Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

Design and Development of a Standards-Based Authoring Framework for Software Requirements Specifications

Kariem Hussein

Design and Development of a Standards-Based Authoring Framework for Software Requirements Specifications

Kariem Hussein

Abstract

In the constantly evolving software industry it is of vital importance for a company to document their intentions in the form of requirements documentation, and to communicate these requirements internally and together with their customers. In order to put down the requirements, authors currently mostly use word processors and a small set of tools to generate graphics. This combination provides very few features that are important to effectively create, review, and maintain a software requirements specification, such as transparent versioning, automated document sharing, and extensible integration points with external systems.

The authoring framework presented in this thesis provides improvements at all these levels, and, due to the standards-based concept, integration points for existing tools. This thesis shows how the different parts of the framework's components — the documentation format, the authoring tool, and the repository — can play together to create an environment for efficient document authoring.

Existing solutions (i.e. implementations) for the documentation format and authoring tool are analyzed and compared with one another. For both components, this thesis collects a set of criteria that can be used to assess how useful the solution may be for authoring processes. The results are taken into consideration for the eventually presented architecture and implementation of the authoring framework.

Keywords:
Document Authoring, *Documentation Maintenance*, *Knowledge Sharing*, *Requirements Analysis*, *Software Engineering*, *Software Requirements Specification*, *Standards-based Documentation*, *Versioning*

Kurzfassung

Da sich die Softwareindustrie immer noch ständig weiter entwickelt, ist es für Firmen nötig, ihre Absichten in der Form von Anforderungsdokumentation niederzuschreiben und diese Anforderungen sowohl intern, als auch mit ihren Kunden zu kommunizieren. Beim Verfassen der Anforderungsdokumentation verwenden Autoren heutzutage hauptsächlich Textverarbeitungsprogramme, sowie zusätzliche Software zur Erstellung von Grafiken. Diese Kombination von Werkzeugen bietet nur sehr wenige der Eigenschaften, die für die Erstellung, Prüfung, oder Wartung von solcher Dokumentation nötig ist, wie etwa transparente Versionierung, automatisierte Publizierung und erweiterbare Schnittstellen zur Verbindung von externen Systemen.

Das in dieser Arbeit vorgestelltle Authoring Framework bietet Verbesserungen auf all den angesprochenen Ebenen und, aufgrund des Standars-basierten Konzeptes, Integrationspunkte für bestehende Software. Diese Arbeit zeigt wie die einzelnen Teile der Komponenten des Frameworks — das Dokumentenformat, das Autorensystem, sowie das Repository — zusammen wirken, um eine Umgebung für die effiziente Erstellung von Dokumenten zu schaffen.

Bereits bestehende Lösungen für das Dokumentenformat und das Autorensystem werden analysiert und gegenübergestellt. Für beide Komponenten beschreibt diese Arbeit Kriterien, die dafür verwendet werden können, Lösungen für den Einsatz zur Dokumentationserstellung zu bewerten. Die daraus resultierenden Ergebnisse werden bei der Architektur und beim design des Authoring Frameworks eingesetzt.

Schlüsselwörter:
Anforderungsanalyse, *Dokumentationserstellung*, *Dokumentationswartung*, *Software Anforderungsspezifikation*, *Softwaretechnik*, *Standards-basierte Dokumentation*, *Versionierung*, *Wissensweitergabe*

Dedication

I have to thank my parents who always supported me and helped where they found a need. Special thanks go to my brother who not only has been escorting and watched over me my whole life, but also keeps asking discerning questions that always bring a better view into all of my work.

Contents

Introduction ... xi
1 Motivation ... xi
2 Importance ... xi
3 Today's Reality ... xii
4 Outline ... xii
I Requirements Documentation Authoring ... 1
1 Requirements – Terminology and Definitions ... 3
1.1 Software Product Lifecycle ... 3
1.2 Basic Definitions ... 4
1.3 Requirements Specifications ... 7
1.4 Requirements Documentation ... 11
1.5 Documentation Types ... 14
1.6 Software Development Models ... 16
2 Goals and Evaluation of Requirements Authoring ... 23
2.1 Goals of Document Authoring for Requirements Documentation ... 23
2.2 Evaluation with Maturity Models ... 27
2.3 Exemplary Industrial Processes ... 30
II Documentation Formats and Authoring Tools ... 35
3 Analysis of Documentation Formats ... 37
3.1 Restrictions ... 37
3.2 Traditional Document Formats ... 44
3.3 Current Document Formats and Standards ... 47
4 Analysis of XML-Based Authoring Tools ... 57
4.1 Capabilities of an Authoring Tool for Requirements Documentation ... 57
4.2 Commercial Authoring Tools ... 59
4.3 Independent Projects ... 60
4.4 Summary ... 63
III Modular Integration Concept ... 65
5 Concept of an Authoring Framework ... 67
5.1 Goals of the Authoring Framework ... 67
5.2 Component Overview ... 67
5.3 Component Architecture ... 68
5.4 Design Discussion ... 77
6 Developing the Authoring Framework ... 81
6.1 Focus ... 81
6.2 Documentation Format ... 82
6.3 Document Viewer ... 87
6.4 Repository ... 91
7 Validation and Conclusion ... 95
7.1 Validation ... 95
7.2 Related Work ... 98
7.3 Conclusion ... 99
References ... 101

List of Figures

1.1 Product Lifecycle with UCD Process Incorporated 4
1.2 Sources Of Software Requirements 7
1.3 Evolution of a Behavior Specification 8
1.4 Application Description as Input for the Synchronized Refinement Method 12
1.5 Stakeholder Documentation 15
1.6 Scrum Skeleton 20
2.1 Levels of the System Documentation Process Maturity Model 28
2.2 Transformation of Requirements 32
2.3 Documentation Handover 33
3.1 Simple WordML document displayed in Microsoft Word 45
3.2 DocBook Publishing Model with XSLT 50
4.1 Vex displaying DocBook Meta-Information. 62
4.2 Etna displaying Information from a Tinydoc File 63
5.1 Components of the Authoring Framework 68
5.2 Content Handlers 73
5.3 Components of the Authoring Tool 74
5.4 Annotations from Comparison or Meta-Information 76
5.5 Annotated View 76
5.6 Meta-Format as Wrapper around Document Structure 79
6.1 Document Viewer Providing an Interface for Lightweight Clients 87
6.2 Create Change Annotations from Comparison 89
6.3 Integration Scenario for Repository 93
7.1 Use Case Creation Process with Transformation Designer 100

List of Tables

1.1 Documentation Types for the Authoring Tool 16
1.2 Home Ground for Agile and Plan-driven Methods 17
2.1 Documentation Maturity Levels associated with KPAs 29
3.1 Useful Documentation Technologies 37
3.2 Exemplary Requirements and Dependencies of a Word Processor 40
3.3 Overview of Current Documentation Formats 56
4.1 Requirements for the Authoring Tool 58
4.2 Comparison of Supported Requirements in Authoring Tools 63
5.1 Documentation Maturity for the Authoring Tool 69
5.2 Stored Information per Documentation Type 69
7.1 Validation Scenarios 95

List of Examples

3.1 Word Processor Version Incompatibility 40
3.2 Simple WordML document 45
3.3 Minimal LaTeX Input File 46
3.4 Meta-Information on a Book in DocBook Syntax 49
3.5 Simple Paragraph using the DocBook Syntax 51
3.6 Stylesheet using CSS 51
3.7 Stylesheet using XSL 52
3.8 Relationships for an Excel Worksheet 55
4.1 CSS Snippet To Render Book Meta-Information in Vex 61
6.1 Complex Type for Text 83
6.2 Complex Type for Precondition 83
6.3 Formatted Postcondition 84
6.4 Primary Scenario in Two Steps 84
6.5 Description with Two Alternatives 84
6.6 Definition of Scenario with Alternatives 85
6.7 Structure of a Use Case Document 86
6.8 Use Case xml:id Attribute 86
6.9 Spring Configuration for XmlUnitDiffer 90
6.10 Single Entry for Revision 10 in the 'Source Changes' Feed 91

Introduction

This thesis presents current documentation practices and tools that are used to build software requirements documentation. It is written in the context of an industry where good requirements documentation is essential and changing requirements are standard. A presentation of modern document formats and tools for technical documentation yields the characteristics for an optimal authoring framework, which is conceptualized and partly implemented towards the end of this thesis.

1. Motivation

I will start with a review of my motivations to choose this specific topic. The following areas of application show realistic problems I had to face while working with requirements documentation

Documentation Maintenance. During my work at the department as a student advisor for the course *software maintenance and evolution* I had to prepare an existing requirements document for a class. The document, written using Microsoft Word (more on this format in Section 3.2.1, “Word Document Format”), described 30 to 40 use cases. At the end of the first class assignment, I incorporated changes from 15 different student groups into this document. This task showed how inflexible and error-prone requirements documents are.

Evolving Documentation. My work as an engineer at a large international telecommunications company involved highly technical solutions to business critical requirements. Together with my team, we had to permanently communicate ever-changing requirements and associated management decisions. This only aside from implementing the requirements in a disciplined manner into a product used internally in 7 different european countries. The process of validating the current implementation, or even finding a reason for some design decisions is at least lengthy, but most of the time a discouraging task.

Documentation Effort. Currently, I work as independent consultant for different customers in IT-related projects. I have seen many cumbersome environments that are held together by hard-working individuals. A lot of energy is wasted just because the choice of tools is bad, and often based on general features instead of requirements in regard to day-to-day work. Generic tools, such as word processors, are licensed in bulk, although only a very limited set of features is actually used. For the tasks at hands, the employed solutions only provide little support.

Without scientific argumentation it is not easy to convince people that are satisfied with their often operose environments to rethink their existing solutions.

2. Importance

Good documentation is important to different phases of the software process, and an important software product. However, not everyone is aware of how important documentation is.

Visconti and Cook [Visc93] present a number of figures that show the importance of good documentation in software maintenance:

- Probably being most crucial to the maintenance phase, which accounts for 60-75 percent of the total cost of the software, documentation accounts for more than 60 percent of maintenance costs. It is involved in about one third of the maintenance tasks.
- A quick understanding of existing software is a key activity in the maintenance process. People in this process spend 40 percent of their time dealing with documentation.
- When making a program modification, 47 percent of a maintenance programer's time is spent studying the program source code and associated documentation. When errors are corrected, this time increases to 62 percent.

3. Today's Reality

It is not my intention to explain anyone the reality[1] he/she faces all day long in the industry. However, I want to emphasize that people know what they should do, but consciously act differently because they are forced to do so. Even if a person knows that quick fixes, dirty hacks or undocumented changes should be avoided, it is sometimes necessary to ignore this, because project plans, test schedules and delivery timelines demand for a fast solution. The problem is usually that a fast solution might not be a good solution.

In [Ruga00] the following possible reasons are mentioned to explain the situation:

- A program typically solves a specific problem, but the model it assumes is much broader. A transactional workflow program can find its application in a wide variety of environments and solution domains. If the program provides transaction based services with good support for transactions, high customizability and a good level of abstraction, it is reasonable to leverage these advantages in a wide array of domains.
- Programs often do not exist in isolation. A set of programs jointly solves a collection of related programs. For example, to enable the telephony service for a single customer, application A has to deal with creating a line on the switch, while application B creates a voice mail account. Application C is responsible for creating a special file with information for the telephone device at the customer's premises. Billing-relevant operations have been ignored.

There were several occasions where the program was changed without appropriate adaptations of the documentation. All the changes were business relevant and either missed prior documentation or a subsequent update of the documentation. The essential documentation was not written, because it would have taken too much time. The lack of documentation resulted in additional effort needed to explain changes to the user, or to find reasons for changes some weeks thereafter. Undocumented behavior is even more dangerous when solutions are handed over between different developers, teams or companies.

If it had been easy to add the appropriate paragraph to the documentation, it would have been done, but the documentation process and the technologies would have made the udpate a lengthy task. Additional lines in diagrams and tables would have had to be updated to document the current implementation. Marsh [Mars99] says that for engineers “it is no longer enough to write clearly and succinctly; they must also be able to lay out documents to spec in the formats required by the contract, the professional society, or in-house guidelines”.

4. Outline

The result of this thesis is a concept for an authoring framework which can be used in the the documentation of requirements. It focuses specifically on the documentation of software requirements targeted to multiple customers, as opposed to targeted for a vertical market: requirements change often, are very specific, and mostly externally driven.

I will start with a general definition of terms and goals for document authoring. After observing current documentation formats and authoring tools, this document will concentrate on the architecture and the design of the framework. The remainder is organized as follows:

Chapter 1, *Requirements – Terminology and Definitions*
: presents requirements and their documentation in the context of a product's lifecycle. It introduces the reader to the common vocabulary of this document and highlights the position of software requirements in software design and engineering methods.

[1] I took the liberty of using the title of the introduction of [Mars99] for this section.

Chapter 2, *Goals and Evaluation of Requirements Authoring*
: shows the main goals of document authoring and lays out the reference terminology for the remainder of this thesis. The second part of this chapter describes formal ways to evaluate documentation on the basis of a maturity model. The chapter closes with a presentation of documentation processes employed in the industry.

Chapter 3, *Analysis of Documentation Formats*
: presents traditional and current documentation formats and compares structured and loose authoring. With a focus on technological aspects, this chapter provides basic information on different alternatives.

Chapter 4, *Analysis of XML-Based Authoring Tools*
: defines technical requirements for authoring software in the requirements context. It continues with a technical presentation of general authoring software for XML-based documents and compares their features and capabilities according to the requirements.

Chapter 5, *Concept of an Authoring Framework*
: summarizes findings of this thesis and presents an architecture for the requirements authoring framework consisting of three components: documentation format, authoring tool, repository. The relation of these three components and design decisions that are necessary to understand the concept are discussed.

Chapter 6, *Developing the Authoring Framework*
: contains the results of an implementation of the concepts described in the previous chapter. Based on a minimal model assumption, all components are implemented in a very reduced manner to showcase standards-based possibilities for integration. This chapter also goes into details on some of the decisions taken during implementation.

Chapter 7, *Validation and Conclusion*
: concludes the thesis by putting current scenarios and the standards-based scenario resulting from this thesis into a comparison, which shows how different tasks could be solved in each scenario. This chapter also contains steps to validate the findings of this thesis.

Part I. Requirements Documentation Authoring

In this first part of the thesis, software requirements and their specification are defined, their relation to software development in general, and their role in a set of selected software development models.

After that, the requirements specific to document authoring and related processes are observed. These goals are used later in this thesis to identify the requirements of the authoring framework. After looking at documentation approaches used in the industry, maturity models for document authoring are presented on the level of the involved process and the resulting documentation.

- Chapter 1, *Requirements – Terminology and Definitions*
- Chapter 2, *Goals and Evaluation of Requirements Authoring*

Chapter 1. Requirements – Terminology and Definitions

At first sight, it seems to be a simple task to define what a certain software has to do. For most obvious needs, a small sentence describing the required result may be sufficient. On the other side, it is easy to underestimate the complexity of larger systems and their implementation.

I will start with a view on the product's lifecycle and try to find out what requirements are and where they come from. As soon as requirements have materialized, they have to be documented and can then be used by different stakeholders for further reference or elaboration. For a more practical view on how requirements affect product development, this chapter closes with a presentation of concrete software development models and their alignment with requirements engineering.

1.1. Software Product Lifecycle

Software products are developed after a certain amount of time. Maciaszek [Maci05] says that “software development follows a lifecycle”, which is “an orderly set of activities”. He identifies the following elements that are defined by a software lifecycle

Elements of of a Software Lifecycle

- The applied modelling approach
- The exact phases along which the software product is transformed
- The methodology and associated development process

According to Maciaszek, a typical product lifecycle starts with a business *analysis* of the current situation and the proposed solution. The analysis is subjected to a more detailed *design* which is followed by the *implementation*. After *integration* and *deployment* at the customer's site, the system is *operational* and undergoes *maintenance* tasks. These phases are usually sequentially ordered, as shown in Typical Phases in a Software Lifecycle.

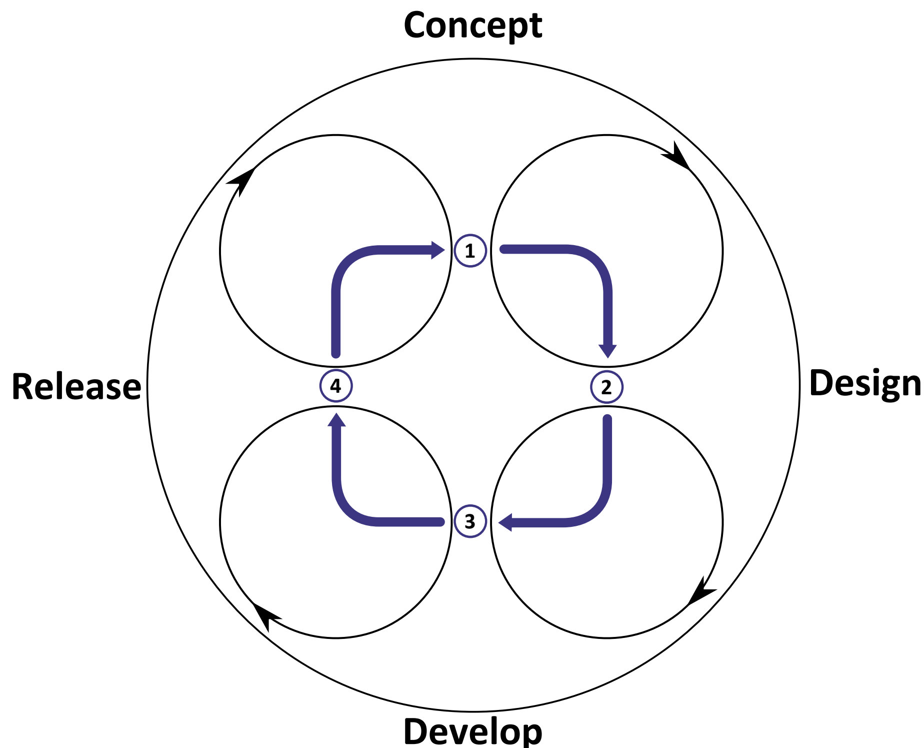
Typical Phases in a Software Lifecycle

1. Business Analysis
2. System Design
3. Implementation
4. Integration and Deployment
5. Operation and Maintenance

I will discuss different software development models and their methodologies and phases in Section 1.6, “Software Development Models”.

For now, I will use a very general software product lifecycle presented by Courage and Baxter [Cour04], depicted in Figure 1.1, “Product Lifecycle with UCD Process Incorporated”. This lifecycle follows a user-centered design (UCD) philosophy, and gives extensive attention to the end users' needs, wants and limitations. It is incremental and iterative, and I will use it as an example implementation of Maciaszek's typical lifecycle and have a look at the results of each phase.

Figure 1.1. Product Lifecycle with UCD Process Incorporated



An incremental and iterative product lifecycle by Courage and Baxter [Cour04]

During the *concept* (analysis) phase, the current situation analyzed and, based on a competitive analysis requirements and a functional specification of the product are defined. In addition, this phase usually results in a UI development plan and develops persona (roles) which are used in the solution. Courage and Baxter call this phase the “idea phase of the product”.

The concept phase is followed by the *design*, where the “information collected in [the concept phase is used] to create iterative designs.” During this phase, detailed information architecture and user interface prototypes are developed. In addition, the taskflows (use cases) are created.

During the *develop* (implementation) phase, “developers or engineers [...] create the product”. According to Maciaszek, this involves installation/adaptation of already available software and creation (coding) of new software. It also implicates loading of databases, testing and user training. He distinguishes analysis, design and implementation in the following manner:

> Business analysis is about what to do, system design is how to do it using the available technology, and implementation is doing it.

Courage and Baxter's incremental product lifecycle concludes with the *release* phase, in which the “product is released to the public or customer”. This phase comprises final tests and the operational rollout. Because this lifecycle is not only incremental (phases are sequential), but also iterative, the requirements collection for the next iteration may start already during this phase. After the *release* phase, the next iteration is started with *analysis*.

1.2. Basic Definitions

I will start with basic definitions used throughout this document.

- Problem and Solution Domain
- Software Requirements

- Roles in the Requirements Process

1.2.1. Domains

Leffingwell [Leff03] distinguishes the problem and the solution domain:

Problem Domain
: In the problem domain there are business or technical problems that have to be solved. In this space there are real users and other stakeholders whose needs must be addressed.

Solution Domain
: In this domain the focus lies in defining solutions to the problems. A *feature* is a service provided by the system that fulfills customer needs. After agreement on the features, more specific software requirements can be defined. Software requirements drive the system's design and implementation.

1.2.2. Software Requirements

Throughout this document the term *requirement* is used as a synonym for *software requirement*. Leffingwell [Leff03] defines a software requirement as:

1. A software capability needed by the user to solve a problem or achieve an objective
2. A software capability that must be met or possessed by a system (component) to satisfy a contract, standard, specification or other formally imposed documentaton.

Item 1 corresponds to *user requirements* in a narrow sense. Courage and Baxter [Cour04] distinguish between business, marketing/sales, and user requirements in the following manner:

Business requirements tend to be high-level and/or technical, and are typically expressed by corporate professionals or executives. These requirements often reflect the current business practices of their company or practices that are being adopted by the company.

Marketing and sales requirements reflect the goals of their proper department who want to ensure that the product sells. They contain requests for features that customers may want or features that mean an advantage over competition. A marketers requirement tends to be at a higher level rather than detailed and tries to address potential customers, while requirements of a sales department may be very customer-specific.

User requirements often overlap with the requirements from sales and marketing, but they not necessarily have to. In order to collect user requirements, it is necessary to gain an understanding of their perspective and needs, including their tasks, goals, context of use and skills. Courage and Baxter claim that the number one mistake is to “think you understand what the end users want and need because other sources have told you on their behalf”. In [Cour04] they describe in detail many practical methods to identify user requirements.

Leffingwell's second type of software capability (item 2) refers to formally documented abilities of the resulting software system. The two types of capabilities are not mutually exclusive. It is often the case, that formally documented reuqirements accrue from user requirements.

1.2.3. Roles in the Requirements Process

In this thesis, I distinguish between active and passive parts or roles that are assigned to or taken by individuals or groups during the requirements process.

Passive Roles

The following roles are defined in the context of software requirements specifications [Stan98]. As they comprise more responsibilities than activities they will be classified as passive roles.

Customer
: The person, or persons, who pay for the product and usually (but not necessarily) make decisions about the requirements. In the context of this recommended practice the customer and the supplier may be members of the same organization.

Supplier
: The person, or persons, who produce a product for a customer. In the context of this recommended practice, the customer and the supplier may be members of the same organization.

User
: The person, or persons, who operate or interact directly with the product. The user(s) and the customer(s) are often not the same person(s).

For a complete picture, the term *contract* denotes a legally binding document agreed upon by the customer and supplier. This includes the technical and organizational requirements, cost, and schedule for a product. A contract may also contain informal but useful information such as the commitments or expectations of the parties involved.

Active Roles

In the context of this thesis, a person or group that is an active part of the product development, does one or more of the following items:

- Transform needs from the problem domain into requirements in the solution domain.
- Fulfill requirements by introducing solutions (on a technical, organisational, or process level)

Courage and Baxter [Cour04] use the term *product development team* (refer to Figure 1.2, “Sources Of Software Requirements” for a depiction). This team consists of a number of people, usually with complementary skills, who are committed to the common purpose of delivering software that fulfills specific requirements. The team is usually a heterogeneous group with different background that drives product development from the concept to the release phase.

Figure 1.2. Sources Of Software Requirements

Requirements of a software product usually come from different sources (image based on [Wieg04] and [Cour04])

In this thesis, the role *author* refers to the writer of the software requirements. For this thesis, the author denotes the central role of the product development team, and may be affiliated with either customer or supplier. This person writes the requirements and is responsible for the initial distribution of the assembled information. It is often the case that authors also incorporate results of reviews or additional information into the requirements and are responsible for the requirements even after the software has been delivered.

Even if the person that updates requirements is different from the initial author, both persons are called authors of the resulting requirements document.

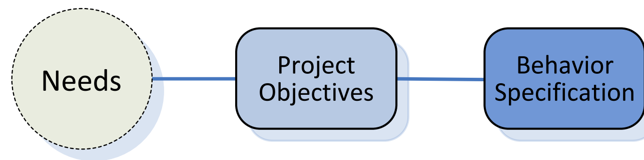
1.3. Requirements Specifications

According to IEEE Std 830-1998 [Stan98], in an ideal model the results of the software requirements specification (SRS) process is an unambiguous and complete specification document. The recommended practice describes the two groups that benefit from a SRS:

- Software customers to accurately describe what they wish to obtain.
- Software suppliers to understand exactly what the customer wants.

Software requirements are a behavioral specification of the target system (see Section 1.3.3, “Classification” for a more structured classification). The specification defines what the software system should be capable of and is based on previously defined project requirements. Requirements specified therein contain the objectives of the project around the software. These objectives accrue from specific needs. See Figure 1.3, “Evolution of a Behavior Specification” for a visual hierarchy

Figure 1.3. Evolution of a Behavior Specification



***Project objectives* (requirements) arise from certain needs. Formally they are the basis for the *behavior specification*.**

A requirements specification consists of [Wier95]

1. a specification of product objectives,
2. a specification of required product behavior.

Bell and Thayer [Bell76] discuss two different definitions ("schools of thoughts") for software requirements. The first definition (A) is being extended by a more detailed version (B) that accrues from the findings in that paper:

Definition A
: Software requirements arise naturally and are correct by definition. If these requirements state a basic need, then that's all that is needed. On the other hand, if the requirements state each subroutine's detailed characteristics, then those are the required characteristics, and the implementer should not question them.

Definition B
: The requirements describe functions that the software must perform, but not how they must be implemented. Not all the various levels of requirements are needed, but "only a single one, one that can usually be identified in large software development projects [...]"

1.3.1. Characteristics

A good SRS [Stan98] has the following characteristics:

Correct
: An SRS is correct, if every requirement stated therein is one that the software shall meet.

Unambiguous
: Every requirement stated therein has only one interpretation. Requirement specification languages, such as Z used in [Crox05], avoid the ambiguity inherent in natural languages, but the length of time required to learn such a language and possible influences on the requirements are strong disadvantages.

Complete
: An SRS is complete, if it includes the following elements:

  1. all significant requirements
  2. definitions of the responses of the software all realizable classes of input data (valid and invalid)
  3. references to all figures, tables and diagrams in the SRS and definition of all terms and units of measure

Consistent
: The specification shows internal consistency, if no subset of individual requirements described in it conflict.

Ranked for importance and/or stability
: As requirements are not equally important or stable, each requirement should have an identifier to indicate the importance or stability of that particular requirement.

Verifiable
: If every requirement in an SRS is verifiable, the SRS is verifiable. A requirement is only verifiable, if there exists a process to check that the resulting software product meets the requirement.

Modifiable
: Changes to requirements can be made easily, completely and consistently while retaining the structure and style. In literature this characteristic is also referred to as maintainable.

Traceable
: The origin of each of its requirements is clear and facilitates the referencing of each requirement in future development or secondary documentation.

Other sources (e.g. [Kovi98] [Laue02]) provide similar classifications for requirements.

Weak Characteristics

Other characteristics are often described as being very important to users of the requirements documentation, although they are not easily measurable. These characteristics are more related to the goals of a good SRS that are described in Section 2.1, “Goals of Document Authoring for Requirements Documentation”. They are called *weak* characteristics. Despite the name, these characteristics are important for the development of the authoring framework, because user reception is essential for a process or framework to be deployed.

According to Wieringa [Wier95], the most important property of a requirements specification is *Communicability*. It is necessary that the specification is understandable, because it “should serve as a channel of communication about the product”.

1.3.2. Content

Although providing a lot of important information, there are many aspects that a SRS should *not* contain. Wieringa [Wier95] is very clear on this subject, and says that the specification should be implementation independent and should “describe the requirements and nothing else”. Leffingwell [Leff03] explicitly excludes information necessary for managing the project (e.g. schedule, budget, test or acceptance procedures), and says that this information should not be included in the software requirements.

Design-Related Information

Croxford and Chapman [Crox05] claim that, in order to avoid the introduction of defects, and to improve error detection, it is important to produce “a software specification that says what the software will do and a design that says how it will be structured. The design does not repeat any information in the specification […]”.

Depending on the type of system to be developed, design constraints should be part of the SRS. In IEEE Std 830-1998 [Stan98], specification writers are encouraged to “distinguish between identifying required design constraints and projecting a specific design. […] The SRS should specify what functions are to be performed on what data to produce what results at what location for whom”. In detail, the following design information should not be a part of the requirements :

- Partition of the software into modules
- Allocation of functions to the modules
- Description of the flow of information or control between modules

- Choice of data structure

Recommendations

Rupp [Rupp06] presents a list of different recommendations for contents of software requirements specifications. He refers, among others, to the following and compares their field of application.

- V-Model
- IEEE 830-1998, which were already referenced previously

V Model

The V Model, in the current version called *V Model XT* (Extreme Tailoring), has evolved from the first version developed in 1986. According to Rupp, the V Model has found widespread application as standard in the public sector, especially in military organisations.

It contains requirements and information related to the context (life cycle and overall architecture) and benefits from a modular structure that can easily be tailored to specific project needs. It has a steep learning curve and needs high effort in order to be successfully integrated.

IEEE 830-1998

According to Rupp, the standard (provided in [Stan98]) has a high detail on the description of the target system. It does not contain information related to the project itself and concentrates more on the product than on the system compared to other recommendations.

The notation-independent standard is used internationally and provides sections next to software-centric requirements that focus on other aspects of the product (e.g. hardware or communication).

1.3.3. Classification

Leffingwell [Leff03] distinguishes three different types of requirements, which I will use as classification.

1. Functional software requirements
2. Non-functional software requirements
3. Design constraints

Maciaszek [Maci05] refers to the expected services of the system as *service statements* (corresponding to functional software requirements) and to constraints the system must obey as *constraint statements* or *supplementary requirements* (corresponding to non-functional requirements).

The term *design constraints* is not used by Maciaszek, because he follows the ideal model where "the specification models should be independent of the hardware/software platform on which the system is to be deployed. Hardware/software considerations impose heavy restrictions on the vocabulary (and therefore expressiveness) of the modelling language." According to Maciaszek, the design constraints have to be considered at the time of system design.

Because both sources have a similar understanding of the different types of requirements, I will use their findings in the detailed description of this classifications.

