Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis Nr. 3467

# Tool Support for Software Architecture Documentation

Dimitrij Pankratz

| | |
|---|---|
| **Course of Study:** | Software Engineering |
| **Examiner:** | Prof. Dr. rer. nat. Stefan Wagner |
| **Supervisor:** | Dipl.-Inf. Ivan Bogicevic |
| **Commenced:** | 2013-03-01 |
| **Completed:** | 2013-08-31 |
| **CR-Classification:** | D.2.1, D.2.2, D.2.7, D.2.9 |

# Abstract

Large and complex software systems cannot be created in one piece. They need to be separated into manageable units, i.e., modules, which can be developed independently from each other. Precise definition and documentation of the modules and other software artifacts is essential for a successful software project. The module documentation is part of the overall software architecture documentation.
However, nowadays most of the utilized tools for documenting software architectures (e.g., word processors) are not fully sufficient for this task. This thesis provides a concept for an extensible tool, which is specialized for documenting software architectures and modules. The concept supports management and visualization of explicitly definable relations, e.g., dependencies among modules or references to external implementation artifacts. Moreover, this thesis suggests a template for module documentation based on its concept. Finally, a proof of concept prototype is implemented and evaluated with promising results.

# Contents

# List of Abbreviations

| | | |
|---|---|---|
| **ADL** | Architecture Description Language | 25 |
| **API** | Application Programming Interface | 19 |
| **CaC** | Component-and-Connector | 25 |
| **CLI** | Common Language Infrastructure | 22 |
| **CSV** | Comma-separated values | 47 |
| **DOM** | Document Object Model | 48 |
| **DSM** | Dependency Structure Matrix | 24 |
| **ER** | Entity-Relationship | 11 |
| **GUI** | Graphical User Interface | 29 |
| **HTML** | Hypertext Markup Language | 37 |
| **IDE** | Integrated Development Environment | 27 |
| **JAR** | Java Archive | 17 |
| **Java EE** | Java Platform, Enterprise Edition | 17 |
| **JAXB** | Java Architecture for XML Binding | 48 |
| **JCP** | Java Community Process | 20 |
| **JRE** | Java Runtime Environment | 22 |
| **JSR** | Java Specification Request | 20 |
| **JSON** | JavaScript Object Notation | 47 |
| **JVM** | Java Virtual Machine | 17 |
| **J-PaD** | Java Package Documentation | 12 |
| **MVC** | Model–View–Controller | 52 |
| **OOP** | Object Oriented Programming | 22 |
| **OSF** | Open Software Foundation | 54 |
| **PDF** | Portable Document Format | 33 |
| **SEI** | Software Engineering Institute | 24 |
| **SQL** | Structured Query Language | 23 |
| **SVN** | Apache Subversion | 12 |
| **UML** | Unified Modeling Language | 11 |
| **UniMoDoc** | Universal Module Documenter | 29 |
| **URL** | Uniform Resource Locator | 38 |
| **USB** | Universal Serial Bus | 70 |
| **UUID** | Universally Unique Identifier | 54 |
| **XML** | Extensible Markup Language | 17 |

# List of Figures

# List of Tables

# List of Listings

# 1 Introduction

Nowadays, a great number of people work together on the development of large and sophisticated software systems. These systems cannot be built in one piece due to the high complexity and enormous communication effort. In contrast, they need to be divided in discrete and manageable parts. These parts are often referred to as modules. Modules hide the complexity and implementation idiosyncrasies internally, while exposing only an interface to the outside world. The interfaces act as contracts between the modules. Thus with respect to the interfaces, modules can be developed fully independent from each other.

However, modules are not a new invention but known in software engineering since the early 1970s [Parnas, 1972]. As well as all the other software artifacts, they need to be precisely defined and documented. For these reasons, it is more astonishing that only little attention is paid to the tool support for documenting modules as of this writing. The module documentation can be considered as a part of the software architecture documentation. Typically software architecture is documented with word processors, formal notation (e.g., Unified Modeling Language (UML), Entity-Relationship (ER), etc.) tools, or a combination of both but specialized tools are hard to find.

## 1.1 Motivation

Using only UML or other formal notation tools for documenting software architectures is not sufficient. These notations support mainly the documentation of the solution. However, architecture documentation needs to provide a detailed description of the issue, its solution, argumentation about the validity of the solution, rejected alternatives, etc. Although, it needs to structure and link this information.

Compared to formal notation tools, word processors make it possible to describe all of the required information and structure it in a document. However, architecture documentation contains various relations among the single parts. These relations are necessary for, e.g., revealing the dependencies among the modules, finding an implementation unit for a specific requirement, or detecting the test cases for a module. Word processors provide very limited functionality for defining these relations. They do not support any central management or visualization for the relations. Since the architecture documentation is not a single artifact in a software project, it is necessary to describe the relations to external resources. Word processors support external references only rudimentarily.

Furthermore, the architecture documentation needs to be put together with the other software artifacts under revision control. However, word processors are poorly compatible with common revision control tools like Apache Subversion (SVN) due to their typically binary data formats and rudimentary referencing of external resources.
Additionally, the free text input in word processors requires further structuring and is less comfortable compared to form-based input.

A previous diploma thesis [Kircher, 2012] at the University of Stuttgart examined the lack of tool support for module documentation. As a result it suggests a form-based tool for module documentation on the source code level named Java Package Documentation (J-PaD) [J-PaD, 2013]. Still, J-PaD solves not all of the previously described issues. Especially, it does not provide any explicit definition, management, and visualization for relations. Moreover, J-PaD is limited to the documentation of Java packages. Though Java packages can be interpreted as modules, there are far more possible module constructs in Java only.

## 1.2 Problem Statement

The goal of this thesis is examining the current state of the art and creating a concept for software architecture documentation tool support. In contrast to [Kircher, 2012], the resulting concept needs to be programming language independent and applicable for various module constructs. Moreover, it should support templates, explicit relation definitions, relation management, and their visualization. Furthermore, it is necessary to build a prototype as a proof of concept and evaluate it.

## 1.3 Research Design

This thesis is created in a close co-operation with another diploma thesis [Casciato, 2013] on purpose. Both theses work on the same concept and prototype with partially different goals. In the concept and implementation phases, this thesis focuses primarily on the module documentation. It keeps an eye on the big picture, i.e., it provides the interfaces, pays attention to the extensibility, etc. This thesis defines a design for the overall application with a common data model and implements it. Additionally, it provides a module documentation template based of the resulting prototype.
[Casciato, 2013] plugs in into the overall design and implementation, focusing on the test aspects of module documentation and the relations, their semantics, management, and visualization.

However, it is not possible to simply draw a clear line between both theses. Many concepts were created in teamwork. Furthermore, the complete evaluation was planned and executed together with [Casciato, 2013].

## 1.4 Outline

This thesis is structured as follows:

**Chapter 1 – Introduction:** High-level overview and motivation of this thesis.

**Chapter 2 – State of the Art:** Definitions, technologies, and programs which are state of the art and related to this thesis.

**Chapter 3 – Requirements and Design:** Firstly, this chapter defines the requirements resulted from Chapter 2. Then it presents the ideas, concepts, and the resulting design with consideration of the defined requirements.

**Chapter 4 – Implementation and Results:** Constructing on top of the general design, this chapter describes the implementation details of this research. The second part of the chapter shows the results of the implementation.

**Chapter 5 – Evaluation:** Evaluation of the results of this thesis.

**Chapter 6 – Conclusion and Future Work** Summary, conclusion and an outlook of this research.

# 2 State of the Art

This chapter describes the foundations, provides an overview of the related work, and introduces relevant technologies.

## 2.1 Software Module

Building large software systems requires dividing them into smaller parts. These parts are usually called modules. Parnas provides the foundation for software modularization in his research [Parnas, 1972] in the early 1970s. Instead of separating the software in single blocks of the processing order (like single blocks of a flowchart), he proposes to decompose the software in modules with the consideration of difficult design decisions and possible changes in the future. One of the key aspects of this approach is information hiding. Hiding a difficult to understand design inside a module and exposing well defined external interfaces results in several benefits:

- It is not necessary to understand the implementation details in order to use the module.
- It is easier and faster to get the big picture of the whole system by understanding the interfaces of the modules and skipping the implementation details.
- Potential changes in the implementation of a module do not affect the implementation of other modules and cause changes inherently.
- The communication effort among the developers of different modules is reduced to the interfaces.

While the benefits of modularization in software engineering are well known and accepted, there is room for interpretation what is actually a module. This thesis focuses on a tool for structured documentation and specification of modules. Therefore, it is essential to clarify this term and its main aspects. There are many different definitions around on this topic.

> **module** - (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine.
> (2) A logically separable part of a program.
>
> [IEEE-610.12, 1990]

The definition of the IEEE standard glossary of software engineering terminology [IEEE-610.12, 1990] is very vague. It makes use of the term "unit" which is vaguely defined, too. In fact, there is a note beneath the module definition with the following contents:

> *Note:* The terms "module", "component", and "unit" are often used interchangeably or defined to be sub-elements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.
>
> [IEEE-610.12, 1990]

In addition there is a newer joint definition of ISO, IEC, and IEEE from 2010 [ISO/IEC/IEEE, 2010]. Still, instead of being more precise, it only adds additional variants to the existing definition. The relationship of the terms "module", "component", and "unit" are not yet standardized.

> **module** - 1. a program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.
> 2. a logically separable part of a program.
> 3. a set of source code files under version control that can be manipulated together as one.
> 4. a collection of both data and the routines that act on it
>
> [ISO/IEC/IEEE, 2010]

These definitions provide almost no criteria for clear identification or specification of software parts as modules. Other definitions mention some criteria, e.g., information hiding in [Ludewig and Lichter, 2007]. With these definitions in practice it is a question of interpretation what is a module and what is not. In order to illustrate this dilemma, the following chapters will describe the possible interpretations for a concrete programming language, namely Java.

> A **module** is a set of operations and data which are only externally visible to the extent defined by a programmer explicitly.
>
> [Ludewig and Lichter, 2007]

### 2.1.1 Modules in Java

At first glance, it seems to be simpler to find a definition of a module for a specific programming language. This chapter will show that this is not true, using the example of Java. There are different constructs in Java, which can be seen as modules. Some constructs are built-in parts of the language and some are not. The next sections will present those constructs in detail, provide the current state of the art for modularity in Java today, and show an outlook about future development.

Classes

There are good reasons to assume Java classes as modules themselves. In fact, modularity and object orientation have similar objectives and benefits, e.g., you can easily ensure information hiding with Java classes. While it is reasonable to argue due to the upper definitions that Java classes are modules, there are some limitations. The granularity of Java classes is relatively fine. Java projects tend to have thousands of classes and this usually requires additional structuring with coarse granularity on top, e.g., packages, OSGi bundles, etc. Referring to the module definition from [Ludewig and Lichter, 2007] and according to [Garlan et al., 2010] information models, e.g., configuration files, Extensible Markup Language (XML) files, etc. can be seen as modules, too. Java classes provide no functionality to group such information models to single modules. This again requires additional constructs on top of Java classes. Finally, Java classes are not deployable units.

Packages

Compared to Java classes, packages provide a coarse granularity and allow to group source code and information models. Still, Java packages have some limitations. Their initial purposes are namespaces in order to prevent naming clashes and simple structuring. Still, they provide almost no information hiding. Used and exposed interfaces can be only found on the class level. Thus, packages provide no explicit interfaces at all. Like classes they can not be deployed on their own.

OSGi

OSGi [OSGi, 2013] is a module platform on the top of the Java Virtual Machine (JVM). Formerly OSGi stood for “Open Services Gateway initiative” but since 2003 the acronym meaning is officially dropped. The specification is managed by the OSGi Alliance, which is a consortium of different companies, e.g., IBM, Oracle, Adobe, etc. OSGi provides a set of specifications for dynamic modularity in Java. Comparable to other Java technologies like Java Platform, Enterprise Edition (Java EE) OSGi follows the approach of a central specification with different implementations. Thus, there are different commercial and open source OSGi Framework implementations, e.g., Eclipse Equinox [Eclipse Foundation, 2013b], Apache Felix [Apache Felix, 2013], Knoplerfish [Knopflerfish, 2013], etc.

One of the key elements of OSGi’s modularity is the bundle. Simply put, from OSGi’s perspective a bundle is a module and can be defined as followed:

> **Bundle** A physical unit of modularity in the form of a Java Archive (JAR) file containing code, resources, and metadata, where the boundary of the JAR file also serves as the encapsulation boundary for logical modularity at execution time.
>
> [Hall et al., 2011]

**Listing 2.1** Example OSGi manifest definition (`Manifest.mf`)

```
Bundle-Name:                 UniMoDoc HTML Exporter
Bundle-Description:          A bundle for HTML export of documentation
Bundle-Copyright:            (c) 2013, University of Stuttgart
Bundle-SymbolicName:         de.unimodoc.export.html
Bundle-Version:              1.0.0
Export-Package:              de.unimodoc.export.html; version="1.0.0"
Import-Package:              de.unimodoc.core; de.unimodoc.export; version="1.3.0"
Bundle-Activator:            de.unimodoc.export.html.Activator
```

Consequentially, a bundle is actually a simple JAR file enriched with additional metadata. These metadata are contained in the `Manifest.mf` file and can be structured in three types:

- Human-readable information
- Bundle identification
- Code visibility

Human-readable information is intended for humans only. It is not required and is ignored by the OSGi framework. This information contains besides others data like:

- `Bundle-Name`: Well readable name for humans, which has not to be unique.
- `Bundle-Description`: Describes what functionality the bundle provides.
- `Bundle-Copyright`: Copyright information of the bundle.

Bundle identification information is required for distinct bundle identification. Since `Bundle-Name` is not unique, not required and for humans only, there is another field for bundle identification. So the bundle identification information contains a unique name with an optional version field as follows:

- `Bundle-SymbolicName`: Required unique bundle name.
- `Bundle-Version`: Version of the bundle in a common OSGi version number format.

Code visibility metadata describe formally and explicitly, which functions from the bundle are visible to the outside world. It provides also information about the dependencies to other bundles, so to speak, inbound visibility. A set of possible data in this metadata type is:

- `Export-Package`: Defines explicitly, which packages will be visible to others.
- `Import-Package`: Lists the required dependencies of this bundle.
- `Bundle-Activator`: The specified class in this field will be invoked after the start of the bundle.

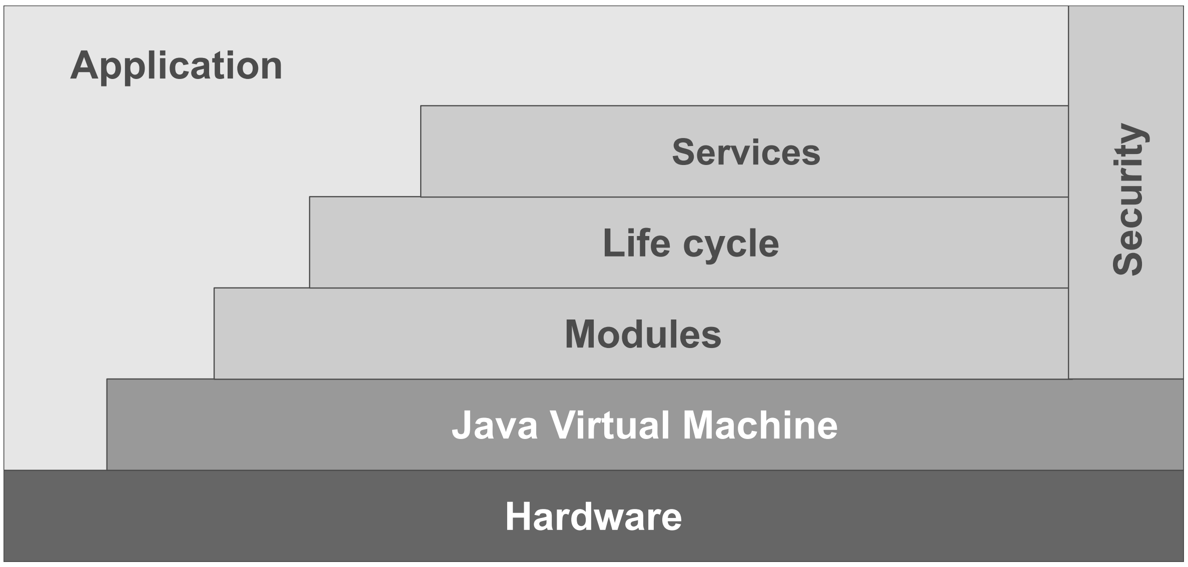


**Figure 2.1:** OSGi architecture

Listing 2.1 shows an example bundle manifest definition with the described metadata.

The OSGi framework builds its layered architecture on top of the JVM. The architecture is displayed in Figure 2.1. As already mentioned, in terms of OSGi, a module is equivalent to a bundle. The definition of bundles as single units of modularization forms the *Module Layer* of OSGi. It ensures and resolves fine-grained dependencies for bundles, adds explicit information hiding and provides a more sophisticated class loading concept than standard Java.
The *Life Cycle Layer* defines the execution-time management of the modules and provides an Application Programming Interface (API) for this purpose. With the defined life cycle of bundles, it is possible to dynamically install, start, update, stop, and uninstall bundles during the runtime of the application, i.e., without a restart. On the one hand, the *Life Cycle Layer* allows the application and the administrator to manage the bundles. On the other hand, it provides an API for the bundles to register and hook into the framework in a well-defined way.
The *Service Layer* extends the modularization with concepts of service-oriented computing. It has a central service registry, so bundles can register the services, which they provide. The application can dynamically request these services from the registry. Basically, this is an implementation of the Publish-Find-Bind interaction pattern in Figure 2.2. Internally, the OSGi services are Java interfaces. At first glance, this is nothing new to the Java world. This approach with Java interfaces is very familiar to the developers and it fits into the language structure. The real benefit of the service registry comes from the combination with the *Life Cycle Layer* and *Module Layer*. Service providers are managed dynamically as bundles. They can be added and removed while the application is running. This provides more flexibility and modularity for Java-based applications.

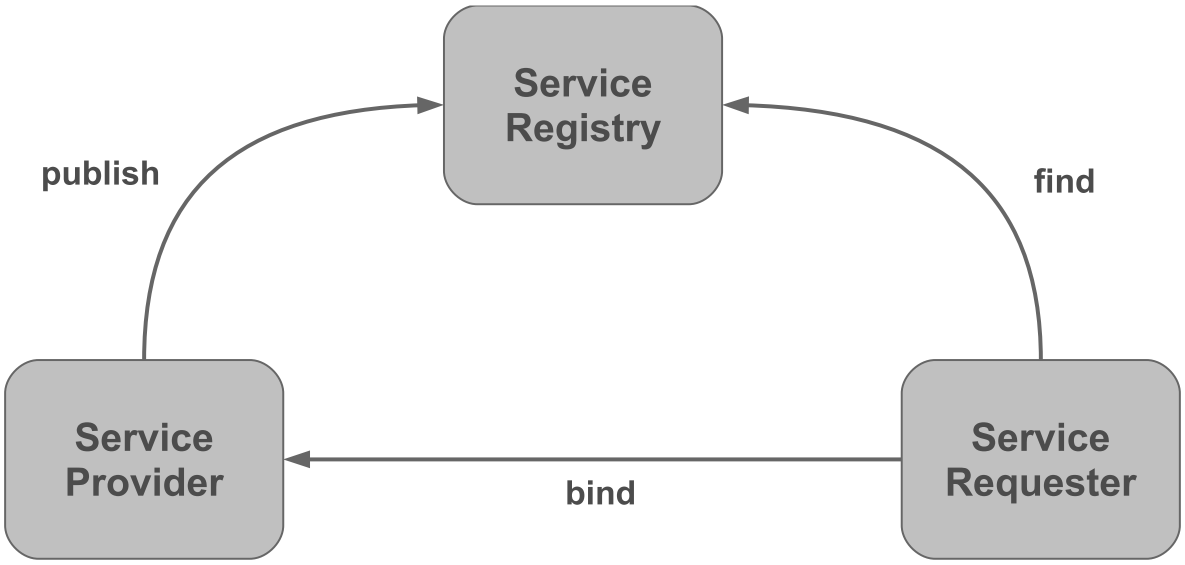


**Figure 2.2:** Publish-Find-Bind interaction pattern

The *Security Layer* is based on Java 2 security and extends it with additional functionality for bundles. It provides two main functionalities. The first one defines how bundles can be securely packaged and digitally signed. The second functionality defines the security options during the runtime, i.e., the permissions of a specific bundle. The *Security Layer* in OSGi is optional and will only run on Java platforms, which provide the necessary *Security APIs*.

OSGi solves many of Java's limitations in modularity. It provides a precise module definition with explicit information hiding, dependency control, life cycle management, and much more. The main downside of OSGi is that it is not a part of the Java standard and it requires an additional custom framework on top of Java. Additionally the configuration and management effort will usually increase firstly with an OSGi framework before you can profit from a better modularity.

Project Jigsaw

Besides OSGi, efforts have been made as well in order to introduce modules as a part of a Java standard. Java Community Process (JCP) defines a formalized process for extending of Java and its standard libraries. Single propositions are managed in so called Java Specification Request (JSR)s. Before a proposition is finally accepted and applied to Java, a team of experts is formed around a JSR. This team is responsible for the specification and the creation of drafts.
In 2005 JSR 277 [Buckley et al., 2006] was announced. Its purpose was to bring a module framework with a defined packaging format and a central module repository into Java. In 2006 JSR 294 [Buckley et al., 2007] was announced with the objective to extend Java with

a `superpackage` notion, which could group normal packages into modules with specified information hiding. This approach was then changed in favor of JSR 277. So JSR 294 attempts now to define a module-level access modifier, which is compatible with JSR 277. Unfortunately JSR 277 is on hold since 2008 and JSR 294 made only little progress since then.

Meanwhile, Sun Microsystems introduced Project Jigsaw [OpenJDK, 2013] for a standard module system in Java. Originally Project Jigsaw was planned to be officially included in Java 7. However, this didn't happen and lastly it was announced for Java 9 in 2015 [Reinhold, 2012]. Project Jigsaw targets on extending Java with modules so that they are part of the language construct. In terms of Project Jigsaw a module is defined as followed:

> A **module** is a collection of Java types (i.e., classes and interfaces) with a name, an optional version number, and a formal description of its relationships to other modules. In addition to Java types a module can include resource files, configuration files, native libraries, and native commands. A module can be cryptographically signed so that its authenticity can be validated.
>
> [Reinhold, 2011]

Project Jigsaw will add the keyword `module` to Java and allow by convention to store the definition of modules in a `module-info.java` file for a compilation unit. The only required information is the module name, which is a qualified Java identifier (comparable to Java packages). Dependencies are managed with the keywords `exports` and `requires` within the module definition. So a module can explicitly define with the exports keyword which parts are externally exposed in order to ensure information hiding. The `requires` keyword defines on which functionalities a module depends. Executable modules may define an entry point with a `class` keyword and an explicit path to a class with a `public static void main` method. In addition, Project Jigsaw allows defining service providers and service consumers. A service within Project Jigsaw is a Java interface or abstract class. A module can define which service implementation it provides by `provides service`, following service name, and then the `with` keyword followed by the path of the service implementation. The `requires service` keywords followed by the service name designate a module as a service consumer. Listing 2.2 shows an example module definition with the concepts previously discussed.

Project Jigsaw will solve many modularity issues in Java comparably to OSGi. In fact, there are many parallels to OSGi. The module definition of Project Jigsaw is comparably powerful as the *Module Layer* of OSGi and provides quite similar features, e.g., module versions, dependencies, entry points, etc. In Project Jigsaw modules can be cryptographically signed like in the *Security Layer* of OSGi. Even the services from Project Jigsaw can be found in OSGi on the *Service Layer* with the same terminology.
Still, there are many differences in detail and some major differences between Project Jigsaw and OSGi. On the one hand, the major benefit of Project Jigsaw is that it extends the Java language construct itself. So there is no need for a custom framework on top of Java. On the other hand, there are some drawbacks about Project Jigsaw. It is still a draft, so it is will be released at the earliest with Java 9. The *Life Cycle Layer* of OSGi is more powerful because it

**Listing 2.2** Example Project Jigsaw module definition in a `module-info.java` file.

```
/**
 * UniMoDoc HTML Exporter
 * A bundle for HTML export of documentation
 * (c) 2013, University of Stuttgart
 */
module de.unimodoc.export.html @ 1.0.0 {
    requires de.unimodoc.core @ >= 1.3.0;
    exports de.unimodoc.export.html;
    class de.unimodoc.export.html.Activator
    provides service de.unimodoc.export.Exporter with de.unimodoc.export.html.HTMLExporter;
    requires service de.unimodoc.export.ExportSettings;
}
```

allows adding and removing modules during the runtime of applications, what is explicitly not planned by Project Jigsaw. Another fundamental difference between these two approaches is that Project Jigsaw aims also to modularize the Java Runtime Environment (JRE) itself.
When Project Jigsaw will be released, many applications will still use OSGi. Also, due to the different features it might be beneficial in some cases to use both approaches from OSGi and Project Jigsaw. This will require interoperability or at least tolerance between these approaches. Project Penrose [Ellison, 2013] aims to solve these issues.

