Diploma Thesis

# Rule Based Software Documentation – Documenting The Collaboration Aspect Of Software Systems


submitted by

Andre Albert

born 14.11.1982 in Löbau

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: Dipl.-Inf. Andreas Bartho
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann



Submitted January 31, 2008

# Abstract

Structuring big software systems into reusable building blocks has proven to be an adequate mean to accomplish the increasing complexity prevailing in todays software projects. In such a scenario 'no object is an island' and relations between participants of these mini architectures are crucial to understand the build-up of the whole system. However, the collaboration aspect of a software system cannot be adequately documented using inline comments as part of the source code. This is mainly because the scope of such a comment is limited to one (following) program construct, the different collaboration contexts of the observed program construct are yet disregarded and references to participants can be only provided literally in an informal manner.

This thesis contributes a concept of a novel documentation approach that allows to adequately document object collaborations in software systems. A key issue is the separation of covered source code and related comments into different documents. A formal description of the collaboration is then needed to glue both concepts together and applies an *M-to-N* relationship between programming code and comments. When browsing source code in an editor, documentation entries fade in context sensitive. In case of multiple matching documentation entries, we will elaborate a set of heuristics to rank resulting entries by their relevance. Furthermore, an Eclipse plug-in utilizing the concepts of this work will be implemented to demonstrate the usage of this new form of an internal software documentation. As a proof of concept, the tool has been tested in a case study covering three different sized software projects with documentation concerning their collaboration aspects.

# Contents

# List of Figures

# List of Tables

# Listings

# 1 Introduction

Within the last decades, software projects have grown up remarkable. To actually accomplish the resulting complexity, humans usually try to raise the level of abstraction. So programming languages evolved from being strictly imperative to higher level languages. A well established result of this trend is object oriented programming, as becoming popular by the language Smalltalk in 1980. Object oriented programming allows the developer to abstract things from the problem domain to corresponding software objects.

This evolvement reduced the effect of so called "spaghetti code", since it allows a much more structured approach. However — when talking about big software systems — this alone is not a universal remedy at all. Instead of "spaghetti code" we now might have something like "tortellini code", which might be also hard to maintain. To actually manage such object oriented systems, architects again raised the level of abstraction. They often think in idioms which are conceptually above sole class-level concepts. An evidence for this trend might be the ample success of the GoF book *"Design Patterns: elements of reusable object-oriented software"* [GHJV95]. These presented patterns (respectively software patterns in general) can be understood as design building blocks providing "a proven solution to a recurring problem within a certain context" [App00]. Such reusable mini architectures consist of a set of related participants where each one fulfills assigned responsibilities (plays a certain role). In conclusion, today the collaboration aspect of a software system is a crucial factor covering all phases of a software engineering process.

**But what about documentation?** A similar evolvement is not recognizable. When commenting source code, programmers seem to be rather conservative. Inline source code comments — which fit well to the imperative programming paradigm — are still widely favored by programmers when commenting code.

Indeed, commenting code using inline comments is really simple, but it currently cannot cope with the progress presented above. Due to the fact that these comments are directly attached to one particular source code fragment, their scope is limited only to the observed fragment. In particular, it is not feasible to document a program construct in context to different collaborations with other participating elements. Therefore, inline comments are not well suited to document collaborations adequately.

Instead of inline comments, some argue that UML diagrams (especially class diagrams) might be a good solution to document the architecture of a software project. Obviously, they have a point, and this work does not plan to replace such diagrams. Nevertheless, class diagrams can grow up fast and are sometimes hardly traceable. Furthermore, UML class diagrams are not

reducible and disallow an adequate *divide-and-conquer* principle to handle complex collaboration phenomena in a controlled manner. Moreover, these charts cannot represent the former mentioned 'design building blocks' explicitly. In addition, class models cannot document an object collaboration task in general. Indeed, it is possible to manually mark special classes that together fulfill a certain collaboration (and assign some commentary-nodes), but in this scenario only one concrete occurrence of the collaboration task is covered – but not the collaboration in general. Finally, UML diagrams are not close enough to the implementation phase, since they are usually not focussed to the current context (such as the current cursor position in an editor). Even if, this information is limited to just highlighting the current class, but it cannot directly clarify the important relations to other building blocks.

In conclusion, there might be a need for a novel documentation approach which is very close to the implementation phase and is able to cover the crucial mini architectures properly. This approach allows us to document source code in context to a collaboration with other source code fragments.

## 1.1 Problem specification

Writing documentation was never considered really popular by developers. Regarded as an unproductive process, developers sometimes even omitted it. But even if they write documentation, studies such as [LSF03, KM01] state that these annotations are rarely updated and sometimes may become out of date.

Nevertheless, documentation becomes more and more important. "Inadequate or missing documentation is a major contributor to the high cost of software maintenance and to the distaste for software maintenance work" [MM83]. Regarding software maintenance, [KC02] expects that "50% of a programmer's time is spent trying to understand existing code." Keeping in mind that "the maintenance phase accounts for over 60% of the development time-line" [Ves04], tool support for software comprehension is therefore an import factor to reduce costs in current software projects.

The introduction section already points out the strong relationships between collaborating source code elements inside a mini architecture. Every hint or advice might be very valuable for a developer within a project team. Often, these hints are given by inline comments as part of the source code. As stated above, this approach is not optimal. The following enumeration lists anomalies resulting when documenting the collaboration aspects of an object oriented design using inline comments:

**Non-Cohesive documentation** Inline comments – as part of the source code – cannot cover interrelated program constructs properly. Because of their fixed attachment to one contiguous syntax fragment and the absent possibility to formally reference related source code [1], inline comments are inadequate to document a whole pattern consisting of more than one participants.

The scenario is depicted in figure 1. The comments *Cmt1*, *Cmt2* and *Cmt3* [2] may document the sole intents and responsibilities of the classes $A$, $B$ and $C$, but they can not refer to other collaboration participants (dashed line) properly to describe the whole concept of the pattern. Indeed, it is currently possible to provide literal references to other collaboration members inside the comment, but this leads to weak related documentation fragments that are unhandy to maintain (such as after renaming classes or an adding new member to the collaboration).

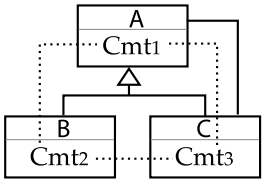


Figure 1: Non-cohesive comments: the references to collaboration participants used in the comments *Cmt1* - *Cmt3* are only informal – the whole collaboration is not adequately covered

**Tangled documentation** When participating in different collaborations, a program construct typically plays different roles according to different relations to other participants. Usually, each distinct role of an element demands a different explanation. This can be seen as an *N-to-1* relationship between documentation (multiplicity $N$) and a source code element (multiplicity *1*). In a primitive approach, one might merge all different comments to one huge agglomerated comment. This is yet also unhandy to maintain, since one comment captures different concerns. Using the vocabulary of aspect oriented programming, we now have a *tangled* documentation.

In the scenario depicted in figure 2 two collaborations (a *Composite* $C$ and a *Proxy* pattern $P$) overlap. The classes $B$ and $C$ are documented according to two different collaborations contexts. To actually realize this scenario, one might merge the comments $CmtC_2$ and $CmtP_1$ in

[1] references can be provide literally

[2] in the following figures, *Cmt* does not denote fields or methods in a class definition, but represent a comment for the given class definition

the class definition of $B$ to an agglomerated comment that contains both explanations.

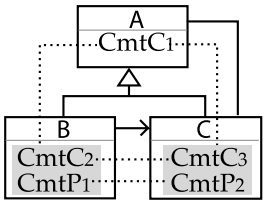


Figure 2: Tangled comments: Members $B$ and $C$ are documented according to two different collaborations

**Scattered documentation** As a counterpart to the observation above, different program constructs can play the same role within one single collaboration. This phenomenon obviously leads to a *scattered* documentation. Here, equal documentation is spread over a set of similar program constructs. A *1-to-N* relationship between documentation (multiplicity *1*) and program constructs (multiplicity $N$) is observable.

In figure 3 the elements $B$ and $D$ are similar in the collaboration since they both describe *leaf-objects* of the defined recursive structure. To explain their role in the pattern, the comment *Cmt2* is spread over different class definitions and thereby breaks the *single source principle*. Because there is no single (editable) source available, such documentation scenarios are unhandy to maintain. In addition, for a developer who wants to document his work, spreading similar comments over different class definitions is a demotivating, inefficient and boring task.

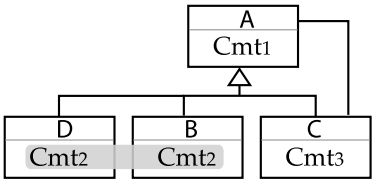


Figure 3: Scattered comments: Members $B$ and $D$ are similar and demand the same comment

As shown in the enumeration above, inline comments lack in documenting the collaboration aspect of object oriented code. This may result in out-dated — or even worst — in absent documentation. By addressing these limitations, this thesis will present a novel approach of documenting source code that is able to handle these three deficiencies.

