technische universität
dortmund

Bachelor Thesis

DSL-Driven Generation of User Documentations for Web Applications

Mukendi Mputu
September 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 [4]. The challenge lies in the fact that the development team tasked to document the software has to 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 has to reproduce the steps or whole scenarios a potential end user would go trough and document where useful information are to be provided in order 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, for it is a cycle that has to 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, but considering complex applications developed by multiple teams, this task can raise the cost in time and resources as the information about the whole project has to be collected 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 on assemble those to generate a complete documentation.

Software projects are bound by budget and deadline requirements. This leads to the fact that automation processes become essential where the reduction of the development time can be increased [12]. 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 [21]. This thesis introduces the use of a graphical domain-specific language (DSL) – a programmatic languages adapted to a specific domain problem [16] – 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 application by means of a graphical DSL, a similar approach to [6]. We developed an applications for modeling different user action sequences in a web application using the Cinco SCCE Meta Tooling Framework [8] and subsequently generate an 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, but 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 static site.

1.2 Related Work

1.3 Outline

Chapter presents the Cinco SCCE Meta Tooling Framework, in particular the two meta-languages (Meta Graph Language (MGL) and Meta Style Language (MSL)) that constitute the specification (Section 2.4.1 and 2.4.2) and also the Cinco Product Definition (CPD) (Section 2.4.3). The chapter then rounds off with an outline of the Generator classes written in Xtend, a Java dialect that offers much flexibility and expressiveness in its syntax (Section

[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

2.4.4).

The specifics of the generated modeling platform are described in chapter 3. Whereby Section 3.1 explains the basic concepts of a graphical domain-specific language, then Sections 3.3, 3.4 and 3.5 relate about different modeling elements for configuring and designing the end user documentation. In chapter 4 a in-depth explanation of the methodology is given. Beginning with the evaluation of the selected work method in Section 4.1 followed by the system setups and the results (Section 4.2 and 4.3) and finally, in Section 4.4 concluding with a discussion on the approach taken in this thesis.

Further improvements of the program are presented in chapter 5. Lastly, chapter 6 winds up with a discussion on the thesis.

Chapter 2

Preliminary

This chapter introduces the various notions necessary for achieving our goal. We first look at what need to appear in the structure of our end user documentation as recommended by known standards and what technologies are actually available to help us complete this task. Then we will allude the fundamentals of model-driven development with the Cinco Meta Tooling Suite.

2.1 End User Documentation

The value of a good software product, in fact, of any product destined to be brought to the consumer is determined on how effective the end user is able to learn to use the product. Hence, putting a great effort to generate and manage a useful, well-structured documentation is as much important as the development of the product itself [2].

End user documentation aim to provide the application end users with the information necessary to properly interact with the software. It is part of the development life cycle and the bridge between the product developer and the user [3]. By improving the structure and usefulness of the information delivered to end user, the developer not only reduces significantly the return of calls for support, but enhances also reputation of the product as well as of the producing company [1].

In this section we will go trough the main characteristics of end user documentation, focusing on the standards establish by the ISO/IEC and the IEEE [1, 2]. Next we will talk about the specifics of documenting web application by giving an example of a documented web application. We will also present the currently used technologies for creating and managing such documentation. Our main goal is to produce an entire end user documentation page from a graphical model. Before going further, we ought to precise that the type of documentation to be created will depend on how the documentation designer lays out the model, that means the model could be designed to represent a technical documentation – requiring deep knowledge of the application documented – or it could also be structured to produce a documentation for simple end users with no technical knowledge at all of the underlying web application [4].

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 (like buttons, navigation links, checkboxes, etc.) 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 mandates i.e. that completeness and accuracy should be of the utmost importance when designing the end user documentation.

Bearing that in mind, a good documentation should respond to three important questions: *why* the application has been programmed, meaning the problem the application intends to solve, *what* is the end user supposed to do to quickly and efficiently get to the solution and *how* is it achieved [2]. This is a task we leave to the information manager. The last question, however, is better answered by providing visual help in form of screen captures or any illustration that fastens the understanding of the documentation.

As we mentioned before, we intend to document web applications from the viewpoint of the application end user. This means that documentation developer should keep in mind the readership of end product [2], so identifying the key tasks and activities ot the common users should be integral part of the planning process. That being said, our solution puts the developer at charge for how detailed the documentation becomes.

2.1.2 Documentation Software Tools

Choosing the format in which the documentation is brought to the end user is as much important as the rest of the milestones of the planning process. A inadequate choice in this matter can reduce the potential of the documentation to reach a maximum number of users, rendering it ineffective. A good decision is to opt for technologies that have already gained acceptance among most users. This ensures greater reach and preexisting knowledge amongst that same audience:

Markdown

Markdown is a lightweight markup language created by John Gruber[1] in 2004. Since, it has established itself as one of the world's most popular markup language [9]. Gruber states on his webpage that Markdown is comprised of formatting syntaxes that can be applied to plain-text file and optionally a converter to other markup languages files like HTML.