### 2.1.2 Software Modules Conclusion

The previous sections show that in practice even for a specific language there can be different interpretations of a module. This is especially the case if the language itself provides no distinct construct for module support like Java. At the same time efforts like OSGi and Project Jigsaw demonstrate that there is a need for modularization beyond the default capabilities of Object Oriented Programming (OOP) in Java. These different approaches illustrate also that there are various degrees of modularization. They provide dissimilar capabilities and features of modularization and therefore can be useful in various use cases depending on requirements.

This confusion in terms of modules appears not only in Java. In C# OOP constructs like class or namespace (comparable to package in Java) and assemblies from the Common Language Infrastructure (CLI) can also be assumed as modules. The CLI assemblies are compiled code libraries with a manifest, versioning and security settings. They are deployable units comparable to OSGi bundles in Java.
There are also programming languages which support no modularization with a specific language construct at all, e.g., JavaScript. Instead, in order to achieve modularity in JavaScript often variations of a Module Pattern [Stefanov, 2010] are used to encapsulate functions and data with an interface in a closure. Table 2.1 shows a set of languages paired with an incomplete list of possible module interpretations.

| **Programming Language** | **Possible Module Interpretations** |
|---|---|
| C# | class, namespace, assembly |
| Java | class, package, OSGi bundle, Jigsaw module |
| JavaScript | Module Pattern |
| PHP | class, namespace |

**Table 2.1:** Module interpretations in different programming languages

In conclusion, there are different approaches and degrees of modularity in practice. Therefore, it is hard to find a general and precise definition of a module to cover and identify all the possible variations. These variations are on their own rights and useful in different use cases depending on particular requirements. Due to these findings this thesis will stick to a very general IEEE definition of a module:

> **module** A logically separable part of a program.
>
> [IEEE-610.12, 1990]

As a result, a tool for documenting software modules should be able to handle different kinds of modules. Even a tool for documenting modules of only one programming language like Java will inevitably come across various kinds of modules. It is also usually not possible to define a finite set of those module types. This is especially the case because software projects in practice are not limited to a single programming language, but are multilingual, i.e., they contain a mixture of programming languages. For example, a single web-based software project can contain a mixture of Java for application back end, Structured Query Language (SQL) for database operations, JavaScript for front end, and ActionScript for complex visual effects in the front end. This leads this thesis to a requirement for a module documentation tool, being generic enough to handle various types of modules in different programming languages.

## 2.2 Software Architecture Documentation

Software architecture describes the high-level structure of software, its components, interfaces, relations, etc. This information is required by different stakeholders, e.g., customers in order to ensure the requested requirements and quality is delivered, implementers in order to understand and build the system, testers in order to create and run tests, etc. There is no unified definition for software architecture similar to modules. In fact, the Software Engineering Institute (SEI) has a collection of over 150 definitions for the term software architecture on its website [SEI, 2013]. This thesis will stick to the IEEE definition:

> **Software architecture** - The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
>
> [IEEE-1471, 2000]

Comparable to the definition of software architecture there is no unified way for documenting software architectures and no standard structure for it. Still, there are some generally accepted approaches for documenting software architectures. One of these approaches is the view concept. Usually it is not possible to describe the complete software architecture in one dimension only. Therefore, different views are used in order to describe the software architecture from different perspectives. IEEE defines the view as follows:

> **View** - A representation of a whole system from the perspective of a related set of concerns.
>
> [IEEE-1471, 2000]

For a specific software architecture there are various possible views and it is in the responsibility of the software architect to choose the relevant ones. In the past, some view categories have proved to be successful for many software architectures. Sometimes these view categories have different naming in the literature, but the meaning is almost the same. This thesis uses the naming from [Garlan et al., 2010]. In [Garlan et al., 2010] the three view categories are called Module Views, Component-and-Connector Views, and Allocation Views. These view categories are described in the following way.

**Module views** divide the whole system into single modules in terms of Chapter 2.1 and define their relationships to each other. Such relations between the modules are *Is part of*, *Depends on*, and *Is a*. There are various possible notations to describe the Module views. Informal graphical and textual notations face a variety of formal notations, e.g., UML, Dependency Structure Matrix (DSM), ER diagram, etc.
The Module views answer several purposes. They provide a plan for building the source code and the modules can usually be mapped to a set of source code artifacts. The single modules always implement a functionality which is part of the system requirements. As a consequence, all modules of a Module view implement all the functional requirements. Therefore, one can analyze whether the architecture meets the requirements. In addition, the Module views are

some kind of communication vehicle. Depending on their granularity, they can serve as a starting point for understanding and learning the complete system.

**Component-and-Connector (CaC) views** represent the system during the runtime. In [Garlan et al., 2010] the single elements of such views are called components, e.g., objects, processes, etc. The relations between them are called connectors, which represent communication, protocols, interaction, data flow, etc. Sometimes components can be mapped to modules but not necessarily. Comparably to the Module views, CaC views can be represented as well with various informal graphical and textual notations, as with formal notions, e.g., UML, Architecture Description Language (ADL)s, etc.
CaC views show which parts of the system exist during the runtime and how they interact together. This information is essential to address issues of parallelism, timing, communication, performance, availability, reliability, and others. CaC views are meant for modelling and solving of those issues in order to meet the functional and non-functional requirements.

**Allocation views** show how modules from the Module views or components from the CaC views can be mapped on single parts of a particular environment. Possible environment can be the hardware on which the software will run or the organization structure of the people, who will implement the software. Depending on the actual view, the relations between modules or components and environmental parts have the semantics of Allocated-to. Besides the informal notions, Allocation views can be represented with UML. In case of a work assignment view, a tabular notation is conceivable.
The Allocations views are useful for estimating and analyzing whether the requirements of software can be met in the corresponding environment. That way issues from the analysis (e.g., performance bottlenecks) can be discovered and resolved. In case of mapping the software on hardware for deploying or installing, Allocation views can be used for creating build and deployment plans.

This thesis focuses on the software architecture documentation especially form the Module views perspective. It assumes that the modules are primarily defined and documented by the architect in the software architecture documentation. The modules from the Module views contain a mapping to the actual source code artifacts. This way the overview information of a module can be found in the software architecture documentation and the fine-grained implementation details of the source code artifacts are linked from the overview. The result is a complete documentation of modules. Since the module documentation is part of the software architecture documentation, their stakeholders are the same. The complete list of software architecture documentation stakeholders is listed in Table A.1.

## 2.3 Templates and Tools for Software Architectures

The lack of standard structure for software architectures is an issue. Each time new software architecture is created a suitable and commonly accepted structure needs to be found. This cause additional time expenditure. Finding and understanding relevant information for readers

is more time consuming. Learning and especially copying from approved software architectures is more complicated.
Therefore various templates for software architectures exist, which try to solve these issues [arc42, 2013], [Garlan et al., 2010], [IEEE-1471, 2000], [Knöpfel et al., 2005], [FMC, 2013], [FEA, 2013], [MODAF, 2013], [TOGAF, 2013], [RM/ODP, 2013]. They prevent from reinventing the wheel and usually provide a tried and tested basis. Software architecture templates are typically simple documents created with a word processor. As an example, [Starke and Hruschka, 2011] proposes a template which is delivered as a Microsoft Word file. However, word processors are not sufficient for creating software architecture documentation and templates for those because they do not provide support for structured input and relations between the single parts of the document. Such templates contain headings for the single document parts and provide free text support only. Forms are better applicable for this kind of documents and templates because they benefit the structure, simplify the input procedure, and are less time consuming during the input. Clear relations between the document parts are essential, e.g., to track down which requirements are implemented and by which software parts, to identify the dependencies between modules, or to find out where are the corresponding source code artifacts for a particular module. Unlike free text, forms are well suitable for input of such relations. Moreover, the software architecture documentation should be in the revision control like the source code artifacts and other software documents. Files generated by word processors, e.g., Microsoft Word or Apache OpenOffice are binary. Thus they are not suitable for revision control tools like SVN or Git. The differences in versions cannot be compared automatically and therefore not efficiently merged.

Beside the word processors there is only little tool support for documenting software architectures. Tools for UML and other formal notations are not sufficient for documenting complete software architectures because they support only the illustration of a solution but not the reason. A software architecture documentation needs to list the requirements (or at least to reference them), present the solution, and to argue the reasons, why this is a suitable solution. For a single issue numerous solutions might exist. Each of which is a trade-off against the requirements. Therefore the architect needs to depict an argumentation for a solution. Software architectures are usually structured as normal text documents. Formal notation tools are used in addition for diagram creating. Single software architecture documentation can contain diagrams in different formal notations, e.g., UML for Java classes and ER diagrams for database tables.

Another category are Web-based documentation tools. Hyper-linking between single pages eases the navigation in the documentation and solves the issue of missing relations between document parts of word processor files. [Garlan et al., 2010] suggests especially wikis for Web-based software architecture documentation. Wikis allow parallel editing and a central repository of documentation for different users. Various stakeholders can work on different parts of the documentation in parallel and create it together. The drawbacks of wikis are that they need to be accessible from the Web or Intranet and they provide no structured form-based input like word processor files. Moreover, [Starke and Hruschka, 2011] notes that wikis tend to grow into information graves. If various stakeholders can contribute in an uncontrolled amount to the documentation, it can easily lose its clarity and relevance.

## 2.4 J-PaD

This thesis is partially based on results and findings of [Kircher, 2012]. In his diploma Kircher points out the relevance of module specification and documentation in software development and created J-PaD [J-PaD, 2013], an Eclipse based tool for module documentation. The following section describes the above mentioned research in more detail and shows the relevant differences.

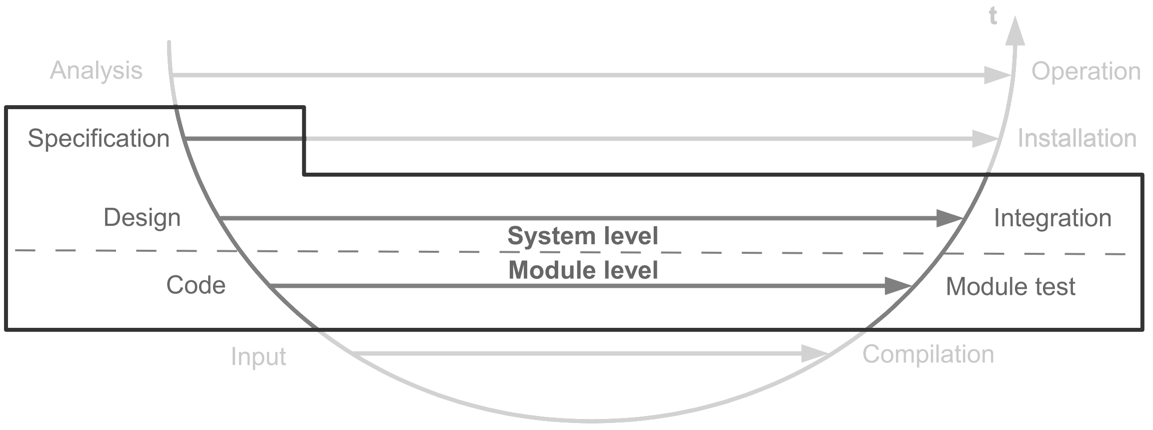


**Figure 2.3:** Bathtub Curve from [Kircher, 2012]

Kircher refers to his interpretation of the Bathtub Curve in Figure 2.3. It shows the different abstraction layers during the software development. According to him, the left-side of the chart describes the top-down approach (usually used during the development) and the right-side describes the bottom-up approach (usually used during the testing). Activities of the same abstraction layer are horizontally interconnected. Further, he argues that the objectives from the overall software specification and design are used for system testing and integration testing. Still, there is need for module specification and documentation in order to test on a lower level, e.g., single modules, unit testing, etc. Kircher concludes this type of documentation as part of the source code. His examination of the available tools for source code documentation shows their lack of module support. As a consequence of the missing tool support, Kircher develops within the scope of his research J-PaD. In the scope of his research and this tool he assumes Java packages as modules but provides no further explanation for this decision. J-PaD is integrated in the Eclipse Integrated Development Environment (IDE) and provides a simple way to create module documentation with predefined data fields. The input is stored in the `package-info.java` file. This is a standard description file for packages in Java. Data which is not part of Javadoc is stored in an additional comment field below the header comment. The structure of the module documentation is configurable by a schema file.

This thesis only partially agrees on Kircher's interpretation of the Bathtub Curve and his conclusion. The original statement of the chart in Figure 2.4 notes: The sooner a fault is made

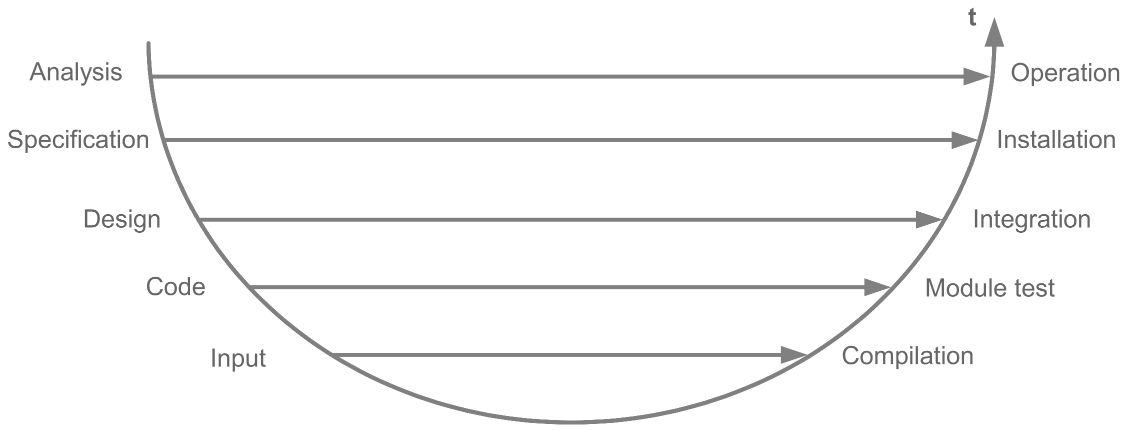


**Figure 2.4:** Bathtub Curve from [Ludewig and Lichter, 2007]

during the software development, the later it will be usually discovered, typically on the same abstraction layer. The fact that a fault made in source code typically can be discovered with unit testing, does not imply that unit tests themselves are only derived from the information out of the source code. This also does not mean that the module documentation is necessarily located on the code level. And it does not imply that the module documentation is only relevant for module testing. On the contrary, information from the module documentation like dependencies or interfaces of modules is required first of all for integration testing.
This thesis assumes software architects as the initial authors of module documentation and in this point it is even consistent with the research of Kircher. Different from his conclusion, this thesis considers module documentation as part of the software architecture.
Another important difference to Kircher's research is the definition of a module. As already discussed in Chapter 2.1, this thesis does not limit modules to Java packages.

This thesis will address the relevance of module specification and documentation in software development pointed out by [Kircher, 2012]. It will also apply some of the identified metadata from [Kircher, 2012] for describing modules in a new tool and extend these metadata. In contrast to [Kircher, 2012] this thesis provides a more general definition of modules described in Chapter 2.1 and support various programming languages, not Java only. Unlike [Kircher, 2012] this thesis will also move the focus from software developers and documentation on the code level to software architects and software architecture level as described in Chapter 2.2. This way the module definition and description is placed within a central and accepted document, fits in its structures, links the source code artifacts, and builds a complete documentation of modules. Compared to [Kircher, 2012], the documentation on the software architecture level extends the interested audience in module documentation to the stakeholders of software architecture documentation in Table A.1.

# 3 Requirements and Design

This chapter defines the requirements for a module documentation tool called Universal Module Documenter (UniMoDoc) that is based on the results of Chapter 2. Further this chapter describes a general design for such a tool, which will take into account and solve the previously defined requirements.

## 3.1 Requirements

This chapter contains the requirements for a software architecture documentation tool. The requirements are split into functional and non-functional requirements. They are based on the initial problem statement of this thesis and the research results of Chapter 2.

### 3.1.1 Functional Requirements

This part describes the required functions of the software.

**Tool for creating module documentation**

The resulting tool supports documentation for modules. This is an initial requirement for this thesis.

**Tool for creating and editing software architecture documentation**

According to the argumentation in Chapter 2.2, the documentation of modules is part of the software architecture. Furthermore, the research shows a lack in tool support for software architectures. Module documentation in terms of Module views is related to other views. Therefore, it is reasonable to create a general tool for software architecture documentation in order to document the modules.
Still, the focus of this thesis lies on the module documentation. Due to the limited time frame for a diploma thesis of six months, this focus results in a rudimentary support for other software architecture parts.

**Form-based Graphical User Interface (GUI)**

The GUI of the tool is form-based in order to ease and structure the input of the user.

**Definition of module metadata**

In order to document modules, common metadata for the description of modules is required. These metadata are defined and supported by the resulting tool.

**Template support**
: The resulting tool supports templates. New templates can be created, saved and loaded within a tool. The templates provide a structure and example data for a complete software architecture document or single parts of it.

**Module documentation template**
: The defined metadata for module documentation are provided as a template in the resulting tool.

**Support of different programming languages**
: Unlike [J-PaD, 2013], the resulting tool is not limited to a specific programming language or module construct.

**Documentation structure support**
: The documentation created by the resulting tool has a structure. The structure of documentation is required to divide it in logically related parts and to prescribe a specific structure in a template.

**Relations between document parts**
: It is possible to crosslink the different parts of the document. This is required for a better navigation within the document and for the support of relations required by software architecture documentation as described in Chapter 2.2.

**Documentation exportability**
: The design of the resulting tool supports interfaces for export functionalities. Thus export format for the documentation can be easily added.

**Function for including graphics**
: It is possible to embed graphics like UML diagrams in the documentation.

**References to external data**
: The resulting tool supports references to external data like source code files, test cases, images, etc.

**Multilingual support**
: The resulting tool is designed and implemented in a way that allows to extend it for other languages, e.g., by externalizing the strings in property files.

**Line-based format**
: All created data and templates of the tool are saved in a textual line-based format in order to ensure best possible compatibility with revision control tools like SVN.

**Java tool**
: The resulting tool is implemented in Java and Swing and is a standalone application.

### 3.1.2 Non-functional Requirements

This part denotes the quality requirements of the software.

**Extensible GUI**
The GUI is as well and easy extensible as possible.

**Extensibility for other software architecture documentation parts**
The focus of this thesis lies on the module documentation and other software architecture parts are only rudimentary supported. Therefore, it is important that the resulting tool can be easily extended in order to better support the other parts of software architecture. This is achieved by foresighted design, defined extension points and good documentation.

**Good maintainability**
The tool is as well maintainable as possible by providing a good documentation and simple extensibility.

**Good usability**
The resulting tool provides good usability by intuitive user guidance and clear messages.

### 3.1.3 Stakeholders

The stakeholders of UniMoDoc consist of two groups, which correspond to following use cases.

#### Documenting and using software architectures

In the first place the stakeholders of UniMoDoc are equal to the stakeholders of the software architecture documentation. Thus software architects may use UniMoDoc for documenting software architectures. Furthermore, they can create and share templates. Implementers can use UniMoDoc in order to understand and learn the entire software system or its parts from the documentation. Architects may communicate with implementers via the documentation by noting implementation advices and constraints. In addition, UniMoDoc supports explicitly testers with extensive test documentation for modules covered in [Casciato, 2013]. The complete overview of software architecture documentation stakeholders is denoted in Table A.1. The use of UniMoDoc for documenting software architectures does not require specialized knowledge.

#### Extending UniMoDoc

Every person who may extend UniMoDoc is regarded as a stakeholder. Therefore, UniMoDoc provides numerous extension points. Extending the tool requires knowledge in Java software development.

## 3.2 Design

This chapter takes into account the current state of the art from Chapter 2 and describes the concepts for solving the requirements from Chapter 3.1. Some of the requirements prescribe a technological frame. Still, the design is kept intentionally abstract. Therefore, it can be easily ported to other technologies or context.
Firstly, this chapter describes the general concepts of this thesis in 3.2.1 Concepts. It provides a high level overview of the concepts mainly from the perspective of the potential user, keeping the most technical details in background. Secondly, this chapter suggests a data model on a technical level in 3.2.2 Data Model, which was taken as a basis for the implementation. Finally, this chapter defines the relevant metadata for modules in 3.2.3 Module Metadata. This collection of metadata can be aggregated to a module documentation template. An example module documentation template using these metadata are suggested in the implementation part in Chapter 4.

### 3.2.1 Concepts

Chapter 3.1 describes the requirements and defines that the resulting tool should support creating and editing software architecture and module documentation in particular. Chapter 2.3 shows the lack of specialized tool support for software architecture. Instead of specialized tools, word processors are often used to create software architecture. However, word processors provide only poor functionality to crosslink the single parts of the document among each other and with external resources, e.g., source code, test data, and other documents. They are not well suitable for common revision control tools due to the binary formats. Furthermore, they do not support complex and structured data input but free text only. The basic idea behind the concept of this thesis is to build on the established and well-known approaches of the word processors and to overcome their disadvantages at the same time. In this manner, users can apply their accustomed procedures to the new tool and get started faster.

#### Document

The document is a metaphor for the complete file containing the software architecture documentation. The name of the document results implicitly from the file name. The document is the top level element in the structure and bundles all other sub elements, which are discussed in the next paragraphs.

#### Chapters

One basic element of the document structure is the chapter. The chapter is also adopted as a metaphor of a document part from the word processors. It bundles related information below a caption. Chapters are hierarchical. They always have exactly one parent, which is either the document or another chapter. Furthermore, chapters can have zero or more children chapters.

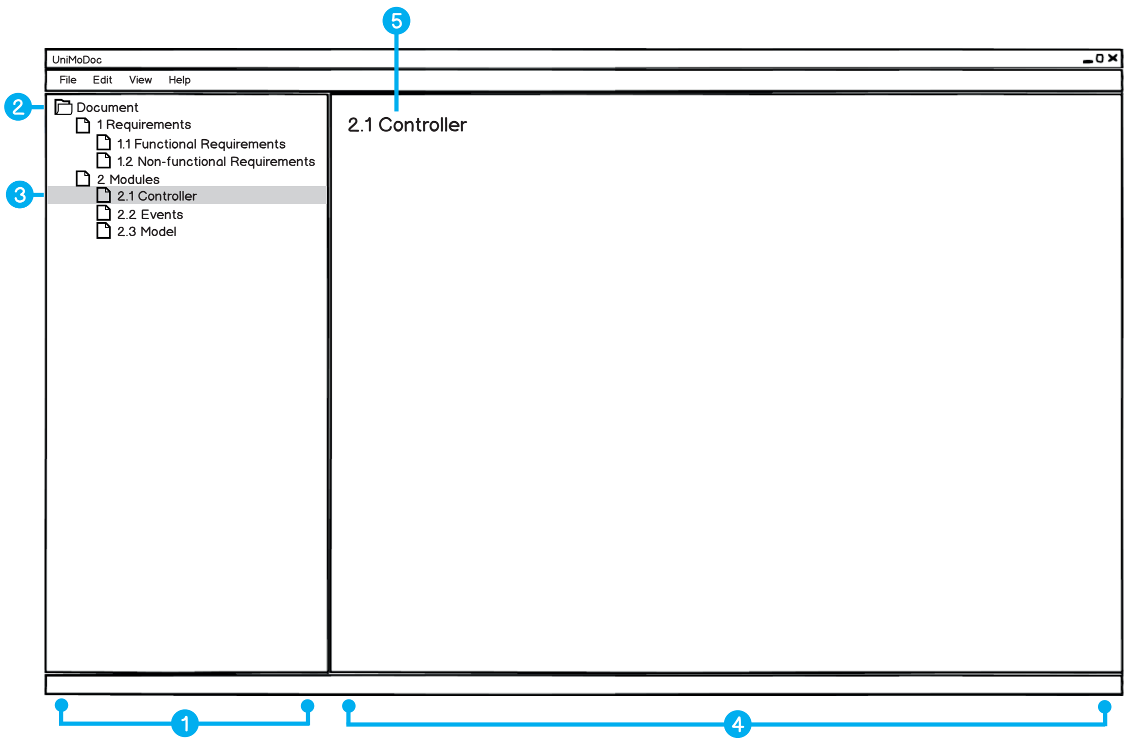


**Figure 3.1:** Document and Chapters in UniMoDoc
*Item ❶: Document Structure, Item ❷: Root Element, Item ❸: Selected Chapter, Item ❹: Content Area, Item ❺: Name of the Selected Chapter*

Since chapters are generic elements and they can be nested freely, this approach can realize various structures.
In addition to it, chapters have a number identifying the position of the chapter in the document structure. It is recursively compound of the order numbers of parent chapters. The described information is represented in Figure 3.1.