## 1.2 Hypotheses

By unlocking the fixed relation between code and inline comments one can document a set of related participants (source code fragments) representing a particular problem solution. An additional intermediate layer between code and human readable comments allows a clean separation of both concepts. Such a layer consists of a formalization of the observed collaboration describing all participants and their inner relations in an unambiguous way. Now, certain source code elements qualify themselves for explaining comments when matching the collaboration description. This results in an *M-to-N* relationship between documentation and source code which enables a clean and easy to maintain documentation.

This solves the anomalies of tangled as well as scattered comments that were previously presented in section 1.1. Hopefully, preventing scattered comments might motivate developers to document their work, since they are now able to cover wide areas of similar code efficiently with just one single documentation entry that can be centrally managed.

Additionally, the source code will be in a position to document itself. Since the relationship between both concepts rely on a formal collaboration description, new or modified source code may qualify itself for certain comments if it matches the description. This solves the problem of outdated documentation that was reported in [LSF03].

## 1.3 Structure of this thesis

The main contribution of this work is a concept of an internal software documentation language that allows to document source code elements depending on their different collaboration contexts. To follow this goal, the thesis is structured as follows:

The subsequent chapter 2 presents and discusses current work. The concern of this consideration is twofold. First, we observe and structure different approaches in the area of software documentation. The following subchapter addresses the aspects of describing and recovering a documented collaboration within the project workspace. This analysis examines different mechanisms that allow documentation entries to address source code elements from the outside to realize the mentioned Code–Comment relation.

Thereon, chapter 3 elaborates a concept for a rule based documentation language, which leads to a corresponding tool implemented further regarded in chapter 4. Finally, chapter 5 proves the taken assumptions by evaluating the implemented tool using some different sized software projects as case study subjects. As a rounding for this thesis, the last chapter 6, embraces the generated knowledge and will give us an outlook for possible future work.

**Note:** In this thesis, the term collaboration is quite open defined as "a set of objects that relate to each other by object relationships." [Rie00]. Sometimes we use the terms 'pattern' and 'collaboration' synonymously.

## 1.4 Demarcation

The goal of this thesis is not to replace existing documentation approaches such as inline comments. Inline comments suit well when it is demanded to explain the interface of a sole program construct such as a class, method or a field. Their assignment to covering source code is yet really simple and exact, and no look-up mechanisms are required to obtain explanations for a currently opened document. Besides, there is an approved tool support for extracting these information to an (offline) hypertext based API documentation.

Concerning this thesis, the proposed approach covers aspects of an object oriented design that were previously not adequately documentable in source code using inline comments. Therefore, the final tool can be rather seen as an addition to the means of current documentation approaches. Having both, an API - as well as a collaboration documentation, the software under study become more accessible for fellow developers in the project team.

# 2 Related Work

The notion of this chapter is twofold. First, a taxonomy basing on the relationship between source code and comments will be presented. This classification is somewhat geared to the one presented in [Ves99]. By the means of that discussion, the approach proposed in this work will be arranged into that taxonomy.

A second section 2.2 is concerned about how to technically realize the chosen Code–Comment relationship. The approaches we are going to examine in this section may serve as a foundation of the proposed documentation tool.

## 2.1 Classification of current documentation approaches

As a brief introduction to the domain of software documentation, this subchapter presents a taxonomy based on the type of the Code–Comment relation. The result of this classification is depicted in figure 4 and is later used to arrange our approach into the area of discussion.

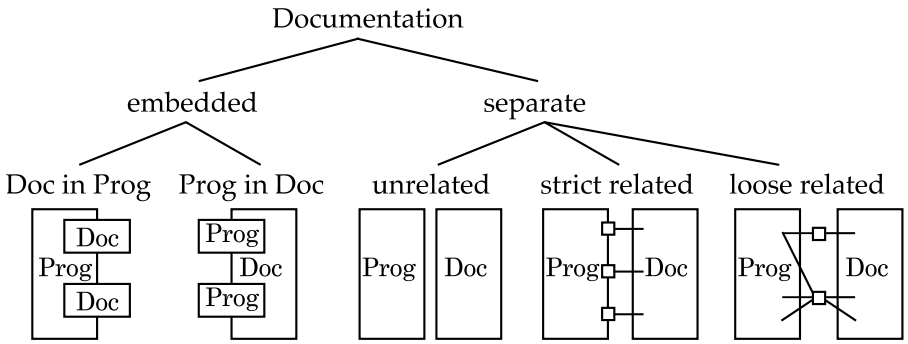


Figure 4: A taxonomy of different documentation approaches

In short, the relationship can be either strong when both concepts are **embedded** into the same document or rather weak (loose coupled) when they are divided into **separate** documents. Each of the forthcoming subsections will describe one of these approaches in more detail and discusses their advantages and disadvantages. Final to each examination, a representative of that approach is particularly covered.

### 2.1.1 Documentation embedded in program

In this model, the documentation is embedded as inline comments into the source code, hence it becomes a fixed part of the program. The primary entity in this model is obviously the programming code. Most developers favor this concept, because when writing code, one is usually focussed with the current problem to solve, and therefore can provide suitable comments.

Another advantage of this fixed relation between code and comments is its convenience, since there are no additional tools or look-up mechanisms required.

However, when applying a more complex architecture with a lot of interdependence relations between source code elements, this model lacks in providing an overall view. Section 1.1 showed that this embedded approach cannot capture the collaboration aspects of a software system properly.

Another disadvantage derives mostly from the primary status of the source code entity. Because of focusing on programing code, the person who writes the explanations has no literate freedom to structure his documentation properly. The structure of the documentation is defined by the program structure and is therefore sometimes hardly traceable. Hence, this approach is rather well suited for an *Interface Documentation* (or *API Documentation*), where the documentation structure is strongly based on the program structure.

**Representatives:** Available tools, following this approach, typically offer mechanisms to extract these inline comments to external – mostly hypertext based – documents. Predefined keywords (attributes) can be used to annotate meta information allowing to structure the later generated document in an adequate way. These tools are called *Doclets* and the *JavaDoc* tool is a well-known representative for the Java language.

### 2.1.2 Program embedded in documentation

As a counterpart to the previous model, this attempt addresses the problems of the absent possibility to structure the documentation properly. Here program code is aligned to the documentation structure and 'should be written similar to literature'. The code develops along with the documentation. Both concepts are embedded into one single document and can be later separated by the means of special purpose tools. This style of documenting software is called *Literate Programming* as coined by Donald Knuth in [Knu84].

This alternative also brings some drawbacks that were discussed in [Nør00]. When mainly focussing on the documentation structure, the program flow is sometimes harder to comprehend for a developer. Programers also do not want that documentation — as an incidental process — affects their source code in such an influential way. In addition, "Literate Programming systems [...] are not suited for modern object-oriented development" [Ves99].

**Representatives:** The first environment supporting this approach was the *WEB system* [Knu84] developed by Knuth in 1984. The system implements the ideas of *Literate programming* by providing tools for extracting compilable source code (called *TANGLE*) and generating formatted, printable documentation (called *WEAVE*). Nowadays, these tools usually play a minor role.

### 2.1.3 Unrelated documentation and source code documents

Here, documentation and programming code are divided into separate documents, which allows one to structure both independently of each other. Thereby, the documentor has all the literate freedom to structure the documentation at his own will, without affecting the source code as in *Literate Programming*.

The relationship between both concepts is defined in an informal mostly literary manner, so that there is no physical binding connecting them strictly together. Due to that unrelated model, consistency problems may occur and need to be supervised. In addition, these documents are not well suited to provide internal software documentation, since the documentation cannot offer access to the source code.

**Representatives:** Examples for this approach might be *Wiki* portal systems. Wikis are online databases, which can be directly edited by everyone (authorized) using a web fronted. Developers can provide general hints and advices to their implemented code. Source code can be pasted in but is not addressed in a formal way. Everyone interested is able browse or search by keyword for relevant comments and advices.

### 2.1.4 Directly related documentation and source code documents

To overcome the interference problems prevailing inside the previously introduced embedded model but also allowing to address concrete elements in source code, we need an external documentation using referencing mechanisms to refer to source code elements unambiguously.

For that purpose, there might be different alternative approaches that serve for such a reference mechanism.

- In a first attempt, the source code stays untouched and is addressed using a Uniform Resource Identifier (in short: URI). Such a URI identifies a resource with a compact string of characters. Examples following that schema are *Qualified Names* or *Path Expressions* in general that describe a source code construct by its hierarchical path.

- Otherwise, it is imaginable to markup source code adequately using special language extensions that do not affect the building and execution of the program. Later, these markups can be addressed from outside, hence gluing both documents together. This approach provides a finer granularity as qualified names, since it allows to exactly address every single line in source code. However, a foregoing proper annotation of the source code is required.

Nevertheless, both mechanisms have in common, that they only allow a fixed relationship that disregards the context of the addressable targets. Since the different relationships between targets are not reflected, the approach is not well suited to serve as a basis technology for documenting collaborations in source code. A set of interacting members can only be addressed by aggregating the result of different references to program constructs (i.e., to the concrete collaboration members). But is the whole (the collaboration) the sum of its constituent parts? Obviously not, because this only covers a particular occurrence of the pattern but not the pattern in general.