For its popularity, it is a great choice for our project, since description of the UI is first made as plain-text and then transform 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. There is tremendous amount of documentation about Markdown syntax available online for the interested reader to get started.

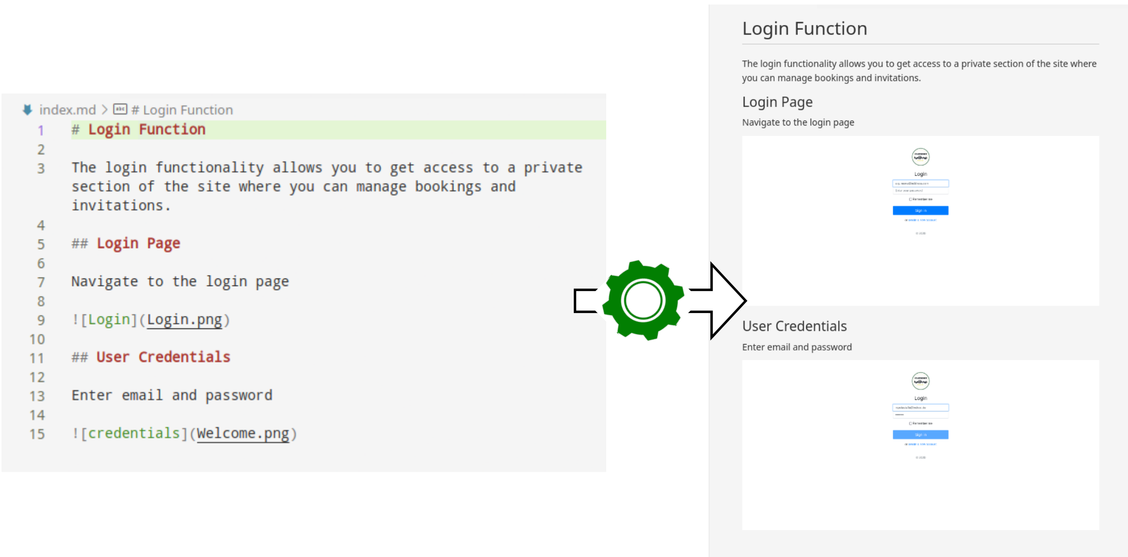

Figure 2.1: Plain-text transformation with Markdown

Although there exists a plethora of applications that transform Markdown files into HTML files to be rendered in a web browser application, we choose to work with one framework that also established itself throughout the developer community and that especially works very well with Markdown files: VuePress.

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

VuePress

VuePress is a minimalistic static website generator powered by VueJS [22], which is a open source JavaScript framework created by Evan You[2]. As an open source software, it is maintained by a huge community of contributors, what makes it robust and up-to-date.

We will use VuePress to statically launch the end user documentation as a website, relieving the developer of the task 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) [13] – 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. To create and use models to represent the software system, a DSL that is close to the problem domain, is needed. Application domain experts possess the knowledge of the application structure, which they represent in form of models. Domain experts and application programmers have to agree on how the specification language determines syntax and semantic of the model elements.

The core principles of MDD reside in the fact that software development is accelerated by providing a simple, but efficient abstraction of the software structure as a model. Those model abstraction, representation of real-world objects, are transposed through a series of model-to-model or model-to-code transformations [18]. Our work is to utilize the DSL provide by the Cinco framework to design our graphical DSL, which in turn will permit the generation of a functioning Selenium-Java application (Selenium is a suite of application tools for automating web browsers) to take screenshots of the different web application states.

Opposed to the common 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 a great software documentation) to accomplish the task of documenting 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.

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

When applied correctly, the result of the model-driven development process is a tailored application to domain. This reflects one of the main advantages of MDD: the accuracy of targeting directly the specific problem [7]. Besides, it is still possible to change the DSL so that it adapts to the new challenges emerging during the development process. This can be iterated until the specification reaches preciseness wanted to solve the problem.

2.3 Domain Specific Language

A domain-specific language (DSL), as mentioned before, is a language adapted to specific development domain. In [15] 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. If we stick to this analogy, just as one would start with a blueprint to construction a mechanical tool, designing as DSL required similar steps. One have to conceptually lay out 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 to our language. *Meta* literally means *situated behind or beyond*[3]. So the metalanguage is the descriptive language coming before our language, describing the meaning (semantic) and the relationship between the objects of our target language. In other words, the meta-DSL is the abstract syntax and the resulting DSL concrete syntax. It is possible to ascend the modeling hierarchy of meta-definition until we reach a self-referencing language, like it is the case 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 possibly 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 [5], a language Workbench for writing textual DSLs [14]. The next section explains those concepts in detail.