Figure 3.1, Item ❶ on the left-hand side shows the hierarchical structure of the document in a tree pane. That way the user can oversee the complete structure of the document, navigate through it, and keep track of the current location in the structure. Figure 3.1, Item ❷ points to the root of the structure, which is the document itself. Figure 3.1, Item ❸ illustrates a selected chapter in the document structure. In this case, the currently selected chapter is *2.1 Controller*. Its content is displayed in the content area (cf. Figure 3.1, Item ❹) on the right-hand side. The content area is meant to display the contents of selected chapters and make them editable to the user. In the top of the content area the caption of the selected chapter is placed (cf. Figure 3.1, Item ❺).

This concept allows the user to navigate, view, and edit the document simultaneously. Similar navigation concepts are well-known from file managers (e.g., Microsoft Windows File Explorer [Microsoft, 2013a], Apple OS X Finder [Apple, 2013], etc.), Portable Document Format (PDF) readers (e.g., Adobe Reader [Adobe, 2013], Foxit Reader [Foxit Corporation, 2013], etc.), IDEs (e.g., Eclipse [Eclipse Foundation, 2013a], Microsoft Visual Studio [Microsoft, 2013b], etc.), and therefore are familiar to many users.

#### Sections

Chapters partition the document into logical parts and provide a coarse-grained structure. Thus chapters are groups of related information and they are not atomic. Therefore, sections provide a finer granularity level of structure. Sections partition the chapters into atomic information chunks. Consequently, unlike the chapters, these chunks are not hierarchical. Still, there are many similarities between chapters and sections. Sections are also logical parts of the document but on a lower level. Comparably to the chapters, they have a caption and an order. The order of the sections in the document is implicit because the number is not displayed to the user but regarded internally. Both, chapters and sections have contents. While the contents of the chapters are the sections, the contents of the sections are discussed in the next paragraph.

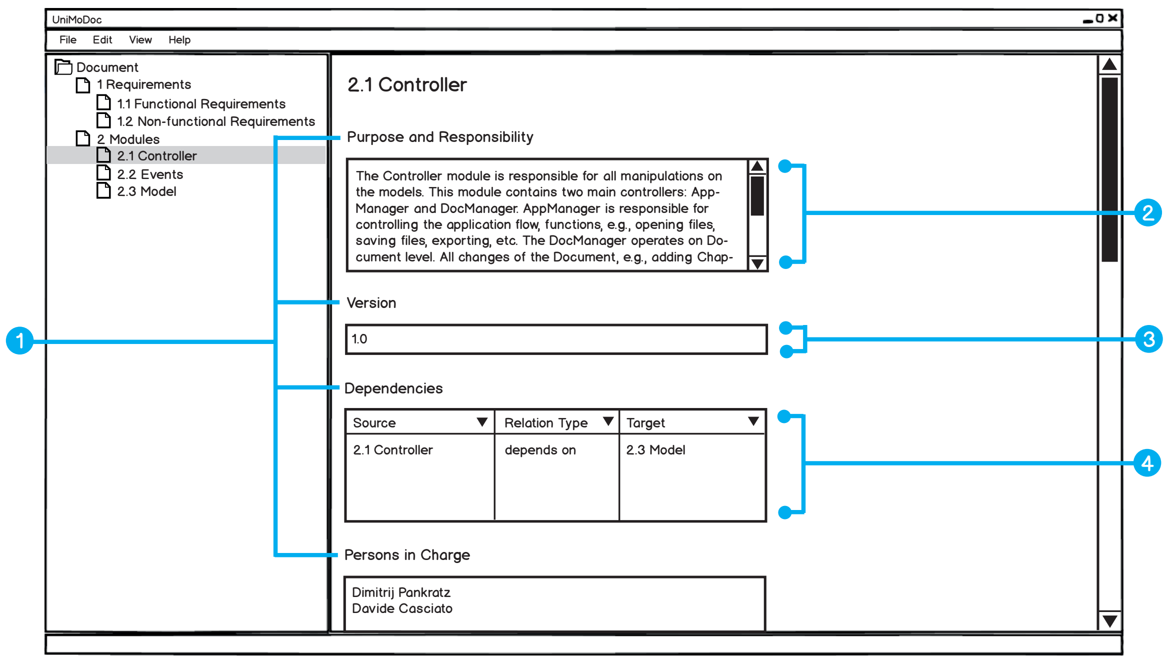
#### Widgets

The content of a single section has a specific data type. It can be a simple text, markup language, programming language, date, image, reference to another file, and many others. Therefore, a section needs to render the content and make it editable to the user somehow. At this point the widgets come into play. Widgets are visual components, which do exactly this job. They are bound to sections, render their data types, and make the data types editable to the user. Since the content of a section cannot be displayed to a user without an applicable widget, each section needs exactly one widget. Therefore, a single widget needs to support at least one data type.

Figure 3.2 shows how chapters, sections, and widgets are combined together. *Purpose and Responsibility*, *Version*, *Dependencies*, and *Persons in Charge* are names of the sections (cf. Figure 3.2, Item ❶). The section*Purpose and Responsibility* uses a *TextArea* widget to display the content (cf. Figure 3.2, Item ❷). *TextArea* widgets are meant to display long textual contents. The *Version* section uses instead a *TextField* widget (cf. Figure 3.2, Item ❸). *TextField* widgets are similar to *TextArea* widgets, but display the text in one line. In the case of the *Dependecies* section, the used widget is *RelationList* (cf. Figure 3.2, Item ❹). It is able to display links to other document parts and more. *RelationLists* will be explained later in detail.

#### Technical Details of Widgets

This part of the thesis will describe the technical details of the widgets using the example of *TextArea* and *TextField*. The internal data type consumed by these widgets is the same. It is the text data type. The only difference between these widgets is how they display it. In cases



2.1 Controller

Purpose and Responsibility

The Controller module is responsible for all manipulations on the models. This module contains two main controllers: App-Manager and DocManager. AppManager is responsible for controlling the application flow, functions, e.g., opening files, saving files, exporting, etc. The DocManager operates on Do-cument level. All changes of the Document, e.g., adding Chap-

Version

1.0

Dependencies

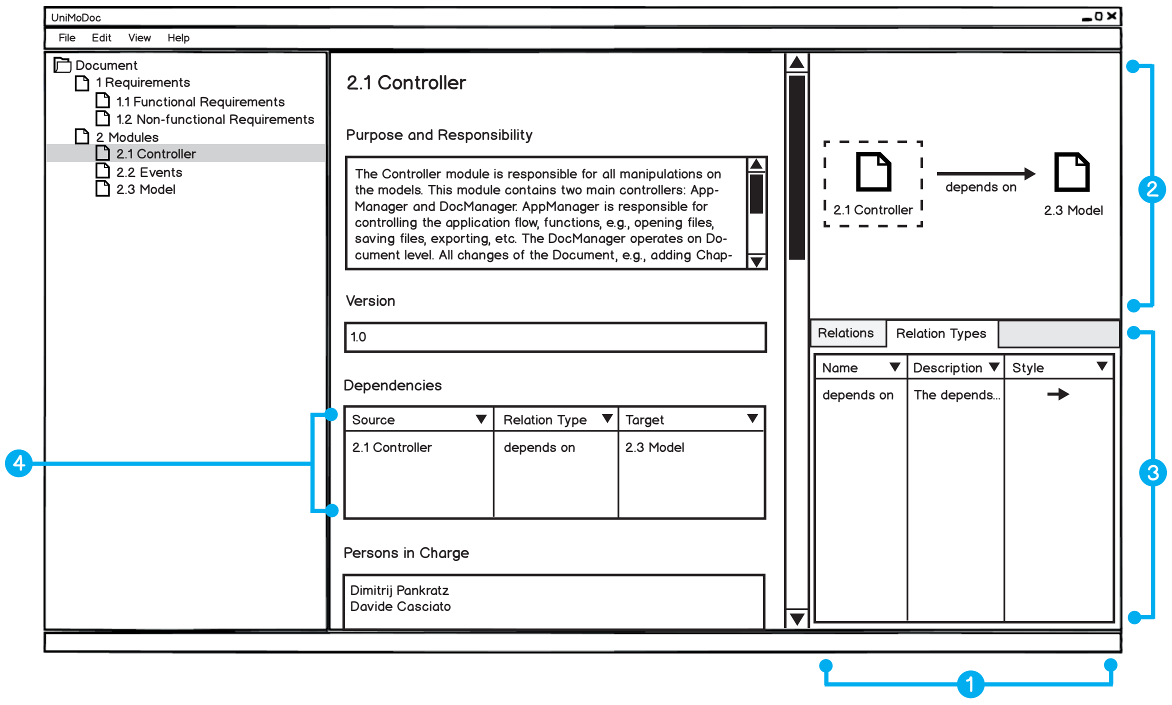
| Source | Relation Type | Target |
|---|---|---|
| 2.1 Controller | depends on | 2.3 Model |

Persons in Charge

Dimitrij Pankratz
Davide Casciato

**Figure 3.2:** Sections and Widgets in UniMoDoc
*Item ❶: Names of the Sections, Item ❷: TextArea Widget, Item ❸: TextField Widget, Item ❹: RelationList Widget*

where more than one widget is applicable to a specific data type, it is possible to change the widget of a section dynamically during the runtime without affecting the content.
The widgets and data types are meant to be easily extensible. In order to achieve this extensibility common interfaces will be provided. Due to this extensibility, it is necessary to consider that a file may use unknown widgets and data types. There are various possibilities how to handle this issue. This thesis suggests defining a default fall back widget, i.e., a *TextArea* widget. Due to the requirement that all generated files of the application are text-based, it is consequently possible to represent the content of all data types as text. In case no applicable widget is found for the data type, the content can be still represented as raw text in a *TextArea* widget.
In consequence of the requirement for various export formats widgets will need to render the content of a section not only in the GUI of the application but also for all export formats. If a widget does not support a specific export format, it can still return the content as text. This behavior is guaranteed by the common interface for widgets.
In addition, it is convenient to provide a configuration for widgets, which is unique for each widget instance. That way the user can configure each widget instance in a desired way, e.g., the maximum length of the text in a *TextField* widget or validation rules for the input. These configurations need to be considered in the common interface of the widgets.


UniMoDoc

File Edit View Help

- Document
  - 1 Requirements
    - 1.1 Functional Requirements
    - 1.2 Non-functional Requirements
  - 2 Modules
    - 2.1 Controller
    - 2.2 Events
    - 2.3 Model

2.1 Controller

Purpose and Responsibility

The Controller module is responsible for all manipulations on the models. This module contains two main controllers: App-Manager and DocManager. AppManager is responsible for controlling the application flow, functions, e.g., opening files, saving files, exporting, etc. The DocManager operates on Do-cument level. All changes of the Document, e.g., adding Chap-

Version

1.0

Dependencies

| Source | Relation Type | Target |
|---|---|---|
| 2.1 Controller | depends on | 2.3 Model |

Persons in Charge

Dimitrij Pankratz
Davide Casciato

| Name | Description | Style |
|---|---|---|
| depends on | The depends... | → |

**Figure 3.3:** Relation and Relation Types in UniMoDoc
*Item ❶: Central Relation Management Area, Item ❷: Visualization of Relations, Item ❸: Relation Tables, Item ❹: Relations in the RelationList Widget*

Relations and Relation Types

In the software architecture documentation relations can reveal, which modules realize a requirement, they define how the different views are connected, show the dependencies between modules, etc. In word processors these relations are often only implicit. There are textual references but the user cannot navigate along these relations.
Even if the user can navigate along these relations, the navigation is often only unidirectional, e.g., a module A depends on module B, this information is denoted in the description of module A but not in the description of module B. In this case, it is cumbersome to find out, which modules depend on module B from the perspective of module B.
In addition, the relations have always a specific semantics. In the example of module A, which depends on module B, the dependency can carry an additional semantic. It can be required, optional, or module A might realize the abstract module B, etc. If the relation is not explicit and formal, the semantic might be ambiguous and not obvious.
Word processors provide no central management or overview about these relations. The relations are spread across the document. It is not possible to find or filter the relations by their semantics.

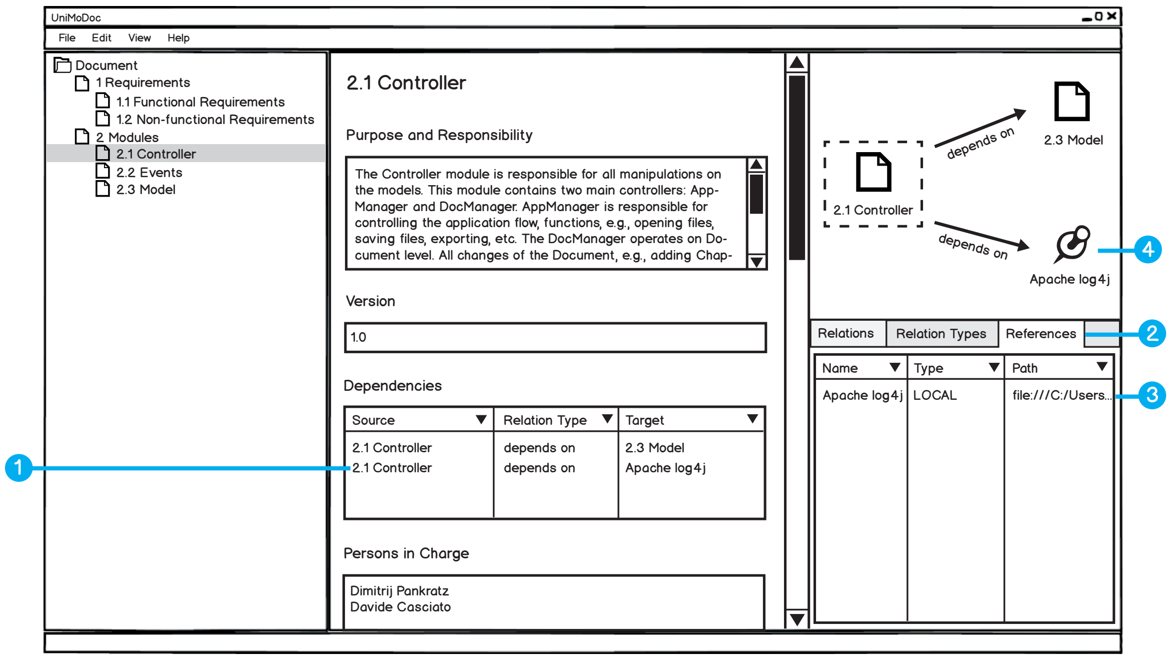
These issues can be solved with an explicit relation definition, relation types encapsulating the semantics, and a central management for the relations. Figure 3.3 puts it all together. It demonstrates how relations can be managed using the example of a *depends on* relation between the modules *2.1 Controller* and *2.3 Model*. The content area makes room for the central relation management (cf. Figure 3.3, Item ❶). This central relation management is separated vertically into the *Relation Visualization* (cf. Figure 3.3, Item ❷) and *Relation Tables* (cf. Figure 3.3, Item ❸). The *Relation Visualization* shows all inbound and outbound relations of the currently selected chapter to the user. The currently selected chapter in the visualization is highlighted with a dashed border around it. In Figure 3.3 it is *2.1 Controller*. The visualization shows here the outbound relation to module *2.3 Model* with the relation type *depends on*. If the user selects *2.3 Model* as current chapter, the visualization will show the same relation as inbound for the selected chapter *2.3 Model*.
The *Relation Tables* are displayed in tabs. There are two tabs in Figure 3.3, Item ❸: *Relations* and *Relation Types*. The *Relations* tab contains a tabular listing of all relations in the document. In addition, this and other tables can be filtered to show only the relations of the currently selected chapter or they can be filtered various other ways. The *Relation Types* tab contains a table of all relation types in the document. Thus it lists the semantics of relations and how they are displayed in *Relation Visualization*. Relations, relation types, and the visualization of them are discussed in [Casciato, 2013] in detail.

### Linking Relations

A central relation management is useful in order to overview and manage the relations of the complete document or parts of it. Still, in some cases it is convenient to view and manage relations directly from the content area of a chapter. If the document is exported in other formats like PDF or Hypertext Markup Language (HTML), e.g., all relations can be printed at the end of the document or all relations of a chapter at the end of the corresponding chapter. The user might want to place the relations or a selected group of relations in a specific section. In that case a *RelationList* widget can be used to do this. The *RelationList* widget links existing relations and lists them in a tabular view. Figure 3.3, Item ❹ illustrates how the *RelationList* widget is displayed. This approach allows defining views on existing relations comparable to the views concept in databases. In addition, it provides a quick access to context-sensitive relations directly from the widget. This means that the user can add, edit, delete, link, and unlink relations from the *RelationList* widget.

### External References

In order to satisfy the requirement for external references, it is not sufficient to reference only across the document parts. Modules may depend on external libraries. UML diagrams, source code artifacts, test cases, protocols, and other external resources need to be referenced somehow from the document. It is essential to provide relations between document parts and external references. Yet discussed relation endpoints are chapters and sections. They need to be extended by external references. In order to achieve this, it is necessary to represent



2.1 Controller

Purpose and Responsibility

The Controller module is responsible for all manipulations on the models. This module contains two main controllers: App-Manager and DocManager. AppManager is responsible for controlling the application flow, functions, e.g., opening files, saving files, exporting, etc. The DocManager operates on Do-cument level. All changes of the Document, e.g., adding Chap-

Version

1.0

Dependencies

| Source | Relation Type | Target |
|---|---|---|
| 2.1 Controller | depends on | 2.3 Model |
| 2.1 Controller | depends on | Apache log4j |

Persons in Charge

Dimitrij Pankratz
Davide Casciato

| Name | Type | Path |
|---|---|---|
| Apache log4j | LOCAL | file:///C:/Users... |

**Figure 3.4:** External References in UniMoDoc
*Item ❶: External Reference Entry in a RelationList Widget, Item ❷: Tab for External References, Item ❸: External Reference Entry in the Central Reference Management, Item ❹: External Reference in the Relation Visualization*

and manage the external references within a document. Figure 3.4 illustrates how this can be done.

In Figure 3.4, Item ❶ a new dependency from the *2.1 Controller* module is added to an external logging library *Apache log4j*. In order to manage the external references, a new tab *References* is added to the *Relation Tables* (cf. Figure 3.4, Item ❷). Figure 3.4, Item ❸ shows the external library listed in the tabular view of the external references. The table contains a *Type* column, which displays what type an external reference has. There are three possible types: *TEXT*, *Uniform Resource Locator (URL)*, and *LOCAL*. References of type *TEXT* are useful if no explicit location of the reference is available, e.g., an external module that is not build yet. References of *URL* type point at an absolute path in URL form. Finally, the references of *LOCAL* type are meant to point at files, which are located at the same machine as the document file. *LOCAL* references are handled in a special manner. They can be absolute or relative. Relative references are essential because the document should be under revision control with other files (external references). Thus absolute references cannot be used, since revision control is usually used by different people and machines. Still, relative references are sometimes very impractical. If the user moves the document to another location, all relative references become invalid. Therefore, the location of external references is stored

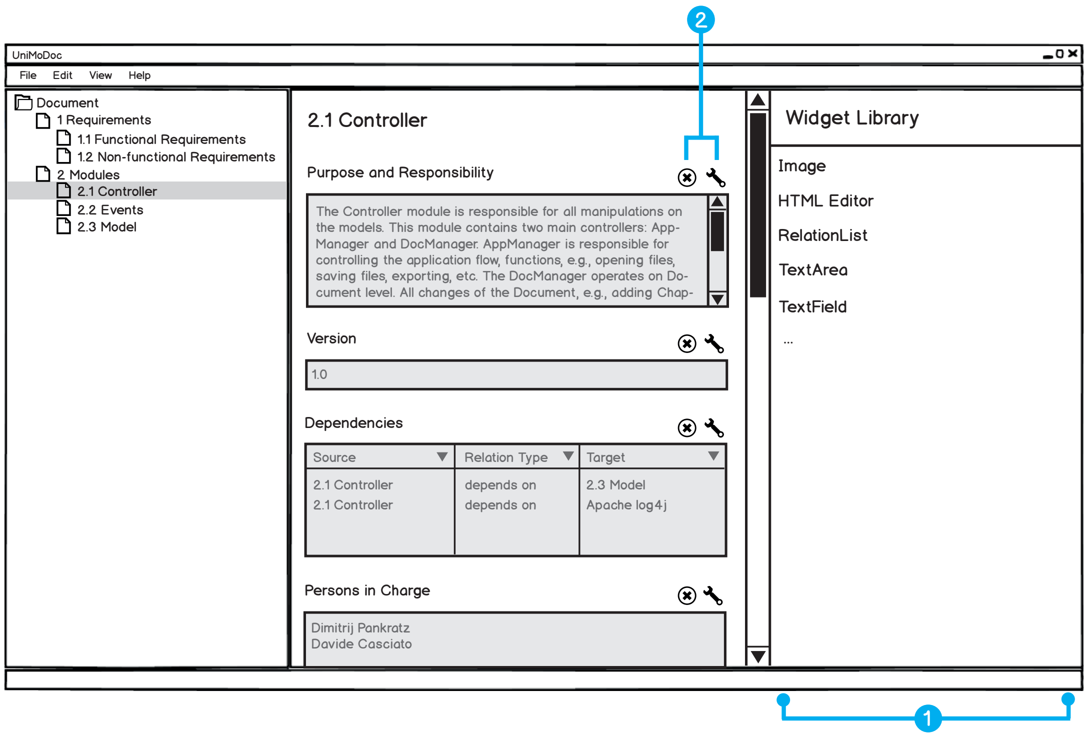
relative internally and in addition the last known absolute path of the document. This allows moving the document to another location on the same machine and the relative references can be found from the previous absolute location of the document.
Still, an issue can occur if the *LOCAL* references themselves are moved to another location. In this case, the application will check the validity of *LOCAL* references while opening a document. If a *LOCAL* reference cannot be found the user will be notified and asked to point at the new location of the reference.
Relations to external references are also displayed in the *Relation Visualization* (cf. Figure 3.4, Item ❹). External references are displayed differently from the internal parts of the document. The user is able to open the external references with the default application for the specific format on the current machine. External references are discussed in [Casciato, 2013] in detail.

#### Editing the Document and its Structure

There are different general approaches how UniMoDoc can be utilized. One use case is read-only. Thus the user is interested in the information within the documentation but does not edit it. Another use case is to edit the information without changing the structure of the document. The user fills in the forms, i.e., the information in the widgets. However, no additional chapters or sections are added. This is the case if someone defines the structure of the document in advance. At first glance this approach may appear uncommon but in fact it is convenient. The structure of a document is crucial for its comprehensibility and acceptance. People need time to define and agree on it. Once the structure is accepted, users need only to fill in the information in the given structure. Common structure for coherent document parts, e.g., requirements, modules, or tests improves the comprehensibility and completeness of the information.
Therefore, UniMoDoc differentiates between defining the structure of the document and editing the information. This difference is achieved by two edit modes for the user. The first and default mode is where the user can consume and fill in information. Thus users can edit data in the widgets and manage the relations. Adding, removing, or rearranging the sections is not allowed in this mode. The visual appearance of UniMoDoc corresponds to Figure 3.4.
In the second mode the user is able to add, remove and rearrange the sections. Editing the contents of widgets is deactivated in this mode. Instead of the central relation management the application displays the *Widget Library* on the right-hand side (cf. Figure 3.5, Item ❶). The *Widget Library* contains all known widgets in the application. The user can create new sections with drag and drop from the *Widget Library* to the content area. Above each section additional controls are shown to the user in order to configure or remove the section (cf. Figure 3.5, Item ❸).

#### Templates

The explicit separation between editing the content and the structure of the document is meant to promote a common structure across coherent document parts and even across different documents. Once a successful structure is found, it is wise to apply it to other documents.



Document
1 Requirements
1.1 Functional Requirements
1.2 Non-functional Requirements
2 Modules
2.1 Controller
2.2 Events
2.3 Model

2.1 Controller

Purpose and Responsibility

The Controller module is responsible for all manipulations on the models. This module contains two main controllers: App-Manager and DocManager. AppManager is responsible for controlling the application flow, functions, e.g., opening files, saving files, exporting, etc. The DocManager operates on Do-cument level. All changes of the Document, e.g., adding Chap-

Version

1.0

Dependencies

| Source | Relation Type | Target |
|---|---|---|
| 2.1 Controller | depends on | 2.3 Model |
| 2.1 Controller | depends on | Apache log4j |

Persons in Charge

Dimitrij Pankratz
Davide Casciato

Widget Library

Image
HTML Editor
RelationList
TextArea
TextField
...

