An Associative Documentation Model

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Fredi Frank

1999

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

Typesetting performed by LaTeX .

Author's address:

Fredi Frank
Software Composition Group
University of Berne
Institute of Computer Science and Applied Mathematics (IAM)
Neubrückstrasse 10
CH-3012 Bern
Switzerland

Email: frank@iam.unibe.ch or ffrank@mail.tic.ch
WWW: http://www.myhome.ch/ffrank

Abstract

This diploma work examines the MAINTENANCE OF TECHNICAL DOCUMENTATION within an software engineering process. The characteristics of technical documentation and its behaviour within an dynamic software development environment are important to understand the problems that occur with technical documentation. I explore the factors that influence the development and the resulting quality of the technical documentation.

The RELATIONSHIP BETWEEN SOFTWARE SOURCE CODE AND TECHNICAL DOCUMENTATION is used to coordinate the development of the technical documentation with the software development. The principle to match software entity names with documentation segments defines the relationship between software and documentation. I demonstrate how it works and how it is used for coordination of the software development and the technical documentation development. An analysis of different name representations and documentation segmentation structures shows the influence of the structures on the creation of relations. I explore under which conditions relations are generated that fit best to the relationship between software and documentation that exists in reality.

The ASSOCIATIVE DOCUMENTATION MODEL (ADM) builds on the relationship between software and documentation that is determined by matching of software entity names within documentation. The ADM focuses on three aspects: It concentrates on the extraction of the names of software entities. ADM uses Famix models that are capable to represent any object-oriented software and detects the software entities to be represented by their names. Additionally, ADM considers structural relationships between the software entities that are given by inheritance, aggregation, invocation and access. The second aspect is the representation of the relationship between software and documentation. Especially the influence of the inner relationship of software entities on the relationship between software and documentation is important. The third aspect is to get the detection of the software entity names and the generation of the relationship between software and documentation into a consistent model. It serves as a uniform model of the software-documentation relationship for any application that uses this model.

CHANGE IMPACT DETECTION is an sample application of the ADM. It determines changes between two software versions by comparison of the models of these versions. Differences between the models are interpreted as changes. The names of the entities that are affected by the changes are taken as representations of the changes. The ADM relates these change representations to the documentation. This way, impact of the software changes on the documentation is detected over the ADM relations. The functionality and usage as well as the power and limitations of Change Impact Detection

performed on the ADM is demonstrated.

The ADM DOCUMENTATION PROCESS demonstrates the integration of the Associative Documentation Model and the Change Impact Detection application into an standard documentation process that corresponds with software development processes. A scenario shows guidelines to maintain dynamically documentation and to get consistent documentation that corresponds with the state of the software development process.

Acknowledgement

The diploma work is a big effort. It takes a lot of engagement to have good ideas, to collect material, to do experiments, to develop models and scenarios and to prove them. All this can not be done if the environment is not right.

THANKS TO ALL !

- I thank SERGE DEMEYER for his work as my supervisor. He gave me great input and analyzed the work. He supported me with his experience with Famix models as representations of object-oriented software. He opened insights into the re-engineering and the metrical analysis of object-oriented software. Most important were the inner software structures that are important and useful for the relationship creation between software and documentation.
- A special thank goes to PROF. DR. OSCAR NIERSTRASZ and the SOFTWARE COMPOSITION GROUP at the Institute of Computer Science and Applied Mathematics of the University of Berne. Their logistic and mental environment is great for interesting input and good ideas.
- The FISCHER MEDIA CONSULTING in Lucerne that gave me the opportunity to make practical experiences and earn money. They were my economic base for my studies.
- The crew and the guests of the MELACHERE in Stans had always a beer in reserve to close the day's hard work. Their political analyses are very interesting and delivers lots of material to talk about and introduce Tobias Roethlisberger into the details of the politics of Nidwalden.
- MARKUS CHRISTEN and GIALO WUETHRICH gave me lots of opportunities to test my Computer knowledge in real world. Their PCs were subject of intensive fights with the Microsoft Windows operating system.
- Finally i thank MUM and DAD who supported me during my studies. I could always rely on the profound background of MY FAMILY. They were the stable basis of my development, my studies and my work.

Contents

List of Figures

Chapter 1

Introduction

> ”Adde parvum parvo magnus acervus erit. [Add little to little and there will be a big pile.]”
>
> *Ovid*

I just made the experience of writing documentation once more. It would be a dream having a system that generates this documentation automatically. Whenever you develop some software, you just generate documentation and all your problems are solved. Whenever you have documentation and do not know where it belongs to, you let check it on potential software and get the correct answer. No incomplete or inconsistent documentation anymore. Whatever kind of documentation you like and whatever you want to know, it’s just one click away.

This is a dream, and it will stay a dream. We can’t solve the problem this way. Documentation is always some kind of brain work. It‘s a human product, affected with all the human imperfectness. So rather we must find ways to deal with incomplete, inconsistent, imperfect documentation. The work described here is a step in that direction.

1.1 Motivation

We‘ll never be able to build and update documentation fully automatically. Documentation is some creative process. It‘s a human effort. Information is too polyvalent and complex to handle completely by computation. Documentation goes beyond information retrievable directly from its subject, i.e. software source code. Motivation and development history of software for example are often essential to understand the characteristics of software. But this information cannot be retrieved directly out of code. They have to be delivered by developers themselves. It’s part of their job to be source and provider of that information.

Documentation work will never be performed fully automatic. But we can develop documentation structures and support strategies to ease documentation writing and maintenance. We can organize documentation more systematic and unified. We can automate some work steps. We can improve control over completeness and correctness of documentation.

Complete automation of documentation will not be possible because of limited possibilities to automate of information retrieval and the polyvalence of information

itself. But it's possible to represent retrieved information in a uniform manner and to relate to each other independently from whether they are obtained automatically or manually. A unified model allows to retrieve information and integrate into the model automatically as far as possible without excluding addition of manually obtained information. The model serves as a platform for various sources of documentation information.

If we succeed to build such a model and provide documentation support, it would ease and rationalize the documentation process. This would be a great step to ameliorate the documentation work. Improving documentation effort is already a huge motivation to investigate documentation more deeply.

1.2 Thesis

This diploma work focuses on technical documentation and its relationship to the software source code it documents. The maintainance of technical documentation a common problem within the software development process. It depends closely on the subject it describes. Any change of the subject may cause changes on documentation. Three questions are of special interest:

- What kind of data can be retrieved from source code ?
- How are these data represented ?
- How does it support the maintainance of technical documentation ?

We propose two categories of software source code data that are useful to support technical documentation. The first category are information about entities of the source code. The entities of the software source code represent the static elements of the software source code and provide therefore substantial data about it. The second category are the relationships between the entities. We investigate the three main relationships aggregation, inheritance and invocation/access. They represent some aspects of the interaction between the software source code entities.

A very important topic is the representation of software source code data. We represent entities by their names. The relationships are represented as a triple consisting of the two involved entity names and the type of the relationship. Name representation of entities is a solution to reflect the meaning of the entity. A triple of two involved entity names and the type of the relationship describes best the dependency between two entities.

The goal of the software source code data extraction and representation as described above is to support technical documentation about software source code. There is a narrow interdependency between software source code and technical documentation. We express this narrow interdependency in the ASSOCIATIVE DOCUMENTATION MODEL (ADM). ADM models dependencies between entities and documentation segments that relate semantically to each other. ADM takes source code relationships into consideration the way that they allow to derive from given dependencies to new dependencies that are given indirectly by the source code relationships.

ADM is the foundation for documentation support. It is capable to indicate potential documentation changes by analyzing source code changes. This is called CHANGE IMPACT DETECTION. Structural requirements of ADM to get best results for Change Impact Detection are of special interest. The following hypotheses formulate guidelines to get best Change Impact Detection results:

- **Hypothesis 1:** The more detailed software source code data are available, the more reliable Change Impact Prediction works.

- **Hypothesis 2:** The finer granularity of documentation segments is chosen, the more precise the location of Change Impact Prediction becomes.

- **Hypothesis 3:** The finer granularity of documentation segments is chosen, the less percentage of documentation is involved by changes and the more percentage of documentation can be excluded as unaffected by change.

Change Impact Detection, a documentation support strategy based on the Associative Documentation Model, proves usability and adequateness of the Associative Documentation Model for technical documentation about software source code support with special focus on synchronization of the documentation and the source code. Change Impact Detection determines software source code changes by comparing different versions of code. This is mainly a process of detecting source code entities, building sets of entity representations and detecting differences between the sets. The source code entity representations that reflect the meaning of the entities are related to all documentation segments that describe the respective source code entities. This relation, formulated as a set of dependencies, allows to indicate source code changes on the related documentation segments. Whenever a source code entity is affected by some change, all documentation segments related by dependencies to this source code entity may need to be updated. This documentation support technique depends on available code elements, documentation segments and dependency generation algorithm. Quality and reliability of this system is expressed by these hypotheses:

- **Hypothesis 4:** The ADM provides a model to manage to relationship between source code and documentation. Its structures are sufficient to perform Change Impact Prediction.

- **Hypothesis 5:** Algorithms that match the source code names on documentation segments of technical documentation to generate dependencies and have a good quality for Impact Prediction and serve as a basis for synchronisation of a technical documentation with an evolving software.

ADM and Change Impact Prediction are the foundation for documentation support. Support Strategies that integrate the advantages of ADM and implement Change Impact Prediction contribute to documentation systems that support developers at concurrent development of source code and documentation.

1.3 Goals

The goal of this work is to get a documentation model that expresses the special relationship between software source code and technical documentation. This model integrates software source code data extraction. It is the basis to perform Change Impact Prediction which allows the synchronisation of the software source code documentation with its related software source code. The model handles especially software source code documentation but should be extensible to fit other types of documentation as well. I focus on four central goals:

- I develop a documentation model that describes the special relationship between software source code and technical documentation. It provides all required structures to fit various kinds of documentation data sources into that model. That model shall benefit from the relationship between software source code and technical documentation. It is called the ASSOCIATIVE DOCUMENTATION MODEL (ADM). The model is adapted to the special requirements of technical documentation but should be extensible for further documentation types.
- I demonstrate the integration of data extraction into ADM. We automatically analyse software source code and extract software source code data. We use it for dependency detection. The influence of the relationships between software source code entities is of special interest.
- ADM serves as a basis for documentation support techniques. We demonstrate the usage of ADM for documentation synchronization with evolving software source code.
- Power and limits, advantages and disadvantages of ADM are investigated. We are interested in the limitation that are caused by our selection of the software source code data and the structure of ADM itself. We are interested in what this limitation means for synchronization of source code documentation with evolving source code.

1.4 Structure of this work

This diploma work is divided into five main parts. The order tries to introduce reader into the idea of associative documentation. The idea shall grow and result into a general, comprehensive and open proposal to handle technical documentation.

- **Chapter 2:** We start with an ANALYSIS OF TECHNICAL DOCUMENTATION and SOFTWARE SOURCE CODE MODELS. I investigate existing documentation techniques, types of documentation and known documentation support techniques. We focus on the characteristics of technical documentation. The second topic are models of object-oriented software. We investigate the Famix model under the aspect of the data that are available by this model.
- **Chapter 3:** Case studies demonstrates PROBLEMS explore the detection of dependencies between software source code and technical documentation scanning the documentation texts about the names of entity names. The entity names are obtained form Famix models that provide all entities of the software source code. We explore the usage of the dependencies for Change Impact Prediction. There we focus on criteria for handling source code names and documentation to get good results for Change Impact Prediction. This corresponds with the hypotheses 1,2 and 3.

  I categorize the types of problems that appear at building, maintenance and usage of documentation. Special focus is set on experiences of developers with documentation. The problem analysis lays foundation for the solution proposal.
- **Chapter 4:** We explore the issue of DATA EXTRACTION and DEPENDENCY DETECTION. Data extraction uses names to identify and represent software source code data. A entity name analysis within the technical documentation lets us

interconnect the technical documentation with the software source code demonstrates.

- **Chapter 5:** An Advancement of the relationship between software source code and technical documentation leads to a generalized model. The abstract model, called the ASSOCIATIVE DOCUMENTATION MODEL (ADM)is founded on this relationship but is principally designed to cover any kind of documentation. We describe the structures of that model and the requirements to make it comprehensive and extensible. We are interested in the strategies to build, maintain and use documentation based on that model in an global valid manner.

- **Chapter 6:** The ASSOCIATIVE DOCUMENTATION PROCESS an sample documentation process that bases on the ADM model. This process utilizes the data and especially the changes of different software versions to support the adaption of the technical documentation to the newest software version. Chapter 6 and 7 refer to the hypotheses 4 and 5.

- **Chapter 7:** The final section is about the CONCLUSIONS AND PERSPECTIVES of the ADM model. We discuss the results and show possible ongoing research.

1.5 Keep in mind !

Documentation is somehow a creative process. This limits the potential computational support. Human interaction will always be required. That affects any solution with some imperfectness. Any technique that integrates a human dependent moment cannot be a hundred percent reliable and predictable. But that‘s not the goal of this work. We try to support documentation as good as possible. But even if we gain just some automated documentation support and some guidelines that help organize documentation more efficient and reliable, we would win a lot. That‘s the goal of this work !

Chapter 2

Resources

> ”We all want to go to heaven, but nobody wants to do what it takes to get there.”
>
> *Calvin*

This thesis examines the interdependencies between software source code and technical documentation. Software source code and technical documentation are the core resources on which the interdependencies are analyzed. So we start with an analysis of this resources to know its characteristics.

We start with a consideration of documentation in general. The analysis of building, maintenance and usage of documentation manifests the nature of documentation development and shows a wide variety of potential problems with documentation. We examine what causes these problems.

It goes on with technical documentation. We are interested in the specific characteristics of this type of documentation and how it can be used in advantage for the interdependencies to software source code.

As the counter part of the resources to the technical documentation, we examine the software source code under the aspect of getting data that are useful to detect and analyse dependencies to technical documentation. We introduce the Famix model as a general model for object oriented software. Famix models provide data about the software source code independent from any concrete coding of the software source code.

2.1 Introduction to documentation

Documentation can be viewed under different perspectives. Each perspective delivers its own specific view of documentation with its own specific characteristics of documentation. A comprehensive overview over documentation is necessary to allows comprehensive documentation support techniques that cover as many requirements as possible.

2.1.1 Definition

Documentation is a medium for information about any subject. It makes information persistent. That means information is stored and can be retrieved at any time and place.

Storage and availability of information are the main purpose of documentation.

2.1.2 Characteristics

We distinguish five aspects that dedicate characteristics of documentation and which legitimate the existence of documentation.

- PRESENTATION OF INFORMATION. Documentation is provided for human beings. Thus, visualization of documentation represents the interface to the reader. Documentation content must be described in a human interpretable way. Reading of documentation is an interpretation process where human retrieve stored information. Structure of documentation categorizes information and lets retrieve desired kind of information without checking everything. Content directories provide cataloguing functionality. Visualization of documentation is also some psychological aspect. People like aesthetic well-designed documentation. It‘s easier to keep overview on a well shaped documentation. Last but not least, presentation of documentation is often associated with the quality of documentation.

- CONTENT. Documentation is a information carrier. Information provided by documentation requires to be structured. That eases to keep overview and to classify information. Documentation cannot contain every potential information. It covers some aspects of its subject that influence selection of information. Purpose and intention of documentation determine the kind of documentation content.

- ORGANIZATION. Documentation development is highly influenced by organization of documentation. Documentation organization includes fixing of authors, subject and aspects, technical form, development integration and maintenance. Lots of reasons for failing documentation base on logistic lacks. Constitution of documentation organization structures is a indispensable presumption for successful documentation development.

- PERSISTENCE. Documentation makes information persistent. Persistent means that information is stored at any time and retrieved any time later. Retaining of information is a fundamental presumption that allows research.

- ACCESSIBILITY. Accessibility is the counterpart of persistence. Persistence manages storage of information over time. Accessibility allows to retrieve stored information. It comprehends retrieval, location and time independence of documentation. Cataloguing systems are necessary to get knowledge about the existence of information. Location of information determines where to obtain information. Communication makes documentation place independent. Documentation can be obtained at any place from any place. Time independence is given by storage of information. Stored documentation can be obtained at any time. All features of accessibility are performed by combination of libraries and communication.

2.1.3 Types

Many different types of documents exist for many different purposes and audiences. We divide two categories of documentation. Process documentation documents the

software creation process. Product documentation provides information about the software that is the subject of the process.

Process documentation

Process documentation collects information about the process of software creation and organizes and describes the software engineering process.

- PLANS, ESTIMATES AND SCHEDULES help managers to calculate and control the software engineering process.
- STANDARDS determine guidelines of the process.
- REPORTS reflect the usage of the resources.
- WORKING PAPERS express the state of work. They reflect the ideas and decisions of the developers.
- MEMOS AND EMAILS serve for intensive communication between developers.

The structure depends on the audience. END-USERS are interested into the help the software provides for their work. They don't care about technical details. SYSTEM ADMINISTRATORS are responsible for managing the software and helping end-users to adapt it to their work.

Product documentation is quite stable and has a long life. Product documentation evolves in step with the software it describes. During evolution, product documentation is subject of many changes, but once the software is completed, it remains stable.

Product documentation

Product documentation describes the software product. It includes USER DOCUMENTATION which tells users how to use or handle and maintain the software.

- FUNCTIONAL DESCRIPTIONS describe the services provides by the software and serve for system evaluators.
- INSTALLATION DOCUMENTS tell system administrators how to install the system.
- INTRODUCTORY MANUALS help novice users getting started with the system.
- REFERENCE MANUALS report experienced users details of all system facilities.
- THE SYSTEM ADMINISTRATOR'S GUIDE tells the system administrators how to operate and maintain the system.

Product documentation also includes SYSTEM DOCUMENTATION. System documents describe design, implementation and testing of a system. System documentations should include

- The REQUIREMENTS DOCUMENT and an associated rationale.
- A description of the SYSTEM ARCHITECTURE.
- An architecture DESCRIPTION FOR EACH PROGRAM in the system.

- The SPECIFICATION AND DESIGN of each component.
- Good structured and good commented program SOURCE CODE LISTINGS.
- VALIDATION DOCUMENTS that report the validation of the system and compare it with the requirements.
- A SYSTEM MAINTAINANCE GUIDE that collects know problems and describes dependencies from hardware or additional software.

Process documentation organizes and follows the process of software engineering. Lots of the documents become outdated very fast. Plans get changed, progress reports and memos become obsolete.

2.1.4 Quality

Quality is a decisive criteria for any documentation. Quality includes various criteria.

- UP-TO-DATE. Documents must be Up-to-date. Information must be correct and reflect the current state.
- CORRECT. The document must be complete. All essential information that fit to the purpose of the documentation must be included.
- UNDERSTANDABLE. Documentation that is hard to understand doesn't help the reader or even won't be read at all.
- APPROPRIATE. A document must be appropriate to the goals of the document and the skill of the audience.

High documentation quality is important for software projects. Low quality documentation is useless or even damages projects. Incorrect information can be more dangerous that no information.

Structures

High quality documentation require structure and process guidelines. Specifications of the structure, standards and writing style improve the quality significantly. The Documentation structure defines the shape of the documents.

- It includes a COVER PAGE which identifies the project, the document, the author, the date, the type of document, configuration management and quality assurance information, the intended recipients and the confidentiality. It also provides an abstract or keyword for fast information retrieval and a copyright notice.
- Large documents provide CHAPTERS with subchapters and sections. Consistent numbering and an content page help to keep the overview of the structure.
- Documents with lots of details or referenced information need an INDEX. It support the accessibility of information.
- A GLOSSARY is recommended to explain the terms and acronyms if the document contains a vocabulary the readers may not understand.

Standards

Documentation standards act as a basis for document quality assurance. Appropriate standard lead to consistent appearance, structure and quality of documentation. The standards fall into three classes.

- PROCESS STANDARDS specify how to produce a documentation. It selects tools and defines a quality assurance procedure. It controls the evolution of a document. Figure 2.1 shows a sample process. Drafting, checking, updating and

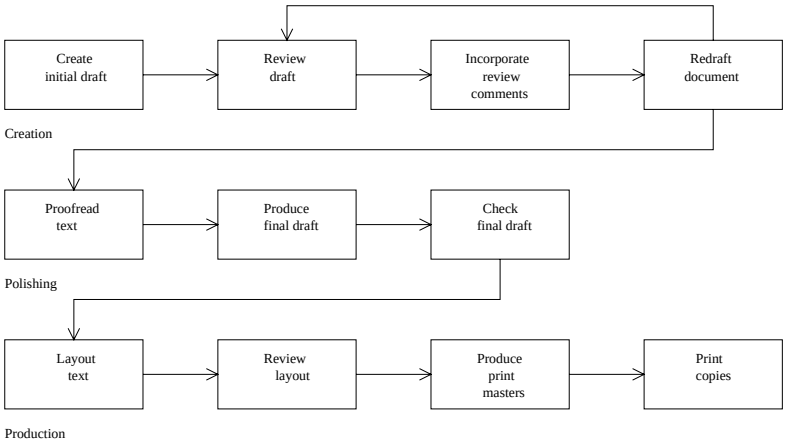

Figure 2.1: A sample documentation process

  re-drafting is an iterative process. It is repeated until documentation with sufficient quality is reached.

- PRODUCT STANDARDS set guidelines to all documents produces during software development. Uniformed structures help organize the documents. Various standards lead to a consistent documentation.
  - DOCUMENT IDENTIFICATION STANDARD defines unique identifiers for the documents. This is essential especially for large projects where thousands of documents exist.
  - DOCUMENT STRUCTURE STANDARDS organize consistent structures for same classes of documents. It includes conventions for titling, structuring, numbering, formatting.
  - DOCUMENT PRESENTATION STANDARDS set guidelines for the visual style of the documents. That includes logos, names, fonts, style.
  - DOCUMENT UPDATE STANDARDS define how to perform changes and how to mark it in a document.
- INTERCHANGE STANDARDS become very important as technical revolution allows fast and extensive electronic exchange of documents. The standards coordinate two aspects of documents interchange. They specify communication standards, for example the use of email or ftp as the document exchange standard. They also define document standards, for example TeX-sources and Postscript-documents. The latter ensures that the documents can be processed in an distributed environment.

Getting good documentation is a very complex process. It is neither easy nor cheap. Managers should pay as much attention to good documentation as to the software product itself.

Writing style

The writing style should be appropriate the content, goal and audience of the document. It depends on the abilities of the author to write good technical prose. Writing documentation is a repetitive process of writing, reading, criticizing and adaption the document. There are recommendations for good writing style.

- Keep paragraphs short
- Don't be verbose
- If a description is complex, repeat yourself
- Be precise and define the terms which you use
- Make use of headings and sub-headings
- Itemize facts wherever possible
- Do not refer to information only be reference number
- Use active rather than passive tenses
- Use grammatically correct constructs and correct spelling
- Do not use long sentences which present several different facts

2.1.5 Preparation

Documentation creation is an iterative process. Figure 2.1 show a documentation process that falls into three parts.

- DOCUMENT CREATION initializes the creation of a document. The document evolves in a iterative process of writing, reviewing and rewriting. At this point, mainly editor tools are required to support document creation.
- DOCUMENT POLISHING means checking of the structure, spelling and grammar. Checkers support this controls.
- DOCUMENT PRODUCTION conducts the visual presentation and the production of the document. Desktop publishing systems support this work.

Tools exist for all steps of the documentation process. Nevertheless the author or the knowledge of a good designer can't be replaced. So all these tools keep their character of support tools.

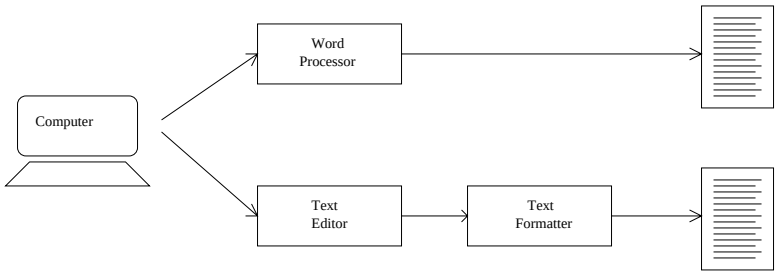

Figure 2.2: Two variants of document processing

Document processing

Documentation processing uses types of documentation processing tools: WORD PROCESSORS and TEXT EDITORS in combination with TEXT FORMATTERS. Figure 2.2 shows the two variants of document processing.

WORD PROCESSORS are in advantage to show immediately any advance of the document. It lets the author keep track of the document formatting during the evolution of the document. Knows word processors are contained in many commercial office packages like Microsoft Office (Microsoft word) or StarOffice.

TEXT EDITORS and TEXT FORMATTERS deliver higher quality and better structured documents. But the programming first has to be mastered by the author. The major disadvantage is that they do not provide immediate review of the results. A known text Formatting system is Latex that formats document sources into dvi-files that can be seen in any appropriate viewer, for example xdvi.

Spell checkers and Grammar checkers support polishing the documents. They ensure in conjunction with the author's review a correct language.

The final stage of Publishing the document begins when content and language of the document are completed. Publishing is more and more supported by desktop publishing and graphics systems, but this tools can't substitute a good document designer.

Document management

Document management systems become essential especially for larger software development projects. The enormous number of document requires coordination of document identifications. They keep track of all documents produced in the project. Documentation management systems can provide additional services like indexing which helps to retrieve documents. They can base on database systems, management tools and system files. The quality of a document management system depends on the discipline of the system users to follow appropriate procedures to collect and index all available documents of the project.