**Representatives:** *Elucidative Programming* [NAC+00] is an approach following this schema. The metaphor of the two separated documents is fully adopted visually. The *Elucidator*, developed by Kurt Nørmark, uses a two-frame fronted to present documentation and program code simultaneously. The result is intended to view on a web browser, somewhat similar to JavaDoc. The defined relationships between documentation entries and the referred source code elements are transformed to hyperlinks by the Elucidator tool. The Elucidator supports references from documentation to source (slink), source to documentation (dlink) and to any external document (xlink). To manage slinks, the elucidator employs both mechanisms presented above — URIs and Source Markers.

### 2.1.5 Indirectly related documentation and source code documents

So far, when using a direct relationship, as presented above, one can only document one particular pattern instance which is dissatisfying since other occurrences remain uncovered. A pattern as a whole cannot be covered, since there is no explicit knowledge about the collaboration structure given.

This leads to a last sort of approaches, where program units are dynamically selected for a comment by matching a certain pattern. Such a pattern formally describes the collaboration participants and their relations to each other and can be used as a query parameter. In contrast to the direct related approach, elements are not fetched by their exact position, but indirectly by their relationships to other participants.

**Representatives:** To the best of our knowledge, no work exists trying to fully adopt the concepts described. However, the InsensiVE Toolsuit [1] [MK06] for the Smalltalk language allows one to define so called "intentional views" on the source code level by using the patterns formalization. Further, it is also possible to attach a comment covering the whole view. But unfortunately, the approach is not fine grained enough to capture each participant and further describing its responsibilities and intention within the collaboration.

### 2.1.6 Conclusion

This chapter provides an overview over the different types of internal software documentation. Using the relationship between covered source code and documentation as its main criterion, the observation results in two main classes.

When **embedding** source code and comments into one document, they either affect each other in an undesirable way or disallow the proper documentation of a collaboration consisting of a set of participants. This approach does well fit if it is required to document the API of a software system, but actually encounters its limitations on adequately documenting an object collaboration.

The **separation** of both concepts might solve these problems, but requires some further work for assigning documentation to source code elements.

During the analysis, we have already denoted that a decoupled relationship might be the best solution when documenting the collaboration aspect of a software system. Due to the fact that collaborations are independent from concrete class level constructs, a fixed relationship to these elements is undesirable and therefore does not gain our attention. In addition, a fixed relation between code and comments only covers one concrete occurrence of a collaboration and is inflexible regarding code changes (relations need to be maintained). Therefore, this work is arranged within the class of indirect related external documents, as presented in section 2.1.5. To archive this goal, we require a formalization of the collaboration as a mean to glue documentation and source code together. For that purpose, the following section examines approaches that may serve as a base mechanism managing the relations between both concepts: Source Code and Documentation.

[1] available at: `http://www.intensional.be/`

## 2.2 Describing and Recognizing Patterns

To realize the desired relationship between source code and its corresponding documentation, an additional intermediate layer, managing all references between both entities, is required. The functionality of this layer is twofold. First, it must provide a language to formally describe the collaboration; and second an inference engine that is able to query for instances of the given formalization in source code. This work does only focus on the static structure aspect of a collaboration. Additionally discussing the runtime behavior of the participants is rather complicated and demands some more elaborated mechanism. Integrating such a feature might be a topic for future work but is currently omitted. Current work that combines structural and behavioral aspects to retrieve patterns in source code can be found in [TL03].

In this thesis, a comment is not a first class construct, thus only exists as part of a parent documentation entry that covers the whole collaboration. This parent entry is designed to make use of the mechanism that we will examine within this section to define the set of class level constructs participating in the collaboration. In this context, a comment only covers the exposed parts (the members) of a pattern query result. It is not desired that each comment holds a (own) query that tells him which items it actually covers.

This concept is not adequately realizable using the 'direct related' approaches presented in chapter 2.1.4, since these attempts disregard the relationships between participating collaboration members. Therefore, mechanism such as *Path Expression* or *Source Markers* are omitted in this section. Instead, we are looking for means to describe the relations of a collaboration formally.

Usually, the approaches presented in this section base on a higher level representation above the one of source code to allow a machine accessible description of the pattern. As an inversion to the direct addressing approach – where participants are exactly addressed – elements now qualify themselves if they match a defined pattern.

When defining such a pattern, we want to make an assertion that the actor that formally describes the collaboration structure of a documentation entry is the same person that has also implemented (or even chosen) the design artifacts for the observed software project. Therefore, we assume that he has some design knowledge to accurately describe the structure of the collaboration to document.

Each of the following considerations presents one approach in general and finally shows how (or if) this work could possible make use of this concept. Terminatory to this whole section, the applicability of each approach will be discussed on the basis of the following requirements.

**concise:** We demand a concise accurate notation resulting from the idea of merging documentation and pattern description into one physical document. Obviously, the documentation takes priority over the collaborations formalization. Therefore, the description language should be concise and in a human accessible textual notation so that it is possible to embed the formalization in each documentation entry.

**simple:** The effort to build up a reasoning environment should be as minimal as possible. Writing documentation is usually not something a developer is thrilled about. Describing a pattern to document should be done out of the box. In particular, it should be avoided to prior model the collaboration using third party tools. Also it is undesirable if the source code needs to be annotated with special formated marks, since this results in additional efforts demotivating the developer.

**flexible:** The person who writes the documentation should be able to adjust the granularity of the pattern as needed. We demand full freedom, ranging from addressing a general pattern with a lot of occurrences or even to only cover one concrete exact pattern instance in code.

**available:** Finally, we have not planned to implement our own collaboration mining algorithm, since this is beyond the scope of this work. So we rely on available implementations. The last requirement therefore demands an approved and maintained tool implementation with a strong community behind it.

Final to this whole chapter, we will use these considerations as a foundation for a brief evaluation that supports the choice for one of these following mechanisms.

This section only covers a preselection of different approaches that seems to be most suitable for this work. However, the area of software pattern formalization is wide, and there is a lot of effort done during the past years. A comprehensive review of current work can be found in [Bar03].

### 2.2.1 By Pattern Role Annotation

When applying a pattern, one has to assign the exposed roles to certain class level constructs that together fulfill a certain collaboration task. But, at the end of this process these roles are not directly identifiable in the source code. Therefore, Hedin in [Hed97] proposed to annotate the source code elements with pattern roles by the means of attribute grammars.

This alone is not powerful enough to distinguish between certain pattern instances in source code, since it will result in a set of role buckets, each consisting of a set of program constructs with similar responsibilities that are annotated by the same role name. So, Hedin further introduced simple

*collaboration rules* that define relations and constraints between particular annotated roles.

A drawback of this attempt is, that it is required to prior mark-up the source code properly to allow a later reasoning. Also – as far as we know – there is no tool implementation available. So, for the aims of this thesis, this approach is somewhat less important.

### 2.2.2 By decomposition to elemental patterns

A (design) pattern is usually given in an informal way, letting some freedom to the developer how to concretely transform it to executable source code as a formal mathematical symbolic language. Because of this creative procedure, the pattern as a whole, cannot be easily identified at a later time.

As mentioned, a possible solution is to formalize the pattern itself, which in turn requires some basic idioms/tokens from that the formalized pattern language consists of. This process leads to a formalization-grade tradeoff. So Jason Smith and David Stotts in [Smi02] already cognized that "a full, rigid formalization of static object, methods, and fields would be another form of source code". This lowers the flexibility of patterns significantly and "defeats the purpose of patterns".

Therefore they investigate to find some basic, elemental idioms of that patterns consists of. They called them *Elemental Design Patterns* as introduced in [Smi02]. In their work, these elemental pattern are formalized by the means of a calculus [1]. More elaborated patterns can be defined by composing already defined respectively elemental patterns together.

A benefit of this approach is that these elemental patterns are easier identifiable in source code, and therefore allows a *"divide and conquer"* principle to find some more elaborated patterns. But to the best of our knowledge, no implementation exists trying to recover patterns in source code following a description that is given by such a theoretical calculus.

However, a similar schema also appeals in [MMvD07] that aims to formalize crosscutting concerns. In this paper, the concept of a '*sort*' is somewhat comparable to an '*elemental design pattern*' as discussed above. A '*sort*' is the elemental part of a concern when talking about aspects. In their work ([MMvD07]) Marin, Monnem, and van Deursen gather a catalogue of 12 most commonly encountered *sorts*.

The whole catalogue was formalized in [Mar01] with a query language that is somewhat similar to the AspectJ pointcut language. A composition of specific sort instances is called a *Concern Model* and represents a certain program aspect. This divide and conquer approach enables the identifica-

[1] the calculus is called $\rho$ calculus which is an extension to the $\varsigma$ calculus known by the work of Abadi and Cardelli [AC96]

tion of aspects within software projects, as demonstrated with the SoQueT [1] Eclipse plugin.

For a source code documentation, this approach might be somewhat to inflexible, due it restricts the documentor to use a granularity depending on the catalog of elemental parts.

### 2.2.3 Using pointcut languages

The separation of comments and programming code and the separation of cross-cutting concerns (when talking about Aspect-Oriented programming) are in a certain way similar to each other. One can think of an advice as a source code commentary. It is therefore imaginable that their "addressing"-language could also be applied in our scenario. By the term '*pointcut language*' we refer to the popular (or default) query language used in AspectJ [2], since there are currently efforts (such as [HOA+06, EMO04, DJ04]) to use different approaches as an underlying query mechanism which we will not regard in this section.

A center concept in the area of *Aspect Oriented Software Development* is the *pointcut*. To explain the term in more detail, we need to further introduce the concept of *joinpoints*. Conceptually, a joinpoint identifies any point within a program's execution flow. In practice, the granularity depends on a chosen aspect language, but mostly satisfying the purpose. Some common joinpoints are for example: Method call and execution, Read/write access to a field, Object and class initialization. Further, a pointcut is a collection of several joinpoints and locates the position(s) where the advice (the aspect code) should be woven in.

There is currently one approach that uses these pointcut expression in a more general way as a code query language. The (query) language is called *Panther* [PHW04] and is "a superset of the (static part) of the pointcut language used by AspectJ" [Fer04] and is based on the PUMA [3] query engine.

As mentioned, the idea of this thesis is to document source code elements in context to a collaboration. But are aspects and collaborations fully comparable?

In the glossary on `http://www.aosd.net` [4], one can read that: "Aspects are one kind of concern in software development." A pointcut query

[1] SOrts QUEry Tool: Available at `http://swerl.tudelft.nl/bin/view/AMR/SoQueT`

[2] `http://www.eclipse.org/aspectj`

[3] abbr. for: **P**attern **U**nderlying **MA**tcher; `http://www.research.ibm.com/cme/components/components.html#Frameworks`

[4] the homepage of the annual Aspect-Oriented Software Development conference (visited on 2007-12-08)

language can be used to locate points in source code that are similar with respect to the observed concern. But participants in a collaborationare usually not similar. They share a common collaboration task, but each member has different distinct responsibilities and behavior.

Pointcut languages are not intended to fetch points that together form a common collaboration task, so "there is no general-purpose mechanism in AspectJ to relate different join points" [EMO04] to specify a whole collaboration. Special purpose predicates such as `cflow` (respectively `cflowbelow`) are able "to go beyond a single join point" [EMO04] but this is not that powerful and comprehensible. Since we have not planed that each single comment holds his own source code query, the redundant means to describe the relation inside collaboration are disadvantageous.

But even if we write a query representing a certain collaboration, the result of the evaluation of such a query are not class level constructs but a set of joinpoints. This is adequate when specifying points to weave in aspect code (the advice), but transformed to documentation: what should be the subject of the external comment? Is it the following line in program flow (like when using inline comments)? This mainly a question about granularity. If comments are only virtually connected to source code elements (as aimed in this thesis) then it is more appropriate to have them attached to the entire collaboration member construct (such as entire classes, methods, fields and so on) instead of single points.

In conclusion, pointcut query languages do not really meet the requirements of this work, since they are not intended to reflect collaboration consisting of different members adequately.

### 2.2.4 Graph based approaches

A linear textual notation is usually hardly accessible to machines, because their logical build-of is not directly observable and depends on the chosen programming language.

Since source code documents are based on a strict structure (syntax), it is possible to analyze such documents and build up more abstract representations, which are called **A**bstract **S**yntax **G**raphs (in short ASG). ASGs normalize the source code and represent their elements and relation in a graph representation. The former hidden structure behind a software unit now becomes more visible/accessible to humans and machines too.

This enables a better reasoning, since finding a pattern in source code can be transfered to finding a pattern of nodes and edges in a graph representation.

Software refactoring tools, such as Fujaba [1] strongly base on this concept. To document a software architecture, the Fujaba tool-suite uses "addi-

[1] abbr. for **F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain; Available at `http://www.fujaba.de`

tional nodes to enrich the ASG with information gathered during analyses" [Wen03]. To enable the identification of certain patterns, an a-priori specification of the consisting sub-patterns to find is required. These specifications are given by graph transformation rules consisting of a left side (the premise) that is a graph representation of the pattern to find, and a right side (the conclusion) that is the same pattern enhanced with annotation nodes that will be added to the final ASG. These transformation rules can be applied on the software projects abstract syntax graph. So, matching subgraphs of the software project become annotated by additional notes identifying the members of a certain pattern.

The whole process is based on a common principle in the field of pattern matching, starting with a *bottom-up* strategy that significantly decreases the search space for a following *top-down* investigation. So phase 1 tries to iteratively find presumably occurrences of a pattern in the graph, which will be closer inspected in phase 2 by means of a decomposition identifying and annotating the participating parts. The whole analysis execution is depicted in figure 5.

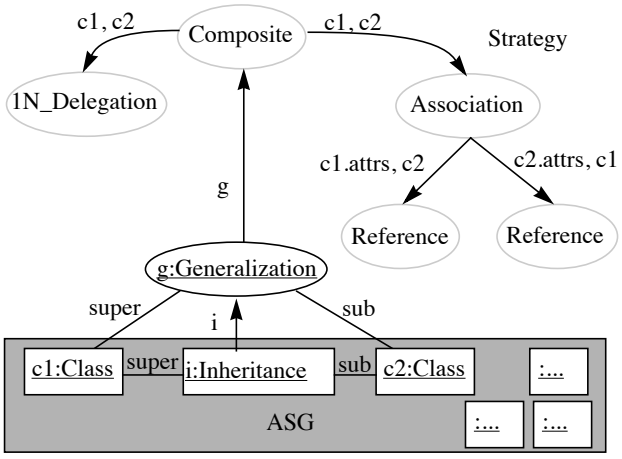


Figure 5: the bottom-up and top-down execution on a composite pattern (taken from [NSW+02])

It is possible – even desirable – to use a semi-automatic approach where FUJABA integrates the reverse engineer by allowing him to halt the execution and correct some immediate results by hand. This leads to better results.

Regarding the needs of our proposed tool, this approach does not fully met the requirements. There is a lot of effort required to initialize the reasoning procedure (for example: specifying graph transformation rules). In addition, there is no concise textual notation and third party tools are usually required to graphically model the patterns to document.

### 2.2.5 Ontology based approaches

The term *ontology* is originated from philosophy, where it is the study of being or existence. It is used in information systems to represent the knowledge of a domain (namely the *universe of discourse*) by describing their objects and relations in a declarative formalism [Gru93]. By giving an explicit specification of a phenomena using a specific vocabulary, the system under study becomes (more) interpretable by machines, thus allowing some reasoning about it.

Describing the concepts of ontologies in detail is beyond the scope of this thesis. A comprehensive introduction into the area ontologies can be found in [Usc96]. But so far, it is sufficient to know that an ontology consists of a hierarchy of *classes* representing real world concepts, *properties* that are attached to the concepts to model relationships between them, and *individuals* that are concrete instances of the concepts.

The idea of describing phenomena by the means of ontologies can be applied on the domain of software patterns, which are usually communicated informal (by graphical diagrams or even literally). Some fundamental work in this area was done by Jens Dietrich and Chris Elgar in [DE05]. They propose to formalize the structural aspect of design patterns by the means of an ontology language.

Class level artifacts (the M1 in OMGs Meta-layer Architecture) instantiates pattern participants. Hence, for conceptualizing a design pattern using an ontology language, Dietrich and Elgar proposed an extended meta level stack. They contribute a layer above M1 (application classes), called PDL (abbr. for **p**attern **d**escription **l**ayer) that formally describe pattern participants using classes (here concepts; not M1 classes) and their relation among each other using properties. By doing so, they employ concepts and restrictions defined in another meta level above PDL which they called object design ontology layer (in short: ODOL). ODOL describes the nature of a pattern thereby forms the language used in the PDL. In their work, [DE05] used the Web Ontology Language (in short: OWL) as description language, so pattern definitions may underly a good physical distribution. “Furthermore, patterns refining other patterns can also refer to them as network resources” [DE05].

Detecting a pattern is about finding a set of instances from the concepts defined in the pattern description layer (PDL). Towards a technical realization, Dietrich and Elgar proposed to map the defined ontology of software patterns to predicate logic formulas to allow reasoning about patterns by means of derivation rules. “Therefore, the task of detecting design patterns in software is reduced to finding a fact base for the above mentioned rule” (citation [DE05]).

Similar to the considerations taken inside the section about graph based

approaches, the effort to build up such a reasoning facility is not negligible. For instance, external tools such as graphical ontology editors are required, because the textual RDF notation is not that comprehensible. In addition, since RDF is a XML derivate, written documents are not concise and would bloat the documentation document.

### 2.2.6 By using a logic programming language

Source code reasoning using a logic programming language is an emerged field within the last years. A lot of work such as [HOA+06, Vol01, KHR07, MK06] was contributed during that period. Actually, it looks very promising to formal describe a pattern by defining relationships between the elements of the observed collaboration in a declarative way. Domain knowledge is given by a set of definite clauses – called Horn clauses – which can "be written equivalently in the form of an implication" (1). The right-hand side of the expression in (1) is called *premise* and the implication end (left-hand side) is called *conclusion*. Such expressions allow to deduce new knowledge from exisiting one. A clause without a premise is known as a fact and can be understand as assertion about a relevant piece of the world [CGT89].

$$p_0 \leftarrow p_1 \wedge p_2 \wedge ... \wedge p_n \tag{1}$$

Since we are not interested in boolean values but rather on relations between elements, we need to transfer the clause in (1) from propositional logic to predicate logic. So each literal $p_0$ to $p_n$ will be a relation (i.e. predicate) over the individuals of a domain. The schema of predicate logic can be transfered to code reasoning if we understand the individuals of the observed domain (the project source code) as a subset of the program constructs (such as concrete classes, interfaces, methods, fields and so on) and predicates as a mean to represent the structural relationships between these individuals. The described approach of merging a logic programming language with a standard object oriented base language is often known under the term *logic meta programming*.

To allow reasoning, a theorem prover requires a representation of the domain in form of a logical fact base that can be constructed by the means of structural analysis. Technically, the nodes of an abstract syntax tree representation get translated to corresponding predicates that together constitute the fact base.

The whole procedure of identifying a pattern in source code using a logic programming language is depicted in figure 6. The schema describes the architecture of the Pat System, developed by Krämer and Prechelt in [KP96]. In their work, they use *C++* as a base language and `Prolog` to describe and query for patterns. The two programs *P2prolog* and *D2prolog* transfer entries from a pattern catalogue ($P$) respectively design artifacts from the given source code ($D$) to a `Prolog` representation.

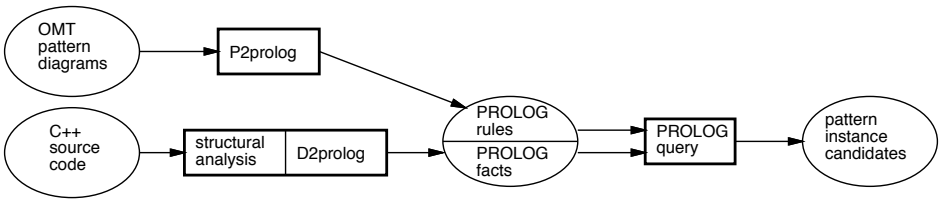


Figure 6: process of recognizing a pattern by means of a general purpose logic programming language (taken from [KP96])

A query is a set of predicates, defining criterions for participating individuals. Each predicate imposes a relationship on the individuals of a domain that occur in the result-set. Further, inter-relationships (i.e. relationships between predicates) can be defined using the concepts of variable binding. In this scenario, a query is a logical representation of the structure of a software pattern. Each valid variable binding therefor represents an occurrence of the pattern.

After this general introduction, the following paragraphs examine some prominent current work gathered by a brief literature research.

**LePUS** When talking about pattern formalization by the means of logic formulas, the work of Eden can not be disregard. His declarative language called LePUS [1] [AHE98], was first introduced in 1997, and is currently spread through many articles. Some of them ([Bar03]) even state that "it seems to be the best solution found". This Architecture Description Language can be represented in a visual or textual way and is defined in [AHE98]. Unfortunately, there is no tool implementation known.

However, there are some current projects following the concept of logic meta programming; amongst others these are *Pat* [KP96], *IntensiVE* [MK06], *CodeQuest* [HVdMdV05],*JTransformer* [SRK07] or *JQuery* [Vol01].

So far, this approach seems to well fit the requirements of this thesis. We have logic variables representing collaboration members, and predicates to express the relationship between these participants.

### 2.2.7 Conclusion

In this chapter we have studied different approaches that can be used to formally describe and query collaborating source code elements. This ex-

[1] abbr. for: **L**anguag**e** for **P**atterns **U**niform **S**pecification

amination focussed on the static structure aspect of a software pattern. To determine which approach best fits the demands of our ideas, we have elaborated requirements during the introduction to this chapter. Each subsection refers to these requirements inside a brief evaluation. Now, as we are finished with the discussion, we resume the facts gathered so far.

First, we considered code annotations using attribute grammars to markup the collaboration roles. This attempt is very exact, but due it requires to prior markup programming code adequately, it gains no further observance.

Thereafter, a rather theoretical approach was considered that describes elemental parts of a pattern by the means of a formal calculus. By using this mechanism, more elaborated patterns are described as a build-of of elemental ones. But, since these elemental building parts of a collaboration are given, the designed flexibility in choosing an appropriate granularity is not warranted. The person writing the documentation has not the full freedom to attach explanations to single class level construct as intended. Also, based on our efforts, we have not found any tool implementation following the ideas of the presented formal calculus.

Subsequent, we have noticed that describing a pattern using an ontology or graph representation is somewhat to laborious for the needs of this thesis. The efforts, that needs to be taken into account to build up a reasoning facility (describing an ontology of the pattern respectively formulating graph transformation rules) are also somewhat too exceeding. Also both do not provide a concise textual notation allowing to embed the description into the documentation document.

Finally, we examined the use of a logic programming language that allows to model the relationships of collaboration participants with a set of predicates. An advantage of this approach is that all pattern members are unambiguously captures as logical variables. This makes it easy to relate comments to constituent parts of a collaboration. Also this approach offers a concise and comprehensible notation that can be embedded into the documentation document.

So we decide to use *Logic Programming* as our source code query mechanism of choice. The forthcoming chapters conceive a concept around logic meta programming as its foundation and discusses some implementation issues occurring when transferring the idea into practice.

# 3 Conception

This work addresses the problems of tangled, scattered and non-cohesive comments in source code that usually accumulates when documenting a collaboration in source code. These problems were already analyzed in the problem specification in section 1.1. The fixed relation between inline comments and the documented element is not able to adequately capture these phenomena, because of its limited scope and the absent facility to formally reference related source code. As discussed during chapter 2.1, a clean separation of both concepts seem be a suitable solution to that problem. For that purpose, the previous subchapter examined different approaches that can serve as an intermediate layer between comments and source code, attending to manage the assignments of source code elements to documentation entries. Now, as we already defined our coarse aims, it is time to deep into the internal concepts of this work.

The approach slightly differs from current standards in the area of internal software documentation that primary aims to document an API or a set of consecutive statements usually without regarding any relations to concerned members. Since collaboration seem to be a crucial factor in current software development, we find that the current focus of documentation should be not only on sole segregated elements but also on the relationships between a set of collaborating elements. This may help to gain an overall view of the architecture of an existing software project.

Conceptually, this thesis combines the ideas behind *elucidative programming* [Knu84] and the *Pat System* [KP96]. Elucidative programing, as invented by Kurt Nørmark, already separates textual annotations from source code. The relationships between both concepts are defined in a separate document called *Edoc* that manages links from source to documentation (dlink) or from documentation to source (slink) [1]. In [NAC+00] one can find out that:

> "An href attribute of the slink tag makes use of a naming scheme that allows the author of the documentation to address Java program constructs in an unambiguous and context independent way."

The keyword in this citation is "context independent". The elucidator is not able to distinguish between different collaboration contexts in which the target may occur. Relations of the target to other members are not regarded. The mechanisms used inside the elucidator aim a very direct addressing mechanism allowing to exactly allocate a textual hint to a source

[1] beside there are external links too, which are actually not that interesting

code artifact without the risk of a false positive assignment. This fits well if it is desired to address sole elements required when writing tutorials or an API documentation. However, this directness and the disregarded context lacks in addressing a pattern of relating elements as desired in this thesis.

For this purpose, a logical description of the pattern to document seems to be more appropriate for fetching a set of related program constructs. Such a code exploring mechanism was already introduced in [KP96] 1996. The *Pat System* is a facility that uses a common logic programming language (prolog) to evaluate queries on an object oriented base language.

Within this thesis, a comment is not a first class entity, thus only exists as a part of a parent documentation entry that covers one collaboration task. Each such a documentation entry holds a formalization of the whole pattern that defines all pattern members and their relations among each other. A source code element qualifies for a comment if it matches a part of the formal pattern description.

To fade in comments, depending on a certain context (such as the current cursor position in a source code editor), we just need to scan all registered documentation entries and select those whose pattern instances intersect with the current context.

This chapter is outlined as follows. To define the needs or conditions to meet, we start with a brief requirements schedule, taking account of common use cases of the proposed tool. We will answer these requirements during this and the following chapter.

The main contribution of this conceptual examination will be provided inside subchapter 3.2 that explains the theoretical model underlying the approach. We will show some similarities to the role model approach that is eminent when talking about object collaborations.

Section 3.3 refines the consideration of the previous subchapter by gathering a domain model for RuBaDoc; our proposed tool. This conceptual model gives an insight of all involved entities and their relations to each other.

Derived from the developed domain model, a language that can be used to write down such documentation entries will be designed in chapter 3.4. These documents serve as an interchange format allowing to spread the written documentation over the project team.

Finally, we will regard some optimization issues in subchapter 3.5. The motivation result from the fact that usually only the first matching documentation entries are particularly observed. However, some entries might be more important or better fit for the actual context than others. Hence, it is desirable to sort matching documentations according to their relevance. For this purpose, section 3.5 elaborates heuristics to determine the relevance of a matching documentation entry. Based on that rating, resulting entries can be sorted according their pertinence.

Terminatory to the present chapter, we will give an aggregation of the ideas elaborated so far in a roundup conclusion chapter 3.6.

## 3.1 Requirements Analysis

This section lists the goals of the proposed approach that are addressed during the forthcoming chapters.

1. Obviously the most important requirement is the task of fetching all relevant documentation entries that are matching the current context. This should be done in a **direct** and **indirect** manner. If a documentation entry covers the current observed element, then we will call this a direct matching. On the other hand, if an entry addresses an element that is hierarchical on a higher level to the matching element, this is called an indirect matching. For instance, such an indirect matching occurs when retrieving a comment that covers the parent class or package above the current observed method or field.

2. A pattern usually consists of diverse participants playing certain defined roles. It would be a valuable feature to offer access to all participating elements after matching one of them. Additionally, for each participant, the tool should clarify, why it is actually selected for that pattern. Beside the manual typed documentation given by the developer this is a contribution to automatically document the source code.

3. As already stated, the approach relies on a *M-to-N* relationship between source code and relating comments. Hence, it is possible that there will be multiple documentation entries for a particular context. This occurs each time a set of collaborations overlap. For this scenario, some heuristics are required allowing to order matching documentation entries by their relevance.

4. The majority of current software projects are implemented by the craftsmanship of a whole project team. Usually there are no "*islands*" that only affect one person, but a lot of interdependencies between modules inside the software system. This aspect strongly requires an

interchange format that allows to share the written documentation inside the team. In turn, this requires a language to formally write down such documentation to a file that can be shared among the development team.

5. An important factor concerning the acceptance of the proposed tool is a proper integration into a common *IDE*. Besides presenting matching documentation entries, this integration comprises typical *CRUD* mechanisms. *CRUD* is a abbreviation for the four basic functions in area of data management and stands for **C**reate, **R**etrieve, **U**pdate and **D**elete.

## 3.2 A theoretical foundation

The goal of this work is to properly document source code in context to an object collaboration. For this purpose, we will need an underlying model as a foundation to illustrate the concepts of this novel documentation approach. Class models are a proven mean when describing the structural aspect of a software system. Because of the class/object duality they are close to the implementation and primary concern about the objects intrinsic properties. However:

> "One of the distinguishing features of object design is that no object is an island. All objects stand in relationship to others, on whom they rely for services and control." [BC89]

Because class models only provide one structural view, they are not able to clearly represent the spectrum of different behaviors of an object when participating in different collaborations. Therefore, we are looking for an alternative that is on a higher level of abstraction and intended to describe the collaboration aspect of objects.

### 3.2.1 The Role Modeling Approach

This section briefly discusses the role model approach that became popular in the late '90s. It shows that class models cannot decompose a complex system according to certain phenomena such as collaborations properly. So Reenskaug in [RWL96] proposes to "isolate an area of concern and to create a role model for it". The advantage of this separation of concern is that we get manageable models. "The role model is an abstraction on the object model where we recognize a pattern of objects and describe it as a corresponding pattern of roles" [RWL96]. A role is a dynamic view on a set of objects that has a similar behavior in context to a particular collaboration task. Furthermore, a *role type* is the abstraction from a set of roles and "defines the operations and the state model of the role" (Definition 3-13 in

[Rie00]). This artifact somewhat corresponds to the relationship between classes and objects, but occurs on a higher level of abstraction.

To specify such a role type, one can use a notation similar to the one of figure 7.

```
rolemodel FigureChain {
  // Predecessor in a chain of figure objects.
  // Provides operation to generically handle requests.
  roletype Predecessor {
     ref<Successor> successor;
     void setSuccessor(ref<Successor> successor);
     void forwardRequest(ref<Request> request);
  }

  // Accept, queue, and dispatch requests.
  roletype Successor {
     void handleRequest(ref<Request> request);
     void handleDeleteRequest(ref<DeleteRequest> request);
     void handleInvalidateRequest(ref<InvalidateRequest> request);
  }
```

Figure 7: Specification of the `Figure` and `Child` role in a `FigureChain` role model (taken from [Rie00])

Figure 8 presents a graphical notation of the role model specified in figure 7 above. Note that *Role constraints* are omitted in the specification since they are currently not that important. A set of all identified constraints between certain roles can be found in [Rie00] in section 3.3.6.

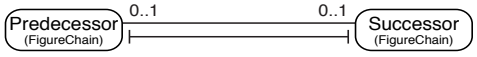


Figure 8: A visual representation of the role model specified in 7 (taken from [Rie00])

Describing a collaboration task corresponds to identifying all related role types and defining associations between them. Such a composition is called a *role model*. This model is a good foundation for our examinations. The following section describes similarities and thereby introduces the ideas of our work on top of these concepts.

### 3.2.2 The Documentation Model of RuBaDoc

After the brief introduction into the area of role modeling, this section discusses the relations of that concepts to our proposed approach.

The contribution of this thesis is a concept of an approach that allows to document program constructs in context to a collaboration. A documentation entry covers a set of similar object collaborations by commenting their

participants. As mentioned, such a set of similar object collaborations can be conceptually described with a *Role Model*. Hence, there is a correspondence between a documentation entry and a *Role Model*. In particular, a documentation entry covers a *Role Model*.

Each *Role Model* consists of a set of role types specifying the qualification of the participants. In particular, it specifies methods and properties of the roles that are involved in the collaboration. The idea of this thesis is to document the constituent parts of a role type specification. Each documentation entry consists of a set of comments covering elements of a role type specification (such as certain *role methods* and *role fields*) of a corresponding role model. It works on a finer granularity than sole roles by describing their constituent parts; but not the role itself.

**Consequences:** As discussed, this work conceptually bases on the concepts of the *Role Modeling* approach to document the collaboration aspect of a software system. Therefore, the following consequence can be derived.

**Comments are independent of class level contructs:** In [Rie00], one can read that "Role types are defined independently of classes, because their primary purpose is to show how an object behaves within a specific collaboration task." Since the targets of the comments are parts of a role type specification, this independence also applies in our scenario. A comment does not directly covers one particular program construct in source code, but parts of all objects that conform to (or play) this role. Thereby, program constructs become transitively documented when applying a Role Model on a Class Model (called Class-Role Model). This indirect relationship allows us to **avoid scattered comments** that were captured during the problem specification in section 1.1. Such a normalization applies a single source principle, since one single comment simultaneously captures all similar members of a collaboration task in source code.

**Comments may overlap each other:** On the other hand; "An objects behavior is defined by the composition of all role types of all roles it may play" [Rie00]. Hence, objects participate in a non-empty set of overlapping collaborations (Definition 3-16 in [Rie00]). This aspect is important for documentation too. Consider documenting a compound design pattern whose constituent parts overlap so that some class level constructs occur inside multiple patterns simultaneously (playing different roles). Obviously — if documented adequately — each such a class level construct requires a different explanation depending on the actual considered collaboration. Since comments are attached to role type parts of a role model, they might also overlap without affecting each other when applying role model synthesis. This documentation superposition **avoids tangled comments**, since explanations

in source code need not to be (physically) merged to a agglomerated comment covering different contexts.

**Comments relate each other:** As same as "Role models are the place where role types are defined" [Rie00] - a documentation entry envelops all comments of the collaboration members. So this parent entry has knowledge which targets are currently covered with documentation. Because of this, each comment is in a position to reference participating collaboration members, by inspecting his parent documentation entry.

Due to the fact that comments are related via the parent documentation entry, they document a collaboration in a cohesive manner.

These three considerations demonstrate the adequacy of the role model approach as a theoretical foundation for this thesis. It solves the three problems identified in the initial problem specification in section 1.1: *non-cohesive comments*, *scattered comments* and *tangled comments*. Figure 9 provides a schematical view over the concepts discussed so far.

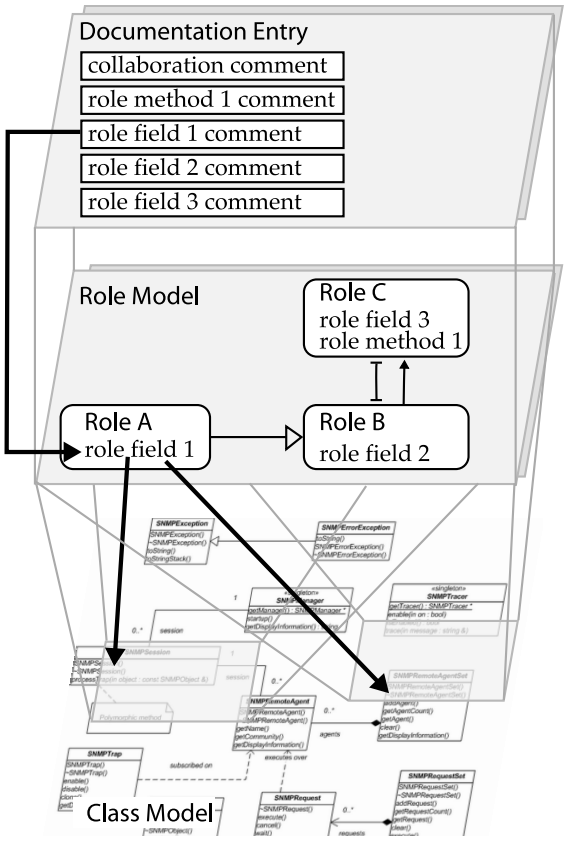


Figure 9: the RuBaDoc approach in concept

The thick black arrow on the left side exemplary shows how source code is transitively documented using a formal description of the collaboration as a glueing medium. In the example, there are two collaborations that conform to the specified *Role Model*. Hence, both are covered with the comments given in the *Documentation Entry* (exemplary "`role field 1 comment`" in figure 9).

After presenting the commonalities of both approaches, we are now concerned about the differences. Actually, someone might argue that our tool is intended to serve as an instrument for a role model documentation. This assumption does not entirely apply, since the core concept of the role modeling approach – namely the role – is missing in our documentation model. Still, we can select all parts of a class definition that are important to the actual collaboration context, but there is no further subsuming from these parts to a role. Hence, we are not able to attach comments to a role as a whole. However, since a role is an aspect of one class, it can be indirectly represented as the sum of its constituent parts, where each of them belonging to the same class definition. This idea could be visually supported by grouping collaboration participating elements (like certain methods,fields, parameter and so on) that belong to the same class.

**Employment:** For a better understanding, a practical example from [Rie00] was chosen to describe the concepts presented so far. The study describes a yet simple *FigureChain* role model that was previously depicted in figure 7 on page 27. The listing comprises two role types – namely `Predecessor` and `Successor` – that can be used to model a chain of figure objects. We now demonstrate how our documentation approach employs the introduced model in practice.

To document a collaboration that follows such a figure chain, a formalization describing the structural aspect of the collaboration is required to serve as a query parameter. Such a formalization unambiguously captures the members of the collaboration task using identifiers (here logical variables). For example, one of such an entity represents the role method `handleRequest()` in the `Successor` role type and another might represent the role field `successor` inside the `Predecessor` role type. The person writing the documentation now needs to connect these entities to each other using object relationships. For example, the previously mentioned *successor* field references a certain `Successor` type.

After having captured all participants, one is now in position to relate a comment to each of these related program constructs. For instance, one might want to attach a textual explanation to the *handleRequest()* role method to document the intent or the responsibilities of that constituent part of the collaboration. It is important to memorize that the subjects

of the comments are the parts that made up a role type specification not concrete parts in the source code. However, through applying a role model on a class model (called a *class-role model*), source code elements are documented transitively. This approach is more efficient compared to a direct Code–Comment relationship, since we are now able to cover a (possible huge) set of program constructs playing a similar role within one particular collaboration. Further, there might be multiple instances of that particular collaboration inside the projects source code, that all becomes automatically documented by the same documentation entry.

In a first conclusion, the role model approach is well suited to describe the basic ideas of that thesis, but was not fully adopted as an underlying model, due to the absent representation of the *Role* itself.

### 3.2.3 Formal describing a collaboration with DataLog

Role Models are important at the analysis phase of an information system development. They provide an adequate mean to specify collaborative behavior that domain experts can easily validate, before implementing them into source code. However, regarding the common phases of a software project life-cycle, our approach plays in a different (later) point of time. Usually, internal software documentation occurs parallel to the implementation phase or even after that. Therefore, the collaboration aspects are already implemented in source code and are not that clearly identifiable.

Now, the main task is to recover these collaboration tasks inside the source code, to retrieve the concrete participating elements covered by the explanations. For that purpose, we demand a mechanism that allows us to evaluate queries on the source code level. As already discussed, this in turn requires a formal description of the considered collaboration as a parameter for such a query. Concerning the discussions taken in the previous section, this formal description ideally correspond to a role model; hence consists of a set of role type specifications and their relationships to each other.

Some fundamental studies were accomplished in subchapter 2.2. At that time, *Logic Programming* seems to be best suited to formalize a collaboration adequately. Actually, we demand no further skills in the area of logic programming such as recursion or the concept of goal reductions. In particular, collaborations are not modeled using a set of rules but only by exactly one query. This implies that no collaboration definition builds on top of an other. Facts constituting the fact base for reasoning are generated automatically by the employed query facility (here JQuery).

Since in our scenario, a query is just a set of relations between collaboration participants, we restrict us to use DataLog to express collaboration tasks. DataLog originates from the database community and “is in many

respects a simplified version of general *Logic Programming*" and "has been developed for applications which use a large number of facts stored in a relational database" [CGT89]. In contrast to [CGT89] we do not demand that facts need to be stored in a relational database [1].

Compared with popular logic programming languages such as Prolog, DataLog "has a purely declarative semantic". Due to the procedural semantics of Prolog, the termination of a recursive program depends strongly on the order of the rules and literals (in the rules) [CGT89]. The following enumeration emphasizes the DataLog character of the mechanism employed in this work.

**Each fact base predicate is *ground*** The facility that structural analyzes the project source code and extracts design artifacts solely produces *ground* facts, i.e. facts which do not contain any variables. These entries form a *Extensional Database* which is partly stored on the file system. Each such an entry is exactly one predicate that represents a relation between individuals (program constructs) gathered during the structural analysis. When expressing a query, one composes a set of these fact base predicates typically by using variables (in order to be general). The appendix A lists all such core predicates.

**No use of compound/complex terms as arguments of predicates** When formulating a query, no complex predicates such as $p(f_1(1), 2)$ are allowed. In addition, we do not need constructs such as *Lists*.

**Queries guaranteed to terminate** A query to recover occurrences of a certain pattern can be defined by a set of core fact base predicates (i.e. predicates that were used to specify facts in the knowledge base). Since there is no recursion, the query might satisfy (if all variable substitutions are compatible) or not, but at least terminates after a finite amount of time. However, the employed reasoning facility contributes some "useful predicates that have been derived from the core fact base predicates using rules."[2] Considering the rule definitions of these derived predicates, their purpose follows convenience reasons, since they mainly subsume a set of predicates without using recursions. There are only 3 predicates namely *child+*, *subtype+* and *implements+* that using recursion to specify that an element is in a hierarchy of an other. Due to the nature of the base language, there are certain stratification restrictions preventing that these queries won't terminate.

**The order of predicates is irrelevant** When formulating a query one need not to care about the order of the predicates. Furthermore, the

[1] However, there are currently approaches such as [HOA+06] using a relation database for source code reasoning

[2] `http://jquery.cs.ubc.ca/documentation/appendix2.html`

employed reasoning facility reorders predicates in order to optimize the query evaluation time. Since predicates are automatically reordered, there are no control structures that were available in Prolog. This underlines the purely declarative character of DataLog.

Now, as we have briefly introduced a mechanism that is able to solve the task of mining collaboration participants, we want to discuss how DataLog can be used to reflect the concepts captured in chapter 3.2.1. The core concepts of such a query are *predicates* (relations), *individuals* (elements of the domain) and *variables* as a placeholder for the second. We have already discussed, that there is obviously a correspondence between variables and the parts a role type specification consists of. A variable represent a set of class level constructs that have a similar behavior in context to the described collaboration. Each valid variable binding therefor represents a particular occurrence of that role model inside the observed project workspace.

If an object needs (wants) to act as a certain role, it must fulfill the specified requirements. So concrete elements of the object model must correspond to the ones of the role type specification. Transfused to the concepts of a DataLog query, this conforms to the process of binding a variable to a concrete individual. Hence, individuals are represented by a subset of all elements appearing inside the application model (like concrete classes, methods, fields and so on) of the observed project.

Finally, we are using predicates to express a subset of the relations that make up an object collaboration task. These relations include common associations such as `extends` or `calls`. A detailed listing can be found in the appendix A.

### 3.2.4 Conclusion

This section introduced the base ideas of our proposed approach to adequately document the collaboration aspect of a software system. Conceptually, our work bases on the role modeling approach as an established mean to describe object collaborations. Further, we use DataLog as a mechanism to formalize and recover collaborations in source code.

Terminatory to this chapter, the properties of the relation between code and documentation will be explicitly highlighted.

**Direction** The Code–Comment relation is unidirectional and only relates a textual comment to source code. Otherwise, retrieving all comments for one particular source code fragment requires an iteration over all registered entries of the documentation repository to look for intersections.

**Targets** Conceptually, targets of the relation are a subset of the elements of a role type specification. Later, when merging classes and roles to a

*class-role* model, an architect attaches roles to classes. These classes must fulfill the requirements specified by the corresponding role type. Parts like methods or fields in a class definition must match those defined in a role type specification. Hence – considered transitively – targets are a subset of the elements of a class definition. The indirect relation between code and comments was previously depicted in figure 9.

From a more practical point of view, the set of possible targets strongly depends on the chosen query facility. This environment is responsible to offer access to the abstract syntax tree of the given software project. It does so by transforming nodes and edges of that syntax tree to logical fact base predicates. This procedure is crucial in determining the addressable targets of a workspace. Every item or relation that was not transfered to a fact base representation is obviously not addressable for documentation.

The set of different sorts that can be identified in source code corresponds somewhat to the set of the unary core predicates from the chosen logic (meta) programming language. Appendix A provides a full list of all known core predicates of the chosen TyRuBa/JQuery query language[1]. As observable, the list primary comprises of interface level constructs that are primary concerned about the program structure. Unfortunately, besides method calls there are no further statements identifiable, hence not comment-able.

In particular, as reported in [KHR07], it is not possible to adequately describe a pattern that relies on loop statements to interact with a set of associated objects. Also, our approach does not qualify to document algorithm implementations, since the statements that build up the single operations are not addressable. For that purpose, inline comments are obviously more suited.