**Figure 3.5:** Editing Document Structure in UniMoDoc
*Item ❶: Widget Library, Item ❷: Controls for Editing Sections*

This prevents the reinvention of the wheel and benefits the comprehensibility across documents. In addition, UniMoDoc supports templates. There are two fundamental template categories. The first one is a document template. In that case a complete document is saved as a template and can be used for creating new documents. The second one is a chapter template. Thus a chapter is saved to a template and can be included into documents. During the creation of a new chapter, the user can choose whether to create an empty chapter or a chapter from a template. That way it is possible to create templates of complete documents or single parts of it. Furthermore, UniMoDoc can save templates with contents. Thus the templates are not necessarily empty but they are filled with example content in order to give the user a better understanding of how to use them.

### 3.2.2 Data Model

This part of the document outlines an abstract data model for the described concept. The UML diagram in Figure 3.6 illustrates the general idea. Although the implementation of UniMoDoc is realized in Java, this data model can be easily applied to other object oriented programming languages. The data model is simplified and many details are faded out in order to provide a better overview.

A central entity in Figure 3.6 is the `Document`. It contains `Chapters`, `RelationTypes`, `Relations`, and `References`. The complete `Document` and all its sub parts have to be serializable into a text form in order to be saved. The `Template` entity extends the `Document` and adds additional information, e.g., a name and description.

In order to realize the `Relations`, their targets and sources need some kind of identification. Therefore, the `Identifiable` interface provides a unique id within a document. All entities, which need to participate in a relation or just to be uniquely identifiable, have to implement this interface. The `Identifiable` interface is implemented by `Chapters`, `Sections`, and `References`.
A `Relation` has exactly one source and one target of type `Identifiable`. In addition, `Relations` have a `RelationType`. The `RelationType` entity provides a specific semantic for `Relations`. It has a name and one can think of an additional description or style, which defines how `Relations` of this type are displayed.

Some elements of the `Document` need an order in the structure like `Chapters` and `Sections`. Therefore, they can implement the `Sortable` interface. It provides an explicit order within the structure of the `Document`. This order is not absolute. It is relative, i.e., the order is only valid within the parent of the corresponding element but not within a complete `Document`.

The `Chapters` are hierarchical. Thus a `Chapter` might have one parent `Chapter`. In case a `Chapter` has no parent the method `getParent` will return a null pointer. Consequently, in that case the parent is de facto the `Document`. In addition, a `Chapter` has zero or more children `Chapters`. The method `getChildren()` returns the children `Chapters` or an empty list. Furthermore, a `Chapter` contains a list of zero or more `Sections`.

The `Sections` have exactly one parent `Chapter`, which is accessible from the `Section` by the method `getParent()`. The content of a `Section` is of type `SectionData`. Basically, it is a wrapper for the raw data. The data itself is serialized to text and can be returned by the `getData()` method of the `SectionData` entity. At this point the data is generic and can be anything. The second information accessible from the `SectionData` by the method `getBeanId()` returns an identifier for the type of the raw data. That way the `Section` knows to which type the raw data can be deserialized from text. Still, the `Section` itself cannot deserialize the raw data. Instead, it needs an `IWidget` in order to handle the data.

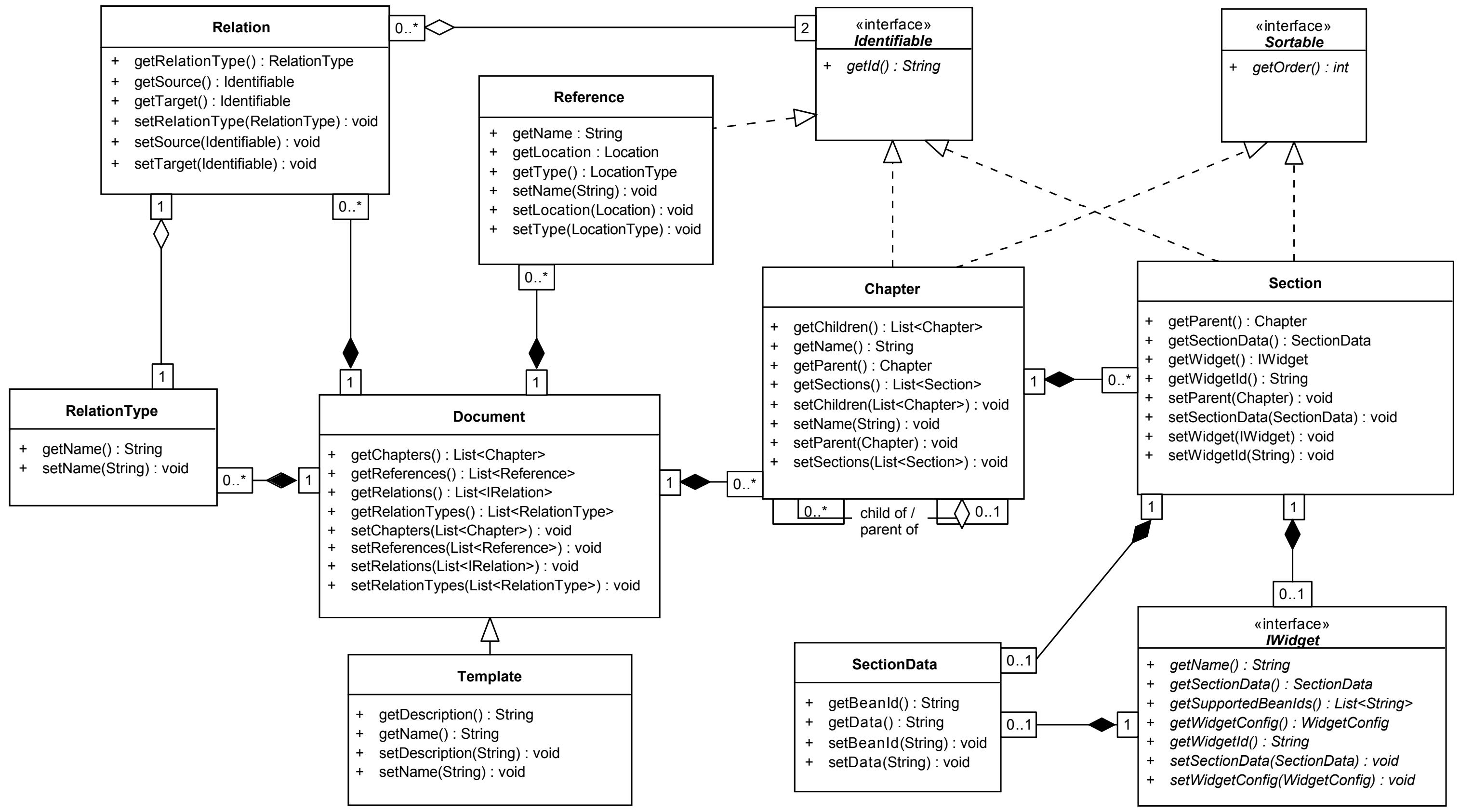

Relation
+ getRelationType() : RelationType
+ getSource() : Identifiable
+ getTarget() : Identifiable
+ setRelationType(RelationType) : void
+ setSource(Identifiable) : void
+ setTarget(Identifiable) : void
0..*
2
«interface»
Identifiable
+ getId() : String
«interface»
Sortable
+ getOrder() : int
Reference
+ getName : String
+ getLocation : Location
+ getType() : LocationType
+ setName(String) : void
+ setLocation(Location) : void
+ setType(LocationType) : void
1
0..*
0..*
1
1
1
RelationType
+ getName() : String
+ setName(String) : void
0..*
1
Document
+ getChapters() : List<Chapter>
+ getReferences() : List<Reference>
+ getRelations() : List<IRelation>
+ getRelationTypes() : List<RelationType>
+ setChapters(List<Chapter>) : void
+ setReferences(List<Reference>) : void
+ setRelations(List<IRelation>) : void
+ setRelationTypes(List<RelationType>) : void
1
0..*
Chapter
+ getChildren() : List<Chapter>
+ getName() : String
+ getParent() : Chapter
+ getSections() : List<Section>
+ setChildren(List<Chapter>) : void
+ setName(String) : void
+ setParent(Chapter) : void
+ setSections(List<Section>) : void
0..*
child of /
parent of
0..1
1
0..*
Section
+ getParent() : Chapter
+ getSectionData() : SectionData
+ getWidget() : IWidget
+ getWidgetId() : String
+ setParent(Chapter) : void
+ setSectionData(SectionData) : void
+ setWidget(IWidget) : void
+ setWidgetId(String) : void
1
1
0..1
Template
+ getDescription() : String
+ getName() : String
+ setDescription(String) : void
+ setName(String) : void
SectionData
+ getBeanId() : String
+ getData() : String
+ setBeanId(String) : void
+ setData(String) : void
0..1
0..1
1
«interface»
IWidget
+ getName() : String
+ getSectionData() : SectionData
+ getSupportedBeanIds() : List<String>
+ getWidgetConfig() : WidgetConfig
+ getWidgetId() : String
+ setSectionData(SectionData) : void
+ setWidgetConfig(WidgetConfig) : void

**Figure 3.6:** Simplified Data Model in UniMoDoc

An `IWidget` supports one or more different Beans, i.e., deserialized data types of `Sections`. Furthermore, `IWidgets` know how to deserialize the raw data from text to actual Beans. The method `getSupportedBeanIds()` from the `IWidgets` returns a list of Bean identifiers, which are supported by the corresponding `IWidget`. This method can be used by a `Section` in order to find an applicable `IWidget`. This situation can occur if the assigned `IWidget` of a `Section` cannot be resolved, i.e., not found in the internal registry of the application. In addition, the user can change the `IWidget` of a `Section` during the run-time. In that case the `getSupportedBeanIds()` method is also used for showing the user a list of applicable `IWidgets`.
By default, when a new `Section` is created an `IWidget` is assigned to it. The `Section` passes then the `SectionData` to the assigned `IWidget`. The `IWidget` deserializes its data into a Bean, displays it to the user and makes it editable. Before the `Document` is saved the `IWidget` serializes the Bean as text, packs it back into `SectionData`, and returns the `SectionData` back to the `Section`.

### 3.2.3 Module Metadata

In order to satisfy the requirement for a module documentation template, this chapter describes possible metadata for it. For each metadata entry there is description why it is important, what information to fill in, and which templates use these metadata already. The metadata are sorted alphabetically. In order to use it in a concrete template, it is necessary to order and group it in a favorable way. An example grouping is suggested in Chapter 4.

**Change History**
: The module description should be up to date. Thus the changes from the past and the evolution of the documentation are not obvious. The software architecture documentation should be in a revision control. Still, relevant changes for a specific module might be hard to track down from the revision control. In such cases a change history in the module description is useful.
[Starke and Hruschka, 2011] proposes to include change history in module description if it is not part of another document.

**Dependencies**
: This information is a list of dependencies to other modules. These modules might not be part of the system, i.e., external modules. Dependencies are usually required in order to compile or execute the module. Still, there can be optional dependencies, which are not essential for a module. As an example, OSGi supports both dependency types.
Dependencies are included in the module description of [J-PaD, 2013] and [Starke and Hruschka, 2011].

**Design**
: These metadata describe the design of a module. This is similar to the Module view of software architecture documentation from Chapter 2.2. The possible notations are textual, graphical, or a combination of both.

The design description is a fundamental part of software architecture documentation and can be found in Resources, [Starke and Hruschka, 2011], and [Garlan et al., 2010].

**Design Decisions**

This part describes the reasons for the chosen design approach. Since there are myriads of possible solutions for a single issue, it is necessary not only to provide the solution but to argue why this is the solution.
The argumentation for a specific design decision is required in [Starke and Hruschka, 2011], and [Garlan et al., 2010].

**Interfaces**

These metadata describe the interfaces of a module, their purpose and visibility. Single interfaces might be referenced to the corresponding source code artifacts. They can be divided into inbound and outbound interfaces.
The interfaces are an essential part of modules according to [Parnas, 1972] and their description can be also found in [J-PaD, 2013], [Starke and Hruschka, 2011], and [Garlan et al., 2010].

**Implementation Constraints**

This part describes information that should be taken into account during implementation. Architects will usually intend a specific implementation strategy during the design of a module, which may be not obvious from the design itself. Therefore, they can advise the implementers to follow specific implementation constraints.
The implementation constraints are part of the module documentation in [Garlan et al., 2010].

**Name**

This information denotes the name of the module. This property is probably most typical to all module constructs, but the sense can slightly vary. Ideally the name is meaningful and implies the functionality or role in the whole system. Moreover, the name can indicate the position of a module in a hierarchy, e.g., Java packages `java.awt.event`, where `event` is a sub package of `awt`. This thesis does not require unique names of modules because different module constructs in practice may have different properties to ensure uniqueness, e.g., name and symbolic-name in OSGi.

**Management Information**

In some cases, it might be useful to have information for the project management on the module level. These metadata can contain the schedule plan, budget information, or a risk estimate for the module.
[Garlan et al., 2010] proposes these metadata as part of the module description.

**Other Artifacts**

The source code artifacts contain the implementation artifacts. Still, there can be other files, which are not part of implementation or test. These files can be images, sounds, videos, documents, etc.
In [J-PaD, 2013] similar information is called *Resources*.

**Persons in Charge**

This is a list of persons who are responsible for this module. In a team of architects this list can reference the contact persons for a particular module.
[J-PaD, 2013] supports a list of authors in the module documentation. This thesis changes the meaning slightly to the responsible persons. For the architecture documentation it is more important to know who is responsible for the complete module than the authors of the source code artifacts. Furthermore, the authors are usually already noted in the documentation or comments of the single implementation files.

**Purpose and Responsibility**

The name can provide a first impression of the function and role in the whole system. Still, it is necessary to provide this information in detail. These data describe why this module is part of the system, what is its functionality and what role does it have from the perspective of other modules.
This information is part of the module description in [J-PaD, 2013], [Garlan et al., 2010], and [Starke and Hruschka, 2011].

**Rejected Design Alternatives**

This data describes considered and rejected designs as well as the reasons for their rejection. Such information is useful to understand the design and its reasons. Furthermore it shows the wrong directions of thought for future design changes and improvements.
This module description information is proposed by [Starke and Hruschka, 2011].

**Requirements**

This part lists the references to the covered requirements of a module. This information is required in order to track down why a module has a specific functionality (traceability) or to ensure that all requirements are covered by the modules.
Covered requirements are part of the module description in [J-PaD, 2013] and [Starke and Hruschka, 2011].

**Source Code Artifacts**

These metadata reference the implementation artifacts of a module. The module documentation in the software architecture documentation provides a high-level overview. This information is mainly publicly visible, provides a big picture and serves as a starting point. Implementation details can be found on the source code level. These references link both document types and make the documentation complete.
Source code artifacts are suggested and used in the module description of [J-PaD, 2013], [Garlan et al., 2010], and [Starke and Hruschka, 2011].

**Test Information**

This documentation part contains test plans, test cases, test data, and other relevant information for testing of a module. It can help the test manager to plan and build the tests or provide an overview of quality assurance on the module level to other stakeholders. [Casciato, 2013] researches this topic in depth and splits module's test documentation in further metadata.
The test information metadata on module level can be found in [J-PaD, 2013], [Garlan et al., 2010], and [Starke and Hruschka, 2011].

**Variability**

Variability describes which possible changes are planned for a module and how it can be configured or adjusted. This is an explicit support of the quality attribute flexibility. These metadata are suggested for module description in [J-PaD, 2013] and [Starke and Hruschka, 2011].

**Version**

The current version number of the module. It is useful to indicate possible changes in the past and to reference a version in the change history.
The version of a module is part of module documentation in [J-PaD, 2013].

# 4 Implementation and Results

The first part of this chapter provides background information on design decisions that were made and describes the details about the implementation. The second part of this chapter deals with the outcome of the implementation. It presents the results and compares them with the design.

## 4.1 Implementation

The overall presented design in this thesis is divided into two parts. The first part from Chapter 3.2 is almost technology independent. It describes the suggested solution in a very abstract manner. That way the concept can be easily transferred to other technologies or areas of application. The second part of the design is a matter of this chapter. It puts the abstract concept in a technological context and argues about the design decisions made of this level. Furthermore, it serves as a proof of concept for the abstract concept and describes the idiosyncrasies of the used technologies.

### 4.1.1 Data Format

The requirements in Chapter 3.1 define a text-based and line-based data format for all data created by the application. Thus especially the data model needs to be serialized as text. The data model has a hierarchical structure, e.g., the document contains chapters, chapters contains sections, etc. Therefore, it is convenient to use a data format that supports this kind of structure. Furthermore, the used programming language and technology is Java. For this reason, it is practical to use a data format supported by Java in order to save time and keep the effort low. Both is true for XML. It supports hierarchically structured data and Java can handle it natively.

One possible alternative to XML is the Comma-separated values (CSV) data format. In CSV the data is divided into columns separated by a special delimiter character (typically the semicolon). In other words, the data is saved in one big table. Therefore, CSV does not directly support a hierarchical structure like the data model of UniMoDoc. Additionally, CSV is not standardized like XML and not explicitly supported by Java.
Another possible alternative is JavaScript Object Notation (JSON). Comparably to XML, it is well applicable to hierarchically structured data and it is standardized. One big disadvantage is a lack of support in the default libraries of Java.

XML Approaches in Java

There are different approaches how to handle XML in Java. They can be categorized in three following parts.

- Document Object Model (DOM) oriented APIs represent the complete XML structure as a DOM tree in the memory.
- Pull and Push APIs do not load the complete XML structure into memory. During the reading the Pull API parses the document in order to find specific elements. That way it pulls the required information from the document. The Push API calls appropriate methods during the reading each time an element occurs. In other words, it pushes the data from the document compared to events.
- Mapping APIs map the XML structure directly to the internal data model and vice versa.

Each approach has its benefits and drawbacks. The DOM-oriented APIs are well applicable for modification of the complete document. Since the complete document is loaded into the memory, these APIs are memory consuming. The DOM structure is abstract, so it can be still necessary to map the information into an internal data model.
The Pull and Push APIs are especially useful for searching specific elements in the XML structure. They are potentially faster than the other APIs because they do not load the complete document into memory. Still, it is cumbersome to transform the complete XML structure into the internal data model.
The Mapping APIs save the effort of transforming the XML structure into the internal data model. They are useful in cases where the internal data model can be adjusted to fit the XML structure or vice versa. If the internal data model is different from the XML structure, the main benefit is lost.

There are no special requirements for the XML structure and it can be adjusted in order to fit the internal data model. Additionally, taking into account the enormous time saving of out of the box mapping, the winner was the Mapping API approach.
For the purpose of avoiding unnecessary dependencies to custom libraries a default Mapping API in Java was chosen, namely Java Architecture for XML Binding (JAXB).

JAXB

JAXB makes it possible to bind Java objects to an XML structure. This Java object can be marshaled to an XML file or an XML file unmarshaled back to a Java object. The binding is controlled by Java annotations. Listing 4.1 shows an extract from the data model of UniMoDoc and how it can be marshaled using JAXB. The listing contains a simplified definition of two classes `Document` and `Chapter`. In the data model of UniMoDoc (see also Chapter 3.2.2) the `Document` class contains all the other elements, e.g., `Chapters`, `Sections`, `Relations`, etc. Consequentially, it is a root element in the XML structure. This fact is expressed by the

**Listing 4.1** Example JAXB Usage

```java
@XmlRootElement
public class Document {
    private List<Chapter> chapters;

    public List<Chapter> getChapters() {
        return chapters;
    }

    @XmlElementWrapper
    @XmlElement(name = "chapter")
    public void setChapters(List<Chapter> chapters) {
        this.chapters = chapters;
    }
}

public class Chapter {
    private String name;

    public String getName() {
        return name;
    }

    @XmlElement
    public void setName(String name) {
        this.name = name;
    }
}


Chapter chapter = new Chapter();
chapter.setName("Functional Requirements");
List<Chapter> chapters = new ArrayList<Chapter>();
chapters.add(chapter);
Document document = new Document();
document.setChapters(chapters);

JAXBContext context = JAXBContext.newInstance(Document.class);
Marshaller m = context.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(document, System.out);
```

**Listing 4.2** XML Output from Listing 4.1

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<document>
   <chapters>
      <chapter>
         <name>Functional Requirements</name>
      </chapter>
   </chapters>
</document>
```

`@XmlRootElement` annotation.
Furthermore, in Listing 4.1 the `Document` has a list of `Chapters`. In order to include this list in the XML structure the setter method of the list is annotated with `@XmlElement`. The additional setting `@XmlElement(name = "chapter")` for the annotation changes the name of the XML element from default attribute name `chapters` to `chapter`. Collections are typically marshaled by JAXB, so that the single, repeated entries have no wrapper element. This behavior can be changed by the `@XmlElementWrapper` annotation. In Listing 4.1 it will add a `chapters` element around all `chapter` element entries.
The `Chapter` class has a name attribute that needs to be saved in an XML file. Therefore, its setter method is annotated with `@XmlElement`.

Beginning from line 30 to 35 in Listing 4.1, an object structure is created from the described classes in order to serialize it to an XML file. Thus a new `Chapter` is constructed with a name *Functional Requirements* and added to a new list. Then a new `Document` is created and the previously constructed list is added to it.
Lines 37 to 40 in Listing 4.1 demonstrate how the objects from above can be marshaled with JAXB. First of all, a `JAXBContext` is created that knows, which classes to use for marshaling or unmarshaling. Then a new `Marshaller` object is constructed from the context. The `JAXB_FORMATTED_OUTPUT` option instructs JAXB to add indentation and linefeeds to the outputted XML file, if it is set to `true`. Finally, the last line in Listing 4.1 passes the previously created data model to the `Marshaller` and instructs it to return the result on the default output stream of the `System`. The created XML document is shown in Listing 4.2.

#### JAXB Issues

JAXB is an enormous time savior when it comes to mapping of information between the data model and the XML structure. Still, it has some significant drawbacks. During the reading of an XML structure all elements have to be known to the data model. If an XML element occurs that cannot be mapped on a corresponding class or attribute, the complete mapping fails. UniMoDoc has to be extensible so that new widgets and section data types can be easily added like plugins. Using one section data type in a document would make it unreadable to other applications without that section data type.

**Listing 4.3** Embedded XML Structure

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<document>
    <chapters>
        <chapter>
            <name>Functional Requirements</name>
            <sections>
                <section>
                    <name>Description</name>
                    <widgetId>de.unimodoc.widgets.TextAreaWidget</widgetId>
                    <sectionData>
                        <beanId>de.unimodoc.model.section.StringBean</beanId>
                        <XMLData>&lt;?xml version=&quot;1.0&quot;
                            encoding=&quot;UTF-8&quot;
                            standalone=&quot;yes&quot;?&gt;
&lt;stringBean&gt;
    &lt;value&gt;This part describes the required functions of the
        software.&lt;/value&gt;
&lt;/stringBean&gt;
                        </XMLData>
                    </sectionData>
                </section>
            </sections>
        </chapter>
    </chapters>
</document>
```

In order to solve this issue, UniMoDoc uses a two level mapping approach. The first level consists of the core document structure that should not be extended by plugins. The second level of mapping is dynamic and provides extension points for plugins. In other words, the first level is the main XML document and the second level consists of other XML documents, which are embedded in the first one.

Listing 4.3 shows a simplified extract from an XML file generated by UniMoDoc. It demonstrates both levels. In order to read this document, firstly UniMoDoc will map the complete document with JAXB to the internal first level data model. In that case it will create among others an object of type `SectionData`. This object will have two `String` attributes: `beanId` and `XMLData`. As shown in Listing 4.3 `XMLData` contains another encoded XML document. This is a second level document. At this point it is only interpreted as a text. The `beanId` contains a fully qualified name of a class to which the content of `XMLData` can be mapped. UniMoDoc will check if the class from `beanId` is known within the current application. If this class is known, it will be instantiated and the content from `XMLData` will be mapped to it using JAXB.
In case the class from `beanId` is not available, the raw data from `XMLData` can be displayed

in a default widget as a `String` to the user. Still, the most important aspect in this case is the fact that JAXB does not fail to read the complete document because of single unknown elements.

### 4.1.2 Application Flow

Most parts of the application stick to a variant of the Model–View–Controller (MVC) pattern. The MVC pattern differentiates between three types of components:

- The model encapsulates the data of the application.
- The view observes the changes in the model and displays the data to the user.
- The controller manipulates the data.

#### Model

The design of the model in UniMoDoc is described in detail in Chapter 3.2.2. The implementation of the data model does not essentially differ from the design. Still, the model has some differences to the default MVC pattern. In UniMoDoc the model is completely passive. In the default MVC pattern the view observes the model directly. Thus if the model changes, it notifies the view about the changes. In UniMoDoc the controller manipulates the model exclusively and notifies also the view about changes in the model.

#### View

All view components used in UniMoDoc come from Java Swing or are based on it. Swing is a default GUI API in Java. The use of Swing components results in another slight variance from the default MVC pattern. Some Swing components like `JTable` or `JTree` come with an own model implementation out of the box. Therefore, in some cases it is reasonable to use these out of the box models in addition instead of changing the own data model in order to fit the Swing components. So UniMoDoc uses these models as mediators between the main data model and Swing components.