Functional Software Requirements. The *functional* requirements express how the system behaves. If the system should behave in a strict pattern that must be obeyed at all times, this service statement defines a *business rule*. Functional requirements can be grouped into

- description of the system's scope
- business functions

- required data structures

Non-functional Software Requirements. The *non-functional* requirements refer to aspects such as usability, performance, and often security. Non-functional requirements are not behavioral in nature, but are constraints on the development and implementation of the system. It depends on the phrasing of the requirement whether it can be considered as functional or non-functional. Security aspects are often described in a behavioral manner and thus become part of the functional requirements. Maciaszek further divides non-functional requirements into requirements that relate to

- usability and re-usability
- reliability
- performance and efficiency
- supportability
- other constraints

Design Constraints. Design constraints impose limits on the design of the system. They do not affect the external behavior of the system but model needs that must be fulfilled to meet technical, business or contractual obligations.

1.4. Requirements Documentation

Thirty years ago computer illiteracy was an issue for the evolving new technologies. One of the main problems was "how best to provide information to the *non*-expert"[Salt75]. Nowadays most people know what they can do with computers, and "operating systems are required for any computer to be usable by a non-specialist."[Fagi99]. This also applies to application software.

Every software product has requirements that define what the software is expected to do when it is completed. Requirements documentation is an essential part of the software requirements process [Powe03], and refers to both the process of documenting the requirements and the resulting work product, the software requirements specification (SRS). Other parts of the requirements process include:

- elicitation and discovery
- analysis and modeling
- management
- validation and agreement

Documentation is crucial in the field of software engineering. Adherence to functional requirements is one of the typical quality attributes of software [Zuse05]. Without the documentation of requirements it would not be possible to measure this quality attribute. "Software engineers rely on program documentation as an aid in understanding the functional nature, high-level design, and implementation details of complex applications."[Thom01]

This section takes a deeper look at how software documentation is used to

- Improve Program Understanding
- Measure the Quality of Software

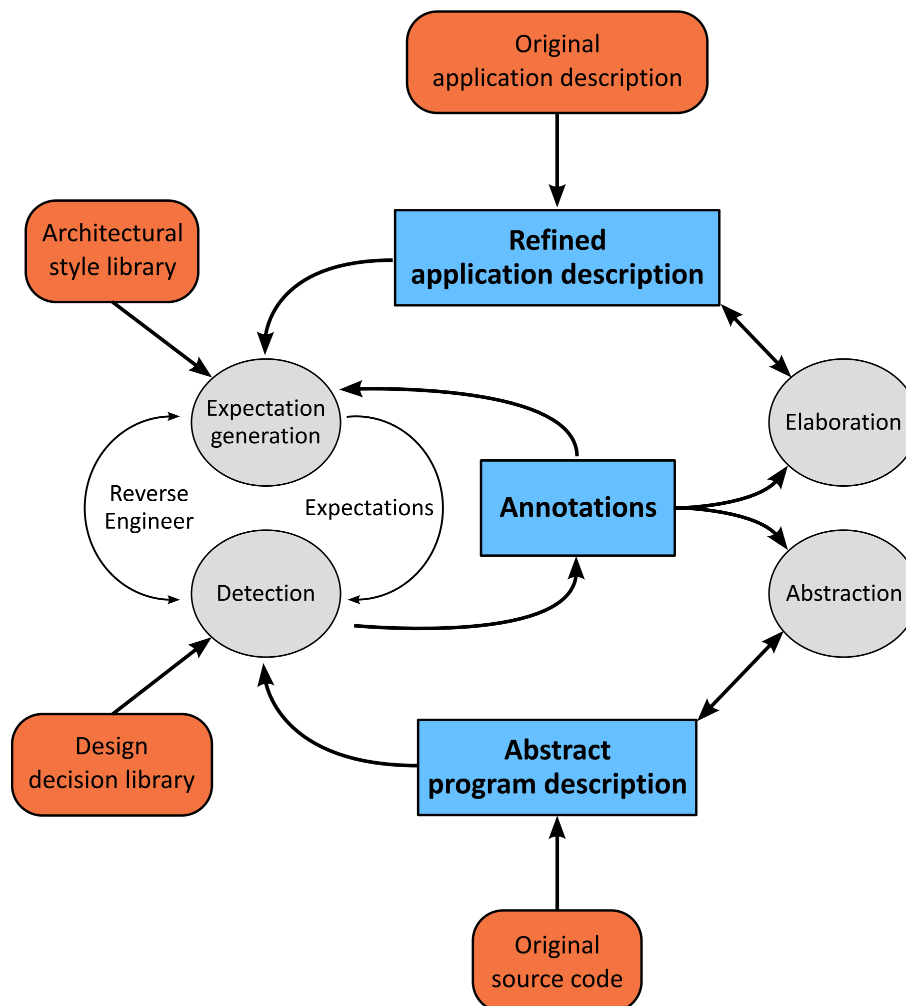
1.4.1. Program Understanding

Rugaber [Ruga00] explains that a *domain description* or *domain model* can "give the reader a set of expected constructs to look for in a program". This information is provided by the set of documents comprising the requirements documentation.

But the software engineer is not the only role interested in good documentation for program understanding. During the study of an industrial maintenance environment, 20 percent of the maintenance problems were found to be due to bad documentation [Visc93]. The most frequent problems were documentation faults and documentation clarifications.

The *Synchronized Refinement* (SR) method, described by Rugaber [Ruga00], takes as input the source code of a software (*Original source code*) and a description of the application domain (*Original application description*). Other sources of programming knowledge, labelled *Architectural style library* and *Design decision library* may also be available. The diagram for SR is depicted in Figure 1.4, "Application Description as Input for the Synchronized Refinement Method", with input sources highlighted.

Figure 1.4. Application Description as Input for the Synchronized Refinement Method



The method *Synchronized Refinement* [Ruga00] needs the original application description (top) as input, in order to reverse engineer software for program understanding

During the study of a production environment [Visc93] to evaluate the effectiveness of different technologies and their impact on productivity and reliability, documentation showed an important factor: high use of documentation improves productivity by 11 percent and reliability by 27 percent compared to low use. To improve quality, effective documentation of each phase of development is necessary. Quality inspections are as important as program inspections when the goal is to increase productivity and final software quality.

1.4.2. Measure the Quality of Software

If software documentation is used to measure the quality of software, it is important that the documentation itself is of high quality. I will distinguish two classes of low quality documentation:

- Missing documentation
- Incorrect documentation

Lack of Documentation

According to Visconti and Cook [Visc93], maintenance programmers report that for most maintenance tasks the source code is the only available documentation. In a survey of 487 data processing organizations, documentation quality ranked 3rd in the list of 26 maintenance problem items. In this survey quality and adequacy of design specifications were accounted for 70 percent of product quality.

Errors in Documentation

Bell and Thayer [Bell76] claim that the search for requirements problems should be a continual one. Software changes, and so do the requirements. Visconti and Cook [Visc93] found out that 85 percent of all software development errors are introduced during the phases requirements, analysis and design. 80 percent of software errors in large real-time systems are requirements and design errors due to ambiguity, incompleteness, or faulty assumptions.

Stamey and Roth [John99] describe five major problems that can arise in the delivery of technical documentation: incompleteness, incorrectness, vagueness, unusability, and nondelivery. In their work, they refer to the two major roles in technical documentation: the writer and the developer. As there are similarities to the area of this thesis, I will adapt the first four classes of problems to errors in requirements documentation. Nondelivery is similar to the lack of documentation described in the previous section. Instead of referring to writer and developer, the roles as described in Section 1.2.3, “Roles in the Requirements Process” are used.

Incompleteness
: The author fails to capture all information necessary for the documentation to explain the needed functionality, or the customer fails to convey all of the required information.

Incorrectness
: The result of either the customer not properly conveying information to the author, or misunderstanding on the part of the author.

Unusability
: Refers to problematic issues relating to the hardware and/or software platform; parts of the documentation that are not applicable to effective target environment.

Vagueness
: This is the opposite of one of the main characteristics of a good SRS: unambiguity (see Unambiguous). This issue, also referred to as *incompatible granularity*, is typically the fault of the customer who determines the requirements.

The above classes of errors may be reported during test (unit, integration test), or arise during software evolution. Volatility also takes a major role in creating errors, as user needs change over time, and so do their expectations in terms of functionality or time scale.

In most cases, software maintainers discover that the available documentation is not current [Visc93]. This is a result of a combination of the two previous classifications: changes that are introduced in software over time have not found their way into the documentation. Consequently, the (a posteriori) outdated documentation does not contain fully correct information.

1.5. Documentation Types

Different elements of the documentation are important for different stakeholders. A customer with no technical background, which is only interested in a single product with very limited features, will not have a business related interest in internal documentation on interfaces or source code. On the other side, technically involved customers with a high level of integration of an application in their environments may even be involved in technological decisions concerning the application.

The types of documentation are inferred from two sides, based on one of the following

- The requirements situation or context.
- The documentation needs of different stakeholders.

While the first perspective distinguishes requirements documentation according to their context (e.g. requirements for a vertical market situation, target customer market or in-house solution), the second perspective uses the stakeholders' role in the project and interests following from this to compile the types of requirements documentation needed.

1.5.1. Requirements Context

Power and Moynihan [Powe03] compiled a list of seven types of requirements situations based on the sources of requirements. In their research, the requirements sources are not directly related to roles or teams as presented in Section 1.2.3, “Roles in the Requirements Process”, but instead their analysis looked for dependencies between requirements, building a precedence network which resulted in 8 root requirement sources.

From the resulting categories of situation types, the *target customer market* resembles most the environment the authoring framework should support. According to Power and Moynihan's research the following documentation types are significant for this situation:

Statements of the required effects
: Are often organized around features with dependencies and other attributes. An emphasis is placed on features that are unique to the product.

Recorded issues and changes
: Relate to specific features or lower-level requirements.

Defined goals and objectives
: System goals, such as non-functional requirements, are important.

Specified constraints
: Application constraints, such as behavioral constraints, are significant.

Recorded agreement
: Takes place at two levels: informal agreement with the target customer(s), and formal internal agreement

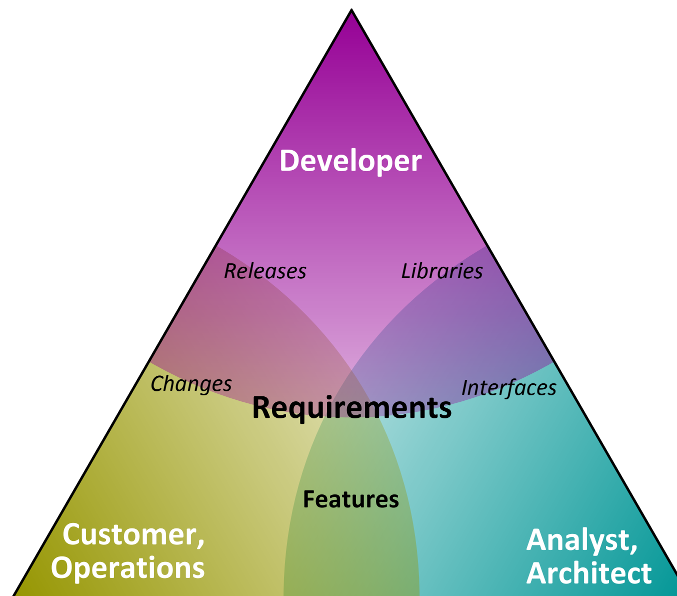
1.5.2. Stakeholder Documentation

Numerous people are involved in delivering and maintaining a solution. The set of stakeholders will vary, depending on the organization and the project. Based on [Clem02], and focusing on requirements and test documentation, the view is reduced to three main groups:

- Customers
- Analysts
- Developers

Figure 1.5, “Stakeholder Documentation” shows a compressed view of the stakeholders and their relationship to different documentation artifacts. The SRS is a central documentation artifact for all stakeholders.

Figure 1.5. Stakeholder Documentation



Documented requirements are relevant for all three groups of stakeholders

Customer and Operations

Being the stakeholder who pays for the projects, customers are interested in costs, progress and convincing arguments that the architecture and resulting system will meet the functional and quality requirements. Depending on the service level agreements, customers will usually have to support the environment operatively. They will want to know how the system will interoperate with other systems in the production environment.

A customer likely wants to see

- Work assignments
- Integration view of the system, in terms of deployment and stability/availability.
- Top-level overview of the system.
- Results of different analysis (certifications, performance, unit tests ...).

Architect and Analyst

In general, an analyst is interested in the ability of the design to meet the system's quality objectives. The architecture is the basis for architectural evaluation methods and must provide the information necessary to evaluate quality attributes such as security, performance, usability, availability and modifiability.

An architect or analyst of the system is likely to be interested in the following documentation, depending on the type of analysis and the level of detail for the person's task

- Information on the modules.
- A view of the deployment and the different components.
- Descriptions of internal and external interfaces and communication between processes and components.

Developer

Responsible for implementing a solution, this group has to be aware of the solution design and architectural documentation. Additional restrictions come from internal directives (based on maintenance, legal, QA), and external processes.

A developer is given responsibility for an element (component, module, application ...) in order to have a certain degree of accountability. This element could also be a commercial off-the-shelf product. For each element there has to be a person that makes sure that the element performs as specified/advertised. This person will want to know

- The general idea behind the system.
- Which element the developer has been assigned, that is, where functionality should be implemented, and the details of the assigned element.
- The elements with which the assigned part interfaces and what those interfaces are.
- The assets the developer can make use of (libraries, frameworks, code ...).
- The constraints, such as quality attributes, legacy systems interfaces, and budget, that must be met.

1.5.3. Summary

For further reference, the documentation types for the creation of an authoring framework for requirements documentation is described. The description is based on Section 1.3.2, “Content” and Section 1.3.3, “Classification”.

Table 1.1. Documentation Types for the Authoring Tool

Documentation Type	Description
Statements of the required effects	*Business rules* capture the business functions in this type of documentation
Recorded issues and changes	Lists information from issue or change management systems relevant for the documentation
Defined goals and objectives	Description of the system's scope and intention.
Specified constraints	Non-functional requirements not already specified in required effects, or goals and objectives are captured in this documentation type.
Recorded agreement	References to the decisions from which certain requirements are derived. This facilitates the traceability of requirements.
Top-level overview of the system	This documentation type contains a description of the different parts of the system, and how they interact with one another.
Interface documentation	Captures required data structures and rules for external interfaces and may also cover a description of the expected internal behavior.

It may be necessary that software developed to integrate with other existing systems uses certain libraries, or is built with specific requirements on a design level. For clarity, this design-related information is omitted from the table.

1.6. Software Development Models

The development of a software product is a complex process. Software development models are structures imposed on the development of a software product which have emerged from the need

to streamline development and raise the quality of the resulting artifacts in order to make development an efficient and predictable activity. This section gives an introduction to the main concepts and terminology. Subsequently this section focuses on the creation of requirements documentation in different development models.

1.6.1. Terminology

The terms *development model*, *development process*, *process model*, and *software lifecycle* are used interchangeably in the literature. Primarily, a process model determines the order of the stages involved in software development and establishes transition criteria for progressing from one stage to the next [Boeh88]. The following transition criteria are distinguished: *completion criteria* for the current stage and *choice criteria* and *entrance criteria* for the next stage.

At a lower point of view, the term *software development methods* (or methodology) refers to the definition of rules and guidelines on how to navigate through each stage or phase and how to represent artifacts (products of the stage or phase).

Barry Boehm distinguishes between two major classes of software development methods [Boeh02], which is used in this section: *plan-driven* and *agile*. The main difference between these two classes lies in the integration of requirements and their changes into the development process. As can be seen in Table 1.2, “Home Ground for Agile and Plan-driven Methods” (based on [Boeh02]), each class of development method is very effective in a different type of project environment. Boehm calls this the *home ground*.

Table 1.2. Home Ground for Agile and Plan-driven Methods

Area	Plan-driven methods	Agile methods
Customers	Access to knowledgeable, collaborative, representative, and empowered customers	Dedicated, knowledgeable, collocated, collaborative, representative, and empowered
Requirements	Largely stable; knowable early	Largely emergent; rapid change
Architecture	Designed for current and foreseeable requirements	Designed for current requirements

In the remainder of this section I will start with a short introduction on each of these classes in general and subsequently focus on a single representative process model. For the representative models, we will try to describe the evolution and documentation of requirements in order to show the application of the different approaches. This section shows that requirements are an integral part of a structured process, both in plan-driven and agile software development models.

1.6.2. Plan-Driven

According to Williams [Will04], designing software with a plan-driven approach can be divided into *waterfall* (strictly incremental) and a combination of iterative and incremental methods. Before comparing these different approaches, the evolution of software process models has to be followed.

History

The model used in the earliest days of software development is the *code-and-fix* model. Boehm describes the approach with the following two steps [Boeh88]:

1. Write some code
2. Fix problems in the code

Boehm further notes the difficulties:

- The poor structure imposed high costs on subsequent fixes and showed the need of a design phase prior to coding.
- The resulting software did not correspond well to the users' needs because of the missing requirements phase.
- The code was not prepared for testing or modification.

Waterfall

Out of the recognition of the above problems and experience in large software systems *stagewise* or *incremental* models were developed, the *waterfall* model being the most widely known.

Waterfall models are a classical, sequential approach to designing software. Royce [Royc87] is often credited with doing early work on waterfall methods [Zuse01].

According to Bell and Thayer [Bell76], each of the documents in the early phases of the waterfall can be considered as stating a set of requirements. At each level a set of requirements serve as the input and a design is produced as output. This design then becomes the requirements set for the designer at the next level.

The difficulty with strictly incremental models is the “emphasis on fully elaborated documents as completion criteria for early requirements and design phases”[Boeh88]. This approach was not an effective way for many classes of software, especially dynamic end-user applications, or areas supported by fourth generation languages.

The concerns led to the *evolutionary development model* where the results are perceived as expanding increments to an operational software product. The direction of evolution is determined by operational experience.

Spiral

Building on experience with refinements of the waterfall model as applied to large government software projects, Barry Boehm describes the *spiral model* [Boeh86] by combining features of waterfall methods with *software prototyping* as found in the evolutionary development model described above.

The spiral consists of several cycles which involve a progression with the same sequence of steps “for each portion of the product and for each of its levels of elaboration, from an overall concept of operation document down to the coding of each individual program”[Boeh88]:

1. Identification of the current objectives, alternative means of implementation of the objectives, and the constraints imposed on the application of the alternatives
2. Evaluate the alternatives relative to the objectives and constraints to identify significant sources of project risks. If risks have emerged, formulate a strategy for resolving the sources of the risk.
3. Engage in the remaining risks by either following evolutionary development, in case a prototype could reduce these risks (e.g. dominating performance or user-interface risks), or pursuing the basic waterfall approach (e.g. if the major part of the risks stem from program development or interface-control).
4. Validate the products developed in this cycle by conducting a review involving the primary people or organizations concerned with the product.
5. Prepare plans for the next cycle. This plan and the involved resources are also validated during a review.

Rational Unified Process

The *Rational Unified Process* (RUP) is an example of an incremental and iterative development model. It is at the same time a software development process framework and a software process product developed by the company Rational and distributed commercially as a product[1]. It is a concrete implementation of the *Unified Process* [Zuse04]

Concepts

It provides a set of recommended best practices for software development. There are four basic concepts [Prie00]: *worker*, *activity*, *artifact*, *workflow*.

Worker
: This role defines a set of behaviors and responsibilities that an individual may perform.

Activity
: Is performed by a worker and provides a meaningful result.

Artifact
: The result of an *activity* is called *artifact*.

Workflow
: Is a logical group of *activities*.

Workflows

The RUP defines the following *core workflows*, which classify the work required to develop a software product within this process:

- Business modeling
- Requirements
- Analysis and design
- Implementation
- Test
- Deployment

I will focus on the first three work flows and identify the documentation artifacts of particular interest from a documentation perspective as outlined in [Prie00]

Business modeling. This is the first workflow to begin work in a software development process. It defines the workers *business process analyst* and *business designer*. These workers produce the following artifacts on a documentation level:

- Glossary, defining the language of the business domain
- Business use case model
- Business workers, entities, and organization units
- Business use cases
- Business use case realizations

Requirements. In this workflow, the RUP defines the main workers *system analyst*, *architect*, and *UI designer*. They produce the following artifacts:

- Glossary and business use case model are also extended in this workflow
- Use case model, created from the business use case model

[1]IBM acquired Rational in 2002, and provides product information and related services at the Rational website [http://www.ibm.com/software/rational/].

- Requirements attributes
- Change requests
- Actors
- UI prototype
- Use case storyboard

Analysis and Design. The workflow for analysis and design defines the workers *architect*, *designer*, and *database designer*. The following artifacts are of particular interest:

- Analysis model, inferred from the use case analysis
- Design model
- Analysis and design classes
- Use case realizations
- Data model

1.6.3. Agile

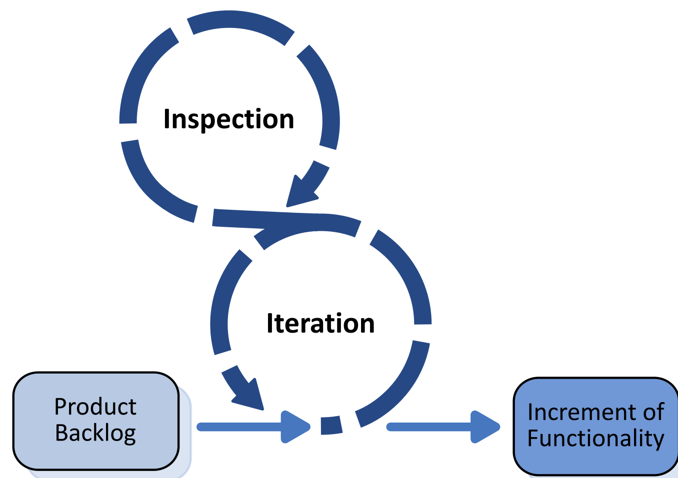
While traditional development models advocate extensive planning, codified processes, and rigorous reuse, followers of newer models call for “an approach to development that dispenses with all but the essentials”[Boeh02]. These models are subsumed under the term *agile* and center around “satisfying the customer through early and continuous delivery of valuable software”[2].

In the literature *Scrum* and *Extreme Programming* (XP) are often referred to as examples for agile methodologies [Vanh03][Bart07][Judy08]. In this thesis, only Scrum is presented.

Scrum

Scrum is a project management method for agile software development. It was developed by Schwaber and Sutherland in the early 1990s [Schw04]. One of the main differences to plan-driven methods is that the iterations start with a team review of what has to be done. The team has to decide what can be turned “into an increment of potentially shippable functionality by the end of the iteration”.

Figure 1.6. Scrum Skeleton



At the start of the iteration, the team selects what has to be done. The result at the end of the iteration is an increment of functionality (based on [Schw04]).

[2]Refer to Agile Alliance [http://www.agilealliance.org/] and the Manifesto for Agile Software Development [http://www.agilemanifesto.org/] for detailed information on the criticism of non-agile development models.

Roles in Scrum

All management responsibilities in Scrum are divided among the three roles [Schw04]:

Product Owner
: Represents the interests of everyone with a stake in the project and its resulting system. He or she achieves initial and ongoing funding for the project by creating the project's initial overall requirements, return on investment objectives, and release plans.

Team
: Teams are self-managing, self-organizing, and cross-functional and are responsible to turn requirements into an increment of functionality within an iteration. Members of the team are collectively responsible for the success of each iteration and the project as a whole.

Scrum Master
: The person responsible for the Scrum process. He or she teaches Scrum to everyone in the project and implements Scrum so that it fits within an organization while delivering the expected benefits.

These are the active roles of the people committed to the project (refer to active and passive roles in Section 1.2.3, “Roles in the Requirements Process”).

Flow in Scrum

A project in Scrum starts with a vision of the system to be developed, which becomes clearer as the project evolves. The *product owner* formulates a plan that includes a *product backlog*, i.e. a list of functional and non-functional requirements according to the vision. The elements of the product backlog are prioritized and divided into proposed releases.

Work is done in *sprint*s, “an iteration of 30 consecutive calendar days”, initiated with a *planning meeting* where the product owner and the team decide what will be done for the next sprint. The product owner provides the information on what is desired (based on the product backlog and its priorities and groupings), while the team provides the information on how much of what is desired it believes can be turned into functionality during the next sprint.

During the sprint, the team gets together for 15 minutes during a *daily scrum* intended to synchronize the work of all team members and to schedule meetings needed to advance the project.

At the end of the sprint, during a *sprint review*, the team presents to the product owner (and other interested stakeholders who want to attend) what was developed during the sprint. This is followed by a *sprint retrospective* meeting held by the scrum master, where the team is encouraged to revise its process to improve it for the next sprint.

Requirements in Scrum

The requirements in the Scrum methodology are stored in the product backlog. The product owner is responsible for the contents, prioritization and availability of the product backlog, which is never complete during an active project. The product backlog is under constant change and exists as long as its product exists. Items for future sprints that have not been started by the team are not analyzed or estimated.

Chapter 2. Goals and Evaluation of Requirements Authoring

I will present a general view on the goals of document authoring and ways to measure maturity of this authoring process in the context of software requirements. This chapter shows how maturity of document authoring can be evaluated on a process and on a document level. The main part of this chapter focuses on presenting documentation approaches effectively used in the industry. With common terminology and main deficiencies to address, the basis for the remainder of this thesis is layed out.

2.1. Goals of Document Authoring for Requirements Documentation

Content-related characteristics of documentation are clear and usually directly determined by the author's education, knowledge, experience and other traits which will not be covered in this thesis. This section concentrates on more easily measurable characteristics. Some of these may overlap with the author's traits, but most of them can be leveraged through well-designed processes and the right tools.

Hard characteristics for good requirements specification, as shown in Section 1.3.1, “Characteristics”, are based on the need to achieve certain goals. A good SRS is in character with the items described and aids in attaining the goals. In [Kylm03] the following goals for document authoring in general are listed:

- Gain understanding
- Share information
- Keep track of information shared

During a survey, Forward and Lethbridge [Forw02] have received similar results regarding the goals of the documentation, the four most important aspects were the document's information, timeliness, availability, and the use of examples. These characteristics are similar to the first two goals in the above list.

These goals are not restricted to the resulting documentation itself (i.e. the specification, design document or diagram), but can be applied to the set of information associated with the authoring of the documentation. In this thesis, these goals are applied to the process of documenting requirements.

2.1.1. Gain Understanding

It is only possible to gain understanding, if the information is shared and the sharing is done in a correct an unambigous way. It does not matter what has been written, but how it is interpreted. More often than not, there are better ways to gain understanding than reading a document. According to [Ruga00], in order to effectively understand software we must appreciate its architecture and other design choices; it is important to take advantage of as much of the reasoning that went into the existing solution.

With sufficient knowledge about a software system it can evolve in a disciplined manner. “The essence of program understanding is identifying artifacts, discovering relationships and generating abstractions”[Huan03].

The characteristics for this goal will be described in more detail:

- Quality of perception
- Learnability

Quality of Perception

This characteristic is supported by main attributes of a good SRS as described in Section 1.3.1, “Characteristics”: unambiguous and consistent. These determine how easily information can be absorbed.

In order to evaluate a process, it is essential to observe whether instruments are in place that assure that documentation is unabiguous and consistent. These instruments can be very diverse:

Measure *Quality of Perception*

- Specially trained technical writer
- Editor for software documentation in the QA department
- Generally accepted, company-wide agreement on authoring
- Template-based documents that adhere to a previously defined structure
- Review on an ambiguity and consistency level

Learnability

According to Dan Tamir et al. [Tami08], *learnability* in the context of software quality (ISO/IEC 9126) can be measured by evaluating the performance of a user in multiple executions of a repetitive task. In their research, the group also explained the term *understandability* in the same context. In the context of document authoring, we merge both terms and define them as a characteristic that measures how much effort is needed to read and understand a document without direct guidance of the document's authors or people involved in the document's creation.

In a business-driven environment it is often impossible to talk directly to the authors of a document, or consult people involved in creating the documentation. In this case, it is important that the available documentation suffices to obtain an understanding of the domain. It should not be necessary to extensively demand other people's resources to gain a better understanding.

Another attribute from Section 1.3.1, “Characteristics” that affects the degree of learnability is traceability. If the origin of a requirement is clear and can be referenced, the process of understanding without additional resources is facilitated. A document that aids autodidacts should clearly indicate authors and source of information for further reference.

Measure *Learnability*

- Contributions of an author are marked as such
- Origins of requirements are indicated
- Review on self-directed understanding (e.g. hallway testing[1], logical structure[2]).

2.1.2. Share Information

It is expected that people read the authored documents or are able to access the necessary information otherwise. For this to work, people have to know what information is available, what to read and where to find it. According to Chris Rupp [Rupp06], the major part of communication

[1]Not only a methodology of usability testing, *hallway testing* can be used to find issues that are not clear to people too deeply involved in a work stream. During hallway testing, random people, indicative of the end users and not involved in the creation of the artifact, perform a test or review.

[2]Barbara Minto [Mint01] explains that written information has to be presented at different layers of abstraction and distinctively grouped in order to be comprehended by the reader. She describes three simple rules to achieve a recommended pyramidial grouping.

is concerned with the distribution and exchange of information. He further notes that for a group of people working together, it is important that their conduct is based on the same information (revision of a document), or otherwise they miss a common understanding which is important for their communication.

Modern information or document management tools (DMS) can support this process goal on a technical level. Still, it is necessecary that the process spans around these tools and leverages them, in order to reduce complexity and manual work for all participants. For widely dispersed product development teams, it is difficult to communicate or exchange information without digital transfer. At the same time, the deployed tools should not impose a high overhead or demand a high amount of direct manual intervention.

This goal is divided into the following characterstics, which are explained in more detail below

- Availability
- Distribution
- Retrieval

Availability

The term availabilty refers to receiving information after submitting an informed request. Bill Albing [Albi96] claims that “technical documentation involves not a chain of discrete projects […] but a network of interrelated documents”. The technological progress nowadays allows the leap “from individual writing and reading to group creation and enterprise-wide […] consumption.”

Requirements from different sources (see Figure 1.2, “Sources Of Software Requirements”) influence the problem solving process. These requirements and their proper specifications have to be accessible when designing a solution. Otherwise, it is possible that the final product does not fully meet expectations, or is disproportionally expensive to adapt compared to a change at design time.

Measure *Availability*

- Short roundtrip between asking for and receiving information
- Low technical effort to access information
- Link directly to documents or other sources of information

Distribution

Members of the product development team have different perspectives on the domain. They talk to stakeholders outside of the team at different levels of abstraction. Maciaszek [Maci05] says that “requirements elicited from customers may overlap or conflict” and that these requirements need to be *negotiated* and *validated*. This is associated with the process of writing up a requirements document.

Maciaszek continues: while “*requirements negotiation* is typically based on the draft of a document […], *requirements validation* requires a more complete version of the requirements document”. Initially, the results of negotiation and validation (during workshops, reviews, inspections …) are only visible to a small group of participating individuals, but have to be effectively communicated to other members of the product development team.

Measure *Distribution*

- Process actively supports knowledge sharing
- Low effort to communicate requirements

Retrieval

This third characteristic refers more to the deployed DMS than to the process itself. Every document managed by the system provides metadata that can range from basic dates as timestamp or content type to information on the author or the context. This data can be used to distinguish a piece of information from others, and hence find it using a query that can be broadened and narrowed (searched and browsed).

This thesis will not go into details of automatically retrieving this information from a structured piece of information. The information provided by the author or the tools employed during authoring can be used by the DMS to put the piece of information into the right context (i.e. project, work stream, status of information, department …). It is essential that the requirements documentation process defines what types of information are essential. This information can then be added directly to the document before storing it in the DMS, or to the metadata attached to the document in the DMS (after being stored).

Measure *Searching and Browsing*

- Central DMS (or similar system) is enforced to be used
- Identification of the metadata that has to be stored in DMS
- Categories, labels, hierarchy of metadata are defined

2.1.3. Track Information

Successful programs not only evolve to fix problems, but also to meet new requirements, improve efficiency and refine existing solutions to adapt the original approximation to the requirements of the real-world context of the program [Ruga00]. Often documentation does not change accordingly. Among the most efficient solutions for a better understanding are documents that keep track of information history [Kylm03].

For this goal, two characteristics are classified

- Versioning
- Transparent changes

Versioning

For this definition, I reuse previous work by Stuckenholz [Stuc05]. He writes about software component evolution and versioning, but selected assumptions and definition also apply to software documentation and software requirements documentation.

> A *version* of a [document] is a specific instance on the time axis, which came into existence due to a revision or change. The way how a version is identified by a *version identifier* […] is defined in a specific *version model*.

If multiple authors work on a single document, author A may not know about the changes author B has put in the new version of a document. Even in a single-author situation, it might be necessary to roll-back to a previous version of a document. A well integrated versioning process makes it clear for participating parties what changes have been applied to a document. The version identifier is used to distinguish different versions of the same document during discussion or in document references.

As already indicated in Section 2.1.1, “Gain Understanding”, it is important to take advantage of as much of the reasoning that went into the existing solution. Changes to previous version of a document (revision history) and metadata associated with these changes are valuable sources

of information. In popular version control systems[3], the meta-information contains at least: date, author, and a free-text comment describing the change.

Measure *Versioning*

- A version model is in place
- Changes are detected by using meta-information (e.g. timestamp)
- Changes are detected by comparing content
- Ability to attach meta-information to a change for subsequent reference
- Differences between arbitrary revisions of a document can be retrieved

Transparent Changes

Changes applied to a running system or test environment have to be documented. The same applies to a document or single requirement. It is possible that person *A* who implemented a change did not fully understand the requirements or introduced unexpected behavior. If *A* is not available at the time a problem concerning this change is raised, someone else will have to address the problem. For this person it will be necessary to know about the changes introduced by *A*.

It may be necessary to restrict changes from being applied to a system: changes can then only be introduced in a controlled manner and have to go through different stages of acceptance. This raises the awareness of and knowledge about changes introduced to the system.

Karl Wiegers claims [Wieg04] that a well-defined *change control process* provides:

- Formal mechanism for proposing changes in requirements
- Basis for well-informed business decisions
- The possibility to track proposed changes

Maciaszek [Maci05] uses the term *change management process*, which “involves tracking large amounts of interlinked information over long periods of time”, being supported by a software configuration management tool.

Measure *Transparent Changes*

- Changes to requirements are documented
- Relation of changes in requirements and the product is clear
- A change management/control policy is in place

2.2. Evaluation with Maturity Models

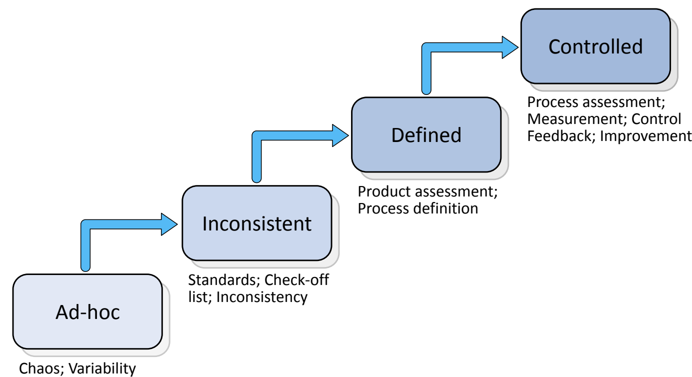
Huang and Tilley [Huan03] describe two dimensions to measure documentation quality: the process, i.e. the manner in which the documentation is produced, and the product, i.e. the attributes of the final product. The following two models define different levels of maturity on the process and the product level.

[3]This is based on open source project hosting sites, such as SourceForge.net, GitHub, java.net, Google Code and similar sites (refer to the comparative Wikipedia page [http://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities]). The most popular version control systems provided by these sites are (in this order at the time of writing) are Subversion, CVS, git, and Mercurial. Realtime version control statistics provided by Cia.vc [http://cia.vc/stats/vcs] show similar numbers.

2.2.1. Process Maturity Model

This 4-level documentation process maturity model, developed by Visconti and Cook [Visc93] tries to address the problem of low quality and missing documentation. The authors define four levels of maturity (see Figure 2.1, “Levels of the System Documentation Process Maturity Model”) together with their process areas, practices, indicators, challenges and significance to software development and maintenance. The following levels are presented.

Figure 2.1. Levels of the System Documentation Process Maturity Model



Visconti and Cook [Visc93] distinguish four cascading levels in this maturity model

Levels in the Process Maturity Model

Ad-hoc
: Documentation does not receive a high priority and is usually missing or out of date. In order to improve the situation, the organization has to establish documentation standards.

Inconsistent
: At this level, documentation is considered important and must be done. The problem lies in inconsistent application of standards. The organization has to exercise quality control where the documentation artifacts are assessed in terms of quality and utility.

Defined
: It is agreed that the documentation must be done well and documentation tools are used consistently. Documentation is updated after each change and quality assessment methods are in place. Measures of documentation process qualities have to be established.

Controlled
: Documentation process quality can be assessed and an improvement feedback mechanism is in place. Documentation tools are integrated with software CASE tools. For this level, it is important to maintain a continual optimization of the documentation process and automate the collection and analysis of the process' data.

The levels represent a structured spectrum of documentation maturity levels for an organization, and can be used to identify problem areas.

2.2.2. Documentation Maturity Model

Huang and Tilley [Huan03] focus more on the product than on the process components. The *Documentation Maturity Model* (DMM) is a five-staged model based on the *Software Capability Maturity Model* (CMM) created by Carnegie Mellon University's Software Engineering Institute.

The product quality component of the DMM is based on the notion of *key product attributes* (KPA). As depicted in Table 2.1, “Documentation Maturity Levels associated with KPAs”. Its product-quality KPAs are efficiency, format (textual and graphical), and granularity.

Table 2.1. Documentation Maturity Levels associated with KPAs

KPA		Maturity Level				
		1	2	3	4	5
Format	Text	Inline & Informal	Inline & Standardized	Hyperlinked	Contextual	Personalized
	Graphics	Static & Informal	Static & Standardized	Animated	Interactive	Editable
Efficiency		Manual	Semi-automatic & Static	Semi-automatic & Dynamic	Automatic & Static	Automatic & Dynamic
Granularity		Source code	Design patterns	Software architecture	Requirements	Product lines

The DMM with its associated KPAs is focused on documentation for the developer. Because it is important for the following chapters, the characteristics are put in the context of a product development team.

Efficiency

This attribute refers to the “level of direct support the documentation provides to the software engineer engaged in a program understanding”. Huang and Tilley list the following aspects for the definition of efficiency:

- accuracy between the documentation and the source code (the implementation in this thesis' broader context)
- ease with which the documentation can be generated
- completeness of the documentation with respect to the information required by the developer (the author in this thesis' broader context)

At the lowest level, *manual*, the documentation is generated manually. At the next two levels, *semi-automatic & static/dynamic*, documentation is generated using tool support. In the case of *static* documentation, it reflects the system only at the time of generation. With a *dynamic* documentation, changes made to the system can be reflected in the documentation. The most advanced two levels remove the need of the author's involvement.

Format

For the *format* attribute, Huang and Tilley distinguish between *textual* and *graphical*. Only one part of this attribute is directly applicable to quality assessment.

Textual documentation ranges from “inline prose written in an informal manner” at the lowest level, to “personalized views dynamically composed from a […] database”. Adhering to documentation standards, hyperlinking, and context-specific documentation, each lead to a higher level (in this sequence from level 2 to 4).

On the other side, *graphical* documentation ranges from static images with “non-standard representation of software artifacts and relationships” to graphical documents that are editable by the user. The levels in between these extremes are static & standardized (level 2), animated (level 3), interactive (level 4).

Granularity

For Huang and Tilley, the *granularity* product attribute refers to the “level of abstraction described by the documentation”. Low granularity means a documentation that is close to the implementation of the system, while high granularity levels are closer to the business functionality of the system.

Source code documentation (lowest level) provides information that supports in “understanding the algorithms and data structures”. From my experience, information at the level of design patterns (level 2) can be provided in good source code documentation at module, class or package level. This aids in understanding the “higher-level rationale behind implementation decisisons” and helps the engineer's work in constructing more abstract mental models.

High-level design (software architecture, level 3) “captures […] aspects of a system's structural characteristics”. Huang and Tilley, explain the difference to requirements (level 4) in the following manner:

> Software architecture captures the engineer’s view of the system’s high-level design. Requirements represent the system’s intended purpose from the point of view of the user.

In their description of this level, they also take into account that requirements-level documentation can greatly aid in mapping maintenance requests through multiple levels of abstractions.

The highest granularity (level 5) and highest level of abstraction is product line documentation. It contains information on “commonalities and variablities in the product [and] how a specific product is related to similar products”.

2.3. Exemplary Industrial Processes

Software development models, such as those presented in Section 1.6, “Software Development Models”, are taught to students during their computer science studies. They represent an important part of the basic education of a software engineering student, and are presented during theoretical courses and confirmed in hands-on labs on real-world projects.

Eventually, the main aspects of these models have found their way into the industry, where the concepts are adapted and applied to the culture and working environment of the product development team responsible for the delivery of the software product (see Section 1.2.3, “Roles in the Requirements Process”). In this section, I will present the documentation aspects from two selected processes as they are used in the industry.

2.3.1. RaPiD 7

The method called *Rapid Production of Documentation, 7 steps* (RaPiD7) presented by Roope Kylmäkoski [Kylm03] was developed in Nokia between 1999 and 2000 for efficient document authoring in software development projects.

It tries to address the following problems, presented as the most typical problems found using traditional authoring approaches:

- A lot of calendar time is allocated for authoring the documentation.
- The contents of the document are mostly based on the opinion of the writer.
- A common theme on the contents of a document on hand cannot be defined.
- A common agreement cannot be reached, because defects are found too late.
- No proper commitment.
- No efficient information sharing, and therefore an inadequate level of understanding.

In RaPiD7, the document on hand is authored in a team during consecutive workshops. In these workshops, which have to be well prepared, brainstorming and problem solving methods are used. The documents are mostly written during the workshops, or immediately after each issue has been resolved and a decision has been made. Each workshop has the following structure:

Steps of RaPiD7

1. Preparation
2. Kick-Off
3. Gather Ideas
4. Analyze Ideas
5. Detailed Design
6. Decisions
7. Closing

A workshop can be considered an iteration in an iterative-incremental process. An iteration addresses a common theme, which means that a fair amount of information has to be obtained during the preparation. The theme and goal of an iteration is defined during the kick-off. Each concrete issue is handled in a loop of gathering and analyzing ideas to the resulting detailed design, where the document is actually written. After deciding on the results, the closing step assures that the desired outcome has been reached and whether a following workshop is scheduled or needs to be scheduled.

It is important to make sure the right people participate in the workshop, and that these people only attend as long as their input is needed. According to Kylmäkoski, the workshops in Nokia are usually scheduled for three to five hours, and should not be longer to remain efficient.

2.3.2. Process T

At the time of writing, the process presented in this section is used by an international company in the telecommunications business. The company that implemented this process operates in several countries and provides a variety of different services to business and private customers. It owns networking infrastructure and continually strives to incorporate other telecommunication companies. While new businesses are acquired, the company has to adapt to technological changes, at the same time providing their products at a high standard of quality. Internal requirements documentation is essential to the company's core business.

I will call the process *Process T* (for telecommunication). This process is used for software design and development and tries to document all changes introduced to a system. *Process T* evolved over the years and is supported by professionally trained staff and process infrastructure, i.e. a sophisticated incident and change management system.

In terms of requirements, the process focusses on documenting functional software requirements and, on a lower level, design constraints.(see Section 1.3.3, “Classification”). A supplementary ambition is to document changes to productive systems.

Process Flow

At a first stage only the evolution of requirements is observed, up to the point where the information has sufficiently developed into a technical description of the software that can be used as a basis for the implementation. From the previously defined stakeholders (see Section 1.5.2, “Stakeholder Documentation”) the following are selected for simplicity:

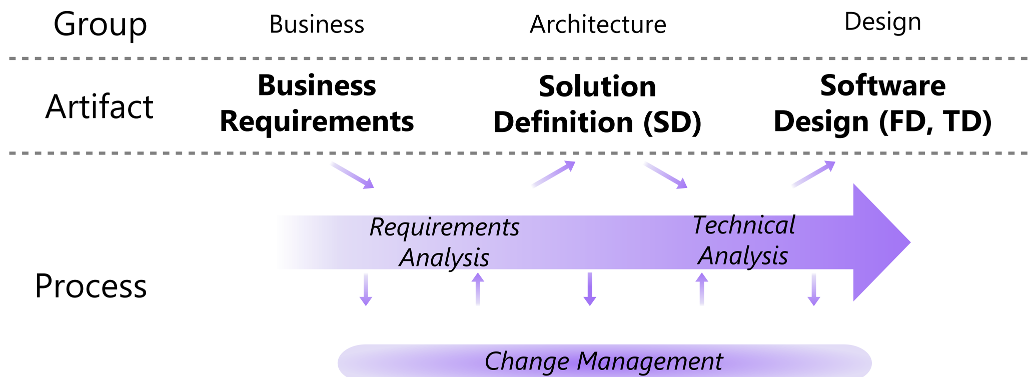
Business
: A combination of the people responsible for the corporate strategy and the customers

Architecture
: People in this group have a broad view of the solution and strategy. They correspond to *Architects and Analysts* in the description of stakeholders.

Design
: This heterogeneous group is comprised of experts from development and operation teams, the customer and software architects. The result of their work is the basis for implementation efforts.

Together, these parties try to assess the needs and transform them into a behavior specification as depicted in Figure 1.3, "Evolution of a Behavior Specification". Teams participate cooperatively in two work streams in order to eventually obtain software design documentation, i.e. the behavior specification for the resulting software.

Figure 2.2. Transformation of Requirements



Business requirements go through several changes before the software design is completed and ready to be implemented.

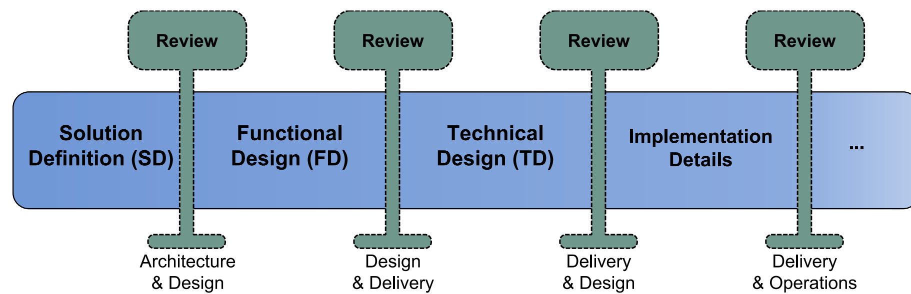
The result of the *requirements analysis*, conducted by business and architecture group is called the *Solution Definition* (SD). The SD is the basis for detailed design documentation. During the *technical analysis* the design group transforms this information into a specific software design for a specific component or module. During the *Technical Analysis* the SD is transformed into functional design (FD) and technical design (TD).

In parallel to the analysis work streams, an additional *Change Management* process has been established. This supportive process is necessary to orchestrate all changes in order to prevent or reduce negative impact. This process is described in more detail in the section called "Introduction of Changes" (p.33).

Reviews

Each documentation phase concludes with a handover review between the two groups involved. During these reviews, possible ambiguities or unclear items have to be identified before proceeding to the next step. See Figure 2.3, "Documentation Handover" for a graphical representation.

Figure 2.3. Documentation Handover



Before the final documentation has evolved, it has to pass four phases and three different participants have to agree on different levels of abstraction.

After the review, the participating groups prepare official sign off from their respective management. As soon as both parties agree on the amount of work, the documents are used as a basis for the commercial contract for the specific work stream.

Introduction of Changes

It is quite common that defects are found after an agreement has been signed and the participating parties have already agreed on a commercial contract. In *Process T*, a signed document can only be adapted after raising an appropriate *change request*.

Information in a Change Request

A change request holds information at different levels of abstraction and for different roles in a project.

Description
: A short, concise description helps to understand the business-related reason why this defect really is a defect. Usually an example describes a problem that appears at a certain stage of implementation or during operations.

Owner
: The owner of the change request is responsible for the changes to be implemented. The owner does not need to have the capabilities or expertise, but knows the person or group in charge to bring about the needed changes or a decision on the further proceeding. It is possible to forward the request to a different person or group, which, upon acceptance, becomes the new owner.

Log
: This information field contains notes from users involved in the change request. Their comments are saved, along with their user name and timestamp.

Linked Incidents
: It is possible to link one or more *incident descriptions* to the change request. An incident describes a single issue or symptom and can be compared to a bug report or ticket.

Solution
: One ore more change requests can be associated with a *solution*. A solution is a container that tracks information to solve the problem throughout its delivery/deployment. A single solution can solve one or more change requests. If the solution is implemented, the associated change requests are marked as implemented.

Meta-Information
: The change request can adopt different states. State changes, from its inception to its completion are tracked, together with the information which user triggered the state change. Some state changes require a short description and special privileges. The time of ownership for the different people or groups in charge can be tracked too.

Attained Goals and Maturity

In order to assess *Process T*, it is necessary to apply the previously defined goals (see Section 2.1, “Goals of Document Authoring for Requirements Documentation”) and maturity models (see Section 2.2, “Evaluation with Maturity Models”). This helps us to see where the implementation of the process can be improved.

Goals

The extent to which the previously defined goals for document authoring in the implementation of *Process T* are met can be assessed using a survey among participants involved in the process. For each of the detailed characteristics of the goals, it is possible to assign one to three points with the following meaning:

- *One point*: nothing or very little has been implemented,
- *Two points*: something has been implemented,
- *Three points*: most of the measures to reach the goal are in place.

With the results of the survey and the above simple rating mechanism, it is possible to obtain a very subjective assessment of the process based on the involved participants. I have not conducted such a survey, but further research in this area might be interesting to prioritize capabilities of the authoring framework.

Maturity Models Applied

The level of process maturity (see Listing “Levels in the Process Maturity Model”, p.28) for *Process T* is located between *inconsistent* and *defined* (levels two and three of four levels total). Documentation is written using standard word processing tools. Structured information from CASE tools are inserted manually into these documents and updated infrequently.

Quality assessment methods are in place, but they do not cover the full process and are not coordinated throughout the participating groups. Documentation reviews are mandatory and occur more frequently before handover than during the time the documentation is being developed. Although the formally structured character of the reviews is understood, it is difficult to review archived changes to the documentation in one central place.

The change management toolset helps in identifying changes to a work stream's overall solution but its nature does not allow a structured query for specific changes on a component or module basis.

Part II. Documentation Formats and Authoring Tools

The previous chapter observes the goals of document authoring for software requirements documentation and how the process of authoring and the resulting artifacts are evaluated.

The two following chapters create a basic understanding of the main components of the authoring framework: the *documentation format* and the *authoring tool*. For both components traditional and current implementations are analyzed, assessment criteria created, and a selected subset compare.

- Chapter 3, *Analysis of Documentation Formats*
- Chapter 4, *Analysis of XML-Based Authoring Tools*

Chapter 3. Analysis of Documentation Formats

I have previously outlined the goals of document authoring in the context of requirements documentation and how the process itself and the resulting documents can be evaluated. This chapter shows an overview of existing, widely employed document formats for requirements documentation. Subsequently, this chapter continues with current document formats and open standards, all of which are XML-based [XML1.0(e4)]. Then the formats are compare and their features, drawbacks and fields of application are summarized.

The term *format* is used throughout this document to narrow the idea of documentation systems and environments – or other synonyms used in the same context – to the underlying data format. A selection of specialized documentation tools for authoring documents are presented in Chapter 4, *Analysis of XML-Based Authoring Tools*. Those authoring tools have been selected to author documents presented in Section 3.3, “Current Document Formats and Standards”.

3.1. Restrictions

The field of documentation formats is large, even if the application is constrained to requirements documentation. I have narrowed the selection of documentation formats in this chapter, in order to keep within the scope of this thesis. Before going into details, the basis for the argumentation and the decisions being made is layed out. The following restrictions will be explained in more detail in this section:

- Document-centric
- Standardized
- Structured
- XML-based

3.1.1. Document-Centric

In a document-centric authoring model, the assembled information is put into one or more documents. In the context of web applications, Boyer [Boye08] describes several advantages of a document-centric approach of which two are applicable in the context of this thesis:

- Ability to digitally sign documents for legally binding agreements.
- Support for document-centric business processes, workflows, and activity-centric systems.

Toolset

In a survey conducted by Forward and Lethbridge [Forw02], 41 participants answered to a question regarding which software tools are most/least helpful to create, edit, browse and/or generate software documentation (see Table 3.1, “Useful Documentation Technologies”). Word and text processors emerged as the most helpful for software documentation, because these processors are flexible and in general easy to use.

Table 3.1. Useful Documentation Technologies

Documentation Technology	Useful	Least Helpful
Word processors	54%	15%
Javadoc and similar tools (Doxygen, Doc++)	51%	12%

Documentation Technology	Useful	Least Helpful
Text Editors	22%	7%
Rational Rose	12%	2%
Together (Control Centre, IDE)	7%	-

During the same survey, a question regarding the most important document attributes on a scale from 1 to 5, 5 being most important, the characteristic “Content – the document's information” received a mean rating of 4.85. Runner-ups in that category were “Up-to-date” with a mean of 4.35, and “Availability” and “Use of examples” with a mean of 4.19 each.

Wiki as an Alternative to Document-Centric

I do not cover wiki-based technologies or documentation approaches until the end of this document. Because they are widely used in corporate environments, this section should provide some pointers on how wikis may be used in the context of requirements authoring.

According to Xu's research on wikis in computer science course project management [XuLi07], wikis are considered “the latest web innovation on content management and sharing”, and a prime example of fulfilling the goal of *Sharing* in documentation authoring (refer to Section 2.1.2, “Share Information”). Xu further notes their application in teaching activities, being used for collective learning, online teaching, knowledge sharing and other elements of the previously identified goal *Gain understanding* (refer to Section 2.1.1, “Gain Understanding”). The advantage of employing wikis to support project management as documented by Xu matches my personal experience.

Developed as a plugin for the commercial wiki Confluence, the Scroll Wiki Exporter[1] combines features from joint authoring in a wiki environment and document-based export of selected content. It uses DocBook (see Section 3.3.1, “DocBook”) as backend to its exporter, and from this several other formats. The export involves: selection of the pages and hierarchy to be exported, addition of supplemental document data, transformation using the reusable docbook publishing process (refer to the section called “Toolchain” (p.49)).

In Section 7.1, “Validation” one of described the scenarios uses a wiki as primary documentation format.

3.1.2. Standardized and Open

There are many definitions of open standards[2], but I will use only one of them as an example: The *European Commission* defines open standards in version 1.0 of the European Interoperability Framework [IDABC04] as a specification including attending documents with the following minimal characteristics:

- The standard is adopted and will be maintained by a not-for-profit organisation, and its ongoing development occurs on the basis of an open decision-making procedure available to all interested parties (consensus or majority decision etc.).
- The standard has been published and the standard specification document is available either freely or at a nominal charge. It must be permissible to all to copy, distribute and use it for no fee or at a nominal fee.
- The intellectual property - i.e. patents possibly present - of (parts of) the standard is made irrevocably available on a royalty-free basis.

[1] At the time of writing, Scroll Wiki Exporter is in beta and available at its website [http://scrollyourwiki.com].
[2] Refer to Wikipedia's Definitions of Open Standards [http://en.wikipedia.org/wiki/Open_Standard#Specific_definitions_of_an_open_standard].

Drawbacks Through Lacking Standardization

There are several disadvantages to non-open formats. For example, in the world of office suites and applications “all documents are stored in a proprietary (often binary) format”[Eise05]. This might lead to one of the problems, explained in more detail below:

- Difficult information extraction
- Reduced Availability

Information Extraction

If the user wants transfer information from a document (e.g. into another format, export to a database) in a way not intended by the authors, the process will become difficult. It is necessary to “convert that data to some neutral […] format”. Conversion mechanisms have to be provided, and potential information loss due to different structure of the formats poses additional requirements.

Perpetual Availability

This kind of availability is orthogonal to the availability described in the section called “Availability” (p.25). It is about the possibility of technically accessing the information of the document at all: If the underlying documentation format evolves to that extent that the vendor develops a new format and stops supporting the old version, data can become inaccessible. In [Stuc05] reasons for this kind of decision are discussed. As there is no other party involved, the user depends on the vendor to deliver a new version with import filters for the old format.

The following example should visualize this problem

Example 3.1. Word Processor Version Incompatibility

A user writes a series of books in format F1 in the word processor W (version W1) on operating system O (version O1). While the user is writing the books, software company that created W adopts innovations and integrates them into new versions: W2, W3, W4. For different forementioned reasons, features are removed (W3 comes bundled with filters for F1 instead of native support, W4 does not support F1 at all) and additional requirements are imposed (W3 only runs on O2 or O3, W3 only runs on O3).

In Table 3.2, "Exemplary Requirements and Dependencies of a Word Processor" the aforementioned dependencies and requirements are depicted. If the user intends to upgrade to a newer version of his operating system (e.g. O3), he/she is forced to use a new version of W, supported by this operating system. Format compatibilities have to be taken into account too: the user cannot upgrade directly to version W4, as it does not support the original format F1 at all.

Table 3.2. Exemplary Requirements and Dependencies of a Word Processor

		Word Processor Version			
		W1	**W2**	**W3**	**W4**
Data Format	F1	S[i]	S[i]	F[ii]	-
	F2	-	S[i]	S[i]	F[ii]
	F3	-	-	S[i]	S[i]
	F4	-	-	-	S[i]
Operating System	O1	S[iii]	S[iii]	-	-
	O2	-	S[iii]	S[iii]	-
	O3	-	-	S[iii]	S[iii]

[i]The word processor can read/write natively from/to this format.
[ii]The word processor can use special filters to import/export from/to this format.
[iii]The word processor can run on this operating system.

Examples of Standards Support

In recent history, a strong trend towards supporting open standards has developed. This section shows some examples from the public sector and the industry.

European Union. In order to evaluate recommendations for document formats that allow flexible interchange between EU member's administrations, the European Commission has conducted the *Valoris Report* [Valo03]. After receiving responses from the industry, the *Telematics between Adminstrations Committee* approved conclusions and recommendations on the open document formats. The European eGovernment Services have consolidated all the related information on a single aggregation page [IDABC05].

US State of Massachusetts. The US State of Massachusetts backs ODF as the standard for office applications [Mass05] and planned to migrate until January 2007: "Agencies should begin to evaluate office applications that support the OpenDocument specification to migrate from applications that use proprietary document formats." This adoption of ODF spurred a lot of discussion in the legal and political domain[3]. Shah et al. [Shah08] analyze the policy shift towards open standards and observe lessons learned from this change.

Sun Grid Service. In November 2005 Sun Microsystems announced [Sunw05] that the company has added new services to the service. One of them, the Sun Grid Utility service is intended to "help simplify the process of converting documents from Microsoft Office to free and open al-

[3]Groklaw has set up an aggregation page [http://www.groklaw.net/article.php?story=20050330133833843] with pointers and comments on this subject.

ternatives that radically lower cost, promote cross-platform communications and help users with older versions of Microsoft Windows avoid the costs and risks associated with deploying a newer release of Windows."

3.1.3. Structured

Walsh and Muellner [Wals05] say that structured authoring has significant shortcomings:

- The authoring process for structured documents is very different from writing with a word processor. Authors do not have direct control over the appearance of their content.
- Authoring tools for structured documents can be more expensive than word processors. They are less popular and have a smaller user base.

However, the research in the field of structured document authoring[4] has found many advantages over (traditional) context-free authoring. Dymetman [Dyme04] claims that, by using chart-parsing techniques, the author may "state knowledge about the document in a flexible way" and can mix top-down and bottom-up authoring, while "the system automatically detects the consequences of the choices already made [and proposes] live choices for the next authoring step."

3.1.4. XML-Based Solutions

An XML document combines the previously mentioned restrictions into a compound requirement:

Centered around a document
: From the W3C recommendation [XML1.0]: "[XML] describes a class of data objects stored on computers and partially describes the behavior of programs which process these objects. Such objects are called XML documents."

Based on standards
: XML itself is being developed as a standard and recommended by the W3C since 1998 [XML1.0] (version 1.0). The specification builds on several other standards: Unicode and ISO/IEC 10646 for character strings, and Internet RFC 3066, ISO 639, and ISO 3166 for language and identification and country name codes.

  With the addition of semantic constraints, application and domain-specific languages can be implemented. Many of today's important communication standards are built on XML[5].

Structured
: XML provides a well-defined markup with tags, references, comments, processing instructions, and special declarations (e.g. for document types, character data, and XML declaration). According to Norman Walsh [Wals05] "XML is a natural system for storing structured documentation".

With a first draft in November 1996 [XML-WD], and the 1.0 release in 1998 [XML1.0], at the time of writing, the standard is in the fourth edition [XML1.0(e4)]. There are many stable XML-based standards powerful enough to deliver to the expectations of a document format or document standard. This section outlines the general characteristics of an XML-based format. Each XML-based document format can benefit from these characteristics, and most implement the one or the other standard way to achieve certain goals.

After a short digression into alternatives to XML. the remainder of this section will cover the following advantages of XML:

- Standardized inclusion methods
- Human readable markup

[4] In the literature, structured document authoring is sometimes referred to as *controlled* document authoring.

[5] Refer to the Wikipedia category XML-based standards [http://en.wikipedia.org/wiki/Category:XML-based_standards]

- Separation of content and style
- Extensibility
- Wide acceptance

Alternatives to XML-Based Formats

Most of the advantages outlined in this section, also apply to alternative structured and human readable formats. Two alternatives for certain fields of application of XML are JSON[6]and YAML[7]. These two languages are primarily used for serialization and data-interchange. Because these alternatives are not document markup languages, their use for requirements documentation authoring is very limited.

Standardized Inclusion Methods

There are many different, standardized possibilities to include external information in an XML document: XLink [XLink], XInclude [XIncl] External Entities. They have different advantages and expectations in terms of implementation. With standard ways of inclusion, document format designers can leverage the same technologies and choose whatever they want.

For example, when using XSL stylesheets in order to style a DocBook document, for Bob Stayton [Stay03] “the best tools are XIncludes and olinks”. XInclude is a standard inclusion mechanism for XML documents. The `olink` element is the “equivalent for linking outside the current document”. The DocBook XSL stylesheets provide different options for resolving `olinks`, depending on the content type of the linked element.

Iacob and Dekhtyar [Iaco05] distinguish between *document-centric* and *data-centric* XML documents. Although in a different context, this distinction is essential to categorize the different document formats that we will present in Section 3.3, “Current Document Formats and Standards”. These two types are defined as follows, quoting from Jacob and Dekhtyar for their definition:

Data-centric
: “Characterized by fairly a regular structure”, this type of document format provides one main document that links to or includes all other elements to provide the information intended by the author.
  - DocBook (Section 3.3.1, “DocBook”)
  - DITA (the paragraph called “DITA” (p.55))

Document-centric
: Document formats in this category have a “much more irregular structure”. These formats need meta-information to combine the different elements into a single document.
  - OpenDocument (Section 3.3.2, “OpenDocument”)
  - Office Open XML (Section 3.3.3, “Office Open XML”)

Human Readable Markup

The XML markup is at its base very simple. The contents can be read with any text editor, provided that the fonts for the document's locale are installed. Limitations at the level of hard disk space or memory consumptions are not as driving as they were some years ago. This is supported by the following quote from the Office Open whitepaper [OOXML-WP]:

> The original binary formats [...] were created in an era when space was precious and parsing time severely impacted user experience. [...] Modern hardware, network, and standards infrastructure (especially XML) permit a new design

[6]Refer to the JSON website [http://www.json.org]
[7]Refer to the YAML website [http://yaml.org]

> that favors implementation by multiple vendors on multiple platforms and allows for evolution.

Content that need not be presented as text to the author (e.g. images, interactive elements ...) does not have to be converted or encoded in order to be serialized with the XML document. The document formats covered in Section 3.3, “Current Document Formats and Standards” allow for extensible linking and embedding mechanisms.

Separation of Content and Style

The underlying basis of XML is at the separation of document structure and presentation style [Bae02]. The document structure and the content are directly visible through the tree structure of an XML document, but in order to present the contents of an XML document, *styling* mechanisms are employed.

For documents in XML's predecessor SGML, style was an important issue, but due to the high number of solutions, “not even the standard ones were widely deployed [, thus] style was an unsolved issue”[Quin04]. In XML, “only two style languages are widely accepted and deployed”: the Extensible Stylesheet Language (XSL) [XSL] and Cascading Stylesheets (CSS)[8].

Designers can use stylesheets to express their intentions about how the content of an XML document should be presented. A stylesheet processor accepts the document in XML and uses a stylesheet to produce a specific presentation of this document. There are to aspects to stylesheet processing [XSL]:

Tree transformation
: is the construction of a result tree from the XML source document.

Formatting
: is the interpretation of the result tree to produce formatted results suitable for presentation (on whatever media).

For a documentation format, the style can be used to adapt the presentation of the content for different purposes. Some examples:

- Comply with interface guidelines to support a wide variety of readers.
- Adhere to the corporate identity of a company, group, or team.
- Prepare the content for accessibility, e.g. special reading interfaces, text-to-speech applications ...

Extensibility

As Quint and Vatton note [Quin04], XML added not only a simplification to its predecessor SGML, but also introduced new concepts and languages specifically developed with XML. With the introduction of namespaces [XMLNS], according to Quint and Vatton, an XML document can “mix several markup languages that represent different parts of the document [which] can be nested within each other” providing a modular approach for structured documents.

With the extensibility provided by namespaces, XML languages can be developed for different types of structured data and combined to compound structured documents.

Wide Acceptance

It is important to notice that many applications, frameworks and standards currently build on XML as a “de facto standard for structured documents”[Chid03] and information exchange in IT applications and systems. Other sources also acknowledge the wide use of the format [Ande02].

[8] Refer to Cascading Style Sheets Specifications [http://www.w3.org/Style/CSS/#specs] at w3.org.

The website *XML Coverpages* provides a[9] provide a comprehensive list of XML applications and initiatives. Currently the website references 594 different XML languages.

Summary

There are several advantages through the use of XML-based formats, as this section has shown. The following list is a set of advantages that all the formats in Section 3.3, “Current Document Formats and Standards” have in common. Some document formats leverage these feature differently, but they are provided by the choice of data format - XML.

Modularity
: The document can be broken up into multiple sections. The combination into a single document is performed by the parser or during transformation at the publication stage.

Version control
: As a content-specific format, differences between two revisions of a document or a section are not more than a *diff* output. The version control system does not have to "know" about the document format, but only store the textual differences. The differences are only content-related, as formatting information is stored in style sheets.

Consistent formatting
: With the separation of content and style, formatting changes are applied to the entire document. A style sheet can be used across different documents and evolves with more specific applications.

Publish to multiple formats
: It is possible to apply different (standardized) style sheets to the same source. Each style sheet may produce a different output presentation. Terms often used in this context are: *Single-Source Publishing* and *Documentation Reuse*.

3.2. Traditional Document Formats

The following formats are being used to create software documentation. The descriptions follow a common structure, in order to present the formats: After a short introduction, common fields of application and an overview of the format's structure or technology is shown.

3.2.1. Word Document Format

Microsoft Word[10] uses this format to save its information. As the authoring tool and the file format share the same name, I will use the term *Word Doc* for the document format and *Microsoft Word* for the authoring tool throughout this document.

Format using Word Doc

With the evolution of Microsoft Word, the file format changed as well. The HWPF project[11], a port of the Microsoft Word file format to pure Java, supports versions 97 to 2007 (no OOXML). See Example 3.2, “Simple WordML document” for version 2003.

[9] list of XML applications [http://xml.coverpages.org/xmlApplications.html]
[10] Microsoft Word website [http://www.microsoft.com/office/word/]
[11] The project website [http://jakarta.apache.org/poi/hwpf]

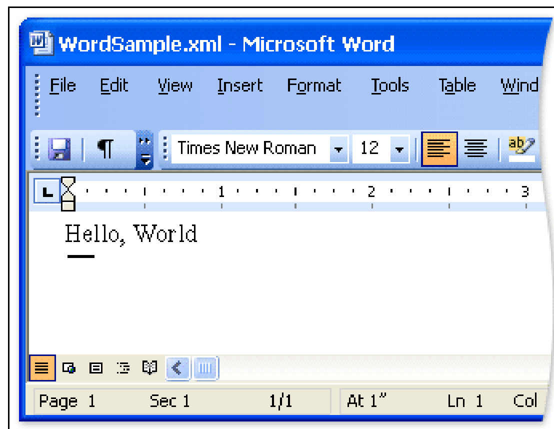
Example 3.2. Simple WordML document

```
1 <?xml version="1.0"?>
2 <?mso-application progid="Word.Document"?>        ❶
3 <wordDocument>                                     ❷
4   <body>
5     <p>
6       <r>
7         <t>Hello, World.</t>
8       </r>
9     </p>
10   </body>
11 </wordDocument>
12
```

❶ The `mso-application` processing instruction specifies Word as the preferred application for processing the file. Even if saved with the extension `.xml` Windows' shell will try to open the file with Word when it is double-clicked.

❷ The namespace is: `http://schemas.microsoft.com/office/word/2003/wordml`

Figure 3.1. Simple WordML document displayed in Microsoft Word



Rendering Example 3.2, "Simple WordML document" in Microsoft Office Word (© by Microsoft[12]).

Recent History for Microsoft Word

At the time of writing, Microsoft Word 2007 is the most current version. Its predecessor Microsoft Word 2003 introduced an XML-based format with attached license terms for the schemas[13], which were publically criticized and adapted shortly thereafter. They are no longer available, because the format has been replaced.

For Microsoft Word 2007 a new file format was announced in 2005 [Shah08]: Office Open XML[14]. It is intended to replace its predecessor's formats (the newly introduced XML format for Microsoft Word 2003 and Word Doc). Refer to Section 3.3.3, "Office Open XML" for more information.

Additional Information on the Word Document Format

Adoption of Word Doc. Although no non-biased numbers could be found, it is common belief that Microsoft Word is currently the most widely used word processor and has a very large

[12]Found in an article on MSDN [http://msdn.microsoft.com/library/en-US/wordxmlcdk/html/cdkPrimerPlaceholder_HV01113631.asp]
[13]Provided at a dedicated web page [http://www.microsoft.com/mscorp/ip/format/xmlpatentlicense.asp]
[14]Refer to the OOXML overview page [http://www.microsoft.com/office/xml] and navigate to *Open XML policy briefing* for more information on the licensing changes.

market share. Based on my personal experience, I can say that most companies (i.e. more than 90%) write their requirements documentation in Word Doc, especially, if they want others to add notes or author separate parts of the document.

Authoring in Word. The application supports the *ad-hoc corrections* pattern [Copl04], which suggests that a master copy of the documentation be kept and that team members write corrections in the margin. One team member is assigned to periodically update the online copies to reflect the corrections. The same pattern is also followed in current versions of Word.

Rich Text Format. The *Rich Text Format* (RTF) is a document markup language[15] developed by Microsoft for cross-platform document interchange[16]. RTF has been updated to support improvements in the evolution of Microsoft Office, the most recent update was in 2004 [RTF-1.8] and introduced changes for Microsoft Office Word 2003. The format was intended to provide a format for text and graphics interchange that can be used with different output devices.

3.2.2. TeX

According to Walsh [Wals94] *TeX*[17] is a typesetting system, a "collection of programs, files, and procedures for producing professional quality documents with minimum effort". Compared to word processors, its strengths are useful for large documents which must be formatted similarily [Salz05]. *LaTeX* is a "macro package [that] uses the TeX formatter as its typesetting engine"[Otei08].

Format using LaTeX

The input file to LaTeX is a plain text file that contains the text of the document and commands used to typeset the text. *Special characters* have a meaning in the system and may be escaped by being prefixed with the backslash character. The backslash character also starts LaTeX commands.

The processor expects a certain structure [Otei08] and order of commands to produce a document. Refer to Example 3.3, "Minimal LaTeX Input File" for a very short example.

Example 3.3. Minimal LaTeX Input File

```
1 \documentclass{article}  ❶
2 \begin{document}        ❷
3 Hello, World.           ❸
4 \end{document}          ❷
```

❶ The document class specifies what sort of document is used. Usually this command is followed by others that influence the style of the whole document.

❷ The document is divided into blocks, the document itself being the top block. Blocks start and end with appropriate commands.

❸ This is the actual content of the document. It contains text and other LaTeX commands.

Existing commands may be extended, or new commands created, and bundled into packages. The packages can then be used to allow for consistent formatting across multiple documents.

Styling is "implemented by defining a specific command or environment for each document element that has to be treated specially"[Mitt04].

[15] Refer to the Wikipedia entry on comparison of document markup languages [http://en.wikipedia.org/wiki/Comparison_of_document_markup_languages].

[16] Further information on RTF can be found at the corresponding Wikipedia page [http://en.wikipedia.org/wiki/RTF].

[17] Traditionally the name is written with a subscript capital *E*: T_EX, but I follow the accepted notation with a lowercase *e*.

Additional Information about TeX

Adoption of TeX. According to Salzberg and Murphy [Salz05], TeX is “the standard format for some academic disciplines [and] required for a variety of journals”. According to my experience, although providing many advantages (multi-platform, programmable, free) compared to other authoring systems, it did not find wide acceptance in the industry.

3.3. Current Document Formats and Standards

This section provides a selection of current standards based on XML that can be used for documentation purposes. The selection follows Tim Bray's claim[18]:

> The value of a markup language is proportional approximately to the square of the number of different software implementations that can process it.

Thus, this section focuses on document formats that already exist and are already in use or have a high probability of being used in the near future. The document formats have to conform to the restrictions explained in Section 3.1, “Restrictions”.

3.3.1. DocBook

DocBook is a collection of open standards and tools originally developed for technical documentation [Stay03][Baye03]. The DocBook schema is maintained by the DocBook Technical Committee in OASIS[19] and exists since 1992. It is available in the following formats [Wals05]:

- SGML
- XML DTD
- RELAX NG Grammar
- W3C XML Schema

DocBook Versions

The versioning syntax in DocBook is always *DocBook VX.Y(.z)*, where `X` is the full version number, and `Y` is the major version number. The minor version number `z` is optional in the version notation. Backwards incompatible changes can only occur at full version revisions (4.0, 5.0, 6.0), while minor revisions are always backwards compatible.

Historically DocBook was an SGML DTD. The explosion of markup languages after 1998 [Rene02] affected the format's development and resulted in a the first DocBook XML DTD in version V4.1.2. The DocBook DTD for V4.2 was released for both SGML and XML [Wals05]. While DocBook V4.x is an XML or SGML DTD, it has an unofficial RELAX NG Grammar and an unofficial W3C XML Schema. DocBook V5.0 is a RELAX NG Grammar and has a non-normative XML DTD and a non-normative W3C XML Schema [Wals05].

DocBook V3.0 marked the beginning of a parametrized, highly customizable DTD. DocBook V4.0 introduced case-sensitive element and attribute names, and DTD was introduced as an official, normative format. As DocBook predates XML and namespaces, the DocBook versions prior to V5.0 did not have any associated namespace. DocBook V5.0 is in the namespace `http://docbook.org/ns/docbook`. The usage of RELAX NG now enforces many constraints that could not previously be enforced by DTDs.

[18]Article Don't Invent XML Languages [http://www.tbray.org/ongoing/When/200x/2006/01/08/No-New-XML-Languages], Bray 2006
[19]Website of the TC [http://www.oasis-open.org/docbook/]

Structure

DocBook elements can be divided into the following categories [Wals05]:

Sets
: A `set` comprises two or more books. It is the hierarchical top of DocBook and can be used for a series of books on a single subject.

Books
: The DocBook definition of a `book` is very loose and general.

Divisions
: Divisions are the first hierarchical level below `book`. They are usually `parts` or `references`.

Components
: Components are the chapter-like elements of `books`. They are used to further subdivide books or divisions: `preface`, `chapter`, `appendix`, `glossary`, `bibliography` …

Sections
: There are several possibilities of sectioning elements in DocBook. The simple `sect1`…`sect5` or `section` elements can be used in most component-level elements. They support nesting. Sections used for divisioning of special components (`glossdiv`, `bibliodiv`, `indexdiv`) do not nest.

Meta-Information
: All elements at the section level and above include a wrapper for meta-information (see Example 3.4, "Meta-Information on a Book in DocBook Syntax" for a `bookinfo` example).

Block Elements
: These elements occur directly below the component and sectioning elements and are usually presented with a break before and after them. They can be divided into several categories: lists, admonitions, synopses, tables, figures, examples etc.

Inline Elements
: Inline elements are generally represented without any obvious breaks. They never contain block elements and are used to mark up data such as cross references, filenames, commands, etc.

In Example 3.4, "Meta-Information on a Book in DocBook Syntax" you can see the meta-information associated with a `book`.

Example 3.4. Meta-Information on a Book in DocBook Syntax

```
<bookinfo>
  <title>Macbeth</title>
  <author>
    <personname>
      <firstname>William</firstname>
      <surname>Shakespeare</surname>
    </personname>
  </author>
  <abstract>
    <para>
      Promised a golden future as ruler of Scotland by three sinister
      witches, Macbeth murders the king to ensure his ambitions come
      true. But he soon learns the meaning of terror — killing once,
      he must kill again and again, and the dead return to haunt him.
      A story of war, witchcraft and bloodshed, 'Macbeth' also depicts
      the relationship between husbands and wives, and the risks they
      are prepared to take to achieve their desires.
    </para>
  </abstract>
</bookinfo>
```

Information in DocBook on the play Macbeth by William Shakespeare formatted as a book

Toolchain

The term *toolchain* is frequently used in the DocBook domain, because in the long history, a large set of different interchangeable tools has evolved. The following steps will be explained in more detail below.

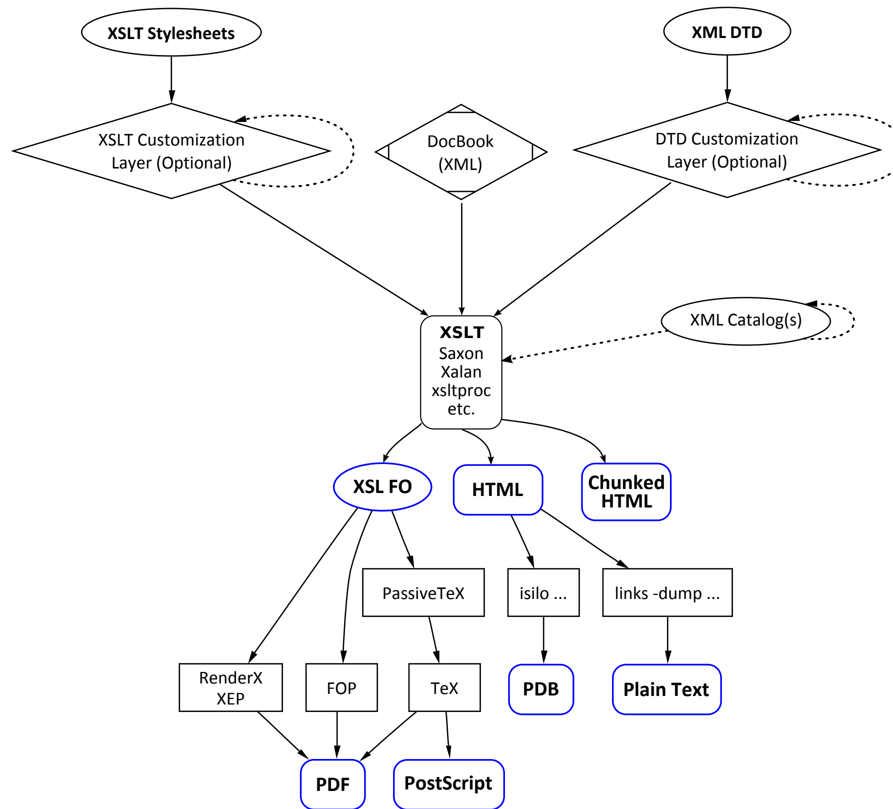
- Parse
- Transform
- Publish

The process is similar for other XML-based formats, but in the case of DocBook the involved steps are always transparent for the user.

Norman Walsh has put together a description of the XML publishing model for DocBook[20]. It shows popular and widely used tools in their context. With minor visual adaptions of the model, I have reduced the set to only the relevant parts for publishing using XSLT: see Figure 3.2, "DocBook Publishing Model with XSLT"

[20]Available in different formats at the dedicated webpage [http://nwalsh.com/docbook/procdiagram/]

Figure 3.2. DocBook Publishing Model with XSLT



DocBook is designed to publish to different media from a single source (based on the DocBook Publishing Model[20] by Norman Walsh)

Elements of the DocBook Toolchain

Parse. There are many different parsers for XML documents. It is recommended that a *validating* parser in the toolchain supports the normative DocBook syntax format [Wals05] (SGML or XML DTD for V4.x and RELAX NG [RELAX] for V5.x). During parsing, it is possible to combine modular elements to a single document by using different XML-related technologies (see the section called "Standardized Inclusion Methods" (p.42) for more information). DocBook V5.0 bundles alternative schemas for explicit use with XInclude.

Transform. The tranformation process applies style sheets to the content and results in a certain presentational output. It is possible that several transformation routines have to be processed subsequently in order to receive the anticipated result. As you can see in Figure 3.2, "DocBook Publishing Model with XSLT", HTML is an output format that can be used to create plain text, which itself is another output format. The DocBook XSL stylesheets are a large collection of files that can be downloaded from the project's website[21], where they are maintained. They have a wide acceptance and support many different output formats. They also provide extensions for XSL and FO processors.

Publish. By using different tools and stylesheets DocBook has a high diversity of possible output formats. The official DocBook documentation [Wals05] shows examples for three different languages used for publishing: CSS, XSL, and XQuery. These are described in more details in the next section.

[21]DocBook page at SourceForge [http://docbook.sourceforge.net/]

Styling in DocBook

The *style* and *publish* steps generate the output of a DocBook document. This section presents two examples of stylesheets to format the DocBook snippet in Example 3.5, “Simple Paragraph using the DocBook Syntax”.

Example 3.5. Simple Paragraph using the DocBook Syntax

```
1 <para>
2 This is an example paragraph. It should be presented in a reasonable body font.
3 <emphasis>Emphasized</emphasis> words should be printed in italics. A single
4 level of <emphasis>Nested <emphasis>emphasis</emphasis> should also be
5 supported.</emphasis>
6 </para>
7
```

The stylesheet using CSS in Example 3.6, “Stylesheet using CSS” sets formatting properties, while the XSL stylesheet in Example 3.7, “Stylesheet using XSL” produces an XML snippet using XSL FO.

Example 3.6. Stylesheet using CSS

```
 1 para ❶            {
 2  ❸ display: block
 3 }
 4 emphasis ❶        {
 5  ❸ display: inline;
 6  ❸ font-style: italic;
 7 }
 8 emphasis emphasis ❷ {
 9  ❸ display: inline;
10  ❸ font-style: upright;
11 }
12
```

❶ Simple selectors for `para` and `emphasis`.
❷ Selects only `emphasis` nested in `emphasis`.
❸ Formatting properties, setting values for `display` and `font-style`.

Example 3.7. Stylesheet using XSL

```
 1 <?xml version='1.0'?>
 2 <xsl:stylesheet>   ❶
 3   <xsl:template match="para">                    ❷
 4     <fo:block>     ❶
 5       <xsl:apply-templates/>
 6     </fo:block>
 7   </xsl:template>
 8
 9   <xsl:template match="emphasis">                ❷
10     <fo:sequence font-style="italic">
11       <xsl:apply-templates/>
12     </fo:sequence>
13   </xsl:template>
14
15   <xsl:template match="emphasis/emphasis">       ❷
16     <fo:sequence font-style="upright">
17       <xsl:apply-templates/>
18     </fo:sequence>
19   </xsl:template>
20 </xsl:stylesheet>
21
```

❶ The namespaces used in this example are `http://www.w3.org/XSL/Transform/1.0` for `xsl` and `http://www.w3.org/XSL/Format/1.0` for `fo`.

❷ The XSL stylesheet uses XPath [XPath] to select the nodes. The last example of this selects `emphasis` nested in `emphasis`, just as the CSS counterpart.

3.3.2. OpenDocument

The term *OpenDocument* refers to the Open Document Format for Office Applications (ODF), and defines an XML schema for office applications and its semantics [ODF]. It is intended to provide an open alternative to proprietary document formats.

The format is based on the development efforts by OpenOffice.org (OOo), an open source project that created an office suite with the same name (refer to the paragraph called “OpenOffice.org Writer” (p.59) for more details).

Standardization

For the designers of the ODF it was essential to reuse existing open XML standards whenever they are available. At the time of writing there are many office applications, search engines and document management applications that support OpenDocument[22].

The OpenDocument 1.0 specification was approved as an OASIS standard in May 2005 [Open05]. Tim Bray commented on the approval and said about the ODF that it is “the only XML office document format that is standardized, and it is also the only one that is complete”[23]. In that same comment, the co-author of the XML specification writes the following: “OpenDocument is almost exactly what we had in mind when we built XML, starting back in 1996.”

[22]Refer to a current list of applications that support the ODF at the Wikipedia page on OpenDocument [http://en.wikipedia.org/wiki/OpenDocument_software]. In May 2008 Microsoft announced support for ODF 1.1 in Microsoft Office 2007 SP2 [http://www.microsoft.com/Presspass/press/2008/may08/05-21ExpandedFormatsPR.mspx].

[23]Article OpenDocument! [http://www.tbray.org/ongoing/When/200x/2005/05/26/OpenDocument], Bray 2005.

Structure

The misson for OOo was to create “the leading international office suite that will run on all major platforms and provide access to all functionality and data through open-component based APIs and an XML-based file format”. The format represents an “idealized” representation of the document's structure[24] which is human-readable.

Package

Artifacts in ODF are packaged in a compound document (refer to the paragraph called “Compound Document” (p.55)). The manifest is located at `META-INF/manifest.xml` and holds the following pieces of information [ODF]:

- Listing of all files inside the package
- Media type of each file in the package
- Encryption information for encrypted files inside the package

3.3.3. Office Open XML

The release of Microsoft Office 2007 introduced an open XML-based format: *Office Open XML File Formats* (*OOXML*, in the literature sometimes referred to as *OpenXML*). The whitepaper available prior to the release[25] said: “Microsoft® Office 12 will introduce new default XML file formats for Microsoft Office Word word processing, Excel® spreadsheet, and PowerPoint® presentation graphics programs [...]”.

The format was officially announced in May 2005 as the default file format for these applications[26]. It was developed to support every feature in the Microsoft Office 97-2003 binary formats.

Standardization

OOXML was submitted to the standards body Ecma International in November 2006, which commissioned the Technical Committee TC45 to work on the standard. TC45 published the standard as *Ecma 376* [OOXML] in December 2006. In parallel to working on the standard, the authors provided a whitepaper [OOXML-WP] which states the following purposes for the task of creating this format as an open standard:

- extremely broad adoption of the binary formats
- technological advances
- market forces that demand diverse applications
- increasing difficulty of long-term preservation

Advantages over Previous Office Formats

The official announcement outlines advantages of OOXML over the traditional office formats.

Open and Royalty-Free
: The Office XML Formats are based on XML and ZIP technologies, thereby making them universally accessible. The specification for the formats and schemas will be published and made available under the same royalty-free license that exists today for the Microsoft

[24]Details of the design decisions are available at the dedicated webpage [http://xml.openoffice.org/xml_advocacy.html].
[25]Published at the Office preview page [http://www.microsoft.com/office/preview/], June 2005.
[26]Refer to History of office XML formats (1998-2006) [http://blogs.msdn.com/brian_jones/archive/2007/01/25/office-xml-formats-1998-2006.aspx] on Brian Jones' blog for a comprehensive overview on the history. Brian Jones is program manager in Microsoft Office and has been working on OOXML.

Office 2003 Reference Schemas and which is openly offered and available for broad industry use.

Interoperable
: With industry standard XML at the core of the Office XML Formats, exchanging data between Microsoft Office applications and enterprise business systems is greatly simplified. Without requiring access to the Office applications, solutions can alter information inside an Office document or create a document entirely from scratch by using standard tools and technologies capable of manipulating XML.

Robust
: The Office XML Formats have been designed to be more robust than the binary formats, and, therefore, will reduce the risk of lost information due to damaged or corrupted files. Even documents created or altered outside of Office are less likely to corrupt, as Office programs have been designed to recover documents with improved reliability by using the new format.

Efficient
: The Office XML Formats use ZIP and compression technologies to store documents. This type of file compression offers potential cost savings as it reduces the disk space required to store files and decreases the bandwidth needed to transport files by way of e-mail, over networks, and across the Web.

Secure
: The openness of the Office XML Formats translates to more secure and transparent files. Documents can be shared confidently because personally identifiable information and business sensitive information, such as user names, comments and file paths, can be easily identified and removed. Similarly, files containing content, such as OLE objects or Visual Basic® for Applications (VBA) code can be identified for special processing.

The above list of advantages highlights the need for open, flexible, and accessible formats. All these advantages are applicable to all open xml-based formats described in this chapter. To describe the point *interoperable*, the whitepapers give the following example:

> With previous Office versions, developers looking to manipulate the content of an Office document had to know how to read and write data according to the structured storage defined within the binary file. This process is known to be complex and challenging, notably because the Office binary file formats were designed to be primarily accessed through the Office programs. The formats were structured to mirror the in-memory structures of the applications and to run on low memory machines with slow, hard drives. Altering Office binary files programmatically without the Office applications has also been identified as a leading cause of file corruption, and has deterred some developers from even attempting to try to make alterations to the files.

With the introduction of this format, it is technically possible to create and edit Microsoft Office documents without Microsoft Office applications.

Structure

The Open XML format consists of several individual `parts` inside a compound document (refer to the paragraph called “Compound Document” (p.55)). The files have an additional `x` at the end of their file name to indicate the different file format: `doc` becomes `docx`, and `xls` becomes `xlsx`. The parts can be extracted using traditional unzip methods.

Modular parts can be of different content types. Traditional office document content is stored in an XML format conforming to a certain XSD. Different worksheets for an Excel workbook are located in different XML parts, where every part conforms to the XSD for worksheets. Other information, e.g. images, are stored in their native binary format. This reduces processing time and space compared to encoding the data in XML.

The semantics of a document are managed by *relationships* which specify the connection between a part and a target resource. The following program listing shows relationships for an Excel worksheet:

Example 3.8. Relationships for an Excel Worksheet

```
 1 <Relationships>                                      ❶
 2   <Relationship ID="rId3"
 3       Type="#prefix#/xlStyles"                       ❷
 4       Target="styles.xml"/>
 5   <Relationship ID="rId2"
 6       Type="#prefix#/xlWorksheet"                    ❷
 7       Target="worksheets/Sheet2.xml"/>
 8   <Relationship ID="rId1"
 9       Type="#prefix#/xlWorksheet"                    ❷
10       Target="worksheets/Sheet1.xml"/>
11   <Relationship ID="rId5"
12       Type="#prefix#/xlMetadata"                     ❷
13       Target="metadata.xml"/>
14   <Relationship ID="rId4"
15       Type="#prefix#/xlSharedStrings"                ❷
16       Target="strings.xml"/>
17 </Relationships>
18
```

❶ The namespace for `Relationships` and its children is: `http://schemas.microsoft.com/package/2005/06/relationships`

❷ The `#prefix#` used in the `Type` attribute resolves to the following URL `http://schemas.microsoft.com/office/2005/8/relationships`

References are not only limited to resources internal to the document. A relationship can point to an external resource. The meta-information defines the type of the resource. This information can be used to identify potential security issues and take appropriate steps without knowing the actual content. Most current email clients provide a setting to block external images. Instead of stripping potentially dangerous files from email (which is done by many corporate mail filters), this same approach could be taken for office documents

The whitepapers present more information on macro-enabled files, content manipulation, styling and other topics. The material presented in this section should cover the basic concepts.

3.3.4. Other Structured Formats

DITA. The Darwin Information Typing Architecture (DITA) is an XML-based framework for the production and publication of technical documentation [Krav05]. It defines document types for authoring and organizing topic-oriented information and mechanisms for combining and extending document types [DITA].

3.3.5. Common Documentation Format Concepts

The current documentation formats presented in this chapter share some design concepts, which are presented for further reference in this section.

Compound Document. Because XML has no native support for non-character based media types, and because uncompressed XML files can become very large, a package file is used to store the XML content and the associated binary data in the *document-centric* formats presented in this chapter. This approach is called a *compound document*. The packaging relies on gzip compression and adds to a reduction in space used by the document. Depending on the format, a manifest file manages the relationships and is provided at a certain location inside the archive.

Inline Versioning. Another characteristic of *document-centric* formats is that the documents are designed to work in isolation from any infrastructure except the document processor (i.e. the editing component or authoring tool). Changes to the document can thus be managed inside the document. This is referred to as *inline versioning*[27]. The versioning information is directly added to the changed content of the document. Common types of changes are insertions, deletions, and format changes.

3.3.6. Summary of Current Documentation Formats

Based on the previously defined restrictions, the current documentation formats presented in this chapter are summarized in Table 3.3, “Overview of Current Documentation Formats”.

Table 3.3. Overview of Current Documentation Formats

Format	Document Type	Styling	Standardization
DocBook	Data-centric	External	Version 4.1: OASIS (2001), Version 4.5: OASIS (2006)
ODF	Document-centric	Integrated	Version 1.0: OASIS [ODF] (2005), ISO/IEC 26300 (2006); Version 1.1: OASIS 2007 [ODF-1.1]
OOXML	Document-centric	Integrated	Ecma-376 (2006), ISO/IEC 29500 (2008, registered for formal approval)
DITA	Data-centric	External	Version 1.0: OASIS (2005), Version 1.1: OASIS (2007) [DITA]

[27] In OOXML this is called *Revisions* (section 2.14.7 of [OOXML] part 3), and in ODF this is called *Change Marks* (section 4.6 of [ODF] and [ODF-1.1]).

Chapter 4. Analysis of XML-Based Authoring Tools

There are already different efficient approaches for editing XML-based, structural information. As first step, I will analyze what an authoring tool for requirements documentation in the context of this thesis has to provide. In the field of authoring tools for structured XML documents, there are already several existing solutions from which a number of commercial and open source tools are selected based on their design, and presented in this section. The conclusion of this chapter provides a comparison for further reference.

4.1. Capabilities of an Authoring Tool for Requirements Documentation

All of the requirements of the authoring tool are based on previous research. First, the sources of the requirements are identified and then summarized to be used in the comparison at the end of the chapter. The literature often refers to the authoring tool as *editor* or *documentation tool*.

4.1.1. Sources of Requirements for the Authoring Tool

The identified requirements used to assess the authoring tools come from two different areas: *documentation engineering* with its general goals for document authoring, and authoring in the field of *structured documents*.

Documentation Engineering

In a survey conducted by Forward and Lethbridge [Forw02] on documentation engineering, results suggest that technologies should

- allow the author to easily create and maintain content rich documents,
- provide comprehensive publishing capabilities,
- facilitate the integration of examples within a document.

From the elaboration in Section 2.1, “Goals of Document Authoring for Requirements Documentation” several requirements are selected which have to be provided by the authoring tool:

- Provide transparent versioning.
- Allow for collaborative authoring.
- Facilitate linking between and inside documents.

Authoring of Complex Structured Documents

Quint and Vatton [Quin04] reviewed the innovations of XML and their impact on the editing techniques for structured documents and present features they have implemented in editors (in this thesis' context: authoring tools):

- Support multiple document types and XML languages.
- Edit documents according to the semantics of the XML language involved.
- Simple means for inline structuring (e.g. **Enter** and **Tab** keys).
- Structure transformations comparable to editing in a word processor.
- Transclusion of external resources.

The second item implies additional expectations not previously mentioned. Quint and Vatton present two features implemented in editors to achieve this requirement: *views* display “the same structure from different perspectives”, while *editing modes* “allow users to work efficiently on all parts of a document”. This leads to the following additional requirement, which renders item one more precisely:

- Provide extension mechanisms to introduce viewers and editors for additional languages.

In the context of structured editors, Walker [Walk81] claims that “adding a new document language to [the] editor's repertoire consists primarily of writing the parser and unparser for that language.” For the authoring tool, the introduction of a new language has to be accompanied by means to provide appropriate viewing and editing functionality to the application.

4.1.2. Consolidated Requirements for the Authoring Tool

The assessment of the authoring tools presented in this chapter is based on a list of requirements. Some requirements have to be classified in order to provide a better overview at the end of this chapter, where the presented authoring tools are evaluated.

I have replaced informal identifiers with classifications where applicable. The assessment *extensible*, used in several of the requirements, refers to possibilities to extend the application in a custom way, and is only used for features where I have personally seen appropriate need. Refer to Table 4.1, “Requirements for the Authoring Tool” for the requirements

Table 4.1. Requirements for the Authoring Tool

Requirement	Assessment	Description
Editing	schema semantics	According to Quint and Vatton [Quin04], *schema*-driven editing allows the manipulation of the document tree according to the imposed grammar. *Semantics*-driven editing takes into account the “specific aspects of the […] application.”
Structuring	inline advanced extensible	*Inline* structuring provides simple means to manipulate the structure of the document, while *advanced* means comprise complex structure transformations similar to formatting as provided in a word processor. *Extensible* refers to user-defined (e.g. macro) structure transformations.
Publishing	basic advanced extensible	*Basic* publishing provides output to at least one format intended for printing, such as PDF, and one flowed format, such as HTML. *Advanced* publishing provides mechanisms to customize the publishing in format-specific ways[i], while *extensible* provides means to introduce additional output formats to the set of formats that the authoring tool can publish to.
Linking	internal external	An editor supporting *internal* linking provides means to add identifiers (anchors) to specific elements and a selectable list of targets, when the author wants to insert a link to such an anchor. Links to external documents are provided by an editor supporting *external* linking, by offering the author means to select other documents and browse anchors inside this document.
Transclusion	reference embedded	Independent from the publishing, a *reference* to an external resource or document can be added to parts of the document and is indicated by a placeholder. In order to provide a better understanding of the final published document, the editor may *embed* the resource or document and display the expected contents comparable to the published document.
Extensible language support	plug-in stylesheet	This classification is on the level of standards support: a *plug-in* is specific to the authoring tool and may be of little practical use in a different environment. A *stylesheet* is written in a standard language and can be used by other applications supporting this standard.
Versioning	comparison integration	In contrast to textual comparison, which does not need any additional knowledge of the document, semantic *comparison* provides differences between two versions of a document (or two different documents) on a content-level. Full *integration* of versioning allows the user to select differ-

Requirement	Assessment	Description
		ent versions[ii] for comparison and apply changes to a document providing additional meta-information.
Collaborative Authoring	explicit parallel	Unrelated to versioning facilities, *explicit* collaboration is provided by the editor, if it stores a reference to the author alongside the edited content. In a *parallel* collaborative authoring scenario, two (or more) authors may simultaneously edit different parts of the same document.

[i]For example, PDF documents could benefit from watermarks or the addition of digital signatures for use in legally binding contracts, while HTML document could be customized via CSS.

[ii]The editor may provide versions from backups stored regularily in the application's workspace, or from a version control system.

4.2. Commercial Authoring Tools

First, this section looks at commercially backed authoring tools. I will not go into details about these applications, except for elements that have to be explained for the comparison at the end of this chapter.

Altova Authentic. Distributed at no charge by Altova[1] the application is a visual XML authoring tool. The *Altova View* is part of the XML editor *XMLSpy* by the same company. The software distribution bundles support for the authoring of several document formats in the form of proprietary stylesheets[2] and minimal document templates.

Microsoft InfoPath. The application was first released as part of the Microsoft Office 2003 suite, and is, at the time of writing, distributed as part of Microsoft Office 2007[3]. Depending on the literature [Bern07], [Jaya08], the software is categorized as forms manager, form generation tool, data translation tool, but it can effectively be used as authoring tool for small documents based on structured XML languages. It provides simple mechanisms for extensibility (XML namespaces, SOAP-based web-services) and user input (XHTML user input, conditional formatting). A major part of the application is the designer, which can be used to create a semantic editor and the documentation format.

Microsoft Word 2007. Natively supports OOXML and, starting with SP2[4] also ODF. In order to edit structured documents, Microsoft Word 2007 provides several facilities. *Building blocks* are reusable parts of the document that can be inserted at different locations. They contain information related to formatting, but may also contain *content controls* which only allow certain types of content or data. Because of the tight integration, the information stored in the content controls may be mapped to SharePoint properties as a repository.

OpenOffice.org Writer. OpenOffice.org (OOo)[5] natively uses ODF, supports many formats via *filters* and runs on a variety of different platforms. It internally uses an XML file format that facilitates transformation into other XML languages[6] and provides an API for extension purposes[7].

[1]Available from the product's website [http://www.altova.com/products/authentic/xml_db_form_editor.html].
[2]Altova Authentic uses StyleVision Power Stylesheet (SPS) files that are created using Altova StyleVision [http://www.altova.com/whitepapers/stylevision.pdf], an XML stylesheet designer.
[3]Refer to the InfoPath Website [http://office.microsoft.com/infopath] for current information.
[4]Refer to the section called “Standardization” (p.52).
[5]Refer to the Wikipedia entry for OOo [http://en.wikipedia.org/wiki/OpenOffice.org] for a brief history of the project.
[6]Filters and templates for DocBook are provided at a dedicated page [http://xml.openoffice.org/xmerge/docbook/] and are part of the OpenOffice.org XML project.
[7]The component model *Universal Network Objects* (UNO) provides bindings for several programming langueages and is the basis for the API. More information can be found at the respective web pages for UNO development kit [http://udk.openoffice.org/] and the API [http://api.openoffice.org/].

4.3. Independent Projects

The need for structured document editing with WYSIWYG features has encouraged individual projects to fill the void. Because these projects are not part of a larger product or product suite efforts, they are subsumed under the notion of independent projects.

4.3.1. General Project Activity

Between the time of research, where these projects were added to the list, and the final review of this document, activity in these project has declined strongly.

For example, version 1.2.1 of the editor *Vex* was released in April 2005, which has stayed the most current release until October 2008, although two additional developers joined the project in he meantime. According to the download statistics[8], Vex was downloaded about 45 times per day in the 10 months following the release. In 2008 the downloads have declined to about 25 per day.

The following question on the Vex developer mailing list was asked in September 2007: “[Why have] so many so alike wysiwyg xml editor project [been] started, were living and are dead or not complete? Why haven't we cooperated?”. In a response to the list[9], John Krasnay tried to explain his view on the reasons for the project (and other similar projects) not to be picked up by users or the community in general:

> I was trying to solve the wrong problem. [… T]he problem was not so much Word as it was Word documents being stored on network shares. […T]he "itch" I had was for document management, not for structured document editing […F]or me, the sweet spot is now Wikis.

4.3.2. Vex — A Visual Editor for XML

Vex is an editor for XML documents and since release 1.0.0 based on the Eclipse platform. “The "visual" part comes from the fact that Vex hides the raw XML tags from the user, providing instead a *wordprocessor-like interface*”[10]. The editor widget can also be used outside of Eclipse, using Swing and SWT.

Document Formats in Vex

The software supports several document types and bundles appropriate configuration items used for editing. A document type can be assigned a stylesheet which is used to render the wordprocessor interface in the editor view. Vex is designed to be extensible: it should be easy for users to create new or extend existing document types. Developers with experience on the Eclipse platform may also contribute functionality

Document Styling in Vex

It is possible to extend Vex to edit different document types by adding a DTD and CSS files to the Vex installation[11]. The process is facilitated by the *Vex Plug-in Project Wizard* which associates DTD with CSS file(s) to be displayed when editing the document.

A Vex installation bundles DTDs their proper style sheets for the Vex editor for several popular document types. In Example 3.4, “Meta-Information on a Book in DocBook Syntax” the XML source

[8] As provided by SourceForge in the Project Download History for vex / 1.2.1 Statistics [http://sourceforge.net/project/stats/detail.php?group_id=67542&ugn=vex&type=prdownload&mode=alltime&package_id=65870&release_id=319036].
[9] From the archives of the Vex developer mailing list: Re: [Vex-developer] The hard way, the big goal - Wysiwyg XML editor [http://sourceforge.net/mailarchive/message.php?msg_name=20070915210756.GB21790%40effectivecommerce.com].
[10] Quoted from the Vex project website [http://vex.sourceforge.net/].
[11] From the Vex cookbook [http://integerservices.no-ip.info/VexCookbook.pdf].

for a bookinfo element in DocBook markup (see Section 3.3.1, “DocBook”) is displayed. In version 1.2.1 of Vex, the style information to display this snippet in the editor is named `docbook-plain.css` and comprises 1143 lines. In Example 4.1, “CSS Snippet To Render Book Meta-Information in Vex” only the styles important for the snippet are shown, ordered by appearance.

Example 4.1. CSS Snippet To Render Book Meta-Information in Vex

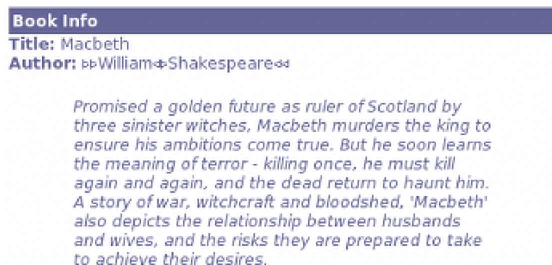
```
 1 bookinfo {
 2   border: 1px solid #669;
 3   color: #669;
 4   display: block;
 5   font: 10pt Verdana, sans-serif;
 6   margin-bottom: .25in;
 7   padding: 3px;
 8 }
 9 bookinfo:before {
10   background-color: #669;
11   color: white;
12   content: 'Book Info';
13   font-weight: bold;
14   display: block;
15   padding: 3px;
16 }
17 bookinfo > title:before {                                      ❶
18   content: "Title: ";
19   font-weight: bold;
20 }
21
22 author {
23   display: block;
24 }
25 author:before {                                                ❷
26   content: 'Author: ';
27   font-weight: bold;
28 }
29
30 firstname {
31   display: inline;
32 }
33 surname {
34   display: inline;
35 }
36
37 abstract {
38   font-style: italic;
39   margin-left: 0.5in;
40   margin-right: 0.5in;
41   display: block;
42   padding: 6px;
43 }
44
45 para {
46   display: block;
47   margin-bottom: 1em;
48   margin-top: 1em;
49 }
50
```

❶ For the title element in bookinfo, the style sheet declares an additional content before showing the content of title. This is only to show which information is to be displayed.

❷ The same approach is taken for author.

The style sheet is applied to the content from Example 3.4, “Meta-Information on a Book in DocBook Syntax” and rendered in realtime in the editor. The resulting editable component is shown in Figure 4.1, “Vex displaying DocBook Meta-Information.”.

Figure 4.1. Vex displaying DocBook Meta-Information.



The `bookinfo` element as it is displayed in the Vex editor

4.3.3. Etna XML Editor

Etna[12] is developed by Laurent Jouanneau and uses RelaxNG schemas to edit and validate Documents. At the time of writing, the most current Etna release (version 0.3.1) is based on Firefox 1.0.7.

Document Formats in Etna

Etna provides support for documents defined by RELAX NG grammars. Support for additional formats is available via *extensions*[13]. As the editor is based on Firefox, these extensions are packaged, distributed, and installed via the Mozilla-based *Cross-Platform Install* (XPI) technology.

The schema is used to generate the semantic editing capabilities and update the UI elements of the editor accordingly. The menu items used to insert siblings or children are generated from the editing context and the schema.

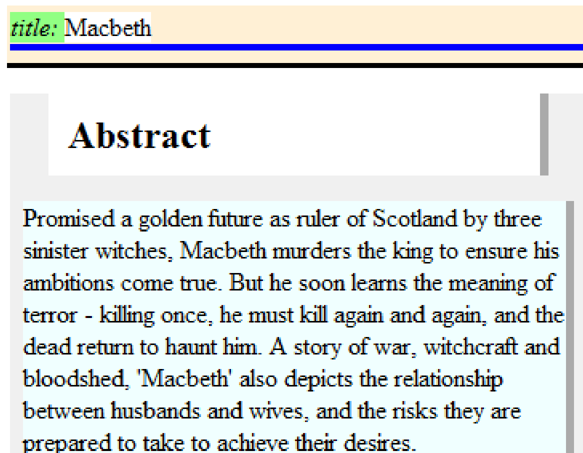
Document Styling in Etna

In order for Etna to enable graphical XML editing while still being based on RELAX NG, it was necessary to add some elements to the grammar[14]. The RNG schema is interspersed with extensions from a different namespace and refers to a CSS file for styling. The style sheet defines how the different parts of the document should be displayed, and, at a different level, how UI elements specific to the currently edited schema are presented.

Refer to Figure 4.2, “Etna displaying Information from a Tinydoc File” for an example how Etna by default renders a short document with one paragraph preceded by a heading. The document title and author are stored in the document's meta-information, but only the author is displayed.

[12] Etna project website [http://rhaptos.org/downloads/editing/etna]
[13] Additional extensions are available from the Etna Extensions page [http://rhaptos.org/downloads/editing/etna/extensions/].
[14] Detailed information is available at Etna's RelaxNG pattern extensions [http://rhaptos.org/downloads/editing/etna/rng_extensions].

Figure 4.2. Etna displaying Information from a Tinydoc File



Information on Macbeth as it is displayed in the Etna editor

4.4. Summary

For a comparison of the previously developed requirements for the authoring tool, refer to Table 4.2, “Comparison of Supported Requirements in Authoring Tools”. Because all of the authoring tools provide semantics-driven editing, this column is removed in the comparison.

Table 4.2. Comparison of Supported Requirements in Authoring Tools

Authoring Tool	Structure	Publish	Link	Transclude	Language Support	Versioning	Collabora-tion
Authentic	inline	extensible	—	—	plug-in	—	—
InfoPath	inline	extensible	—	—	plug-in	comparison	—
Word	advanced	advanced	internal	reference	plug-in	comparison	explicit
OOo	advanced	extensible	internal	reference	plug-in	comparison	explicit
Vex	—	—	—	—	stylesheet	—	—
Etna	—	—	—	—	stylesheet	—	—

As can be seen from the table, the modern applications primarily categorized as word processors already provide a large set of the requirements developed for the authoring tool.

Part III. Modular Integration Concept

Together with the previously defined three main goals and their sub-characteristics as presented in Chapter 2, *Goals and Evaluation of Requirements Authoring*, it is possible to draw a picture of what is needed for an authoring framework for requirements documentation. The main goals and deficiencies of current documentation approaches are already identified. Chapter 3, *Analysis of Documentation Formats* and Chapter 4, *Analysis of XML-Based Authoring Tools* presented a combination of technological approaches to remedy these problems.

The following sections will try to conclude the findings of this thesis with a working proof of concept. The technical characteristics for a documentation authoring framework are defined and translated into requirements. Based on the resulting concept, I will go into details on how the different modules could be integrated with one another. This part concludes in a chapter where the findings of this thesis are validated

- Chapter 5, *Concept of an Authoring Framework*
- Chapter 6, *Developing the Authoring Framework*
- Chapter 7, *Validation and Conclusion*

Chapter 5. Concept of an Authoring Framework

Based on what was observed in the previous chapters, the findings are now used to sketch out an authoring framework for requirements documentation. This framework should provide support to handle changes in the specifications. Most of the framework's characteristics are also applicable outside of this context for very general document authoring frameworks, but the focus lies in requirements documentation that tries to cover evolving product specifications.

I will first recapitulate on the goals for document authoring and draft specific purposes of the framework, so that the components of the framework can then be analyzed with these objectives in mind.

5.1. Goals of the Authoring Framework

Sets of objectives for document authoring, the documentation format, and the authoring tools have already been collected, and will be summarized in this section as objectives of the authoring framework.

In Chapter 2, *Goals and Evaluation of Requirements Authoring* I have declared three main goals and specific characteristics for document authoring. Objectives for the documentation format are taken from Chapter 3, *Analysis of Documentation Formats*, while requirements for the main environment in which document authoring takes place are elaborated in more detail in Chapter 4, *Analysis of XML-Based Authoring Tools*.

Document Authoring

1. Gain understanding: quality of perception, learnability
2. Share information: availability, distribution, retrieval
3. Track information: versioning, transparent changes

Documentation Format

4. Standards-based
5. Extensible: pluggable architecture, stylesheets
6. Versioning support
7. Inclusion in automated processes

Authoring Tool

8. Flexible document editing capabilities
9. Platform independent
10. Small memory foot print
11. Fast

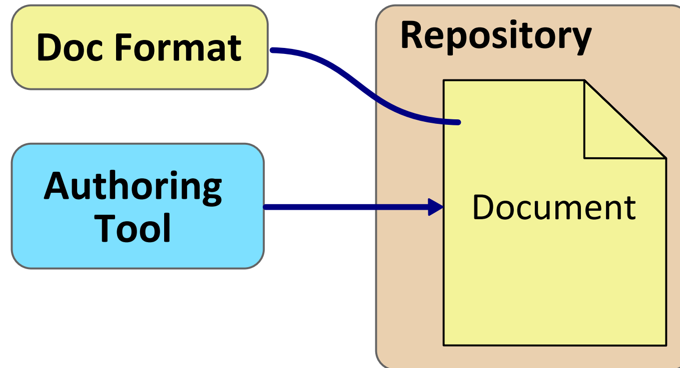
5.2. Component Overview

The framework consists of three components, each providing a proper set of functionality:

- Documentation Format
- Authoring Tool
- Repository

I have already discussed existing implementations of the first two components, but until now the *Repository* has not been mentioned, because it only makes sense when viewing all components collectively. Refer to Figure 5.1, “Components of the Authoring Framework” for a graphical representation.

Figure 5.1. Components of the Authoring Framework



The *authoring tool* accesses documents conforming to a *documentation format* via the *repository*.

Documentation Format. This is the base component of the framework responsible for storing the requirements. It defines how the information is structured (*structure*) inside the document instance and implicitly constrains the presentation of the information (*style*).

Authoring Tool. This set of modules denotes the interface for human interaction. The authoring tool is used to edit (*editor*) the document instance and compare (*comparison*) different versions of a document.

Repository. This abstract component has the purpose of storing multiple versions (*storage*) of a document and make these uniquely accessible. Links between components and from systems external to the framework are solved by this component (*linking*).

5.3. Component Architecture

The three main components of the authoring framework are primarily based on the analysis in Part II, “Documentation Formats and Authoring Tools”. In this section, the components are observed from a functional level to construct the framework's architecture.

5.3.1. Base Documentation Format

At the lowest level of the framework's document model, the documentation format defines how information is stored within the document, the *structure*. The presentation or *style* of the information depends on the document model and, in the concept of our architecture, is therefore also located at the documentation format level (refer to the discussion in Section 5.4.2, “Versioning”).

The documentation format is based on one of the formats presented in Section 3.3, “Current Document Formats and Standards” (refer to goal 4). The format should provide support related to changes in different versions of the document (goal 3), although meta-information regarding these changes will be stored in the repository.

Content

Primarily, the documentation format should provide means to store the documented requirements. For the authoring framework, the types of documentation as presented in Section 1.5, “Documentation Types” are relevant. In order to know what to store for specific documentation types, it is

necessary to know what will be done with this information, either by a human user (author or reader), or by automated processes (scripts, other applications ...).

Requirements for the authoring tool are restricted to a certain combination of *key product attributes* (KPA) and maturity levels (refer to Section 2.2.2, “Documentation Maturity Model”) as presented in Table 5.1, “Documentation Maturity for the Authoring Tool”.

Table 5.1. Documentation Maturity for the Authoring Tool

KPA	Maturity Level	Description
Text	Hyperlinked (3)	It should be possible to set anchors at arbitrary positions inside the text. It should be possible to reference these anchors from any part of the text.
Graphics	Static & Standardized (2)	Graphics in a standardized format and representation are directly embedded into the documentation. Alternatively they are linked similar to hyperlinks in textual formats.
Efficiency	Semi-automatic & Static (2)	Parts of the documentation representing the actual implementation should be built automatically and be based on the implementation. These parts should be built using tool support at the time of documentation generation.
Granularity	Requirements (4)	Although evidently, but part of the maturity model, the documentation should capture the system's intended purpose.

That allows to summarize what information should be stored in the documentation format for specific documentation types. The documentation types are explained in Table 1.1, “Documentation Types for the Authoring Tool”. In Table 5.2, “Stored Information per Documentation Type” the type of documentation is mapped to the actually stored information for this type. In addition, the last column shows in what kind of format this information could be represented.

Table 5.2. Stored Information per Documentation Type

Documentation Type	Stored Information	Example Manifestation
Statements of the required effects	A well-structured sequence of input to and output of the designed system, description of process flows.	Behavior diagrams or descriptions
Recorded issues and changes	Issues and changes are listed and may be associated with either features, lower-level requirements, or revisions of the document or application. The listing may include the full context of the issue/change, but a summary including a link to the item in the issue or change management system is preferred.	Structured change log, QA reports, reports from the issue or change management system
Defined goals and objectives	General description of the intention of the application and detailed description of the objectives of certain features.	Textual description
Specified constraints	Description of constraints for certain features or business rules that cannot be described in a structured way.	Textual description
Recorded agreement	References to the origins of the requirements which can either be existing documentation, meeting minutes, or other types of written communication. The references may be attached to single feature descriptions or business rules.	Links to other documents, communication archives (meeting minutes, mail archive), and search facilities
Top-level overview of the system	Description of the system's parts and their relationship.	Deployment description
Interface documentation	Summarized for architect and developer roles, this type contains information on the employed technologies and interface-level requirements.	Sequence diagram, Component diagram

Structure

Now, after observing what is stored in the document model, it is possible to fdefine how this information will be stored. Summarizing the representation formats from the previous section, the following list can be compiled:

Types of Information Stored in the Requirements Specification

- Text in prose
- Structured and semi-structured text
- Images and diagrams
- Links to other information sources
- Aggregation of information regarding the system
- Aggregation of external information

This list does not claim to be comprehensive, but for the design of the authoring framework it should be sufficient. This section will have a look at how each of these different information formats can be integrated into the document structure.

The *target document structure* is based on one of the previously presented document formats (refer to Section 3.3, “Current Document Formats and Standards”). That means eventually the information will be stored in one of these formats. Before that a domain-specific language (DSL) is employed, in order to store the information in a clear way. This follows the motivations described by Heitmeyer et. al [Heit97] and Shani and Sela [Shan08] (refer to the discussion in Section 5.4.1, “Domain-Specific Language”).

In addition, each semantic element (e.g. paragraph, section, table, image …) can be assigned an id for further reference. The id is assigned via the attribute `xml:id` [XML-ID] as already used by XML-based formats.

The design of the document structure is restricted in two ways:

- Characteristics as described in Section 3.1, “Restrictions”.
- Transformation into one of the previously presented document formats should be possible.

The following sections describe how each type of information representation from Table 5.2, “Stored Information per Documentation Type” can be stored in the target document structure and indicate whether support for specific information is already present in the standard document formats.

Text in Prose and Structured Text

Normal, textual content can be handled without additional design efforts. Semantic formatting (emphasis, indentation, super- or subscript …) on character or paragraph level is also supported by the documentation formats.

For the documentation purposes in the authoring tool, structured text is referred to as textual representation of information with the following characteristics:

- The text is partitioned into one or more logical elements.
- Logical elements may be assigned an id for further reference from other elements.
- Structural elements may contain other structural elements, and form a hierarchy (part, chapter, section …).
- Block-level elements contain the actual information (paragraph, list …).

- Inline elements wrap text within a block-level element and can provide special treatment to the text (emphasis, hyperlinks, …).

With these characteristics the text does not have to contain presentation information, but instead inline elements may indicate how the wrapped text should be formatted.

Structured text adhering to these criteria is supported by ODF and OOXML, and enforced by DocBook. Through their extension mechanism, all formats allow for the introduction of custom structures that model special types of requirements.

Images and Diagrams

Only two types of visual information representation are considered, depending on how they have been generated. Both do not contain any animations or allow for interactivity:

Data-Independent
: Is not directly generated from data and needs manual interaction for editing. Examples: logo, photo, screenshot, high-level overview graphic …

Derived
: Is directly based on data and may be updated by changing the underlying parameters. Examples: graph visualizing statistics, UML diagram generated in an editor and updated via round-trip engineering, deployment diagram generated from deployment scripts, …

Another characteristic is how the visual representation is integrated into the document. *Embedded* images are stored inside the document using either special encoding for binary data[1], or direct integration for text/XML-based image formats[2]. *Linked* visual representation inside the document structure is composed of a non-ambigous link to the target image, and required meta-information (e.g. clipping, anchor for further reference inside the document). XML provides several mechanisms to implement this behavior, which are described in the section called “Standardized Inclusion Methods” (p.42)

One special situation has to be noted: linked images may change without affecting the document version. This may generate confusion and should be avoided. One possible solution to this potential problem are compound documents (refer to the paragraph called “Compound Document” (p.55).

Links to External Resources

On a document structure level, links to external resources are very similar to linked images. For each linked external resource, the document itself contains the link to the resource and meta-information required for the correct handling.

Aggregated Information

Especially for the case where requirements for existing systems have to be extended, it is useful to integrate current system information. For example: the implementation is the best source of information to build a deployment diagram, or representation of the amount of different configuration options. The problem is that this information is usually not directly available, at least not in the anticipated form. In order to aggregate this information and provide it in a way suitable for a requirements specification, certain preparations have to be made on the implementation and the aggregation side.

I will remain with the example of a deployment diagram. The information required for the diagram is implicitly contained in the dependency graph of the top-level components. In an environment

[1]Different algorithms have been developed to efficiently encode binary data in XML. Refer to [FI] or [EXI] for more information. More current information on this subject is available at the Wikipedia page on Binary XML [http://en.wikipedia.org/wiki/Binary_XML].
[2]Examples: Computer Graphics Metafile (CGM, ISO 8632) or Scalable Vector Graphics [SVG].

using a declarative dependency mechanism[3], this specific information has to be made easily available (by following naming conventions for certain components, or putting all relevant components into a single logical container), and can then be aggregated into an appropriate list or table inside the document structure.

Similar steps are necessary for information aggregated from external systems. If these systems provide their information in a computer-readable, standards-based syndication format (Atom, RSS ...) and results can be parameterized, the aggregation component only has to declare what information is relevant for that specific context.

Style

The last component of the documentation format covers the presentation or styling of the content. The information inside the structure of the documentation format is transformed into a target *presentation format*. This process involves processing the different types of content and resolving information external to the specification.

Presentation Format

For this solution, only two types of static *presentation format*s are considered with the following application:

HTML
: Primarily intended for on-screen reading and as output for web applications.

PDF
: Delivery format for final releases of specifications. Used for reviews and document-based communication.

Both formats provide a sufficient set of mechanisms to integrate pieces of information presented in Listing "Types of Information Stored in the Requirements Specification", p.70.

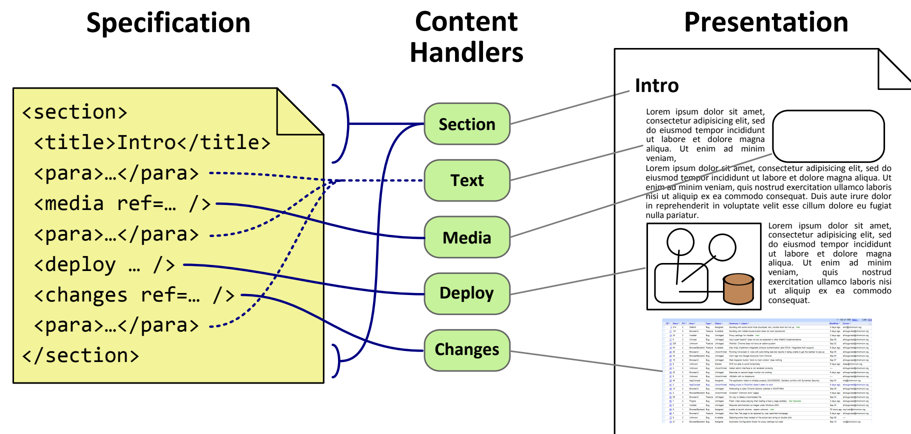
Content Handler

For several elements presented in the section on the document format's structure only a reference is stored. The styling component has to resolve the reference and appropriately display the retrieved information at the specified position in the document. As there are different types of elements that can be referenced, different means are necessary.

These *content handler*s are responsible for the correct representation of the referenced information. In case of simple linked images, most of the information can be delegated to the presentation format. Example: in case of HTML, an appropriate `IMG` element is used. Depending on the type of referenced or embedded resources, the presentation format supports integration, and may always fall back to linking to the resource and delegating the handling to the user's environment via standard URI schemes and protocols.

[3]Several platforms and programming languages provide frameworks with dependency injection (refer to a list of existing frameworks on Wikipedia [http://en.wikipedia.org/wiki/Dependency_Injection#Existing_frameworks]). Similar information can be extracted from the bundles deployed inside a container following the OSGI framework specification [http://www.osgi.org/Specifications].

Figure 5.2. Content Handlers



Different types of information are processed by different content handlers. References with additional meta-information are resolved and rendered in the presentation format[4].

In the example used in Figure 5.2, “Content Handlers” the content handlers *Deploy* and *Changes* collect their information from systems external to the specification (dependency graph in the source code, or change request system respectively), and aggregate this information for the presentation format.

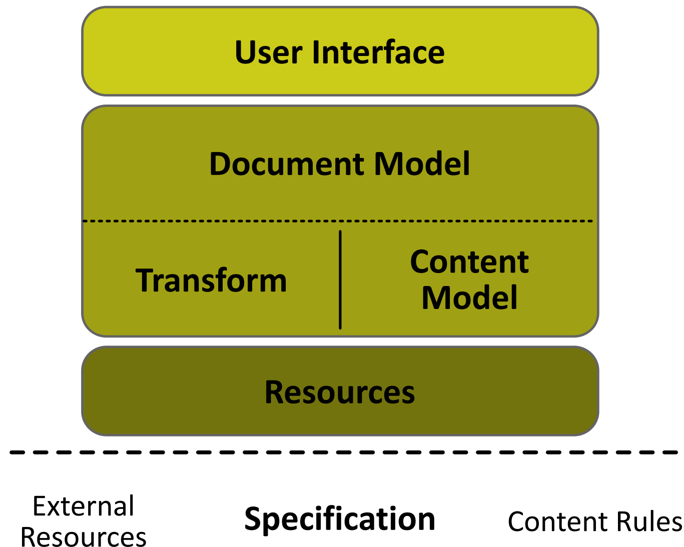
A clean architectural approach for the style component assigns a default content handler for the document and specific handlers for more complex information aggregation. Types of information that only require trivial transformation into the presentation format are processed by the default handler.

5.3.2. Authoring Tool

This tool is the main interface between the document author and the specification. It serves two needs: editing of the specification and comparison of different versions of a document. The components are depicted in Figure 5.3, “Components of the Authoring Tool”.

[4]The representation of changes is actually a screenshot from the bug tracker of the Chromium project [http://code.google.com/p/chromium/issues/list].

Figure 5.3. Components of the Authoring Tool



The UI is the entry point for the author. The document model is an abstraction of the specification which is accessed through the resource layer.

Details on the different components are presented in the detailed design in the next chapter, but for now the main functions have to be explained.

From an author's perspective, the user interface component is the entry point to the authoring tool: the *editor*. The first layer below the UI is abstracted by the *document model* which itself is divided into the *transformation* component and the *content model*. All access to resources — which includes access to external images, rules for the documentation format, access to the repository and the specification itself — is abstracted by the *resources* layer.

Editor

The editing component is the interface used to manipulate the specification's contents through the document model. It uses the presentation format to render an up-to-date preview of the document that corresponds to the final presentation. In the component diagram this responsibility is assigned to the transformation module.

As in the scenario described by Meyer [Meye02], the editor should directly create documents in the base documentation format. The author is expected to have a moderate knowledge in standard word processing and does not need to have knowledge of the underlying format.

The preview does not have to exactly render the document according to the styling, and thus may use less complex mechanisms[5] as long as the following requirements are met:

- Rendering of the preview should not require large amounts of resources (goal 10).
- Presentation of the document should update fast (goal 11)

In order to improve performance and usability, incremental updates to the presentation layer as proposed by Onizuka et. al [Oniz05] could be employed. Especially for large specifications with simplified transformation rules this approach could prove beneficial.

Through the content model, the editor stores the information in the documentation format. The content model provides the editor with the type of information available at the current editing position. This may be very different depending on the editing context.

[5]This paradigm is also called W*hat You See* Is W*hat You* M*ean*(WYSIWYM) and is used in applications such as Altova Authentic (the paragraph called “Altova Authentic” (p.59)) or L_YX, a document processor for T_EX (Section 3.2.2, “TeX”).

Example *Block-Level Element*. Inside the block-level element (paragraph, list ...) the author may add content. As soon as the author hits enter, the block-level element is closed and a new element is opened.

Example *Selected Text*. Selecting a part of the text inside a block element allows for changing the character formatting or adding references.

Example *Section*. The section's position and level inside the document hierarchy can be adapted. As soon as the author hits enter, a new paragraph is started and the current editor position switches to this element.

Comparison

Tracking information (goal 3) is an important aspect in document authoring. Its support in the documentation format is largely provided by versioning capabilities (goal 6). These facilities have to be provided by the editor in the form of comparison.

Primarily the framework allows for the comparison of different versions of the same document. Conradi and Westfechtel [Conr98] define a version intended to supersede its predecessor a *revision*, while versions intended to coexist are called *variants*. Information on the relations between versions (as retrieved from the version graph) have to be made available to the user. It should be possible to compare the document with previous versions of a revision or variants of the current specification.

Differences

According to Rönnau et. al [Rönn05], a structural diff algorithm is a prerequisite to version control for XML documents. They define separate requirements on the level of version control and office documents, concentrating on delta inference. The following list shows a subset applicable for the problem domain of the authoring tool:

Version Control

- Equality with respect to the ordered tree model of XML
- Delta provides exact location of the change
- Delta should be invertible to provide forward and backward reconstruction
- Delta computation time and size must scale to large document sizes and large amounts of changes

Document-level

- Support wide and flat trees
- Recognize move operations of nodes, also to deeper or upper hierarchy levels

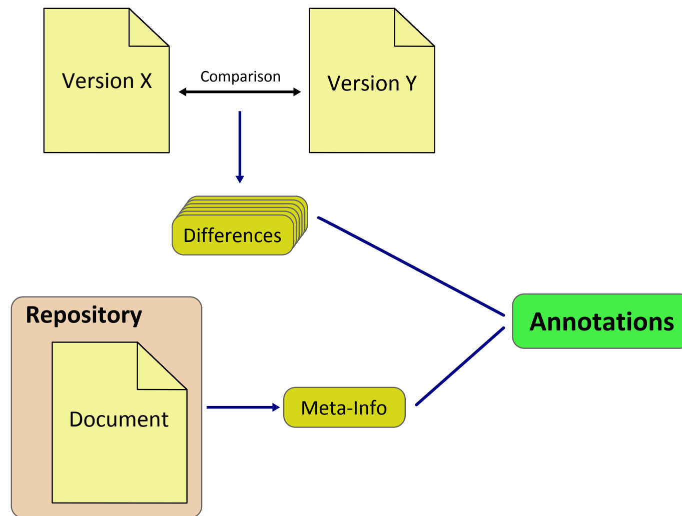
The deltas generated from the comparison can be used to improve the user's understanding of the document. Changes to different versions can be seen immediately.

Annotations

As previously discussed, information from the contents of the specification is presented using the transformation component. On top of this presentation, the editor exposes additional data generated from the differences. These pieces of information are stored in *annotation*s which are associated with specific nodes in the document.

For the editor component, annotations can be generated from the differences calculated during comparison, or from the document's meta-information. In Figure 5.4, “Annotations from Comparison or Meta-Information” the meta-information is provided by the repository.

Figure 5.4. Annotations from Comparison or Meta-Information

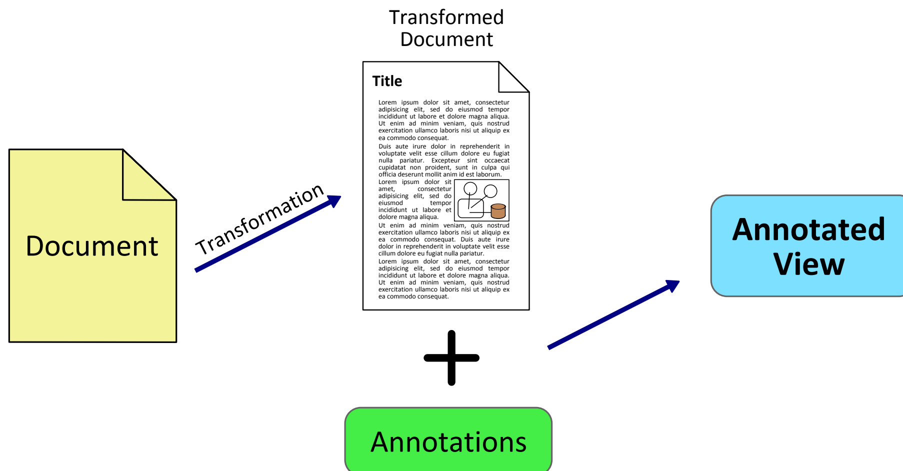


Annotations can be generated from the comparison of two different versions of a document or retrieved from the meta-information of a document instance.

Annotated View

The information provided by the annotations is displayed in an *annotated view*. The annotated view's presentation is based on the standard transformation algorithms and enriched with annotation information, as depicted in Figure 5.5, “Annotated View”

Figure 5.5. Annotated View



Standard document presentation is enriched with annotations to result in the *Annotated View*.

5.3.3. Repository

The only component of the authoring framework that has not been yet covered in this thesis is the repository. It provides a central mean for the *storage* of documents, and a unique identification of document versions used for *linking*.

Implementation of simple repositories are already provided by current versioning systems[6]. More sophisticated repository implementations could provide indexing and search mechanisms to improve user experience and reduce the time to find requested information[7]. These implementations are not covered.

Storage

The specification itself is stored at a central location that is accessible by authorized users. The repository provides access to different versions of a document and stores the following meta-information for each change:

Author
: The author of this document, i.e. the person who has stored this version of the document.

Date
: Date and time of the change, i.e. when this version has been stored.

Change Comment
: The authors comment related to the change, indicating a summary of the changes.

Parent Version
: A reference to the previous version of the document.

Through the *parent version* meta-information changes to a document can be tracked (refer to goal 3) and associated with a date and author. This helps to establish accountability and traceability of the requirements. According to Conwell et. al [Conw00] traceablity “helps ensure that the user's needs are being met in the implementation and provides the accrediting authority confidence throughout the development”. As mentioned in Section 2.1.1, “Gain Understanding”, this is essential for understanding (goal 1).

Linking

All documents can be retrieved from the repository via a unique URI. Different versions of documents can also be accessed through their own URIs. This approach makes all information uniformly available (goal 2), and also allows for relative inter-document links to improve understanding (goal 1).

Versions of a document that are of special interest (e.g. most recent version, passed review ...) may be selected for further reference with a global identifier. If a different version attains the special status, the global identifier may be assigned to this version. In revision control this identification is called tag or label.

5.4. Design Discussion

Some of the objectives of the framework can be solved in more than one component, but to keep the architecture as clean as possible, responsibilities are not spread among multiple components. These yet unclear responsibilities are discussed in this section.

5.4.1. Domain-Specific Language

Instead of directly using general purpose documentation formats as presented in Chapter 3, *Analysis of Documentation Formats*, the base documentation format employs a limited language that can be translated into the former formats. The literature uses the term *general-purpose language* (GPL) for languages of the former category and *domain-specific language* (DSL) for members of the latter.

[6]Examples: Subversion [http://subversion.tigris.org/], Git [http://git.or.cz/], Mercurial [http://www.selenic.com/mercurial/]
[7]Examples for commercial applications: Microsoft Office SharePoint Server [http://www.microsoft.com/sharepoint/], Atlassian FishEye [http://www.atlassian.com/software/fisheye/]

In this section, the elements of a DSL are discussed, and how the DSL can be used in the context of a requirements authoring tool.

Definition of a DSL

According to Martin Fowler [Fowl08] (his previous work is also referenced in [Shan08]), a DSL is a "computer programming language of limited expressiveness focused on a particular domain" with the following elements:

- Used by humans to instruct a computer to do something, as well as helping communication between humans
- Its expressiveness does not just come from individual expression but also the way they can be composed together
- Supports a bare minimum of features needed to support its domain.
- Has a clear focus on a limited domain.

Fowler distinguishes *internal* and *external* DSLs: internal DSLs use the same GPL that the wider application uses, but in a particular and limited style. External DSLs use a different language to the main language of the application that uses them.

DSLs in the Authoring Framework

In the context of the framework, the base documentation format uses an external DSL to store its model. According to Chidlovskii [Chid03] "many schemas [i.e. the grammar for documentation formats] are too general", they can be used for validation, but "allow much more degree of generality than the real documents do expose." Generic documents imply more complex requirements to the framework building on the document structure. A DSL greatly reduces the complexity compared to using a GPL for the model information and results in the following benefits:

Simple styling
: Transformation into a GPL or directly into a presentation format is limited to the domain and the features described in the DSL.

Selective expression support
: Not all elements of the DSL have to be supported in the transformation to the presentation format in order to provide a human viewer with an understandable specification. For certain fields of application (refer to the section called "Editor" (p.74)) a limited transformation is even desirable.

Trivial Comparison
: When comparing two documents, the differences are on a model level and can be easily retrieved. This simplifies diff algorithms to that extent, that a human editor could even find differences on a content/document-level by employing a textual diff.

Annotation
: With a DSL, associations in annotations (reference to node + annotation data) receive additional semantic information. The reference itself provides information as to what hierarchy or type of information is annotated.

5.4.2. Versioning

This responsibility is covered in goals 3 and 6 and primarily consists of storing different versions of a document (see the section called "Versioning" (p.26)). Separate versions are referred to as separate *instances* of the same document. For this discussion, differences between document instances are only on a content level. Changes to the styling may be versioned separately depending on the styling mechanism employed and are not covered.

$\Delta D_{n,m}$ is defined as the difference between instance *n* and *m* of document *D*: $\Delta D_{n,m} = D_m - D_n$

This *versioning information* contains not only content changes, but at least the following information in regard to the change:

- Date and time
- Author responsible for the change
- Comment summarizing the change and/or the reason for the change

It is possible to store this information inside the document (refer to the paragraph called "Inline Versioning" (p.56)) or external to the document; each alternative providing its own set of advantages.

In simple scenarios, only a single document instance is considered the current version of the document. In this case, the document holds all ΔD and provides means to extract information related to certain changes. All information is stored in a single compound object and may be accessed easily and independently from other resources.

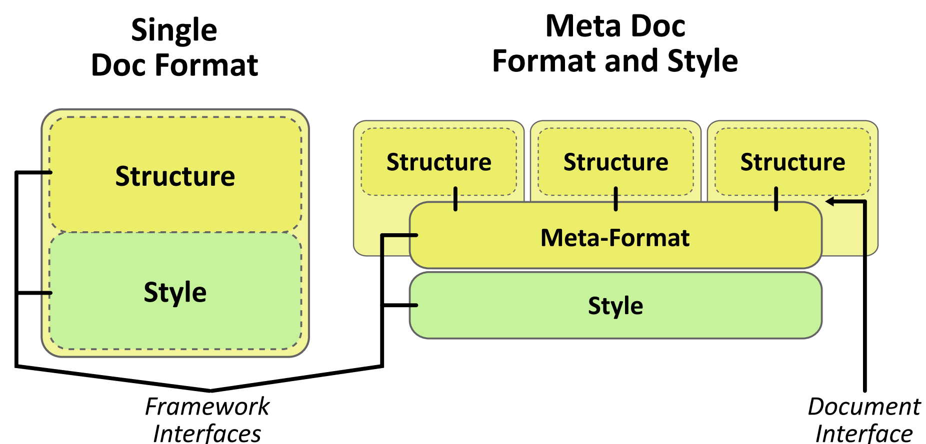
Depending on the *version model*, it is possible that multiple instances exist at the same time and are being concurrently developed. In a very simple case, a document instance is the basis for two teams that need to incorporate two different requirements and merge the results back into a single document upon completion. Especially for long durations of parallel development or manipulation of common regions inside the document this approach is difficult to implement with the versioning information stored inside the document.

5.4.3. Meta-Format

In Section 5.3.1, "Base Documentation Format" the documentation format is presented as the format's main document model. It is standards-based and its presentation is very dependent on the underlying information structure. In a more generic approach, a single format could combine features of different document formats.

The main advantage of using a single format as a wrapper around different document formats, is the reduction of the implementation efforts for the framework's presentation component for additional formats. The styling is only specific to this single combined meta-format, and does not have to be adapted as different document models are used or the framework supports additional formats.

Figure 5.6. Meta-Format as Wrapper around Document Structure



From a framework perspective, a wrapping *meta-format* reduces the number of interfaces for different document formats. Interfaces for every wrapped document structure have to be developed.

Chapter 6. Developing the Authoring Framework

The concept and the requirements for the documentation authoring framework are covered in the previous chapter. Now, the requirements are converted into a technical design for the *Use Case Creator*. This section starts by focussing the design decisions and the field of application. Based on the architectural overview, this section subsequently goes into details on how the modules are designed for this specific implementation.

The results of the implementation have been made available at http://usecases.googlecode.com/.

6.1. Focus

I have decided to reduce the implementation to a reasonable size, in order to be able to maintain a full understanding of the solution, while remaining in the scope of this thesis. To achieve this, I have focused the previous work at two dimensions: the framework design, and the field of application.

6.1.1. Framework Design

A small change to the previously presented architecture is the separation the of authoring tool into a *document viewer*, and the *editor* itself, providing the two different types of functionality. I will also provide a more detailed view on the integration aspects of the *repository* component of the application.

6.1.2. Field of Application

As seen previously, the information covered in requirements documentation is very diverse (refer to Table 5.2, “Stored Information per Documentation Type”), and document authors employ different techniques to create the documentation.

In this thesis, a single, simple, structured document format is used, and the remaining framework is designed with this format in mind (refer to the discussion in Section 5.4.1, “Domain-Specific Language”). This format does not cover all types of requirements documentation or authoring processes. Instead, it will only deal with a single type of requirements document, the *use case analysis description*. That means, initially two restrictions are accepted, the second being a consequence of the first:

- Simple document format
- Restriction to a certain set of information

Being part of the design, the document format is explained in detail in Section 6.2, “Documentation Format”. For now, the focus lies on the type of information, the use case analysis description.

Use case analysis is a technique used to identify the requirements of a system. Use cases have an important role in the requirements process and are used very commonly. In an article on the history and future of use cases, Jacobson [Jaco03] states that “use cases are basically a simple and obvious idea” and that they “work well with objects and object thinking.” They can be used to manage requirements and bind together all the activities within a project.

According to Ashley Williams [Will04], the “heart of the requirements of the use cases are provided [in the] scenario description”, which is referred to as the *use case analysis description*, or *use case description* in short.

The implementation scope of this chapter only focuses on use case descriptions based on a very simple document format for the framework's application design. I expect that a transformation of the findings reported herein to other types of requirements documentation or even structured documents in very different applications should not be too difficult.

6.2. Documentation Format

The documentation format covers most of the information necessary for *use case description*s. A single document can contain one or more use cases that may be grouped and structured using a logical hierarchy. In addition, means to store *change annotation*s are provided directly in the documentation format. Effectively changes are stored in the repository (refer to the discussion in Section 5.4.2, “Versioning”), but for intermediate steps in the presentation it may be necessary to persist such calculated information inside the document.

6.2.1. Content Description

Before going into details on how use case descriptions and change annotations are stored in the document, the basic rules for the content have to be defined.

Validation

In addition to the textual description of the content, a grammar in XSD is provided. The full schema is available at the project website, and I will present the relevant sections throughout this chapter where applicable. Documents will be validated according to this XSD.

Coverage

The employed DSL only covers a very narrow field. In this initial version, the primary intention is to solve a small problem from end to end, rather than trying to detect all edge cases. Due to the architecture it should be possible to extend the language, or replace it either with a standard that better fits the solution domain, or a standard already employed in the target toolchain or corporate environment.

Namespaces

All content related to use case descriptions and their changes is stored in the same namespace: `urn:use-case:description`, further referred to as *home namespace*. This namespace is a valid URN [RFC2141], and as such designed with the goals of namespaces in mind [XMLNS]. Different namespace specific strings (i.e. the sequence of characters after the last colon character) may be employed to avoid collisions, in case more detailed pieces of information are used.

The document format allows for extensibility, primarily basic formatting, by allowing other namespaces to be used inside the content description. In this implementation, XHTML is deemed a reasonable choice for basic formatting and thus used inside textual elements[1]. Refer to Example 6.1, “Complex Type for Text”.

[1]This is based on the approach taken by Atom [RFC4287]

Example 6.1. Complex Type for Text

```
1 <complexType name="text" abstract="true" mixed="true">
2  <sequence>
3   <any namespace="http://www.w3.org/1999/xhtml"
4    minOccurs="0" maxOccurs="unbounded"
5    processContents="lax"/>
6   …
7  </sequence>
8 </complexType>
```

I have decided to use `lax` processing for elements outside the scope of the home namespace. That means that these nodes will only be validated, in case an appropriate schema is available.

6.2.2. Use Case Description

The description of the use cases denotes the model of the documentation format. The information for the use case description is divided into the following areas:

Single Use Case
: At this level of abstraction, the information related to a single use case is discussed.

Logical Hierarchy
: If there is more than one use case description inside a document, the use cases may be put into a logical hierarchy to improve the understanding of relation and order

Inclusion and Extension
: According to Zuser [Zuse04], in use case diagrams two types of associations between use cases can be distinguished: include and extend.

Single Use Case

The smallest unit in the semantic structure is composed of the following elements with their XML element name in parenthesis: The title (`title`) of the use case is used for human identification and reference. A short description (`description`) provides a summary. The actor or actors (`actor`) execute the use case. Pre- (`pre`) and postcondition (`post`) are the conditions prior to and expected condition after the execution of the use case respectively. The flow (`flow`) denotes the course of events or *scenario description*.

Formatting

Title and actor are very simple, and only allow basic text as content type. Description, pre- and postcondition have `mixed` content and the included information may be formatted as shown in Example 6.2, “Complex Type for Precondition” where the complex type for `pre` is defined.

Example 6.2. Complex Type for Precondition

```
1 <element name="pre">
2  <complexType>
3   <xs:complexContent>
4    <xs:extension base="uc:text"/>
5   </xs:complexContent>
6  </complexType>
7 </element>
```

In order to employ formatting in the final document, the `xhtml` namespace has to be used. For content that uses more elements of a foreign namespace and fewer elements of the home namespace, it is possible to declare the default namespace for the parent element. This is shown in Example 6.3, “Formatted Postcondition” for the postcondition.

Example 6.3. Formatted Postcondition

```
1 <uc:post xmlns="http://www.w3.org/1999/xhtml">
2  The document is displayed together with the following meta-information:
3  <ul>
4   <li>Revision of the document</li>
5   <li>Older revisions of this document</li>
6  </ul>
7 </uc:post>
```

Scenario Description

In order to model, and subsequently be able to easily format the primary scenario and several alternatives, a special model design is employed for the scenario description: Primarily the flow contains a non-empty list of steps that are ordered sequentially according to the scenario execution. This list conveys the *primary scenario* of the use case, as shown in Example 6.4, "Primary Scenario in Two Steps".

Example 6.4. Primary Scenario in Two Steps

```
1 <flow>
2  <step>User enters path of the location to view.</step>
3  <step>Contents of the location are displayed.</step>
4 </flow>
```

After the steps involved to describe the primary scenario, the `flow` element may contain an optional list of alternatives, each with an appropriate description and a proper list of steps. A single alternative may start at a specified step of the primary scenario and either end outside of the primary scenario or route back into it at a later step. In order to correctly designate the starting point of an alternative, or the point where the primary scenario is continued, the steps have to be assigned optional names, as shown in Example 6.5, "Description with Two Alternatives".

Example 6.5. Description with Two Alternatives

```
 1 <flow>
 2  <step name="select_path">User enters the path to view.</step>
 3  <step name="display">The specified path is displayed.</step>
 4  <step name="document">The path does not denote a document</step>
 5  <step>The contents of path are displayed</step>
 6  <alternatives>
 7   <alternative start="select_path" continue="display">
 8    <description>No path selected</description>
 9    <step>The default path is the root path of the repository</step>
10   </alternative>
11   <alternative start="document">
12    <description>Document selected</description>
13    <step><include uc:ref="uc.view_document"/></step>
14   </alternative>
15  </alternatives>
16 </flow>
```

The steps of the primary scenario are assigned a key that can be referenced from the alternatives as start and continuation. In the grammar, this is implemented using `xsd:unique`[2], as can be seen in the declaration of `flow` in Example 6.6, "Definition of Scenario with Alternatives".

[2] Initially the design involved using `xsd:key` on the step level and `keyref` on the declaration of the alternative. Due to the `name` attribute of step being optional, this could not be implemented using XSD.

Example 6.6. Definition of Scenario with Alternatives

```
 1 <element name="flow">
 2  <complexType>
 3   <sequence>
 4    <element maxOccurs="unbounded" ref="uc:step"/>
 5    <element minOccurs="0" ref="uc:alternatives"/>
 6   </sequence>
 7  </complexType>
 8  <unique name="stepName" >
 9   <selector xpath="uc:step"/>
10   <field xpath="@name"/>
11  </unique>
12 </element>
13 <element name="step">
14  <complexType mixed="true">
15   <complexContent>
16    <extension base="uc:text">
17     <sequence>
18      <element ref="uc:include" minOccurs="0"/>
19     </sequence>
20     <attribute name="name" type="NCName"/>
21    </extension>
22   </complexContent>
23  </complexType>
24 </element>
25 <element name="alternatives">
26  <complexType>
27   <sequence>
28    <element maxOccurs="unbounded" ref="uc:alternative"/>
29   </sequence>
30  </complexType>
31 </element>
32 <element name="alternative">
33  <complexType>
34   <sequence>
35    <element ref="uc:description"/>
36    <element ref="uc:step"/>
37   </sequence>
38   <attribute name="continue" type="NCName"/>
39   <attribute name="start" use="required" type="NCName"/>
40  </complexType>
41 </element>
```

Logical Hierarchy

In order to provide a minimal organizational structure, use cases may be grouped in sections. A `section` element contains a title and a non-empty list of use cases, and may also contain other sections. The root element is `use-cases` and has a title, description and a non-empty list of sections. Refer to Example 6.7, “Structure of a Use Case Document”.

Example 6.7. Structure of a Use Case Document

```
 1 <use-cases xmlns="urn:use-case:description">
 2  <title>Collection of Use Cases</title>
 3  <uc:desc xmlns="http://www.w3.org/1999/xhtml">
 4   Use cases for the <i>Use Case Viewer</i> application.
 5  </uc:desc>
 6
 7  <section>
 8   <title>Main Use Cases</title>
 9   <uc … />
10   <uc … />
11   <uc … />
12  </section>
13  <section>
14   <title>Other Use Cases</title>
15   <section>
16    <title>Short Use Cases</title>
17    <uc … />
18    <uc … />
19   </section>
20   <section>
21    <title>Long Use Cases</title>
22    <uc … />
23    <uc … />
24    <uc … />
25   </section>
26  </section>
27 </use-cases>
```

Inclusion and Extension

Corresponding to *include* or *extend* associations between use cases, as employed in use case diagrams [Zuse04], the references between use cases have to be supported. In addition to elements for `include` and `extend`, a generic reference (`ref`) is provided that can be used inside textual descriptions.

All three elements contain a required `ref` attribute that indicates the target of the reference. The last step of the last alternative in Example 6.5, “Description with Two Alternatives” shows such a reference.

In order for this to work, the use case itself has to provide an id that is unique for this document. Employing XSD, the attribute `xml:id` is used, which is of type *xs:ID*. Refer to Example 6.8, “Use Case xml:id Attribute”.

Example 6.8. Use Case xml:id Attribute

```
1 <element name="uc">
2  <complexType>
3   <sequence>
4    … <!-- elements of use case -->
5   </sequence>
6   <attribute ref="xml:id" use="required"/>
7  </complexType>
8 </element>
```

6.2.3. Change Annotation

In addition to the information to cover use case descriptions, it is necessary to place change meta-information into the document. The format distinguishes three types of changes, similar to those

types of changes employed in current document-centric formats (refer to the paragraph called "Inline Versioning" (p.56)):

- Additions
- Removals
- Content changes

Change annotations are applied during the comparison process and directly attached to the changed nodes inside the document. Depending on the type of change, nodes are either annotated via an additional `change` attribute, or wrapped into a `change` element. Both, the attribute and the element, indicate the type of change and trigger appropriate highlighting in the style layer.

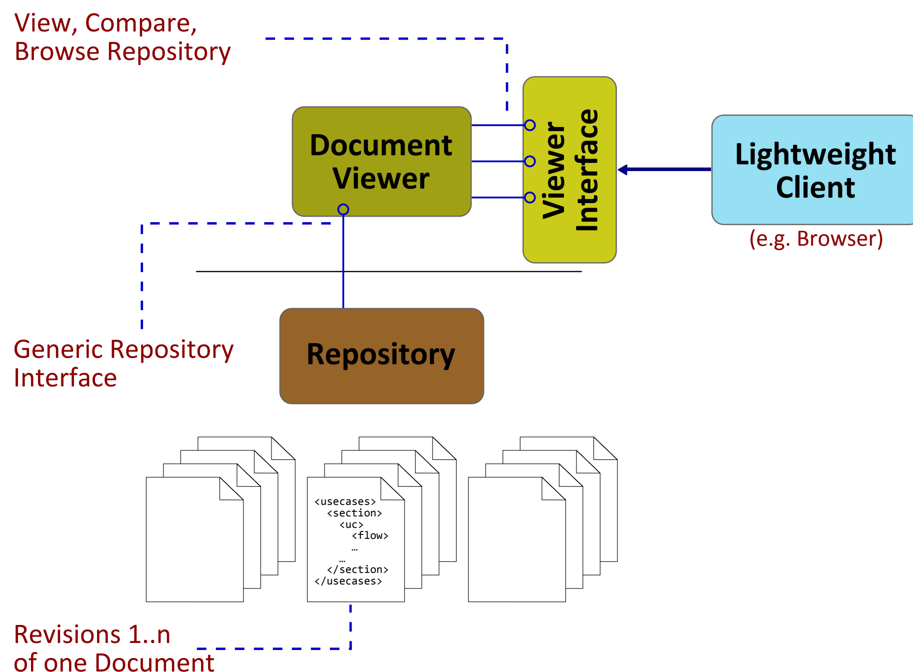
Attribute-Level Change Annotation. This is used for single elements that have been removed or added. The attribute's value is used to indicate the type of change.

Element-Level Change Annotation. This "wrapper"-element is used, if the changes do not have an enclosing parent element, which is often the case for textual changes, or deletions of text nodes. The `type` attribute is used to indicate the type of change.

6.3. Document Viewer

The document viewer provides a read-only presentation of the document. It is primarily intended as a document browser with transparent versioning capabilities. Refer to Figure 6.1, "Document Viewer Providing an Interface for Lightweight Clients" for a depiction of the viewer's architecture.

Figure 6.1. Document Viewer Providing an Interface for Lightweight Clients



The documents provided by the repository are managed inside the *document viewer*. Basic functionality to access versioned documents is provided via the *viewer interface*.

This section describes the document viewer's functions and how they are implemented.

6.3.1. Document Comparison

The primary functionality of the document viewer lies in providing means for the comparison of documents. For a single document, the viewer has to retrieve the available versions (revisions or variations) of interest and display a list for selection. The user may then select two different versions for comparison.

Revision Streams

For a location in the repository, the viewer therefore retrieves a `RepositoryEntry` which holds a reference to its name and location, and provides access to one or more `RevisionStreams`. The revision streams are based on version graphs as described by Conradi and Westfechtel [Conr98]. Revision streams are distinguished according to:

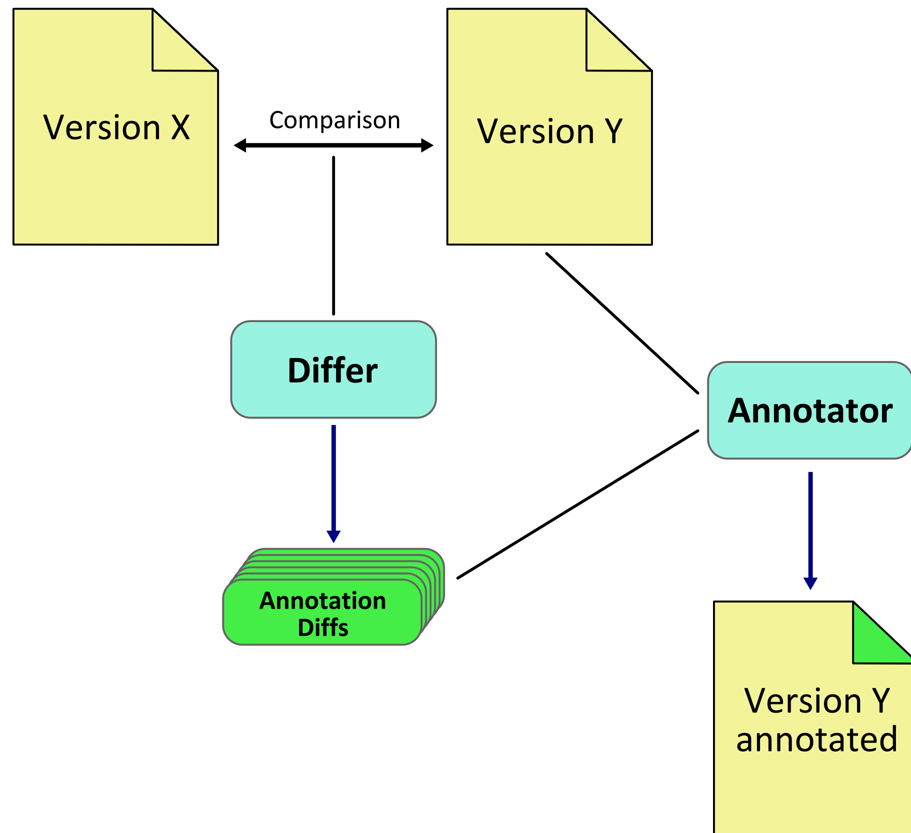
Branches
: Calculated via successor relationships, the initial version of a document starts the main branch. With each offspring, a new branch is created and will be treated as separate revision stream.

Label
: The different labelled versions of a document are put into a separate revision stream. Labels pointing to the same version are merged in order to reduce false duplicates.

Diff and Annotation Inference

The computation of differences as a basis for the creation of change annotations is based on the XML diffing library xmlunit and is split into two main components encapsulated in the `XmlDiff-Comparator`, as shown in Figure 6.2, “Create Change Annotations from Comparison”.

Figure 6.2. Create Change Annotations from Comparison



From the results of the comparison, the *differ* creates a set of annotation diffs. The *annotator* then applies these annotations to version Y in order to receive an annotated document.

The `XmlUnitDiffer` ignores attribute order and namespaces in order to produce difference annotations on an element level. The differ employs a configurable set of diff annotators depending on the type of difference as indicated by xmlunit. Refer to Example 6.9, “Spring Configuration for XmlUnitDiffer” for this part of the configuration.

Example 6.9. Spring Configuration for XmlUnitDiffer

```
1 <bean id="xmlUnitDiffer" class="….XmlUnitDiffer">
2  <property name="xpathNodeExtractor" ref="xpathExtractor"/>
3  <property name="diffAnnotators"><map>
4   <entry>
5    <key>
6     <util:constant static-field="….CHILD_NODE_NOT_FOUND_ID"/>
7    </key>
8    <bean class="….AnnChildNotFound" parent="annTemplate"/>
9   </entry>
10   <entry>
11    <key>
12     <util:constant static-field="….TEXT_VALUE_ID"/>
13    </key>
14    <bean class="….AnnTextValue" parent="annTemplate"/>
15   </entry>
16   <entry>
17    <key>
18     <util:constant static-field="….NODE_TYPE_ID"/>
19    </key>
20    <bean class="….AnnNodeType" parent="annTemplate"/>
21   </entry>
22   <entry>
23    <key>
24     <util:constant static-field="….ELEMENT_NUM_ATTRIBUTES_ID"/>
25    </key>
26    <bean class="….AnnElementAttributes" parent="annTemplate"/>
27   </entry>
28  </map></property>
29 </bean>
30
31 <bean id="annTemplate" class="….AnnNodeType">
32  <property name="xpathNodeExtractor" ref="xpathExtractor"/>
33 </bean>
```

A single difference annotation as created by one of the diff annotators contains the following information:

- `ChangeType`: added, removed, adapted, content
- XPath pointing to the location of the change
- Reference to the changed node

These annotations are then ordered and consolidated. The `XmlAnnotator` reversely applies the annotations to the content depending on the type of change and the location of the affected node. The changed node is either replaced with an appropriate `change` element, or receives a new `change` attributed.

6.3.2. Aggregation

At the downstream interface, the interface to the repository, the document viewer provides an extensible design that can be used to attach to different repositories. The `BackendCatalog` manages a list of handlers, each used to connect to a different type of repository. A handler implements two methods:

- `canProcess(String url)` indicates whether a certain URL is supported by this handler.
- `process(String url)` connects to the repository using the URL and returns a `UseCaseRepository`.

The references to the repositories are cached, and connections may be reused. It is possible to connect to different remote repositories which are only accessible from the physical location of the document viewer. In case new repository types have to be integrated, this design only requires the implementation of the handler interface.

6.4. Repository

In my choice of modules, the repository uses Subversion, a very popular version control system. On top of this, the hosting facility Google Code[3] that provides the infrastructure for the Use Case Creator project, has implemented several mechanisms that can be used by project members. This section shows how the combined system of Subversion concepts and hosting services can be leveraged with the examples of feeds and integration.

6.4.1. Feeds

Every project hosted in the employed hosting facility publishes updates in the form of Atom [RFC4287] feeds[4]. Commits to the version control system are published in a single feed, with one entry per revision, as can be seen in Example 6.10, “Single Entry for Revision 10 in the 'Source Changes' Feed”. Being in a standard feed format, this information can easily be reused for integrating with other applications or services.

Example 6.10. Single Entry for Revision 10 in the 'Source Changes' Feed

```
 1 <entry>
 2  <updated>2008-05-30T10:06:36Z</updated> ❶
 3  <id>…/svnchanges/basic/10</id>         ❷
 4  <link rel="alternate" type="text/html" href="…/detail?r=10" />❸
 5  <title>Revision 10: use case: […]❹</title>
 6  <author>
 7   <name>kariem.hussein</name>
 8  </author>
 9  <content type="html">
10 Changed Paths: ❺
11     Modify    /trunk/uc/style/preview.css
12     Modify    /trunk/uc/style/uc2html.xsl
13     Modify    /trunk/uc/uc-viewer.xml
14
15 ❹use case:
16  - id for first use case
17  - new section and use case for generic repository navigation
18 […]
19   </content>
20 </entry>
```

❶ The entry's `updated` element shows the commit date in UTC.

❷ `id` corresponds to the URL inside the feed and ends in the revision number.

❸ An alternative link points to a repository viewer of the hosting facility, showing the details of the revision.

❹ Commit comments attached to the revision are displayed in the title with whitespace normalized and at the end of the HTML-escaped `content`. Both were trimmed for better readability.

❺ The first part of the `content` contains an HTML-escaped list of changed paths prefixed by *Add*, *Delete*, and *Modify*. This was converted to spaces and newlines for better readability.

[3]Google Code [http://code.google.com/hosting/] is a free software hosting facility for open source projects, comparable to SourceForge.net [http://sourceforge.net/]. Refer to the Wikipedia category on free software hosting facilities [http://en.wikipedia.org/wiki/Category:Free_software_hosting_facilities] for an overview. Other hosting facilities provide similar infrastructure for the integration with other services.

[4]Refer to Feeds in the Google Code support project [http://code.google.com/p/support/wiki/Feeds].

Outside of the context of open source hosting facilities, there are applications that provide similar information on top of an underlying version control system. Examples are Edgewall Trac[5] and Atlassian FishEye[6], which both provide RSS feeds.

Trac. provides a *Timeline* view that incorporates several types of events that can occur inside Trac. In addition, the *Revision Log* provides feeds on the basis of the repository path, allowing for file-specific feeds.

Fisheye. has two main views which both provide feeds as alternative output format. The *Browse* view allows for navigating to the source tree and provides feeds on a resource basis. The *Changelog* provides customizable feeds that can be restricted based on branches, authors, or tags.

If the underlying version control system is, for example, Mercurial or Git, appropriate changelog feeds (in both cases in Atom) are already provided by the respective web interfaces that are distributed with the systems.

6.4.2. Integration

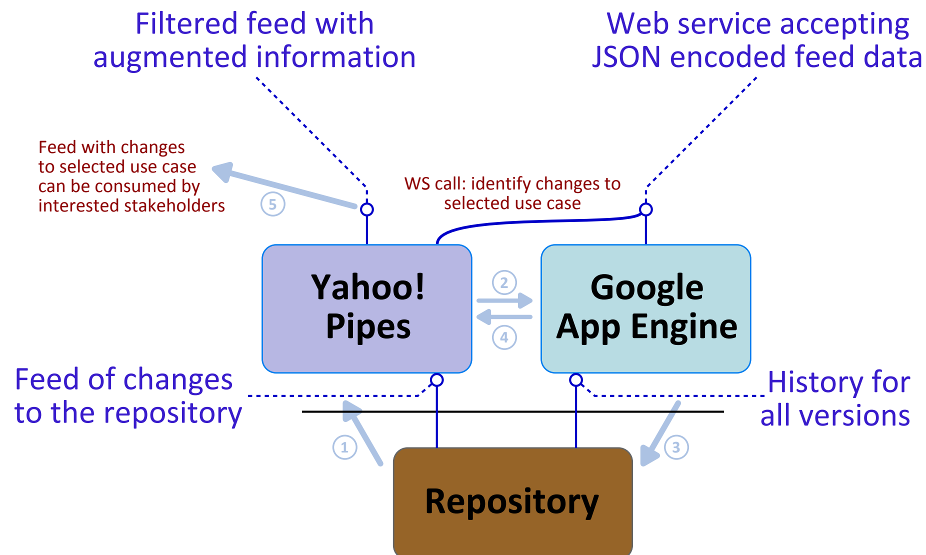
Information that has been made accessible in a standard way through the use of feeds, can now be integrated with other tools or applications. A sample scenario and implementation for the integration is provided.

Scenario: Observe Changes to Single Use Case. An interested stakeholder identifies the specification and the use case that should be observed. Updates to the specification document that do not touch the specified use case are ignored. Wong and Hong [Wong08] classify this as *Real-time Monitoring* "intended to make the user aware of changes."

As implementation for this integration example, I have selected two publicly available web services that will work together in the solution for the repository: Yahoo! Pipes[7] is used to provide a user interface for parameterization and for basic filtering. It will also provide the final feed URL. Google App Engine[8] will search through the filtered elements to see whether changes have affected the selected use case and filter out entries in which the use case has not changed. The layout and responsibilities are depicted in Figure 6.3, "Integration Scenario for Repository". I will refer to the first application as *Pipe* and the second as *App Engine*.

[5] Trac [http://trac.edgewall.org/] is a web-based project management tool. It provides feeds as documented in TracRss [http://trac.edgewall.org/wiki/TracRss].
[6] FishEye [http://www.atlassian.com/software/fisheye/] is a web application that tracks code changes and provides search facilities and statistics. Update notifications are implemented through feeds or email.
[7] According to Riabov et. al [Riab08], Yahoo Pipes [http://pipes.yahoo.com] is a "configurable application for automated processing of syndication feeds [that] offers hosted feed processing".
[8] Google App Engine [http://code.google.com/appengine/] is a web platform and SDK provided by Google. At the time of writing, applications can be written using the Python programming language and are hosted at Google premises.

Figure 6.3. Integration Scenario for Repository



The changes feed provided by the repository is processed by a Yahoo Pipe (selection, filtering). In order to identify changes to specific use cases, a web service hosted on Google App Engine is used.

Details on Pipe Implementation

The flow composed to implement the pipe requires two parameters:

path
: This is the path of the specification relative to the repository root. By default, a value of *trunk/uc/uc-viewer.xml* is assumed.

uc_id
: Every use case has a required `xml:id` attribute. The value of this attribute is used to identify the use case within the specification.

The *path* parameter is used to filter only relevant revisions from the feed of changes provided by the repository. Only elements of the feed are considered in which the path was affected. After filtering, the processing in the pipe consists of the following steps:

1. Assign the revision associated with a feed entry to a separate element of the entry. The revision is stored in `item.uc.revision`.
2. Construct a URL that can be used to view the parts of the specification that were affected by the revision represented by the entry. The URL is stored in `item.uc.diffurl`.
3. Create a URL that points to the version of the specification at the revision represented by the entry. This URL is stored in `item.uc.location`.
4. Call app engine with the current feed as payload and *Use Case ID* parameter in order to block out entries that did not affect the given use case. This rather complex step could not be modeled in the pipe, and thus was externalized.
5. The resulting feed is provided as pipe output, and can be consumed by interested stakeholders in different formats.

The pipe is published at http://pipes.yahoo.com/kariem/usecases.

Details on App Engine Implementation

A pre-filtered feed is sent as JSON-encoded payload to the app engine web service. For further filtering, the *uc_id* is passed as request parameter. Subsequently, the following steps are performed:

1. The payload is un-marshalled into objects for further processing. Still the objects are referred to as feed and its elements as entries.
2. The affected version of the document and its predecessor are retrieved from the repository
3. The range inside the affected version for the use case is calculated by using a SAX parser.
4. Differences between the affected version and its predecessor are transformed into a set of ranges.
5. If the use case range and difference ranges for a certain revision overlap, the use case is deemed to have been affected in this revision. Otherwise the entry representing the revision is removed from the feed.
6. The feed is marshalled into a JSON-encoded string and returned as result of the web service call.

The app engine is published at http://uccreator.appspot.com/. The main page provides a form to trigger the web service, while the effective web service is located at http://uccreator.appspot.com/changes. The sources are published — alongside the sources for all other implementation — at http://usecases.googlecode.com/.

Chapter 7. Validation and Conclusion

Before concluding the thesis, this chapter presents a validation based on different scenarios and tasks.

7.1. Validation

In order to validate the presented framework, it has to be put into the context of a scenario that can be compared to similar scenarios without the framework. After presenting the different scenarios, they are compared on the basis of several tasks.

7.1.1. Scenarios

Two other scenarios are observed for comparison: the first being considered prevalent in many IT companies (personal opinion based on my experience) will be called *traditional*, while the second is a real-world scenario taken from a single IT company, and referred to as *informed*. The framework presented in this section will be referred to as *standards-based*.

The classification of the scenarios is conducted at the following levels:

Documentation Format
: This is the primary base format in which the SRS is stored. If a variety of derivative documentation is generated, the primary base format is the format being considered the master. Changes to derivatives have to be included into the specification in the base format.

Communication
: The primary means of communicating requirements in the form of full documents (SRS) or single requirements and rules. Requirements that have only been transported by word of mouth and have not found a way into written communication are not considered.

Versioning
: Another assumption is that a reliable version model is in place, which is considered not being a technical, but an organizational challenge. It is also assumed that a version control system is an integral part of working professionally in an IT company. On this level the following types are distinguished: *implicit* versioning that is triggered automatically upon committing changes, and *explicit* versioning, which is the process of assigning identifiers for further reference.

The description of the scenarios is shown in Table 7.1, “Validation Scenarios”.

Table 7.1. Validation Scenarios

Classification	Scenario A *Traditional*	Scenario B *Informed*	Scenario C *Standards-based*
Documentation Format	The format natively provided with the team/company-wide word processor being used for all documentation. This could be for example Word Doc, OOXML, or ODF. The choice of format is based on the word processor being used.	All documents are authored using a wiki. Derivatives are created from one or more pages in the wiki, which are selected during export.	Most of the general documentation is written in a data-centric documentation format, such as DocBook or DITA. Requirements documentation uses a DSL that provides only features that are necessary for the documentation.
Communication	Email is the primary means of communicating requirements. The SRS is attached to the email.	Email is primarily used to communicate changes or major updates. They contain a link to the wiki page containing the requirements.	Similar to *B*.

Classification	Scenario A *Traditional*	Scenario B *Informed*	Scenario C *Standards-based*
Versioning	Implicit, using a standard version control system. Explicit versioning is technically possible.	Implicit, on a per-resource (wiki page) level, as provided by the system. Explicit versioning is not possible[i]	Implicit, using a standard version control system. Explicit versioning is required upon completion of certain milestones or as preparation for reviews.

[i]Wikis used by projects in the open source project hosting facility Google Code [http://code.google.com/hosting] are stored with a Subversion backend. Such an approach technically allows explicit versioning of a set of pages.

7.1.2. Tasks

The following tasks can be used to compare the processes involved in the different scenarios to attain certain targets or solve a certain problem. The problem description is followed by an assumed order of steps for each scenario. The following tasks are used:

- Distribute version as a reference
- Integration of feedback
- Inform interested stakeholders of an updated document
- Review changes that have been made to a document

For simplicity, the assumption is made that a single person is largely responsible for a single document. I explicitly exclude simultaneous joint authoring as provided by popular online services.

Distribute Version as a Reference

How does a person distribute a certain version as a reference, e.g. to a person in a developer or tester role?

Traditional

1. The document is assigned a version for further reference.
2. A list of interested stakeholders is compiled.
3. The document is sent out to the interested stakeholders[i].

Informed

1. The most recent version of a single wiki page is used as a reference[ii].
2. A list of interested stakeholders is compiled.
3. The link to the wiki page is sent out to the interested stakeholders. The version refers to the version of the wiki page at the time of sending the mail[iii].

It is very difficult to manage variations in parallel in a wiki. From my personal experience, the most widely used approach is copying the contents of one wiki page to a new page and using this page for one variation. Thus, this results in managing two separate wiki pages, one for each variation.

Standards-based

1. A link to the requested version in the repository is distributed.
2. Additional stakeholders may subscribe to changes to the document.

Relevant links to other documents or resources are provided in the document.

[i]In a best-case scenario, the most recent version, or requested version is stored in the version control system and, given that the requestor has access to that part, does not need to be distributed as attachment.

[ii]Depending on the editors of the wiki page, the information contained therein is linked with other relevant information.

[iii]Inherently the page and associated (linked) wiki pages change over time.

Integration of Feedback

How is feedback received and integrated in order to create a successor?

Traditional

1. Reviewers update documents they have received directly[i] and send them back to the author.
2. If necessary the author contacts one or more of the reviewers for clarification via private communication.
3. Feedback from each reviewer is merged into the document one by one.

Informed

1. The author may restrict access to certain sections (e.g. content not under review) and assign privileges to users.
2. Users that can access the wiki may add comments to the wiki page, or edit directly.
3. The author (as editor) is responsible to clarify and structure the wiki page's content.

Standards-based

1. Reviewers may edit or annotate sections in the documentation and store their versions as branches to the version under review.
2. The author then merges the changes into the base document: Merges can be performed automatically (all changes of all authors are accepted), or stepwise with tool support.

[i]This is usually facilitated by enabling features such as change tracking available in some word processors, or locking regions which should not be edited.

Inform Interested Stakeholders of Updated Document

After receiving feedback, how does the author of the document inform a group of people about the updates?

Traditional

1. The author decides whether to replace the predecessor of the document (revision) or create a new version (variant).
2. Updates are stored in the document[i].
3. The updated document is sent to members of a recipient list.

Informed

1. Interested stakeholders may subscribe to changes in the most recent version of a document by subscribing to changes on the appropriate wiki page.
2. Updates to the wiki page are published to the subscribers without additional intervention from the author.

Standards-based Similar to *Informed*: stakeholders may subscribe to changes to a document, or a certain variation of a document. It is even possible to subscribe only to changes in single use cases, as shown in Section 6.4.2, “Integration”.

[i]From my experience, in this scenario it is very useful that the author manages a list of revisions at the start of the document, thus duplicating the information from the version control system in a less verbose manner.

Review Changes that have been made to a Document

How can an interested person review all changes that have been made to a specific document over a certain period of time?

Traditional

1. Retrieve logs from the version control system via a client external to the authoring tool.
2. In order to see differences between revisions, the client has to be instructed to download the two versions to show the differences[i].

Informed Wiki pages provide a history, but this history is usually not in context with other changes made simultaneously. This revision information may not be sufficient for specifications spread over more than one page.

1. Retrieve the history of the specific wiki page.
2. For each item in the history, refer to the history of linked pages to retrieve versions that existed at the same time.

Standards-based The document is opened in the *document viewer*, which provides transparent versioning as an integrated feature.

[i]With two distinct versions available, authoring tools usually provide sufficiently sophisticated ways of comparison.

7.1.3. Further Steps required for Validation

In order to validate the proposed framework, a statistically relevant number of teams has to be found which is in a working environment similar to either *traditional* or *informed*. These teams have to be willing to participate in a migration process towards the framework.

In a first step, the presented tasks have to be assessed in the current environment of the teams. After a migration to the proposed framework and appropriate time for accomodation, the same assessment criteria are applied to the processes involved in solving these tasks with the framework.

The following criteria are proposed:

- Total time involved for the author to finish a task.
- Total number (manual) steps performed by the author to finish a task.
- Total time involved for all participants per task.

In addition to the more easily measurable criteria above, the user acceptance of the framework should be assessed. The following questions are being asked before and after the migration. It is expected that answers to these questions are not correct, but the results give an indication of how much confidence users have in the framework and the surrounding processes.

- Is most of the important information covered in the requirements documentation?
- Does everybody have access to the information he/she needs?
- How long would a handover between authors of document X take?

Subjects of the questions have to be similar in old and new scenario in order to provide meaningful results.

7.2. Related Work

A lot of research has been done in the field of requirements authoring and structured documentation. Even before the gaining popularity of XML, structured documents and authoring environments were positively affected by work in this area[1]. This section shows a selection of similar work closely related to the topic of this thesis.

Liz Fraley — Single Sourcing [Fral03]. The intention of the project was to allow writers and editors to be more productive by introducing the concept of single-sourcing. The primary result denotes the migration from a combination of Adobe FrameMaker and WebWorks Publisher as authoring/publishing tools (with an implicit documentation format) and a network drive as repository (augmented by FrameMakers locking mechanism) to an environment based on DocBook/CALS as documentation format, Arbortext Epic Editor as authoring tool and Interwoven's TeamSite as repository. Publishing to PDF uses Arbortext E3 with FOSI (comparable to XSL-FO) while the HTML output is generated via XSLT.

Oliver Meyer — Validated XML Documents using Word Processor [Meye02]. In order to improve the exchange of information between authors and their publishers, the project supported the move from a scenario where authors mainly submit Microsoft Word documents which are then transformed into SGML for further processing to an authoring environment in which authors can create valid documents themselves, which are then submitted to the publishing company, thus reducing the misunderstandings between author and publisher. The described result *aTool*

[1]For examples refer to [Walk81], [Pres96], [Quin04].

is an extension to Microsoft Word that can be parameterized with a DTD and used to create and validate structured XML documents.

7.3. Conclusion

This section concludes the thesis. The summary is followed by an outlook to features and components to improve the benefit of the presented authoring tool.

7.3.1. Summary

In this thesis, I have presented an authoring framework for software requirements specifications (SRS) consisting of three main components that may work together on the basis of open standards: *documentation format*, *authoring tool*, and *repository*.

Tools that are currently widely employed provide very few features that are important to efficiently create and review an SRS. The authoring framework presented in this thesis provides several improvements and, due to the standards-based orientation, points for integration with existing tools.

Requirements Documentation Authoring

The characteristics of an SRS are widely understood and its content may be structured according to different standards. Elements and types of documentation used for different types of requirements are also very clear and mostly manageable. As shown, the role of requirements and their maintenance and management is very important in different software development processes.

The processes around the authoring of requirements documentation, such as those found in change management and quality assurance, provide a frame around what is expected from an author and, in general, the team creating the requirements documentation. I have presented existing maturity models for the processes and the documentation created therein.

Documentation Formats and Authoring Tools

Subsequently the focus lies on the *documentation format*. After a presentation of currently employed and traditionally widely used documentation formats, standardized formats that already found wide acceptance are introduced. For the selection of those formats, a set of restrictions is defined that should help drawing a close frame around this selection, and what the formats all have in common.

In the switch to the second component, the *authoring tool*, a set of tools that can be used to create and maintain an SRS is analyzed. The requirements for these tools — i.e. what features these tools should provide — are mostly taken from the previously mentioned sources and what was found out during the analysis of the processes involved in documentation authoring. A small set of criteria was created for the evaluation of such authoring tools, and compared the previously presented tools according to these criteria.

Modular Integration Concept

The final component of the framework is introduced during the architectural concept of the whole framework. The *repository* is a very generic component that is central to the authoring framework and, as shown in the architecture, provides essential mechanisms intended to facilitate document authoring and the related processes. The concept of the framework explains the mechanisms between the three components of the authoring framework.

For the first implementation of the framework, a *domain-specific language* (DSL) that covers the aspects needed to model use case decriptions was created. This DSL is used as a documentation format, and documents created with this language can be viewed using the *document viewer*,

which is a (feature-wise) small version of the authoring tool. As an integration example, a repository based on a version control system that publishes changes in a standardized format is used to show how easy it can be to implement a subscription mechanism for specific use cases inside documents, relying solely on open standards.

This chapter concludes with a descriptive validation based on a set of scenarios and tasks that should show the different approaches. It provides instructions and questions that can be used in an analysis to clearly validate the benefits of the framework presented in this thesis.

7.3.2. Outlook

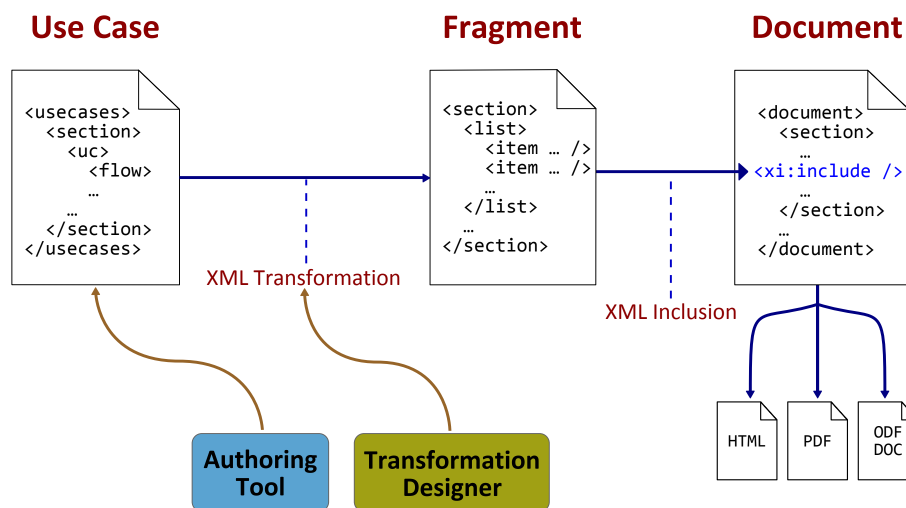
As this thesis should have shown, the introduction of open standards allows for an extensible and heterogeneous collection of technologies. There are still some elements that I believe would complete the picture of the authoring framework.

Transformation Designer

The styling component of the documentation format, either used inside the editor or for publication, has been presented as being based on one of two stylesheet languages. It might be difficult for a professional designer to handle the documentation format and a stylesheet language.

A dedicated tool to design and preview stylesheet languages could prove beneficial for the design process, resulting in fewer errors and more iterations. The position of the transformation designer in the toolchain is depicted in Figure 7.1, “Use Case Creation Process with Transformation Designer”.

Figure 7.1. Use Case Creation Process with Transformation Designer



The *transformation designer* may be used to develop different styles. In this diagram, the custom XML transformation converts use cases into DocBook fragments which are then included in larger documents, and subsequently transformed into the target format, depending on the employed toolchain.

References

Books

[Clem02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional. 2002. 0-20170-372-6.

[Copl04] James O. Coplien and Neil B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice Hall PTR. 2004.

[Cour04] Catherine Courage and Kathy Baxter. *Understanding Your Users: A Practical Guide to User Requirements Methods, Tools, and Techniques (Interactive Technologies)*. Morgan Kaufmann. 2004.

[Eise05] David J. Eisenberg. *OASIS OpenDocument Essentials*. O'Reilly & Associates, Inc.. December 2005. http://books.evc-cit.info/.

[Fowl08] Martin Fowler. *Domain Specific Languages (WIP)*. 2008. http://martinfowler.com/dslwip/.

[Kovi98] Benjamin L. Kovitz. *Practical Software Requirements: A Manual of Content and Style*. Manning Publications. 1998.

[Laue02] Soren Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley Professional. 2002.

[Leff03] Dean Leffingwell. *Managing Software Requirements*. Pearson Education. 2. 2003.

[Maci05] Leszek A. Maciaszek. *Requirements Analysis and System Design*. Addison Wesley Publishing Company. 2 Pap/Cdr. 2005.

[Mint01] Barbara Minto. *The Pyramid Principle: Present Your Thinking So Clearly That the Ideas Jump Off the Page and into the Reader's Mind*. Financial Times / Prentice Hall. 2001.

[Mitt04] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LaTeX Companion (Tools and Techniques for Computer Typesetting)*. Addison-Wesley Professional. 2004.

[Otei08] Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl. *The Not So Short Introduction to LaTeX 2e*. 4.26. September 2008.

[Rupp06] Chris Rupp. *Requirements-Engineering und -Management. Professionelle, iterative Anforderungsanalyse für die Praxis*. Hanser Fachbuchverlag. 2006.

[Schw04] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press. March 10, 2004. 0-73561-993-X.

[Stay03] Bob Stayton. *DocBook XSL: The Complete Guide*. Sagehill Enterprises. 3. 2002, 2003. http://www.sagehill.net/docbookxsl/.

[Wals05] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc.. 2. 1999-2003, 2005. http://docbook.org/tdg/. 1-56592-580-7.

[Wals94] Norman Walsh. *Making TeX Work (A Nutshell Handbook)*. O'Reilly & Associates, Inc.. 1994. http://makingtexwork.sourceforge.net/mtw/. 1-56592-051-1.

[Wieg04] Karl E. Wiegers. *Software Requirements, Second Edition*. Microsoft Press. 2003.

[Wier95] Roel J. Wieringa. *Requirements Engineering: Frameworks for Understanding*. John Wiley & Sons Ltd.. 1995.

[Zuse01] Wolfgang Zuser, Stefan Biffl, Thomas Grechenig, and Monika Köhle. *Software Engineering mit UML und dem Unified Process*. Pearson Studium. 2001.

[Zuse04] Wolfgang Zuser, Thomas Grechenig, and Monika Köhle. *Software Engineering. Mit UML und dem Unified Process*. Pearson Studium. 2. 2004.

Papers

[Albi96] Bill Albing. “Process constraints in the management of technical documentation”. *SIGDOC '96: Proceedings of the 14th annual international conference on Systems documentation*. ACM Press. 67–74. 1996. http://doi.acm.org/10.1145/238215.238257.

[Ande02] Kenneth M. Anderson, Susanne A. Sherba, and William V. Lepthien. “Towards large-scale information integration”. *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. ACM Press. 524–534. 2002. http://doi.acm.org/10.1145/581339.581403.

[Bae02] Hyerim Bae and Yeongho Kim. “A document-process association model for workflow management”. *Comput. Ind.*. Elsevier Science Publishers B. V.. 47. 139–154. 2002. http://dx.doi.org/10.1016/S0166-3615(01)00150-6.

[Bart07] Brent Barton and Evan Campbell. “Implementing a Professional Services Organization Using Type C Scrum”. *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. 275a-275a. 2007.

[Baye03] Petra Saskia Bayerl, Harald Lüngen, Daniela Goecke, Andreas Witt, and Daniel Naber. “Methods for the semantic analysis of document markup”. *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*. ACM Press. 161–170. 2003. http://doi.acm.org/10.1145/958220.958250.

[Bell76] T. E. Bell and T. A. Thayer. “Software requirements: Are they really a problem?”. *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press. 61–68. 1976.

[Berg01] Erik Berglund and Michael Priestley. “Open-source documentation: in search of user-driven, just-in-time writing”. *SIGDOC '01: Proceedings of the 19th annual international conference on Computer documentation*. ACM Press. 132–141. 2001. http://doi.acm.org/10.1145/501516.501543.

[Bern07] Philip A. Bernstein and Sergey Melnik. “Model management 2.0: manipulating richer mappings”. *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 1–12. 2007. http://doi.acm.org/10.1145/1247480.1247482. 978-1-59593-686-8.

[Boeh02] Barry W. Boehm. “Get Ready for Agile Methods, with Care”. IEEE Computer Society. 64–69. 2002. http://doi.ieeecomputersociety.org/10.1109/2.976920.

[Boeh86] Barry W. Boehm. “A spiral model of software development and enhancement”. *SIGSOFT Softw. Eng. Notes*. ACM. 14–24. 1986. http://doi.acm.org/10.1145/12944.12948. 0163-5948.

[Boeh88] Barry W. Boehm. “A Spiral Model of Software Development and Enhancement”. *Computer*. IEEE Computer Society. 61–72. 1988. http://doi.ieeecomputersociety.org/10.1109/2.59. 0018-9162.

[Booc02] Grady Booch. “Growing the UML”. *Software and Systems Modeling*. 1. 157–160. December 2002. http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s10270-002-0013-7.

[Bouk04] A. Boukottaya, C. Vanoirbeek, F. Paganelli, and O. Abou Khaled. “Automating XML documents transformations: a conceptual modelling based approach”. *CRPIT '31: Proceedings of the first Asian-Pacific conference on Conceptual modelling*. Australian Computer Society, Inc.. 81–90. 2004.

[Boye08] John M. Boyer. “Interactive office documents: a new face for web 2.0 applications”. *DocEng '08: Proceeding of the eighth ACM symposium on Document engineering*. ACM. 8–17. 2008. http://doi.acm.org/10.1145/1410140.1410145. 978-1-60558-081-4.

[Byer04] Simon Byers. “Information leakage caused by hidden data in published documents”. *Security & Privacy, IEEE*. 2. 23–27. 2004.

[Chid03] Boris Chidlovskii. “A structural adviser for the XML document authoring”. ACM Press. 203–211. 2003.

[Conr98] Reidar Conradi and Bernhard Westfechtel. “Version models for software configuration management”. *ACM Comput. Surv.*. ACM. 30. 232–282. 1998. http://doi.acm.org/10.1145/280277.280280. 0360-0300.

[Conw00] Candace L. Conwell, Rosemary Enright, and Marcia A. Stutzman. “Capability maturity models support of modeling and simulation verification, validation, and accreditation”. *CRPIT '27: Proceedings of the fifteenth conference on Australasian database*. WSC '00: Proceedings of the 32nd conference on Winter simulation. 819–828. 0-7803-6582-8. 2000.

[Dong04] Ce Dong and James Bailey. “Static analysis of XSLT programs”. *CRPIT '27: Proceedings of the fifteenth conference on Australasian database*. Australian Computer Society, Inc.. 151–160. 2004.

[Dyme04] Mark Dymetman. “Chart-parsing techniques and the prediction of valid editing moves in structured document authoring”. *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*. ACM. 229–238. 2004. http://doi.acm.org/10.1145/1030397.1030440. 1-58113-938-1.

[Eber02] Christof Ebert and Josef De Man. “e-R&D – Effectively Managing Process Diversity”. *Annals of Software Engineering*. 14. 73–91. December 2002. http://www.springer-link.com/openurl.asp?genre=article&id=doi:10.1023/A:1020545406509.

[Fagi99] Barry Fagin. “Computers, science, and the microsoft case”. *SIGCAS Comput. Soc.*. ACM Press. 29. 15–22. 1999. http://doi.acm.org/10.1145/382018.382026.

[Forw02] Andrew Forward and Timothy C. Lethbridge. “The relevance of software documentation, tools and technologies: a survey”. *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*. ACM Press. 26–33. 2002. http://doi.acm.org/10.1145/585058.585065.

[Fral03] Liz Fraley. “Beyond theory: making single-sourcing actually work”. *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*. ACM Press. 52–59. 2003. http://doi.acm.org/10.1145/944868.944880.

[Heit97] Constance Heitmeyer, James Kirby, and Bruce Labaw. “The SCR method for formally specifying, verifying, and validating requirements: tool support”. *ICSE '97: Proceedings of the 19th international conference on Software engineering*. 610–611. 1997. ACM. http://doi.acm.org/10.1145/253228.253498.

[Hopp03] Kenneth B. Hopper and Keith T. Rainey. “A pilot study of self-assessment of word processing and presentation software skills in graduate students in technical communication”. *Professional Communication Conference, 2003. IPCC 2003. Proceedings. IEEE International*. 8 pp.. 2003.

[Huan03] Shihong Huang and Scott Tilley. “Towards a documentation maturity model”. *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*. ACM Press. 93–99. 2003. http://doi.acm.org/10.1145/944868.944888.

[Iaco05] Ionut E. Iacob and Alex Dekhtyar. “xTagger: a new approach to authoring document-centric XML”. *JCDL '05: Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*. ACM Press. 44–45. 2005. http://doi.acm.org/10.1145/1065385.1065395.

[Jaya08] Magesh A. Jayapandian and H. V. Jagadish. “Expressive query specification through form customization”. *EDBT '08: Proceedings of the 11th international conference on Extending database technology*. ACM. 416–427. 2008. http://doi.acm.org/10.1145/1353343.1353395. 978-1-59593-926-5.

[Jazz95] Abdulaziz Jazzar and Walt Scacchi. "Understanding the requirements for information system documentation: an empirical investigation". *COCS '95: Proceedings of conference on Organizational computing systems*. ACM Press. 268–279. 1995. http://doi.acm.org/10.1145/224019.224048.

[John99] John W. Stamey, Jr. and Thomas M. Roth, III. "Technical documentation and related contractual liability". *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*. ACM Press. 105–109. 1999. http://doi.acm.org/10.1145/318372.318566.

[Judy08] Ken H. Judy and Ilio Krumins-Beens. "Great Scrums Need Great Product Owners: Unbounded Collaboration and Collective Product Ownership". *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*. 462–462. 2008.

[Kirn97] Tereza G. Kirner and Janaina C. Abib. "Inspection of software requirements specification documents: a pilot study". *SIGDOC '97: Proceedings of the 15th annual international conference on Computer documentation*. ACM Press. 161–171. 1997. http://doi.acm.org/10.1145/263367.263389.

[Korg96] Susan Korgen. "Object-oriented, single-source, on-line documents that update themselves". *SIGDOC '96: Proceedings of the 14th annual international conference on Systems documentation*. ACM Press. 229–237. 1996. http://doi.acm.org/10.1145/238215.238301.

[Kylm03] Roope Kylmäkoski. "Efficient authoring of software documentation using RaPiD7". *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society. 255–261. 2003.

[Lee02] Beum-Seuk Lee and Barrett R. Bryant. "Automated conversion from requirements documentation to an object-oriented formal specification language". *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. ACM Press. 932–936. 2002. http://doi.acm.org/10.1145/508791.508972.

[Lowr03] Paul Benjamin Lowry and Jay F. Nunamaker, Jr.. "Using Internet-Based, Distributed Collaborative Writing Tools to Improve Coordination and Group Awareness in Writing Teams". *IEEE Transactions on Professional Communication*. 46. 277–297. 2003.

[MacK03] Neil MacKinnon and Steve Murphy. "Designing UML diagrams for technical documentation". *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*. ACM Press. 105–112. 2003. http://doi.acm.org/10.1145/944868.944891.

[MacK04] Neil MacKinnon and Steve Murphy. "Designing UML diagrams for technical documentation: continuing the collaborative approach to publishing class diagrams". *SIGDOC '04: Proceedings of the 22nd annual international conference on Design of communication*. ACM Press. 120–127. 2004. http://doi.acm.org/10.1145/1026533.1026565.

[Makr05] Kristis Makris and Kyung Dong Ryu. *2005 USENIX Annual Technical Conference, FREENIX/Open Source Track*. April 2005.

[Mars99] C. Hugh Marsh. "The engineer as technical writer and document designer: the new paradigm". *SIGDOC Asterisk J. Comput. Doc.*. ACM Press. 23. 57–61. 1999. http://doi.acm.org/10.1145/311147.311159.

[Mern05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". *ACM Comput. Surv.*. ACM. 37. 316–344. 2005. http://doi.acm.org/10.1145/1118890.1118892.

[Meye02] Oliver Meyer. "aTool: creating validated XML documents on the fly using MS word". *SIGDOC '02: Proceedings of the 20th annual international conference on Computer documentation*. ACM Press. 113–121. 2002. http://doi.acm.org/10.1145/584955.584973.

[Oniz05] Makoto Onizuka, Fong Yee Chan, Ryusuke Michigami, and Takashi Honishi. "Incremental maintenance for materialized XPath/XSLT views". *WWW '05: Proceedings of the 14th international conference on World Wide Web*. ACM Press. 671–681. 2005. http://doi.acm.org/10.1145/1060745.1060843.

[Powe03] Norah Power and Tony Moynihan. “A theory of requirements documentation situated in practice”. *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*. ACM Press. 86–92. 2003. http://doi.acm.org/10.1145/944868.944887.

[Pres96] Paul Prescod. “Multiple media publishing in SGML”. *SIGDOC '96: Proceedings of the 14th annual international conference on Systems documentation*. ACM Press. 3–9. 1996. http://doi.acm.org/10.1145/238215.238241.

[Prie00] Michael Priestley and Mary Hunter Utt. “A unified process for software and documentation development”. *IPCC/SIGDOC '00: Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation*. IEEE Educational Activities Department. 221–238. 2000.

[Quin04] Vincent Quint and Irène Vatton. “Techniques for authoring complex XML documents”. *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*. ACM. 115–123. 2004. http://doi.acm.org/10.1145/1030397.1030422. 1-58113-938-1.

[Raja05] Prasad Rajagopal, Roger Lee, Thomas Ahlswede, Chia-Chu Chiang, and Dale Karolak. “A new approach for software requirements elicitation”. *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks. SNPD/SAWN 2005. Sixth International Conference on*. 32–42. 2005.

[Reid80] Brian K. Reid. “A high-level approach to computer document formatting”. *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press. 24–31. 1980. http://doi.acm.org/10.1145/567446.567449.

[Rene02] Allen Renear, David Dubin, and C. M. Sperberg-McQueen. “Towards a semantics for XML markup”. *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*. ACM Press. 119–126. 2002. http://doi.acm.org/10.1145/585058.585081.

[Riab08] Anton V. Riabov, Eric Boillet, Mark D. Feblowitz, Zhen Liu, and Anand Ranganathan. “Wishful search: interactive composition of data mashups”. *WWW '08: Proceeding of the 17th international conference on World Wide Web*. 46. 775–784. 2008. ACM. http://doi.acm.org/10.1145/1367497.1367602. 978-1-60558-085-2.

[Rönn05] Sebastian Rönnau, Jan Scheffczyk, and Uwe M. Borghoff. “Towards XML version control of office documents”. *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*. ACM Press. 10–19. 2005. http://doi.acm.org/10.1145/1096601.1096606.

[Ross88] John Minor Ross. “Documentation - the good, the bad and the ugly”. *SIGDOC '88: Proceedings of the 6th annual international conference on Systems documentation*. ACM Press. 41–46. 1988. http://doi.acm.org/10.1145/358922.358930.

[Royc87] W. W. Royce. “Managing the development of large software systems: concepts and techniques”. *ICSE '87: Proceedings of the 9th international conference on Software Engineering*. IEEE Computer Society Press. 328–338. 1987.

[Rube00] Philip Rubens and Sherry Southard. “Using new technologies for communication and learning”. *IPCC/SIGDOC '00: Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation*. IEEE Educational Activities Department. 185–189. 2000.

[Ruga00] Spencer Rugaber. “The use of domain knowledge in program understanding”. *Ann. Softw. Eng.*. J. C. Baltzer AG, Science Publishers. 9. 143–192. 2000.

[Salt75] Rita Seplowitz Saltz. “The non-technical person as technical writer/editor and documentation for the computer illiterate”. *SIGUCCS '75: Proceedings of the 3rd annual ACM SIGUCCS conference on User services*. ACM Press. 88–89. 1975.

[Salz05] Ben Salzberg and Trevor Murphy. “LaTeX: when word fails you”. *SIGUCCS '05: Proceedings of the 33rd annual ACM SIGUCCS conference on User services*. ACM Press. 241–243. 2005. http://doi.acm.org/10.1145/1099435.1099490.

[Sche88] Benson H. Scheff and Tom Georgon. “Letting software engineers do software engineering or freeing software engineers from the shackles of documentation”. *SIGDOC '88: Proceedings of the 6th annual international conference on Systems documentation*. ACM Press. 81–92. 1988. http://doi.acm.org/10.1145/358922.358938.

[Schn92] G. Michael Schneider, Johnny Martin, and W. T. Tsai. “An experimental study of fault detection in user requirements documents”. *ACM Trans. Softw. Eng. Methodol.*. ACM Press. 1. 188–204. 1992. http://doi.acm.org/10.1145/128894.128897.

[Shah08] Rajiv Shah, Jay Kesan, and Andrew Kennis. “Implementing open standards: a case study of the Massachusetts open formats policy”. *dg.o '08: Proceedings of the 2008 international conference on Digital government research*. Digital Government Society of North America. 262–271. 2008. 978-1-60558-099-9.

[Shan08] Uri Shani and Aviad Sela. “Software design using UML for empowering end-users with an external domain specific language”. *WEUSE '08: Proceedings of the 4th international workshop on End-user software engineering*. 52–55. 2008. ACM. http://doi.acm.org/10.1145/1370847.1370859.

[Shin05] Dong-Hoon Shin and Kyong-Ho Lee. “Generating XSLT scripts for the fast transformation of XML documents”. *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*. ACM Press. 1098–1099. 2005. http://doi.acm.org/10.1145/1062745.1062887.

[Stan98] IEEE Standards. “IEEE Recommended Practice for Software Requirements Specifications”. *IEEE Std 830-1998 (Revision of IEEE Std 830-1993)*. *IEEE Std 830-1998 (Revision of IEEE Std 830-1993)*. IEEE. 1. vi+31. October 1998.

[Stuc05] Alexander Stuckenholz. “Component evolution and versioning state of the art”. *SIGSOFT Softw. Eng. Notes*. ACM Press. 30. 7. 2005. http://doi.acm.org/10.1145/1039174.1039197.

[Tami08] Jari Tamir, Oleg Komogortsev, and Carl Mueller. “An effort and time based measure of usability”. *WoSQ '08: Proceedings of the 6th international workshop on Software quality*. 47–52. 2008. ACM. http://doi.acm.org/10.1145/1370099.1370111.

[Tant97] Duke Tantiprasut, John Neil, and Craig Farrell. “ASN.1 protocol specification for use with arbitrary encoding schemes”. *IEEE/ACM Trans. Netw.*. IEEE Press. 5. 502–513. 1997. http://dx.doi.org/10.1109/90.649464.

[Thom01] Bill Thomas and Scott Tilley. “Documentation for software engineers: what is needed to aid system understanding?”. *SIGDOC '01: Proceedings of the 19th annual international conference on Computer documentation*. ACM Press. 235–236. 2001. http://doi.acm.org/10.1145/501516.501570.

[Thom05] Peter L. Thomas and David F. Brailsford. “Enhancing composite digital documents using XML-based standoff markup”. *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*. ACM Press. 177–186. 2005. http://doi.acm.org/10.1145/1096601.1096647.

[Town88] George Towner. “Auto-updating as a technical documentation tool”. *DOCPROCS '88: Proceedings of the ACM conference on Document processing systems*. ACM Press. 31–36. 1988. http://doi.acm.org/10.1145/62506.62514.

[Vanh03] Jari Vanhanen, Juha Itkonen, and Petteri Sulonen. “Improving the interface between business and product development using agile practices and the cycles of control framework”. *Agile Development Conference, 2003. ADC 2003. Proceedings of the*. 71–80. 2003.

[Velo99] Lisa Veloz. “Implementing the Microsoft Office User Specialist program”. *SIGUCCS '99: Proceedings of the 27th annual ACM SIGUCCS conference on User services*. ACM Press. 195–198. 1999. http://doi.acm.org/10.1145/337043.337142.

[Vill02] Lionel Villard and Nabil Layaïda. “An incremental XSLT transformation processor for XML document manipulation”. *WWW '02: Proceedings of the 11th international conference on World Wide Web*. ACM Press. 474–485. 2002. http://doi.acm.org/10.1145/511446.511508.

[Visc93] Marcello Visconti and Curtis Cook. “Software system documentation process maturity model”. *CSC '93: Proceedings of the 1993 ACM conference on Computer science*. ACM Press. 352–357. 1993. http://doi.acm.org/10.1145/170791.170869.

[Walk81] Janet H. Walker. “The document editor: A support environment for preparing technical documents”. *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*. 44–50. 1981.

[Will04] Ashley Williams. “The documentation of quality engineering: applying use cases to drive change in software engineering models”. *SIGDOC '04: Proceedings of the 22nd annual international conference on Design of communication*. ACM Press. 4–13. 2004. http://doi.acm.org/10.1145/1026533.1026538.

[Wils97] William M. Wilson, Linda H. Rosenberg, and Lawrence E. Hyatt. “Automated analysis of requirement specifications”. *ICSE '97: Proceedings of the 19th international conference on Software engineering*. ACM Press. 161–171. 1997. http://doi.acm.org/10.1145/253228.253258.

[Wong08] Jeffrey Wong and Jason Hong. “What do we "mashup" when we make mashups?”. *WEUSE '08: Proceedings of the 4th international workshop on End-user software engineering*. ACM. 22. 35–39. 2008. http://doi.acm.org/10.1145/1370847.1370855. 978-1-60558-034-0.

[XuLi07] Li Xu. “Project the wiki way: using wiki for computer science course project management”. *J. Comput. Small Coll.*. Consortium for Computing Sciences in Colleges. 22. 109–116. 2007.

[Zuse05] Wolfgang Zuser, Stefan Heil, and Thomas Grechenig. “Software quality development and assurance in RUP, MSF and XP: a comparative study”. *3-WoSQ: Proceedings of the third workshop on Software quality*. ACM Press. 1–6. 2005. http://doi.acm.org/10.1145/1083292.1083300.

Specifications

[DITA] Michael Priestley, Robert D. Anderson, and JoAnn Hackos. “DITA Version 1.1”. August 2007. http://docs.oasis-open.org/dita/v1.1/OS/overview/overview.html. OASIS.

[EXI] John Schneider and Takuki Kamiya. “Efficient XML Interchange (EXI) Format 1.0”. July 2008. http://www.w3.org/TR/2008/WD-exi-20080728/. World Wide Web Consortium. 4th Working Draft.

[FI] ITU-T Study Group 17. “ITU-T Rec. X.891 (05/2005) Information technology - Generic applications of ASN.1: Fast infoset”. May 2005. http://www.itu.int/rec/T-REC-X.891-200505-I. International Telecommunication Union.

[ODF] Michael Brauer, Patrick Durusau, Gary Edwards, David Faure, Tom Magliery, and Daniel Vogelheim. “Open Document Format for Office Applications (OpenDocument) v1.0”. May 2005. http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf. OASIS.

[ODF-1.1] Patrick Durusau, Michael Brauer, and Lars Oppermann. “Open Document Format for Office Applications (OpenDocument) v1.1”. February 2007. http://docs.oasis-open.org/office/v1.1/OS/OpenDocument-v1.1.pdf. OASIS.

[OOXML] Ecma TC45. “Office Open XML File Formats”. December 2006. http://www.ecma-international.org/publications/standards/Ecma-376.htm. Ecma International.

[OOXML-WP] Ecma TC45. “Office Open XML Overview”. December 2006. http://www.ecma-international.org/news/TC45_current_work/OpenXML%20White%20Paper.pdf. Ecma International.

[RELAX] James Clark and Makoto Murata. “RELAX NG Specification”. December 2001. http://www.relaxng.org/spec-20011203.html.

[RFC2141] R. Moats. *RFC 2141: URN Syntax*. May 1997. http://tools.ietf.org/html/rfc2141. proposed standard.

[RFC4287] M. Nottingham and R. Sayre. *RFC 4287: The Atom Syndication Format*. December 2005. http://tools.ietf.org/html/rfc4287. proposed standard.

[RTF-1.6] Microsoft Corporation. “Rich Text Format Specification, version 1.6”. May 1999. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnrtfspec/html/rtfspec.asp.

[RTF-1.8] Microsoft Corporation. “Rich Text Format Specification, version 1.8”. April 2004. http://www.microsoft.com/downloads/details.aspx?familyid=ac57de32-17f0-4b46-9e4e-467ef9bc5540.

[SVG] Jon Ferraiolo, [illegible] and, Dean Jackson. “Scalable Vector Graphics (SVG) 1.1 Specification”. January 2003. http://www.w3.org/TR/SVG11/. World Wide Web Consortium. W3C Recommendation.

[XIncl] Jonathan Marsh and David Orchard. “XML Inclusions (XInclude) Version 1.0”. December 2004. http://www.w3.org/TR/xinclude/.

[XLink] Steve DeRose, Eve Maler, and David Orchard. “XML Linking Language (XLink) Version 1.0”. June 2001. http://www.w3.org/TR/xlink/.

[XML-ID] Jonathan Marsh, Daniel Veillard, and Norman Walsh. “xml:id Version 1.0”. September 2005. http://www.w3.org/TR/xml-id/.

[XML-WD] Tim Bray and C. M. Sperberg-McQueen. “Extensible Markup Language (XML)”. November 1996. http://www.w3.org/TR/WD-xml-961114.

[XML1.0] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. “Extensible Markup Language (XML) 1.0”. February 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[XML1.0(e4)] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. “Extensible Markup Language (XML) 1.0 (Fourth Edition)”. September 2006. http://www.w3.org/TR/2006/REC-xml-20060816/.

[XMLNS] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. “Namespaces in XML 1.1 (Second Edition)”. August 2006. 2. http://www.w3.org/TR/xml-names11/.

[XPath] James Clark and Steve DeRose. “XML Path Language (XPath)”. November 1999. http://www.w3.org/TR/xpath.

[XSD-1] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. “XML Schema Part 1: Structures Second Edition”. October 2004. http://www.w3.org/TR/xmlschema-1/.

[XSD-2] Paul V. Biron and Ashok Malhotra. “XML Schema Part 2: Datatypes Second Edition”. October 2004. http://www.w3.org/TR/xmlschema-2/.

[XSL] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. “Extensible Stylesheet Language (XSL) Version 1.0”. October 2001. http://www.w3.org/TR/xsl.

[XSLT] James Clark. “XSL Transformations (XSLT) Version 1.0”. November 1999. http://www.w3.org/TR/xslt.

Web references

[Anis05] Chris Aniszczyk and Lawrence Mandel. “Authoring with Eclipse”. December 2005. http://www.eclipse.org/articles/Article-Authoring-With-Eclipse/AuthoringWithEclipse.pdf.

[Crox05] Martin Croxford and Roderick Chapman. *Correctness by Construction: A Manifesto for High-Integrity Software*. *Journal of Defense Software Engineerng*. 2005. http://www.stsc.hill.af.mil/crosstalk/2005/12/0512CroxfordChapman.html.

[IDABC04] IDABC. "European Interoperability Framework for Pan-European eGovernment Services". 2004. http://europa.eu.int/idabc/document/3761.

[IDABC05] IDABC. *Documentation on the Promotion of Open Document Exchange Format*. 2005. http://europa.eu.int/idabc/document/3439.

[Jaco03] Ivar Jacobson. *Use Cases: Yesterday, Today, and Tomorrow*. November 2003. http://www-128.ibm.com/developerworks/rational/library/775.html.

[Krav05] Christian Kravogel and Boris Horner. *DITA - Getting Started*. 2005. http://idealliance.org/proceedings/xtech05/papers/04-02-02/.

[Mass05] Commonwealth of Massachusetts. *Enterprise Technical Reference Model*. 2005. http://www.mass.gov/Aitd/docs/policies_standards/etrm3dot5/ETRM_v3dot5draft_information.pdf.

[Open05] OASIS Open. *Approval of OpenDocument v1.0 as OASIS Standard*. 2005. http://lists.oasis-open.org/archives/tc-announce/200505/msg00001.html.

[Orch03] David Orchard and Norman Walsh. *Versioning XML Languages*. October 2003. http://www.w3.org/2001/tag/doc/versioning-20031003.

[Sunw05] Sun Microsystems. *Sun Microsystems Adds Two New Services to Sun Grid Utility, Easing Transition to Emerging Web 2.0*. November 2005. http://www.sun.com/smi/Press/sunflash/2005-11/sunflash.20051101.2.html.

[Valo03] Valoris. "Comparative assessment of Open Documents Formats". May 2003. ec.europa.eu/idabc/servlets/Doc?id=17982.

[Wals05] Norman Walsh. "DocBook: From Syntax to Publication". November 2005. http://nwalsh.com/docs/tutorials/xml2005/slides.pdf.

Colophon

This thesis was produced with DocBook and the XSL stylesheets. The transformation from XML to FO employed xmllint for preprocessing and Saxon for the final big step. The formatting processor used to create the final PDF output is RenderX XEP. The bibliography was created with JabRef and a custom DocBook export layout, with minor manual modifications. Most of the diagrams were created with Inkscape. The thesis was edited in jEdit and Kate.