[3]According to the definition given at https://www.merriam-webster.com/

2.4 CINCO SCCE Meta Tooling Framework

The CINCO Framework is a generator-driven development environment for domain-specific graphical modeling tools [8]. It is actually developed by the chair of programming systems at the Technical University of Dortmund and one of the many 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 [17]. 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 tools called CINCO product. The MSL is where the CINCO developer defines the look every node and edge element, as well as the font and color of the text to be displayed in the graphical model [14]. The created metamodel is based on Ecore, the metamodeling language of the Eclipse Modeling Framework (EMF) and the Graphiti framework is used to generate the corresponding graphical model editor. Additionally, you find the right 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.

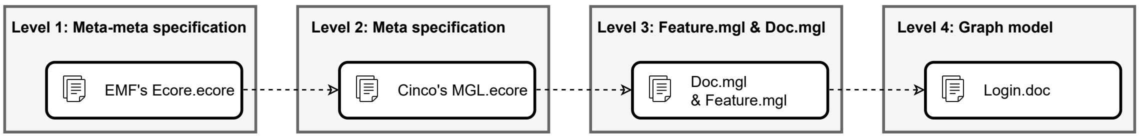

Figure 2.2: Hierarchy of our graphical DSL specification

The full generation process of a graphical modeling tool (of a CINCO Product Application) comprises four essential (meta) levels [15]. The first level, associated with the role Eclipse Developer, 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 used the metamodel from the first level to develope the MGL.ecore, which in turn becomes metamodel of the third specification level, where the CINCO Product Developer operates and also where this thesis comes in, see fig. 2.2. This means the Eclipse Developers are now at the meta-metalevel, 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 level. Latter is the level where the CINCO Product Users use the generated graphical editor to create the domain specific models.

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

2.4.1 Meta Graph Language

The MGL sketches the behavior, the constraints and gathers the all the graphical components that will constitute the model elements that come in use in every graph diagram created in the modeling tool. As a Cinco Product developer, this is most likely the first place to start: the main elements that can be define in a MGL are **nodes**, **containers** and **edges** as illustrates listing 2.1.

```
1  id info.scce.cinco.product.features.main
2  stylePath "model/Feature.style"
3
4  @generatable("info.scce.cinco.userdocumentation.Generate", "/src-gen/")
5  graphModel FeatureGraphModel {
6      iconPath "icons/16/feature_16.png"
7      diagramExtension "feat"
8      containableElements(FeatureContainer,DocNode[1,*])
9      attr EString as modelName := "New Feature"
10     attr EString as description := "New Description"
11 }
12
13 container FeatureContainer {
14     style featureContainer("${title}")
15     containableElements(Start[1,1], Stop[1,1],DocNode[1,*])
16     attr EString as title := "New Feature"
17     @multiLine
18     attr EString as documentation := "This feature is about ..."
19 }
20
21 node DocNode{
22     style docNode("${mgl.modelName}")
23     prime docMgl::DocGraphModel as mgl
24     attr EBoolean as createScreenshots := true
25     incomingEdges (Edge[1,1])
26     outgoingEdges (Edge[1,1])
27 }
```

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

From line 5 to 11, we define some important attributes of our graphical DSL. Models instantiated from it will have the icon and the file extension we specified here, as well as the model name and a list of element that can be contained with the modeling canvas.

Lines 13 and 21 define graph model nodes by associating an existing style (i.e. `featureContainer`) from the file specified by the **stylePath** keyword and restricting the type and number of edges that can be connect from and to them. In our case, the feature container node does not have any edges, they are meant to be independent from one another; opposed to that the DocNode node can only have one incoming and outgoing transition edge. This forces more or less the use of single "pathed" sequences without ramification.

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 are 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 2.2). 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.

```
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 2.2: 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 element of the same type, hence avoiding repetitive definition of the same attributes within multiple different elements and allowing some elements to extends the properties of the parent elements. For example, we see in line 3 the `featureContainer` appearance extends the default one and at the same time redefines the background color.

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

2.4.3 Cinco Product Definition

The Cinco Product Definition (CPD) offers an entry point when it comes to generating the application code. Herein, the Cinco Product Developer has to provide key information like the Cinco product name, at least one or more MGL files to be included into the generation process. Optionally, one can setup a splash screen with branding images, add a descriptive text about the application and specify plugins and/or features [11]. The listing below gives an insight into the CPD specification language.