#### Controller

There are two main controller classes inUniMoDoc. The first one is called `AppManager` and operates on the application level. It is responsible for functions, e.g., opening, saving, closing, exporting the documents or templates. The second one is called `DocManager` and it is responsible for one document only. The `DocManager` is a central point for modifying a document, e.g., adding, editing, deleting chapters, sections, relations, etc.

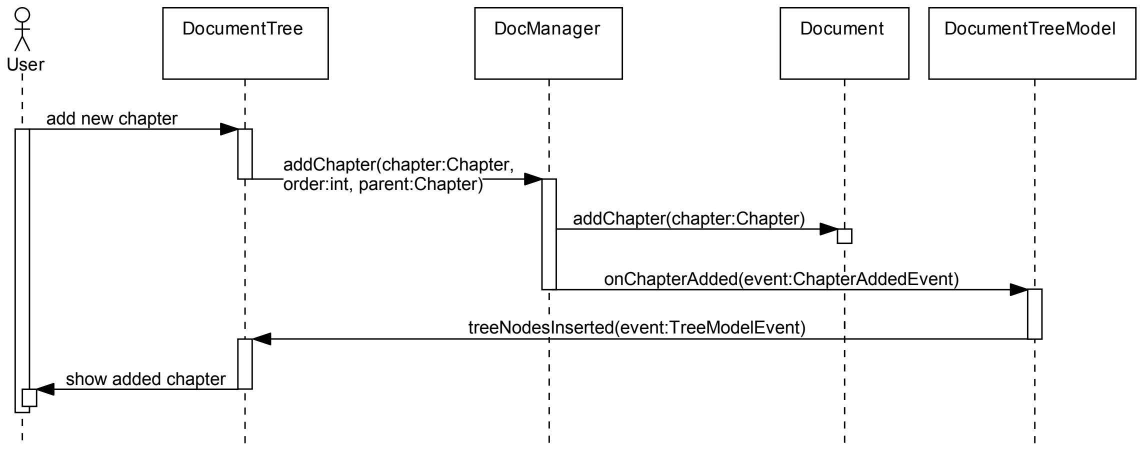

User
DocumentTree
DocManager
Document
DocumentTreeModel
add new chapter
addChapter(chapter:Chapter, order:int, parent:Chapter)
addChapter(chapter:Chapter)
onChapterAdded(event:ChapterAddedEvent)
treeNodesInserted(event:TreeModelEvent)
show added chapter

**Figure 4.1:** UML Sequence Diagram of Adding New Chapters

Figure 4.1 shows a UML sequence diagram demonstrating how a new chapter is added to the document. First, the user adds a new chapter using the graphical component `DocumentTree`. `DocumentTree` will invoke an action that calls the appropriate controller for modifying the corresponding `Document`. In other words, it calls the `addChapter(chapter:Chapter, order:int, parent:Chapter)` method on the `DocManager`. The `DocManager` is responsible to check if this modification on the `Document` is permitted and valid. Then, the `DocManager` adds the given chapter to the `Document` entity. Since the data model is completely passive, the `DocManager` has to notify all observers about the modification. In case of the `DocumentTree` a Swing specific model is implemented called `DocumentTreeModel`. The `DocumentTreeModel` is registered in the `DocManager` as an observer for the `Document`. For that reason the `DocManager` notifies the `DocumentTreeModel` about a new chapter with a suitable event. That way the `DocumentTreeModel` can adjust its internal structure to the change. The `DocumentTree` in turn observes the `DocumentTreeModel`. Therefore, after adjusting the internal structure `DocumentTreeModel` calls the `treeNodesInserted(event:TreeModelEvent)` method on the `DocumentTree`. Finally, the `DocumentTree` updates its view and shows the added chapter to the user.

### Events

The controllers notify the observers about modifications on the data model with corresponding events. Therefore, UniMoDoc provides various events and an appropriate interface for the event listeners. There are basically three types of events. These event types are for added, updated, and deleted objects. Java provides out of the box functionality for handling such events, e.g., `java.beans.PropertyChangeSupport`. UniMoDoc uses an own implementation instead, which improves some shortcomings of the default implementations. Especially the update events are cumbersome to handle in the default implementations. It is necessary to provide the name of each changed attribute, its old value and its new value. This approach results in an error-prone boilerplate code. In UniMoDoc update events accept the old and new objects. Possible changes on the attribute level are detected via the Java Reflection API. This approach avoids the boilerplate code and makes the application easier extensible.

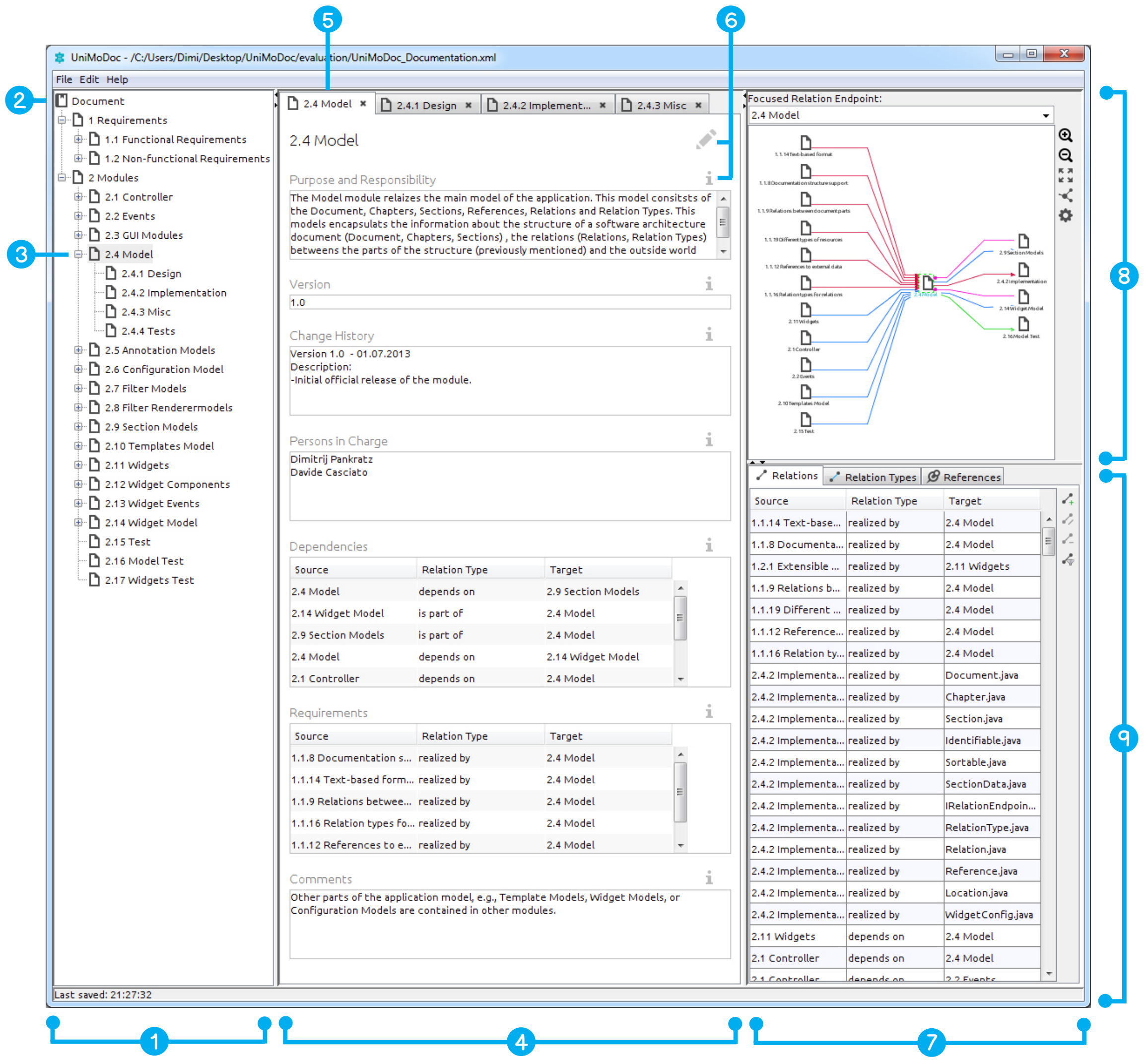
### 4.1.3 Id Generation and Modification

Chapter 3.2.2 describes the data model and the `Identifiable` interface. It is necessary to uniquely identify some elements in the document. Furthermore, unique identification of elements is essential for the relations. Such elements need to implement the `Identifiable` interface, which guarantee a unique id within a document. Various approaches are possible to provide this id.
A naive approach is to use a simple counter as an id. The drawback of this approach is that it needs a central id generation. Additionally, the id generation is dependent on the current counter in the document.
Another approach consists of using a Universally Unique Identifier (UUID). The UUID is a 128-bit number standardized by Open Software Foundation (OSF). It is specially designed for distributed systems in order to provide a non-centric id generation. The concrete implementation of the UUID generation may vary on different JVMs. The drawback of this approach is the possibility of collisions. Although, the possibility of a collision for a UUID is very small, it needs to be considered.
Due to the benefits of a non-centric id generation UniMoDoc uses Java's UUIDs for identifying and referencing the elements in a document.

Normally an id should not change for an element after its assignment. However, in some cases it is unavoidable. When saving a template, two different cases can appear. In the first case, a complete document is saved as a template. This case will not need any further id modification. In the second case, only a part of the document is saved as a template. Thus it is necessary to cut out all the elements hanging on this part, e.g., chapters, sections, relations between the cut out chapters and/or references, attached references and relation types. When these elements are saved as a template and then are imported in an existing document, new ids should be generated. Since the document where the template is imported, may be the one from where it is generated.
It is not trivial to change all identifiers because they may be referenced by other elements like relations or sections (in case of sections with relation widgets). For that reason an algorithm is implemented, which firstly changes the identifiers of the relation types, references, and relations. The old and new identifiers of the relations are saved in a table. Then, the identifiers of the chapters and sections are changed. Their identifiers are also saved in a table. In the next step this table is processed by the relations in order to change the old references ids to the new ones. Finally, in the last step the sections with relation widgets get the tables with old and new identifiers of the relations, chapters and sections. That way before a template is imported in an existing document, all identifiers are changed in this template remaining its internal consistency.

## 4.2 Results

This chapter describes the outcomes of the implementation and shows screenshots of the resulting application. First, a short overview outlines the basic structure of the implemented application. Additionally, the overview compares the design with the actual results. Second, the implemented module template is presented.



Document
- 1 Requirements
  - 1.1 Functional Requirements
  - 1.2 Non-functional Requirements
- 2 Modules
  - 2.1 Controller
  - 2.2 Events
  - 2.3 GUI Modules
  - 2.4 Model
    - 2.4.1 Design
    - 2.4.2 Implementation
    - 2.4.3 Misc
    - 2.4.4 Tests
  - 2.5 Annotation Models
  - 2.6 Configuration Model
  - 2.7 Filter Models
  - 2.8 Filter Renderermodels
  - 2.9 Section Models
  - 2.10 Templates Model
  - 2.11 Widgets
  - 2.12 Widget Components
  - 2.13 Widget Events
  - 2.14 Widget Model
  - 2.15 Test
  - 2.16 Model Test
  - 2.17 Widgets Test

2.4 Model

Purpose and Responsibility

The Model module relaizes the main model of the application. This model consitsts of the Document, Chapters, Sections, References, Relations and Relation Types. This models encapsulats the information about the structure of a software architecture document (Document, Chapters, Sections) , the relations (Relations, Relation Types) betweens the parts of the structure (previously mentioned) and the outside world

Version

1.0

Change History

Version 1.0 - 01.07.2013
Description:
-Initial official release of the module.

Persons in Charge

Dimitrij Pankratz
Davide Casciato

Dependencies

| Source | Relation Type | Target |
|---|---|---|
| 2.4 Model | depends on | 2.9 Section Models |
| 2.14 Widget Model | is part of | 2.4 Model |
| 2.9 Section Models | is part of | 2.4 Model |
| 2.4 Model | depends on | 2.14 Widget Model |
| 2.1 Controller | depends on | 2.4 Model |

Requirements

| Source | Relation Type | Target |
|---|---|---|
| 1.1.8 Documentation s... | realized by | 2.4 Model |
| 1.1.14 Text-based form... | realized by | 2.4 Model |
| 1.1.9 Relations betwee... | realized by | 2.4 Model |
| 1.1.16 Relation types fo... | realized by | 2.4 Model |
| 1.1.12 References to e... | realized by | 2.4 Model |

Comments

Other parts of the application model, e.g., Template Models, Widget Models, or Configuration Models are contained in other modules.

Focused Relation Endpoint:
2.4 Model

| Source | Relation Type | Target |
|---|---|---|
| 1.1.14 Text-base... | realized by | 2.4 Model |
| 1.1.8 Documenta... | realized by | 2.4 Model |
| 1.2.1 Extensible ... | realized by | 2.11 Widgets |
| 1.1.9 Relations b... | realized by | 2.4 Model |
| 1.1.19 Different ... | realized by | 2.4 Model |
| 1.1.12 Reference... | realized by | 2.4 Model |
| 1.1.16 Relation ty... | realized by | 2.4 Model |
| 2.4.2 Implementa... | realized by | Document.java |
| 2.4.2 Implementa... | realized by | Chapter.java |
| 2.4.2 Implementa... | realized by | Section.java |
| 2.4.2 Implementa... | realized by | Identifiable.java |
| 2.4.2 Implementa... | realized by | Sortable.java |
| 2.4.2 Implementa... | realized by | SectionData.java |
| 2.4.2 Implementa... | realized by | IRelationEndpoin... |
| 2.4.2 Implementa... | realized by | RelationType.java |
| 2.4.2 Implementa... | realized by | Relation.java |
| 2.4.2 Implementa... | realized by | Reference.java |
| 2.4.2 Implementa... | realized by | Location.java |
| 2.4.2 Implementa... | realized by | WidgetConfig.java |
| 2.11 Widgets | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.2 Events |

**Figure 4.2:** UniMoDoc GUI Overview
*Item ❶: Document Structure, Item ❷: Root Element, Item ❸: Selected Chapter, Item ❹: Content Area, Item ❺: Selected Tab, Item ❻: Additional Controls, Item ❼: Central Relation Management, Item ❽: Visualization of Relations, Item ❾: Relation Tables*

### 4.2.1 Overview

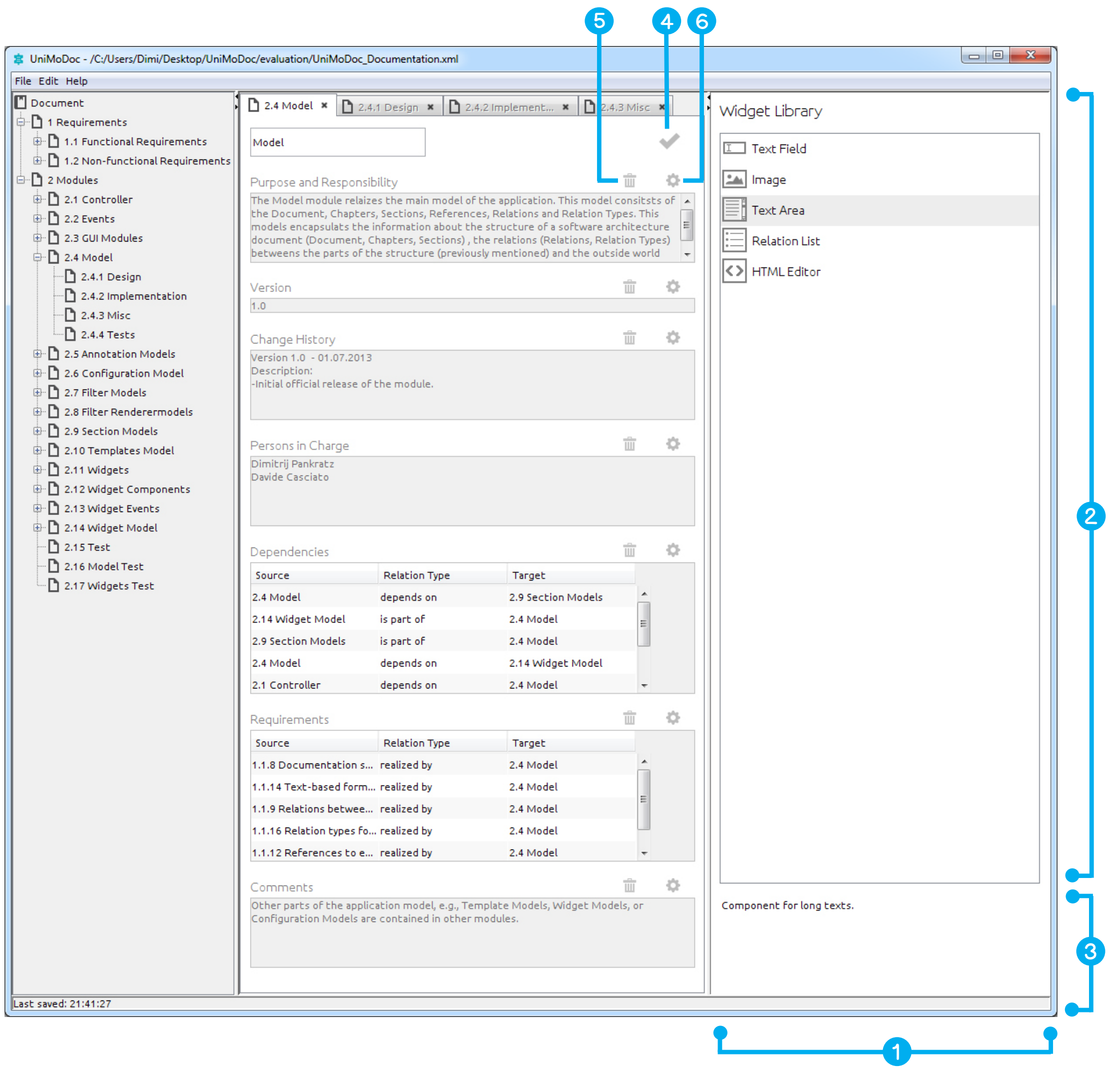
An overview of the implemented application is shown in Figure 4.2. Figure 4.2, Item ❶ highlights the structure of the opened document. It provides the primary way for navigating through the document. Figure 4.2, Item ❷ points on the root of the tree structure, which is the document itself. Figure 4.2, Item ❸ points on the currently selected and opened part of the document, i.e., *Chapter 2.4 Model*.

The content of this chapter is displayed in the content area of the application, which is outlined in Figure 4.2, Item ❹. Additionally, the content area is organized by tabs. Figure 4.2, Item ❺ shows the currently selected tab of the *Chapter 2.4 Model*. Previously selected tabs, are shown on the right hand-side next to the currently selected tab. The content of the selected chapter contains its name and a list of sections. Each section displays a name and a widget. All in all the content area is very similarly structured compared to the design in Chapter 3.2.1. However, some details differ from the design. Figure 4.2, Item ❻ shows to additional controls. The pencil button allows the user to edit the structure of the chapter, i.e., to add, modify, or delete the sections. The information button above every section shows a hint to the corresponding section. This should provide the user a general idea of what information to fill in.

Figure 4.2, Item ❼ illustrates the central relation management of the application. It is vertically divided into two parts. The top part visualizes the relations of the currently selected chapter in a graph, which is outlined in Figure 4.2, Item ❽. It provides an alternate navigation through the document. The user is able to see, which other document parts or external references are related with the currently selected chapter and directly navigate to them by a double click. The relations are directed and displayed with an arrow in order to show the direction. The semantics of the relations are defined by the used relation types. Different relation types can be differently visualized, e.g., by using colors, line or arrow styles.
The visualization of the relations in a graph is useful for displaying a small number of relations. However, large numbers of relations result in a vast and confusing graph. Therefore, the relations, relation types, and external references are additionally displayed in a tabular view (cf. Figure 4.2, Item ❾). The tabular view does not necessarily display only the relations of the currently selected chapter. In Figure 4.2 it displays all relations of the document. The user can even configure the displayed information of all three tables by filters. The filters provide logical functions and are extensible. Additional information about the visualization and management of the relations in UniMoDoc is described in [Casciato, 2013].

#### Editing Document Structure

When the user clicks on the pencil button from Figure 4.2, Item ❻, another view is presented. This view is illustrated in Figure 4.3. The obvious difference is that the central relation management is hidden. Instead, the *Widget Library* is shown (cf. Figure 4.3, Item ❶). The *Widget Library* consists of two vertically separated parts. The top part shows all available widgets of the application (cf. Figure 4.3, Item ❷). The user can select a widget from the *Widget Library* and create a new section with it via drag and drop in the currently opened chapter. The bottom part of the *Widget Library* in Figure 4.3, Item ❸ shows the description for the selected widget.



UniMoDoc - /C:/Users/Dimi/Desktop/UniMoDoc/evaluation/UniMoDoc_Documentation.xml

File Edit Help

- Document
  - 1 Requirements
    - 1.1 Functional Requirements
    - 1.2 Non-functional Requirements
  - 2 Modules
    - 2.1 Controller
    - 2.2 Events
    - 2.3 GUI Modules
    - 2.4 Model
      - 2.4.1 Design
      - 2.4.2 Implementation
      - 2.4.3 Misc
      - 2.4.4 Tests
    - 2.5 Annotation Models
    - 2.6 Configuration Model
    - 2.7 Filter Models
    - 2.8 Filter Renderermodels
    - 2.9 Section Models
    - 2.10 Templates Model
    - 2.11 Widgets
    - 2.12 Widget Components
    - 2.13 Widget Events
    - 2.14 Widget Model
    - 2.15 Test
    - 2.16 Model Test
    - 2.17 Widgets Test

2.4 Model | 2.4.1 Design | 2.4.2 Implement... | 2.4.3 Misc

Model

Purpose and Responsibility

The Model module relaizes the main model of the application. This model consitsts of the Document, Chapters, Sections, References, Relations and Relation Types. This models encapsulats the information about the structure of a software architecture document (Document, Chapters, Sections) , the relations (Relations, Relation Types) betweens the parts of the structure (previously mentioned) and the outside world

Version

1.0

Change History

Version 1.0 - 01.07.2013
Description:
-Initial official release of the module.

Persons in Charge

Dimitrij Pankratz
Davide Casciato

Dependencies

| Source | Relation Type | Target |
|---|---|---|
| 2.4 Model | depends on | 2.9 Section Models |
| 2.14 Widget Model | is part of | 2.4 Model |
| 2.9 Section Models | is part of | 2.4 Model |
| 2.4 Model | depends on | 2.14 Widget Model |
| 2.1 Controller | depends on | 2.4 Model |

Requirements

| Source | Relation Type | Target |
|---|---|---|
| 1.1.8 Documentation s... | realized by | 2.4 Model |
| 1.1.14 Text-based form... | realized by | 2.4 Model |
| 1.1.9 Relations betwee... | realized by | 2.4 Model |
| 1.1.16 Relation types fo... | realized by | 2.4 Model |
| 1.1.12 References to e... | realized by | 2.4 Model |

Comments

Other parts of the application model, e.g., Template Models, Widget Models, or Configuration Models are contained in other modules.

Widget Library

- Text Field
- Image
- Text Area
- Relation List
- HTML Editor

Component for long texts.

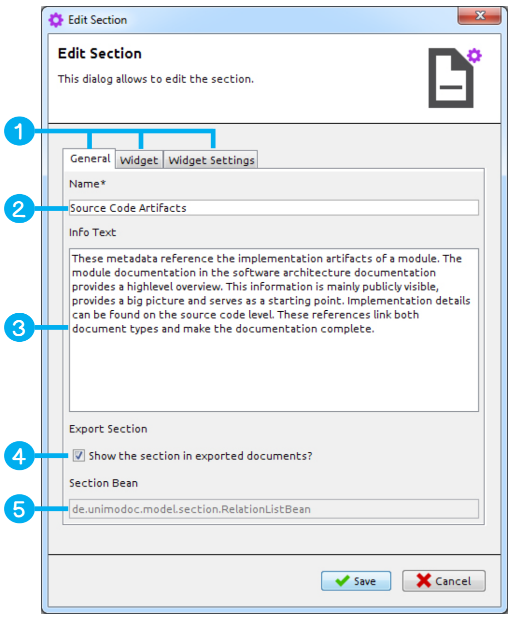
Last saved: 21:41:27

**Figure 4.3:** Editing Document Structure in UniMoDoc
*Item* ❶*: Widget Library, Item* ❷*: Available Widgets, Item* ❸*: Widget Description, Item* ❹*: Apply Button, Item* ❺*: Delete Section Button, Item* ❻*: Configure Section Button*

The widgets in the content area are not editable. This behavior is a result of a strict separation between the editing of the content and the editing of the structure of a chapter. The user can return to the editing of the content with the button in Figure 4.3, Item ❹. Figure 4.3, Item ❺ points on a button for deleting of sections. The configuration of sections is accessible via the button in Figure 4.3, Item ❻.

