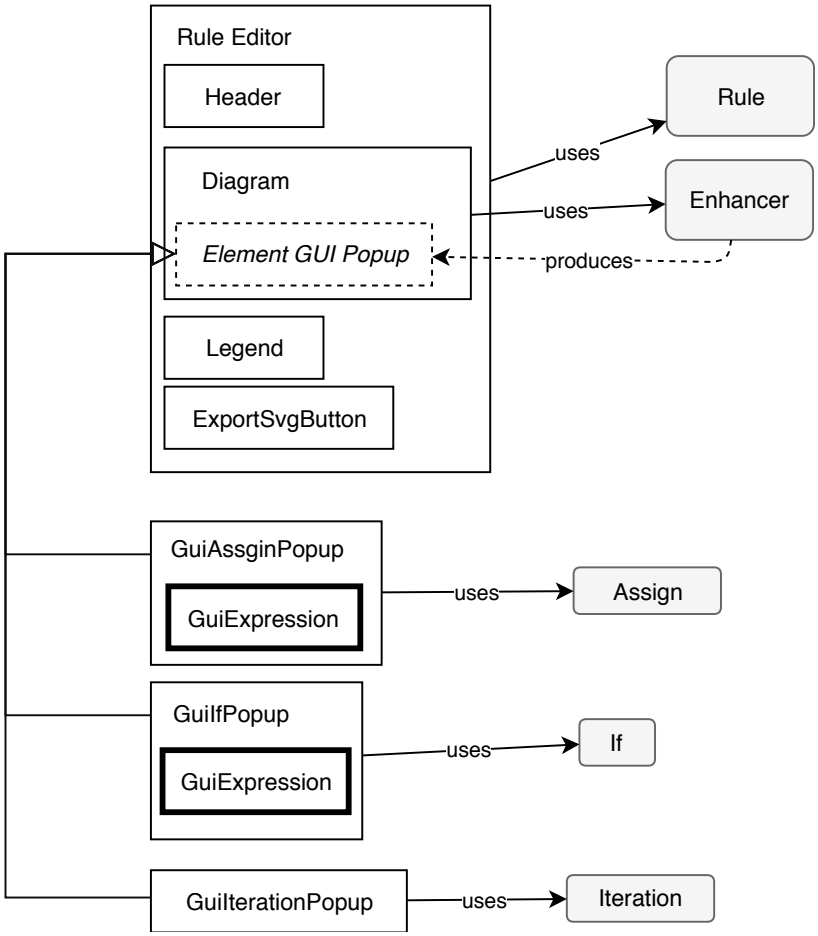


Figure 5.9: GUI rule editor

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

The RuleEditor module provides a graphical way of editing a rule.

- **Diagram** - displays the graphic representation of a rule interactively and allows editing.
- **Element GUI Popup** - appears when an element is clicked and it allows editing of its data. Each type of element has its own popup.

  The editing of elements works in the following way. When a popup is opened, the element is cloned and all subsequent changes made to that

element by the user are preformed on the clone only. When the user wants to save the changes, the clone is validated. If the clone is valid, its data get merged into the original element, otherwise the save action is prevented. If the user cancels the editing, the clone is discarded and the rule is not affected in any way.

- **ExportSvgButton** - exports the graphical representation of the whole rule rendered as SVG.
- **Header** - displays and allows editing of the name and description of a rule.
- **Legend** - displays a list of keyboard shortcuts available on the Diagram component.

5.4.2 GUI Expression Module

The GuiExpression module displays and allows editing of an expression.

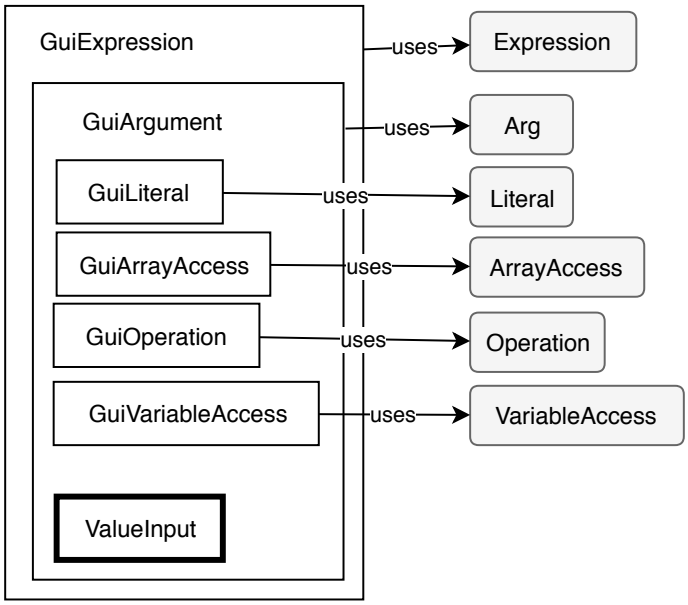


Figure 5.10: GUI expression module

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **GuiArgument** - displays a value or an empty argument, allows deletion of the value and the input of a new value.
- **GuiArrayAccess** - displays an array access.
- **GuiLiteral** - displays a literal.
- **GuiOperation** - displays an operation and its arguments.
- **GuiVariableAccess** - displays a variable access.

5.4.3 GUI Value Input Module

The ValueInput module allows the input of a value (e.g. literal, operation, etc) with powerful auto-completion similar to the intellisense of an IDE.

It offers a list of all matching variables accesses, array accesses and operations, and it only offers the ones that have the correct type (convertible to the type of the argument).