2.1.6 Languages

Storage of information requires to express information in a suitable form. Language allow to express information in a way that it can be retrieved. It determines form of documentation. Lots of different documentation forms exists: Plain text written in natural language, formal description for special purposes like software source code, tables,

graphics, images. Used language depends on used technical requirements, purpose and intention, and social environment.

- TECHNICAL REQUIREMENTS. Visualization of documentation requires a visual formatting schema. Thus, documentation language has to provide formatting capabilities. Formatting elements determine visual interpretation of a document. Visual Formatting can be explicitly or implicitly. Explicit Formatting provides visual formatting parameters like font types of font sizes. Implicit formating defines role of document elements within documents. Role of elements then implies visualization formatting.

  Another technical Requirement is relation documentation content to each other. That asks for language capabilities to express relations. Hypertext links are an example to realize references to other documents. Combined with easy technical accessibility they allow referencing networks to retrieve rapidly related information.

  Storage and accessibility of documentation influences technical requirements as well. Classic form of documents are printed documents. Its advantage is independence from technique equipment to access printed documents. But they are hard to be handled. Archives and libraries are required. Accessibility is very location dependent. Transferring printed documents is costly.

- PURPOSE AND INTENT. Purpose and intent of documentation influences character of the language. To explain the idea of the documentations subject you need some natural language that‘s capable to express the meaning. You need natural language to express the thoughts that are in behind.

  Compendiums and References need another approach. Listings of information collect information in a very schematic manner. Main focus is set on completeness of information. Listings serve as information archives, not as carrier for concepts or even ideas.

  Conceptual and architectural documentation uses graphs and schemas to express its information. Thats a non-textual approach to express documentation content. Graphs and schemas are highly interpretation dependent. They’re often used with textual additions to clarify interpretation.

  Coding Information are often documented ‘as is‘ in form of listings. Code documented this way is complete and correct reported. But it does not contain any information beyond code itself. Structural and conceptual information are at most regarded in a very poor fashion.

- SOCIAL ENVIRONMENT. Last but not least social environment highly influences nature of documentation language. If its important to make documents readable to a wide readership you have to choose a language the readership understands. Scientific documents for example are preferably written in English. Chosen language is also influenced by the author‘s capabilities. It’s easier for authors to write in their own native language.

  Social environment influences style of documentation. Style must be suitable to the readership‘s preferences and skill. The readership must be capable to understand the documents.

The different requirements and goals of documentation leads to variety of documentation languages and data formats. Three well-known and widespread examples are the Hypertext Markup Language, LaTex and UML.

- HTML. The Hypertext Markup Language (HTML) is a documentation description language and data format that is designed for the World Wide Web [Klu95]. The main features are a uniform text formatting, embedding of documentation resources like images, graphics, tables, and the hyperlink capability. Especially the hyperlink capability that allows referencing and direct accessing of related document over the network, make HTML to a very powerful and widespread documentation platform.
- TEX. TeX is a type setting system for documents[Knu91, Sch91, App94]. The documents are specified as descriptions. TeX files are compiled to get printable documents. Common formats of compiled Tex documents are dvi or ps.
- UML. The Unified Modelling Language (UML) is an example for an non-classical documentation system [Fow97]. UML is a description language for object-oriented models. Although the main goal of UML is modelling, a is role as a documentation language for object-oriented models is important.

Documentation languages can also be seen as data formats that are interpreted in a specific, for documentation relevant manner. Data formats for images, tables and models are relevant for documentation as well. Any format of documentation data can be seen as some kind of a documentation language.

2.1.7 Interpretation of content

Interpretation of documents is the way to get back information. Documents have all in common to store information in an interpretable manner. Possibility to retrieve of information is an elementary part of documentation. Documentations with content that cannot be interpreted are useless.

Information retrieval is performed by interpretation of descriptions. Used language plays the key-role on interpretation. The language determines the rules how to interpret descriptions and get its meaning.

It depends on used language of descriptions whether information can be interpreted automatically. If formal languages are used for descriptions automatic interpretation is possible. The formal language provides a well-defined syntax associated with a well-defined semantic. Otherwise interpretation works partially or even not automatic at all. In this case human interpretation is central. Human interpretation is much more flexible but not perfect as well. Human interpretation are often ambivalent. Let‘s reflect interpretation of description on three different examples.

```
# Show segments
  sub show {
    for(@segment) {
      $h = $_;
      if(!($h eq "")) { print $h,"\n"; };
    };
  }
```

This piece of Perl code prints some segment strings that is not empty. This code is written according the Perl language. Perl is a formal scripting language with a well-defined syntax and semantic. That piece of Perl code can be parsed using the grammar of the Perl language and interpreted according Perl semantics. Interpretation is unambiguous.

```
<html>
<title>Output of counters</title>
<head><h2>Functionality of the segment print routine</h2></head>
<body bgcolor="#c0c0c0">
  <p>The segment print routine prints all segment strings
     that are not empty.
</body>
</html>
```

This HTML-code describes functionality of the segment print routine. HTML (Hypertext Markup Language) is a language to describe formated text. Tags, used as a pair of start-tag and end-tag, mark parts of text. The type of the tag determines the function of the embedded part of text. 'Functionality of the counter', surrounded by the h2-tag, means that 'Functionality of the counter' is a header of category 2. Viewers visualize it according this description.

Documentation coded like this HTML-code is partially parseable and interpretable. Text-parts can be determined and their function within documentation can be interpreted according the HTML structure. Meaning of text parts itselves is coded in natural language written as plaintext and cannot be interpreted automatically.

```
The segment print routine prints all segment strings
that are not empty.
```

This plaintext description contains information about the segment print routine as well but uses natural language. There is no formal grammar nor is there an unambiguous semantic. It cannot be interpreted automatically in an unambiguous manner. Its human work to get its meaning.

2.1.8 Purpose of documentation

Persistence of documentation is the core purpose of documentation. Furthermore, documentation serves for communication of information. Communicating means that information contained in documentations gets available for others. Anyone who has access to the documentation can obtain its information.

Documentation allows to exchange thoughts and ideas. Research results are published and exchanged using papers being documentations of research work. Documentation provide the framework for discussions. That brings people and their ideas together and merges their work contributing to all‘s advantage.

2.1.9 Importance in software engineering

Documentation plays a central role within software lifecycle. Field of application of documentation is many-sided. Any aspect at any state of software can and should be documented. Specifications at the beginning of a software project are documented as well as tutorials for users or technical descriptions for system administrators or programmers.

HANDLING OF DOCUMENTATION is delicate. Problems occur especially when building and maintaining documentation. Memorizing information in a suitable form, keeping documentation complete and correct or controlling availability of documentation are crucial points of documentation. Completeness and correctness are decisive for reliability of documentation. Documentation whose information is not reliable is worthless. Questions about completeness and correctness of documentation affects writers as well as readers. Writers have to check it permanently to guarantee quality of their documentation. Readers check it to evaluate reliability of documentation.

MAINTENANCE OF COMPLETENESS AND CORRECTNESS is crucial. It is normally checked by the writers knowledge about the topics. The writer compares contents of documentation with the described subject. He decides what information are included in documentation and whether it fulfils the object of documentation in a adequate manner. Such a strategy is fully human dependent. Completeness and correctness are ensured for as much as writer of documentation works solid.

To DISBURDEN THE WRITER and to make documentation's completeness and correctness less human dependent automated and human independent techniques are required. Investigation of characteristics of documentation lays fundamentals for documentation support techniques.

Documentation makes information of software projects persistent and available. This makes documentation to an important tool for STORAGE OF INFORMATION AND COMMUNICATION. Software engineering is also a process of information flow. Whenever the work of a software engineer depends from others work - what is the normal case - information is required for work coordination.

The overwhelming importance of documentation makes documentation to a project critic part of any software development. That means that failing documentation can cause failing projects. Refer to [Som92] for more details.

2.1.10 Questions and Problems

Documentation is a manifold source of problems. Building, maintenance and usage of documentation have various hidden dangers that cause problems. Moreover, documentation problems affect the whole software engineering process. Therefore, problems with documentation have to be considered as very serious.

Technical Problems

Problems can be caused by technical problems. That are the problems that most probably can be managed by technical provisions. Technical problems are the field where tool support could help.

- LACKING TOOL SUPPORT. Tools support eases building and maintenance of documentation. Lacking tool support causes lower quality of documentation. Documentation work increases. Tendency not writing documentation raises simultaneously to required effort to write documentation.

- LIBRARIES AND ARCHIVES. Larger documentation must be organized in the libraries or archives. Insufficient libraries and archives raise danger of loosing parts of documentation. Moreover, documentation is hard to be retrieved. Maintenance and controlling of documentation state is almost impossible.

- STORAGE AND RETRIEVABILITY. Storage of documentation must be organized. Otherwise documents may be lost. Lacking storage complicates retrievability and decreases use of documentation. Documentations have to be collected and organized. Only structured documentation guarantees that documented information can be retrieved. A not accessible documentation is the same as a non existent documentation. Accessibility it not given only by the existence of documentation. The knowledge about its existence must be efficiently accessible as well.

- COMMUNICATION. Documentation serves for information exchange. It should fit into communication infrastructure. Documentation that is hard to be exchanged is of limited use.

Logistical problems

Logistic problems affects the organization of the documentation lifecycle. That kind of problems requires logistic measurement to be solved. We need more an organization concept than some tool support.

- INFORMATION RETRIEVAL. Information is obtained during the software creation process. Software must be documented whenever it appears. If documentation is not seen as an integral part of the software evolution process but something that is done separately, it increases the effort to retain information. It's a great danger that nobody is aware of that most of information is gained during software evolution process and is lost if they are not documented. Documentation written afterwards run into trouble by lacking important information.

- COORDINATION AND RESPONSIBILITIES. Coordination of documentation is important specially on larger projects. Lots of developers may work on the same project. Their work must be coordinated, responsibilities ensure documentation. Projects with uncoordinated documentation lead to multiple documents while other parts of documentation are not written at all.

- CONTROLLING. Controlling of documentation covers completeness and correctness of documentation. Documents on projects in process are always in danger of being out of date. They have to be updated. Documentation information must be completed and corrected. Uncontrolled documents loose rapidly reliance and become worthless.

Management

Problems by management are somehow the most difficult documentation problems to be fixed. They do not require some technical or logistic requirement but some insights of the management. That‘s the border that leads developers to the limits.

- BAD SOFTWARE ENGINEERING PROCESS. Bad software engineering and management causes bad documentation. That kind of documentation problems has to be solved on the software engineering level. If software engineering problems are not solved, documentation problems are not central as the whole software project fails anyway.

- ECONOMIC PRESSURE. Economic pressure is another major reason for failing documentation. Documentation is always suffering under a lack of time. Projects under time pressure concentrate on development of the core running system. That means mainly developing the software and integrate it in a running environment. Commercial Projects are mostly product oriented. The have a clear specified goal, the development of a concrete product. Aspects not concentrating on this, mainly the ones behind the narrow borders of the project are not seen as essential. Documentation is not paid and software seems to work without documentation as well. This is very short sight but normal as these projects are dominated by short sighted economic guidelines.

- LACKING INSIGHTS. Last but not least, the need of documentation is just not seen by managers and developers. Specially on smaller projects, everything seems to be under control. The knowledge in the developers brain is seen as a sufficient documentation. Developers are not aware that documentation may be important for others. Under this condition, projects fail whenever there is a staff mutation, or projects grow so this documentation 'technique' does not work anymore.

Problems with documentation are detected by checking all potential problem sources. According the nature of detected problems, technical, logistic or management retaliatory action must be taken.

2.1.11 Documentation Support and Research

There are lots of existing techniques and running projects on documentation. This projects provide support for building and maintenance of documentation.

Features

Research ameliorates the features of documentations. We distinguish three topics of research to ameliorate capabilities of documentation.

- VISUALIZATION. Visualization of documentation shall be improved. More sophisticated elements are developed. The freedom in forming documentation is improved. The shape of documentation shall be adaptable to any specific requirement.

- DYNAMIC ELEMENTS. Todays documentation techniques break static structures. Animations for example allow to illustrate a dynamic situation better that static graphics.

- INTERACTION. Interaction is a great target of todays documentation. Information shall not be presented in every case but just on demand. Interaction also allows readers to contribute to the documentation.

Support

Research develops better definition capabilities for documentation that integrate the targets of visualization, dynamic elements and interaction. To keep documentation manageable, research focuses on tool support and integration.

- TOOL SUPPORT. Tool support includes editors for document creation and management tools to sustain documentation organization. Editors ease document creation. Developers do not care about crucial coding details. Hypertext editors for example allow to create hypertext documents without knowing anything about HTML. Management tools ease handling of large collections of documents. Sitebuilder software for example provides functionality to maintain and control large hypertext sites.
- INTEGRATION. Not every form of documentation can fit to any documentation requirement. This insight leads to the idea to use for each piece of documentation a suitable documentation form. The documentation is then a merger of these pieces. Such a documentation is polymorph. Structures are required to manage such a polymorph documentation. Hypertext documents are examples of polymorph documents. Texts, images, animations, sounds are stored as separate resources. A central hypertext document links them together. Its the responsibility of the viewer tool to visualize such a polymorph hypertext document as one homogeneous document.

2.2 Technical documentation

Technical documentation is one type of documentation. Technical documentation distinguishes from other documentation by its subject and its goals. Technical documentation handles technical descriptions of the structure and the technical functionality of the software. It contains technical data about the software and is often a reference for technical data about the software.

We focus on three central aspects of the introduction that are central: the language and interpretation of documentation and its embedding within a software and documentation development process.

2.2.1 Characteristics

Technical documentation provides general documentation a documentation structure that divides the documentation in several topics. It consists in its form of text parts, tables, graphics. Nevertheless there are two essential points that are special in technical documentation.

- Technical documentation has a NARROW STRUCTURAL RELATIONSHIP TO THE SOFTWARE. The structure of the technical documentation given by its chapters often corresponds with the structure within the software. This could be either an architectural or functional structure of the software.
- Technical documentation contains lots of data from the software source code. This could be names of classes, methods, attributes, functions, variables. This is essential as it lets identify relationships from parts of the technical documentation to entities of the software source code that are mentioned there.

2.2.2 Existing Systems and Research

Existing Documentation Support Systems and Ongoing Research play their role on two fields: Embedded within Software Engineering Environments and Managing/Formatting documentations.

Documentation within Software Engineering Environments

Documentation within Software Engineering Environments appears as an integral part. Documentation capabilities focus on descriptions of source code, models, specifications. Preferred documentation types are listings, graphics, figures.

RATIONAL ROSE is an example of an Software Engineering Tool that includes documentation capabilities[Qua98, Cor98]. It's an OMT/Booch/UML analysis and design tool with C++/Java/Ada code generation. A special feature of Rational Rose is it's capability to construct Models out of given source code.

Rational Rose's documentation capabilities include the models, algorithms as far as they are specified by developers and the source code. Documentation is reduced to the raw technical descriptions and listings. There is no additional documentation that illustrates the idea and the features of the software.

Documentation Formatting and Management

The second area of Documentation tools and ongoing research is the presentation and managements of documentation. Its topics are visualization, structuring and accessibility of documentation. It focuses on documentation development environments and tools. ROADS, MCF and HotDoc are examples of projects on this field.

- ROADS is a software to manage webbased Subject Gateways[Bra99b]. It allows Informations managers to maintain searchable Internet based catalogues of information. Roads can be configured for specific needs. Gateway administration and user interfaces can be adapted to specific purposes. Roads supports different types of network based resource identification, indexing and cataloguing.

- META CONTENT FRAMEWORK (MCF) is another software to provide information about information[Bra99a]. MCF works by attaching properties to objects. Documentation Resources are regarded as objects. They can obtain information about itself contained within properties. I.e. A webpage can be seen as an object and it's size as a property.

- HOTDOC is a project to create a framework for compound documents performed by the University of Darmstadt[Buc97, Buc99]. Compound documents aggregate polymorph resources into a documentation. I.e. Texts, vektorgraphics, images, tables, formulas etc. shall be integrate into the same document. HotDoc implements editor suites instead of heavy monolitic editors. Each resource type has its own editor. Developers can adapt editor for their specify needs.

  HotDoc handles documentation as an encapsulated matter. It's topic is storage of information in suitable forms and combine them to documentation. HotDoc develops techniques for storage and retrieving heterogeneous information. HotDoc does not care about where and how information are retrieved.

Very few projects deal with the relation between source code and documentation. Lots of approaches exist for information retrieval based on code. Lot of effort is made to develop suitable structures of documentation to store information in suitable forms into that documentation. But the relation of information obtained out of code and information stored within documentation isn't a important criteria there.

2.3 Software source code

Software source code exists in many different shapes. It distinguishes in its architecture, language and implementation. We know logical, functional, procedural or object-oriented languages. They have different formal languages with different syntax. They differ in the way they are implemented and the provided development environment.

This thesis focuses on object-oriented software. We introduce the Famix model to get an general model of the object-oriented software that is as independent as possible from language or implementation specific issues.

2.3.1 The Famix model

The Famix project was started to investigate reengineering methods to transform object oriented legacy code into frameworks [SDS98, TD98]. Current research focuses on three research areas: Metrics and Heuristics (to detect design problems and measure design improvements), Grouping (to form software modules and target architectures) and Reorganization Operations (to perform actual program translation).

Tool prototypes were developed for conducting various experiments within those research areas. Source code used for case studies is written in a variety of different object oriented languages like C++, Smalltalk, Ada, Java. To avoid necessity of adapting all tools on different source code languages, the Famix project provides a common information exchange format with language specific extensions. The common information exchange format provides an language independent description of object oriented software without exclusion of language specific options.

The Famix information interchange format fulfils different requirements concerning the data model and issues of the representation:

- The exchange format is extensible. Language specific options can be added by language specific entities and properties.
- Sufficient basis for metrics, heuristics, grouping and reengineering operations. The minimal requirements are specified by minimal requirements of experiments performed by investigating these operations.
- Readily distillable from source code. The upper limit of information provided by the exchange model is given by all information that are commonly parseable out of source code.
- Easy to generate by available parsing technology. The exchange model is performed by well known parsing technologies.
- Simple to process. Common used parsing technologies allow simple transformation into internal data structures. The exchange format can be handled by existing tools like Perl or python.
- Convenient for querying. Much of the reengineering work is querying the exchange model. The exchange model allows transformation into input-stream for querying tools.
- Human readable. For human inspection and debugging purposes the exchange format is human readable. Naming of entities and attributes supports easy retrieval of semantics.

- Allows combination with information from other sources. External information can be merged with the exchange format. That is used for where external resources are included.
- Supports industry standards. Usage of industrial standards eases implementation of tools within industrial context.

Famix offers a basic model for object oriented legacy code that fulfils these requirements (Figure 2.3). The core model is a meta-model that provides the Object-Property relationship. Objects are entities, associations and arguments. The Famix model is a generalization of all object oriented models. It provides the common basic structure of all object oriented models. Object oriented models define instances of entities, associations and arguments. Instances of properties allow to extend the model for specific needs.

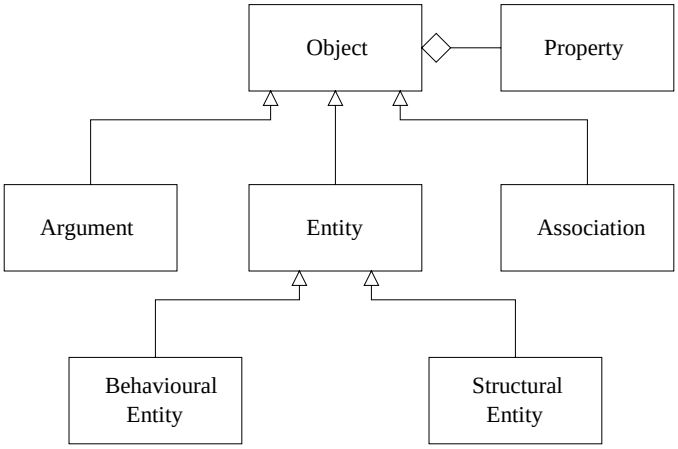

Figure 2.3: The Famix Model

Many object oriented models have similar entities and associations. The Famix core model (Figure 2.4) is a standardized model that includes the most common entities and associations and their relational structure. It is an instance of the core meta-model as shown in Figure 2.3. The entity instantiations Class, Method and Attribute are the most common entities that exist in almost any object oriented model. The association InheritanceDefinition models the inheritance relation between classes, a core feature of any object oriented model. The association Invocation models method calls. The association Access represents the access on variables.

2.3.2 Data provided by Famix

Famix is an model of object-oriented software that includes data about the entities of the software and its relationships expressed by associations. Famix does not provide data about the dynamic behaviour like the functionality of the Software. I.e. it does not provide data about algorithms or protocols implemented by the software.

We are interested into two types of data that are provided by the Famix models: The entities and the associations of the software. We extract and identify the entities by their names. The names are one of different properties, but they name and represent the entities. As entities are mentioned within descriptions like technical documentation

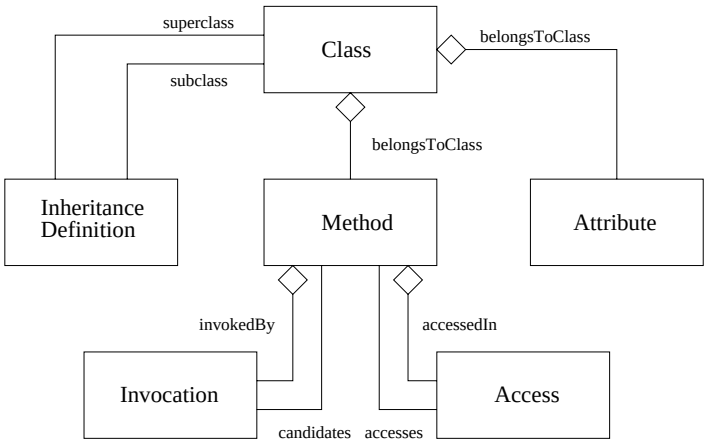

Figure 2.4: The Famix Core Model

by their names, the names of the entities are very probable to appear within technical documentation.

Associations represent the relationship between entities. They express how entities depend from each other and how they potentially influence each other. This is essential to infer characteristics and behaviour between the entities. Associations do not provide names but relate two entity names and tell us the aspect under which they relate to each other.

Chapter 3

Case studies

> ”Documentation is the castor oil of programming. Managers know it must be good because the programmers hate it so much.”
>
> *Unknown*

3.1 Introduction

We investigate the interdependencies between software source code and technical documentation. The goal is a documentation system that links the development of source code and source code documentation together to profit in the documentation development from the data that are retrieved from source code. The investigation of this approach to documentation shows three aspects that lead to the core of such a system.

- DATA about software is contained within the source code. Differences of source code allow insights into the development process of the software. The question is how to retrieve and represent data about the source code and its development process.
- DOCUMENTATION provides information by its definition. The information has a so manifold diversity of shapes that makes it impossible to specify it with a definite syntactic definition. The problem is how to identify and manage it.
- LINKING source code and documentation together means linking its information. How are such links defined and how can they be computed and retrieved ?

3.1.1 Name matching

The answer for these questions is the essential part for a source code documentation system. We start with two case studies to investigate how to deal with those questions. It proposes to detect and represent information by the names of source code entities. The names represent the meaning of the entities. The entities represent a central part of the source code, so the names represent a central aspect of the information about the source code. The names of entities are convenient to link with documentation. The following schema explains the concrete steps.

- SOURCE CODE INFORMATION are the names of Entities, Associations and Properties of the source code. The names serve as representations, they reflect the meaning of Entities, Associations and Properties.
- DOCUMENTATION INFORMATION are identified by Keywords. The names of source code information serve as keywords. The keywords reflect the essential information of documentation.
- LINKING OF SOURCE CODE AND DOCUMENTATION is specified by the names and the keywords respectively. As the set of names and the set of keywords are equivalent, links are generated over the keywords. Source code and documentation that are represented by the same name and keyword are linked together.

This approach is based on object-oriented models that represent the source code. It assumes that essential information about the software are contained within the models and that this information are found within documentation about the software again.

3.1.2 Goals

This proposal is subject of case studies. The case studies investigate the interdependencies between source software source code and documentation source code. The case studies have five goals:

- We prove the reliability of dependencies specified by matching of entity names on technical documentation. Dependencies are reliable when they express real semantic interdependencies between source code and documentation. The case studies shall explore if the chosen method of name matching leads to such reliable dependencies.
- We infer source code changes on technical documentation using the dependencies and detect change requirements on the documentation that are caused by the source code changes. The case studies examine if dependencies are convenient for Change Impact Prediction.
- The case studies shall examine how different schemas for entity naming and documentation segmentation influence the structure and the reliability of dependencies and its capability to serve for change impact prediction.
- The case studies shall demonstrate if dependency detection by name matching is technically reasonable.
- Power and limits of name matching for dependency detection shall be observed. It shows under which preconditions name matching delivers its best results.

The results of the case studies build the basis for further exploration of the dependencies and an model that expresses the relationship between software source code and documentation.

3.2 The HotDraw case study

The HotDraw case study explores interdependencies of source code and documentation on the basis of MANUALLY CONSTRUCTED DEPENDENCIES. It shows how to build up

manual dependencies and what specific problems may occur. We study the implications that can be derived from source code changes on documentation and what is their quality if they base on manually determined dependencies.