### Section Configuration

The button shown in Figure 4.3, Item ❻ opens a dialog for section configuration. This dialog is shown in Figure 4.4. Its content is structured in three tabs Figure 4.4, Item ❶. The first tab shows the general settings and information about the section. The second tab shows the selected widget of the section and provides a selectable list of alternate applicable widgets. The content of the last tab is dependent on the selected widget. It displays the settings provided by selected widget.
In the general settings the user can define the name of the section (cf. Figure 4.4, Item ❷). The info text from Figure 4.4, Item ❸ is a hint of what information to fill in, which is accessible from the info button displayed in Figure 4.2, Item ❻. When exporting the document to another format, single sections can be omitted. This is controlled by the checkbox in from Figure 4.4, Item ❹. Figure 4.4, Item ❺ shows the *Section Bean*, i.e., the internal data type of a section. Once the section is created, the *Section Bean* cannot be changed.



Edit Section

This dialog allows to edit the section.

Name*

Source Code Artifacts

Info Text

These metadata reference the implementation artifacts of a module. The module documentation in the software architecture documentation provides a highlevel overview. This information is mainly publicly visible, provides a big picture and serves as a starting point. Implementation details can be found on the source code level. These references link both document types and make the documentation complete.

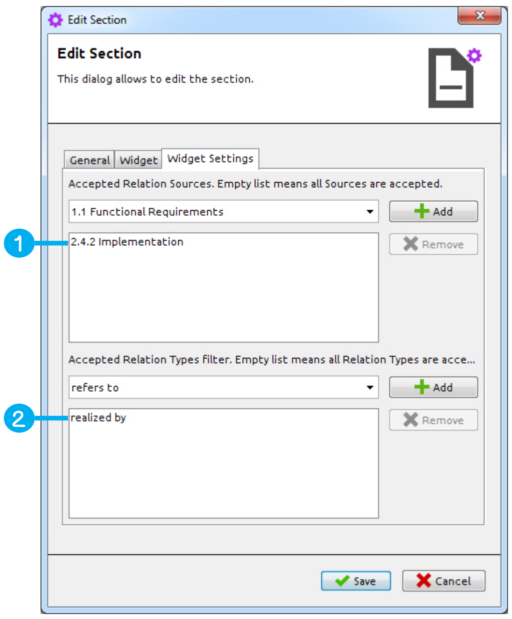
Export Section

☑ Show the section in exported documents?

Section Bean

de.unimodoc.model.section.RelationListBean

**Figure 4.4:** Section Configuration
*Item ❶: Tabs, Item ❷: Section Name, Item ❸: User Hint, Item ❹: Export Option, Item ❺: Internal Data Type*



Edit Section

This dialog allows to edit the section.

Accepted Relation Sources. Empty list means all Sources are accepted.

1.1 Functional Requirements

2.4.2 Implementation

Accepted Relation Types filter. Empty list means all Relation Types are acce...

refers to

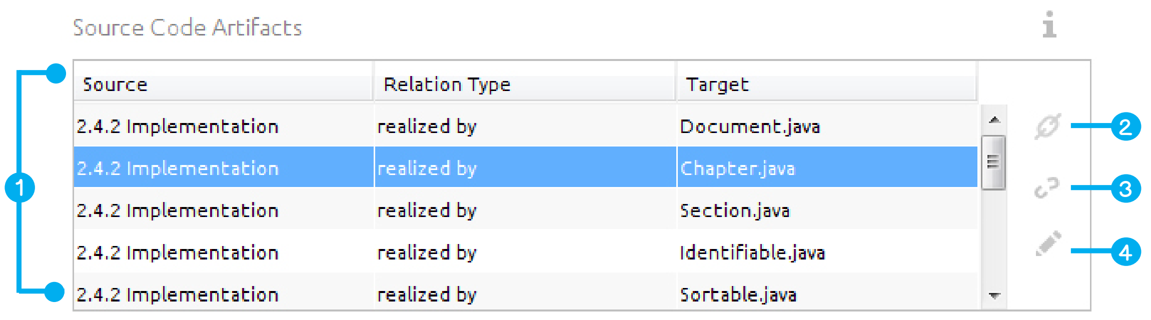
realized by

**Figure 4.5:** RelationList Widget Settings
*Item ❶: List of Accepted Relation Sources, Item ❷: List of Accepted Relation Types*

### RelationList Widget

Available relations of the document can be linked in sections. In order to achieve this, special *Section Beans* and widgets are provided. This concept will be demonstrated on the example of the `de.unimodoc.model.section.RelationListBean` and *RelationList* widget. Figure 4.4, Item ❺ indicates that the section uses a `de.unimodoc.model.section.RelationListBean` as internal data type. Additionally, a *RelationList* widget is assigned to it.

The *RelationList* widget provides its own settings, which are illustrated in Figure 4.5. Basically, it has two settings. The first one is a list to filter the accepted relation sources in Figure 4.5, Item ❶. If the list is empty, all sources are accepted. However, the list in Figure 4.5, Item ❶ contains one entry. This means that the user can only link relations with the selected source. Figure 4.6, Item ❶ shows the rendered *RelationList* widget of the corresponding section. The *Source* row contains only the accepted relation source from the settings. This approach has several reasons. Firstly, the usability is improved because the relation sources are preselected. Secondly, the possibility to link wrong relations decreases. Finally, if the relation source changes over time and it is not accepted by the *RelationList*, the link will be automatically removed.
The second setting of the *RelationList* widget is a list that filters the accepted relation types. Its function and purpose are similar to the accepted sources. In Figure 4.5, Item ❷ the only accepted relation type is *realized by*. This relation type is used for relations between a module and its source code artifacts. Since the section lists source code artifacts only, no other relation types are required for this *RelationList*.
Furthermore, Figure 4.6 shows additional buttons on the right hand-side. The buttons are per default invisible. Only if the user hovers the mouse over the *RelationList* widget, the buttons become visible. The button in Figure 4.6, Item ❷ allows to link relations to the widget. Figure 4.6, Item ❸ points on the button for deleting the link of the selected entry in the list. Finally, the button in Figure 4.6, Item ❹ provides a quick access to the options of the relation, relation type, and reference of the selected link. More information about the management of relations can be found in [Casciato, 2013].

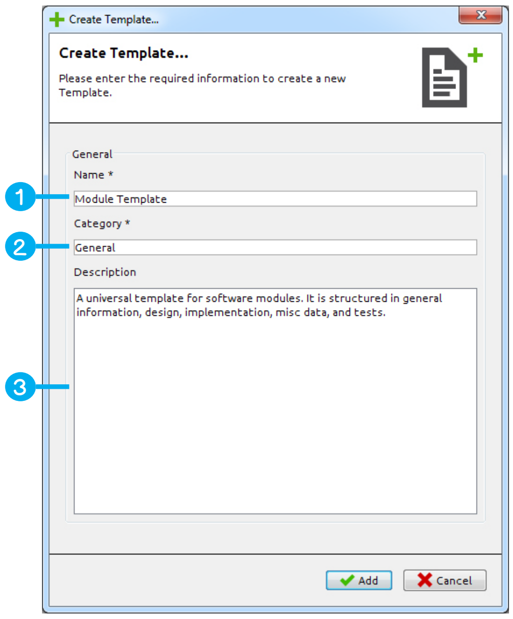


| Source | Relation Type | Target |
|---|---|---|
| 2.4.2 Implementation | realized by | Document.java |
| 2.4.2 Implementation | realized by | Chapter.java |
| 2.4.2 Implementation | realized by | Section.java |
| 2.4.2 Implementation | realized by | Identifiable.java |
| 2.4.2 Implementation | realized by | Sortable.java |

**Figure 4.6:** RelationList Widget
*Item ❶: RelationList Widget, Item ❷: Link Relation Button, Item ❸: Remove Relation Button, Item ❹: Quick Access to Options Button*

### Templates

UniMoDoc supports two types of templates. The first type is for complete documents, i.e., a document is saved as template. The second type operates on the chapter level. This means that a chapter and all its subchapters are saved into a template. The procedure for creating new templates is the same for both types. The user needs to open a context menu for the document or a chapter in the tree structure and to select the entry *Save as Template*. In the next step UniMoDoc will present the dialog displayed in Figure 4.7. The dialog requires a name for the template (cf. Figure 4.7, Item ❶) and a category (cf. Figure 4.7, Item ❷). The category is meant for structuring templates. Finally, a description for a template can be provided (cf. Figure 4.7, Item ❸). The templates are saved in a central directory and registered in the application.



**Create Template...**

Please enter the required information to create a new Template.

General

Name *

Module Template

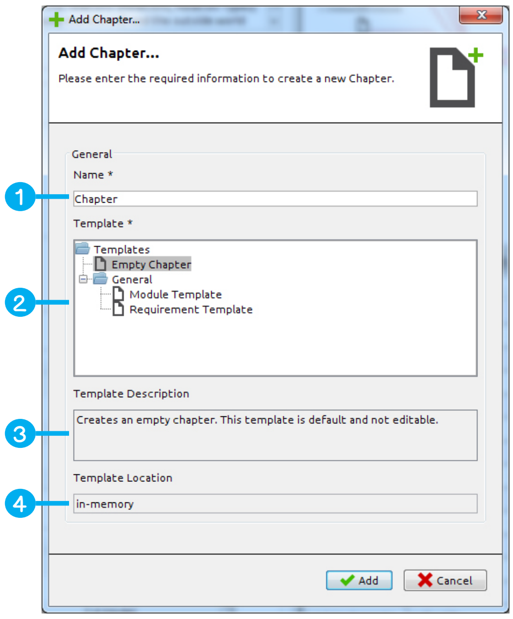
Category *

General

Description

A universal template for software modules. It is structured in general information, design, implementation, misc data, and tests.

**Figure 4.7:** Creating Template
*Item ❶: Template Name, Item ❷: Template Category, Item ❸: Template Description*



**Add Chapter...**

Please enter the required information to create a new Chapter.

General

Name *

Chapter

Template *

Templates
- Empty Chapter
- General
  - Module Template
  - Requirement Template

Template Description

Creates an empty chapter. This template is default and not editable.

Template Location

in-memory

**Figure 4.8:** Using Templates
*Item ❶: Chapter Name, Item ❷: Used Template, ❸: Template Description, ❹: Template Location*

When creating a new document or chapter, it is possible to choose between an empty object or template. Figure 4.8 demonstrates this approach on the example of creating chapters. The user has to enter the name for the new chapter (cf. Figure 4.8, Item ❶) and to select a template (cf. Figure 4.8, Item ❷). Whereby the *Empty Chapter* displayed in Figure 4.8, Item ❷ is a special template that cannot be deleted. In addition, the user is informed about the purpose of the template (cf. Figure 4.8, Item ❸) and its location on the machine (cf. Figure 4.8, Item ❹).

### 4.2.2 Module Template

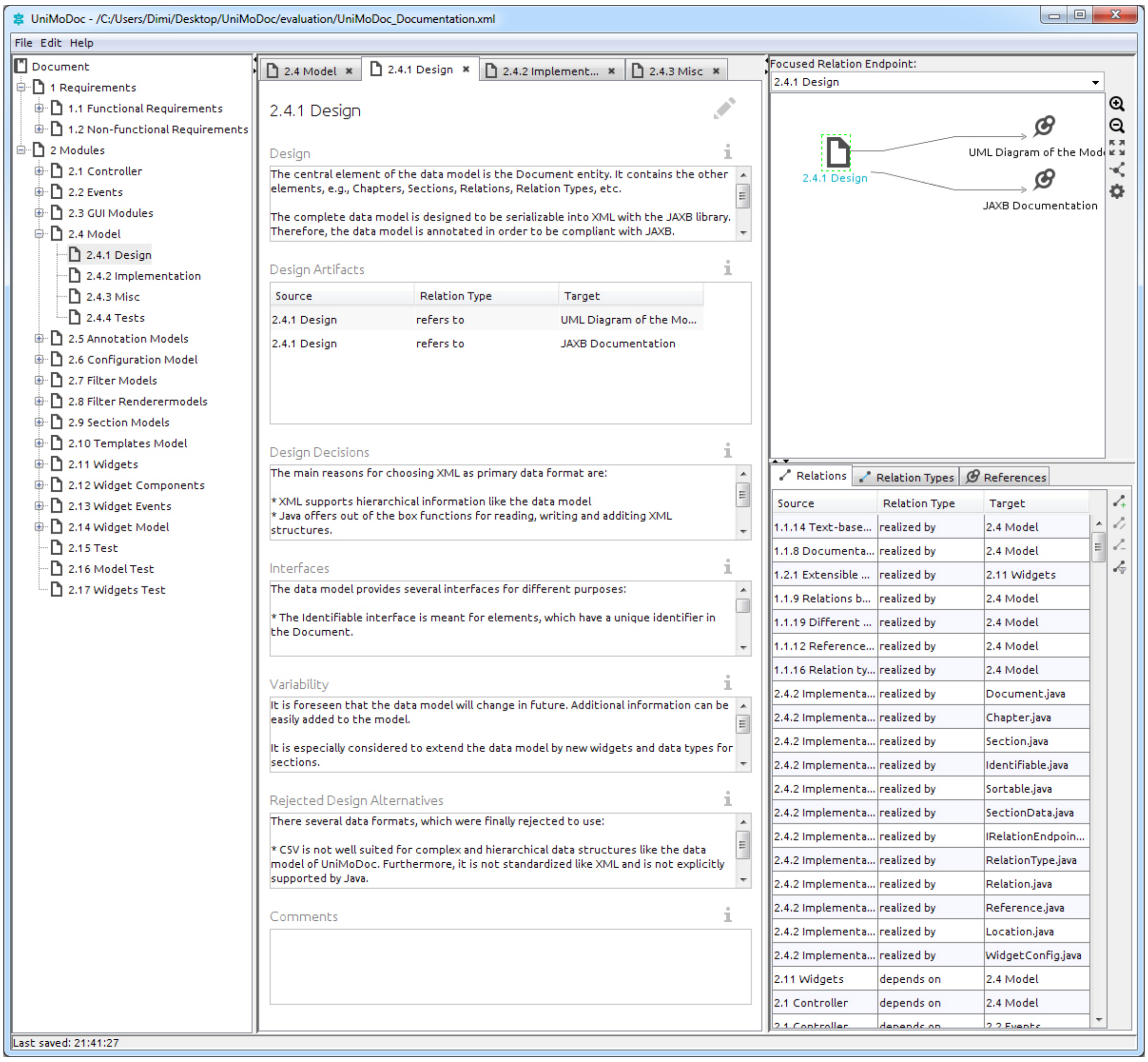
The module metadata defined in Chapter 3.2.3 is meant to be used in a template for module documentation. However, the metadata can be aggregated to a template in various ways. This chapter serves several purposes. It suggests a universally applicable template using the metadata from Chapter 3.2.3 and the implemented concepts of UniMoDoc. In this manner it satisfies the requirement for a module template from Chapter 3.1 and demonstrates how UniMoDoc can be utilized.

The overall suggested template uses only three types of widgets:

- TextFields for short, single lined texts
- TextArea for all other types of text
- RelationList for displaying the relations to other modules, requirements, external resources, etc.

It is easily conceivable to utilize more specialized and comfortable widgets. However, this template provides a ready to use foundation and starting point for future customizations.

The suggested template is structured into five parts. The first part provides a quick overview of the module. This part is displayed in Figure 4.2. The name of the chapter equals the name of the module. Furthermore, it contains the following metadata: *Purpose and Responsibility*, *Version*, *Change History*, *Persons in Charge*, *Dependencies*, *Requirements*, and *Comments*. Only the *Comments* section is not described in Chapter 3.2.3. The template should be preferably universally applicable. Therefore, the *Comments* section provides a possibility to add information, which fits into a particular chapter but not into other sections. *Comments* sections are also used in other parts of the template.
The purpose of this part is to provide the most important information about a module at first glance. If more detailed information is required, the user can directly navigate to one of the subchapters: *Design*, *Implementaion*, *Misc*, or *Tests*. This subchapters form the other parts of the template. However, the *Tests* subchapter is not part of this thesis. It is described in [Casciato, 2013].



2.4.1 Design

Design

The central element of the data model is the Document entity. It contains the other elements, e.g., Chapters, Sections, Relations, Relation Types, etc.

The complete data model is designed to be serializable into XML with the JAXB library. Therefore, the data model is annotated in order to be compliant with JAXB.

Design Artifacts

| Source | Relation Type | Target |
|---|---|---|
| 2.4.1 Design | refers to | UML Diagram of the Mo... |
| 2.4.1 Design | refers to | JAXB Documentation |

Design Decisions

The main reasons for choosing XML as primary data format are:

* XML supports hierarchical information like the data model
* Java offers out of the box functions for reading, writing and additing XML structures.

Interfaces

The data model provides several interfaces for different purposes:

* The Identifiable interface is meant for elements, which have a unique identifier in the Document.

Variability

It is foreseen that the data model will change in future. Additional information can be easily added to the model.

It is especially considered to extend the data model by new widgets and data types for sections.

Rejected Design Alternatives

There several data formats, which were finally rejected to use:

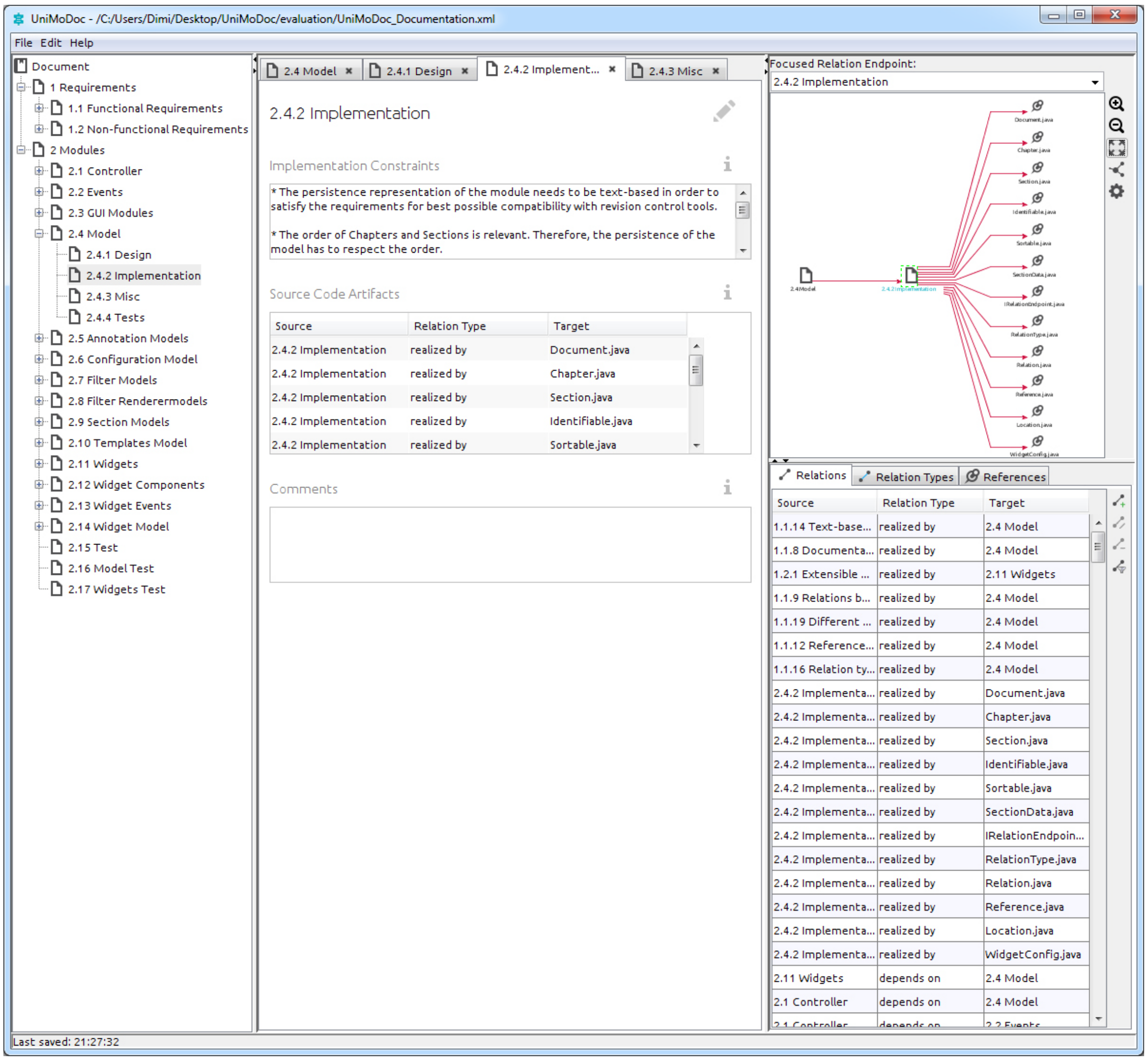
* CSV is not well suited for complex and hierarchical data structures like the data model of UniMoDoc. Furthermore, it is not standardized like XML and is not explicitly supported by Java.

Comments

| Source | Relation Type | Target |
|---|---|---|
| 1.1.14 Text-base... | realized by | 2.4 Model |
| 1.1.8 Documenta... | realized by | 2.4 Model |
| 1.2.1 Extensible ... | realized by | 2.11 Widgets |
| 1.1.9 Relations b... | realized by | 2.4 Model |
| 1.1.19 Different ... | realized by | 2.4 Model |
| 1.1.12 Reference... | realized by | 2.4 Model |
| 1.1.16 Relation ty... | realized by | 2.4 Model |
| 2.4.2 Implementa... | realized by | Document.java |
| 2.4.2 Implementa... | realized by | Chapter.java |
| 2.4.2 Implementa... | realized by | Section.java |
| 2.4.2 Implementa... | realized by | Identifiable.java |
| 2.4.2 Implementa... | realized by | Sortable.java |
| 2.4.2 Implementa... | realized by | SectionData.java |
| 2.4.2 Implementa... | realized by | IRelationEndpoin... |
| 2.4.2 Implementa... | realized by | RelationType.java |
| 2.4.2 Implementa... | realized by | Relation.java |
| 2.4.2 Implementa... | realized by | Reference.java |
| 2.4.2 Implementa... | realized by | Location.java |
| 2.4.2 Implementa... | realized by | WidgetConfig.java |
| 2.11 Widgets | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.2 Events |

**Figure 4.9:** Module Template - Design

The *Design* subchapter shown in Figure 4.9 contains the information, which is typically produced during the design phase of a module. It lists the metadata: *Design*, *Design Artifacts*, *Design Decisions*, *Interfaces*, *Variability*, *Rejected Design Alternatives*, and *Comments*. The section *Design Artifacts* does not appear in the metadata from Chapter 3.2.3. Design resources like UML or ER diagrams were originally planned to be grouped under the topic *Other Artifacts*. However, this topic contains not only design relevant information and is placed in another chapter. Additionally, it is possible to use the Image widget to display graphical design notations. It is also conceivable to implement a widget for creating, e.g., UML diagrams directly in UniMoDoc.



- Document
  - 1 Requirements
    - 1.1 Functional Requirements
    - 1.2 Non-functional Requirements
  - 2 Modules
    - 2.1 Controller
    - 2.2 Events
    - 2.3 GUI Modules
    - 2.4 Model
      - 2.4.1 Design
      - 2.4.2 Implementation
      - 2.4.3 Misc
      - 2.4.4 Tests
    - 2.5 Annotation Models
    - 2.6 Configuration Model
    - 2.7 Filter Models
    - 2.8 Filter Renderermodels
    - 2.9 Section Models
    - 2.10 Templates Model
    - 2.11 Widgets
    - 2.12 Widget Components
    - 2.13 Widget Events
    - 2.14 Widget Model
    - 2.15 Test
    - 2.16 Model Test
    - 2.17 Widgets Test

2.4.2 Implementation

Implementation Constraints

* The persistence representation of the module needs to be text-based in order to satisfy the requirements for best possible compatibility with revision control tools.

* The order of Chapters and Sections is relevant. Therefore, the persistence of the model has to respect the order.