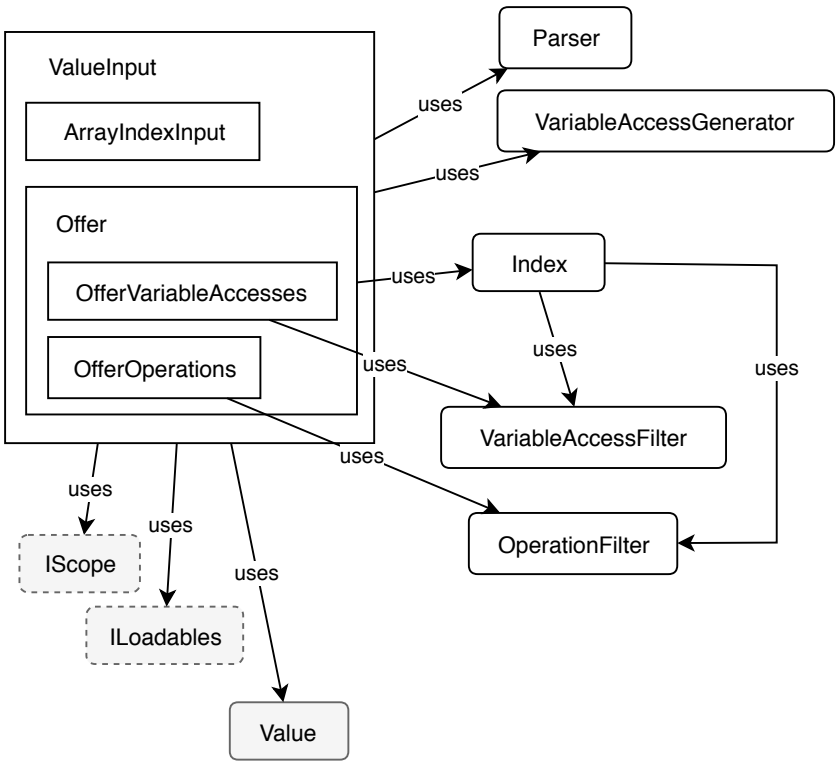


Figure 5.11: GUI value input module

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **ArrayIndexInput** - allows the input of an integer index for an array access.
- **Index** - keeps track of which auto-completable option is currently selected.
- **Offer** - shows a list of auto-completable operations and variable/array accesses.
- **OfferOperations** - shows a list of auto-completable operations.
- **OfferVariableAccesses** - shows a list of auto-completable variable/array accesses.
- **OperationFilter** - filters groups of operations by a string prefix.
- **Parser** - parses the current input string, decides what type of value it represents and returns the correctly typed value (e.g. a literal, an operation, etc).
- **VariableAccessFilter** - filters variable/array accesses by a string prefix.

- **VariableAccessGenerator** - generates all auto-completable variable/array accesses of compatible type based on the current variable scope and a list of all type definitions.

5.5 Rule Module

The rule module implements the rule according to the specified logical model, plus it provides all the necessary functionality related to editing the rule, such as computing the scope of variables, validation, etc.

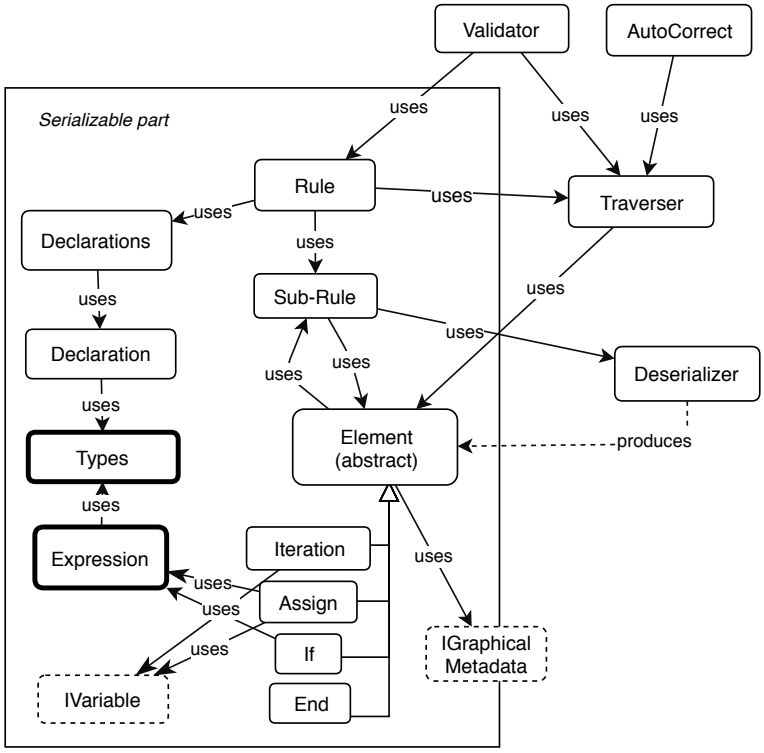


Figure 5.12: Rule module

The serializable part of the Rule module is the one that gets sent to the server component.
This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **AutoCorrect** - is used after copying and pasting. It scans the pasted subset of a rule for all variable declarations and it redeclares all those variables and auto-corrects the IDs of all occurrences of those variables in the pasted sub-set of the rule.

- **Declaration** - represents a declaration of a variable.

- **Declarations** - stores local and global declarations and it can create new ones.

- **Deserializer** - de-serializes elements based on the *elementType* attribute of their JSON serialization.

  This is the only class that needs to be updated when a new element type is implemented.

- **Element** - implements elements according to the specification. Plus it offers many useful methods - it can clone itself, serialize, return IDs of all used variables and all declared variables, etc.

  Elements also hold a list of errors *(see Validator module for more details)*.

  - **Assign** - implemented per specification.

  - **End** - it is a trivial useful element that simplifies traversal of rules and working with empty rules. End elements are not visible, except the ones in the first sub-rule in the hierarchy.

  - **If** - implemented per specification.

  - **Iteration** - implemented per specification.

  - **IterationHelper** - solves the visibility problem of the iteration variable of the Iteration element. The iteration variable is declared in the iteration element, but it should be visible only inside the iteration cycle (so it should actually be declared inside the sub-rule and that is exactly what the IterationHelper does).

    The IterationHelper is placed as the second element of each iteration cycle. It declares the iteration variables and serves as a proxy for the Iteration element. Any mutation methods called on the IterationHelper are also called on the Iteration element (e.g. if an error is set on the IterationHelper, it automatically gets set on the Iteration element as well).

    The purpose of IterationHelpers only to provide correct variable scoping in the RMI. They are not part of the serialization and are never sent to the server component. They are also not visible.

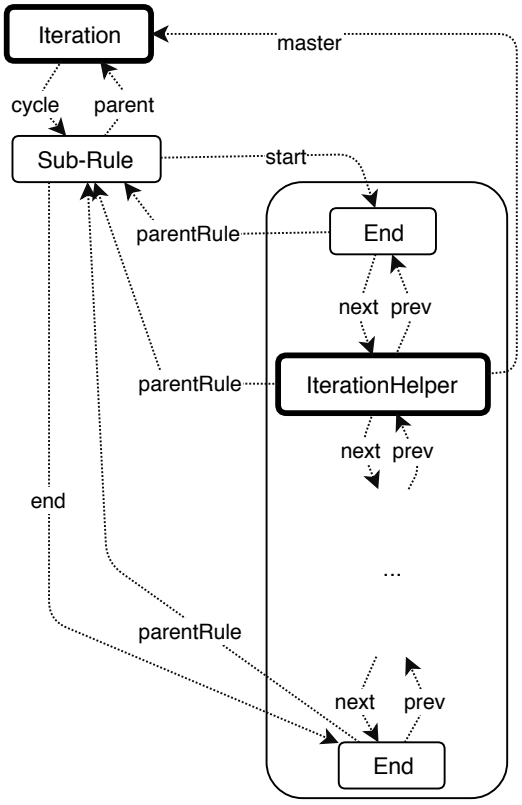


Figure 5.13: Iteration helper element

The problem with iteration element is that it holds the iteration variable, but it should only be visible inside its sub-rule, not outside. This is why IterationHelper element is introduced to hold the iteration variable instead. The IterationHelper is placed as the second element of the iteration cycle and it forwards all mutation method calls to the *master* Iteration element.
Legend: This is an example of an Iteration element instance. Arrows are pointers.

- **IGraphicalMetadata** - is a data object, which holds all visualization-related data of an element (whether it is visible, selected, collapsed, etc). This data is not relevant to the server component.

- **Rule** - implemented per specification. It contains all variable declarations and the rule (a recursively defined structure of sub-rules and elements).

  It is responsible for variable scoping. It can find out which variables are visible in the scope of a given element and which variable names cannot be declared by an element, because they would collide with declarations further down the rule.

  The Rule class and everything it contains (i.e. declarations, sub-rules, elements, expressions, variables, etc.) is serializable and deserializable, so that it can be sent to the server component. Serialization formats are defined in the corresponding interfaces for each class (e.g. ISubRule, IElement, IExpression, IValue, etc).

- **SubRule** - it contains a linked list of elements.

  Elements are stored in a linked list instead of an array because this makes it easier to traverse the sub-rule, add and remove elements.

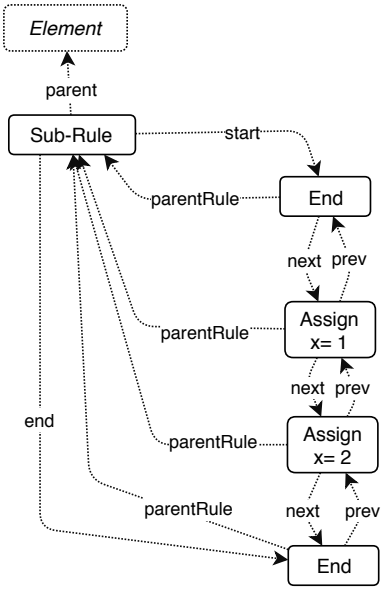


Figure 5.14: Example of a sub-rule

Each sub-rule starts and ends with an End element. Elements of a sub-rule are kept in a doubly-linked list and each of them also references the sub-rule object.
Each sub-rule has references its parent element (with the exception of the first sub-rule in the hierarchy).
Legend: This is an example of a sub-rule instance with element instances. Arrows are pointers.

- **Traverser** - implements all kinds of traversal algorithms for traversing a rule (all predecessors, all ancestors, all successors, etc). All traversals can be stopped an configured with an options object.

- **Validator** - checks for logical errors in a rule (introduced specifically to handle errors arising from copy and pasting).

  The following errors can occur:

  - an undeclared variable is assigned to
  - an undeclared variable is used as a value (e.g. in an expression)
  - redeclaration of a variable with the same name
  - redeclaration of a variable with the same ID and name (never happens thanks to auto-correction of declaration IDs)

  The validator class checks for these errors and invalidates erroneous elements. It also marks all ancestors of invalid elements, in order to facilitate easy localization of the invalid elements when browsing the graphical representation of a rule.

5.5.1 Types Module

The Types module handles types and type conversions.

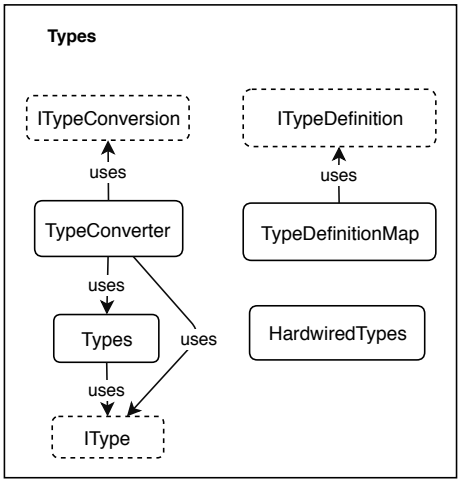


Figure 5.15: Types module

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **HardwiredTypes** - is an enum of types that are hard-wired to the RMI, because the code depends on them (boolean, integer, float, string).

- **TypeConverter** - tells whether it is possible to convert from one type to another (transitively).

- **IType** - represents a type.

- **ITypeConversion** - represents a type conversion.

- **ITypeDefinition** - represents a type definition.

- **TypeDefinitionMap** - maps type names to type definitions.

- **Types** - has utility methods for working with types (checks equality, clones type objects, etc).

5.5.2 Expression Module

The Expression class represents a mathematical expression. It can infer the type of the expression and check whether it is empty or unfinished.

It can also get IDs of all variables used in the expression and replace them by new IDs (useful for auto-correction after copy and pasting). Lastly, it can traverse all values of the expression.

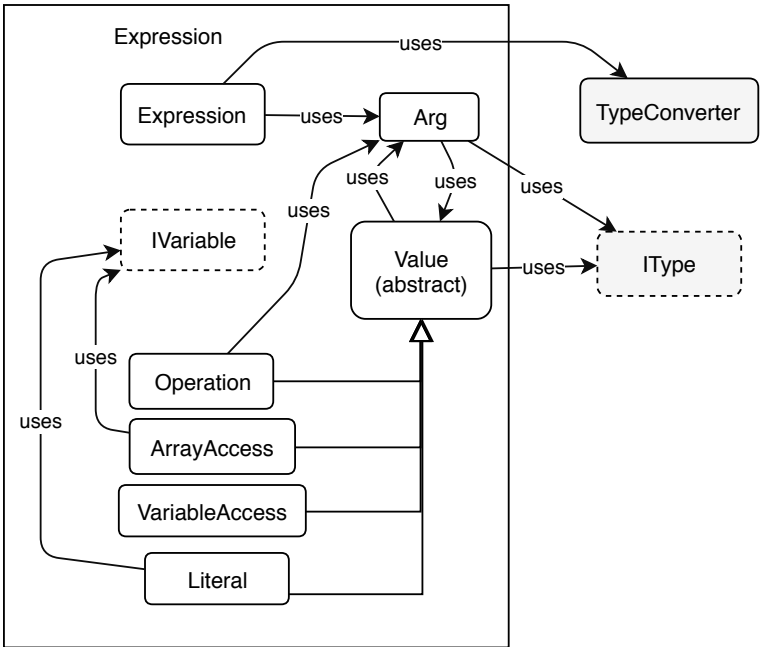


Figure 5.16: Expression module

This is an architectural module usage view of TS classes and UI components. (Legend in figure 5.1.)

- **Arg** - holds a typed and named argument.

  It de-serializes values based on the *what* attribute of their JSON serialization. It can also recursively traverse all values beneath it.

- **IVariable** - represents a variable.

  Variables have a boolean flag *isDeclaration*, this is an important detail, because scoping and Validator and AutoCorrector modules depend on it. The flag was introduced to resolve confusing situations that could otherwise arise during copy and pasting.

- **Value** - abstract class for classes that hold a value.

  Each value can return a set of IDs of variables that it uses, which is useful for validation of the rule (especially after copy and pasting). Each can also replace IDs of their used variables, which is useful for auto-correction (used after copy and pasting).

  - **ArrayAccess** - represents an array access, which is a mixture of variable attribute access and an integer argument (e.g. *Doc.Items[1]*).
  - **Literal** - represents a literal value and is capable of type inference.
  - **Operation** - represents an operation.
  - **VariableAccess** - represents a variable attribute access (e.g. *Doc.ID*).

5.6 Validation

Before every time a rule is rendered it undergoes validation and if errors are found they are reported in the GUI 5.17. A rule can get into an invalid state after a

paste or a delete operation. Other operations do not allow the user to create invalid elements thanks to autocompletion and checking of variable names and attributes.

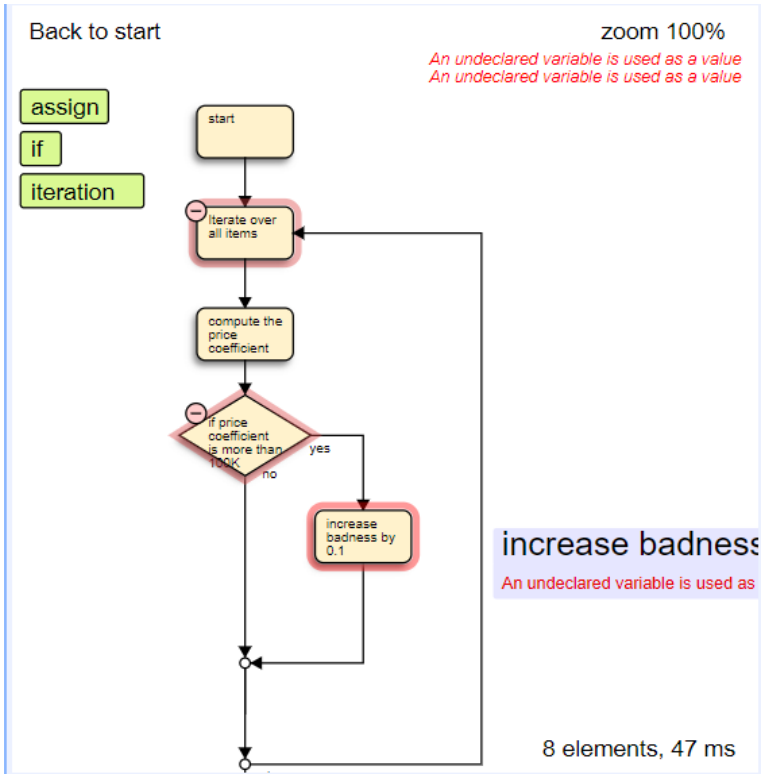


Figure 5.17: Validation of a rule

Validation is performed before each time a rule is rendered. If errors are discovered, they are reported in the upper right corner. Clicking on the error brings the invalid element into the center of the screen. Hovering over the invalid element reveals information about its invalidity.

Validation is done by the Validator class 5.5 and the following cases of invalidity can occur:

- an undeclared variable is assigned to (the declaration was deleted)
- an undeclared variable is used as a value (e.g. in an expression)(the declaration was deleted)
- redeclaration of a variable with the same name (the declaration was copied under a different one of same name, this includes iteration variables as well)
- redeclaration of a variable with the same ID and name (the declaration was copied under itself, this includes iteration variables as well)

  (this should never happen thanks to auto-correction of declaration IDs during copy/paste operations. However, the check is still used in order to keep consistency even in case of bugs)

The validation algorithm is based on multiple traversals of the rule and its time complexity is quadratic in the number of elements.

6. Evaluation

All functional and quality requirements were met.

React framework is used for the definition of UI components *(requirement 2b)* and internationalization *(requirement 2a)* is handled by *react-i18next* library. *(Details about internationalization can be found in attachment 6.)*

It should be noted this thesis is concerned only with the client side and therefore evaluation does not cover the server side of system ERIAN.

6.1 Performance of Rule Rendering

Rendering of the graphical representation of a rule and export as SVG image *(requirement 1a and 1b)* are both sufficiently fast and the RMI can handle large rules efficiently.

- Rendering a rule containing 1000 elements takes on average less than 30 ms, provided the shape of the rule is not very dense, which easily meets the requirement for 1 s. 6.1

- Rendering a "pathological" worst-case scenario rule takes on average 302 ms, which meets the requirement for 1 s. 6.1

- Exporting a rule containing 1000 elements as an SVG image takes on average 1.2 s, which meets the requirement for 3 s. 6.2

- There is no limit on the number of sub-rules nested inside of each other.

(The reproducible benchmarks can be found in Attachment 3. The benchmarks were measured on the same computer as specified in quality requirements.)

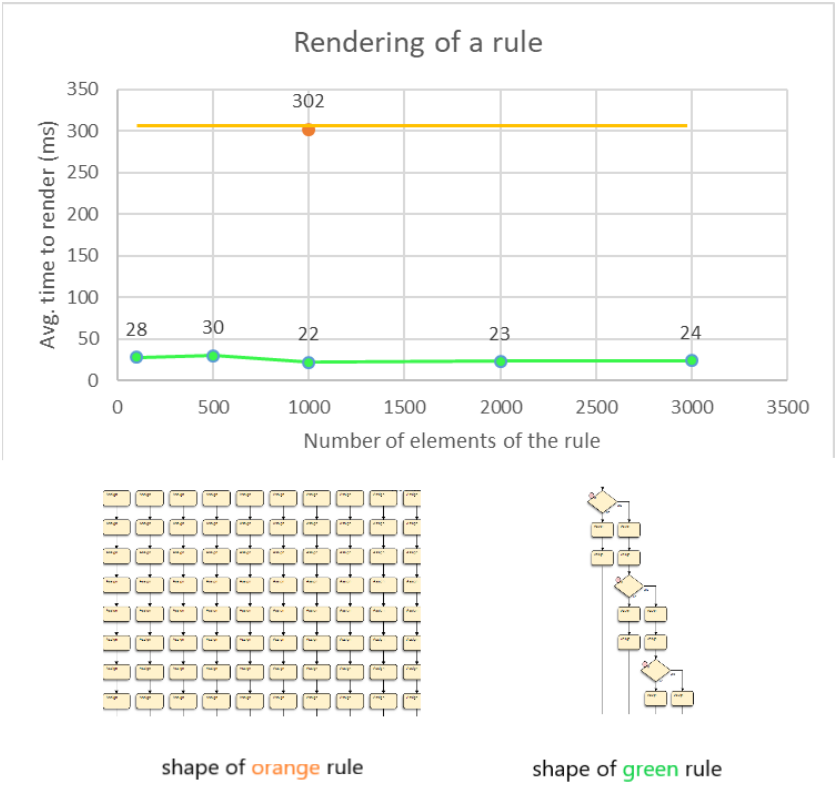


Figure 6.1: Rendering speed of a rule in the UI

The time complexity function is constant. It is independent of the number of elements comprising the rule but it is dependent on the shape of the rule. If the shape is normal and not dense, the rendering is very fast at approximately 30 ms. If the rule is denser the rendering slows down to the extreme case of 302 ms when the whole screen is covered by 186 elements.
In any case, the speed is enough to meet the quality requirements.

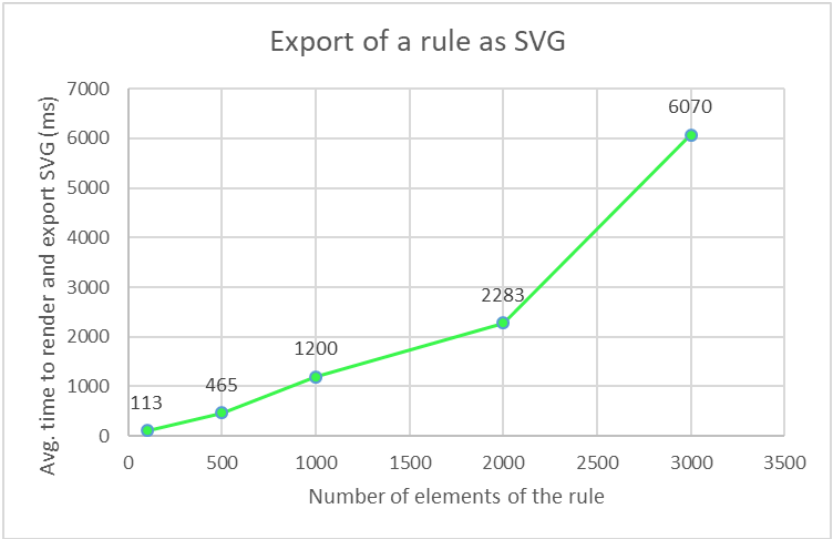


Figure 6.2: Speed of exporting a rule as SVG

The time complexity function is linear and becomes steeper with extreme number of nodes, which can probably be attributed to DOM behavior, but it is irrelevant to this application. The export is fast enough to meet the quality requirements easily.

6.2 Customization Mechanism

A customization mechanism *(requirement 2c)* was successfully implemented based on Webpack code-splitting and dynamic component instantiation in React (as described in section 4.6.3). The simplicity of loading and using UI components in runtime in React was the main reason why it was chosen over Angular.

Attachment 1 demonstrates the usage of the customization mechanism. Two versions of the RMI are implemented for two different customers of Komix. For simplicity the customers are called "1" *(figure 6.3)* and "2" *(figure 6.4)*.

6.3 Testability

Test runners and libraries for testing TS classes, React UI components are configured and ready to be used. The same is true of end-to-end tests *(*reqirements 3a and 3b).

Testability was an important reasons why SVG was chosen for the graphical representation of rules.

Test examples which sufficiently demonstrate the APIs of the different testing libraries are included with the code *(attachment 4)*.

1. */src/rule* contains many examples of TS class tests using Jest API.
2. */src/gui* contains examples of React UI component tests using Enzyme API and Jest API.
3. */src/cypress/integration* contains examples of end-to-end tests using Cypress API.

(Testing guides and APIs are described in attachment 5.)

6.4 Usability

Usability of the prototype was tested using the System Usability Scale (SUS).[66] *(Note: usability improvement was not one of the requirements, this evaluation is purely informative.)*

The results show that the usability of both the original software and the new prototype are comparable (according to their SUS score). However, using the prototype, test users were able to complete most of their tasks in half the time (unless they were well trained in using the original software).

(Details about the test and its results can be found in attachment 7.)

6.5 Acceptance Tests

All functional requirements 3.1 were tested with acceptance tests using browser automation.

(Acceptance tests can be found in attachment 8.)

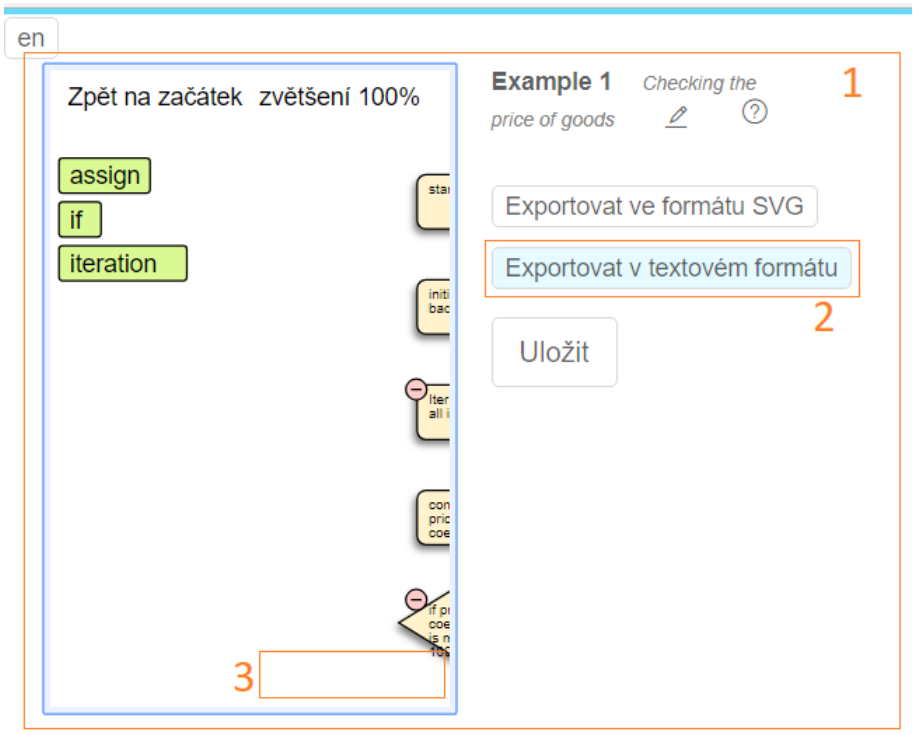


Figure 6.3: Customization example - user 1

User 1 has several customizations made to the RMI.

1. The layout of the rule editor is customized (diagram on the left, buttons on the right).
2. A button, which exports the rule serialized in a JSON representation.
3. Rendering information (number of elements and elapsed time) is hidden.

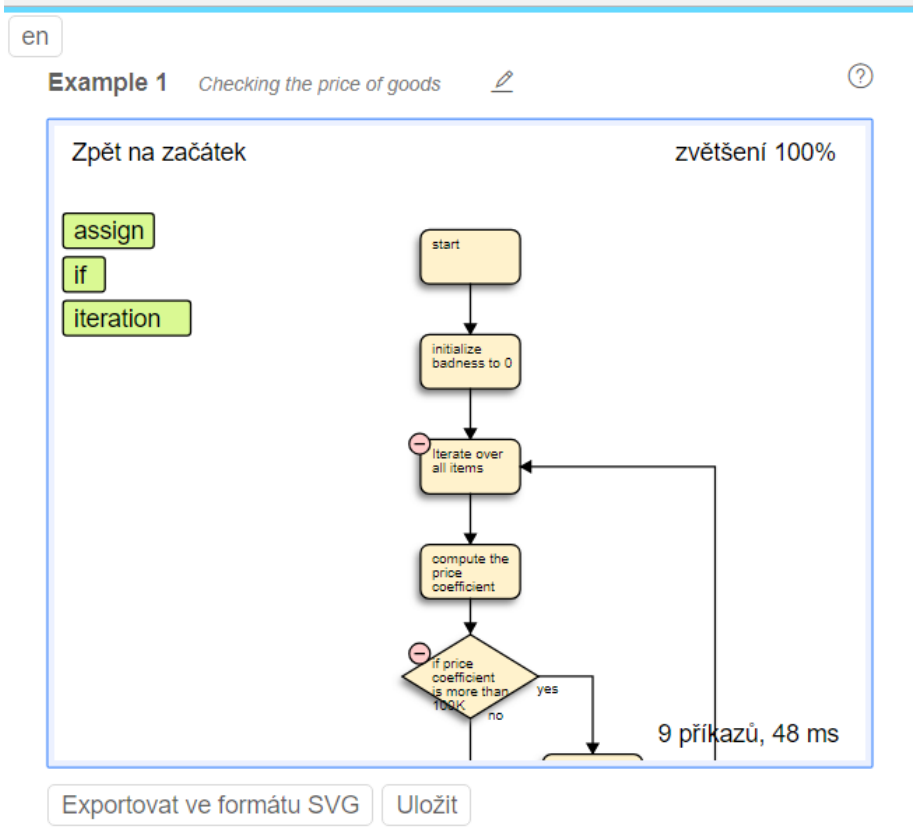


Figure 6.4: Customization example - user 2

User 2 has no customizations made to the RMI.

Conclusion

This thesis is concerned with the implementation of a thin client prototype of the Rule Management Interface of system ERIAN. The purpose of the prototype is to prove the viability of a thin client solution, which will make the existing thick client obsolete, and lay a foundation for the full implementation of the new thin client based on modern web technologies.

First, functional and quality requirements were analyzed based on a demonstration of the existing thick client, prototyping and discussing the needs of Komix stakeholders.

Then the prototype was designed in order to satisfy the requirements. A lot of attention was given to choosing the appropriate technologies for the task. The prototype focuses on staying responsive even when handling exceptionally large data (business rules). Another very important property of the design is that it provides the ability to develop and maintain multiple customized versions of the RMI for different enterprise customers. It is also well testable, internationalized and uses a modern component-based approach for building a UI.

Finally, the prototype was implemented using the chosen technologies and partially tested. *Thorough testing does not make sense, since this is just a prototype and it will likely go through many changes before it is production-ready.*

Even though the prototype is not supposed to cover the full functionality of the existing thick client, it is already an improvement over it in serveral aspects. The most important functional improvements are copy and pasting, error prevention while editing elements and type-aware value auto-completion during editing of expressions.

It is expected that the full thin client of the RMI will be implemented based on this prototype. The prototype is ready for the addition of the remaining modules and all the technologies needed for the implementation and testing are tried and ready.

Bibliography

[1] Schultheiss, Louis A. *Techniques of flow-charting.* Graduate School of Library Science. University of Illinois at Urbana-Champaign ISSN 0069-4789

[2] Singh, Anil. *Angular 5 vs. Angular 4.* https://www.code-sample.com/2017/09/angular-5-vs-angular-4-whats-new-in.html [Online; accessed 2018]

[3] Silva, Filipe. *Angular mutliple apps integration.* https://github.com/angular/angular-cli/wiki/stories-multiple-apps [Online; accessed 2018]

[4] Google, Angular team. *Angular official tutorial and documentation.* https://angular.io [Online; accessed 2018]

[5] Motto, Todd. *Angular, TypeScript and JavaScript articles.* https://toddmotto.com/ [Online; accessed 2018]

[6] Kwiatek, Wojciech. *Making use of RxJS in Angular.* https://auth0.com/blog/making-use-of-rxjs-angular/ [Online; accessed 2018]

[7] Podwysocki, Matthew et. al. *RxJS (aka Reactive Extensions library) official documentation.* http://reactivex.io/rxjs/manual/overview.html [Online; accessed 2018]

[8] Brocchi, Mike. *Using non-UMD libraries with Angular.* https://github.com/angular/angular-cli/wiki/stories-global-scripts [Online; accessed 2018]

[9] *Angular articles.* https://alligator.io/angular/ [Online; accessed 2018]

[10] Romuald, Brillout. *Catalog of Angular 2+ Components & Libraries.* https://github.com/brillout/awesome-angular-components [Online; accessed 11.3.2018]

[11] You, Evan. *Vue official documentation.* https://vuejs.org/ [Online; accessed 2018]

[12] Kazuya, Kawaguchi. *A curated list of awesome things related to Vue.js (plugins).* https://github.com/vuejs/awesome-vue [Online; accessed 2018]

[13] Facebook. *React official documentation.* https://reactjs.org/ [Online; accessed 10.3.2018]

[14] Abramov, Dan. *Getting Started with Redux.* https://egghead.io/courses/getting-started-with-redux [Online; accessed 10.3.2018]

[15] Stoiber, Max et al. *create-react-app: a zero-configuration, pre-packaged React with TS, linting, Jest test framework and more.* https://github.com/facebook/create-react-app [Online; accessed 10.3.2018]

[16] Rauschmayer, Axel. *ES proposal: import() – dynamically importing ES modules.* http://2ality.com/2017/01/import-operator.html [Online; accessed 2018]

[17] James, Kyle. *If TypeScript is so great, how come all notable ReactJS projects use Babel?.* https://discuss.reactjs.org/t/if-typescript-is-so-great-how-come-all-notable-reactjs-projects-use-babel/4887 [Online; accessed 2016]

[18] Zavelevsky, Doron. *Protractor vs. Selenium: Which is Easier?.* http://testautomation.applitools.com/post/94994807787/protractor-vs-selenium-which-is-easier [Online; accessed 10.3.2018]

[19] Resig, John. *JQuery official documentation.* http://jquery.com/ [Online; accessed 12.3.2018]

[20] Ashkenas, Jeremy. *Backbone official documentation.* http://backbonejs.org/ [Online; accessed 12.3.2018]

[21] Fowler, Martin. *Feature Toggles (aka Feature Flags).* https://martinfowler.com/articles/feature-toggles.html [Online; accessed 2018]

[22] Koppers, Tobias (probably). *Webpack official documentation.* https://webpack.js.org/ [Online; accessed 4.4.2018]

[23] Koppers, Tobias (probably). *Code Splitting, webpack official documentation.* https://webpack.js.org/guides/code-splitting/ [Online; accessed 4.4.2018]

[24] Google, Angular team. *Lazy Loading Feature Modules.* https://angular.io/guide/lazy-loading-ngmodules [Online; accessed 4.4.2018]

[25] Cooter, Kaelan. *Frontend in 2018: More consensus, less complexity.* https://blog.logrocket.com/what-im-looking-for-from-frontend-in-2018-2f1de300b548 [Online; accessed 13.3.2018]

[26] Chartrand, Ryan. *The Top JavaScript Trends to Watch in 2018.* https://x-team.com/blog/top-javascript-trends-2018/ [Online; written 27.12.2017]

[27] Codeburst, (company). *JavaScript Trends in 2018.* https://codeburst.io/javascript-trends-in-2018-3fb0077259 [Online; written 1.3.2018]

[28] Morelli, Brandon. *The 2018 Web Developer Roadmap.* https://codeburst.io/the-2018-web-developer-roadmap-826b1b806e8d [Online; published 22.1.2018]

[29] Cuelogic, *(company). Top 3 Best JavaScript Frameworks in 2018.* http://www.cuelogic.com/blog/top-3-best-javascript-frameworks-in-2018/ [Online; published 1.3.2018]

[30] Ember team. *EmberJS official documentation.* https://www.emberjs.com/ [Online; accessed 14.3.2018]

[31] Moshe, Dor. *ES8 was Released and here are its Main New Features.* https://hackernoon.com/es8-was-released-and-here-are-its-main-new-features-ee9c394adf66 [Online; published 10.7.2017]

[32] Delgado, Carlos *(probably). Top 5 : Best free diagrams javascript libraries.* https://ourcodeworld.com/articles/read/159/top-5-best-free-diagrams-javascript-libraries [Online; accessed 10.3.2018]

[33] OfficeJS (group). *10 Javascript Flowcharting Libraries (a draft).* https://www.erp5.com/officejs/javascript-10.Flow.Chart [Online; accessed 10.3.2018]

[34] Ed-Douibi, Hamza. *10 JavaScript libraries to draw your own diagrams.* https://modeling-languages.com/javascript-drawing-libraries-diagrams/ [Online; accessed 21.3.2018]

[35] Nexedi (company). *Javascript Flowcharting Libraries.* https://www.nexedi.com/javascript-Flow.Chart [Online; 9.5.2017]

[36] Dascalescu, Dan. *Graph visualization library in JavaScript.* https://stackoverflow.com/questions/7034/graph-visualization-library-in-javascript [Online; 23.5.2017]

[37] Philipp, Johann. *Dracula Graph Library official documentation.* https://www.graphdracula.net/ [Online; accessed 21.3.2018]

[38] zenluca (username at sourceforge.net). *Js-graph-it a javascript library for graph representation.* http://js-graph-it.sourceforge.net/index.html [Online; accessed 21.3.2018]

[39] MindFusion LLC. (company). *JavaScript Diagram Library.* https://www.mindfusion.eu/javascript-diagram.html [Online; accessed 21.3.2018]

[40] Durman, David et. al. *Rappid Diagramming Framework.* https://www.jointjs.com/ [Online; accessed 21.3.2018]

[41] Northwoods Software (company). *GoJS - Interactive JavaScript Diagrams in HTML.* https://gojs.net/latest/index.html [Online; accessed 21.3.2018]

[42] JGraph Ltd. (company). *mxGraph 3.9.3 - a diagram library.* https://jgraph.github.io/mxgraph/ [Online; accessed 10.3.2018]

[43] JGraph Ltd. (company). *mxGraph API documentation.* https://jgraph.github.io/mxgraph/docs/js-api/files/index-txt.html [Online; accessed 21.3.2018]

[44] Finger, Artur. *mxgraph issue #116.* https://github.com/jgraph/mxgraph/pull/116 [Online; accessed 10.3.2018]

[45] jsPlumb Pty Ltd. (company). *JsPlumb diagram library.* https://jsplumbtoolkit.com/ [Online; accessed 21.3.2018]

[46] Cytoscape Consortium (non-profit company). *Cytoscape - Network Data Integration, Analysis, and Visualization in a Box.* http://www.cytoscape.org/ [Online; accessed 21.3.2018]

[47] Durman, David et. al. *JointJS - JavaScript diagramming library official documentation and tutorial.* http://resources.jointjs.com/docs/jointjs/v2.0/joint.html [Online; accessed 10.3.2018]

[48] Durman, David et. al. *A generated documentation for JointJs (more accurate and complete than the official documentation).* http://definitelytyped.org/docs/jointjs–jointjs/modules/joint.html [Online; accessed 10.3.2018]

[49] Almende B.V. (company). *vis.js Network - a visualization to display networks consisting of nodes and edges.* http://visjs.org/docs/network/ [Online; accessed 21.3.2018]

[50] JGraph LTd. (company). *Draw.io - a well known online graph editor tool.* https://www.draw.io/ [Online; accessed 22.3.2018]

[51] Pettitt, Chris. *Dagre - directed graph layout for JavaScript.* https://github.com/dagrejs/dagre [Online; accessed 13.6.2018]

[52] Real-Time and Embedded Systems Group, Kiel University. *ELK.js library - Eclipse Layout Kernel.* https://github.com/OpenKieler/elkjs [Online; accessed 13.6.2018]

[53] Baranovskiy, Dmitry. *Snap.svg - an SVG library.* http://snapsvg.io/ [Online; accessed 13.6.2018]

[54] Bruckner, Roman. *JointJS performance tips.* http://jsfiddle.net/fjzvqhhk/287/ [Online; accessed 2.3.2018]

[55] Finger, Artur. *Implementing customer-specific software versions in React.* https://github.com/fingerartur/react-cssv [Online]

[56] Finger, Artur. *Angular - a good practical example of testing using all kinds of tests (testing the tour of heroes).* https://github.com/fingerartur/angular-tour-of-heroes [Online]

[57] Nielsen, Jakob. *Usability Engineering, 1993.* Excerpt: https://www.nngroup.com/articles/response-times-3-important-limits/

[58] Schulz, Marius. *TypeScript vs. Flow.* https://blog.mariusschulz.com/2017/01/13/typescript-vs-flow [Online; published 13.1.2017]

[59] Clark, Paul. *List of languages that compile to JS.* https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js [Online; accessed 13.7.2018]

[60] Meteor, team. *Meteor Guide - User Interfaces.* https://guide.meteor.com/ui-ux.html [Online; accessed 16.7.2018]

[61] Facebook, Inc. *React Without JSX.* https://reactjs.org/docs/react-without-jsx.html [Online; accessed 24.7.2018]

[62] Google, Inc. *Dynamic Component Loader.* https://angular.io/guide/dynamic-component-loader [Online; accessed 24.7.2018]

[63] Koretskyi, Maxim. *Avoiding common confusions with modules in Angular.* https://blog.angularindepth.com/avoiding-common-confusions-with-modules-in-angular-ada070e6891f [Online; 9.8.2017]

[64] Voss, Laurie. *The State of JavaScript Frameworks, 2017.* https://www.npmjs.com/npm/state-of-javascript-frameworks-2017-part-1 [Online; 3.1.2018]

[65] Cypress.io, company. *Cypress: How is this different from 'X' testing tool?.* https://docs.cypress.io/faq/questions/general-questions-faq.html [Online; accessed 6.8.2018]

[66] Sauro Jeff. *Measuring Usability with the System Usability Scale (SUS).* https://measuringu.com/sus/ [Online; 2.2.2011]

List of Figures

List of Abbreviations

1. AMD - Asynchronous Module Definition
2. API - Application Programming Interface
3. CLI - Command Line Interface
4. CSS - Cascading Style Sheets
5. CZK - Czech currency (koruna česká)
6. DI - Dependency Injection
7. DOM - Document Object Model
8. ERIAN - Electronic Risk Analysis (name of a software product)
9. ES8 - Ecmascript 8 (the latest version at the time of writing)[31]
10. GUI - Graphical User Interface
11. HTML - HyperText Markup Language 5
12. IDE - Integrated Development Environment
13. JS - JavaScript
14. JSX - JavaScript XML (an HTML-like syntax used for writing UI components in React)
15. SPA - Single Page Application - a modern web application written in JS that appears very fast because it uses only one page load (e.g. gmail.com).
16. SVG - Scalable Vector Graphics 1.1
17. TDD - Test-Driven Development
18. TS - TypeScript 2.7 (the latest version at the time of writing)
19. UI - User Interface
20. UMD - Universal Module Definition

Attachments

1. /code/readme-customize.md *(Implementation of a customization mechanism in React)*
2. /customization_angular *(An example implementation of a customization mechanism in Angular)*
3. /code/src/_benchmark *(Code for running benchmarks along with their description)*
4. /code *(All code of the RMI along with readme files)*
5. /code/readme-testing.md *(Advice about and APIs of testing tools)*
6. /code/readme-i18n.md *(API and demo of internationalization)*
7. /code/sus_test *(Details and results of System Usability Scale test)*
8. /code/readme-acceptance.md *(Acceptance tests)*