```
CincoProduct UserDocumentationTool {
    mgl "model/Feature.mgl"
    mgl "model/Doc.mgl"

    splashScreen "branding/splash.bmp" {
        progressBar (37,268,190,10)
        progressMessage (37,280,190,18)
    }

    image16 "branding/Icon16_dark.png"
    image32 "branding/Icon32.png"
    image48 "branding/Icon48.png"
    image64 "branding/Icon64.png"
    image128 "branding/Icon128.png"
    linuxIcon "branding/Icon512.xpm"

    about {
        text "UserDoc is a DSL-driven generator of end user documentation
for web application. It is a bachelor thesis project developed with the
Cinco SCCE Meta Tooling Suite ( http://cinco.scce.info )."
    }

    plugins {
        info.scce.cinco.product.userdocumentation.edit,
        info.scce.cinco.product.userdocumentation.editor
    }
}
```

Listing 2.3: UserDocumentationTool.cpd

2.4.4 Xtend Generators

For our documentation application we need to create two folder structures: one will be the Selenium-Java application with the specific maven project structure. We apply the rule of convention over configuration and generate the Java project as recommended on the Maven Apache [1] website and add Selenium as a dependency. The other project structure we generate by following convention is for the VuePress project, which in fact is the end result we aim to obtain.

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

This generation approach utilizes Java's and Xtend's text templating feature which is based on the generation pattern used in the Java Application Building Center (jABC) [19, 20]. In fact, many generator classes in our example project implement and extends interfaces from the generator runtime and template package of the jABC CINCO metaplugin.

Additionally, other template classes are implemented to create configuration files, application class files and also Markdown files with the content modeled by the documentation designer in the graph model. In chapter 1 section 1.1 we provided a link to the Github repository, where the complete application code can be found. In the next chapters, we deliver an extended explanation of the generator and feature classes, while introducing our application as an on going example.

Chapter 3

WebDoc - Web Application Documentor

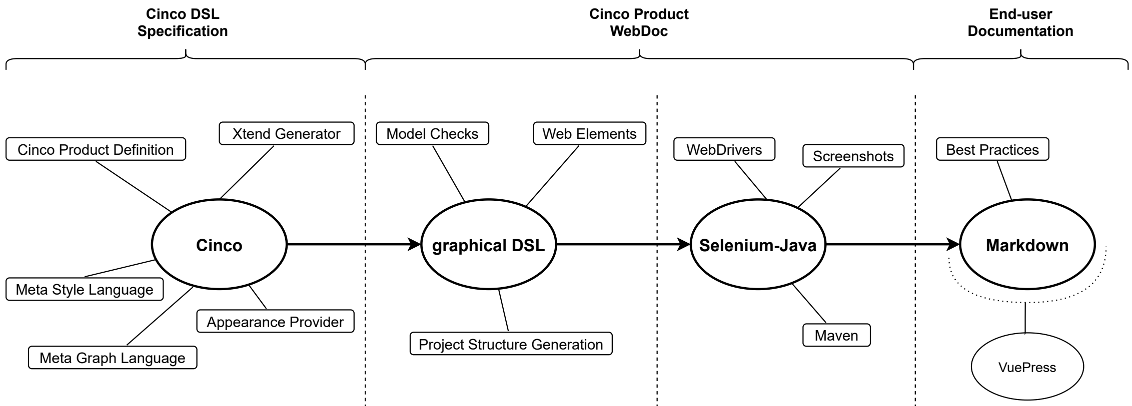

Figure 3.1: End user Documentation creation process workflow

Having described the metalevel of the graphical modeling tool, we come now to the description of the model editor instantiated from it. The goal of this chapter is to explain the the appearance and behavior of the various model elements. First, we give a succinct definition of a graphical DSL, then illustrate the fundamental building blocks of our documentation model and by presenting at the same time the CINCO product application. Later on, we demonstrate the use of these graphical elements to specify the important configuration of the website we want to document. At the end, we show how using the editor built-in generator, a project structure, that constitute the target application, is generated.

3.1 Graphical DSL

Under domain-specific languages (DSLs) we understand a languages tailored to describe or solve problems in a specific computational domain. They represent the core concept of most state-of-the-art software development paradigms [16]. As stated in [15], one of their great advantage is that they permit domain experts with no programming experience to design application by means of graphical components, whose behavior and semantic have been or will be programed by developers with coding experience. The graphical aspect – meaning that there is no use of textual grammar to construct a model – adds an abstraction layer that eliminates a bit further the necessity of mastering the syntax of the underlying DSL, as well as the APIs intertwining the metamodel elements.

In our case, the graphical model is composed of node elements, which have been applied different appearances to, in order to resemble to some extent the corresponding web elements they represent. The purpose behind this approximated replication is not to recreate all possible web elements, but to give the developer a sense of control over the interactable UI elements. With those elements at hand, the designer can simply model a user action by arranging node elements following the sequence it would require in the web application. This graphical language is in such way domain specific, that it is tailored for modeling sequences in a web application driven inside a web browser; modeling a documentation for a desktop application for example is not possible.