Source Code Artifacts

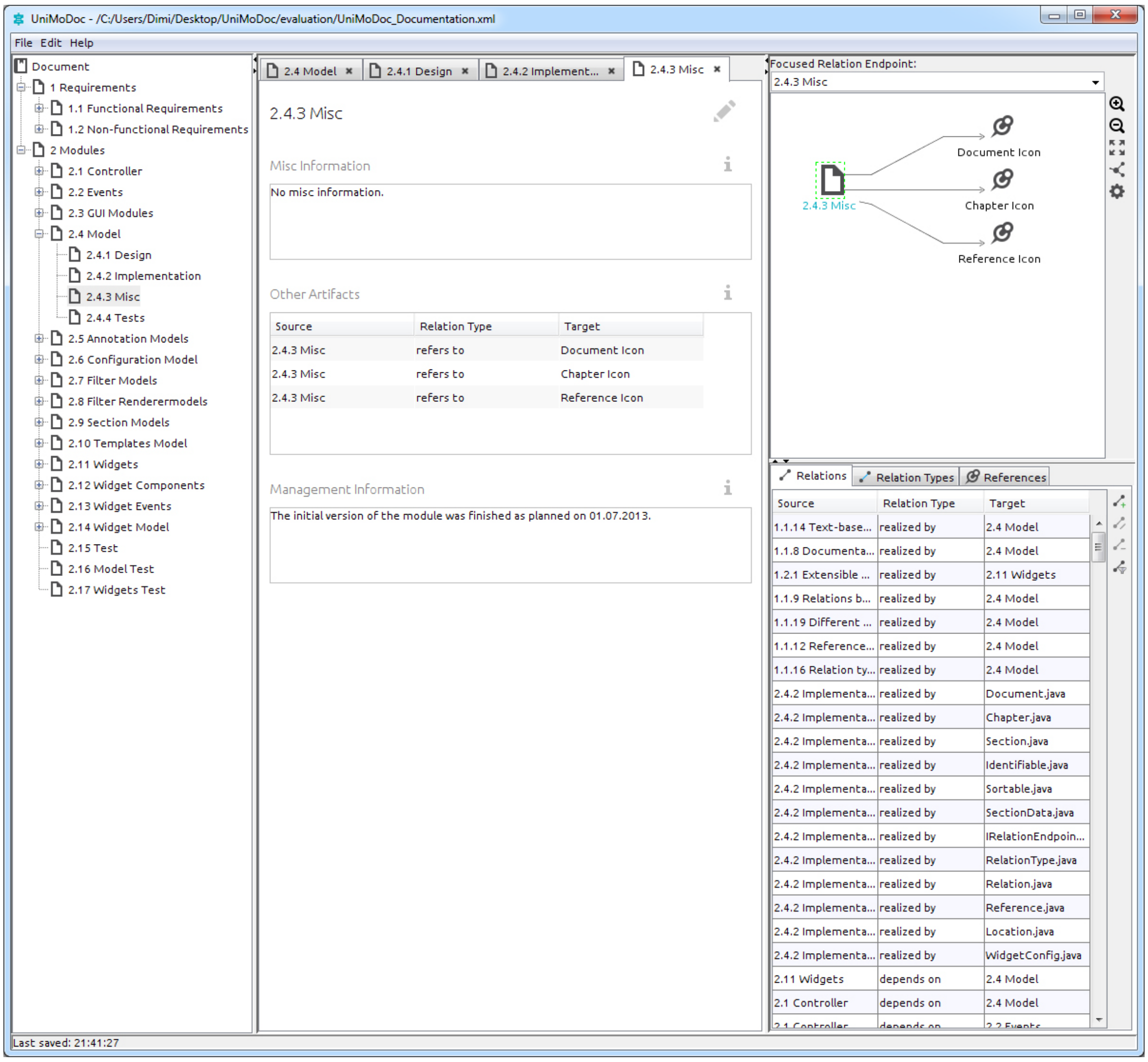
| Source | Relation Type | Target |
|---|---|---|
| 2.4.2 Implementation | realized by | Document.java |
| 2.4.2 Implementation | realized by | Chapter.java |
| 2.4.2 Implementation | realized by | Section.java |
| 2.4.2 Implementation | realized by | Identifiable.java |
| 2.4.2 Implementation | realized by | Sortable.java |

Comments

| Source | Relation Type | Target |
|---|---|---|
| 1.1.14 Text-base... | realized by | 2.4 Model |
| 1.1.8 Documenta... | realized by | 2.4 Model |
| 1.2.1 Extensible ... | realized by | 2.11 Widgets |
| 1.1.9 Relations b... | realized by | 2.4 Model |
| 1.1.19 Different ... | realized by | 2.4 Model |
| 1.1.12 Reference... | realized by | 2.4 Model |
| 1.1.16 Relation ty... | realized by | 2.4 Model |
| 2.4.2 Implementa... | realized by | Document.java |
| 2.4.2 Implementa... | realized by | Chapter.java |
| 2.4.2 Implementa... | realized by | Section.java |
| 2.4.2 Implementa... | realized by | Identifiable.java |
| 2.4.2 Implementa... | realized by | Sortable.java |
| 2.4.2 Implementa... | realized by | SectionData.java |
| 2.4.2 Implementa... | realized by | IRelationEndpoin... |
| 2.4.2 Implementa... | realized by | RelationType.java |
| 2.4.2 Implementa... | realized by | Relation.java |
| 2.4.2 Implementa... | realized by | Reference.java |
| 2.4.2 Implementa... | realized by | Location.java |
| 2.4.2 Implementa... | realized by | WidgetConfig.java |
| 2.11 Widgets | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.2 Events |

**Figure 4.10:** Module Template - Implementation

The *Implementation* subchapter presented in Figure 4.10 lists the information typically produced during the implementation phase. It has following metadata: *Implementation Constraints*, *Source Code Artifacts*, and *Comments*. Compared to the *Design* subchapter only few information is contained. That's because UniMoDoc does not support the implementation phase explicitly. Still, it links all relevant parts together and makes them traceable.



2.4.3 Misc

Misc Information

No misc information.

Other Artifacts

| Source | Relation Type | Target |
|---|---|---|
| 2.4.3 Misc | refers to | Document Icon |
| 2.4.3 Misc | refers to | Chapter Icon |
| 2.4.3 Misc | refers to | Reference Icon |

Management Information

The initial version of the module was finished as planned on 01.07.2013.

| Source | Relation Type | Target |
|---|---|---|
| 1.1.14 Text-base... | realized by | 2.4 Model |
| 1.1.8 Documenta... | realized by | 2.4 Model |
| 1.2.1 Extensible ... | realized by | 2.11 Widgets |
| 1.1.9 Relations b... | realized by | 2.4 Model |
| 1.1.19 Different ... | realized by | 2.4 Model |
| 1.1.12 Reference... | realized by | 2.4 Model |
| 1.1.16 Relation ty... | realized by | 2.4 Model |
| 2.4.2 Implementa... | realized by | Document.java |
| 2.4.2 Implementa... | realized by | Chapter.java |
| 2.4.2 Implementa... | realized by | Section.java |
| 2.4.2 Implementa... | realized by | Identifiable.java |
| 2.4.2 Implementa... | realized by | Sortable.java |
| 2.4.2 Implementa... | realized by | SectionData.java |
| 2.4.2 Implementa... | realized by | IRelationEndpoin... |
| 2.4.2 Implementa... | realized by | RelationType.java |
| 2.4.2 Implementa... | realized by | Relation.java |
| 2.4.2 Implementa... | realized by | Reference.java |
| 2.4.2 Implementa... | realized by | Location.java |
| 2.4.2 Implementa... | realized by | WidgetConfig.java |
| 2.11 Widgets | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.4 Model |
| 2.1 Controller | depends on | 2.2 Events |

**Figure 4.11:** Module Template - Misc

The *Misc* subchapter from Figure 4.11 groups the remaining metadata. It contains the sections: *Misc Information*, *Other Artifacts*, and *Management Information*. Compared to previous chapters the *Misc* subchapter contains no *Comments* section. Instead, the *Misc Information* bundles all the data that does not fit into the other chapters.
The suggested module documentation template tries to be universal. It contains common metadata for modules discussed in Chapter 3.2.3. Still, no template can be generic enough to cover all possible needs. For that reason the template is only considered as a starting point for customization and as proof of concept. UniMoDoc is designed for quick and easy template creation. Thus the template can be easily tailored to meet custom requirements.

# 5 Evaluation

In order to verify the expected effects of the implemented concepts, an experiment was planned and executed. The overall evaluation is a cooperative effort of this thesis and [Casciato, 2013]. This chapter describes the evaluation and is structured based on [Prechelt, 2001] and [Jedlitschka and Pfahl, 2005]. First, the initial goals of the evaluation are described in Chapter 5.1. Then, the selected experiment type and rejected alternatives are presented in Chapter 5.2. The parameters of the experiment are described in Chapter 5.3. Considering the parameters, the initial goals are adjusted in Chapter 5.4. Chapter 5.5 describes how the single parts of the experiment are planned in detail. A short overview of the experiment parts, their timing, and materials is provided in Chapter 5.6. Chapter 5.7 discusses the internal and external validity of the experiment. Chapter 5.8 describes briefly the experiment pilot and the experiment execution. The results of the experiment are presented in Chapter 5.9. Finally, Chapter 5.10 interprets the results and draws a conclusion.

## 5.1 Initial Goals

One of the main research objectives of this thesis is suggesting concepts for a better tool support of module documentation. Chapter 3.2 describes these concepts in detail. The implementation of the suggested design serves as a proof of concept. Details on the implementation and the resulting application can be found in Chapter 4. The next required step is to find out, whether the concepts really provide the expected advantages. This intention results in a concrete question that can be formulated more precisely as follows:

1 *Do the implemented concepts in UniMoDoc offer an added value in module documentation compared to conventional documentation tools (e.g., word processors, documentation generators)?*

The implementation of the concepts, in form of UniMoDoc, serves as a proof of concept and it provides a more realistic testability of the concepts. However, it introduces additional complexity for the evaluation of the concepts. Thus the implementation may influence the evaluation of the concepts negatively. The aspects added to the concepts by the implementation, which can be responsible for a negative influence, are mainly stability und usability. In other words, the application fails, crashes, or it is not user-friendly, i.e., the user is not able to accomplish the tasks efficiently. Furthermore, the usability is part of the requirements of UniMoDoc. Thus this thesis is generally highly interested in a stable prototype with a good usability. Consequentially, an additional question arises that can be formulated as follows:

2 *Is UniMoDoc operational according to its stability and usability?*

## 5.2 Experiment Type

This thesis makes use of a controlled experiment in order of evaluation. The specialty of controlled experiments is the isolation of single attributes of interest, so-called independent variables. All other attributes remain strictly unchanged. They are called dependent variables. Results of samples containing the independent variables are compared to results with dependent variables only. To put it another way, the experimental group is compared to a control group. Thus the objective is to prove the effects caused by independent variables.
If properly executed, controlled experiments provide a highly reliable and accurate statement about the impact of the independent variables. Furthermore, it is possible to obtain objective and subjective results. Another reason for choosing a controlled experiment is its generally good repeatability. Other considered and rejected experiment types are:

**Case Studies**
Case studies are typically custom-designed scenarios that examine a selected use case, e.g., of a tool or a method. Similar to the controlled experiments they can compare different approaches. However, unlike the controlled experiments they do not attempt to control the other variables, i.e., the dependent variables. Thus it is not possible to determine the cause of the observed effects exactly.

**Field Experiments**
Experiments executed in a real world environment are called field experiments. They benefit from more realistic results than experiments with an artificial setting. Still, due to their complexity, field experiments are hard to repeat and even to document. Furthermore, they face the same problems like case studies. The observed effects cannot be precisely mapped to the variables in a complex setting.

**Surveys**
In surveys participants answer selected questions of interest. Their answers are subjective only. Surveys are typically less effort consuming than other experiment types. The main disadvantages of the surveys are the subjectivity of the answers and their difficult interpretation.

## 5.3 Parameters

The experiment is designed and executed in a particular context. This context consists of a set of parameters. The description of the parameters is necessary in order to fully understand the experiment and to ensure the repeatability. Additionally, the further design of the experiment has to consider the parameters. The parameters of the experiment are:

- The experiment was conducted during the tutorial of the *Software Quality Assurance and Software Maintenance* lecture at the University of Stuttgart with a time limit of 90 minutes.

- The participants were master students of Software Engineering. Therefore, at least a basic experience in Java programming can be assumed.
- The tutorial of the *Software Quality Assurance and Software Maintenance* lecture has maximally 12 attendees. Thus the experiment assumed 12 participants at maximum.
- The computer pool of the computer science building at the University of Stuttgart served as the infrastructure for the experiment. Thus the participants were able to work with UniMoDoc or access internet.

## 5.4 Adjusted Goals

In order to answer the first question defined in Chapter 5.1 it is necessary to specify it more precisely and to select concrete functions of UniMoDoc for the evaluation. UniMoDoc provides various functions and supports different use cases. One typical and important use case is to utilize the documentation created with UniMoDoc, in order to retrieve information about the documented modules. This thesis expects that especially the explicitly defined relations and their visualization in UniMoDoc will help the participants to retrieve the required information more efficiently compared to conventional documentation. As a result two questions can be formulated:

$Q_1$ *Will the participants find more required information using UniMoDoc within a given time frame compared to using similar module documentation created with a word processor and Javadoc?*

$Q_2$ *Will the explicitly defined relations and their visualization help the participants to find the required information using UniMoDoc?*

The second question from the Chapter 5.1 chapter according the stability and usability can be divided into two different questions in order to observe the attributes separately:

$Q_3$ *Will UniMoDoc crash during the evaluation?*

$Q_4$ *Do the participants feel positive about the usability of UniMoDoc and can they accomplish typical tasks without additional instructions?*

Initially it was planned in addition, to evaluate the benefits of UniMoDoc while documenting the modules. It was expected that the templates from this thesis and from [Casciato, 2013] would help to get more detailed and better structured module documentation. This idea was finally dropped due to the given parameters, especially the small time frame. Ideally the participants would need to document the modules, which they have created themselves. However, this was not possible within the 90 minutes. Alternatively it would be necessary to provide the module documentation to the participants in another form. However, the documentation in other form would unavoidably prescribe and influence the results. Therefore, the use case of documenting modules is mainly evaluated according to the usability of the application.

## 5.5 Experiment Design

This chapter describes the single parts of the experiment in detail. The overall experiment can be divided into 5 parts: *Introduction*, *First Assignment Part*, *First Feedback*, *Second Assignment Part*, *Second Feedback*. In the first *Introduction* part the participants are instructed and receive the initial materials. In the *First Assignment Part* they need to find information in a module documentation with and without UniMoDoc. The *First Feedback* collects the subjective impressions of the participants after the *First Assignment Part*. In the *Second Assignment Part* the participants have to complete a module documentation with UniMoDoc and to provide their feedback in the final *Second Feedback* part.

### 5.5.1 Introduction

This part aims to answer the questions $Q_1$ to $Q_4$. During this part the participants receive an information note with a short description of the experiment and a leaflet with a unique number. These materials are distributed absolutely randomly to the participants. The unique number identifies a particular participant and the corresponding results. It starts with one and increments by one.
Additionally, the information note contains a link to an archive downloadable from the internet with additional materials required for the next parts and an executable file of UniMoDoc. The participants are instructed to unpack the archive on their machines and to check whether they can execute UniMoDoc.

### 5.5.2 First Assignment Part

In the first assignment part the participants are separated into 4 groups. The number of the group of each person is derived from the following function, where x equals the unique number of the participant:

$$f(x) = \begin{cases} x \bmod 4 & \text{if } x \bmod 4 > 0 \\ 4 & \text{if } x \bmod 4 = 0 \end{cases}$$

Since the unique numbers of the participants are distributed randomly, this approach ensures that the participants are randomly assigned to the groups.

There are two similar assignments named A and B with questions. At the beginning each group receives the first assignment and works with or without UniMoDoc. Then, each group receives another assignment. If a group previously used UniMoDoc, it works in the second part without UniMoDoc and vice versa. In the notation of [Prechelt, 2001] the execution can be described as follows:

$$\begin{array}{ll}
G_1 : A/UniMoDoc & B/!UniMoDoc \\
G_2 : A/!UniMoDoc & B/UniMoDoc \\
G_3 : B/UniMoDoc & A/!UniMoDoc \\
G_4 : B/!UniMoDoc & B/UniMoDoc
\end{array}$$

$G_n$ identifies the number of the group. *A/UniMoDoc* means that the assignment A is processed with the utilization of UniMoDoc. *B/!UniMoDoc* means that the assignment B is processed without the utilization of UniMoDoc. Consequentially, the first line says that the group number 1 will firstly process the assignment A with UniMoDoc and secondly it will process the assignment B without UniMoDoc.
This approach provides several benefits. It is possible to objectively compare the results between the utilization of UniMoDoc and without it. The participants are able to compare UniMoDoc with conventional documentation and to provide subjective impressions. Furthermore, the permutation of the sequence helps to detect possible sequence specific effects.

### Artifacts

The participants receive an archive during the introduction, which contains the source code artifacts of UniMoDoc. The source code is organized into modules, which are structured as Java packages. Additionally, the archive contains a conventional module documentation and a documentation created with UniMoDoc. The conventional documentation of modules is contained in the `package-info.java` files of the corresponding packages. It is also displayed in the Javadoc documentation. Additionally to the usually expected Javadoc information, the documentation contains references to the realized requirements and dependencies to other modules. An explicit note provided with the assignment informs the participants about this information in the Javadoc documentation.
The module documentation created with UniMoDoc uses an adjusted module template of this thesis. The adjusted template does not contain a design and misc part because there is no equivalent for this in Javadoc. The content of the module documentation is identical in Javadoc and UniMoDoc. In order to ensure the identical documentation, the Javadoc comments in the source code were generated by UniMoDoc.
Furthermore, the tests for each module are organized in an additional test module (package). The archive contains also automatically generated JUnit reports as HTML files. The requirements are provided as a PDF file. The participants are informed about these files and their location in the archive.

### Assignments

Both assignments, A and B, contain five questions of identical type for different modules. The types of questions are:

- Which modules realize the requirement ...?
- List all JUnit test cases, which test the requirement ... and list the number of methods per class.
- Complete the description of the requirement ..., which is realized by exactly one module.
- The class ... of module ... was tested and protocolled several times. List the date, time and number of failures for each protocol.
- List all ingoing and outgoing dependencies of module ... separately.

The results of the assignments are collected at the end of this part.

### 5.5.3 First Feedback

After the first assignment part a feedback is collected from the participants. The main goal of this part is to capture the fresh and subjective impressions of the participants. The participants are asked, whether it was easier to process the assignments with UniMoDoc or without. Furthermore, they are asked, whether they used the relation visualization in UniMoDoc and if they were able to process the assignments faster with it. Additionally, there are questions about the usability, helpful functions, missing functions, etc.

### 5.5.4 Second Assignment Part

In the second assignment part all participants are assigned to one single group. The goal of this part is to answer the questions $Q_3$ and $Q_4$. Therefore, the participants are instructed to process the assignment C, which requires to create a documentation with UniMoDoc. The contents and appearance of the documentation are prescribed. The participants receive only a short getting started document. At the end, they have to save the created documentation on a Universal Serial Bus (USB) flash drive. This way, it is possible to objectively control the results. In addition, the participants will provide a subjective feedback in the next part of the experiment.

#### Artifacts

The archive from the introduction contains additionally a UniMoDoc file with incomplete module documentation. Furthermore, it contains a UML diagram and a source code artifact of the corresponding module. These files are not linked in UniMoDoc yet.
The participants have no experience in documenting modules with UniMoDoc and they are not instructed how this can be achieved. Nevertheless, they receive a short getting started document, which describes the most important functions.
Additionally, the participants get a USB flash drive for saving the results. The USB flash drive is collected at the end of the assignment.

Assignments

The assignment C contains a detailed description of the module documentation, which needs to be completed with UniMoDoc. The description displays a screenshot of how the documentation should look like at the end. Furthermore, it describes which sections, widgets, relations, relation types, and references need to be created and what content has to be filled in. In particular, the participants need to create three sections, with *TextArea*, *Image*, and *RelationList* widgets. They have to fill in the information text of the sections and the content of the *TextArea*. The participants need to reference the corresponding UML diagram and the source code artifact. Then, they have to create the appropriate relations and link them in the *Image* and *RelationList* widgets. Finally, the participants save the completed module documentation on the USB flash drive.

### 5.5.5 Second Feedback

The second feedback is meant to obtain the subjective impressions of the participants according the usability of UniMoDoc during the second assignment part. Consequentially, the participants are directly asked about the usability and indirectly about helpful, missing, and improvable functions. They are also asked, whether they would like to document modules with UniMoDoc in future and if they would like to use the documentation created with UniMoDoc. Additionally, the participants need to answer control questions about their skills in Java and Javadoc.

## 5.6 Experiment Overview

This chapter provides a short overview of the single experiments parts. The parts are listed in the execution order with the corresponding execution duration in minutes. The overall experiment is limited to 90 minutes. The total duration sum of the single parts equals 80 minutes. The remaining 10 minutes are planned as buffer time, e.g., between the single experiment parts.

**Introduction (5 min)**
*Materials:* Information note, leaflet with a unique identification number, archive with all required digital artifacts
*Description:* Participants receive the initial materials and get informed about the further experiment process.

**First Assignment Part (40 min)**
*Materials:* Assignment A and B
*Description:* The participants are separated into 4 groups $G_1$ to $G_4$. They need to find information in the given module documentation with UniMoDoc and without UniMoDoc. Depending on the group, this experiment is structured into two parts of 20 minutes as follows:

$G_1$ works on assignment A with UniMoDoc, then on assignment B without UniMoDoc

$G_2$ works on assignment A without UniMoDoc, then on assignment B with UniMoDoc

$G_3$ works on assignment B with UniMoDoc, then on assignment A without UniMoDoc

$G_4$ works on assignment B without UniMoDoc, then on assignment A with UniMoDoc

**First Feedback (10 min)**
*Materials:* First feedback form
*Description:* The participants answer questions about the first assignment part.

**Second Assignment Part (15 min)**
*Materials:* Assignment C, short getting started sheet, USB flash drive
*Description:* The participants need to finish incomplete module documentation from the archive and save the results on the given USB flash drive.

**Second Feedback (10 min)**
*Materials:* Second feedback form
*Description:* The participants answer questions about the second assignment part.

## 5.7 Validity of the Experiment

This chapter describes how the internal and external validities are regarded in this experiment.

### 5.7.1 Internal Validity

According to [Prechelt, 2001], the internal validity describes the extent to which the dependent variables are controlled. In other words, it describes whether only the independent variable is responsible for the observed effects and thus how accurate the results are. Therefore, it is necessary to consider the possible threats to the internal validity during the planning and to argue about the precautions took against them. There are myriads of possible threats but most of them can be outlined in several classes. The considered classes of threats for this experiment and their counter-measures are listed below. However, no threats were detected for the classes *History*, *Instrumentation*, and *Regression.*

Selection

The selection of the participants and their assignment to groups can significantly affect the results. It can happen that the characteristics of the participants, e.g., skills, mental abilities, motivation, etc. are not equally distributed among the groups and therefore influence the experiment.
In this experiment the participants are randomly assigned to the groups. Therefore, possible distinctions between them are ideally statistically balanced.

Maturation

Maturation refers to the issue that the participants may evolve during the execution of the experiment and hereby affect the results. It occurs if participants, e.g., learn to accomplish the tasks better during the experiment, employ the knowledge from previous tasks to advantage, fatigue in the course of the experiment, etc.
In the first experiment part a threat exists that the participants will evolve between the assignments A and B. Therefore, the processing order is permutated for the assignments and approaches (with UniMoDoc and without it).

Mortality

The mortality defines a threat that participants may withdraw from the experiment before its ending. This can result in a complete abort of the experiment, e.g., in a case with not random but systematic reason for the attrition.
This experiment is executed exactly in the time frame of a tutorial and the students are informed about it in advance. Additionally, they are advised and motivated to stay to the end of the experiment.

Demand Characteristics

It is possible that the experimenters treat the participant groups differently. They may prefer the group from which they expect better results by, e.g., motivating, helping, demotivating the other groups, etc.
In this experiment each participant has to solve assignments with and without UniMoDoc. Thus no group exists, where only negative or positive results are expected. Additionally, the assignments are provided to the participants in text form. Except from the instruction to use UniMoDoc or the conventional documentation, care has been taken to ensure that the assignments are absolutely equally.

#### Processing Errors

Processing errors can occur in the course of the experiment and during the evaluation of the results. Thus, e.g., the groups can be mixed up, results can be lost, subjective appraisal of the results can vary, transfer errors of the results can occur, etc.
The participants receive a unique number for identification before the start of the first assignment part. This number is used for matching all generated results to a specific participant and group. Moreover, there is a portfolio for all results of a single participant, which ensures that the results are not mixed up. The accuracy of the evaluation of the results is increased by a defined evaluation method and double evaluation of the both experimenters.

### 5.7.2 External Validity

According to [Prechelt, 2001], the external validity describes how well the results of an experiment can be generalized. Thus it describes whether the results are applicable to other, especially real world use cases. The design of this experiment considers the external validity and attempts to maximize it. Nevertheless, this experiment has some limits due to the scope of the both theses and the parameters. The considerations and limits of the external validity are listed below.

This experiment utilizes the artifacts of UniMoDoc as materials for the assignments. Thus the utilized software project is not a specially created mock-up but a real software project. However, UniMoDoc was created in the scope of two diploma theses and cannot be compared with mature and professional software projects. It is therefore advisable to conduct experiments with another software projects as assignment material.

The assignments in the first part of the experiment attempt to simulate typical use cases for finding information in module documentation. They address especially the traceability in software engineering. Still, they are only an extract of the real world use cases. Therefore, the design of this experiment can be extended to more use cases in the future executions.