3.2.1 The HotDraw graphics framework

HotDraw [Bra95] is a two-dimensional graphics framework for structured drawing editors that is written in Smalltalk [Joh92]. It can be used in many different applications from a simple painting program to a visual drawing inspector.

A HotDraw application edits drawings that are made up of figures. Figures are graphics elements such as lines, ellipses, and text, and they can represent other objects. HotDraw provides a set of interactive graphical tools to manipulate figures. Figures include a set of handles. Manipulating a handle changes some property of its figure or performs some action. Figures can also have constraints. Constraints can make sure that two figures are connected, are the same size, or have the same colour. Constraints are important since they make figures more compositional. Also, they make it easier to define new kinds of figures by composing and placing constraints between simpler existing figures.

3.2.2 Points of interest

HotDraw is of interest as an evolving software evolution process. As all resources like different source code versions and documentation versions are available, it is possible to follow the development process of the source code and the parallel evolution of the documentation. Thats important to have a comparison reference to validate the investigations of documentation change prediction with the real development behaviour.

The HotDraw case study tries to predict change requirements on the basis of the analysis of different source code versions. It compares the predictions with real changes within the different documentation versions and indicates if automatically performed prediction meets the real documentation change requirements. Two topics are of special interest:

- DATA EXTRACTION. Source code serves as only data resource. We try to get data automatically by extraction of entities, associations and attributes. The names represent their meaning. We investigate how to get these names. We also see if representation of meaning by names is appropriate.
- MANUAL DEPENDENCIES. Manual dependencies are fully user defined. That means that the developers have to associate documentation with according source code. We investigate how to perform this and how to implement it. We are interested into the advantages and disadvantages of manual dependencies.

3.2.3 Available resources

HotDraw is an open software. The resources, especially source codes and documentation are freely available. This case study works with the source code versions 2.5 and 3.0. The code is written in Smalltalk, so object-oriented models are given implicitly and can be extracted by parsers.

The HotDraw case study uses the documentation of HotDraw version 2.5. It's a technical description that focuses on the architecture and the features of the software. The content of documentation meets narrowly the content of the software source code.

case study

3.2.4 Famix model data

HotDraw is written according an object-oriented model. The models of version 2.5 and version 3.0 are the basis for data extraction from source code. The models are expressed as standard Famix models. A parser detects the required model information and generates a Famix model.

We use Famix models because they are capable to represent object-oriented models in an uniform manner. They include all standard features of object-oriented-models. The more they are extensible so special features could be included as they would be required [TD98].

The HotDraw case study generates a Famix model for both Smalltalk source code versions 2.5 and 3.0. The models are generated as level 2 models. That means they include include information about classes methods attributes and inheritance definitions.

The information retrieval of the HotDraw reduces the information to the names of all classes, methods and attributes. Properties of this entities are not included. Table 3.1 shows potential information available by source code. Not all potential information of the Famix models is required. The column REASON explains why this kind of information is included or not by the HotDraw case study.

Information	Usage	Reason
Classes, Methods Attributes	The names are used by the HotDraw case study	They represent the entities, the names reflect the meaning of the entities
InheritanceDefinition	Used for implication	Reduction to the basics Reserved for more sophisticated models
Invocation, Access	Used for implication	Reduction to the basics Provided by Famix model of level 3
Properties	Not used	Reduction to the basics Reserved for more sophisticated models
Algorithms	Not used	Not provided in the Famix models

Table 3.1: Usage of available information from Famix models

Properties and InheritanceDefinitions are provided by Famix models of level 2. They are not considered within the HotDraw case study with the intention to reduce it to the basics, namely Classes, Methods and Attributes. Properties and InheritanceDefinitions are reserved for more sophisticated models.

Invocation and Access are not provided by level 2 models. To get this information, Famix level 3 models are required. Invocation and Access serve for more sophisticated models as well.

Famix Models don't provide algorithmic information that exceed the Invocation- and Access-Structures. So, they are not available for any source code information analysis that is based on Famix models.

3.2.5 Representation by names

Classes, Methods and Attributes of Famix models are used for information. Information about these entities are provided by their names. The names reflect the meaning of these entities. Table 3.2 shows some samples of entities, their names and their meaning.

Type	long name	short name
Class	PolylineFigure	PolylineFigure
	Representation of a polyline figure	
Method	PolylineFigure.recalculateBoundingBox()	recalculateBoundingBox
	Recalculates the BoundingBox of the polyline figure	
Attribute	PolylineFigure.fillcolor	fillcolor
	Tells about the fillcolor of the Polyline	

Table 3.2: Usage of available information from Famix models

These samples rise two questions: How to format the names and what happens with entities whose names are not chosen according their meaning ? Different formatting schemas for entity names are possible. We consider two different schemas: A short naming schema and a long naming schema. Short names use only the name of the entity itself ignoring any names of its containing class. They describe directly the meaning of the entity but do not provide any information about its context. Long names use the name of the entity itself together with the names of its container. Additionally to the meaning of the entity itself, long names set the meaning into a context. The Attribute name POLYLINE.FILLCOLOR i.e. tells us not only that we are regarding a fillcolor, but that we are talking about the fillcolor of a Polyline. Long names provide more detailed information about the entity but increase the complexity of the information as well.

3.2.6 Documentation segmentation

The documentation segments must be associated with the documentation. Association means that the names that represent source code entities are associated with the documentation parts that belong to these entities. That means partitioning of the documentation into a set of segments. We use the documentation structure of the HotDraw documentation as the partitioning principle: The documentation is divided into segments according the chapter, subchapter and section structure of the documentation. Any subchapter represents a documentation segment. The HotDraw documentation is divided into 115 segments. It includes all text parts of the documentation, but no additional table, graph and image.

We observe that such a segmentation of the documentation leads to segments that normally include just one major topic. It results from fact that documentation contents are structured according the documentation structure. This documentation behaviour eases isolation of the topics and is essential for associating documentation segments with according source code elements.

3.2.7 Manual Dependency

A manual dependency is defined by the developer. He decides which names are related to which documentation segments. The criteria for dependency generation is the

criteria is the equivalence of the meaning. A name is related to a segments if the segments describes some meaning that belongs to the source code that is represented by the name. The decision about the equivalence of the meaning is done by the developer.

```
Source Code Name:  DrawingController
Source Code Type:  Class

Segment Reference: 21
Related Segment:   DrawingController The DrawingController, like most
                   controllers in Smalltalk, handles input from the
                   user. This input can be either mouse or keyboard
                   based. The controller handles input from the operate
                   (middle) or window (right) mouse buttons the same,
                   but it handles input from the select (left) mouse
                   button and the keyboard differently depending on
                   which tool is selected. For example, a click and
                   drag operation results in different actions being
                   performed. ....
```

The Name DRAWINGCONTROLLER represents the class DrawingController. The name reflects the meaning and functionality of the class. Drawing Controller is related the the documentation segment 21. Segments 21 describes the functionality of the DrawingController. Thought, it is concerning the meaning of the class controller. The equal meaning of both the name and the segment indicates a relation. A dependency expresses such a relation.

```
Source Code Name:  Tool.drawing()
Source Code Type:  Method

Segment Reference: 50
Related Segment:   Tools are responsible for handling much of the user
                   input. All keyboard and left mouse button input are
                   handled by the current tool using its processKeyboard
                   and press methods. These methods translate the user
                   input into actions in the drawing such as creating
                   a new figure or moving figures.
```

The Name TOOL.DRAWING() represents the Method Drawing of the Object Tool. The segment 50 describes the functionality of the the feature Tools and is therefore concerning the Method Tool.drawing(). So the must be related what is expressed as an dependency.

The decision that the names and segments concern each other is in both cases decided by the developer. His decision indicates to create a dependency. That means the dependencies are created manually.

The segments of the HotDraw documentation normally include only one topic. That eases the work of dependency generation. As a documentation segment includes one major topic, the developer has only to look for all entities whose names contribute to the feature described in the segment.

```
Segment Reference: 50
Segment:           Tools are responsible for handling much of the user
                   input. All keyboard and left mouse button input are
                   handled by the current tool using its processKeyboard
                   and press methods. These methods translate the user
                   input into actions in the drawing such as creating
                   a new figure or moving figures.
Related names:     Tool
                   Tool.processKeyboard()
                   Tool.Tool.press()
```

Dependencies are generated between the three selected names and the segment. The name selection is fully performed by the developer. It is his decision to select which entities may have influence on the topic described in the segment.

3.2.8 The results of the HotDraw case study

The HotDraw case study delivers insights into the representation of software by Famix models and entity names. It demonstrates how to build manual dependencies and shows what influences the reliability of such dependencies.

- The HowDraw case study shows that detection of dependencies by name matching is practicable. We get Famix models by analysing the source code and retrieve the names of the entities by parsing the models. Name matching to detect the dependency is a String manipulation operation that can be performed by i.e. Perl.

- The representation of the software by entity names covers major aspects of the software and can be extended by additional information from Famix models. But it is limited to the data provides by the Famix models.

- The reliability of a dependency is defined related to whether meaning of the name and the segments associated by the dependency is the same or not. If it's equal, then the reliability is high. If the dependency does not meet the equality of meaning, the reliability is low. The reliability of the dependencies is decisive for any application that builds on the dependencies.

  The HotDraw case study works with manually specified dependencies. It shows that the reliability of manually generated dependencies depends on the developer. If he determines all dependencies as correct and complete as possible, the reliability is very high. It's a great advantage of manual dependencies that developers have direct influence on the dependency quality. If they do a good job, they get very reliable dependencies.

- Manual dependencies have one disadvantage. It takes a lot of work and time to create them. For each documentation segment, the developer has to look for every entity that may contribute to the meaning of the documentation segment. Tool support is required to support the developer at this point.

- Validation of dependencies means to check if a dependency expresses a real semantic interdependency between an entity and an documentation segments. A dependency is valid if it fulfils this requirement. It is a human task to decide if a dependency is valid. Once the dependencies are validated, the reliability of the

dependencies is given by the amount of valid dependencies. The more dependencies are valid, the more reliable they are. Manual dependency specification as performed in the HotDraw case study lead to reliable dependencies because it ensures valid dependencies when it is performed correctly.

3.3 The Visual Works case study

The Visual Works case study is an experiment to detect source code changes and their impact on documentation. The application is based on a technique called change impact detection. Change impact detection explores two versions of source codes and detects differences. These differences, called changes, are then related to parts of documentation called documentation segments. This allows to infer potentially affected documentation segments, thus change impact detection is capable to predict potential change requirements on documentation.

3.3.1 The Association Concept

Change Impact Detection is an application that is based on the Associative Documentation Model. The idea is to compute differences between different code versions and to detect implications on documentation using dependencies. I.e. an entity Controller that has been removed is regarded to imply changes on all segments of documentation that relate to Controller. Relationship to controller is defined by dependencies from Controller to any documentation segment that refers to Controller.

3.3.2 Basic Steps

Change Impact Detection is separated into 5 basic steps with encapsulated functionality. These steps can be substituted by equivalent functionality. Equivalent means that external behaviour remains. I.e. code element generation based on names can be substituted by another, more sophisticated method including additional information about code elements. Substituting functionality would just have to include additional information into names an generate a list of code elements represented by those names. Change impact detection is divided into the following steps.

- Code element generation.
  Code element generation delivers lists of code elements. Code elements are extracted out of uniform code models and described by names. Names represent meaning of code.

- Change computation.
  Change computation detects differences between two sets of code element names. The result is a set of code element names that have either been added or removed.

- Dependency generation.
  Dependency generation detects occurrences of code element names within documentation segments. It is performed by matching code element names on documentation segments.

- Impact detection.
  Impact detection determines potential impact of code changes on documentation.

It is based on change detection results and dependency generation. Code changes are implied according to dependencies on segments. Impact detection results in indicating which code changes may have impact on which documentation segments.

- Impact evaluation.
  Impact evaluation interprets results of impact detection. For each documentation segment the number of software elements having impact on respective documentation segment is counted. Numbers will be the basis of quality analysis.

3.3.3 Computation

The core part of VisualWorks case study is performed based on code element names matching with documentation segments. Each code element possesses its own, unique name. That name is coded as a string and expresses meaning of the software element. I.e. a class Controller is a code element whose functionality is being a controller or at least part of software's controller feature.

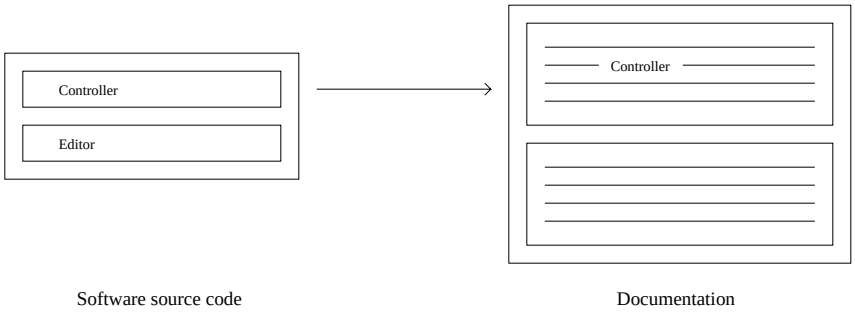

Figure 3.1: Dependency definition by name

The documentation is scanned for presence of named code elements. Whenever a name of a code element is found, the respective documentation segment is associated with the respective code element as a dependency. Figure 3.1 shows an object Controller that is associated by dependency with a documentation segment where the name CONTROLLER appears. Dependencies are the basis for impact detection. Impact detection first detects changes on code. Using the dependencies changes, we infer potential changes on documentation.

3.3.4 Complexity

If we are only interested in change impact detection, but not on all dependencies generated by matching code element names on documentation segments, then generation of dependencies may be reduced. We then need to generate only dependencies to the names of code elements that are affected by change. All the names involved by change are selected and dependencies are only generated for that selection. The result is a more efficient algorithm.

The algorithm works basically with quadratic computational expense of O(m*n) where m represents the number of entities and n the number of segments. Using large sets of named code element or large documentation it is a time consuming process. Reduction of dependency generation on the names affected by change increases efficiency

of dependency generation significantly. Computational expense is linear to the amount of change.

3.3.5 Implementation

The Visual Works case study implements Change Impact Detection in Perl. Perl is a text manipulation language. Algorithms for extraction of entity- and property names as source code information, change detection, dependency generation, impact detection and results evaluation are basically text operations, therefore Perl is best suited as an implementation environment. Perl's advantages are optimization for text operations and easy adaptability of scripts for specific purposes.

3.3.6 Analysis

Analysis of Change Impact Detection delivers results about change requirements on documentation. According the goal to investigate distribution and location of impact, most interest is in amount of impact on documentation segment. Amount of impact on documentation segments can be determined by counting for each documentation segment the impact. That numbers are a measure for how much respective documentation segment is affected by change. Counting of numbers of impact can be performed based on Perl scripts that detect change impact.

3.3.7 Experiment

The Visual Works environment offers an ideal example to experiment with Change Impact Detection. Visual Works provides all essential resources. Both different source code and documentation versions are available. It's a real case study that means it's not prepared specially for this experiment but used "as is". It offers insights into changes between different versions of Visual Works source code and the changes between the documentation versions caused by the source code changes.

3.3.8 Available Material

The Visual Works case study provided following resources necessary for the experiment.

- Available Code Resources.
  Visual Works, Version 2.0, Smalltalk Source Code
  Visual Works, Version 2.5, Smalltalk Source Code
- Documentation Resources.
  Visual Works Cookbook for Version 2.0
  Visual Works Cookbook for Version 2.5

Both source code and Documentation are available electronically, which eases automatic manipulation. Code has to be prepared to identify requested information and to adapt and unify it for further steps. Smalltalk source code of Visual Work is parsed, interpreted, and Famix Models of level 3 are generated out of retrieved information. Level 3 Famix Models contain information about Classes, Methods, InheritanceDefinition, Attributes, MethodInvocation and VariableAccess. Famix models allow to extract software elements. Documentation is converted into a standardized text version. The

Visual Works case study considers only documentation text. Documentation structure is conserved. That includes titles of chapters and subchapters, which is required for automated segmentation of documentation.

3.3.9 Preparation

Source code of Visual Works is prepared as well as Documentation resources to adapt it to requirements.

- CODE ELEMENT EXTRACTION. Code element extraction delivers a list of code elements. Visual Works case study uses Classes, Methods, InheritanceDefinitions, Attributes, MethodInvocations and VariableAccesses as code elements. Code Elements are represented by their names. Code elements generated according their naming are unique. Two different code elements won't lead to the same names.

  Two sets of named code elements are generated for each Visual Works version 2.0 and version 2.5. One set contains only names of code elements representing classes, methods, InheritanceDefinitions and attributes. The selection is performed according information level provided by Famix models of level 2. The other set contains classes, methods, InheritanceDefinitions, attributes, MethodInvocation, VariableAccess what corresponds with information level of Famix models of level 3. The set based on Famix model of level 2 is a coarse model, the set based on Famix model of level 3 is a fine model. The coarse model is a subset of the fine model. Table 3.3 presents a numerical overview. These sets of named code elements are representations for their respective software version.

- DOCUMENTATION PREPARATION. Documentation is prepared as a set of segments. Separation into segments is performed using chapter structure. This is reasonable as chapter structure of documentation tends to describe more or less encapsulated topics. Each chapter and subchapter normally describes a separated, well specified and identifiable topic.

3.3.10 Change Analysis

Detected change within code performs basis to determine change impact on documentation. The amount of change indicates potential amount of change requirements on documentation.

Software element type	*V20*	V25	Removed	Added	Changed
Classes	734	718	47	31	78
Methods	7781	7951	190	360	550
Attributes	1168	1191	33	56	89
InheritanceDefinitions	716	696	48	28	76
MethodInvocation	6006	6131	396	521	917
VariableAccess	3727	3811	328	412	740

Table 3.3: Number of changed code elements

The Visual Works case study shows an average of 3% to 8% of change. Distribution of change is similar among different types of code elements.

3.3.11 Impact on Documentation

Impact of code change on documentation segments is based on dependencies that relate the code change elements to the documentation segments. Dependencies that do not involve change elements do not influence Change Impact Detection. Thus only dependencies relating to change elements have to be generated to perform Change Impact Detection.

Change Impact Detection generates data that describe a relation between the change elements and the documentation segments. There are different possibilities to represent the data. One possibility is to list all data. No information is lost. But it is a very large list of data and meaningful information is not evident.

Meaningful information is derived from the raw data. Tables 3.4 - 3.7 show the data under the aspect of the amount of change impact on the documentation segments. Refer also to the complete tables in Appendix A.1 - A.4. Impact of Code Change on documentation segments is measured by counting numbers of code changes relating to a particular documentation segment. Focus is set on the numbers of detected change impact counted for each segment.

Chapter	*Subchapter*											
10	265	26	9	9	10	184	9	9	9	-	-	-
11	92	20	12	32	9	19	-	-	-	-	-	-
12	62	14	9	9	11	19	-	-	-	-	-	-
13	1116	213	48	58	203	184	194	43	43	43	44	43
14	57	17	9	13	9	9	-	-	-	-	-	-
15	145	11	11	11	12	12	11	11	48	18	-	-
16	65	27	11	9	9	9	-	-	-	-	-	-

Table 3.4: Impact distribution based on fine code granularity

Chapter	*Subchapter*											
10	54.2	5	2	2	2	38	2	2	2	-	-	-
11	18.8	4	2	7	2	4	-	-	-	-	-	-
12	8.8	3	2	2	2	4	-	-	-	-	-	-
13	228.2	44	10	12	42	38	40	9	9	9	9	9
14	11.7	3	2	3	2	2	-	-	-	-	-	-
15	29.7	2	2	2	2	2	2	2	10	4	-	-
16	13.3	6	2	2	2	2	-	-	-	-	-	-

Table 3.5: Part of total impact with fine granularity expressed in per mille

Tables 3.4 - 3.7 list in extracts the results of Change Impact Detection on the Visual Works case study. Refer to the Appendix A for the complete tables. The tables list the chapter vertically and the appropriate subsections horizontally. For each subchapter the according number shows the amount of code change elements that have impact on the particular subchapter. Tables 3.4 - 3.7 show the results for the following cases:

- The tables 3.4 and 3.6 show the numbers of code change elements that may have impact on the documentation segment.

Chapter	*Subchapter*											
10	84	18	9	9	10	11	9	9	9	-	-	-
11	66	15	12	16	9	14	-	-	-	-	-	-
12	60	14	9	9	9	19	-	-	-	-	-	-
13	129	19	9	14	14	11	-	-	-	-	-	-
14	57	17	9	13	9	9	-	-	-	-	-	-
15	84	9	9	9	10	10	9	9	9	10	-	-
16	47	9	11	9	9	9	-	-	-	-	-	-

Table 3.6: Impact distribution based on coarse code granularity

Chapter	*Subchapter*											
10	32.3	7	3	3	4	4	3	3	3	-	-	-
11	25.3	6	5	6	3	5	-	-	-	-	-	-
12	23.0	5	3	3	3	7	-	-	-	-	-	-
13	49.5	7	3	5	3	3	-	-	-	-	-	-
14	21.9	7	3	5	3	3	-	-	-	-	-	-
15	32.3	3	3	3	4	4	3	3	3	4	-	-
16	18.0	3	4	3	3	3	-	-	-	-	-	-

Table 3.7: Part of total impact with coarse granularity expressed in per mille.

- The tables 3.5 and 3.7 transform the numbers of the tables 3.4 and 3.7 to fraction. Metric units are per mille. Transformation allows to compare results of the tables 3.5 and 3.7.

The tables 3.4 and 3.5 as well as the tables 3.6 and 3.7 represents two cases:

- The tables 3.4 and 3.5 show the results of Change Impact Detection that is performed on software source code data with fine granularity. Fine granularity means that classes, methods, attributes, InheritanceDefinitions, MethodInvocations and VariableAccesses are utilized as source code elements.
- The tables 3.6 and 3.7 show the results of Change Impact Detection that is performed on source code with coarse granularity. Coarse granularity means that classes, methods, attributes and InheritanceDefinitions are used as software source code data.

The results correspond with the **Hypothesis 1** that claims better Change Impact Prediction when using more detailed software source code data.

3.3.12 Situations

The analysis of Change Impact Detection data starts with the analysis of the situations that appear in the relation between source code and documentation. The situations let us understand what causes the numbers of Change Impact Detection and how they are to be interpreted. Two situations are essential for interpretation of Change Impact Detection data:

- Lots of change relates on a segment.
  The documentation segment contains lots of change elements. Many different source code changes influence that documentation segment.

- A change element has influence on lots of segments.
  The change element appears widely distributed in the documentation. It influences many different documentation segments, also in different contexts.

Data of tables 3.4 - 3.7 cover the requirements for analysing the first situation. For each documentation segment identified by the subchapters of the documentation, the number of the change elements that may have influence on the segment are listed. The higher the numbers, the more change elements relate to the specific documentation segment and may have impact on that segment. Two reasons may cause large numbers:

- The names of lots of change elements are mentioned within the segment. There's a big density at mentioning of change elements.

- It's a large documentation segment with a lot of documentation text. That increases the probability of code elements names to be mentioned what potentially increases the amount of dependencies as well. The large amount of dependencies includes potentially a large amount of dependencies that relate from change elements to the segment. Thus the probability of large impact numbers increase simultaneously to the size of documentation segments.

Tables 3.4 - 3.7 show all segments having a minimum of a few change elements counted that may have impact on the segment. Two reasons may cause this behaviour:

- There's a change element that is mentioned almost everywhere within documentation. It causes potential change impact on almost every documentation segment.

- The name of a change element is indifferent and appears at many places within documentation but without any semantic relation to that change element. I.e. the name of a class IS is represented by the name IS. Dependency generation associates class to all documentation segments where the word IS appears. It's almost everywhere the case and dependency is generated to almost every documentation segment although there is no semantic relation to the class IS.

3.3.13 Results and Interpretation

The Change Impact Detection data of the VisualWorks case study let us infer essential results about convenience of the dependencies for Change Impact Detection and the influence of structural facts concerning the naming schema and the documentation segmentation schema on Change Impact Detection.

Change requirements on the documentation

Situation analysis shows that two different situations cause large numbers within Change Impact Detection data. Interpretation of the large numbers has to distinguish these two cases. The first case where lots of change elements mentioned in a segment cause the large number refers to documentation segments with a huge density of names of change elements. The huge number of mentioned change element names means a high

probability of change requirement on the documentation segment. That meats the interpretation of the large number generated for this segments by the Change Impact Detection as a sign for change requirement.

The second case shows that another reason for large numbers are large documentation segments. In this case a large number does not mean automatically a high probability for change requirement on the documentation segment. If the large number is caused only by the large size of the documentation segment, probability of change requirements is not higher as for smaller documentation segments with smaller numbers. In contradiction to the first case, the second case does not allow to derive form the large number on a high probability for change requirements. This is critical as the large number itself does not indicate whether a conclusion on a high probability of change requirements is valid or not. It requires a distinction of the two cases to know whether a large numbers indicates a big probability for change requirement or not. The size of the documentation segment is an indicator to distinguish. The number of characters, words or lines can serve as a suitable definition of the documentation segments size. They are easy to retrieve.

Numbers of Change Impact Detection under consideration of the documentation segment let us make statements about the probability of change requirement on documentation segments. Large numbers on small segments indicate highly probably change requirements on the documentation segment. Large Numbers on large segments do not indicate necessarily change requirements. Partitioning into smaller segments and recomputation of the Change Impact Detection Data for that segments clarifies the probability of change requirement on that segment. This corresponds with **Hypotheses 2 and 3** that claims small documentation segments being better for Change Impact Prediction than large ones.

Influence of structural facts

The Visual Works case study perform the Change Impact Detection on two different representations of source code. Tables 3.4 and 3.5 show the numbers for a fine code granularity. Tables 3.6 and 3.7 reflect the results performed on coarse code granularity. The differences indicate the influence of the different granularity structure of the source code. Results performed on fine code granularity indicate more clearly the subchapters with large numbers of change elements that have impact on the subchapter. While the results with coarse code granularity are more diffuse, the results with fine code granularity indicate chapters and subchapters with a high probability more clear. I.e. Table 3.4 shows a high probability for change requirement for the whole chapter 13 while this is not obvious in table 3.6.

3.3.14 Observations

The observation of the Visual Works case study shows potential power and limitations that influence source code documentation techniques.

- SOURCE CODE INFORMATION RETRIEVAL
  - LIMITED ON AVAILABLE RESOURCES.
    Retrievable information is basically limited to available resources. Code impact detection uses source code for change analysis. Though available source code is an indispensable precondition.

  - STATIC DATA BY PARSING.
    Source data retrieval is strictly limited to data available from source code. Moreover, Change Impact Prediction focuses on data that is obtained by parsing. Change Impact Detection uses static source code data. There is no analysis of dynamic code behaviour.
  - LIMITED BY COMPUTATION.
    Although source code data is limited to static data, analysis of source code is a time consuming process. Source code parsing and detection of data of interest has linear complexity. That means used time is linear to source code size.
  - DYNAMIC DATA IS VERY HARD TO RETRIEVE.
    Basically data about dynamic behaviour can be retrieved by simulation of all potential dynamic behaviour of the software. Such an behavioural analysis is very computational complex. In practice, either the number of cases to be simulated or the number of states of the software must be limited.

- HANDLING DATA AS CLOSE SUBJECTS

  - DIVISION OF DATA.
    Change Impact Detection uses object oriented entities like classes, methods, attributes as data resource. Entities are handled as data atoms. Data about an entity is provided as a whole. There‘s no subdivision of data. Subordinated data about an entity, i.e. data about attributes contained in a class, are represented as its own independent data atoms.
  - EXPRESS COMPLEX DATA.
    Data about source code is designed as a set of atomic data. Complex data may depend on other data and get their meaning from relation of other data. Complex data is resolved into a set of atomic data.
  - NAMES ASSUMED TO EXPRESS MEANING OF ENTITIES.
    Names of source code entities are used as data about the entities themselves. That assumes that entity names express the meaning, role and behaviour of the entity. That works with well-written source code that fits to this requirement but is a problem with code that uses bad naming schemas.

- CHANGE DETECTION

  - COMPARISON OF ENTITY DATA.
    Code Impact Detection is based on Code Change. Code change computation uses comparison operations to check data entities for equality. Code Impact Detection handles entity data as Strings. Equality of two entity data is given if the two String representations are equal. This tends to overkill change detection. It may be more appropriate to substitute absolute equality by some kind of similarity.
  - COMPUTATION COMPLEXITY.
    Code Change detection has a high computation complexity. Basically each entity data of one source code version has to be compared with each entity data contained in the other version and vice versa. This leads to a computation complexity determined by the product of the sizes of version one and two. Although Code Change Detection ameliorates using sorting and binary search techniques it remains a huge computation effort.

- RELATING SOURCE CODE TO DOCUMENTATION
  - RELATION BUILDING IS A VERY SPECIFIC ISSUE.
    Change Impact Detection relates Code Changes to documentation segments. The visual works case study explores occurrences of source code data within documentation segments. It detects occurrences of of Strings representations of source code data that are affected by code change within the textual representation. Whenever it matches, tested data is related to the matching documentation.

    This approach assumes that documentation segments that describe some source code mention source code elements like classes, methods and attributes by their names. That works with technical documentation that refer closely to subjects contained in source code. Nevertheless, three situations must be distinguished:
    - Code change is related to a documentation segment and the segment‘s content is affected by this Code Change. That‘s the best case. That means that detected change impact meats a real change requirement.
    - Code Change is related to a documentation segment but the segment‘s content is not affected by this Code Change. In this case potential change impact is detected correctly but change is not required actually. Most common reason for this is that the documentation segment describes some aspects of changed code elements but changes do not have any effect on this aspects. This case is not critical as it just means some additional checking work for developers what will never be too bad.
    - Code Change is not related to a specific documentation segment but this segment is affected by this code change. That‘s the most critical case as it means that there can be some documentation segments where no change impact is detected although there would be a change requirement. Occurrence probability of this case is decisive for quality of any Change Impact Detection technique.
  - RELATION BUILDING RULE IS CENTRAL.
    Relation building rule determines relation structure between source code and documentation. It’s a decisive element for the quality of Change impact prediction. Relation building rules have to be chosen the way the generated relations meet real relation given by documents content as closely as possible.
  - CHANGE DETECTION AND RELATION BUILDING DETERMINES IMPACT PREDICTION QUALITY.
    Quality of Change impact prediction depends on quality of Change Detection and the relation building technique.

- PRACTICAL EXPERIENCES
  - COMPUTATION AND STORAGE RESOURCES.
    Computation and storage become a problem when data increase. Visual Works i.e. has quite a large source code. It causes quite a large number of entity- and property names. This sets of names must be handled and stored which requires lots of storage resources.

Moreover, computation of code change becomes a problem. Detecting code change is mainly a set operation that detects all elements being either in the first or the second set but not in both at a time. This operation gets very computation intensive.

Problems with storage and computation resources may be fixed by reduction of source code data. But it is very critical as available source code data influence the quality of Change Impact Detection.

3.4 Comparing manually and automatically detected dependencies

The Hotdraw case study and the Visual Works case study used two different ways to specify or detect dependencies. We compare the advantages and disadvantages of these two methods and compare with the results we got with both variants.

- EFFICIENCY. Dependency generation. Manual dependency specification forces the Developer to check all names representing entities manually about existences within all documentation segments and to specify an dependency if it matches. This process must be repeated whenever the source code has changed. Automatic dependency detection as performed in the VisualWorks case study is much more efficient. The dependencies detection works automatically as matching of the entity names on the documentation segments is a text scan process that can be performed automatically.

- RELIABILITY Reliability of manually specified dependencies is high when the specification is performed very properly. The reliability only depends only from the quality of the dependency specification performed by the developer. The automatic dependency detection process relies on the principle that the names of the entities occur in the documentation segments that refer semantically to the respective entity. The reliability of the dependencies is high when the documentation fulfils this principle. Automatically detected dependencies are more limited to technical documentation with an narrow relationship to the software source code than manually specified dependencies.

3.5 Lessons learned

The case studies introduce into four aspects that must be explored more deeply.:

- SOURCE CODE DATA EXTRACTION.
  Are there alternative methods to extract software source code data ? How far does it depend on given resources ? What is its influence on the quality of the detection of dependencies ?

- DEPENDENCY DETECTION.
  What are alternative methods to build relations ? How do they depend on source code data and on documentation ? What's the influence of relation building methods on quality of Change Impact Detection ?

- QUALITY AND RELIABILITY.
  Quality and reliability of Change Impact Detection is decisive. We know that more detailed source code data used to detect dependencies to small documentation segments deliver good quality and reliability for Change Impact Detection on technical documentation. A question is where the limits are for this rule.

- COMPREHENSIVE MODEL.
  The Visual Works case study performs a specific Change Impact Detection. Can we build a general, comprehensive model that fits to Change Impact Detection but is more independent from source code data retrieval and relation building algorithms ?

Source Code data retrieval and dependency building are explored in chapter 4 where the detection, maintainance and structure of dependencies is explored in details. It includes examinations of the quality and reliability of dependencies. The results lead to the Associative Documentation Model that is handled in chapter 5. The model formulates the the general dependency relationship between software source code and technical documentation. It is determined by the results to get reliable dependencies and serves as a basis for change impact prediction, a technique to predict change requirements on documentation based on the changes in the software source code.

Chapter 4

Source code data and dependencies

> ”I know what you’re saying, Bart. When I was young, I wanted an electric football machine more than anything else in the world, and my parents bought it for me, and it was the happiest day of my life. Well, goodnight.”
>
> *Homer Simpson*

Dependencies are the central means for associating source code data with technical documentation. Two elements influence the characteristics of dependencies: the source code data and the documentation segment as the foundation of the dependencies and the relationship expressed by the dependencies.

We consider now the characteristics of dependencies. It starts with an investigation of the extraction of source code entity names. The objectives are what influences it, what kind of data are available and how they are obtained. We focus on the representations of the source code entity names and especially its uniqueness.

The source code entity names are the basis for dependency detection. We investigate an extended dependency detection concept that includes Famix associations and documentation structures as additional information. We are interested in the characteristics and advantages of such dependencies and how to evaluate them.

4.1 Extraction of source code model data

The extraction of source code data is the foundation for dependency detection. The set of source code data determine the type and the amount of the dependencies that can be obtained.

4.1.1 Requirements for source code data extraction

Logistical, technical and conceptual requirements influence the extraction of source code data. They refer especially to the preconditions and environment of dependency detection. These requirements are one aspect that determine what kind of source code data could be obtained.

- LOGISTICAL REQUIREMENTS.
  - The AVAILABLE RESOURCES determine the maximal limits of any source code data extraction. The source code i.e. is indispensable to obtain any information out of it. It's a critical point specially in practice where source code may not be available.
  - TECHNICAL EQUIPMENT like computers and networks allow automatic source code data extraction. Software to extract and analyse source code data require require powerful computer infrastructure especially to handle extensive amounts of source code data.
  - PEOPLE expect extraction of source code data to be easy to use. They require handy tools that let retrace and control the results. It should ease the developer's work. Acceptance by users and developers is the key to integrate source code data extraction and dependency detection into development and documentation environments.
  - TIME. Source code data extraction must be efficient. Results are required at reasonable time and with reasonable effort.
- TECHNICAL REQUIREMENTS.
  - ACCESSIBILITY AND COMPUTABILITY. Electronically available resources ease access and analysis. It's a precondition for any automatic source code data extraction. Resources not available electronically must be converted into a suitable electronic form.
  - FORM AND STRUCTURE. Electronic resources like source code have many different forms and structures. Not every form or structure is appropriate for source code extraction. Resources with defined syntactic structures are easier to analyse and interpret. Danger of misinterpretation is much higher when no well defined syntax and pertinent semantic is present.
  - REPRESENTATION. The results of source code data extraction are source code data. Source code data must be represented in a suitable manner. That means source code data must be converted into representations and representations into source code data in an unambiguous manner.
- CONCEPTUAL REQUIREMENTS.
  - AUTOMATION. Source code data extraction tools allow automation. Automated tools are based on parsing and interpretation technologies. This limits automatic source code data extraction. Good tools should be easy to adapt to specific environments
  - ADAPTABILITY. Resources with different forms and structures require adaption of source code data extraction results. The concept of source code data extraction must be flexible enough to fit to different resources.
  - EXTENSIBILITY. Source code data extraction can be extended for additional information. That's essential when special purpose information is required.
  - INTEGRATABLE. Retrieved source code data build the basis for further analysis. Many applications like Change Impact Detection integrate information retrieval and use its results as information basis. Integratablity of any information retrieval concept is a essential design target.

4.1.2 Definitions and Terminology

The extraction of source code model data is based on SOURCE CODE ENTITY NAMES. A source code entity is an entity as described in the Famix model. It contains a property name. This name represent the entity.

4.1.3 Famix model extraction

The software resources like the software source code are analysed to extract all data that are available and retrievable and that are relevant for the source code. In the HotDraw and VisualWorks case studies of chapter 3, the Smalltalk source code is parsed and all entities and association within the source code are extracted. Based on these data, the Famix models are generated and stored according the CDIF specification to describe Famix models.

Any resource can be used. The essential precondition is the availability of data about entities and associations of the software. Java software i.e. can be analysed without knowing the source code. Java lets access and analyse the entities through special classes that provide access to the entities. Another possibility is reengineering Java byte code to determine the structure of the Java software.

4.1.4 The data extraction

The data extraction consists of analysis of source code and the extraction of the different desired source code data. It divides into three steps.

- The PREPARATION OF THE SOURCE CODE starts with the collection of all required source codes. The structure and language are analysed as preparation of the model extraction.
- The MODEL EXTRACTION delivers a model of the source code that contains all required data and structures for the source code data extraction.
- The DATA EXTRACTION retrieves the source code data and converts them into String representations so they can be used for dependency detection.

We extract Famix models from software source codes. This is performed by parsing the source code and converting the extracted information into Famix models. The source code data are then retrieved from the Famix Model and represented by their names as Strings. Basically, the source code data could be obtained directly from the source code. The extraction of the Famix model is inserted to separate the analysis of the source code and the source code data extraction and representation. This way we have to adapt the extraction of Famix model to all types of source code with all their different structures and languages, but not the source code data extraction and representation that only refers to the Famix model.

4.1.5 Famix data extraction

We consider the source code data extraction using Famix models. Famix models are introduced in chapter 2. They provide a model that describes the entities of a software. Associations express the relationship between the entities. We consider the extraction of Famix models, examine what kind of data are provided by Famix models and show

how to obtain and represent these data. The relationships between the entities influence the dependencies. This aspect is described later in chapter 5 under the aspect of the inner relationships of the software source code entities.

4.1.6 Entities of Famix models

The Famix models provide a standard set of entities. It includes the most common entity and association types as they occur within object oriented software models.

CLASS, METHOD and ATTRIBUTE are the most common entities within Famix models. They are represented by their name which is a mandatory property.

INHERITANCEDEFINITION, METHODINVOCATION and VARIABLEACCESS are the most common associations of the Famix model. They specify a relationship between Famix entities and provide essential information about the structure of a Famix model.

The six entity types belong to the Core Famix Model. The selection corresponds with standard Famix Models that provide standardized, most used entity information.

4.1.7 Naming schema

Representation of detected entities must be easy to handle, represent the entity uniquely and reflect the meaning of the entity. Famix Information Retrieval uses the names of the entities provided in the Famix models. Names are easily available and fully automatically retrievable.

Famix provides a naming schema that reclines on the naming schema of the Unified Modelling Language (UML) [JR99]. We distinguish names and unique names. The names denominate entities and associations. They do it independently from any context. Unique names aggregate the name together with the context. Inclusion of the context makes such names unique.

Names are independently from its uniqueness represented as Strings. Basically lots of different forms of String representations are possible. Nevertheless some common notation is mostly used. The following examples use such a notation:

```
Drawing
ToolBuilder.addTool()
ConnectionCommand.Point
```

The common notation makes clear that DRAWING is a class, TOOLBUILDER.ADDTOOL() is a method and CONNECTIONCOMMAND.POINT is an Attribute. Common notation says that classes are represented by their names, methods by the name of their containing class and their own name, separated with a point and terminated by double brackets that may contain additional parameters, and attributes by the name of their containing class and their own name that are separated with a point.

The names normally reflect the meaning of the entity. This is very important if an application like an association with documentation builds on this fact. This fact is not enforced by Famix Models. It depends on naming schemas chosen by programmers. It makes it a critic point as we can't know if the developers chose the names according the meaning of the entities and associations. Applications that build on the names as representation have to consider this fact. They must find ways to deal with this uncertainty.

4.1.8 Uniqueness of Names as Famix Data Representations

The names of entities serve as representations of the meaning of the entities. The meaning of an entity is given by its role within the software. Structure and behaviour of entities determine its meaning.

Representation by names opens a trade-off between representing an entity as uniquely as possible and meeting the meaning of an entity as narrow as possible. Representation by the whole unique identifier identifies entities uniquely, but they do not get the meaning in an appropriate manner. Appropriate means that the string representations have a similar shape like the names that appear when the meaning of an entity is mentioned within an textual description.

```
Controller.initialize()
ControlManagerclass.initialize()
```

The methods initialize are represented uniquely by unique identifiers. It gets its uniqueness by appending of context information. The method names are extended by the name of its containing class CONTROLLER or CONTROLMANAGERCLASS respectively. Other extensions would be scopes, parameter lists or return types.

```
initialize
initialize
```

The same methods but reduced on their names loose their uniqueness. The names meets correctly its meaning as initializing something. But there are no structural informations that would indicate where there methods belong to.

Reduction of the string representation reduces its information content and the probability of uniqueness. In contrary, it's more probable that a short string representation meets the entity representation within a textual description. Its more likely that the method 'initialize' is mentioned than the method CONTROL.INITIALIZE(). The more a short string representation would also meet CONTROL.INITIALIZE().

On the other side lost uniqueness complicates to associate a detected entity representation to the correct entity. Representation itself does not identify unambiguously an entity. There are two ways to deal with it: either to include additional environment information to make it unique of to deal with it by relating to all entities may be represented by the short representation. ADM deals with the second solution. Ambiguity is included into consideration of impact prediction quality. As long as ambiguity does not influence reliability of change impact prediction too significantly it is acceptable for this purpose.

4.1.9 Example for short name uniqueness

The Visual Works case study delivers an example for representation by short names and its influence on uniqueness of the representations.

Table B.2 shows multiple elements represented by the same short representation. Case n describes a case where n different elements are represented by the same short representation string. The number of cases encounters the number of short representations that represent the particular number of different elements. Percentage of cases reflects part of the cases from the point of view of representations. Percentage of elements shows part of the cases from the point of view of the entities.

Case	*Occurrence*	Percentage of cases	Percentage of elements
1	4791	79.41	55.81
2	775	12.85	18.05
3	221	3.66	7.72
4	111	1.84	5.17
5	53	0.88	3.09
6	21	0.35	1.47
7	19	0.31	1.55
8	8	0.13	0.75
9	6	0.10	0.63
10	4	0.07	0.47
11-20	20	0.34	3.26
21-101	4	0.08	2.03

Table 4.1: Ambiguity of short name representations

A considerable part of the entities are represented uniquely even with short names. This limits influence on ambiguity significantly. Influence of remaining ambiguity is subject of consideration about reliability of Change Impact Prediction.

4.1.10 Limitations

Retrieving of entities is basically limited to available resources. Famix data extraction has its limits to available Famix Models. Limitation work under two aspects: first, Famix models must be available. That‘s primarily a problem of Famix Model detection and depends on specific cases. For Smalltalk source code i.e. parsers are available and getting Famix Models is no problem.

The second aspect is the limitation given by Famix Models themselves. There are limitations by definition like dynamic behaviour of programs what is out of scope of Famix Models and therefore not includes within the model. Practical limitation are given as Famix Models provide various levels of information. Depending on the level, more or less details about the program are available. Famix Information retrieval limits on six types of entities:

4.1.11 Advantages and Disadvantages

Advantages and Disadvantages depend on available resources, applied extraction techniques and the purpose and uses of retrieved information. Change Impact Prediction works on Famix Models as resource bases. It extracts entities and represents them by their names. These names are detected within textual descriptions or documentations. This approach shows some remarkable advantages:

- AUTOMATION. All steps are fully automatic. It reduces requirement for human interaction: it allows extensive tool support and leads to a remarkable reduction of work.
- STANDARDIZED. Famix models act as basic resource for the whole information retrieval on a standardized platform. Any resource that can be converted into Famix models can serve as an information resource. It also make tools independent form specific resource formats.

- COMPLETENESS RELIABILITY. Within some information categories, information retrieval is complete. I.e. Retrievable information categories like class names, method names or attribute names are generated completely and reliably. Not every category of information is generated, but the one that can be generated are generated and reliable. That makes it calculable.

There are some disadvantages of this method as well:

- LIMITATION OF AVAILABLE INFORMATION. Retrieved information is limited to static information available within Famix models. It includes some categories of elements from source code, which represents only a part of all information. It filters all information with this specific aspects.
- LIMITED UNIQUENESS OF INFORMATION ELEMENTS. It is not ensured that representations represent uniquely some specific entities. It limits reliability of associations between representations and source code entities.
- RELIABILITY OF REPRESENTATION OF MEANING. We assume that an entity name represents meaning of the entity. Reliability depends on quality of the entity naming schema used by the developers. Change Impact Detection is capable to limit influence of this factor on its quality.

4.2 Dependency detection

The goal of source code data extraction is the detection of dependencies to documentation. Dependencies represent a relation between the source code and technical documentation. A dependency indicates that a particular source code data belongs to a particular documentation segment. That means that the content of the documentation segment describes an aspect that is concerning the source code data.

4.2.1 Terminology

We refer to the definitions of the source code data extraction as specified at section 4.1.2. On addition we introduce definitions and terminology that is used within the dependency detection.

- DOCUMENTATION SEGMENT. A documentation segment represents a piece of documentation. Dependency detection reduces documentation segments to pieces of documentation texts treated as ASCII Strings. They are independent from any specific format or formal language like i.e. html or tex.
- DEPENDENCY. A dependency relates a source code data to a documentation segment. It represents a belongsTo-relation between a source code entity and a documentation segment. Sets of dependencies allow to express any m:n-relation.

4.2.2 Structure of dependencies

A dependency is a pair (x,y) where x is a name of a source code entity and y is a reference to a documentation segment. Both the name and the documentation segment are represented as ASCII-Strings. So, a dependency can be described as a ASCII-String as well, i.e. with a syntax like this:

```
(#<source code entity name>#,#<reference to a documentation segment>#)
```

The round brackets surround the dependency. The source code information and the documentation segment are both separated by comma. # serves as delimiter to mark the boundaries.

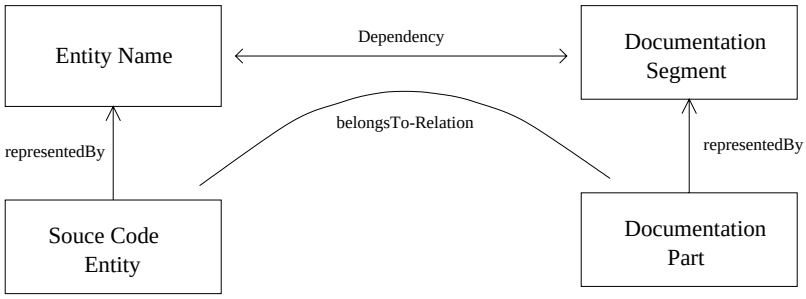

Figure 4.1: Dependency structure

A dependency is a REPRESENTATION OF A SEMANTIC BELONGSTO-RELATION. Such a relation is uniquely specified as a pair of a source code entity name and reference to a documentation segment. A pair of source code entity and documentation part is specified when the semantic meaning of the documentation part is concerning the semantic meaning of the source code entity. Such a pair is a dependency. As names represent source code element and documentation segment represent documentation part, a dependency is represented as a pair of this representations. The pair consisting of a name and a documentation segment represents a dependency. The representation of a dependency is often called dependency as well. It is then either obvious or irrelevant whether the dependency or its representation is meant.

4.2.3 Dependency detection

The goal of dependency detection is a process that builds associations on the basis of source code entity names and documentation segments to reflect semantic interdependencies between source code and documentation as good as possible.

Dependency detection has to work with available resources. As for source code information there are names that represent source code elements. Pieces of documentation text stand for documentation segments.

Plain text of documentation segments provide no structures that would support determination of semantic meaning of documentation segment. Documentation segments are written in natural language. Dependency detection regards documentation segments as sets of words. It cannot assume additional information structures as they are not provided by documentation segments.

Dependency detection works with the names that represent the source code entities. It looks for occurrences of the names within documentation segments. That‘s mainly a name matching process. Whenever a name appears in a documentation segment, an association between the name an the documentation segment is created. A dependency between a entity name and a document segment is created only once also if the entity name has multiple occurrences within the particular documentation segment.

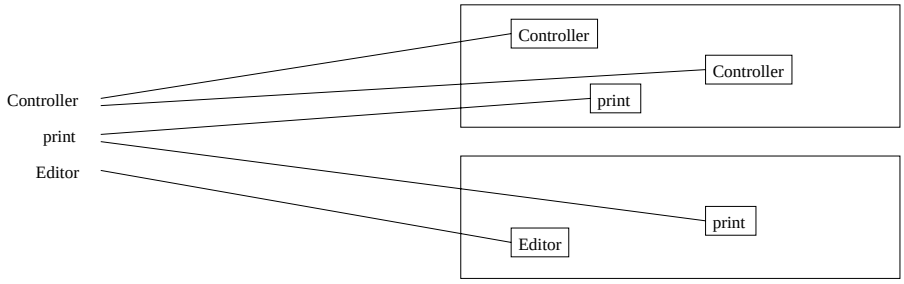

Figure 4.2: Dependency detection using source code entity names

Dependency detection determines meaning of documentation segments through determination of entity names within the documentation. The entity names reflect main aspects of the semantic meaning of the documentation segment's content.

4.2.4 Inferring Dependencies

Dependency detection explores the relationship between Famix models and technical documentation and expresses this relationship by dependencies. Additional dependencies can now be obtained by inferring new dependencies from already detected dependencies.

Inferring new dependencies is based on existing dependencies but includes additional structural information obtained either from the Famix model or the technical documentation. The additional structural information determine the rules how to infer the new dependencies. We consider Famix Associations as additional information from Famix models and documentation structures as additional information from technical documentation. We examine how to use this information for inferring new dependencies and reason why it is appropriate to extend the dependencies this way.

4.2.5 Inferring Dependencies using Famix Associations

Inferring dependencies using Famix Associations is a code-side dependency extension variant that takes inner interdependencies of source code into consideration. Software source code data do not exist semantically independent from other source code data. I.e. methods depend semantically with their containing classes and vice versa. Any semantic change on a method may influences semantics of its containing class. The potential influence of semantic changes on related source code elements also affects dependency detection.

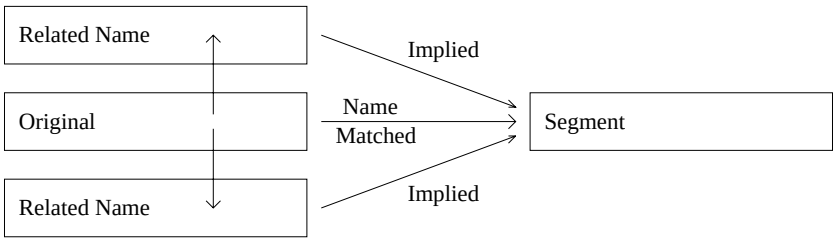

Figure 4.3: Code-side implication

Figure 4.3 shows the general way to take code interdependencies into consideration when creating additional dependencies. Assume the name of a Famix entity relates to a given documentation segment. The code element relates to two other code elements. I.e. the original code elements is a method that relates to a containing class and an attribute that is used as parameter. Then we say that the names of the two related code elements are related to the documentation segments by dependencies as well. Dependencies are generated for the related elements too.

The code-sided variant of dependency detection profits from the structures of Famix models that represent the software. Famix models contain associations that represent the relationship between the entities. There are three types of Famix associations. Refer to Appendix B for the complete List of Famix associations.

Inheritance

The inheritance structure relates classes as superclasses or subclasses. A classe relates to its subclasses it inherits the structure and that way its semantic to the derived subclasses. A semantic relationship is given to the superclasses as well as it inherits and eventually overrides the structure of the superclass.

Entity	*Relates to*	*Defined by*
Subclass	Superclass	InheritanceDefinition Property SUPERCLASS
Superclass	Subclass	InheritanceDefinition Property SUBCLASS

Table 4.2: Inheritance relationship of Famix entities

Invocation and Access

Invocation and Access are the third type of structural dependency of Famix entities that lead to semantic dependencies. An Invocation represents the definition of a BehaviouralEntity invoking another BehaviouralEntity. An Access presents the definition of a BehaviouralEntity accessing a Structural Entity. Invocation and Access infer semantic dependency as semantics of invoked or accessed entities influences the semantics of the invoking or accessing entity.

Entity	*Relates to*	*Defined by*
Method	Method	Method property ACCESSES and ACCESSEDIN
Method	Function	Method property ACCESSES and ACCESSEDIN
Function	Method	Method property ACCESSES and ACCESSEDIN
Function	Function	Invocation properties ACCESSES and ACCESSEDIN

Table 4.3: Invocation relationship of Famix entities

Aggregation

Famix entities relate other entities by aggregation. Classes contain methods and attributes, they aggregate these entities. The aggregated entities relate to their containing entity as their meaning is part of the semantic meaning of its containing entity. There is a wide variety of aggregation relations in Famix models.

Entity	*Relates to*	*Defined by*
Method, Function	Attribute	Access properties ACCESSES and ACCESSEDIN
Method, Function	FormalParameter	Access properties ACCESSES and ACCESSEDIN
Method, Function	ImplicitVariable	Access properties ACCESSES and ACCESSEDIN
Method, Function	LocalVariable	Access properties ACCESSES and ACCESSEDIN
Method, Function	GlobalVariable	Access properties ACCESSES and ACCESSEDIN

Table 4.4: Access relationship of Famix entities

Entity	*Relates to*	*Defined by*
Class	Method	Method Property BELONGSTOCLASS
Class	Attribute	Attribute Property BELONGSTOCLASS
Method	Class	Method Property BELONGSTOCLASS
Attribute	Class	Attribute Property BELONGSTOCLASS

Table 4.5: Aggregation relationship of Famix entities

4.2.6 Inferring dependencies using multiple referred documentation segments

Inferring dependencies using multiple referred documentation segments is a documentation-side dependency extension variant. It takes into consideration that the names of different source code entities may be related to the same documentation segments. This means that different source code entity names may contribute to the same topic, namely the topic described by the segments.

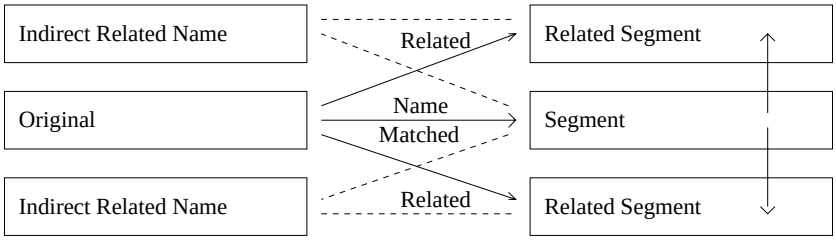

Figure 4.4: Documentation-side implication

Figure 4.4 illustrates how additional dependencies are generated indirectly over source code entity names that relate to the same source code element like a given name. For this name, all source code entity names are taken into consideration that relate to the same documentation segment. We determine all documentation segments that are related by these source code entity names. We generate now dependencies from the originally given source code entity name to these documentation segments as well.

4.2.7 Mixed Variant

The mixed Variant is a combination of the code-side extension variant and the documentation-side extension variant. Inner interdependencies of both the source code and the documentation are taken into consideration. Additional dependencies are generated concurrently. That means all additional dependencies are generated based on original dependencies that come from a simple dependency detection like name matching. Additional dependencies should not be generated based on other additional dependencies.

4.2.8 Advantages of inferred dependencies

The extension of the dependencies has two advantages:

- The dependencies get more independent from the source code entity names. It is more likely to detect dependencies as does not only rely on a explicitly given name but also on implicitly related names.
- It integrates additional structural information about the source code and the documentation. Inner Relations given by the Famix associations and the documentation topics are taken into consideration as well.

4.2.9 The name-detection trade-off

Dependency detection relies on name matching. The names are provided by and retrieved from Famix models that represent the source code. The names are representations of come from Famix entities and associations. There is some freedom to select either the whole fully qualified unique identifier as name or parts of it that may not be unique anymore. Short names are in advantage to detect more occurrences of that name within documentation. They are less sensitive for variation of names used in documentations. But short names are dangerous as they lead to false detection. False detection are detections that do not refer to the entity represented by the name we check.

4.2.10 Semantics and the definition of Correct

Dependency detection tries to establish a semantic connection between source code and documentation. Determination of entity name within a technical documentation works on a syntactic level. So it must be proved if the syntactically generated dependencies are semantically correct.

A dependency is SYNTACTICALLY CORRECT when it relates a code element to a documentation segment that contains the name of the code element. Dependency detection produces by definition syntactically correct dependencies. Syntactical proving of dependencies is checking this requirement.

The definition of SEMANTIC CORRECTNESS is more sophisticated. A dependency is semantically correct when it relates a source code element to a documentation segment that semantically affects the source code element. A documentation segment affects a source code element when its meaning includes some aspects of the source code elements. A source code element relates semantically to a documentation segment when any change of the source code element has potential influence on the documentation segment. A dependency is called semantically correct when it fulfils this requirement.

4.2.11 Validation of semantic Correctness

The semantic correctness of a dependency is proved by checking the requirements for a semantic dependency. It's a very critical part that requires human support. It's the developers responsibility to decide on semantic dependency. Developers have to specify semantic dependencies.

Dependency detection substitutes the specification of semantic dependency by syntactic procedures. That works if the syntactic procedures generate the same dependencies that would be specified by the developers on the semantic level.

Syntactically generated dependencies normally don't meet semantic dependencies to 100 %. There are different situations that influence the degree of syntactic dependencies being correctly semantic dependencies as well.

Syntactic Level	*Semantic Level*	*Case*	*Validation*
TRUE	TRUE	true positive	optimal
FALSE	FALSE	true negative	optimal
TRUE	FALSE	false positive	false, not critical
FALSE	TRUE	false negative	critical

Table 4.6: Coherence of syntactic and semantic dependency

Table 4.6 shows the four cases that occur in the syntactic-semantic-relationship. Identification of validation of this four cases is essential to determine the quality of change impact prediction.

The cases TRUE POSITIVE and TRUE NEGATIVE are optimal. The syntactic dependency detection creates correctly the semantic dependencies. Dependencies indicate correctly potential impact of source code elements on documentation segments. Source code elements that are not related to a documentation segment have no impact on that segment. The FALSE POSITIVE case decreases reliability of dependency detection. It occurs when a syntactic dependency does not correspond with a semantic dependency. That means the syntactically generated relationship does not correspond with semantic relationship. The FALSE POSITIVE case is not critical. Semantic checking of all syntactic dependencies detects all dependencies that are FALSE POSITIVE.

The most critic case is the FALSE NEGATIVE case. A semantic dependency is not detected covered by a syntactic dependency. That means that the syntactic dependency detection did not find the semantic dependency. This case is very critical because it prohibits to exclude documentation segments that are not related by any of the syntactic segments. If there are FALSE NEGATIVE cases, documentation segments with no syntactic relationship to any source code element may have semantic dependency to the source code anyhow. FALSE NEGATIVE cases can only be detected by checking semantically the whole documentation. It's a very critic part of any documentation support system. If there are many FALSE NEGATIVE cases, any system looses its reliability in dependency detection.

Chapter 5

Associative Documentation Model

> ”Each generation imagines itself to be more intelligent than the one that went before it and wiser than the one that comes after it.”
>
> *George Orwell*

This chapter introduces the basic architecture of the Associative Documentation Model (ADM). ADM is a general model to associate technical documentation with software source code. It’s an abstract model that describes the code-documentation relationship independently from the specific problems of source code and documentation handling. ADM provides the basic relationship structures that fix to several implementation variants. ADM allows to explore the global characteristics of any associative documentation.

ADM integrates the knowledge about representation and handling of source code and documentation. It determines namely the representation of source code within ADM. It influences structural requirements for the documentation representation. ADM takes general characteristics of source code and documentation into consideration.

ADM is a basic model that introduces a new way of reflection of documentation. It regards documentation as being information carrier. Special focus is set on dependencies between the information contained by documentation and the information contained in described subject. ADM is a new, very abstract approach to see documentation. This approach is target oriented. It lays foundation to investigate the balance between documentation and real subjects it describes. I.e. changes on real subjects can be implied based on the ADM model on describing documentation.

Requirements first determine characteristics of the model. Axioms lay foundation of the ADM model. ADM is built up on that axioms. Specially treated are documentation, subject and dependencies that represent the core part of the ADM model. Possible situations and configurations are analyzed. ADM is proved about reasonableness. Extensibility and adaptibiliy onto different environments are investigated. Special focus is set on expression power and limits of this model.

5.1 Basics of ADM

The goal of the Associative Documentation Model is to get an model that integrates the two entities SOFTWARE SOURCE CODE and TECHNICAL DOCUMENTATION and describes the dependencies that represent an association as a relation between the two entities. The ADM Model must fulfill a bunch of requirements that determine the characteristics and the expression power of the model. The Specifications must deliver a global valid model that expresses special dependency of the documentation from the described subject in an appropriate manner. Following requirements determine a model with that abilities.

- ABSTRACT. The ADM model is detached from any concrete form of technical documentation. Documentation is regarded as a text. It doesn‘t consider implementation specific aspects nor does for storage and retrieving mechanisms. ADM includes no specifications about technical implementation and accessibility of documentation. ADM is independent from content or purpose of documentation.

- COMPREHENSIVE. The ADM model shall cover all types of technical documentation. It represents a metamodel with metastructures that lets integrate specialized models. Each documentation can be reduced and adapted to ADM.

- OPEN. ADM is open for new forms of technical documentation and future developments in documentation technology. We cannot wait for future technologies but we can provide structures that allow easy integration and adaption of that technologies.

- EXTENSIBLE. Special extensions required for special purposes are possible. Features for special purpose documentation support techniques can be added.

- REFINABLE. The abstract, comprehensive ADM model can be concretized by refinement. Special case dependent properties can be added. The ADM Model gets more concrete that way but less generally valid as well.

- ADAPTABILITY. The global valid model can be adapted and implemented for concrete documentation environments.

- INTEGRATABILITY. ADM can be integrated into existing systems. It is built on existing, commonly available technologies and allows support by existing tools.

- REAL. The ADM model reflects characteristics of real existing technial documentation. The model delivers a specific definition that corresponds with experiences with documentation.

- UNDERSTANDABLE. The ADM model is intuitively understandable. It supports human control specially required for controlling and debugging purposes.

- REASONABLE. The ADM model is reasonable. Reasonable statements about its characteristics, its possibilities and its power and limits can be expressed.

5.1.1 Basic Definitions

The Associative Documentation Model bases on 5 basic definitions.

> **Definition 1:** Software source code is represented by a set of data with inner relationships.

Software source code is regarded as a set of entities that relatate to each other by associations. Refer to the Famix model as an example of an model that follows this view. ADM builds on the entities. The associations are represented as inner relations of the entities.

> **Definition 2:** Data are regarded as encapsulated.

ADM models the software source code entities as data. Data are encapsulated. Encapsulation signifies that data are treated as a unity. Data are not identified semantically. It is not defined what constitutes and incorporates data.

> **Definition 3:** Technical documentation is regarded as a set of segments.

The technical documentation is divided into an set of segments. That set of segments represents the documentation. It is by definition not limited to an maximum number of segments. Technical documentation is focused on its role as information carrier providing its information within its text. Neither its form nor its contents are regarded. That handles documentation on a very abstract level.

> **Definition 4:** Documentation segments are text that contain any information.

The Segments of technical documentation are handles as pieces of text that contain any information. It does not define what constitues information.

> **Definition 5:** Documentation information are either atomic or an aggregation of information.

Atomic information are information that represent out of itself any semantic information. Atomic information are elementary by meaning of bringing information independently of other information. Examples would be term like 'Controller', 'File', 'Object'. They all deliver information about what real object is talked about. Aggregated information are merged out of other information. That could be both atomic information or aggregated information as well. Aggregated information get their meaning from relations between other information.

> **Definition 6:** There is a relationship between the set of information representing documentation and the software source code it describes.

Information provided by the technical documentation describes real facts of the described software source code. There is a strong relation between the segments of technical documentation providing information and the software source code data it relations to.

5.2 The Model description

The Associative Documentation Model is an Entity-Relationship-Model that expresses the relationship between technical documentation and the software source code. It founds on the basic definitions of Section 5.1.1 and fullfills the requirements for a general abstract model. We focus on the specific needs for modeling relationships between source code and documentation. ADM should serve as a platform for applications that build on that relationship.

5.2.1 The basic architecture

The ADM architecture is defined on a very abstract level. According the rules, terms like information, facts and dependencies are central. The Associative Documentation Model (ADM) is a very basic model for describing interdependencies between real subjects and documentation. Its core element is the relationship between the reality subjects and the documentation. The relationship fixes dependencies between the real subject and the documentation that allows implications to each other. What exactly constitutes these entities and segments respectively is out of scope of the core definition.

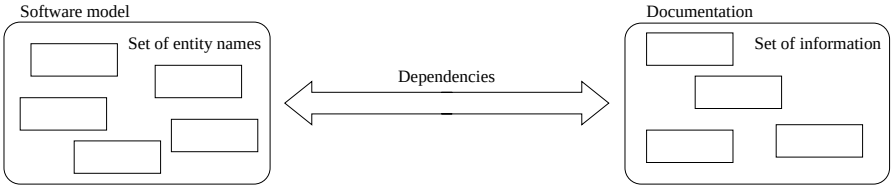

Figure 5.1: The ADM architecture

Figure 5.1 illustrates the dependency of documentation model and real model in an relational manner. Both the real subject and the documentation are represented as entities. The dependency can be seen as a relation between these two entities.

Lets consider now the specific relationship between source code and documentation. We need models of both the source code and the documentation to fit them into the general ADM model. The ADM model lets relate them to each other by dependencies and build up applications that derive from the relationship.

5.2.2 Source code model

The source code model is an entity of the ADM model. It represents the real source code. Real source code is regarded as a set of facts. The source code model is designed as a set. The set elements are representations of source code elements. Analogous to the documentation model, the source code model is infinite. Theoretically, a source code model is a perfect projection of source code. In practise, finite sets are used. They represent a subset of all reality facts. Figure 5.5 shows the architecture of the source code model.

The source code model takes into consideration that source code elements relate to each other semantically. The semantics of a method or an attribute i.e. has influence on the semantics of its containing class. Or the semantics of a method affects the semantics of an attribute accessed be the method.

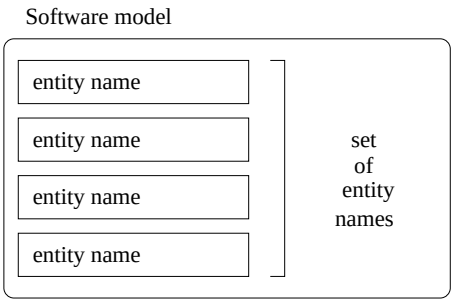

Figure 5.2: The source code model as a set of elements

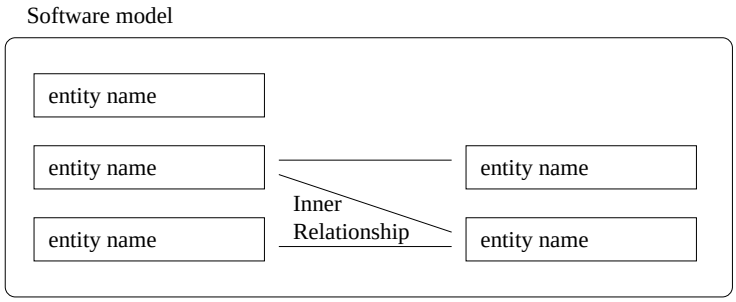

Figure 5.3: The source code model with inner relationship

Figure 5.3 shows a source code model that includes the inner relationship of source code elements. The meaning of the inner relationship is semantic influence. The definition of inner relationship depends on the elements. Refer to section 4.2.5 as an example.

5.2.3 Documentation model

The documentation model is an entity of the ADM architecture that represents statically isolated documentation. Figure 5.4 shows the basic documentation model.

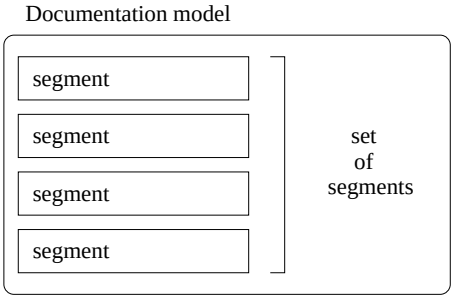

Figure 5.4: The documentation model as a set of information

The documentation model is defined as a set. The elements of this set are the information provided by documentation. The set of information elements represents the

documentation. Documentation and its representation are two different things. Theoretically the documentation model is a exact representation of the documentation. In practice it provides a representation of a subset of information contain in the particular documentation.

Documentation information are represented by elements of the documentation representation. Information itselves are represented according rule 1 and rule 3: Information are existent, encapsulated and either atomic or an aggregation. The representation of information represents that characteristics as well. Refer to Figure 5.5 for a typical structure of information within a documentation model.

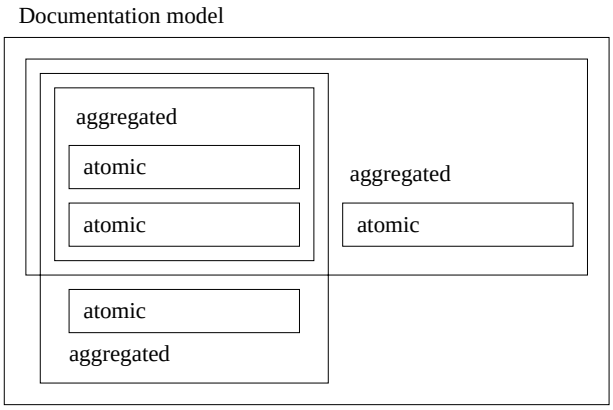

Figure 5.5: Aggregation of information

Atomic elements represent elementary information. Aggregation represents information that result of relation of different information. Both atomic representation and aggregated representation participate in aggregations. Aggregations are recursive structures. They can overlap. Atomic information and aggregated information can be included in different aggregations. That corresponds to the fact hat information can relate to many other information Each of this relations provides some aggregated information that is represented by an aggregated representation.

ADM regards atomic representations and aggregated representations equivalently. Dependencies relate to both types of representations the same way. Aggregated representations are treated as independent, encapsulated representations. That means serialization of the aggregated and recursive structures into a series of independent elements.

5.2.4 Dependencies

The ADM model gets its power from dependencies between the documentation model and the reality model. Dependencies are called associations as they associate informations with facts. Associations are formulated as relations.

The are mainly a m-n-Relation. This distinguishes from common document patterns where software entities are described by a particular documentation pattern what implies an n-1-Relation.

The dependency has by default the meaning of a *has impact* relation. The participated elements of an association transfer any change of their status to each others. That way dependencies allow to maintain consistence between the documentation model and the reality model.

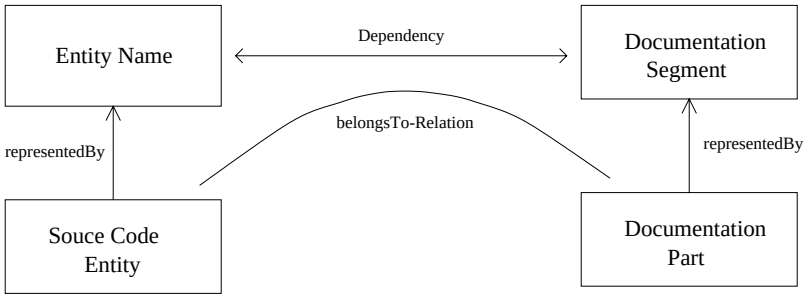

Figure 5.6: Dependency structure

The meaning of association can be changed in two ways. The one possibility is to change the meaning of association implicitly be choosing specific methods to build the dependencies. For example, relating entities of an software code (representing a reality model) to segments of an documentation (representing a documentation model) by using its names would lead to dependencies with the meaning that any entity belongs exactly to all segments containing its names. Validating usability of such methods depends on intended purposes.

Entity Name	Documentation Segment	Properties (i.e. has_changed, is_old)

Figure 5.7: Dependency Properties

Another way to change the meaning of dependencies is using properties (Figure 5.7). Each property then stands for a specific aspect the relation stands for. Imagine an association with two attributes *has_changed* and *is_old*. They would be both of some boolean type. An dependency containing the *has_changed* attribute set to true would stand for the meaning of the respective entity may has impact on the related segment whenever a change status occurs at the entity. The *is_old* attributes would imply any effect to the documentation segment when the respective entity gains some oldness state.

The architecture of dependencies depends on the architecture of the available software. Dependency attributes not being derived out of the source code model cannot be set at any time and are therefore absolutely useless. On the other hand, derivable aspects of software model with not according association attribute cannot be associated with documentation.

Lets consider this on an example: *Entity2*, *Entity3* and *Entity4* are elements of software code. *Segment2* and *Segment3* belong to a documentation model. The Association shows that *Entity2* belongs to *Segment2* and *Entity3* as well as *Entity4* belong to *Segment3* under the aspect of change. That means that if any change happens to some of these entities, it would may have impact on the respective segments.

The given association provides the attribute new. This attribute will never be used as its value cannot be derived out of the software model. On the other side, the derivable aspect old cannot be used to associate entities with documentation segments under this aspect as it is not supported by any association attribute.

The dependency has, like ADM as a whole, a very open architecture. It does not

limit neither the choice nor the number of attributes, nor does it on any specification of the attribute values. The great advantage is its adaptability to a wide variety of situations. As some disadvantage must be seen the fact that the responsibility for reasonable and meaningful dependencies is delegated to external authorities. Out of itselves, associations cannot guarantee any of this qualities. We have to deal with that.

The values of dependency attributes are not described by ADM as well. Their form and range definition and interpretation must be given externally by definition of association attribute. On addition it also an responsibility of attribute definition to guarantee consistency of values generated by association building with the value range of the attributes.

5.3 Change Impact Prediction

The Associative Documentation Model is used for Change Impact Prediction. Change Impact Prediction is an application of ADM that infers changes within the software source code into the technical documentation and indicate potential change requirements within the documentation.

Dependencies have many applications they could serve for. One of them is IMPACT PREDICTION. Dependencies can be used to perform comparison operations. Involved elements and facts are proved on whether they fulfill the association purpose. If this is not the case the documentation element can be adapted to the source code element it relates to. This way the documentation is synchronized with the real topics it intends to describe.

Synchronization is the way to detect impact. Impact occurs whenever a given dependency is not fulfilled by the dependency elements and facts. Impact indicates that involved elements and facts are not synchronous and there is a requirement for change of the documentation element to reestablish truth of the dependency.

Comparing is the core operation of impact detection. The comparison operation is based on the dependency. Related source code elements and documentation information are compared to each other. Results are either true or false. True signifies that the meaning of the source code element and the documentation information is equal, false represents them being not equal. Equality can be defined differently and depends on the aspects we're interested in as well as the purpose of the dependency. I.e. within dependencies binding software code elements with documentation segments using code element names and occurrences of these names within documentation segments, equality of two related elements is specified by functions that check whether related elements and facts fulfill this building rule.

Impact detection and synchronization are two steps to regenerate consistent documentation. Impact detection identifies inconsistencies of documentation. Synchronization eliminates inconsistencies and reestablishes a synchronous state between the documentation model and the source code model.

5.4 Model completeness

Models are projections of their subject. Projections reduce the models to some aspects. The models are not complete by meaning of containing any information of its subject. As visualized in figure 5.8, a model is a subset of all available information.

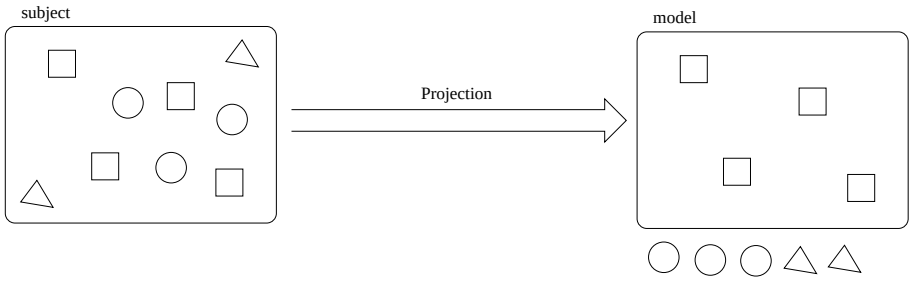

Figure 5.8: Projection of models

Reduced models force distinction of information contained in the model and information not included in the model. Distinction is essential for analysis of different situations within ADM. There are 8 essential configurations to be considered. Figure 5.9 illustrates the most important configurations that may occur in an ADM model. This configurations have huge influence on specific applications that base on ADM.

- **Case 1** shows an ADM model with an empty set of associations. None of the real facts is associated with any documentation element. That means that documentation does not contain information about the reality subject. Its the wrong documentation relative to the reality subject. The documentation does not belong to the reality subject.

- **Case 2** is the case with one reality fact being associated with just one documentation element. That means the reality fact is described by exactly that documentation element. This case occurs for example in listings of code objects where each code object is described 1:1 just by one specific description.

- **Case 3** shows 1:n-dependency of reality facts and documentation elements. A reality fact is described by lots of different documentation elements. This happens when a reality fact is mentioned under different aspects at different places of documentation.

- **Case 4** reflects the reverse case. Many different reality facts are described in one documentation element. That happens when features are described within documentation that affect many reality facts.

- The mixed case as shown in **case 5 and case 6** occurs when reality facts are included in the reality model but are not documented within the documentation model. The reverse case is also possible where the documentation model describes reality facts not provided by the reality model. This mixed cases are essential as they limit quality of application based on ADM.

- **Case 7 and case 8** show complex cases. Association 1:n and n:1 are mixed as well as model internal and external documentation elements and reality facts are involved. This case occurs often with general documentation having no narrow relation to the structures of the described reality structure.

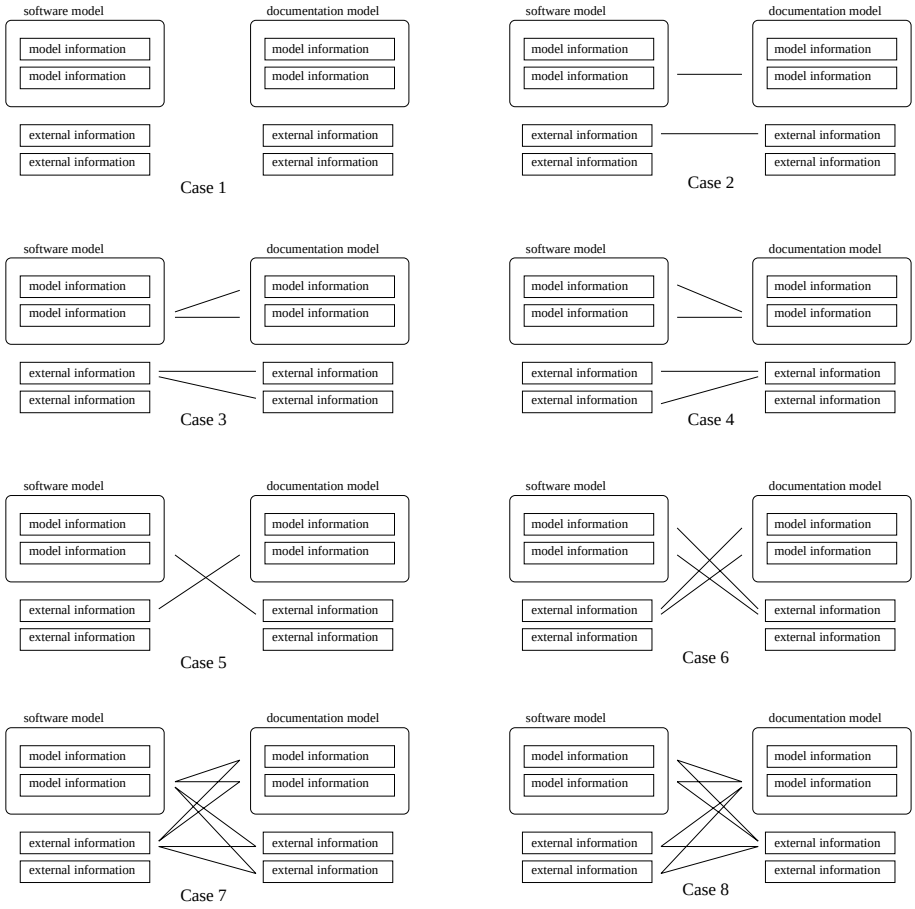

Figure 5.9: Dependency configurations

5.5 The result

The ADM model is designed as a abstract, open and extensible model. It fits to any type of documentation as a basic model. Its core feature is the ability to describe any source code related documentation. ADM models that dependency and makes it expressible and usable for further applications.

The analysis of the ADM model shows that it contains models for software source code as well as for documentation and is capable to express the relationship between these two models by dependencies. So all resources for Change Impact Prediction are available. This means that the ADM model is suitable to perform Change Impact Predition. This corresponds with the **Hypothesis 4**.

Chapter 6

The ADM documentation process

”Any sufficiently undocumented code is indistinguishable from magic.”

Unknown

The goal of ADM is to support the DOCUMENTATION DEVELOPMENT PROCESS. The ADM documentation process describes a documentation development process that is based on the ADM model. The process shows how to create and maintain documentation using the ADM environment, especially dependencies between source code and documentation. Features like SOFTWARE CHANGE ANALYSIS and IMPACT DETECTION deliver additional data about the software and support the dynamic process of documentation development. We explore sufficiency of ADM for documentation processes that run concurrently to evolving software.

6.1 The ADM Documentation Utility

The ADM documentation utility is a documentation support tool. It implements the ADM concept and provides features to support of a dynamic documentation process. Its essential abilities are SOURCE CODE INFORMATION ANALYSIS, DEPENDENCY EXPLORATION and IMPACT INVESTIGATION on documentation. The ADM utility supports the documentation process with analytical data about the software and their impact on the documentation.

The ADM data resources are organised as a relational database. The software model as well as the documentation model are seen as entities. Dependencies are represented as relations between the software model and the documentation model. All data resides in a database and serve for further analytical operations.

The ADM documentation utility is implemented in a Visual Basic for Access environment and runs on 32-Bit Windows operating systems. Visual Basic for Access was chosen as it allows rapid prototyping of applications that could be easily adapted to specific requirements. Additionally it corresponds with the Access database that serves as experimental database environment that runs on 32-Bit Windows operating systems as well.

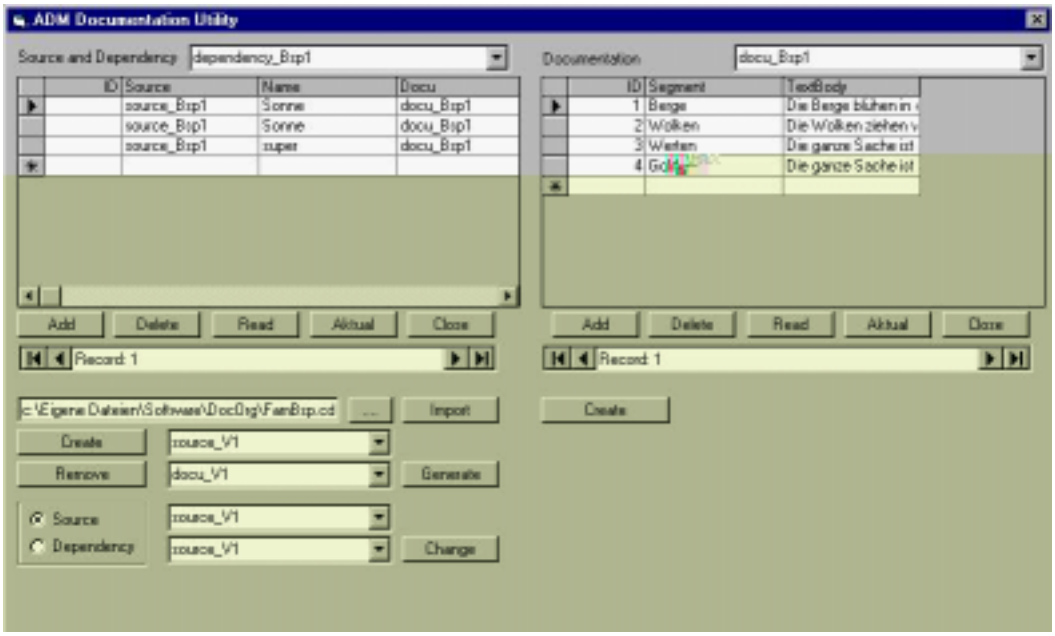

Figure 6.1: The ADM documentation utility

6.1.1 ADM data import

The ADM documentation utility provides features to retrieve and edit information about the software that shall be documented. It supports parsing of Famix models and manual manipulation of software description data. Obtained information serves as description of the software.

The Famix data import allows the import of software description data from Famix models. It parses CDIF-files that contain Famix models and provides the information required by ADM. The Famix data import looks for entities of the model and detects the names of these entities. The names are taken as representations of the entities. The Famix CDIF-parser of the ADM documentation Utility generates both short names and long names. Short names are only the name of the entity itself while long names include additional information about the environment of the entity.

Manual editing is the second possibility to generate and manipulate software description data. The ADM documentation utility provides full editor capabilities to add, change and remove ADM data. This feature is not primarily provided for creation of complete software models but for manipulation of models obtained by the Famix import facility.

6.1.2 Change analysis

The ADM documentation utility works with a dynamic process where software is seen as a dynamically changing issue. It analyses changes of different software models and determines potential impact on the documentation of this software.

The ADM documentation utility detects the differences of software description data sets. All names that occur only in one of two sets, but not in the intersection of these sets, incorporate the difference of the sets. The utility compares all names of one set

for presence in the other set. A name exists in the other set if it occurs unchanged, that means with the same unique name, in the other set.

Two fields in the ADM documentation utility serve for the selection of two sources. Preferably the upper field is for the older source, the lower field for the newer one. When clicking the generate-button, a dialog appears that asks for a name of the database entity in which all detected changes are stored. When change detection is performed, the results are available in that entity.

Computation of the change is a very time consuming process. The ADM documentation utility stores computed software changes for further use. So change between two software versions has to be computed only once and could be reused at any suitable moment.

6.1.3 Dependency generation

The ADM documentation utility provides automatic dependency generation. The dependencies represent the relation between the software description data and the documentation segments. The dependencies are stored as an entity in the database. They can be completed and edited manually.

A dependency relates a name that represents a software entity to a documentation segments. So the dependency must contain references to both the name and the segment. The ADM documentation utility implements a dependency with the properties SOURCEMODEL, TYPE and NAME as reference to the name and DOCUMODEL and SEGMENT as reference to a particular documentation segment.

Dependency generation is mainly a name matching process within the documentation segments. The ADM documentation Utility provides various variants of name matching. They differ in what kind of name representations to take as search patterns and how to search patterns within the documentation segments. The following list encounters the variants implemented by the ADM documentation utility.

- SHARP SHORT NAME DETECTION. The Sharp short name detection uses short names as representations of software entities and looks for it in the segments in a sharp manner. It only matches if the short name occur as a single word, but not as a part of a more extensive word.

- FUZZY SHORT NAME DETECTION. The fuzzy short name detection uses short names, but is more flexible at detection with documentation segments. It matches if the short name occurs as a word or as a part of a larger word.

- SHARP LONG NAME DETECTION. This dependency generation variant uses long names that include context information as search pattern within the documentation segments. It matches if the long name occurs as one word. It must not be part of a larger word.

- FUZZY LONG NAME DETECTION. This variant uses long names as search pattern and is flexible as searching within the documentation segments. It matches if it occurs as one word or as a part of a larger word.

Which variant is used is declared when generating the dependencies. The variant selection influences the results of dependency generation. This must be considered when analysing and interpreting the impact detections that use this dependencies.

6.1.4 Documentation

The documentation is organised as a set of segments. Each segment is regarded as a unity. The definition of such a segment is part of the developer's documentation work.

The ADM documentation utility provides editor capabilities for creation and manipulation of documentation segments. The segments are organised as a table. Each row represents a segments. The segment properties are implemented as fields. The properties values occur as values in the fields. Editing any values means changing the values in the respective fields.

A documentation segment includes the properties SEGMENT and TEXTBODY. The property segment contains a unique identifier that identifies the segment. It is recommended to take a short name to describe the content in the TextBody content. The TextBody property contains the documentation text of the segment. The ADM documentation utility allows any kind of documentation text. This includes additional text formating like html tags. That's of interest if the documentation segments are merged together to get a complete version of documentation.

The documentation segments must provide additional properties for documentation organisation purposes. The properties VERSION and ORDER are required for documentation versioning and documentation merging. The property version identifies which segments belong to a given documentation version. It's essential for generation of a documentation version from a set of segments that belong to various versions. The order property specifies the sequential order of documentation segments of a given documentation version. The order is required for merging complete documentation from a set of segments.

6.1.5 Documentation analysis and Impact detection

The ADM Documentation Utility uses dependency generation for two specific features, the DOCUMENTATION ANALYSIS and the IMPACT DETECTION.

Documentation analysis is based on dependencies that are generated for a particular source code model and an documentation version. The analysis of generated dependencies investigates how far the software is documented. The ADM utility offers statistic analysis about occurrences of software descriptions within the documentation.

Impact Detection focuses on the influence of software changes on documentation. The ADM documentation Utility generates dependencies between software changes and documentation. The dependencies deliver direct indication which segments are influenced by the dependency. All segments that participate at any of the dependencies might be influenced by the software changes.

The ADM documentation utility presents STATISTICAL AND VISUAL INDICATIONS for the segments that are involved in the dependencies. Marking of the segments visualises all segments affected by the change. That helps to control the documentation as it indicates easily which parts are to check and which parts remained unattached.

The NUMBER OF CHANGE that influences a particular documentation segment indicates how much software changes influence the segment. The ADM utility offers statistic analysis that shows how many software changes and what kind of software changes influence the segment. It presents an analysis of all the types of changes and allows the selection of some types. There are different variants to count changes. The class-change counting i.e. counts changes within a class only once while details change counting counts each change of its own.

The documentation utility also VISUALISES THE CONCRETE CHANGES that influence a particular segment. It lists on demand all changes with effect on the segment. This helps the developers to consider what might cause changes on the segment and indicates what has to be changed semantically.

6.1.6 Data storage and documentation generation

All data about software, documentation segments, software changes and dependencies are stored within a database. The database manages persistence and accessibility of the data. The ADM documentation utility has no limitation on the amount of data. Limitations are only caused by the software environment and resources of the system.

The ADM documentation utility provides the feature to generate documentation from the documentation segments. It merges the documentation segments of a selected documentation version sequentially together according the order implied by the segments order property. Documentation formatting is part of the documentation segments. It is not added when generating a documentation.

6.2 The scenario

We use the ADM documentation utility for documentation of an evolving Java-Implementation of the Famix model. This implementation provides a representation of Famix models as a hierarchy of Java classes. The inheritance hierarchy of the classes models the abstraction hierarchy of the Famix models. All classes provide the standard properties of the standard Famix models as attributes. The Famix Java implementation allows any additional properties which reflects the extensibility of the Famix entities.

The Java implementation of Famix provides additional features for import and export of CDIF-Descriptions of Famix models. It parses CDIF-Files and instantiates a Java representation of the Famix model in that file. The print-method works as counterpart that writes down a Java represented Famix model as a CDIF description.

The Famix Java implementation is documented using the ADM documentation utility. The scenario provides documentation in two steps.

- DOCUMENTATION OF THE OLDER VERSION. The older version is first be documented. This is a classic documentation exercise. A complete new documentation is created that describes technical aspects of the software. The documentation provides information about the structure of the object oriented software, especially information about the entities, inheritance hierarchies, aggregation and invocations. It contains information about functional groups, their interactions and their meaning and role within the software. It also includes general information that relates to the software as a whole.

- DOCUMENTATION OF THE NEWER VERSION. The documentation of the newer version is the second step of the scenario. It is developed on the basis of the older documentation version. The resources are available from step one. We analyse the changes that occur from the older to the newer software version. We relate the changes to the documentation segment and analyse how far the changes influence the documentation segments. If required, the segments are updated. We get an updated documentation version whose updating is supported by the ADM features CHANGE ANALYSIS and IMPACT DETECTION.

The goal of this scenario is to explore on a concrete example the suitability and fitness of the ADM architecture for a documentation process that creates documentations and supports update operations that lead to documentation updates. It examines the use of the update support features CHANGE DETECTION and IMPACT DETECTION for a dynamic documentation process.

6.3 The documentation process

The documentation process of the Famix Java Implementation is designed as a dynamic process that includes the creation and adaptation of documentation segments. It can be divided into two phases.

Phase 1: Documentation creation

The documentation process starts with the documentation of version 1 of the Famix Java implementation. The main part of this phase is the creation of new documentation segments. It gains from information about the software.source code provided by the ADM Documentation utility.

The documentation starts with a description of the general architecture. It shows the main functional blocks and describes their characteristics and purpose. This documentation part describes the intention and function of the Famix Java implementation as a whole.

The documentation goes on with a detailed description of the functionality and characteristics of the functional blocks. It shows their structure and how to integrate it into the software. A special aspect is the interaction of the functionality block to each other. The description of the general structure of the Famix representations shows the common characteristics of all Famix representation classes. The structure leans narrowly on the abstraction structure of the Famix model. This information is very important for further extensions or adaption of the Famix Java implementation.

Finally, all classes are described in detail. It contains detailed information about methods and attributes of the classes. It focuses on the features of the concrete Classes as representations of Famix entities.

Phase 1 leads to a detailed documentation about the version 1. It profits from information about the source code but is mainly a common documentation work about a given software version.

Phase 2: Documentation adaption

The essential support for the ADM documentation utility appears during phase 2 of the documentation process, the documentation adaption. The goal of the documentation adaptation is to get a documentation for version 2 of the Famix Java implementation on the basis of the documentation of version 1.

The author determines the changes of the software and examines which parts of version 1 must be adapted. He determines obsolete segments and eliminates or changes them. He adds new documentation segments and completes the documentation's content according to version 2.

The ADM documentation utility supports this phase with change detection, dependency generation and impact prediction. Change detection determines the changes from version 1 to version 2. It shows the developer which parts of the source code have

changed and must be considered as potential change requirement on the documentation version 1. Dependency generation allows developers to determine which software parts are already documented and which parts must be added. Impact prediction combines change detection and dependency generation and indicates which changes must be considered and where they may take effect on the documentation.

The documentation of the version 2 of the Famix Java implementation profits significantly from the ADM support features. It is much easer and more efficient to retrieve the information about changes and possible impact on documentation. It serves as a solid basis for the authors decisions about which documentation contents to change and which new information to introduce. The ADM documentation utility delivers remarkable support, the final decision about the documentations adaption remains to the authors.

6.3.1 States of Documentation

The ADM documentation process runs through states of documentation. Transitions from one state to another represent actions that move the documentation forward to another state. Figure 6.2 illustrates the ADM documentation process.

The Famix Java implementation documentation of the scenario is created according this process. Let us look at the detailed steps to the process and show how they are performed in the sample scenario. Refer also to the example in appendix C.

The documentation process starts with the development of the first software version (1). Its a normal software development process that results in a version of the software.

```
Software:  Java implementation of Famix, version 1

Resources: Java source code
           Java binaries

Remark:    Source code and documentation are fully available.
           That is important source code analysis and change
           analysis.
```

At this point we make the decision to document this software (2). We first fix the guidelines. The guidelines include type and structure of the documentation, audience and goals and content definitions (type, amount and degree of details of the information). Additional guideline specifications are added at this point if required for a particular documentation.

```
Decision:  Documentation of the Famix Java implementation, version 2

Type:      Technical documentation
           Documentation as an addition to the source code

Audience:  Developers

Content:   Basic information about the application
           Architecture
           Functional blocks
           Description of entities, inheritance, aggregation
           and invocation \\
```

We examine the available software (3) and generate a software model (5) that includes essential technical information like entities, inheritance, aggregation and invocation (4). This information are obtained by computation and serve as an information basis for documentation.

```
Generation of a Famix model that describes the Famix java
implementation.
Retrieve the names of classes, methods, attributes
Detect inner dependencies of the Famix model, namely
inheritance and aggregation
```

The documentation process continuous which description of documentation segments(6). Its the core part of the documentation process. The segments are written according the basic guidelines as declared at the beginning of the documentation process (2) The segments are built on the model information (4). The documentation developers contributes additional information and completes it. Writing of segments is the central work of the developer. It's his responsibility to decide about concrete structure and content of the segments (7).

```
A set of documentation segments that contain information about
the Famix java implementation. It contains 3 kinds of segments:
Segments with general information: title, author, date, purpose
Segments with functional groups: Famix core, CDIF parser, Printer,
Interfaces Segments with code details: entities, inheritance,
aggregation
```

Generation of documentation merges the documentation segments into one consistent documentation (8). Documentation structures and formatting schemas are introduced at this point. The result is a concrete documentation that includes the information of the segments and follows the structure and formatting guidelines.

```
Select all segments of version one.
Use html as formatting schema.
Generate html pages that serve as documentation.
```

6.3.2 Iteration

The ADM documentation process provides iterative updates of the segments. Its a feature as it supports adaptions and revisions of the documentation.

A documentation update starts with changes of the described software (10). That includes any kind of change as it occurs in an ongoing software evolution process. It results an updated software version(11).

```
Improvements of the first Famix java implementation lead to version two
of the implementation. The source code of the new version is available
and serves as the basis for the Famix model generations.
```

We generate a model of the new updated software version (12). That works analogous to (4). Comparing the models of the older (5) and the newer (12) software models delivers the software model changes (4).

```
Comparison of the older and the newer model of the Famix java
implementation. The changes are represented as names of all entities
that are added or removed.
```

We relate the changes (13) to the set of segments we have about version one (7). It indicates all segments that might be influenced by the changes (14,15).

```
The documentation segments are scanned about names of the changes.
If it matches, the change is related to the documentation segment.
```

As the last step of the update, the developer must evaluate all detected impact. He changes documentation segments if it is required. That includes addition of new segments.

```
Update of version one segments
Addition of new segment
```

Merging of the updated segments delivers a new documentation version that fits to the new software version. Its the same process as shown at (8).

6.4 Conclusions

The ADM Documentation Utility leads to some interesting conclusions. The features to vary the dependency generation and change impact predictions have to be used for suitable situations. Each variant has its advantages and disadvantages that qualify it for particular purposes.

- NAMING IN THE DYNAMIC DOCUMENTATION PROCESS. Name matching as the fundamental dependency generation principle influences the documentation creation. The intended dependency of the documentation segments influences the structure and the description of the segments. Documentation writers are more likely to segments the documentation according the structures given by the source code. Segments are written in a more encapsulated manner. They tend to describe just one topic per segment.

  As naming is essential for dependency generation, the writers use a more consistent naming in their documentation. They tend to lean closer to the naming schemas of the source code. Beside better performance of the dependency generation and documentation support, it clarifies to which software parts a segment belongs to. It helps developers to check a documentation about its descriptions of the software. So, under this conditions, dependencies that are detected by matching software entity names within documentation support the synchronisation of technical documentation with the evolving software. They contribute to technical documentation that is up-to-date. This corresponds with the **Hypothesis 5**.

- INCLUSION OF FAMIX STRUCTURES.

  The Inclusion of Famix structure information into dependency generation indicates possibles dependencies that may only occur indirectly. This presents the documentation authors the environment of the models. It helps for two aspects: when documentation writers adapt some documentation parts, it indicates in a more contextual environment what belongs to a particular segments and what may have impact on the content of the segment. It shows all software elements, that are either directly detected within a segments or belong to such a element as a structural relative within the source code model.

The second aspect is more flexibility for validating dependencies. The inclusion of Famix structure information lets determine also documentation segments that may be influenced by software change but only indirectly over structural relatives of that element.

- SEGMENTED DOCUMENTATION.

  Segmented documentation supports partial, ongoing documentation. It forces documentation of information when it occurs, independently from whether they be complete or not. It animates developers to document their work without concerning any questions about the structure or content of the final document.

  Segmented documentation also helps to distribute documentation writing among more developers. They contribute parts of documentation independently from each other. The Developers deliver the resources from which a documentation then can be generated. They do not have to care about coordination or final documentation building.

- NARROW INTERDEPENDENCY TO THE SOURCE CODE. The narrow interdependency to the source code supports the control of the documentation's content. Faults or lacking information are easier to be found. It forces the developers to be more precise and complete with their information.

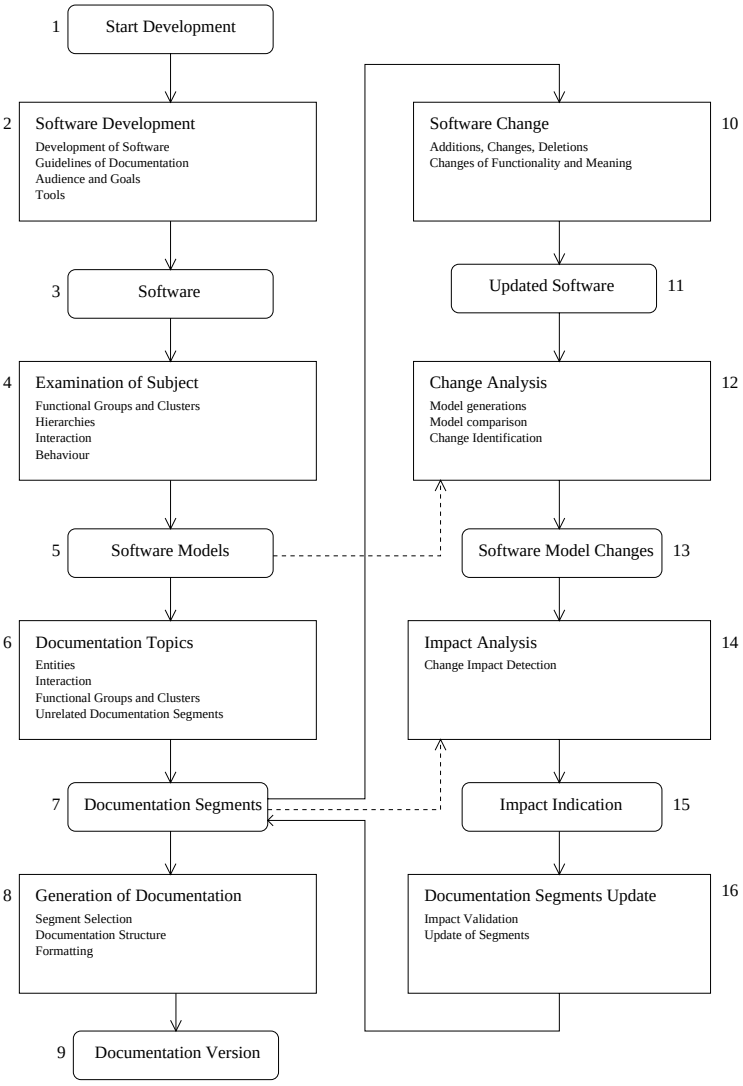

Figure 6.2: The ADM documentation process

Chapter 7

Conclusions and Perspectives

> ”A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.”
>
> *Antoine de Saint-Exupery*

This diploma work concentrated on an Associative Documentation Model and a dynamic documentation process that benefits from associations to software models. We compare now the reached results with the specified goals. The Discussion shows advantages and disadvantaged of this approach to documentation. We close with further developments and the perspectives for future research.

7.1 Discussion

We walk through the chapters and compare the results with our goals and expectations. We are interested in solutions and options, but also in problems and hints that occurred. A final overview investigates the advantages and disadvantages of the chosen approach to manage documentation.

- CHAPTER 2: THE ANALYSIS OF THE RESOURCES
  We started with an analysis of the resources. The form and content of documentation have been investigated. We explored different types of documentation and their role within the software engineering process. We focused on problems that occur while documenting and what has to be considered for any solution.

  We found out that consistency between documentation and the described software is the major hint for good documentation. We saw that technical reasons and logistical or economical requirements cause the majority of the inconsistency problems.

  The analysis of the resources focused on the two specific topics TECHNICAL DOCUMENTATION and FAMIX MODELS as a general model for object-oriented software. We saw that Famix models provide data about the entities, especially the names that represent the entities. On the other side, we observed that technical documentation with an narrow relationship to the software it documents often mention the name of the entities when they refer to some specific parts of

the software. These characteristics of Famix models and technical documentation open the possibility to link Famix models and documentation over the names of the software entities.

The specific results about technical documentation and Famix models lead us to the central documentation problem of synchronization of technical documentation with evolving software. So we get the foundation for an proposal to deal with this problem.

- CHAPTER 3: DEPENDENCIES BETWEEN SOFTWARE MODELS AND DOCUMENTATION

  We investigated in chapter 3 with the Hotdraw and the Visual Works case studies the dependencies between software models and technical documentation. The idea was to detect such dependencies using entity names and looking for them within the text of the technical documentation. This approach is based on the observations of chapter 2. We saw that dependency detection using entity names works for technical documentation as they have a narrow relationship to the software and its structure.

  We tested the dependencies with an application of the dependencies called CHANGE IMPACT PREDICTION. Change Impact Prediction analysed the changes between two software versions and inferred the changes over the dependencies on the documentation. These way, possible change requirement were indicated. We saw that Change Impact Prediction worked well what proved the quality the the dependencies.

  The Change Impact prediction delivered interesting results about the structure of the entity names and the documentation segments. Additional data from the Famix model improved the probability to detect dependecies. On the other side, we saw that small documentation segments let dependencies refer more precise to the documentation. This resulted in a better quality of Change Impact Prediction.

  Dependency detection and Change Impact Prediction were validated by comparing with the real changes between different versions of the technical documentation. This comparison and validation was an human issue.

- CHAPTER 4: ENTITY NAMES AND DEPENDENCIES

  The data extraction considered the detection of Famix entity names. Two aspects are important: How are these data retrieved and how are they represented. The major problem was to find a representation that fits to the meaning of the information and is simultaneously suitable for detection within documentation.

  We explored different variants of representation and found out that none of them can be preferred unconditionally. They all have their advantages and disadvantages. Their is a trade-of between reflecting a comprehensive information and being suitable for detection within documentation.

  The dependencies are the central part of an Associative Documentation Model. The dependencies are by definition open. That means they don't depend on how they are created. Nevertheless, we examined automatic dependency generation. We used name matching as it promises independence from particular structures. The results showed that support is reached this way. But it remains in the developers responsibility to complete, control and validate the dependencies and all services that rely on them.

We extended the dependencies using the inner relationships between the Famix entities. These relationships let us infer additional dependencies that are not detected using the entity names. These extention of the dependency set makes them more independent from the entity names and takes additionally the dependencies between the entities into consideration. This extention is interesting as the inner relationships between the Famix entities are provided and available by the Famix model.

- CHAPTER 5: THE ASSOCIATIVE DOCUMENTATION MODEL

  The Associative Documentation Model composes the information retrieval from software models and the dependency generation to segmented documentation into a consistent model of associated documentation. It describes the required structures to maintain a documentation that is organised as sets of segments. It relates the segments to information from software models. It models the relationship that exists between the documentation and its documented software.

  Change Impact Detection and Impact Detection are two documentation support features that rely on the Associative Documentation Model. Their service quality depends on the ADM models. We explored under which conditions good quality can be expected and what delimits the results. The exploration of the conditions delivers indications to decide where possible ADM model hints for the services Change Detection and Impact Detection must be considered.

- CHAPTER 6: THE ADM DOCUMENTATION PROCESS

  The ADM documentation process describes a dynamic process for creation of documentations. This process builds on the features of ADM. It shows how to create documentation concurrently to the software development process. The core feature is an iterative update cycle that is supported by the segmented documentation structure of ADM.

  Synchronisation of documentation with the documented software is a central target of the ADM documentation process. We saw that the dependencies of ADM support synchronisation: It indicates potential change and update requirements on documentation segments, and it detects software parts that might not been documented yet. The core element of the dynamic documentation process is not a completed, final documentation but sets of documentation segments relating to the software that are subject of iterative updates.

  This approach to documentation consists especially the requirements of documentation within a running software development process. It supports the request for immediate documenting. It forces developers and project managers to see documentation as a integral part of the software development.

7.2 Research Perspectives

The Associative Documentation model provides a basic model for the relationship between software models and documentation models. The core relationship was the central criteria. Further exploration could go into various directions.

7.2.1 Extended Software models

The research on software models is a ongoing discipline. New research results about the structure and the features of software models may open new and additional possibilities for information retrieval and information representation of software models.

- NOT-OBJECT-ORIENTED MODELS
  Lots of software is not object-oriented. We need models that express the structures of such software. Information retrieval from not-object-oriented models and representation of information allows to apply the Associative Documentation Model to such software as well. This would be a major extension of the ADM environment.

- DYNAMIC INFORMATION
  Dynamic Information handle information about the dynamic behaviour of software. This aspect is complete excluded from static software models. There are approaches with simulations to get the dynamic behaviour out of the static description of the software. Within software models one might consider identification of sequences of interaction between objects as objects. Due to polymorphism, not all method invocations con be resolved at compile time. This makes it hard to identify all invocations. The potentially large number of software states increases the number of interaction sequences exponentially. So computation of all interaction sequences is very inefficient.

  Even if one could identify all relevant interaction sequences, its interpretation and the semantic meaning could not be derived easily. So dynamic information might allow the analysis of the invocation structure of a sequence, but it remains a part of interpretation to get its meaning. The structure of an implemented algorithms as a part of the software i.e. can be analysed, but this doesn't deliver information about its meaning and purpose what is the central part of an algorithm.

- INDEPENDENT INFORMATION
  Software includes lots of information that relates to the software as a whole. It can't be referred to a particular part of this software. Examination of such independent information would extend the types of documentation that can be handled. It would move ADM toward non-technical documentation.

7.2.2 Extended Documentation models

ADM handles documentation as sets of segments. This basic schema could be extended towards documentation models.

- DOCUMENTATION MODELS define documentation as an object-oriented model whose entities represent parts of the documentation. The entities define their role within documentation. Formatting is performed according this role but is not fixed definitively.

  Documentation segments of the ADM model can be replaced by documentation model entities. Subject of exploration is how to handle entities and how to handle relations between entities. It would be interesting to see if entity properties improve the dependency generation and if the relations between the entities would help at impact prediction.

7.2.3 Integration of the Associative Documentation Model

A third option to proceed with the ADM documentation model is its integration into existing models. Further exploration could go into two directions: extension and integration of the Associative Documentation Model toward extended software models and towards documentation models.

- INTEGRATION INTO SOFTWARE MODELS requires extensions of the software models for the features of ADM. The relationship to documentation segments and the segments itselves must be expressed within the software model. Subject of exploration is how to do this and to see if its advantageous.
- INTEGRATION INTO DOCUMENTATION MODELS requires the documentation to express the relationship to the software. The question is how to adapt model entities and properties to integrate ADM and to explore if its reasonable

7.3 The Future

The Associative Documentation Model brings two worlds together: Software engineering and documentation engineering. Their often handled independently what leads to the know problems with documentation. ADM show how they can relate to each other. It conducts the advantages of both fields into a concept that links them together. Both software engineering and documentation engineering profit from each other. The Associative Documentation Model considers software engineering and documentation engineering not as independent, but as concurrent processes. It supports the requirement for immediate and consistent documentation within software engineering. The Associative Documentation Model contributes to enforce developers and project managers to see documentation as an integral and important part of software engineering.

Appendix A

The Visual Works data

The following 4 tables contain all data from the Visual Works case study.

Chapter	*Subchapter*											
1	171	10	8	17	10	9	9	8	8	26	26	8
		8	8	8	8	-	-	-	-	-	-	-
2	261	8	27	28	189	9	-	-	-	-	-	-
3	149	19	13	24	8	8	14	14	14	9	8	9
		9	-	-	-	-	-	-	-	-	-	-
4	390	26	13	9	17	9	18	194	22	22	9	9
		9	8	8	9	8	-	-	-	-	-	-
5	86	14	15	10	14	33	-	-	-	-	-	-
6	137	15	26	15	11	43	9	9	9	-	-	-
7	38	9	20	9	-	-	-	-	-	-	-	-
8	126	27	28	28	28	15	-	-	-	-	-	-
9	55	9	10	9	10	9	8	-	-	-	-	-
10	265	26	9	9	10	184	9	9	9	-	-	-
11	92	20	12	32	9	19	-	-	-	-	-	-
12	62	14	9	9	11	19	-	-	-	-	-	-
13	1116	213	48	58	203	184	194	43	43	43	44	43
14	57	17	9	13	9	9	-	-	-	-	-	-
15	145	11	11	11	12	12	11	11	48	18	-	-
16	65	27	11	9	9	9	-	-	-	-	-	-
17	93	11	11	9	11	10	11	10	9	11	-	-
18	93	23	9	9	9	23	9	11	-	-	-	-
19	387	57	8	37	35	33	57	19	70	35	36	-
20	109	9	0		1	9	8	8	8	8	8	8
		17	8	8	9	-	-	-	-	-	-	-
21	43	8	8	10	-	-	-	17	-	-	-	-
22	42	8	9	-	8	8	9	-	-	-	-	-
23	120	9	9	12	10	10	9	8	8	8	10	9
		9	9	-	-	-	-	-	-	-	-	-
24	103	8	8	9	9	8	9	8	9	9	8	9
		9	-	-	-	-	-	-	-	-	-	-
25	209	8	8	9	8	8	10	8	17	8	8	8
		8	9	8	8	16	14	10	9	9	9	9
26	101	9	9	8	9	8	9	8	8	17	8	8
27	46	9	9	9	9	10	-	-	-	-	-	-
28	97	8	9	8	10	8	8	8	8	20	10	-
29	150	10	13	9	9	9	8	9	9	10	14	10
		23	9	8	-	-	-	-	-	-	-	-
30	53	9	8	9	8	9	10	-	-	-	-	-
31	48	9	9	9	9	12	-	-	-	-	-	-

Table A.1: Impact distribution based on fine code granularity

Chapter	*Subchapter*											
1	35.0	2	2	3	2	2	2	2	2	5	5	2
		2	2	2	2	-	-	-	-	-	-	-
2	53.4	2	6	6	39	2	-	-	-	-	-	-
3	30.5	4	3	5	2	2	3	3	3	2	2	2
		2	-	-	-	-	-	-	-	-	-	-
4	79.8	5	3	2	3	2	4	40	4	4	2	2
		2	2	2	2	2	-	-	-	-	-	-
5	17.6	3	3	2	3	7	-	-	-	-	-	-
6	28.0	3	5	3	2	9	2	2	2	-	-	-
7	7.8	2	4	2	-	-	-	-	-	-	-	-
8	25.8	6	6	6	6	3	-	-	-	-	-	-
9	11.2	2	2	2	2	2	2	-	-	-	-	-
10	54.2	5	2	2	2	38	2	2	2	-	-	-
11	18.8	4	2	7	2	4	-	-	-	-	-	-
12	8.8	3	2	2	2	-	-	-	-	-	-	-
13	228.2	44	10	12	42	38	40	9	9	9	9	9
14	11.7	3	2	3	2	2	-	-	-	-	-	-
15	29.7	2	2	2	2	2	2	2	10	4	-	-
16	13.3	6	2	2	2	2	-	-	-	-	-	-
17	19.0	2	2	2	2	2	2	2	2	2	-	-
18	19.0	5	2	2	2	5	2	2	-	-	-	-
19	79.1	12	2	8	7	7	12	4	14	7	7	-
20	22.3	2	-	-	0	2	2	2	2	2	2	2
		3	2	2	2	-	-	-	-	-	-	-
21	8.8	2	2	2	-	-	-	3	-	-	-	-
22	8.6	2	2	-	2	2	2	-	-	-	-	-
23	24.5	2	2	2	2	2	2	2	2	2	2	2
		2	2	-	-	-	-	-	-	-	-	-
24	21.1	2	2	2	2	2	2	2	2	2	2	2
		2	-	-	-	-	-	-	-	-	-	-
25	42.7	2	2	2	2	2	2	2	3	2	2	2
		2	2	2	2	3	3	2	2	2	2	2
26	20.7	2	2	2	2	2	2	2	2	3	2	2
27	9.4	2	2	2	2	2	-	-	-	-	-	-
28	19.8	2	2	2	2	2	2	2	2	4	2	-
29	30.7	2	3	2	2	2	2	2	2	2	3	2
		5	2	2	-	-	-	-	-	-	-	-
30	10.8	2	2	2	2	2	2	-	-	-	-	-
31	9.8	2	2	2	2	2	-	-	-	-	-	-

Table A.2: Part of total impact with fine granularity expressed in per mille

Chapter	*Subchapter*											
1	129	10	8	11	10	9	9	8	8	8	8	8
		8	8	8	8	-	-	-	-	-	-	-
2	52	8	9	10	16	9	-	-	-	-	-	-
3	123	9	13	14	8	8	8	14	14	9	8	9
		9	-	-	-	-	-	-	-	-	-	-
4	151	10	13	9	12	9	8	11	8	11	9	9
		9	8	8	9	8	-	-	-	-	-	-
5	48	9	10	10	9	10	-	-	-	-	-	-
6	86	10	20	9	11	9	9	9	9	-	-	-
7	27	9	9	9	-	-	-	-	-	-	-	-
8	56	11	12	12	12	9	-	-	-	-	-	-
9	55	9	10	9	10	9	8	-	-	-	-	-
10	84	18	9	9	10	11	9	9	9	-	-	-
11	66	15	12	16	9	14	-	-	-	-	-	-
12	60	14	9	9	9	19	-	-	-	-	-	-
13	129	19	9	14	14	11	-	-	-	-	-	-
14	57	17	9	13	9	9	-	-	-	-	-	-
15	84	9	9	9	10	10	9	9	9	10	-	-
16	47	9	11	9	9	9	-	-	-	-	-	-
17	93	11	11	9	11	10	11	10	9	11	-	-
18	65	9	9	9	9	9	9	11	-	-	-	-
19	98	9	8	12	9	11	9	9	11	9	11	-
20	109	9			1	9	8	8	8	8	8	8
		17	8	8	9	-	-	-	-	-	-	-
21	43	8	8	10	-	-	-	17	-	-	-	-
22	42	8	9	-	8	8	9	-	-	-	-	-
23	120	9	9	12	10	10	9	8	8	8	10	9
		9	9	-	-	-	-	-	-	-	-	-
24	103	8	8	9	9	8	9	8	9	9	8	9
		9	-	-	-	-	-	-	-	-	-	-
25	209	8	8	9	8	8	10	8	17	8	8	8
		8	9	8	8	16	14	10	9	9	9	9
26	101	9	9	8	9	8	9	8	8	17	8	8
27	46	9	9	9	9	10	-	-	-	-	-	-
28	85	8	9	8	10	8	8	8	8	8	10	-
29	135	10	13	9	9	9	8	9	9	10	13	10
		9	9	8	-	-	-	-	-	-	-	-
30	53	9	8	9	8	9	10	-	-	-	-	-
31	48	9	9	9	9	12	-	-	-	-	-	-

Table A.3: Impact distribution based on coarse code granularity

Chapter	*Subchapter*											
1	49.5	4	3	4	4	3	3	3	3	3	3	3
		3	3	3	3	-	-	-	-	-	-	-
2	20.0	3	3	4	6	3	-	-	-	-	-	-
3	47.2	3	5	5	3	3	3	5	5	3	3	3
		3	-	-	-	-	-	-	-	-	-	-
4	58.0	4	5	3	5	3	3	4	3	4	3	3
		3	3	3	3	3	-	-	-	-	-	-
5	18.4	3	4	4	3	4	-	-	-	-	-	-
6	33.0	4	8	3	4	3	3	3	3	-	-	-
7	10.4	3	3	3	-	-	-	-	-	-	-	-
8	21.5	4	5	5	5	3	-	-	-	-	-	-
9	21.1	3	4	3	4	3	3	-	-	-	-	-
10	32.3	7	3	3	4	4	3	3	3	-	-	-
11	25.3	6	5	6	3	5	-	-	-	-	-	-
12	23.0	5	3	3	3	7	-	-	-	-	-	-
13	49.5	7	3	5	3	3	-	-	-	-	-	-
14	21.9	7	3	5	3	3	-	-	-	-	-	-
15	32.3	3	3	3	4	4	3	3	3	4	-	-
16	18.0	3	4	3	3	3	-	-	-	-	-	-
17	35.7	4	4	3	4	4	4	4	3	4	-	-
18	25.0	3	3	3	3	3	3	4	-	-	-	-
19	37.6	3	3	5	3	4	3	3	4	3	4	-
20	41.9	3	-	-	0	3	3	3	3	3	3	3
		7	3	3	3	-	-	-	-	-	-	-
21	16.5	3	3	4	-	-	-	7	-	-	-	-
22	16.1	3	3	-	3	3	3	-	-	-	-	-
23	46.1	3	3	5	4	4	3	3	3	3	4	3
		3	3	-	-	-	-	-	-	-	-	-
24	39.6	3	3	3	3	3	3	3	3	3	3	3
		3	-	-	-	-	-	-	-	-	-	-
25	80.3	3	3	3	3	3	4	3	7	3	3	3
		3	3	3	3	6	5	4	3	3	3	3
26	38.8	3	3	3	3	3	3	3	3	7	3	3
27	17.7	3	3	3	3	4	-	-	-	-	-	-
28	32.6	3	3	3	4	3	3	3	3	3	4	-
29	51.8	4	5	3	3	3	3	3	3	4	5	4
		3	3	3	-	-	-	-	-	-	-	-
30	20.4	3	3	3	3	3	4	-	-	-	-	-
31	18.4	3	3	3	3	5	-	-	-	-	-	-

Table A.4: Part of total impact with coarse granularity expressed in per mille.

Appendix B

Famix model structure

B.1 Famix Interdependencies

B.1.1 Explicit relationship by Association

Association	*Entity1*	*Entity2*	*Relationship*
InheritanceDefinition	Class	Class	superclass
	Class	Class	subclass
Invocation	Behavioural Entity	Behavioural Entity	invokes
	Behavioural Entity	Behavioural Entity	invokedBy
Access	BehaviouralEntity	StructuralEntity	accesses
	StructuralEntity	BehaviouralEntity	accessedIn

Table B.1: Explicit Relationship

Behavioural Entity = { Function, Method }
Structural Entity = { Attribute, ImplicitVariable, LocalVariable,
GlobalVariable, FormalParameter }

B.1.2 Implicit relationship by Property

Association	*Entity1*	*Entity2*	*Relationship*
Aggregation	Method	Class	belongsToClass
	Attribute	Class	belongsToClass
	ImplicitVariable	Class	belongsToClass
	LocalVariable	Method	belongsToMethod
	FormalParameter	Method	belongsToMethod
Aggregation	Invocation	Argument	hasArgument
	SimpleAccess	Access	hasAccess

Table B.2: Implicit Relationship

Argument = { ComplexExpression, SimpleAccess }

Appendix C

Example of the ADM documentation scenario

This example shows the update process of the documentation about the Famix class ENTITY.

C.1 Java Source Code

Java Implementation of an Famix class ENTITY, called FMCOREFAMOOSABSTRACTENTITY. It is the older implementaton version 1.

```
// Classname: FamoosAbstractEntity
// Language:  java, jkd, v.1.1.4
//
// Author:    Fredi Frank
// Date:      12.03.1999
//
// Purpose:   FamoosAbstractEntity implementation
//
// Notes:     Attributes are OK
//            Inheritance OK

import Stream;
import OrderedCollection;
import FamoosHelpCDIF;
import FamoosHelpPROP;

public class fmCoreFamoosAbstractEntity extends fmCoreFamoosAbstractObject {

  // Instance Variables

     private String name;
     private String uniqueName;

  // Constructor

     public fmCoreFamoosAbstractEntity() {
       super();
       name = "";
       uniqueName = "";
```

```
    };

  // Protocoll for read/write Attributes

    public void name(String aName) {
      name = aName;
    };

    public String name() {
      return name;
    };

    public void uniqueName(String aName) {
      uniqueName = aName;
    };

    public String uniqueName() {
      return uniqueName;
    };

  // Protocoll for Enumeration

    public FamoosHelpPROP enumPropertyNames() {
      FamoosHelpPROP p = super.enumPropertyNames();
      p.set("name");
      p.set("uniqueName");
      return p;
    };

  // Protocoll for printing

    public void print(FamoosHelpCDIF p) {
      super.print(p);
      p.setBody("name "+name);
      p.setBody("uniqueName "+uniqueName);
    };

}
```

Java Implementation of an Famix class ENTITY, called ENTITY. It is the newer implementation version 2.

```
// Classname: Entity
// Language:  java, jkd, v.1.1.4
//
// Author:    Fredi Frank
// Date:      25.05.1999
//
// Purpose:   Entity implementation
//
// Notes:     -

  import java.util.Enumeration;
  import com.objectspace.jgl.SList;
  import BehaviouralEntity;
  import StructuralEntity;
  import Class;


  public class Entity extends FamoosObject
```

```
                          implements EntityInterface {

    // Constants Resources

       private final String NULLWERT = "<NULL>";
       private final String ME = "Entity";
       private final String NAME = "name";
       private final boolean M_NAME = true;
       private final String D_NAME = NULLWERT;
       private final String D_UNIQUENAME = NULLWERT;

    // Instance variables

       private String name;
       private String uniqueName;

    // Constructor

       public Entity() {
         super();
         name = D_NAME;
         uniqueName = D_UNIQUENAME;
       };

    // Protocoll for read/write

       public void name(String na) {
         name = na;
       };

       public String name() {
         return name;
       };

       public void uniqueName(String aName) {
         uniqueName = aName;
       };

       public String uniqueName() {
         return uniqueName;
       };

    // Protocoll for Aggreations ++++++++++++++++++++++++++++++++++++++++++

       public void property(String name, String value) {
         if(name.equals(NAME))   { name(value); }   else {
           super.property(name,value);
         };
       };

       public String property(String name) {
         if(name.equals(NAME))   { return name(); }   else {
           return super.property(name);
         };
       };

       public void clearProperty(String name) {
         if(name.equals(NAME))   { name(D_NAME); }    else {
           super.clearProperty(name);
         };
       };
```

```
    // Protocoll for information

       public String me() {
         return ME;
       };

    // Protocoll for queries

       public boolean isMandatory(String prop) {
         boolean b = false;
         b = super.isMandatory(prop);
         if(prop.equals(NAME))  b = M_NAME;
         return b;
       };

       public boolean isOptional(String prop) {
         boolean b = false;
         if(prop.equals(NAME))  b = true;
         return b;
       };

       public boolean isNull(String prop) {
         boolean b = true;
         b = super.isNull(prop);
         if((prop.equals(NAME))&&(!name().equals(NULLWERT))) b = false;
         return b;
       };

       public boolean isMinimal() {
         // super nicht vergessen.
         return((isNull(NAME)));
       };

    // Extended Protocoll

       public Enumeration propertyNames() {
         SList s = new SList();
         s.add(NAME);
         Enumeration e = super.propertyNames();
         while(e.hasMoreElements()) s.add((String)e.nextElement());
         return s.elements();
       };

       public String CDIF(String id) {
         String h = "";
         if((!isNull(NAME))||(isMandatory(NAME)))
           h = h +"\t(" +NAME          +" "+name()  +")\n";
         h = h+super.CDIF(id);
         return h;
       };

   // Registering ++++++++++++++++++++++++++++++++++++++++++++++++++++++++

       public void register(String id, ObjectSet es, ObjectSet bu) {
         if(this instanceof BehaviouralEntity)
          ((BehaviouralEntity)this).register(id,es,bu);
         if(this instanceof StructuralEntity)
          ((StructuralEntity)this).register(id,es,bu);
         if(this instanceof Class)
          ((Class)this).register(id,es,bu);
        };
```

```
        public void release(String id, ObjectSet es, ObjectSet bu) {
          if(this instanceof BehaviouralEntity)
           ((BehaviouralEntity)this).release(id,es,bu);
          if(this instanceof StructuralEntity)
           ((StructuralEntity)this).release(id,es,bu);
          if(this instanceof Class)
           ((Class)this).release(id,es,bu);
         };

        public void registerProperty(String name, String value) {
          if(this instanceof BehaviouralEntity)
           ((BehaviouralEntity)this).registerProperty(name,value);
          if(this instanceof StructuralEntity)
           ((StructuralEntity)this).registerProperty(name,value);
          if(this instanceof Class)
           ((Class)this).registerProperty(name,value);
        };

        public void releaseProperty(String name) {
          if(this instanceof BehaviouralEntity)
           ((BehaviouralEntity)this).releaseProperty(name);
          if(this instanceof StructuralEntity)
           ((StructuralEntity)this).releaseProperty(name);
          if(this instanceof Class)
           ((Class)this).releaseProperty(name);
        };

}
```

C.2 Famix Models

The Famix Model of fmCoreFamoosAbstractEntity. This is the model of the older implementation version.

```
(Class FM0001
  (name "fmCoreFamoosAbstractEntity")
  (isAbstract -FALSE-)
)

(Method FM0002
  (name "fmCoreFamoosAbstractEntity")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (isConstructor -TRUE-)
  (isAbstract -FALSE-)
)

(Method FM0003
  (name "name")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (declaredReturnType "void")
  (uniqueName "fmCoreFamoosAbstractEntity.name(String)")
)

(Method FM0004
  (name "name")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (declaredReturnType "String")
  (uniqueName "fmCoreFamoosAbstractEntity.name()")
)
```

```
(Method FM0005
  (name "uniqueName")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (declaredReturnType "void")
  (uniqueName "fmCoreFamoosAbstractEntity.uniqueName(String)")
)

(Method FM0006
  (name "uniqueName")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (declaredReturnType "String")
  (uniqueName "fmCoreFamoosAbstractEntity.uniqueName()")
)

(Method FM0007
  (name "enumPropertyNames")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (declaredReturnType "FamoosHelpPROP")
  (uniqueName "fmCoreFamoosAbstractEntity.enumPropertyNames()")
)

(Method FM0008
  (name "print")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (declaredReturnType "void")
  (uniqueName "fmCoreFamoosAbstractEntity.print(FamoosHelpCDIF)")
)

(Attribute FM0009
  (name "name")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "fmCoreFamoosAbstractEntity.name")
 )

(Attribute FM0010
  (name "uniqueName")
  (belongsToClass "fmCoreFamoosAbstractEntity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "fmCoreFamoosAbstractEntity.uniqueName")
)

(InheritanceDefinition FM0011
  (subclass "fmCoreFamoosAbstractEntity")
  (superclass "fmCoreFamoosAbstractObject")
)
```

The Famix Model of Entity. This is the model of the newer implementation version.

```
(Class FM0001
  (name "Entity")
  (isAbstract -FALSE-)
)

(Method FM0002
  (name "Entity")
  (belongsToClass "Entity")
  (isConstructor -TRUE-)
```

```
  (isAbstract -FALSE-)
)

(Method FM0003
  (name "name")
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.name(String)")
)

(Method FM0004
  (name "name")
  (belongsToClass "Entity")
  (declaredReturnType "String")
  (uniqueName "Entity.name()")
)

(Method FM0005
  (name "uniqueName")
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.uniqueName(String)")
)

(Method FM0006
  (name "uniqueName")
  (belongsToClass "Entity")
  (declaredReturnType "String")
  (uniqueName "Entity.uniqueName()")
)

(Method FM0007
  (name "property")
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.property(String,String)")
)

(Method FM0008
  (name "property")
  (belongsToClass "Entity")
  (declaredReturnType "String")
  (uniqueName "Entity.property(String)")
)

(Method FM0009
  (name "clearProperty")
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.clearProperty(String)")
)

(Method FM0010
  (name "me")
  (belongsToClass "Entity")
  (declaredReturnType "String")
  (uniqueName "Entity.me()")
)

(Method FM0011
  (name "isMandatory")
  (belongsToClass "Entity")
```

```
  (declaredReturnType "boolean")
  (uniqueName "Entity.isMandatory(String)")
)

(Method FM0012
  (name "isOptional")
  (belongsToClass "Entity")
  (declaredReturnType "boolean")
  (uniqueName "Entity.isOptional(String)")
)

(Method FM0013
  (name "isNull")
  (belongsToClass "Entity")
  (declaredReturnType "boolean")
  (uniqueName "Entity.isNull(String)")
)

(Method FM0014
  (name "isMinimal")
  (belongsToClass "Entity")
  (declaredReturnType "boolean")
  (uniqueName "Entity.isMinimal()")
)

(Method FM0015
  (name "propertyNames")
  (belongsToClass "Entity")
  (declaredReturnType "Enumeration")
  (uniqueName "Entity.propertyNames()")
)

(Method FM0016
  (name "CDIF")
  (belongsToClass "Entity")
  (declaredReturnType "String")
  (uniqueName "Entity.CDIF(String)")
)

(Method FM0017
  (name "register")
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.register(String,ObjectSet,ObjectSet)")
)

(Method FM0018
  (name "release")
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.release(String,ObjectSet,ObjectSet")
)

(Method FM0019
  (name "registerProperty")
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.registerProperty(String,String)")
)

(Method FM0020
  (name "releaseProperty")
```

```
  (belongsToClass "Entity")
  (declaredReturnType "void")
  (uniqueName "Entity.releaseProperty(String)")
)

(Attribute FM0021
  (name "NULLWERT")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "Entity.NULLWERT")
)

(Attribute FM0022
  (name "ME")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "Entity.ME")
)

(Attribute FM0023
  (name "NAME")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "Entity.NAME")
)

(Attribute FM0024
  (name "M_NAME")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "boolean")
  (uniqueName "Entity.M_NAME")
)

(Attribute FM0025
  (name "D_NAME")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "Entity.D_NAME")
)

(Attribute FM0026
  (name "D_UNIQUENAME")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "Entity.D_UNIQUENAME")
)

(Attribute FM0027
  (name "name")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "Entity.name")
)

(Attribute FM0028
```

```
  (name "uniqueName")
  (belongsToClass "Entity")
  (accessControlQualifier "private")
  (declaredType "String")
  (uniqueName "Entity.uniqueName")
)

(InheritanceDefinition FM0029
  (subclass "Entity")
  (superclass "FamoosObject")
)
```

C.3 Entity name representation

The list of names of the fmCoreFamoosAbstractEntity implementation of version 1.

```
("Class",     "fmCoreFamoosAbstractEntity",
              "fmCoreFamoosAbstractEntity")
("Method",    "fmCoreFamoosAbstractEntity",
              "fmCoreFamoosAbstractEntity.fmCoreFamoosAbstractEntity()")
("Method",    "name",
              "fmCoreFamoosAbstractEntity.name(String)")
("Method",    "name",
              "fmCoreFamoosAbstractEntity.name()")
("Method",    "uniqueName",
              "fmCoreFamoosAbstractEntity.uniqueName(String)")
("Method",    "uniqueName",
              "fmCoreFamoosAbstractEntity.uniqueName()")
("Method",    "enumPropertyNames",
              "fmCoreFamoosAbstractEntity.enumPropertyNames()")
("Method",    "print",
              "fmCoreFamoosAbstractEntity.print(FamoosHelpCDIF)")
("Attribute", "name",
              "fmCoreFamoosAbstractEntity.name")
("Attribute", "uniqueName",
              "fmCoreFamoosAbstractEntity.uniqueName")
```

The list of names of the Entity implementation of version 2.

```
("Class",     "Entity",           "Entity")
("Method",    "Entity",           "Entity.Entity()")
("Method",    "name",             "Entity.name(String)")
("Method",    "name",             "Entity.name()")
("Method",    "uniqueName",       "Entity.uniqueName(String)")
("Method",    "uniqueName",       "Entity.uniqueName()")
("Method",    "property",         "Entity.property(String,String)")
("Method",    "property",         "Entity.property(String)")
("Method",    "clearProperty",    "Entity.clearProperty(String)")
("Method",    "me",               "Entity.me()")
("Method",    "isMandatory",      "Entity.isMandatory(String)")
("Method",    "isOptional",       "Entity.isOptional(String)")
("Method",    "isNull",           "Entity.isNull(String)")
("Method",    "isMinimal",        "Entity.isMinimal()")
("Method",    "propertyNames",    "Entity.propertyNames()")
("Method",    "CDIF",             "Entity.CDIF(String)")
("Method",    "register",         "Entity.register(String,ObjectSet,ObjectSet)")
("Method",    "release",          "Entity.release(String,ObjectSet,ObjectSet)")
```

```
("Method",    "registerProperty", "Entity.registerProperty(String,String)")
("Method",    "releaseProperty",  "Entity.releaseProperty(String)")
("Attribute", "NULLWERT",         "Entity.NULLWERT")
("Attribute", "ME",               "Entity.ME")
("Attribute", "NAME",             "Entity.NAME")
("Attribute", "M_NAME",           "Entity.M_NAME")
("Attribute", "D_NAME",           "Entity.D_NAME")
("Attribute", "D_UNIQUENAME",     "Entity.D_UNIQUENAME")
("Attribute", "name",             "Entity.name")
("Attribute", "uniqueName",       "Entity.uniqueName")
```

C.4 Change from fmCoreFamoosAbstractEntity to Entity

```
(42, "Class",     "fmCoreFamoosAbstractEntity",
                  "fmCoreFamoosAbstractEntity",
     "removed",   "version 10")
(43, "Method",    "fmCoreFamoosAbstractEntity",
                  "fmCoreFamoosAbstractEntity.fmCoreFamoosAbstractEntity()",
     "removed",   "version 10")
(44, "Method",    "enumPropertyNames",
                  "fmCoreFamoosAbstractEntity.enumPropertyNames()",
     "removed",   "version 10")
(45, "Method",    "print",
                  "fmCoreFamoosAbstractEntity.print(FamoosHelpCDIF)",
     "removed",   "version 10")
(46, "Class",     "Entity",           "Entity",
     "added",     "version20")
(47, "Method",    "Entity",           "Entity.Entity()",
     "added",     "version20")
(48, "Method",    "property",         "Entity.property(String,String)",
     "added",     "version20")
(49, "Method",    "property",         "Entity.property(String)",
     "added",     "version20")
(50, "Method",    "clearProperty",    "Entity.clearProperty(String)",
     "added",     "version20")
(51, "Method",    "me",               "Entity.me()",
     "added",     "version20")
(52, "Method",    "isMandatory",      "Entity.isMandatory(String)",
     "added",     "version20")
(53, "Method",    "isOptional",       "Entity.isOptional(String)",
     "added",     "version20")
(54, "Method",    "isNull",           "Entity.isNull(String)",
     "added",     "version20")
(55, "Method",    "isMinimal",        "Entity.isMinimal()",
     "added",     "version20")
(56, "Method",    "uniqueName",       "Entity.uniqueName()",
     "added",     "version20")
(57, "Method",    "propertyNames",    "Entity.propertyNames()",
     "added",     "version20")
(58, "Method",    "CDIF",             "Entity.CDIF(String)",
     "added",     "version20")
(59, "Method",    "register",         "Entity.register(String,ObjectSet,ObjectSet)",
     "added",     "version20")
(60, "Method",    "release",          "Entity.release(String,ObjectSet,ObjectSet)",
     "added",     "version20")
(61, "Method",    "registerProperty", "Entity.registerProperty(String,String)",
     "added",     "version20")
```

```
(62, "Method",    "releaseProperty",  "Entity.releaseProperty(String)",
     "added",     "version20")
(63, "Attribute", "NULLWERT",         "Entity.NULLWERT",
     "added",     "version20")
(64, "Attribute", "ME",               "Entity.ME",
     "added",     "version20")
(65, "Attribute", "NAME",             "Entity.NAME",
     "added",     "version20")
(66, "Attribute", "M_NAME",           "Entity.M_NAME",
     "added",     "version20")
(67, "Attribute", "D_NAME",           "Entity.D_NAME",
     "added",     "version20")
(68, "Attribute", "D_UNIQUENAME",     "Entity.D_UNIQUENAME",
     "added",     "version20")
```

C.5 Documentation segments about the implementation of fmCoreFamoosAbstractEntity

```
(section 0122
  (version 10)
  (name "entity1")
  (order 122)
  (text "fmCoreFamoosEntity is an Java-implementation of the basic Famix
         class Entity. Entity provides the common properties of the structural
         and behavioural entities.")
)

(section 0123
  (version 10)
  (name "entity2")
  (order 123)
  (text "fmCoreFamoosEntity inherits from fmCoreFamoosAbstractObject.")
)

(section 0124
  (version 10)
  (name "entity3")
  (order 124)
  (text "fmCoreFamoosEntity supports the standard properties name and
         uniqueName. The constructor initializes both properties with
         empty Strings (""). The properties are accessed by the methods
         name and uniqueName.")
)

(section 0125
  (version 10)
  (name "entity4")
  (order 125)
  (text "The method enumPropertyNames enumerates the properties of this
         class. A call to the same method of the superclass ensures the
         enumeration of the superclass properties as well.")
)

(section 0126
  (version 10)
  (name "entity5")
  (order 126)
  (text "The method print generates a CDIF-String that represents the class.
```

```
        This method is usefull for conversion of an Java-implementation of
        an Famix model into an CDIF-description of that model.")
)
```

C.6 Impact and Addition

Change Impact Detection determines the following impact on the documentation segments related to the class fmCoreFamoosAbstractEntity.

```
(entity1, 42, 43, 46, 47)
(entity2, 42, 43)
(entity3, 42, 43)
(entity4, 44)
(entity5, 45, 58)
```

This changes are not detected within the documentation segment related to the class fmCoreFamoosAbstractEntity. This changes are probale to be effective for additions to the documentation.

```
(48, "Method",    "property",         "Entity.property(String,String)",
     "added",     "version20")
(49, "Method",    "property",         "Entity.property(String)",
     "added",     "version20")
(50, "Method",    "clearProperty",    "Entity.clearProperty(String)",
     "added",     "version20")
(51, "Method",    "me",               "Entity.me()",
     "added",     "version20")
(52, "Method",    "isMandatory",      "Entity.isMandatory(String)",
     "added",     "version20")
(53, "Method",    "isOptional",       "Entity.isOptional(String)",
     "added",     "version20")
(54, "Method",    "isNull",           "Entity.isNull(String)",
     "added",     "version20")
(55, "Method",    "isMinimal",        "Entity.isMinimal()",
     "added",     "version20")
(56, "Method",    "uniqueName",       "Entity.uniqueName()",
     "added",     "version20")
(57, "Method",    "propertyNames",    "Entity.propertyNames()",
     "added",     "version20")
(59, "Method",    "register",         "Entity.register(String,ObjectSet,ObjectSet)",
     "added",     "version20")
(60, "Method",    "release",          "Entity.release(String,ObjectSet,ObjectSet)",
     "added",     "version20")
(61, "Method",    "registerProperty", "Entity.registerProperty(String,String)",
     "added",     "version20")
(62, "Method",    "releaseProperty",  "Entity.releaseProperty(String)",
     "added",     "version20")
(63, "Attribute", "NULLWERT",         "Entity.NULLWERT",
     "added",     "version20")
(64, "Attribute", "ME",               "Entity.ME",
     "added",     "version20")
(65, "Attribute", "NAME",             "Entity.NAME",
     "added",     "version20")
(66, "Attribute", "M_NAME",           "Entity.M_NAME",
     "added",     "version20")
(67, "Attribute", "D_NAME",           "Entity.D_NAME",
```

```
     "added",     "version20")
(68, "Attribute", "D_UNIQUENAME",     "Entity.D_UNIQUENAME",
     "added",     "version20")
```

C.7 Updated documentation segments of Entity

The segments are updated according the detected impact of the changes.

```
(section 0122
  (version 20)
  (name "entity1")
  (order 122)
  (text "Entity is an Java-implementation of the basic Famix
         class Entity. Entity provides the common properties of the structural
         and behavioural entities.")
)

(section 0123
  (version 20)
  (name "entity2")
  (order 216)
  (text "Entity inherits from FamoosObject.")
)

(section 0124
  (version 20)
  (name "entity3")
  (order 217)
  (text "Entity supports the standard properties name and
         uniqueName. The constructor initializes both properties with
         default values. The properties are accessed by the methods
         name and uniqueName.")
)

(section 0125
  (version 20)
  (name "entity4")
  (order 221)
  (text "The method PropertyNames enumerates the properties of this
         class. A call to the same method of the superclass ensures the
         enumeration of the superclass properties as well.")
)

(section 0126
  (version 20)
  (name "entity5")
  (order 222)
  (text "The method CDIF generates a CDIF-String that represents the class.
         This method is usefull for conversion of an Java-implementation of
         an Famix model into an CDIF-description of that model.")
)

(section 0210
  (version 20)
  (name "entity6")
  (order 218)
  (text "The access-methods property and the method clearProperty provide
         access to the properties. The methods decides if it is a standard
         property or an optional additional property.")
```

```
)

(section 0211
  (version 20)
  (order 219)
  (text "The method me gives access to the implementation class itself.")
)

(section 0212
  (version 20)
  (order 220)
  (text "The methods isMandatory, isOptional, isNull and isMinimal provide
         informatation about the state of the properties.")
)

(section 0213
  (version 20)
  (order 223)
  (text "The methods register, release, registerProperty and releaseProperty
         register the class and properties into the Famix model.")
)
```

C.8 Updated Documentation of Entity

The updated documentation about the implementation of Entity looks like this.

```
Entity is an Java-implementation of the basic Famix
class Entity. Entity provides the common properties of the structural
and behavioural entities.
Entity inherits from FamoosObject.

Entity supports the standard properties name and
uniqueName. The constructor initializes both properties with
default values. The properties are accessed by the methods
name and uniqueName.

The access-methods property and the method clearProperty provide
access to the properties. The methods decides if it is a standard
property or an optional additional property.

The method me gives access to the implementation class itself.

The methods isMandatory, isOptional, isNull and isMinimal provide
informatation about the state of the properties.

The method PropertyNames enumerates the properties of this
class. A call to the same method of the superclass ensures the
enumeration of the superclass properties as well.

The method CDIF generates a CDIF-String that represents the class.
This method is usefull for conversion of an Java-implementation of
an Famix model into an CDIF-description of that model.

The methods register, release, registerProperty and releaseProperty
register the class and properties into the Famix model.
```

Bibliography

[App94] Wolfgang Appelt. *TeX für Fortgeschrittene*. Addison-Wesley, 1994. 2. erw. Aufl.

[BJ94] Kent Beck and Ralph Johnson. Patterns generate architectures. In R. Pareschi M. Tokoro, editor, *Proceedings ECOOP'94*, pages 139–149. Springer-Verlag, Bologna, Italy, July 1994.

[Boo96] Grady Booch. *Object Solutions*. Addison-Wesley, 1996.

[Bra95] John Michael Brant. Hotdraw. Master's thesis, Gratuate College of the Univrsity of Illinois at Urbana-Campaign, Urbana, Illinois, 1995.

[Bra99a] Tim Bray. An mcf tutorial. Internet Webpages, April 1999. http://www.textuality.com/mcf/MCF-tutorial.html.

[Bra99b] Tim Bray. Roads. Internet Webpages, April 1999. http://www.textuality.com/mcf/MCF-tutorial.htm.

[Buc97] Jürgen Buchner. Hotdoc. Ein Ueberblick. Internet Webpages, June 1997. http://www.pu.informatik.tu-darmstadt.de/Projekte/HotDoc/HotDoc.html.

[Buc99] Jürgen Buchner. Hotdoc, Ein Anwendungsrahmen zum Aufbau von Dokumenten aus Bausteinen. Internet Webpages, September 1999. http://www.pu.informatik.tu-darmstadt.de/Projekte/HotDoc/.

[Cor98] Rational Software Corporation. Rational software. Internet Webpages, 1998. http://www.rational.com.

[DBL95] Yuichi Nakamura Danny B. Lange. Interactive visualisation of design patterns can help in framework understanding. 1995.

[DG95] John Ockerbloom David Garlan, Robert Allen. Architectural mismatch: Why reuse is so hard. volume 12, pages 17–26E. IEEE Software, November 1995.

[Fow97] Martin Fowler. *UML Destilled*. Addison-Wesley, 1997.

[GB99] Ivar Jacobson Grady Booch, James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[GCM97] David Notkin Gail C. Murphy. Reengineering with reflexion models: A case study. *Computing Practices*, 1997.

[GCML97] David Notkin Gail C. Murphy and Erica S.-C. Lan. An empirical study of static call graph extractors. Technical report, Department of Computer Science & Engineering, 1997.

[HEH+96] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to CARE tools. In *Automated Software Engineering, Vol. 3 Nos 1/2, June 1996*. 1996.

[IJ97] Patrik Jonsson Ivar Jocobson, Martin Griss. *Software Reuse*. Adison Wesley Longman, 1997.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings ECOOP'92*, volume 27, pages 63–76. ACM SIGPLAN Notices, October 1992.

[JQNK94] Andre Engberts Jim Q. Ning and W. (Voytek) Kozaczynski. Automated support for lecacy code understanding. *Communications of the ACM*, 37(5), May 1994.

[JR99] Ivar Jacobson James Rumbaugh, Grady Booch. *Unified Modeling Language Reference Manual*. Adison-Wesley Longman, 1999.

[Klu95] Rainer Klute. *Das World Wide Web*. Addison-Wesley, 1995.

[Knu91] Donald E. Knuth. *Computers & Typesetting*. Addison-Wesley, 1991.

[LMK94] Russel Brand Scott Burson Lawrence Markosian, Philip Newcomb and Ted Kitzmiller. Using and enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5), May 1994.

[LMW96] James H. Cross II Linda M. Wills. Recent trends and open issues in reverse engineering. *Automated Software Engineering*, 3:165–172, 1996.

[Mey90] Bertrand Meyer. *Objektorientierte Softwareentwicklung*. Carl Hanser Verlag, Prentice-Hall International Inc., 1990.

[MN] Gail C. Murphy and David Notkin. Lightweight source model extraction. Technical report, Department of Computer Science & Engineering, University of Washington.

[Mul97] Pierre-Alain Muller. *Modelisation object avec UML*. Eyrolles, 1997.

[PB94] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5), May 1994.

[Qua98] Terry Quatrani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.

[REJ93] William F. Opdyke Ralph E. Johnson. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium, Lecture Notes in Computer Science*, volume 742, pages 264–278. Springer-Verlag, November 1993.

[Sch91] Norbert Schwarz. *Einführung in TeX*. Addison-Wesley, 1991. 3. "uberarbeitete Auflage.

[SDS98] Sander Tichelaar Serge Demeyer and Patrick Steyaert. Definition of a common exchange model. Technical report, University of Berne, July 1998.

[Som92] Ian Sommerville. *Software Engineering*. Addison-Wesley, fourth edition, 1992.

[SRC91] Chris F. Kemerer Shyam R. Chidamber. Towards a metric suite for object oriented design. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 197–211, November 1991.

[TD98] Sander Tichelaar and Serge Demeyer. An exchange model for reengineering tools. *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, July 1998.