3.2 Graph Editor

The editor is mainly composed of the canvas in the middle of the working environment (1), where the developer can drag and drop model elements from the palette located on the right-hand side (2). Herein, elements are grouped in a single category; this is achieved using the `@palette("category name")` annotation in the .mgl meta-specification. 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. By hovering over a model element, an arrow symbol appears, allowing the developer to similarly drag and drop connecting edges from the source to the target element.

All diagrams that are open in the editor are checked in the background for compliance with the requirements described on the metalevel using the `@mcam("check")` and shown in the Model Checking view right beneath the project explorer (4). There's also the CINCO

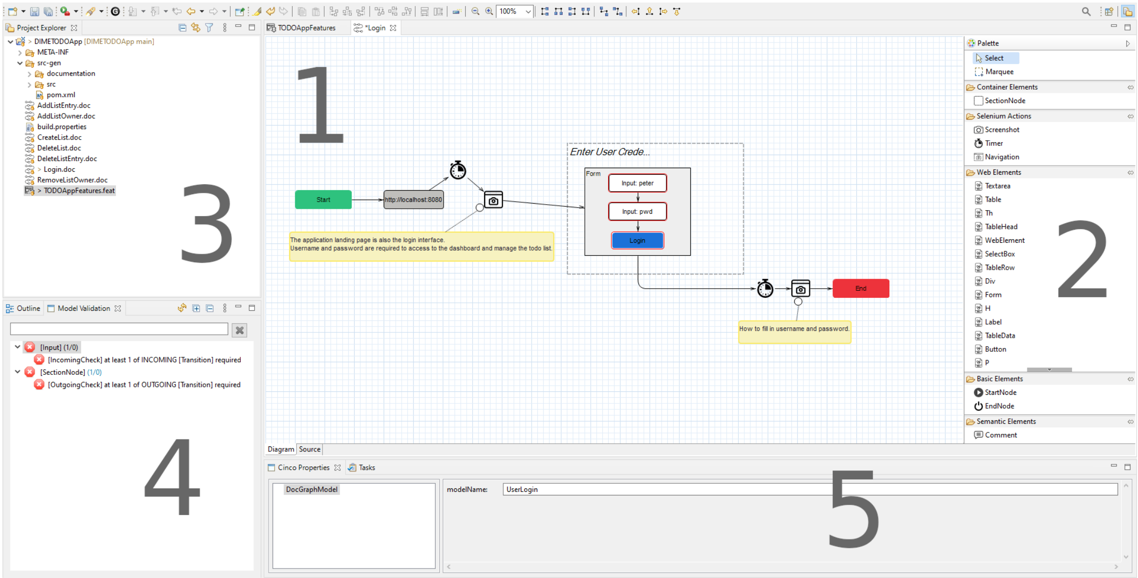

Figure 3.2: CINCO Product Application - Graph model editor

property view (5) that displays the attributes and values of any selected element in the editor. This is where the developer can modify those values if the attribute field allows it.

The example model shown in Figure 3.3 illustrate the sequence an end user would eventually undergo to login to the web application. Showing the web elements that will be interacted with and how combined into a logical sequence, they form a user workflow. The sequence begins with the start node then comes a navigation node with the link attribute pointing to `http://localhost:8080`, the landing page of our TODO Web App in development.Next, the timer node waits explicitly for an amount of second the designer specified in the property view. The palette view presents a list of all the available graph model elements (see table A.1). Since we specified two different MGLs, the list in the DocGraphModel differs from the one in the FeatureGraphModel. For instance, the DocGraphModel has a whole category for web elements specified, because the user action sequence is modeled using common UI elements of the web interface. On the other hand, the FeatureGraphModel is specially there to provide some start configuration (e.g. the path to the webdriver executable to be used), meaning that it regroups all the features modeled in every DocGraphModel, builds up both project structures mentioned before and triggers the code generation for every single one of them.

3.3 Feature Model Elements

The feature graph model is the application starting point. Here, the developer groups all the feature that needs to be documented in feature containers. Figure 3.4 depicts a

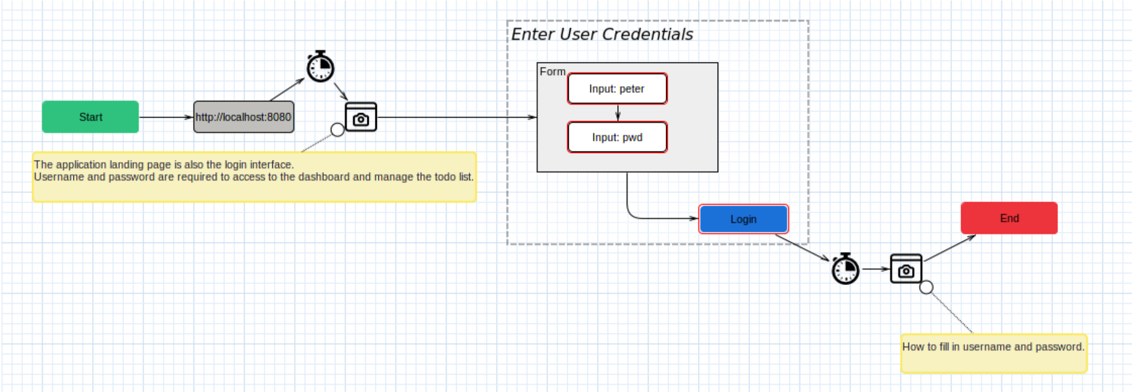

Figure 3.3: Example of a user workflow: here the login sequence

portion of the features modeled for our TODO application. The important abilities of the web application are regrouped here: the ability to login, to create a new list, add a task to that list or remove it and lastly delete the whole list. Nonetheless, our TODO application offers the possibility to also add a new list owner, which already exist in the system as regular user. In order to have a reasonable size for the image, the last features are not shown on the picture. On the top left-hand corner you can see an property container holding the webdriver property, whose value is set to `FIREFOX`. This value will be assigned to the Selenium webdriver variable in the Java class. Remember that the executable file for the chosen webdriver muss already exist somewhere in file system and the path to it has to been specified here in the property view.

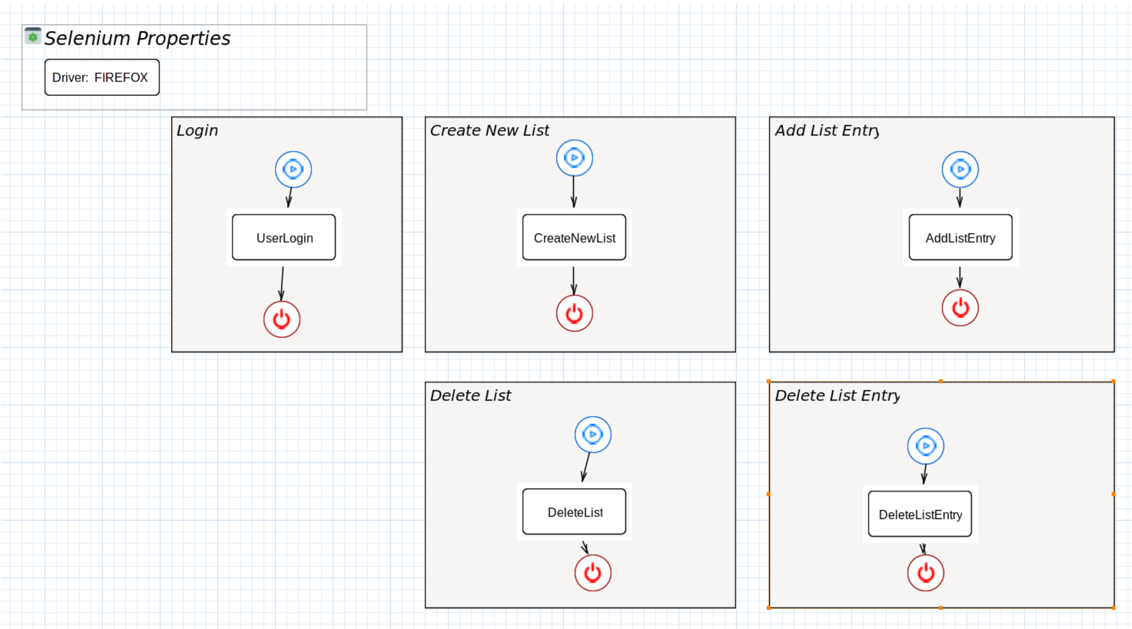

Figure 3.4: CINCO product - Feature Graph Model with Feature Containers

Even though the features are presented in logical workflow order, they still are independent from one another and will as well be generated independently. However, as mentioned in the previous chapter, this separation does not exclude reusability, since it is possible to integrate a whole DocGraphModel inside another one. considering, for instance, the CreateNewList feature, it requires the user to be logged in to be able to create a new tasks list. So the documentation developer does not have to repeat the login sequence inside the new, but can simply drag and drop the UserLogin.doc file inside the diagram of the new model graph and connect it within the sequence as if it were a regular graph node (see figure 3.5). Double-clicking on the imported subgraph leads directly to the original graph model. This double-click action has been implemented using the annotation `@doubleClickAction("info.scce.cinco.product.userdocumentation.action.DocNodeOpenSubmodel")`, where the DocNodeOpenSubmodel class holds the implementation logic.

The feature graph model contains also semantic elements, which are integrated in the existing featureContainer for simplicity's sake. Clicking on such a featureContainer opens up simultaneously the Cinco property view, where a multiline input field name description is found. This is where the documentation designer can enter some explanatory text to give meaning to the diagram elements and at the same time provide text content for the Markdown documentation files. Those information will be collected during the code generation process.

3.4 User Action Model Elements

In the user action model, just as the naming suggest, is where the action takes place. Here, the diagram editor has more elements to design the conceived user workflow: the graph elements categorized under "Web Elements", the Selenium actions and the comment node under the category "Semantic Elements".

One of the most import purposes of the web elements is to allow HTML element to be addressed. This implies that appropriate actions can be applied to such representation of an HTML element. considering for example the input field in the picture below, the action of inputting a text is made possible by provide a property variable `content`, whose value is then display inside the node element (here i.e. "Shopping"). The same holds true for button elements, which can be applied the click action or for selectboxes, which can be dropped down to reveal the options they contain, etc. In addition to that, all web element can be highlighted either by setting the `highlighted` property to true or by letting the Highlight node amongst the Selenium actions take care of it. unsure about this idea

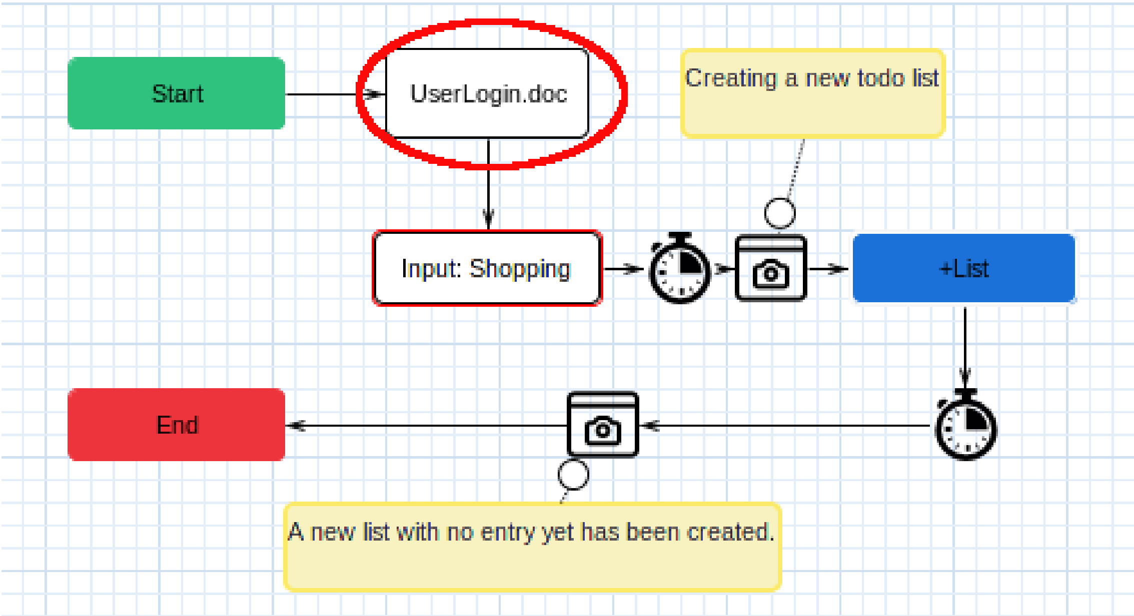

Figure 3.5: Cinco product - Reusing Login graph model in the CreateNewList graph model

A particular element present in this diagram is the screenshot node, whose only task is to take a screen capture of the current state of the web application. Put in conjunction with the highlight functionality, screenshots with specific web elements highlighted (framed with a red border) can be created. That way, the documentation will contain visual help with the right emphasis on described HTML element. Furthermore, the comment node allows the documentation creator to add descriptive text about the picture, that will be later added to the Markdown file as image caption. By specifying as different kind of edge for the comment node, we restricted its connection target to the screenshot node only. The idea behind is that all other diagram elements be commented on using the description property in the property view to avoid having diagrams filled with comment nodes.

One last particular element that need explanation is the timer node – style by the chronometer. It belongs to the Selenium actions category and is intended to slow the webdriver down for a few seconds, while checking for a certain condition to become true before continuing. Said conditions are from the Selenium class named ExpectedConditions, from which we implemented a selected few applicable to our case, i.e. `presenceOfElementLocated`, `presenceOfAllElementsLocatedBy`, `elementToBeClickable`, etc. If we take for instance the condition `presenceOfElementLocated`, the timer holds the webdriver execution for 3 seconds, checking every 500 milliseconds if the target web element appears within the DOM

3.5 Generation Process

As mentioned before, the prominent feature of CINCO is the generate button. This allows the developer, after the modeled has been laid out without errors, to generate the a fully realized Selenium-Java application, which is going to execute the user sequence model just created. Figure 3.6 shows where the generate button is located.

Figure 3.6: CINCO product - generator button

Chapter 4

Evaluation