The experiment does not provide special requirements for the different groups. Therefore, the participants are randomly assigned to them. However, the participants were not randomly selected for the experiment due to its parameters. They are all master students in *Software Engineering* at the University of Stuttgart. Thus it is hard to generalize their results. Consequentially, it is advisable to execute further experiments with randomly selected developers in order to increase the external validity.

## 5.8 Pilot Experiment and Execution

It is beneficial to test the execution of an experiment with a pilot experiment. This way, it is possible to minimize the risk of a fail during the execution, to examine the assignments, questions, time slots, etc. and to improve or adjust them. The experiment was tested with

one *Software Engineering* student at the University of Stuttgart. During the pilot experiment the student was about to begin his diploma thesis. Thus the educational experience of the student was comparable to the final participants of the experiment.
The pilot experiment revealed several weaknesses in the experiment. Mainly, some questions and assignments were not precisely enough formulated. These questions and assignments were improved.

The final experiment was executed as expected. No issues occurred during the execution. The number of participants was 8. No participant left the experiment before the end.

## 5.9 Results

This chapter lists the results according to the adjusted goals from Chapter 5.4. The results are not interpreted in this chapter. The interpretation and conclusion of the experimental results is described in Chapter 5.10. The assignments are analysed by the number of correct answers. Thus either an answer is fully correct or it is regarded as false. The control questions about Java and Javadoc skills confirmed the minimum requirements of the participants for the experiment and are not discussed further.

### $Q_1$ Will the participants find more required information using UniMoDoc within a given time frame compared to using similar module documentation created with a word processor and Javadoc?

The objective analysis of the first assignment part shows that the participants solved correctly 23 of 40 (58.0%) questions with UniMoDoc and 15 of 40 (38.0%) without UniMoDoc. In other words, the participants solved 53% more questions correctly with UniMoDoc compared to conventional documentation.

After the first assignment part the participants were asked how well they could solve the assignments with UniMoDoc compared to conventional documentation on a scale from 1 to 7. 1 means very much better without UniMoDoc and 7 means very much better with UniMoDoc. The participants answered as follows:

- 25.0% said 7 (very much better with UniMoDoc)
- 37.5% said 6 (much better with UniMoDoc)
- 25.0% said 4 (equally good/bad with and without UniMoDoc)
- 12.5% said 3 (better without UniMoDoc)

In the second assignment part the participants were asked, what type of module documentation they would like to receive in order to maintain and develop existing software. The participants answered:

- 62.5% said that they would like to receive a module documentation created with UniMoDoc
- 36.5% said that they would like to receive a module documentation in a wiki.

### $Q_2$ Will the explicitly defined relations and their visualization help the participants to find the required information using UniMoDoc?

In the first feedback the participants were asked whether they used the visualization of the relations in UniMoDoc. The results show that 100% of the participants used it.

The participants were asked to compare the time effort in the visualization of UniMoDoc with conventional documentation for completing different assignments on a scale from 1 to 5. 1 stands for much slower with the visualization of UniMoDoc and 5 for much faster with the visualization of UniMoDoc compared to the conventional documentation. The answers for different assignments are as follows:

Finding modules, which realize a particular requirement

- 75% said 5 (much faster)
- 25% said 4 (faster)

Detecting whether a requirement is tested or not

- 62.5% said 5 (much faster)
- 25.0% said 4 (faster)
- 12.5% said 3 (equally fast)

Finding the test cases for a particular requirement

- 37.5% said 5 (much faster)
- 50.0% said 4 (faster)
- 12.5% said 3 (equally fast)

Finding the test protocols for a particular test case

- 25.0% said 5 (much faster)
- 25.0% said 4 (faster)
- 50.0% said 3 (equally fast)

Finding the dependencies across modules

- 75.0% said 5 (much faster)
- 12.5% said 4 (faster)
- 12.5% said 3 (equally fast)

Additionally, the participants were asked to mention especially helpful functions in UniMoDoc for solving the assignments as free text. 62.5% of the participants mentioned the visualization of the relations and 25.0% mentioned the explicit relation definitions.

### $Q_3$ Will UniMoDoc crash during the evaluation?

The participants were asked in the first and second feedback about the number of crashes of UniMoDoc. All participants reported no crashes in both feedback forms.

### $Q_4$ Do the participants feel positive about the usability of UniMoDoc and can they accomplish typical tasks without additional instructions?

The participants were asked to describe the usability of UniMoDoc in the first feedback after the first assignment part on a scale from 1 (very user-unfriendly) to 7 (very user-friendly). The participants answered:

- 50.0% said 6
- 12.5% said 5
- 37.5% said 2

The participants were asked to describe the usability of UniMoDoc in the second feedback only according the second assignment part on a scale from 1 (very user-unfriendly) to 7 (very user-friendly). The participants answered:

- 62.5% said 6
- 12.5% said 5
- 25.0% said 3

In the second assignment part the participants had to complete a module documentation using UniMoDoc. The participants received no further instructions besides the assignments and a short getting started sheet. They solved 23 of 24 (96.0%) assignments correctly.

Additionally, based on the second assignment part, the participants were asked, which tools they would prefer to use for documenting modules. They answered as follows:

- 50.0% said UniMoDoc
- 25.0% said wiki
- 12.5% said Javadoc
- 12.5% said UML tool

Furthermore, the participants were asked in free text fields about improvable and missing functions in UniMoDoc. They named several functions in user guidance, e.g., creation of relations, references, sections, etc.

## 5.10 Interpretation of the Results and Conclusion

This chapter interprets the results of the experiment for each question and attempts to find an answer for the questions based on the results. Subsequently, it draws a conclusion about the overall results of the experiment.

#### $Q_1$ Will the participants find more required information using UniMoDoc within a given time frame compared to using similar module documentation created with a word processor and Javadoc?

The objective results of the assignments show that the participants were able to find significantly more (53.0%) asked information with UniMoDoc than with conventional module documentation. This observation is confirmed by the subjective feedback of the participants. The majority (62.5%) of them considered that they were able to solve the assignments better with UniMoDoc compared with conventional module documentation. Finally, 62.5% of the participants would prefer to receive a module documentation created by UniMoDoc. According to the results, the above question can be answered with yes.

#### $Q_2$ Will the explicitly defined relations and their visualization help the participants to find the required information using UniMoDoc?

The results show that the participants productively utilized the visualization of the relations. Furthermore, they demonstrate that the visualization of the relations helped the majority of participants in all types of assignments. 62.5% of the participants named the visualization of the relations and another 25.0% the explicitly defined relations as an especially helpful function for solving the assignments in a free text field. These results allow answering the above question with yes. Additionally, they support the conclusion that especially the concept of explicit relations and their visualization in module documentation are responsible for the advantage of UniMoDoc observed in the results of $Q_1$.

#### $Q_3$ Will UniMoDoc crash during the evaluation?

During the overall experiment no crash of UniMoDoc was reported. Therefore, the question can be answered with a clear no. Together with the unit tests of UniMoDoc these results support the conclusion that UniMoDoc can be seen as stable according the typical assignments processed during the experiment. Furthermore, the stability of UniMoDoc did not influence the results for $Q_1$ and $Q_2$ negatively.

**$Q_4$ Do the participants feel positive about the usability of UniMoDoc and can they accomplish typical tasks without additional instructions?**

According to the results, the majority of the participants felt positive about the usability of UniMoDoc. The ability of documenting the modules with UniMoDoc in the second assignment part was mainly evaluated according the usability. 96.0% of the assignments were completed correctly. Based on the second assignment part, 50% of the participants would prefer to document modules with UniMoDoc. According to these results, the above question can be answered with yes. Thus it can be concluded that the usability did not influence the results for $Q_1$ and $Q_2$ negatively, at least to a large extent.
However, the results of the free text comments in the experiment discovered some weaknesses in the usability of UniMoDoc, which can be improved in the future.

### Conclusion

The overall results of the evaluation can be interpreted as positive and promising. The initial goal to prove the advantages of the concepts implemented in UniMoDoc compared to conventional module documentation is regarded as fulfilled. At the same time the experiment revealed some minor weaknesses in the usability of UniMoDoc. The next step should be repeating the experiment with a larger number of participants, in order to verify the current results and obtain more precisely ones. Furthermore, it is necessary to utilize UniMoDoc in real projects, e.g., in a field experiment, in order to evaluate its usefulness.

# 6 Conclusion and Future Work

This chapter describes briefly the results of this thesis and draws a conclusion. Subsequently it provides a short outlook for future work.

## 6.1 Conclusion

The initial goals of this thesis are: creating a concept for adoptable module documentation tool based on the research of [Kircher, 2012], implementing an operational prototype and evaluating it. Furthermore, this thesis, its concepts, the resulting prototype, and the evaluation are created in a close co-operation with [Casciato, 2013].

The state of the art in Chapter 2 examines the various types of modules and their requirements. Additionally, it argues that the module documentation is ideally placed within the architecture documentation. Thus the scope of this thesis moves to architecture documentation tools. Furthermore, the state of the art chapter reveals the lack of tool support for architecture documentation and shows the issues of the available tools.
Based on the state of the art research, this thesis suggests a technology independent design for a highly extensible architecture documentation tool in Chapter 3. The main specifics of the design are the extensibility with widgets, continuous support of templates, linking of resources inside and outside the documentation, explicitly definable relations and their visualization. Additionally, this thesis suggests a collection of metadata for a general module documentation template.
The suggested design is implemented as a proof of concept. The idiosyncrasies of the implementation, the resulting prototype named UniMoDoc, and the implemented module documentation template are presented in Chapter 4.
The evaluation of UniMoDoc and the concepts behind it is described in Chapter 5. In order to prove the expected effects, a controlled experiment was planned and executed with eight master students of *Software Engineering* at the University of Stuttgart. The experiment compared module documentation created with UniMoDoc to module documentation created with Javadoc and word processors. The results show a significant advantage of UniMoDoc in finding typical information in module documentation, e.g., dependencies, test cases, and requirements. It can be assumed that the observed advantages of UniMoDoc result from its linking capabilities, explicit relation definitions, and their visualization.
Thus the first step of evaluating the concepts has been taken. However, it is necessary to prove the benefits of UniMoDoc in real world utilization, e.g., a field experiment. Furthermore, there is potential for improvement of the prototype according its usability and functions.

## 6.2 Future Work

The future work requires primarily further evaluation of UniMoDoc. Thus it is advisable to repeat the experiment described in this thesis with a larger number of participants in order to obtain more precise results. Furthermore, it is necessary to evaluate UniMoDoc in a real project, e.g., a field experiment.

The implementation part shows an enormous potential for future work. Especially the free text results of the evaluation show how the usability of UniMoDoc can be improved. Due to the tough time schedule UniMoDoc misses some common comfort functions, e.g., undo and redo. Nevertheless, these functions can be added with an acceptable effort. Additionally, the evaluation results reveal that UniMoDoc would benefit from a search function. At the moment UniMoDoc provides no search function at all. The experiment participants were still able to find the required information faster with UniMoDoc than with the conventional module documentation because of the navigation concept, explicitly defined relations and their visualization in UniMoDoc. Probably this advantage can be even improved by a search function with a relatively slight effort.

The widgets provide an explicit extension point in UniMoDoc. Besides the actually shipped widgets (*TextField*, *TextArea*, *Image*, *RelationList*, *HTML*) further widgets can be created. The *HTML* widget provides only very basic formatting options, which are improvable. It is also possible to implement a simple UML widget using the graph library [JGraph, 2013] from relations visualization.

The support of templates in UniMoDoc is one of the central features. However, in a scenario where a requirement template was used $n$ times in a document and the *description* section needs to be renamed in all requirements to *definition*, it is necessary to rename it individually in each of the $n$ requirements. This behavior results from the high flexibility of the structure, which is not bound to a template after it is imported in a document.
Nevertheless, this behavior can be improved by labels. Labels can be used to mark chapters, sections, relations, relation types, and references in a UniMoDoc document. Thus labels can provide grouping and additional semantics for these elements. The above scenario can be solved by a specific label for the *description* section. Consequentially, a user could select and configure all sections with a specific label simultaneously. Furthermore, a synergy effect can be achieved with the labels and a search function.

The documentation created in UniMoDoc can be additionally exported to the source code artifacts with a comparable result to [Kircher, 2012]. This use case can be beneficial if architecture documentation is created with UniMoDoc before the implementation. Then source code artifacts can be directly generated from UniMoDoc with initial documentation.
There is a rudimentary implementation of this function in UniMoDoc. This function was used for the evaluation in order to create Javadoc module documentation with identical content. In this case, the `package-info.java` files with the module documentation were generated from the module documentation in UniMoDoc.

The implementation of this function can be improved by providing additional settings, e.g., the exported attributes. The current implementation uses the names of the sections to identify the attributes, which need to be exported. This approach can be significantly improved by using labels.

A reverse use case to the upper scenario is where UniMoDoc is utilized in order to document an already existing project. In this case, the existing information can be extracted from the source code. UniMoDoc utilizes a parser for Java source code artifacts [Gesser, 2013] already. This feature can be extended to other programming languages.

The currently used approach for saving and loading data in UniMoDoc is a combination of XML and JAXB. The reasons for choosing these technologies are described in Chapter 4. However, negative effects are observed concerning the file size and loading time for large documentations. It is possible to counteract these effects by reducing the file size and changing the used approach. The overall file size can be reduced by compressing the XML structure. There are various compression techniques for XML but it is necessary to consider that the compressed file needs to stay text-based.
The overall loading speed can be improved by a utilizing a simpler format than XML. The design of UniMoDoc allows changing the underlying data format with small effort. Alternatively, the JSON data format can be considered.

# A Appendix

| Name | Description | Use for Architecture Documentation |
|---|---|---|
| Analyst | Responsible for analyzing the architecture to make sure it meets certain critical quality attribute requirements. Analysts are often specialized; for instance, performance analysts, safety analysts, and security analysts may have well-defined positions in a project. | Analyzing satisfaction of quality attribute requirements of the system based on its architecture. |
| Architect | Responsible for the development of the architecture and its documentation. Focus and responsibility is on the system. | Negotiating and making trade-offs among competing requirements and design approaches. A vessel for recording design decisions. Providing evidence that the architecture satisfies its requirements. |
| Business manager | Responsible for the functioning of the business/organizational entity that owns the system. Includes managerial/executive responsibility, responsibility for defining business processes, and more. | Understanding the ability of the architecture to meet business goals. |
| Conformance checker | Responsible for assuring conformance to standards and processes to provide confidence in a product's suitability. | Basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions. |
| Customer | Pays for the system and ensures its delivery. The customer often speaks for or represents the end user, especially in a government acquisition context. | Assuring required functionality and quality will be delivered, gauging progress, estimating cost, and setting expectations for what will be delivered, when, and for how much. |

Continued on next page

| Name | Description | Use for Architecture Documentation |
|---|---|---|
| Database administrator | Involved in many aspects of the data stores, including database design, data analysis, data modeling and optimization, installation of database software, and monitoring and administration of database security. | Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals. |
| Deployer | Responsible for accepting the completed system from the development effort and deploying it, making it operational, and fulfilling its allocated business function. | Understanding the architectural elements that are delivered and to be installed at the customer's or end user's site, and their overall responsibility toward system function. |
| Designer | Responsible for systems and/or software design downstream of the architecture, applying the architecture to meet specific requirements of the parts for which they are responsible. | Resolving resource contention and establishing performance and other kinds of runtime resource consumption budgets. Understanding how their part will communicate and interact with other parts of the system. |
| Evaluator | Responsible for conducting a formal evaluation of the architecture (and its documentation) against some clearly defined criteria. | Evaluating the architecture's ability to deliver required behavior and quality attributes. |
| Implementer | Responsible for the development of specific elements according to designs, requirements, and the architecture. | Understanding inviolable constraints and exploitable freedoms on development activities. |
| Integrator | Responsible for taking individual components and integrating them, according to the architecture and system designs. | Producing integration plans and procedures, and locating the source of integration failures. |
| Maintainer | Responsible for fixing bugs and providing enhancements to the system throughout its life (including adaptation of the system for uses not originally envisioned). | Understanding the ramifications of a change. |

Continued on next page

| **Name** | **Description** | **Use for Architecture Documentation** |
|---|---|---|
| Network administrator | Responsible for the maintenance and oversight of computer hardware and software in a computer network. This may include the deployment, configuration, maintenance, and monitoring of network components. | Determining network loads during various use profiles and understanding uses of the network. |
| Product line manager | Responsible for development of an entire family of products, all built using the same core assets (including the architecture). | Determining whether a potential new member of a product family is in or out of scope and, if out, by how much. |
| Project manager | Responsible for planning, sequencing, scheduling, and allocating resources to develop software components and deliver components to integration and test activities. | Helping to set budget and schedule, gauging progress against established budget and schedule, and identifying and resolving development-time resource contention. |
| Representative of external systems | Responsible for managing a system with which this one must interoperate, and its interface with our system. | Defining the set of agreement between the systems. |
| System engineer | Responsible for design and development of systems or system components in which software plays a role. | Assuring that the system environment provided for the software is sufficient. |
| Tester | Responsible for the (independent) test and verification of the system or its elements against the formal requirements and the architecture. | Creating tests based on the behavior and interaction of the software elements. |
| User | The actual end users of the system. There may be distinct kinds of users, such as administrators, superusers, and so on. | Users, in the role of reviewers, might rely on architecture documentation to check whether desired functionality is being delivered. Users might also refer to the documentation to understand what the major system elements are, which can aid them in emergency field maintenance. |

**Table A.1:** Stakeholders of software architecture documentation from [Garlan et al., 2010]

# Bibliography

[Adobe, 2013] Adobe (2013). *Adobe Acrobat Family.* Available from: http://www.adobe.com/products/acrobat.html. (Cited on page 34)

[Apache Felix, 2013] Apache Felix (2013). *Apache Felix.* Available from: http://felix.apache.org/. (Cited on page 17)

[Apple, 2013] Apple (2013). *Mac Basics: The Finder.* Available from: http://support.apple.com/kb/ht2470. (Cited on page 33)

[arc42, 2013] arc42 (2013). *Free Portal for Software Architects.* Available from: http://www.arc42.de. (Cited on page 26)

[Buckley et al., 2006] Buckley, A., Bock, D. W., Halloway, S., Hall, R. S., Kriens, P., Lea, D., Leuck, D., Pullara, S., and Neward, T. (2006). *JSR 277: Java Module System.* Available from: http://jcp.org/en/jsr/detail?id=277. (Cited on page 20)

[Buckley et al., 2007] Buckley, A., Hall, R. S., Kriens, P., Lea, D., Leuck, D., and Pullara, S. (2007). *JSR 294: Improved Modularity Support in the JavaTM Programming Language.* Available from: http://jcp.org/en/jsr/detail?id=294. (Cited on page 20)

[Casciato, 2013] Casciato, D. (2013). *Verwaltung von Testinformationen in der Moduldokumentation.* Diploma, University of Stuttgart. (Cited on pages 12, 31, 37, 39, 45, 56, 59, 61, 65, 67 and 81)

[Eclipse Foundation, 2013a] Eclipse Foundation (2013a). *Eclipse.* Available from: http://www.eclipse.org/. (Cited on page 34)

[Eclipse Foundation, 2013b] Eclipse Foundation (2013b). *Eclipse Equinox.* Available from: http://www.eclipse.org/equinox/. (Cited on page 17)

[Ellison, 2013] Ellison, T. (2013). *Project Penrose.* Available from: http://openjdk.java.net/projects/penrose/. (Cited on page 22)

[FEA, 2013] FEA (2013). *Federal Enterprise Architecture.* Available from: http://www.whitehouse.gov/omb/e-gov/fea. (Cited on page 26)

[FMC, 2013] FMC (2013). *Fundamental Modeling Concepts.* Available from: http://www.fmc-modeling.com/. (Cited on page 26)

[Foxit Corporation, 2013] Foxit Corporation (2013). *Foxit Reader.* Available from: http://www.foxitsoftware.com/Secure_PDF_Reader/. (Cited on page 34)

[Garlan et al., 2010] Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P., and Merson, P. (2010). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition. (Cited on pages 9, 17, 24, 25, 26, 44, 45 and 87)

[Gesser, 2013] Gesser, J. (2013). *Java 1.5 Parser*. Available from: https://code.google.com/p/javaparser/. (Cited on page 83)

[Hall et al., 2011] Hall, R., Pauls, K., McCulloch, S., and Savage, D. (2011). *OSGi in Action: Creating Modular Applications in Java*. Manning Pubs Co Series. Manning Publications Company. (Cited on page 17)

[IEEE-1471, 2000] IEEE-1471 (2000). *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. *IEEE Std 1471-2000*, pages i–23. Available from: http://ieeexplore.ieee.org/servlet/opac?punumber=7040. (Cited on pages 24 and 26)

[IEEE-610.12, 1990] IEEE-610.12 (1990). *IEEE Standard Glossary of Software Engineering Terminology*. *IEEE Std 610.12-1990*, pages 1–84. (Cited on pages 15, 16 and 23)

[ISO/IEC/IEEE, 2010] ISO/IEC/IEEE (2010). *Systems and software engineering – Vocabulary*. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418. (Cited on page 16)

[J-PaD, 2013] J-PaD (2013). *Java Package Documentation*. Available from: http://www.j-pad.de/. (Cited on pages 12, 27, 30, 43, 44, 45 and 46)

[Jedlitschka and Pfahl, 2005] Jedlitschka, A. and Pfahl, D. (2005). *Reporting guidelines for controlled experiments in software engineering*. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–. (Cited on page 65)

[JGraph, 2013] JGraph (2013). *Java Graph Drawing Component*. Available from: http://www.jgraph.com/. (Cited on page 82)

[Kircher, 2012] Kircher, M. (2012). *Integrated Documentation for Software Modules*. Diploma, University of Stuttgart. (Cited on pages 8, 12, 27, 28, 81 and 82)

[Knöpfel et al., 2005] Knöpfel, A., Gröne, B., and Tabeling, P. (2005). *Fundamental modeling concepts: effective communication of IT systems*. J. Wiley & Sons. (Cited on page 26)

[Knopflerfish, 2013] Knopflerfish (2013). *Knopflerfish - Open Source OSGi Service Platform*. Available from: http://www.knopflerfish.org/. (Cited on page 17)

[Ludewig and Lichter, 2007] Ludewig, J. and Lichter, H. (2007). *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag. (Cited on pages 8, 16, 17 and 28)

[Microsoft, 2013a] Microsoft (2013a). *How to work with files and folders*. Available from: http://windows.microsoft.com/en-us/windows-8/files-folders-windows-explorer. (Cited on page 33)

[Microsoft, 2013b] Microsoft (2013b). *Microsoft Visual Studio*. Available from: http://www.microsoft.com/visualstudio. (Cited on page 34)

[MODAF, 2013] MODAF (2013). *Ministry of Defense Architecture Framework.* Available from: http://www.modaf.org.uk/. (Cited on page 26)

[OpenJDK, 2013] OpenJDK (2013). *Project Jigsaw.* Available from: http://openjdk.java.net/projects/jigsaw/. (Cited on page 21)

[OSGi, 2013] OSGi (2013). *OSGi Alliance.* Available from: http://www.osgi.org. (Cited on page 17)

[Parnas, 1972] Parnas, D. L. (1972). *On the criteria to be used in decomposing systems into modules. Commun. ACM*, 15(12):1053–1058. Available from: http://doi.acm.org/10.1145/361598.361623. (Cited on pages 11, 15 and 44)

[Prechelt, 2001] Prechelt, L. (2001). *Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik.* Springer-Verlag GmbH. (Cited on pages 65, 68, 72 and 74)

[Reinhold, 2011] Reinhold, M. (2011). *Project Jigsaw: The Big Picture - DRAFT 1.* pages 1–418. Available from: http://cr.openjdk.java.net/~mr/jigsaw/notes/jigsaw-big-picture-01. (Cited on page 21)

[Reinhold, 2012] Reinhold, M. (2012). *Project Jigsaw: On the next train.* Available from: http://mreinhold.org/blog/on-the-next-train. (Cited on page 21)

[RM/ODP, 2013] RM/ODP (2013). *Reference Model for Open Distributed Processing.* Available from: http://www.rm-odp.net/. (Cited on page 26)

[SEI, 2013] SEI (2013). *Software Architecture Overview.* Available from: http://www.sei.cmu.edu/architecture. (Cited on page 24)

[Starke and Hruschka, 2011] Starke, G. and Hruschka, P. (2011). *Software-Architektur Kompakt: Angemessen Und Zielorientiert.* IT kompakt. Spektrum Akademischer Verlag GmbH. (Cited on pages 26, 43, 44, 45 and 46)

[Stefanov, 2010] Stefanov, S. (2010). *JavaScript Patterns.* O'Reilly Media. (Cited on page 22)

[TOGAF, 2013] TOGAF (2013). *The OpenGroup Architecture Framework.* Available from: http://www.opengroup.org/. (Cited on page 26)

All links were last followed on August 26, 2013.

**Decleration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature