technische universität
dortmund

Bachelor Thesis

DSL-Driven Generation of User Documentations for Web Applications

Mukendi Mputu
November 2021

Supervisors:
Prof. Dr. Bernhard Steffen
M.Sc. Alexander Bainczyk

Technical University of Dortmund
Department of Computer Science
Chair 5 for Programming Systems (LS-5)
http://ls5-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

One of the most challenging tasks in software development is developing a long-term strategy for creating, managing, and updating the documentation for the end users as the software product develops, adapts, or increases in complexity [6]. The challenge lies in the fact that the development team tasked to document the software must do it manually by first selecting the information most valuable to the end users, structuring it, and then when required updating it. In other words, the developer must reproduce the steps or several complete scenarios a potential end user would go through and then document where helpful information is necessary to reduce the time needed to understand the main functionality of the software.

Completing this process once does not necessarily free the developer from the task. It is a cycle that must be repeated every time an update is introduced or a relevant part of the program has been changed. For a small static Web page, this might not sound dramatic. Still, considering complex applications developed by multiple teams, this task can raise the cost in time and resources as the information manager must gather the information about the whole project for the design of the documentation [1]. So, the problem is to find a way to automatically go through all those steps and generate parts of the documentation model extended with semantic description and later assemble those to create complete documentation.

Software projects are bound by budget and deadline requirements, and this explains why the automation processes become essential, leading to an increase in the development time can be increased [15]. It is, therefore, of significant advantage for the developing team *and* the end user that the generation of user documentation is modeled in such a way that it always reflects the most current development status of the software product [31]. This thesis introduces the use of a graphical domain-specific language (DSL) – a programmatic

language adapted to a specific domain problem [25] – as an alternative way of modeling end user documentation.

1.1 Objectives

This thesis proposes a solution for automatically generating end user documentation for Web applications utilizing a graphical DSL, a similar approach to [9]. We developed an application for modeling different user action sequences in a Web application using the Cinco SCCE Meta Tooling Framework [12] and for subsequently generating end user documentation in Markdown syntax. The Cinco framework is a generator-driven development environment for domain-specific graphical modeling tools. It is based on the Eclipse Modeling Framework and Graphiti Graphical Tooling Infrastructure and aims to hide much of their complexity and intricate Application Programming Interface (API)[1]. For evaluation purposes, we document a task management Web application of our own. Nonetheless, the applied method can be extrapolated to any other Web application since the elementary building blocks of those applications are the same.

The project has a Maven nature, which allows us to manage the package dependencies and take advantage of its build life cycles for building and distributing the application[2].

The Markdown-based documentation is supplemented with screenshots taken with the Web browser automation engine, which drives the Web application to replicate user actions. In addition to that, we serve the generated Markdown files containing references to the screenshots as a static site.

1.2 Related Work

Documenting a software application – be it as a desktop application or a Web application – has always been a daunting task for the developer. Nonetheless, it is a task many developers are neither enthusiastic nor motivated to do [20]. This piqued the interest of many scholars on the subject, and earlier comparable solutions were presented in the past:

[14] also proposed in 2015 the Écrit Toolkit, a solution in which the Eclipse Rich Client Platform [16] application model is used to provide a semantic description of the

[1] More information can be found at https://cinco.scce.info/
[2] The code for the application can be found here https://github.com/MukendiMputu/UserDocGenerator

model element. Those semantic descriptions were later aggregated to a documentation text complying with ISO/IEC 26514 [14]. The generation steps consisted of an application model with the semantic description being analyzed and converted into an EcritDocument model that was used as input for a Document Outputter, which produced LaTeXor HTML code. A preexisting application model based on the Eclipse RCP was required to begin the documentation project, and coding knowledge was also necessary to verify the produced documentation. The project is no longer under active development, partly because, at that time, few companies built applications using the Eclipse RCP model.

GuideAutomator [24] was proposed in 2016 as part of a bachelor's degree thesis. The approach was to provide Markdown files with short JavaScript chunks that determine how to capture each screencast [7]. This implied that preexisting Markdown files were required as input for the generator, which then executed the JavaScript code snippet using Selenium. This approach limited itself in a way that knowledge of the JavaScript programming language was required. The Markdown file had to manually be generated before the generator's execution, which did not support the rt the reusability of such Markdown file. According to the last GitHub commit, which goes back to September 2018, the project is no longer being actively developed.

In this thesis, we decided to keep the philosophy adopted by the developers of the Cinco framework. Our approach prioritizes the reusability and simplicity of our graphical language amongst the most important features we aimed to implement. Those features enable non-programmers to design graph models of the Web application they desire to document by arranging the graphical elements our language proposes to a logical sequence of actions. The editor takes care of the generation of the executable application code and all the documentation files necessary. Furthermore, we also assist the documentation designer in creating correct models by checking the syntactical correctness of the diagram at runtime.

1.3 Outline

Chapter 2 presents the preliminary notions on which our approach is based: section 2.1 describes the requirements of end user documentations, whereas section 2.2 and 2.3 explain the concept of model-driven development. Next, section 2.4 introduces the Cinco SCCE Meta Tooling Framework, in particular the two metalanguages (Meta Graph Language (MGL) and Meta Style Language (MSL)) that constitute the meta-specification (section 2.4.1 and 2.4.2) of our graphical domain-specific language. Lastly, section 2.5 presents the Web applications we use as ongoing example throughout this thesis.

In Chapter 3, we describe the application developed with the CINCO framework and its special features. First, we briefly describe in section 3.1 what our graphical language is about and how it looks in the graphical editor in section 3.2. Then section 3.3 describes the different graph models for modeling the end user documentation, whereby section 3.4 eludes the checks applied to verify those models. Finally, section 3.5 rounds off the chapter with an outline of the generator classes written in Xtend – a Java dialect that offers much flexibility and expressiveness in its syntax.

An in-depth explanation of the implemented methodology is presented in chapter 4. It starts with an evaluation of the chosen work method in section 4.1, then moves on to system setup and a discussion of the outcome of the strategy used in this thesis (section 4.2 and 4.3). Lastly, chapter 5 wraps up the thesis and considers future enhancements to the application program.

Chapter 2

Preliminaries

This chapter introduces the key concepts that were used for achieving the objectives assigned for this thesis. We begin by considering the format of our end user documentation as recommended by known standards (from the ISO/IEC[1] and IEEE[2]) and the technologies available to assist us in completing this task. The CINCO Meta Tooling Suite will next be used to introduce the principles of model-driven programming, as well as introduce the metalanguages that ground our graphical models and quickly describe the product generated from those models. Finally, we elude the overall generation process to achieving our goal (see figure 2.1).

[1] International Standards Organization (ISO) and International Electrotechnical Commission (IEC)
[2] Institute of Electrical and Electronics Engineers (IEEE)

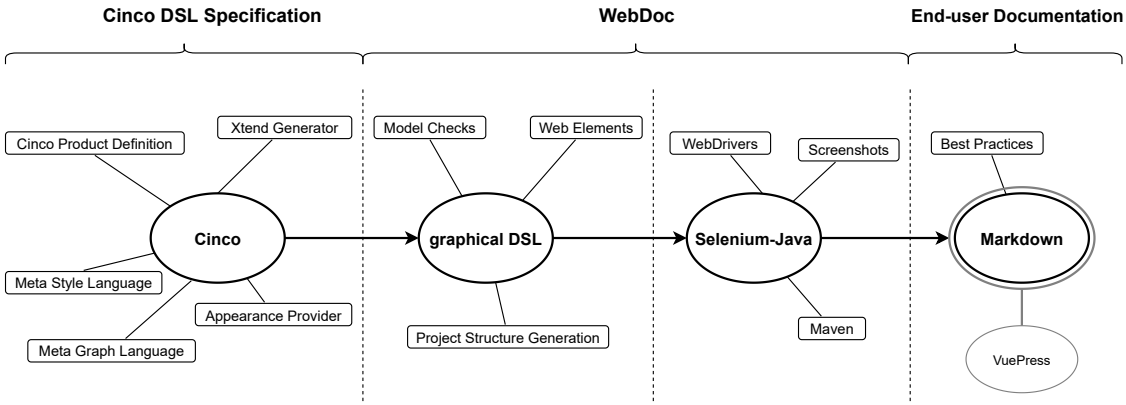


Figure 2.1: Development path of the end user documentation

2.1 End User Documentation

The value of a good software product, in fact, of any product destined for the consumer, is determined by how effective the end user can learn to use and work with it. Hence, putting a great effort to generate and manage sound, well-structured, and well-written documentation is as essential as the development of the product itself [3].

End user documentation aims to provide application end users with the knowledge to interact with the software appropriately. It is an integral aspect of the development process and serves as a link between the product creator and the end user [4]. By increasing the structure and usability of the information supplied to end users, the developer not only reduces the number of support calls but also improves the product's and the developing company's reputation [1].

In this section we will describe the main characteristics of an end user documentation, focusing on the standards established by the ISO/IEC and the IEEE [1, 3], before writing about the specifics of documenting Web applications by giving an example of a documented Web application. We will also present the technologies used currently for creating and managing such documentation. Our main goal is to produce an entire end user documentation page from a graphical model. It is worth noting at this stage that the type of documentation to be created will depend on how the documentation designer lays out the model. Meaning that the model could be designed to represent technical documentation – requiring deep knowledge of the application documented – or it could also be structured to produce documentation for simple end users with no technical knowledge at all of the underlying Web application [6].

2.1.1 Documentation Characteristics

Since our focus is primarily on Web applications, we will be documenting the User Interface (UI), which is composed of Web elements (such as buttons, navigation links, checkboxes) the user can interact with. This settles implicitly the type of documentation we aim to create: a step-by-step guide for navigating UI to reach the expected result. Nonetheless, the ISO/IEC/IEEE Standard 26511:2012 [2] mandates, i.e., that completeness and accuracy should be of the utmost importance when designing the end user documentation.

A good documentation should respond to three essential questions: *why* the application has been developed, meaning the problem the application intends to solve, *what* is the end user supposed to do to get to the solution quickly and efficiently and *how* it is achieved [3]. Information gathering is a task we leave to the information manager. However, the last question is better answered by providing visual help as screen captures or any illustration that fastens the understanding of the documentation.

As previously stated, we plan to document Web applications from the perspective of the application's end user. That means that the documentation developer must consider the product's audience, and identifying the core jobs and activities performed by the ordinary users should be an essential element of the design process [3]. Thus, our solution puts the developer in charge of the production of a detailed and comprehensive documentation.

2.1.2 Documentation Software Tools

Choosing the format to bring the documentation to the end user is as important as the rest of the milestones of the planning process. A poor decision in this area might limit the documentation's ability to reach many consumers, rendering it useless. Choosing technologies that have already received widespread adoption is a wise move. That ensures a broader reach and prior knowledge within the same audience:

Markdown

Markdown is a lightweight markup language created by John Gruber[3] in 2004. Since then, it has established itself as one of the world's most popular markup language [13]. Gruber states on his webpage that Markdown is comprised of formatting syntaxes that

[3]John Gruber's official project website https://daringfireball.net/projects/Markdown/

can be applied to plain-text files and optionally a converter to other markup languages files like Hypertext Markup Language (HTML). Hypertext means the link that connects Web pages, either within the same website or between different websites [21]. On the Mozilla Developer Network website[4], HTML is described as the fundamental building block of any Web page that uses markup syntax to annotate text, images, and other content for display in a Web browser.

Markdown's popularity makes it an excellent choice for our project since the description of the UI is first made as plain-text and then transformed to a rich-text format using markup syntax as depicted below. Moreover, Markdown files can be opened and edited with any kind of text editor available. Finally, a tremendous amount of documentation about Markdown syntax is available online for the interested reader to get started.

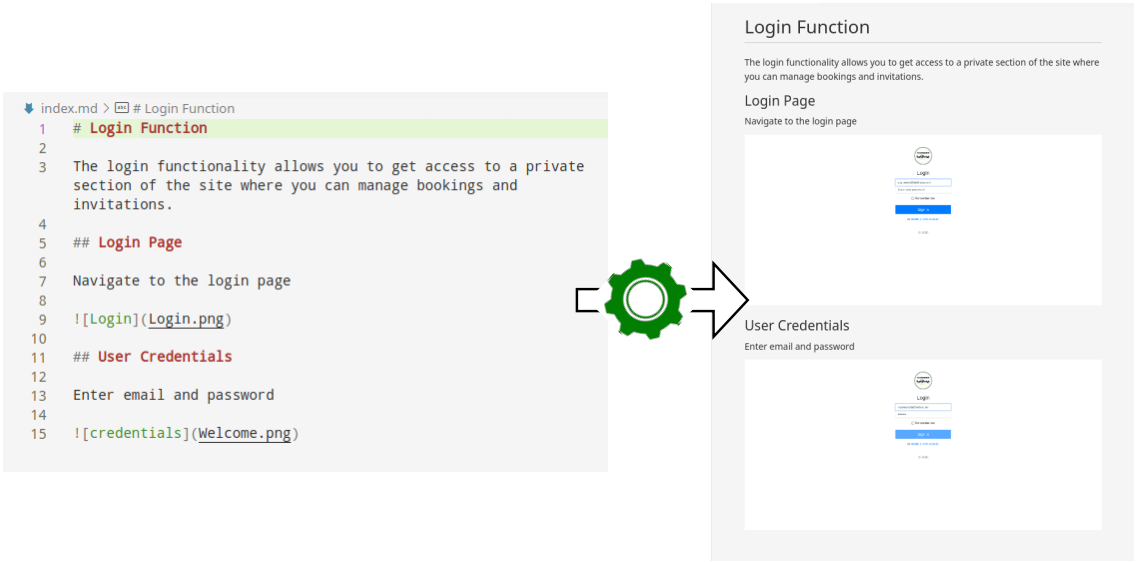


Figure 2.2: Plain-text transformation with Markdown

A plethora of applications exists for transforming Markdown files into HTML files to be rendered in a Web browser application – such as MacDown [1] for Mac users, ghostwriter [2] for Windows and Linux, or an online tool like Dillinger [3]. Even so, we choose to work with one framework that has also established itself throughout the developer community and that works especially very well with Markdown files: VuePress.

[4]MDN: https://developer.mozilla.org/en-US/docs/Web/HTML
[1]MacDowns: https://macdown.uranusjr.com/
[2]Ghostwriters: https://wereturtle.github.io/ghostwriter/
[3]Dillinger: https://dillinger.io/

VuePress

VuePress is a minimalistic VueJS-powered static website generator. VueJS [32] is an open-source JavaScript framework designed by Evan You[4]. It is minimalistic because it only takes a few lines of configuration and two commands to create and launch a static website. It is resilient and up-to-date since it is open-source software and is maintained by a large community of contributors.

By respecting the VuePress conventional folder structure, it can explore folders, get Markdown files, and convert them to HTML files. The default appearance of the generated website is defined by a customizable configuration file included in the project folders. By simply adding certain configuration elements, one may create, i.e. a sidebar menu and a navigation bar. VuePress produces a static website by building hyperlinks between HTML files in the project folder hierarchy. That means that subfolder Markdown (or HTML) files are rendered as menu items on the folder's HTML page. A server-rendered website with a live reload mechanism is constructed and launched after inputting the correct command. Thus, every modification to the Markdown files is immediately visible on the website. The final step is to create the website that will be saved in a particular folder and exported to any other website.

We will use VuePress to statically launch the end user documentation as a website, relieving the developer of configuring the server. The principal task remaining is, therefore, to create a solid documentation model.

2.2 Core Principles of Model-Driven Development

This section lays down the fundamentals of the model-driven development (MDD) – also referred to as model-driven software development (MDSD) [18] – using the Cinco SCCE Meta Tooling Framework, as well as the steps necessary to get up and running with the framework. The term MDD refers to a development paradigm where the functionalities of the software are first specified as models, from which then executable code can be automatically generated. A DSL close to the problem domain is needed to create models to represent the software system. Application domain experts know the application structure, which they represent in the form of models. Domain experts and application programmers must agree on how the specification language determines the syntax and semantic of the model elements.

[4]Eva You's website https://evanyou.me/

The core principles of MDD are that software development is accelerated by providing a simple but efficient abstraction of the software structure as a model. Those model abstractions, representing real-world objects, are transposed through a series of model-to-model or model-to-code transformations [28]. Our work is to utilize the DSL provided by the CINCO framework and design our graphical DSL to generate a functioning Selenium-Java application (Selenium is a suite of application tools for automating Web browsers) that takes screenshots of the different Web application states.

As opposed to the standard development method, applying a graphical model to layout the different user sequences allows even non-programmer (here the domain expert with much more expertise on how to design excellent software documentation) to document the features offered by the Web application. Nonetheless, the programmer has the tasks – in collaboration with the domain expert – to specify the meaning of each model element for the code generation process.

When applied correctly, the result of the model-driven development process is a tailored application to a domain. That reflects one of the main advantages of MDD: the accuracy of directly targeting the specific problem [10]. Besides, it is still possible to change the DSL to adapt to the new challenges emerging during the development process. That can be iterated until the specification reaches preciseness wanted to solve the problem.

2.3 Domain-Specific Language

A domain-specific language (DSL), as described before, is a language adapted to a specific development domain. In [23] a succinct analogy to DSLs is given by saying that it is comparable to a tool specially crafted only for one specific task as opposed to general programming languages, which can be seen as tools for multiple different tasks. Just as one would start with a blueprint to construct a mechanical tool, designing as DSL required similar steps. One must conceptually layout the behavior and eventually – in case it is a graphical language we seek to design – the look of each element that can be used in our DSL.

Blueprinting our DSL is equivalent to defining a metalanguage or meta-DSL tool. *Meta* literally means *situated behind or beyond*[5]. So, metalanguage is a descriptive language that comes before and describes the meaning (semantic) and the relationship between the objects of our target language. Put differently, the meta-DSL is the abstract syntax, and the DSL concrete syntax is the result. It is feasible to work our way up the meta-

definition modeling hierarchy until we reach a self-referencing language, as is the case for the Unified Modeling Language (UML).

In our case, we want a graphical language for creating models that represent the different parts of a Web application and how the end user can interact with them. The metalanguage to our graphical DSL is provided by the Cinco SCCE Meta Tooling Suite. It comprises the Meta Graph Language (MGL), the Meta Style Language (MSL) and the Cinco Product Definition (CPD). All have been constructed using Xtext [8], a language Workbench for writing textual DSLs [22]. The next section explains these concepts in detail.

2.4 Cinco SCCE Meta Tooling Framework

The Cinco Framework is a generator-driven development environment for domain-specific graphical modeling tools [12]. The chair of programming systems currently develops it at the Technical University of Dortmund amongst many other projects of the SCCE Group [1], which aims at allowing the application-domain experts, rather than programming experts, to take charge of the development tasks [26]. One of the great features of this framework is that it allows us to generate an entire editor application with just one click from a simple textual specification language – the MGL mentioned in the previous section.

The MGL, together with the MSL, form the metamodel from which Cinco generates a ready-to-run modeling tool called Cinco product. The MSL defines the look for each graph element, the font, and the color of the text displayed in the graphical model [22]. The created metamodel is based on Ecore, the metamodeling language of the Eclipse Modeling Framework (EMF). Furthermore, the Graphiti framework generates the corresponding graphical model editor. Additionally, there is a button to trigger code generation in the created editor. Chapter 3 is devoted to our Cinco product (the editor application), in particular section 3.5 gives an in-depth explanation of the generation process.

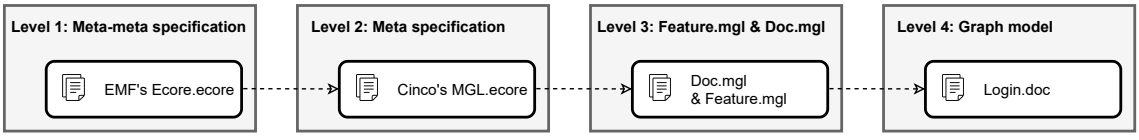


Figure 2.3: Hierarchy of our graphical DSL specification

The full generation process of a graphical modeling tool (of a Cinco Product Application) comprises four essential (meta) levels [23] as shown in figure 2.3. The first

[1] https://www.scce.info/

level, associated with Eclipse Developer's role, is where the Ecore.ecore and GraphitiDiagram.ecore metamodel are developed. The second level is where CINCO Developers of the Chair 5 for Programming Systems developed the MGL.ecore using the metamodel from the first level. That second-level metamodel, in turn, becomes metamodel of the third specification level. The level where the CINCO Product Developer operates and where this thesis comes in, see fig. 2.3. That means the Eclipse Developers are now at the meta-meta level, the CINCO Developers at metalevel of the specification elaborated in this work. For the remainder of the chapters, we will be focusing on the third and fourth specification levels because the fourth level, for example, is where the CINCO Product Users use the generated graphical editor to create the domain-specific models.

2.4.1 Meta Graph Language

The MGL sketches the behavior, the constraints and gathers all the graphical components that will constitute the model elements that come into use in every graph diagram created in the modeling tool. This is where we start developing our editor application: the main elements that can be define in a MGL are **nodes**, **containers** and **edges** as illustrated in listing 2.2.

The main graph model, the `FeatureGraphModel`, is the one containing all the features of the Web application we want to be documented. For instance, our Web application provides the login feature as the starting point. After successfully logging in and accessing the dashboard, task lists can be created or deleted; tasks can be added to or removed from them. The graph diagram can be assigned a name and description. We specified `.feat` (short for feature) as model extension. Moreover, since the login feature, i.e., requires a series of user actions to be performed, the FeatureGraphModel contains another graph model type, where those sequences of actions are modeled: the `DocGraphModel`. The DocGraphModel contains the meta specification of all the frequently used Web elements we want to use in our graph models. It also comprises sectioning elements to structure the diagram and edges to connect them. The listing below shows an excerpt from the concrete FeatureGraphModel implementation.

```
1  graphModel FeatureGraphModel {
2      iconPath "icons/16/feature_16.png"
3      diagramExtension "feat"
4      containableElements(FeatureContainer,DocNode[1,*])
5      attr EString as modelName := "New Feature"
6      attr EString as description := "New Description"
7  }
8
9  container FeatureContainer {
10     style featureContainer("${title}")
11     containableElements(Start[1,1], Stop[1,1],DocNode[1,*])
```

```
12      attr EString as title := "New Feature"
13      @multiLine
14      attr EString as documentation := "This feature is about ..."
15  }
16
17  node DocNode{
18      style docNode("${mgl.modelName}")
19      prime docMgl::DocGraphModel as mgl
20      attr EBoolean as createScreenshots := true
21      incomingEdges (Edge[1,1])
22      outgoingEdges (Edge[1,1])
23  }
```

Listing 2.1: Excerpt from the feature.mgl, meta-specification of the FeatureGraphModel

It can be seen from listing 2.1 that we define a container element to hold the DocGraphModel instances (see lines 9 and 17). To integrate a whole DocGraphModel inside a FeatureGraphModel, we used the so-called *PrimeReference*, which is a CINCO feature that allows referencing another model by only one attribute of a node or container [12]. It is applied by adding the keyword `prime`, as we did in line 9. By doing so, we can drag and drop a DocGraphModel diagram into a FeatureGraphModel diagram to incorporate it.

As for the DocGraphModel, we specified four categories of model elements that are used while designing the diagram: the most important ones, the Web elements, represent the UI element the user can interact with. Then we have the Selenium action nodes, which perform specific actions to drive the Web browser using the Selenium WebDriver. Those are, i.e., the Navigation node to change from one Web page to another and the Screenshot node to capture the current application state as an image. Finally, we have the semantic element `Comment` to give a descriptive text to the screenshots and the basic elements (start and end as well as the section node), which help structure the user sequence graph.

```
1   graphModel DocGraphModel {
2       iconPath "icons/16/sequence_16.png"
3       diagramExtension "doc"
4       containableElements(*)
5       attr EString as modelName := "UserSequence"
6       @multiLine
7       attr EString as documenation := "Lorem ipsum dolor et si met"
8   }
9
10  node Screenshot {
11      style screenshotNode
12      incomingEdges (Transition[1,1], Anchor[1,*])
13      outgoingEdges (Transition[1,*])
14      attr EString as pictureName
15      attr Comment as description
16  }
17
18  node Input extends WebElement {
19      style inputNode("Input: ${content}")
20      attr EString as content
21      incomingEdges (Transition[0,*])
```

```
22      outgoingEdges (Transition[0,*])
23  }
```

Listing 2.2: Excerpt from the Doc.mgl, meta-specification of the DocGraphModel

For demonstration purposes, we kept our example listings short. An exhaustive list of all the usable elements and annotations can be found on the CINCO's documentation page[1].

2.4.2 Meta Style Language

The appearance of all the elements defined in the MGL is laid down using the textual Meta Style Language (MSL). As explained in [11], three essential elements constitute the design of a MSL model, namely: **appearance**, **nodeStyle** and the **edgeStyle**.

Each nodeStyle specification makes use of an appearance element, which in fact determines the attributes like background color, the thickness of the drawn lines and so on (see listing 2.3). The nodeStyle is hierarchically composed of a shape that can be given a `size`, `position` and a `text` element with a `value` attribute that takes a (format) string (line 12) that will be display in the graphical model. We can say that it is left to the CINCO product developer's imagination to style the elements as seen fit.

```
1  nodeStyle featureContainer(1) {
2      rectangle {
3          appearance extends default {
4              background (246,245,244)
5          }
6          size (300,75)
7          text {
8              appearance {
9                  font("Sans", BOLD, 10)
10             }
11             position (LEFT 5, TOP 5)
12             value "%s"
13         }
14     }
15 }
```

Listing 2.3: Excerpt from feature.style to be applied to feature.mgl

It is also worth mentioning that the concept of inheritance from the Object Oriented Programming (OOP) can be applied between metamodel elements of the same type. Thus, avoiding repetitive definitions of the same attributes within multiple different elements and allowing some elements to extend the properties of the parent elements. For example,

[1]Wiki page : https://gitlab.com/scce/cinco/-/wikis/Cinco-Product-Specification

we see in line 3 that the `featureContainer` appearance extends the default one and at the same time redefines the background color.

2.4.3 Cinco Product Definition

In the Cinco Product Definition (CPD) is where it all comes together. Herein, the CINCO Product Developer must provide key information like the CINCO product name, at least one or more MGL files to be included in the generation process. Optionally, one can set up a splash screen with branding images, add descriptive text about the application and specify plugins and features [11]. The listing below gives an insight into the CPD specification language.

```
1  CincoProduct UserDocumentationTool {
2      mgl "model/Feature.mgl"
3      mgl "model/Doc.mgl"
4
5      splashScreen "branding/splash.bmp" {
6          progressBar (37,268,190,10)
7          progressMessage (37,280,190,18)
8      }
9
10     image16 "branding/Icon16_dark.png"
11     image32 "branding/Icon32.png"
12     image48 "branding/Icon48.png"
13     image64 "branding/Icon64.png"
14     image128 "branding/Icon128.png"
15     linuxIcon "branding/Icon512.xpm"
16
17     about {
18         text "WebDoc is a DSL-driven generator of end user documentation
   for Web applications. It is a bachelor thesis project developed with the
   Cinco SCCE Meta Tooling Suite ( http://cinco.scce.info )."
19     }
20
21     plugins {
22         info.scce.cinco.product.userdocumentation.edit,
23         info.scce.cinco.product.userdocumentation.editor
24     }
25 }
```

Listing 2.4: UserDocumentationTool.cpd

2.4.4 Xtend Generators

We need to create two different application folder structures from our model diagrams for our application documentation: the first is the Selenium-Java application, which replays the modeled user action sequences and takes screenshots as laid out by the designer. It follows the specific Maven project structure, for which we applied the rule of convention

over configuration as recommended on the Maven Apache [1] website and added Selenium as a dependency. The second project structure we generate following convention is the VuePress project that comprises the Markdown files containing all semantic text the documentation developer specified as description or comment in the model elements. We create the modeled end user documentation by following each sequence beginning from the start node to the end node and constructing a cohesive documentation text.

This generation approach utilizes the Java's and Xtend's text templating feature, which is based on the generation pattern used in the Java Application Building Center (jABC) [29, 30]. In fact, many generator classes in our example project implement and extends interfaces from the generator runtime and template package of the jABC Cinco meta plugin.

Additionally, other template classes are implemented to create the configuration files, the application class files, and project files. In chapter 1 section 1.1 we provided a link to the GitHub repository, where the complete application code can be found. In the following chapters, we show the output of the generator and feature classes while introducing our application as an ongoing example.

2.5 Tasks management Web application – TODO-App

The TODO-App is a Web application entirely generated with the DyWA Integrated Modeling Environment (DIME) [9], a development environment for creating Web applications. DyWA stands for dynamic Web applications; it is the container used to host the generated application. While the modeling and the code generation happen in DIME, DyWA provides support for the product deployment phase, constitutes the runtime environment, and manages the data persistence [9].

The TODO-App is an application that helps manage lists of tasks to be done. It provides a logged-in user with the possibility to create such lists, add various tasks to them, and after their completion, remove them [9]. Users also can add co-owners, which are other existing users, that can manage that respective list with them. The application is launched in development mode, meaning that it is accessible via a local Web address (`http://localhost:8080`) from the host it has been started in. The TODO-App is ideal for demonstration purposes since it presents all the Web elements necessary to a Web application interface, as depicted in figure 2.4.

[1] https://www.apache.org/

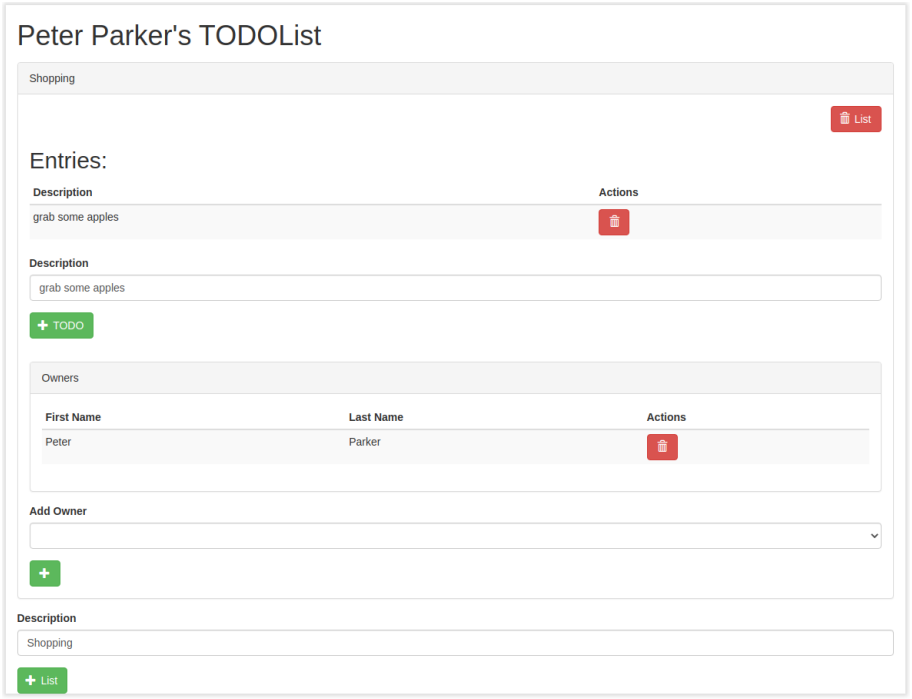


Figure 2.4: Impression of the TODO-App Web interface

DIME's model-driven development concept was our starting point. Thus, the graphical elements meta specification focuses on specifying elements that represent the graphical user interface (GUI) and those that implement processes. The latter are elements whose semantic implementation will direct the Web browser to recreate the procedures necessary to construct the intended documentation. However, in our scenario, the concept of data permanence is not required. Furthermore, even though our concept grounds on DIME, it does not mean that our application is only for DIME-generated Web apps. In this respect, any navigable Web application can be modeled with our editor and thus documented.

Chapter 3

WebDoc - Web Application Documentor

Having described the CINCO DSL specification for the graphical modeling tool, and we will now go over the editor created from it. First, we begin with a brief description of our graphical DSL; then, we demonstrate the critical building elements of our documentation architecture before presenting our CINCO product: WebDoc[1]. Secondly, we will show how to use these graphical elements to simulate a specific user workflow on the website we are documenting later. Finally, we demonstrate how to construct the target application that generates the Markdown files shown in the Web browser using the editor's built-in generator.

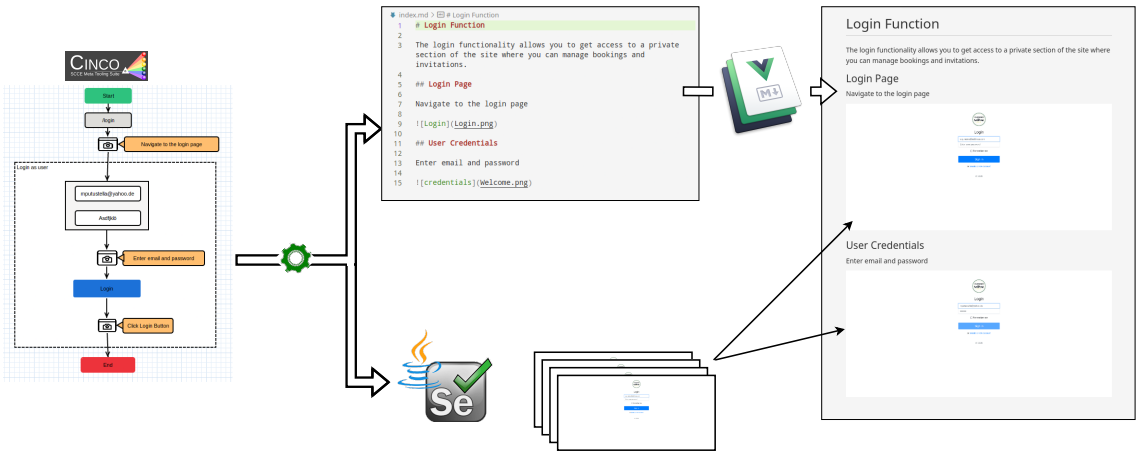


Figure 3.1: Generation process of End user Documentation

[1]GitHub repository: https://github.com/MukendiMputu/UserDoc

3.1 Graphical DSL

Our graphical DSL primarily consists of node components with various appearances that match the concrete Web items they represent and connector elements that connect the nodes to sequence graphs. This approximated replication aims to provide the developer with a sense of control over the interactable UI elements rather than replicating all potential Web elements. With those elements in hand, the designer can easily simulate a user activity by arranging nodes according to the Web application's navigation path. This graphical language is domain-specific because it is designed for modeling Web applications within a Web browser; for example, modeling documentation for a desktop application would be impossible.

3.2 Graph Editor

The WebDoc editor, shown in fig. 3.2, is mainly composed of the canvas in the middle of the working environment (1). The developer can drag and drop model elements from the palette on the right-hand side (2), grouped into categories. On the left-hand side, we have the project explorer showing the current project structure (3). The main model files have the extensions .doc and .feat, where the latter is the entry point of the documentation application.

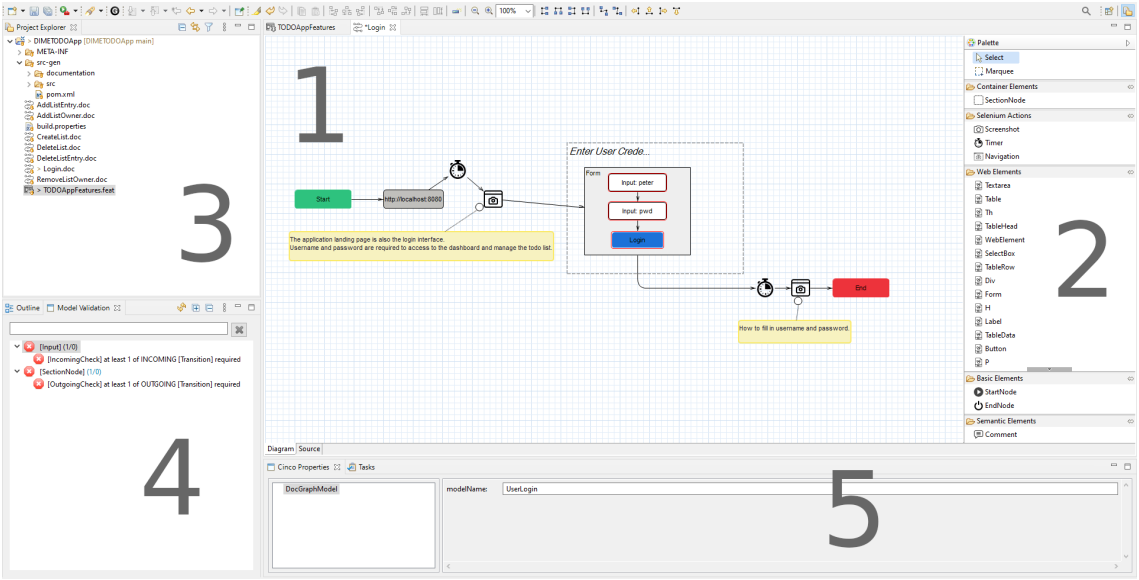


Figure 3.2: Cinco Product Application - Graph model editor

All diagrams that are open in the editor are checked in the background for compliance with the requirements described on the metalevel and shown in the Model Checking

view right beneath the project explorer (4). There is also the CINCO property view (5) that displays the attributes and values of any selected element in the editor. In this view, the developer can modify those values if the attribute field allows it.

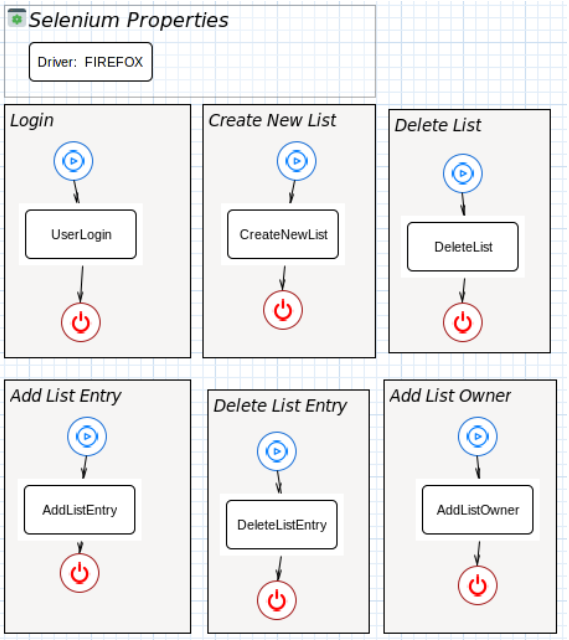
3.3 WebDoc's Model Elements

On a Web page rendered in the Web browser, there is a small amount of HTML elements the user can interact with. Most of them reside within the HTML forms, tools that are used for collecting data from the user or allowing them to control a user interface [21]. In addition, input fields (such as password, text area, checkbox) and buttons are prominent. Those are the Web elements that primarily constitute the palette of our model elements. In the previous chapter, we defined two different MGLs: one for modeling the Web application features and the other one for modeling the user actions that make up those features.

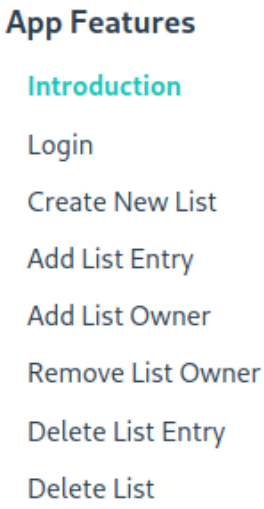
3.3.1 FeatureGraphModel

The FeatureGraphModel is the application starting point specified by the `feature.mgl`. Here, the developer groups all the features that need to be documented in feature containers. For example, figure 3.3a shows a sample of the features modeled for our task management application, including the ability to log in, create a new list, add or remove a task from that list, and delete the entire list. The documentation website's resulting sidebar is depicted in Figure 3.3b, albeit the "Introduction" is not an application feature, instead the introductory page displaying the various functionalities. In addition, our application offers the possibility also to add a new list owner already existing in the system as a regular user.

On the top left-hand corner, there is a property container holding the WebDriver property, whose value is set to `FIREFOX`. It is one of the additional model elements that help the developer define configurations that otherwise could not be modeled but are still essential for the execution of the application. This property value will be assigned to the Selenium WebDriver variable in the Java class. Considering the fact that the executable file for the chosen WebDriver must already exist somewhere in the file system, and the path to it must be specified here in the property view.



(a) Features in the FeatureGraphModel



(b) Menu structure on the documentation website

Even though the features follow a logical workflow order, they are still independent and generated separately. However, as indicated in the last chapter, this separation does not preclude reusability since it is possible to embed a whole DocGraphModel inside an existing different one. Consider the CreateNewList functionality, which requires the user to sign in to create a new task list. Instead of repeating the login sequence in this one, the documentation developer may drag and drop the UserLogin.doc file into the diagram of the new model graph and connect it to the sequence as if it were a regular graph node. (see figure 3.4). Double-clicking on an imported subgraph sends straight to the original graph model. This double-click action was implemented by annotating the associated node specification with the `@doubleClickAction` annotation and providing a custom action class that holds the implementation logic. Furthermore, by unchecking the createScreenshots checkbox in the property view of a subgraph, the designer can disable the creation of screenshots for that specific model, which is enabled by default. This prevents the same screenshot from being taken several times when the subgraph is reused.

The semantic elements in the feature graph model are incorporated into the underlying featureContainers for reasons of simplicity. Clicking on a featureContainer opens the Cinco property view, which includes a multiline input form called 'description.' This is where the documentation designer can offer text content for the Markdown documentation files providing descriptive text for the diagram objects. These information texts will be gathered during the code creation process to create the complete documentation.

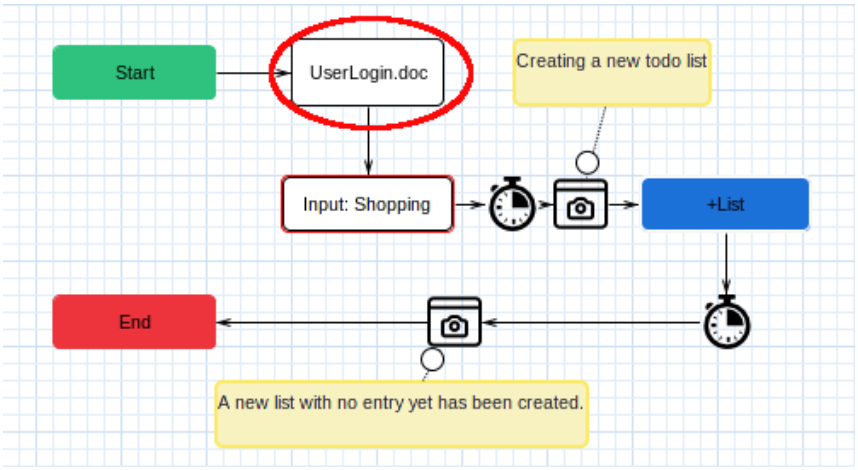


Figure 3.4: CINCO product - Reusing Login graph model in the CreateNewList graph model

3.3.2 DocGraphModel

Figure 3.5 illustrates the steps a user would have to carry out in order to log into our example Web application, as well as the Web elements that will be interacted with. They create the user login sequence for our Web application when connected in a logical sequence.

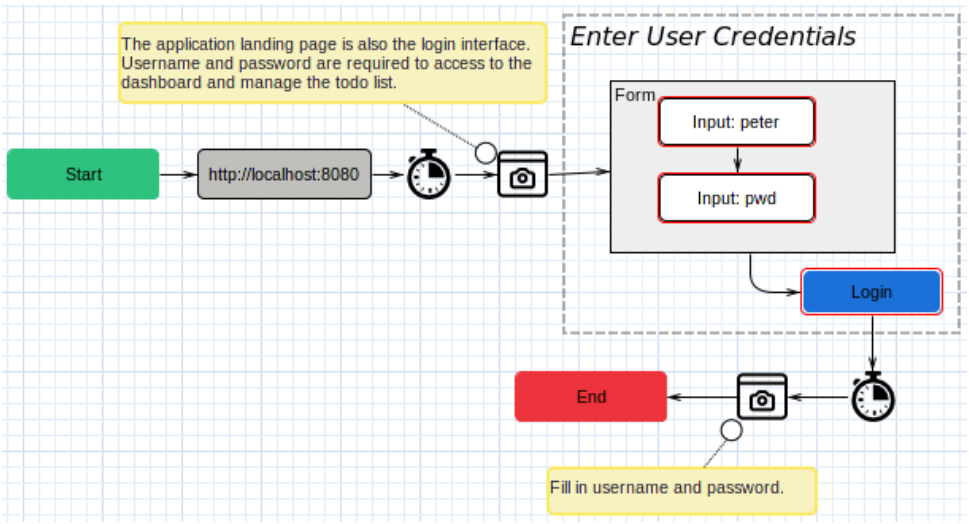


Figure 3.5: Example of a user workflow: here the login sequence

The sequence begins with the start node, which is the starting point of every user sequence, then comes the Navigation node used to navigate from one Web page to another. Next, the timer node waits explicitly for the number of seconds determined by the designer in the property view and for a certain condition to become true before continuing. Those ExpectedConditions are i.e., `presenceOfElementLocated`, `elementToBeClickable`, just to name a few. If we take, for example, the condition `presenceOfElementLocated`, the timer holds the

WebDriver execution for 3 seconds, checking every 500 milliseconds if the targeted Web element appears on the page. Because some items take time to load completely, this step is required for capturing all page elements when taking the screen capture. Right after the timer comes the Screenshot node that captures the current application state, displaying the application landing page, the login page. In the property view, one can give a unique file name to each image to be saved. The comment node allows the document writer to contribute descriptive information about the image that will later be appended to the Markdown file as an image caption. Next comes the Section node that regroups form elements for entering and validating user credentials. It can be seen from fig 3.5 that all input nodes, as well as the button node, have a red border that represents the highlighted state of the element. The model designer may see which elements will be highlighted in the following screenshot, after which the login sequence will end.

The purpose of the Web element nodes is to allow the concrete HTML elements in the browser to be addressed. This implies that appropriate actions can be applied to such a representation of an HTML element. For instance, considering the input field in the picture below, the action of typing in a text is made possible by providing a property variable `content`, whose value is then displayed inside the node element (i.e., ”peter” and ”pwd”). The same holds for button elements, which can be applied to the click action, or for select boxes, which can be dropped down to reveal the options they contain. In addition to that, all Web elements can be highlighted either by setting the `highlighted` property to true or by letting the Highlight node under the *Selenium Actions* category take care of it. The screenshot coming after will have that specific element surrounded with a red border as well (see fig. 3.6).

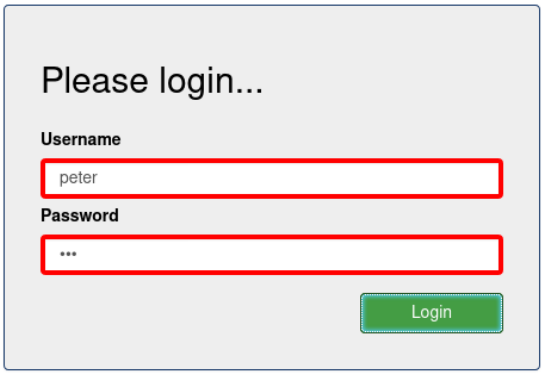


Figure 3.6: Login pane with screenshot of the highlighted input element

In addition, each node element includes a description attribute, which, as discussed in the preceding section, provides additional content for Markdown files. We decided to hide this attribute in the property view to avoid overcrowding the diagram with semantic

components describing each and every node member. All model diagrams are traversed during the generation process, and all descriptive texts are collected into paragraphs, then sections, and finally full Markdown files, depending on the feature they are included in. As a result, the documentation content is organized in the following fashion (see fig. 3.7): the featureContainer title is rendered as page headings, the DocGraphModel name is displayed as section headings with all other element descriptions underneath it, and so on.

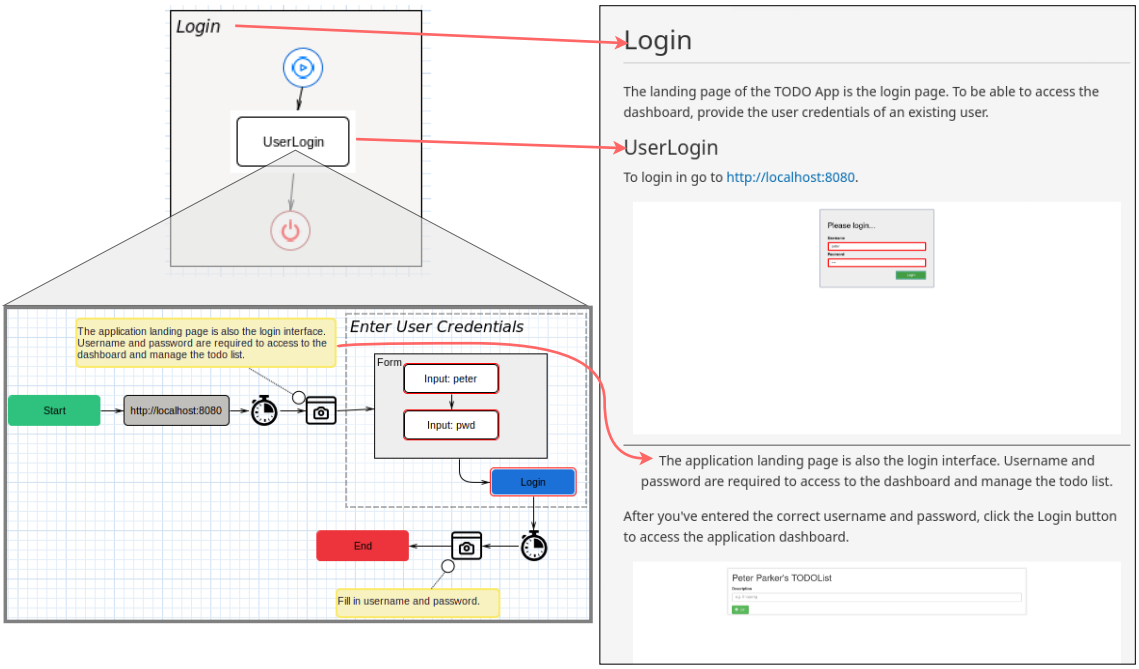


Figure 3.7: Structural mapping of models to Markdown

3.4 Graph Model Checks

Modeling functional graph diagrams is crucial for the correctness of the generated application code. Especially the Selenium-Java application code must correctly replicate the user sequence to capture the correct application state as the user would. This task is particularly daunting if the navigation graph of the underlying Web application is vast.

Therefore, we need to check that two distinct graph models do not have the same name or that for each user activity, there is a distinct path and no loop inside that path. These checks are reminiscent of various graph theory problems, such as determining whether a path from the start to the end node exists and whether a graph is loop-free. The CINCO framework incorporates the MCaM plugin that does this operation [11]. specifying the `@mcam("...")` or the `@mcam_checkmodule("...")` annotation with the corresponding parameter

in parentheses activates the plugin. Also, nodes and model graph attributes can be specified with the `unique` qualifier to enforce the uniqueness of the model name at design time.

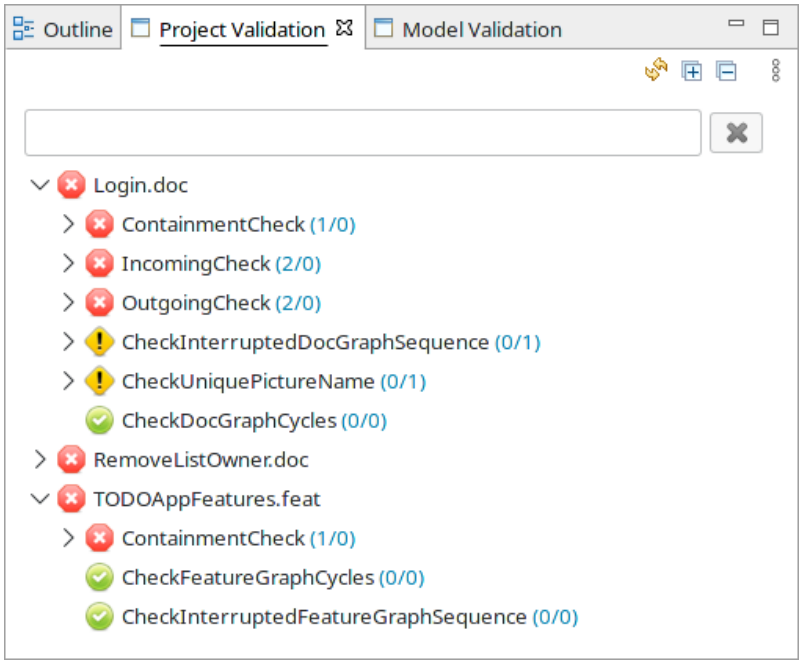


Figure 3.8: Project Validation view showing passed and failed checks

3.4.1 MCaM Check

The first one must be declared right above the `graphModel` declaration with the parameter "check". It activates the check of the constraints on the graph nodes in the graph model scope. At design time, the model creator receives feedback shown as errors. Those mistakes either cause the generator to throw a runtime exception, or the resulting application code will be error-prone, hence, not executable.

For this check to show a successful Incoming- or OutgoingCheck, the multiplicity of the relationship between node elements must respect the specified value. For example, a node element specified with an incoming Edge in this fashion `incomingEdges (Edge[1,*])`, must have at least one incoming connection of the `Edge` type. Any other kind of multiplicity can be defined; even not specifying any implies one. For instance, writing `incomingEdges (Edge)` is equivalent to `incomingEdges (Edge[*,*])`, which means at least and at most any number of `Edge` transition.

When specifying the containable elements of a container node (see i.e. line 11 in listing 2.1), with the `containableElements` property, each elements listed in that property

has a one type of multiplicity explained above. If not respected, the validation view will show a ContainmentCheck error until the designer resolves it.

3.4.2 MCaM Module Check

The second annotation allows us to specify our module checks. For instance, we need to ensure that two different screenshot nodes do not bear the same file name or that each sequence starts with the appropriate StartNode and ends with the EndNode. The first check makes sure that no screenshot overwrites another one that has the same name. The second one ensures that the resulting code reflects the model sequence from start to end. We also need to ensure that by integrating a DocGraphModel into another, we do not create cycles within the execution path. Otherwise, this would result in a never-ending execution loop once the generation process has started.

This module check allows us also i.e. to check for unique feature names in the FeatureGraphModel scope. Although the featureContainers ensure the independence of each feature, two featureContainers bearing the same name would result in two documentation pages having the same title. The *Project Validation* view shows all the checks done on the entire project and marks those that failed or passed the checks accordingly. Figure 3.8 shows, for example, the checks done on our example project, where the Login.doc DocGraphModel passed the cycle check but failed the one checking for interruptions in the graph path and unique picture names.

3.5 Generation Process

After all of the essential models have been designed, the developer must generate executable code. Recall that we want to generate two different projects: the Selenium-Java project to capture the screenshots and the VuePress project containing all the documentation Markdown files. Therefore, we thought of two ways to reach the intended result.

The first one consists of generating the Selenium-Java application from our model graphs and, subsequently, when executing generated application, creates the VuePress project with all the documentation files as shown in fig. 3.9. This approach would imply letting Java take care of the generation process of the Markdown files and the creation of the needed screenshot. An erroneous generation of the Selenium-Java application will leave the designer with no tangible result at all. In this approach, the generated application becomes a single point of failure.

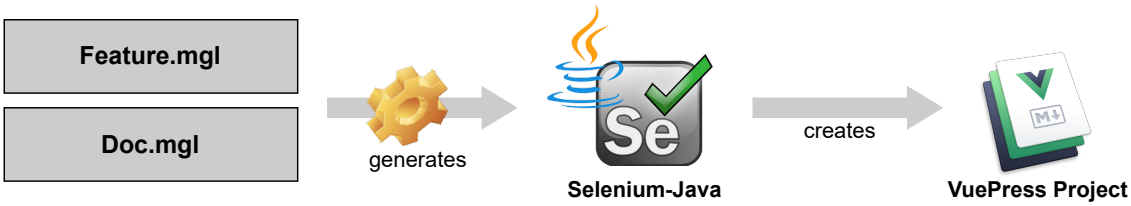


Figure 3.9: First approach: Sequential generation of the documentation

The second approach applies the concept of separation of concerns (see fig. 3.10). That means that we separate the task of taking screen captures with Selenium-Java from those generating the documentation files that will contain the screen captures. This way, if the Selenium-Java application contains errors that prevent taking screenshots, we can still generate a documentation website with placeholder pictures that can be later replaced with manually taken screen captures. It is almost apparent now that we opted for the second approach to create the documentation website.

So, clicking on the generate button creates an executable Selenium-Java application, which, once started, runs the user sequence model as a Selenium script in the Web browser. As mentioned in the preliminary chapter, the generator classes behind this process are Xtend classes. Xtend is a statically-typed programming language based on Java and translates to Java source code [17].

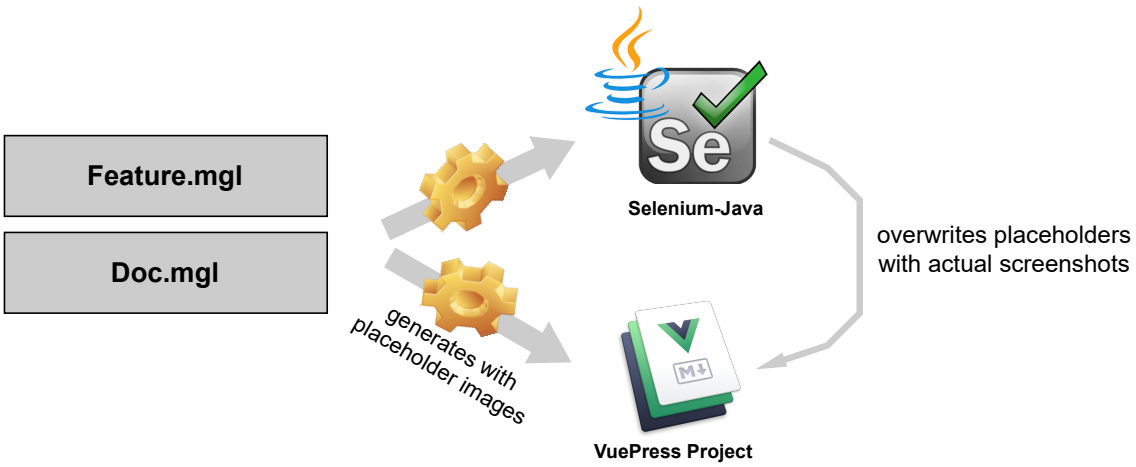


Figure 3.10: Second approach: Separate generation of the documentation

To start a generation process from a graph model, we must declare it "generatable". We do so by adding the `@generatable("path.to.generator.Class")` annotation in the graph model meta-specification. This annotation accepts as a parameter an Xtend or Java class that implements the IGenerator, whose `generate` method starts the whole process and the path to a location where the generation product will be saved (in our case, it is in the `src-gen` folder). We create two folder structures inside the src-gen folder: the first is the

Java project structure with the Selenium script, and the second is the VuePress project holding all the Markdown files with the documentation text. Going deeper into technical details is beyond the scope of this paper; hence we illustrate the whole generation process in figure 3.1. We begin by designing a graph model in the CINCO product application, the WebDoc editor; clicking on the generate button starts a simultaneous creation of both the project files mentioned earlier. At this point, the documentation pages are already available. We have chosen to generate the documentation file with references to placeholder pictures that will be overwritten once the Selenium script has been executed. After that, two simple commands are needed to launch the documentation website. Then again, we will not go deeper in details to stay within the scope of our thesis, but there is a great online guide on how to get started with VuePress [32]. As for the Selenium-Java project, it can be imported in an integrated development environment (IDE) such as Eclipse and launched there. After successful execution, actual pictures overwrite all the placeholder, and we have a generated documentation.

Chapter 4

Evaluation

This chapter presents the results of our thesis subject approach and describes the problems we encountered while developing the application. We will delve a little deeper into specific graph models from the TODO-App documentation and explain their desired outcomes in detail. We will also talk about our WebDoc application's limitations and overall effectiveness.

4.1 Method selection

In this thesis, we have implemented a graphical language to allow non-programmers to use an automated method for developing end user documentation for Web applications.

The first challenge we encountered with our approach was to determine the complete list of Web elements necessary for automating the documentation generation process. Even though websites have an inherent uniqueness, they still are constructed with the same fundamental elements: structural elements like paragraphs or tables and interactable elements like input forms and buttons. We specified, therefore, those elements in one of the metalanguages of our graphical DSL, the Doc.mgl. The approach led to the next challenge, giving the developer a way to give semantic meaning to those model elements. We added, therefore, a String property named description that enabled documenting each node of the graph model. The subsequent step of generating the Markdown files would produce a detailed documentation page for that model. Another challenge we faced was to produce a Selenium script in Java that, based on the same model sequence, would automate the Web browser and take screenshots of the indicated elements. Thus, we mapped each node element with a specific action implemented as a Java method. That

allowed us to construct a Selenium script in the same generation process by collecting the methods mapped to the model node elements.

The fact that we used a graphical DSL to solve the problem makes our editor user-friendly. The developer can easily make model representations of the Web application and then generate documentation from them. Moreover, if we recall the similar approaches presented in section 1.2, no programming knowledge is required to use the WebDoc. For instance, taking a deeper look at the login feature (cf. fig. 3.5), we created a model that starts by leading the WebDriver to the application landing page and logs in into the TODO-App using a Navigation node, Timer nodes that ensure that crucial Web elements appear before taking screenshots and an input form, which comprises two input fields for the username and password and a submit button. We implemented a feature to address and highlight all WebElements to make them visible on the screenshots. We also implemented the Comment node, which adds text as a caption to each screenshot in the Markdown file. The model has a Section node that acts as a syntactical delimiter and as a semantic section providing additional information. So, the effort to form such a model is small for a domain expert familiar with the Web application.

Additionally, the approach applied in this thesis efficiently produced a documentation website for the TODO-App, since the WebDoc allows one to create a documentation page without using additional resources. That is, from the concept to the finished documentation, the developer only needs to construct the diagram and, in another step, run the Selenium script.

4.2 Setup

The WebDoc editor application is a CINCO editor based on Eclipse. Setting up the development environment is straightforward – at least if a particular acquaintance with the Eclipse environment exists. Beforehand, a version of Java must be present on the system to run the application. Also, to automate a Web browser of choice, the corresponding Selenium WebDriver executable must be downloaded, and its path location must figure in the system's PATH variable.

We have already taken care of the Selenium libraries by generating the pom.xml file containing the required dependencies, and the resulting application structures, as mentioned in previous chapters, follow the convention. So, the application needs no additional configurations.

After downloading and extracting the application package, it must be started just like a standard Eclipse IDE would typically be. Next, a splash screen presenting the application and indicating the progress of the launch process appears, and within a few seconds, the WebDoc IDE starts.

4.3 Results

Our goal to document the TODO-App introduced in section 2.5 was addressed by specifying a graphical DSL from two different metalanguages using the Cinco Meta Tooling Suite. One allowed us to design user activities using graphical representations of UI elements. Another allowed us to gather those activities into application functionalities and served as the generation starting point. Tailoring our graphical DSL to this end ensured the adequacy of the WebDoc to model and create end user documentation for Web applications.

As a result, we began by identifying and then modeling seven features of the TODO-App: the first is the login capability, which allows users to access the dashboard. Then there are the abilities to add and remove tasks from task lists and create and remove tasks from them. We also have the functionalities for adding and deleting a list owner, which allows a logged-in user to add and remove another user as co-owner of a tasks list. All of those features were then regrouped in the FeatureGraphModel and kept separated within featureContainers (see fig. 3.3a). This separation allowed us to generate feature documentation independently. However, it also created the issue of the Selenium-Java program having to reset the Web application state after the automation process terminates. We could not address this issue by hard-coding the solution because each Web application has its navigational graph, hence different reset sequence paths. The documentation designer must deal with the application reset by modeling a sequence that does this.

Although we could document the TODO-App with all its features, the WebDoc still has limitations in some execution points:

The FeatureGraphModel acts as the entry point of the documentation model; herein, the user regroups all application features and starts the generation process. Nevertheless, it is possible to create two FeatureGraphModel files inside one project, breaking the idea of having a single generation entry point. We still have to remove this possibility by restricting the number of FeatureGraphModel files (.feat) the user can create to just one.

Furthermore, documenting a Web application that requires HTTP basic authentication, for example, is still not realizable with the model elements our graphical DSL

offers (HTTP basic authentication or htaccess is a content password protection measure that occurs before even loading the complete website, and ask for credentials in a modal dialog. [19]). On the Selenium FAQ (Frequently Ask Questions) [27], a proposed workaround is to provide the user credentials in the URL of the website like following `http://username:password@example.com`. Even though we still can add documentation text for the reader about how to handle those floating windows before or after their appearance in our model, we still cannot pass such windows and continue with the execution of the Selenium script. Using CSS or HTML selectors to address Web elements in a modal is still a work in progress since they do not belong to the website's document tree.

Moreover, choosing a sequential execution for our generator made it challenging to perform specific Selenium actions outside the execution workflow. That is, highlighting a Web element just for the screenshot without executing that element's specific action is still not working as intended. We implemented, therefore, an additional node that takes the highlighting and unhighlighting capability of a Web element outside of the element itself. That way, elements can be highlighted for the screen capture and unhighlighted right after, returning the appearance of those elements to their previous state for subsequent script execution.

Consequently, our editor application offered the capability to create a graphical model, which upon clicking the *generate button*, triggers the creation of both the Java application and the VuePress project structure. Subsequently, the Selenium WebDriver takes the indicated screenshots and saves them within the VuePress project folders.

Chapter 5

Final Remarks

5.1 Conclusion

We have proposed, designed, and implemented a solution for a DSL-driven generation of end user documentation for Web applications based on graph models using a graphical DSL, the Web Application Documentor (WebDoc). Visual elements resembling HTML are advantageous because the application is easy to use and building model graphs occurs almost intuitively. Also, we have added the implementation of module checks to assist the designer in constructing a correct model graph that will generate executable application code.

Making use of the Cinco Meta Graph Language (MGL) and Meta Style Language (MSL), we determined the look of each model element of our graphical DSL by keeping a close fidelity to the actual HTML elements they represent. That allowed us to create an editor application that enables the documentation designer to use those model elements to create graph diagrams illustrating the needed user workflow. Furthermore, by applying model checking methods to validate resulting diagrams, we ensure that generated executable application code be within the corresponding folder structure following established conventions.

We have been able to identify the characteristics of an end user documentation by following well-known standards and wisely applying the appropriate format to structure our documentation. As a result, we chose VuePress as the proper framework technology to transform it into a static website rendered in the Web browser.

The solution proposed in this thesis yields a well-structured and fully functional documentation website. Our approach uses the Selenium WebDriver, which navigates the

Web application in a short amount of time while taking screen captures at the indicated places. Writing such a Selenium script would require deep knowledge of the Web automation framework and good programming skills in Java. In our solution, the WebDoc takes care of it for the developer.

The documentation developer would also benefit because the WebDoc conveniently creates the VuePress project with all the configuration files ready to go. That helps save time since a tutorial on how to create such a project is not necessarily needed.

5.2 Future Work

Web applications appear in many different forms. They can appear as static page applications or complex constructs with many interfaces and an enormous navigational graph. The results for a simple Web application, such as the TODO-App, were satisfactory to show the feasibility of our approach. Potential growth of the capabilities of the WebDoc lies in testing the other extremum of the type of Web applications we could document with it. By leveraging the capabilities of modern Web browsers, i.e., to display responsive websites, we can improve the way of executing the Selenium script to obtain screenshots even for the mobile version of the same applications.

In the evaluation of the thesis, we have identified a couple of development opportunities for future improvements:

- **Start configuration**
  When creating a new documentation project using the project wizard, the user gets an empty project folder. Therefore, we could facilitate the creation process by generating a project template containing a start configuration. That is, with a single .feat file containing an example graph as the starting point of the documentation model. Such an example graph can jump-start the learning curve and facilitate the use of the WebDoc editor.

- **Cross-referencing model graphs**
  Reusability dictates that we must have the ability to integrate complete model graphs in others to avoid modeling sequences multiple times. In this thesis, we achieved it by using the PrimeReference feature offered by the Cinco framework. However, cross-referencing DocGraphModels inside others could be improved in the future to allow us, for example, to list all available graph models in a separate view. Thus, they would be visible to the WebDoc user and more intuitive to integrate into the currently edited model.

- **Implement more checks**
  We have implemented module checks that help build syntactically correct model graphs by enforcing the constraints defined on the meta-specification level of our graphical DSL. In the future, we could implement more checks to assist the developer in validating some other aspects of the model. For example, we could validate that the string values for the Web element selectors have the correct syntax of CSS selectors or XPath selectors by validating them against a well-constructed regular expression. We could even verify in advance if the HTML element we are looking for can be found with that given selector by running a selector query search in the background. The purpose of any check that we could implement is to prevent before generation, at design-time, interrupting errors. In that sense, we can add the verification of string pattern interfering with the Markdown syntax, for example. Thus, it is evident that there still is plenty of space for the WebDoc to grow.

- **Language extension**
  The example mentioned in the previous point raises the concern of allowing the developer to address Web elements by XPath or CSS. So far, we have implemented only the use of CSS selectors to address them. However, many development frameworks for Web applications use dynamic CSS id attributes, making it challenging to use as a selector for the WebDriver to find.

  One last point that requires our attention is the list of available Web elements to use while modeling. For those experienced in Web development, it is not surprising that new HTML elements can be implemented in the future. In addition, with fast-developing technologies and the growing number of devices that can launch browser applications, we might have to adapt and offer more model elements to reflect new ones.

List of Figures

Listings

Glossary

DOM

The Document Object Model (DOM) is a cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document. The DOM represents a document with a logical tree. https://en.wikipedia.org/wiki/Document_Object_Model.

Ecore

The core Eclipse Modeling Framework (EMF) includes a metamodel (Ecore) for describing models and runtime support for the models, including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.

Selenium

Selenium is a suite of tools for automating Web browsers. See https://www.selenium.dev/about/.

VuePress

VuePress is a Static Site Generator that generates pre-rendered static HTML for each page, and runs as an SPA once a page is loaded.

Abbreviations

API Application Programming Interface.

CPD Cinco Product Definition.

DIME DyWA Integrated Modeling Environment.

DOM Document Object Model.

DSL domain-specific language.

EMF Eclipse Modeling Framework.

HTML Hypertext Markup Language.

IDE integrated development environment.

IEC International Electrotechnical Commission.

IEEE Institute of Electrical and Electronics Engineers.

ISO International Standards Organization.

jABC Java Application Building Center.

LOC lines of code.

MDD model-driven development.

MDSD model-driven software development.

MGL Meta Graph Language.

MSL Meta Style Language.

OOP Object Oriented Programming.

UI User Interface.

UML Unified Modeling Language.

WebDoc Web Application Documentor.

Bibliography

[1] *IEEE Standard for Adoption of ISO/IEC 26514:2008 Systems and Software Engineering–Requirements for Designers and Developers of User Documentation*, January 2011.

[2] *ISO/IEC/IEEE International Standard - Systems and software engineering – Requirements for managers of user documentation*, March 2012.

[3] *ISO/IEC/IEEE International Standard - Systems and software engineering - Requirements for managers of information for users of systems, software, and services*. ISO/IEC/IEEE 26511:2018(E), pages 1–90, 2018.

[4] *ISO/IEC/IEEE International Standard - Systems and software engineering–Software life cycle processes–Part 2: Relation and mapping between ISO/IEC/IEEE 12207:2017 and ISO/IEC 12207:2008*. ISO/IEC/IEEE 12207-2:2020(E), pages 1–278, 2020.

[5] *Definition of meta- prefix*. https://www.merriam-webster.com/dictionary/meta, November 2021. Accessed: November 02, 2021.

[6] Amalfitano, Domenico, Anna Rita Fasolino and Porfirio Tramontana: *Using dynamic analysis for generating end user documentation for Web 2.0 applications*. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 11–20. IEEE, September 2011.

[7] aside ufba: *GuidAutomator GitHub Repository*. https://github.com/aside-ufba/guide-automator. Accessed: November 04, 2021.

[8] Bettini, Lorenzo: *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[9] Bosselmann, Steve, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff and Bernhard Steffen: *DIME: A Programming-Less Modeling Environment for Web Applications*. In Margaria, Tiziana and Bernhard Steffen (editors): *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 809–832, Cham, 2016. Springer International Publishing.

[10] Brambilla, Marco, Jordi Cabot and Manuel Wimmer: *Model-driven software engineering in practice*. Synthesis lectures on software engineering, 3(1):1–207, 2017.

[11] Busch, Daniel and Annika Fuhge: *Cinco Product Specification*. https://gitlab.com/scce/cinco/-/wikis/Cinco-Product-Specification. Accessed: October 02, 2021.

[12] Cinco Dev. Team: *About Cinco SCCE Meta Tooling Framework*. https://cinco.scce.info/about/,. Accessed: July 14, 2021.

[13] Cone, Matt: *Markdown Guide | Getting Started*. https://www.markdownguide.org/getting-started/, October 2021. Accessed: November 02, 2021.

[14] Descher, Marco, Thomas Feilhauer and Lucia Amann: *Automated user documentation generation based on the Eclipse application model*. 04 2015.

[15] Despa, Mihai Liviu: *Comparative study on software development methodologies*. Database Systems Journal, 5(3):37–56, 2014.

[16] Eclipse Foundation: *Rich Client Platform*. https://wiki.eclipse.org/Rich_Client_Platform. Accessed: July 12, 2021.

[17] Eclipse.org: *Java with Spice*. https://www.eclipse.org/xtend/documentation/. Accessed: September 24, 2021.

[18] Fowler, Matin: *ModelDrivenSoftwareDevelopment*. https://martinfowler.com/bliki/ModelDrivenSoftwareDevelopment.html, 2008. Accessed: August 02, 2021.

[19] HTACCESS-Guide.com: *Apache .htaccess Guide & Tutorial*. http://www.htaccess-guide.com/.

[20] Kipyegen, Noela Jemutai and William PK Korir: *Importance of software documentation*. International Journal of Computer Science Issues (IJCSI), 10(5):223, 2013.

[21] MDN contributors: *HTML: HyperText Markup Language*. https://developer.mozilla.org/en-US/docs/Web/HTML. Accessed: November 02, 2021.

[22] Naujokat, Stefan: *Heavy meta: model-driven domain-specific generation of generative domain-specific modeling tools*. PhD thesis, Technical University of Dortmund, 2017.

[23] Naujokat, Stefan, Michael Lybecait, Dawid Kopetzki and Steffen Bernhard: *CINCO, a simplicity-drivern approach to full generation of domain-specific graphical modeling tools*. International Journal on Software Tools for Technology Transfer, 20(3), 2018.

[24] Oliveira, Allan dos Santos: *GuideAutomator: Automated user manual generation with Markdown*. 2016. Accessed: November 04, 2021.

[25] Pérez Andrés, Francisco, Juan de Lara and Esther Guerra: *Domain Specific Languages with Graphical and Textual Views*. In Schürr, Andy, Manfred Nagl and Albert Zündorf (editors): *Applications of Graph Transformations with Industrial Relevance*, pages 82–97, Berlin, Heidelberg, October 2008. Springer Berlin Heidelberg.

[26] SCCE Group: *Sustainable Computing for Continuous Engineering*. https://www.scce.info/. Accessed: September 29, 2021.

[27] StackOverflow: *Getting Selenium to login via .htaccess popup*. https://stackoverflow.com/questions/2085348/getting-selenium-to-login-via-htaccess-popup. Answered by Bozho. Accessed: November 09, 2021.

[28] Stahl, Thomas, Markus Voelter and Krzysztof Czarnecki: *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc., Hoboken, NJ, USA, July 2006.

[29] Steffen, Bernhard, Tiziana Margaria, Ralf Nagel, Sven Jörges and Christian Kubczak: *Model-Driven Development with the jABC*. In Bin, Eyal, Avi Ziv and Shmuel Ur (editors): *Hardware and Software, Verification and Testing*, pages 92–108, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[30] TU Dortmund, Chair of Programming Systems: *Java Application Building Center*. http://ls5-www.cs.tu-dortmund.de/projects/jabc/index.php. Accessed: October 04, 2021.

[31] Waits, Todd and Joseph Yankel: *Continuous system and user documentation integration*. In *2014 IEEE International Professional Communication Conference (IPCC)*, pages 1–5, 2014.

[32] You, Evan: *VuePress | Vue-powered Static Site Generator*. https://vuepress.vuejs.org/, October 2021. Accessed: November 02, 2021.

I hereby certify that I have written this paper independently and have not used any sources or aids other than those indicated, and that I have clearly marked any citations.

Dortmund, November 15, 2021

Mukendi Mputu