The purpose of this chapter is to explain our approach and its results. First, we justify the decision for the chosen approach to the topic, as well as the challenges posed and the solutions found for them. A brief description of the design pattern used and the resulting benefits is also given. Subsequently, a minimal system is used to demonstrate the development of end user documentation. In addition, we present the resulting application, how it can be created and launched. Finally, the overall results and any alternative approaches will be discussed.

4.1 Method selection

Selenium is at the core of our methodology, as it is per se the default automated testing and manipulation framework for web application. Not only the API documentation is thorough and exhaustive, due to the fact that it is open-source, but it also allows the manipulation of the majority of the browser engines through WebDrivers [10]. In addition, it integrates very well in a java as a maven dependency, hence well-fitted for purposes. For now, it has to be note that we do not intend to use Selenium for testing our application, rather for automatically executing the tasks or steps the documentation developer would have to do in order to create screen captures. In that sense, the Selenium WebDriver simulates the end user steps and saves them as pictures in a dedicated folder.

Another state-of-the-art framework we chose is VuePress; it is well suited for creating static websites and is therefore perfect for designing technical documentation pages. Moreover, the effort required to configure a VuePress project is so low that the website can be up and running quickly. Alternatively, you can effortlessly bind the generated website to an existing domain, for example a company wiki page or similar. As for

Selenium, VuePress also has a great online documentation page, with a step-by-step guide on how to quickly setup a project and launch the server.

The link between the Selenium WebDriver and the VuePress project is the generated java maven project. Here, we decided to relieve the documentation developer of the task of adjusting the project properties, adding the dependencies, naming the different packages and so on. This are the tasks an experienced java programmer would and as we intend to empower non-programmer to be comfortable using our application, it is best we determine for ourselves implementation of said application.

Consequently, our editor application offered the capability to create a graphical model, which upon clicking the *generate button* triggered the creation of both the java application and the VuePress project structure, and subsequently, the Selenium WebDriver took the indicated screenshots and saved them within the VuePress project folders.

4.2 Setup

Since our editor application is a CINCO editor, which is based on Eclipse, setting up the development environment is straight forward – at least if a certain acquaintance with the Eclipse environment already exists. Beforehand, a java version must be present of the system in order to run the application. Also, to automate a web browser of choice, the corresponding Selenium WebDriver executable has to be downloaded and its location path has to be added to the system's PATH variable. Concerning the Selenium libraries, we already take care of it by generation the pom.xml file, containing the required dependencies, along with the whole application structure.

After downloading and extracting the application package, it has to be started just like a common Eclipse IDE would normally be. A splash screen presenting the application and indicating the progress of the launch process appears. And within a few second a CINCO product IDE is started.

4.3 Results

4.4 Discussion

tion issue with Sele-
ver,
ning wheel protocol)
eenshots at the right

ment by XPath or
dynamic ids compli-

quire a certain knowl-
e syntax of css selec-

highlight on each
taking the screenshot
for screenshots

Chapter 5

Future Work

- Better cross-referencing sequences in other models

Chapter 6

Conclusion

Appendix A

Further Information

Table A.1: List of available model elements

Model Elements		
Palette Category	**Nodes**	**Description**
Basic Elements	StartNode	cell3
	EndNode	cell3
Web Elements	WebElement	cell3
	Input	cell3
	Button	cell3
	TableHead	cell3
	TableRow	cell3
	Form	cell3
	P	cell3
	Textarea	cell3
	Span	cell3
	H	cell3
	SelectBox	cell3
	TableData	cell3
	Label	cell3
	Th	cell3
	Div	cell3
	Table	cell3
	TableBody	cell3
Selenium Actions	Screenshot	cell3
	Navigation	cell3
	Timer	cell3
	Highlight	cell3
	UnHighlight	cell3
Semantic Elements	Comment	cell3
Container Elements	SectionNode	cell3

```
package info.scce.cinco.product.userdocumentation.codegen

import info.scce.cinco.product.features.main.feature.FeatureGraphModel
import de.jabc.cinco.meta.plugin.generator.runtime.IGenerator
import org.eclipse.core.runtime.IProgressMonitor
import org.eclipse.core.runtime.IPath

/**
 * @author Mukendi Mputu
 */
class Generate implements IGenerator<FeatureGraphModel> {

        /*  */
        override generate(FeatureGraphModel model, IPath srcGenPath,
    IProgressMonitor arg2) {
                new UserDocProjectGenerator(model).createProject
        }

}
```

Listing A.1: Generate.xtend clas implementing the IGenerator infterface

List of Figures

Listings

Glossary

Selenium

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

Abbreviations

API Application Programming Interface.

CPD Cinco Product Definition.

DOM Document Object Model.

DSL domain-specific language.

EMF Eclipse Modeling Framework.

HTML HyperText Markup Language.

MDD model-driven development.

MDSD model-driven software development.

MGL Meta Graph Language.

MSL Meta Style Language.

OOP Object Oriented Programming.

UML Unified Modeling Language.

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 information for users of systems, software, and services*. ISO/IEC/IEEE 26511:2018(E), pages 1–90, 2018.

[3] *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.

[4] 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.

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

[6] 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.

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

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

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

[10] Conservancy, Software Freedom: *Getting Started | Selenium*. https://www.selenium.dev/documentation/getting_started/, October 2021.

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

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

[13] Fowler, Matin: *ModelDrivenSoftwareDevelopment*. https://martinfowler.com/bliki/ModelDrivenSoftwareDevelopment.html, 2008.

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

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

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

[22] You, Evan: *VuePress | Vue-powered Static Site Generator*. https://vuepress.vuejs.org/, October 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, October 30, 2021

Mukendi Mputu