**Multiplicity** Roles are independent of objects and can be 'played' by different objects that fulfill the specified requirement. Hence, a comment for a role part impacts all elements playing this role.

Otherwise, "A class defines a non-empty set of role types" (Def. 3-16 in [Rie00]), so classes may participate in different collaborations playing different roles. This is a result of composing collaborations. The bureaucracy pattern presented in[Rie98], for instance, merges several GoF design patterns [2] whose constituent patterns overlap. In particular, the *Manager* class plays three distinct roles (Mediator in Mediator pattern, Successor in a Chain of Responsibility pattern and Observer

[1] list taken from `http://jquery.cs.ubc.ca/documentation/appendix2.html`
[2] namely Observer, Mediator, Composite, Chain of Responsibility

in an Observer pattern), hence if documented adequately demands (at least) three different explanations.

Combining both considerations, we will get a *M-to-N* multiplicity between the elements from the source code and comments from the documentation repository. The following subchapter further presents all identified relations in more detail.

## 3.3 Domain Model

Footing on the documentation model presented in section 3.2, this section dives more into the internals of our conceived approach. To do so, a domain model identifying and relating all involved entities was elaborated.

Since we want to focus on the relationships between these concepts, an *Entity Relationship Diagram* representation following a *CHEN* notation was chosen. The final diagram is depicted in figure 10

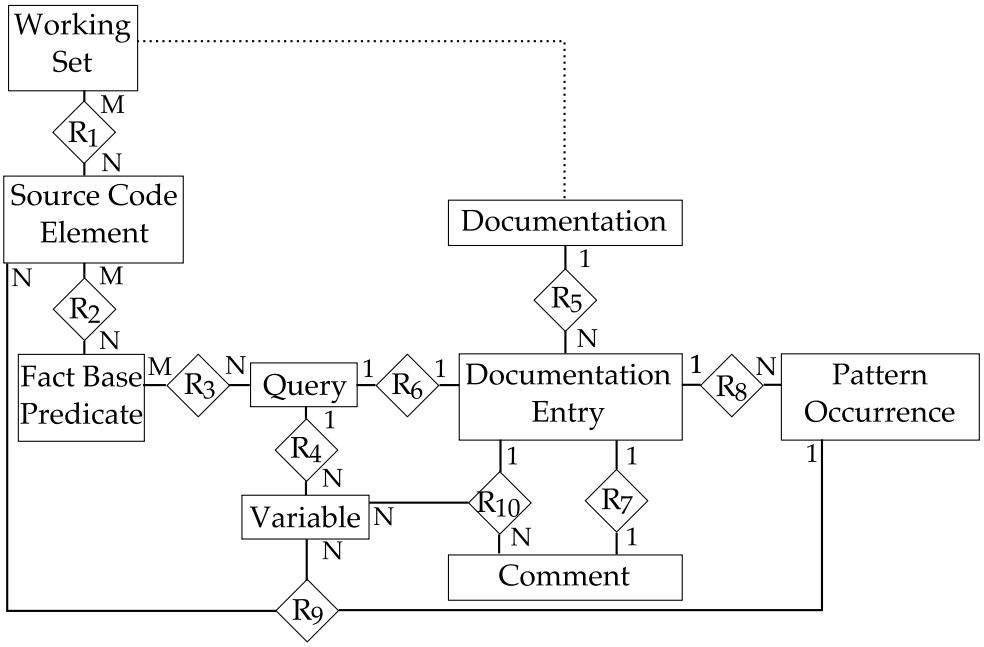


Figure 10: A Domain Model of RuBaDoc

The following explanations are structured according to the order of the relations $R_1$ to $R_{10}$ in figure 10. Each of the following considerations will cover one particular relationship by presenting involved entities and additionally discussing the relation-multiplicities.

$R_1$ To bound the search domain, we are applying the concept of so called *Working Sets* to represent the domain of the observations. The term is originated from the Eclipse IDE and can be understand as a container for source code documents [1] (multiplicity $N$). It is possible to manage many working sets (multiplicity $M$) and to choose one as a foundation for a documentation (dashed line).

In figure 10, the term 'Source Code Element' represents one concrete language concept instance that is a program construct inside the observed Working Set. This work only considers languages following the object oriented paradigm, since the different relationships between rogram constructs are used to define the collaboration to document. In other paradigms, relations between language concepts are usually not that powerful and important as in OO.

$R_2$ We have already mentioned that we apply DataLog as a reasoning mechanism. Therefor, the queries that select program constructs for comments are not evaluated directly on source code but on a logical representation — the fact base. The mapping from source code to fact base predicates is very important, since it determines which targets can be possible attached with comments. Technically, fact base predicates can be constructed by the means of a structural analysis. This will be done by traversing the *abstract syntax tree* (AST) representation of each source code document and transforming (a subset of) its nodes and edges to logical predicates. The *N-to-M* multiplicity results from the use of n-ary predicates such as *class(*X*)* (n=1) or *derivedFrom(*Sub,Sup*)* (n=2).

$R_3$ A *Query* (or a *Goal*) logically describes a certain collaboration structure by defining relationships between collaboration involved elements using a set of predicates (multiplicity $M$).

$R_4$ In order to be general, a *Query* employs *Variables* that may become substituted by an theorem prover when evaluating the query. If there exists a compatible variable substitution, the collaboration occurs in the *Working Set* and each variable assignment represents a collaboration member. As mentioned in section 3.2.2, each collaboration member is a program construct that corresponds to a constituent part of an object role.

$R_5$ After describing the targets of the comments and the inference mechanism, we now want to introduce the structure of a documentation document. A *Documentation* is a repository for *Documentation Entries* (*1-to-N* multiplicity) and covers a certain "Working Set" (dashed

[1] Since whole documents are not important, the relation between documents and containing source code elements is subsumed into $R_1$

line). This entity is unique, and at any time, there is only one such a repository instance existent.

$R_6$ A *Documentation Entry* covers an object collaboration task formally described by (exactly) one logical *Query*. This entity is the central concept of this work that gets visualized each time the editor context (i.e. the current cursor position) intersects with the (compatible) variable substitutions of the related *Query*.

$R_7$ A *Documentation Entry* can be documented as a whole using one single *Comment*. This explanation is intended to describe the intent of the collaboration without regarding the participants of the pattern.

$R_8$ The virtual *Pattern Occurrence* entity represent a compatible variable substitution of the *Query* that describes the structure of the collaboration. If there are multiple variable assignments possible, then we might have detected different pattern instances **or** a role part of a particular collaboration can be played by multiple program constructs (different pattern occurrence). In practice, we do not distinguish between both cases, since this requires further work on subsuming results. Also it is hard to distinguish if a detected occurrence is also a different pattern instance. Consider, for example, the well known Observer pattern that defines a loose coupled relation between a *Subject* and an *Observer* role. Each distinct matching Observer defines a different pattern occurrence (since different class are intended to play the Observer role) whereas a different matching Subject additionally defines a further pattern instance. Since we do not care about association multiplicities when formalizing a collaboration, it is somewhat hard to distinguish between both cases. However, in practice, this is not dramatic since comments are attached to parts of a role type specification and are therefor independent of concrete pattern instances or occurrences.

$R_9$ This relationship describes a particular variable binding. The $N$ variables employed by the predicates of one logical *Query* are related to concrete source code elements in a way that satisfies the *Query*. Each such a variable binding relation corresponds to exactly one *Pattern Occurrence*.

Technically, for a *Documentation Entry* a *Pattern Occurrence* is the only medium that allows access to the source code elements it currently covers.

$R_{10}$ Relation $R_{10}$ is responsible for relating comments to source code artifacts. On a closer look, a comment for a participant is invariant between different collaboration occurrences, since the intent of this element stays the same. Therefore, the assignment of comments ($R_{10}$

in figure 10) was separated from the assignment of the actual source code positions ($R_9$) of a collaboration participant. This results in two different mappings that share an identical *key set*. The key set consists of the names of the used logical variables inside the collaboration query. This form of a *normalization* allows us to vary both artifacts independently and employs a single source principle, since one single textual comment covers all source code elements playing the specified role part.

## 3.4 Documentation Language

Within the previous Domain Model we introduced and discussed all involved entities and their relation in more detail. Footing on that considerations, we now want to conceive a language to write down documentation according to that model. These documents can be consulted by the proposed tool. Therewith, we met the 4th requirement formulated in section 3.1 by providing a specification for an interchange format. This allows developers to spread their written documentation in the project team.

A good starting point for that goal is to look back on the domain model on figure 10 and identify the entities that need to be specified by the documentor and therefore need to flow into our documentation language. For that purpose, this discussion is aligned to the structure of 3.3.

- Obviously a *Working Set* must be specified since it defines the universe of discourse. But the documentation need not to be dependent of a certain *Working Set*. Furthermore, there might be documentation entries covering general collaborations. So it is sufficient to leave this determination open until tool startup. At tool runtime, the developer may decide which *Working Set* he actually wants to have covered with documentation. Therefor, the *Working Set* is not a part of the proposed documentation language.

- *Source Code Element* and *Fact Base Predicate* are already specified by the chosen programming language respectively the mechanism that transforms them into logical facts. There is no need for a documentor to adjust these entities, so both are omitted.

- Obviously, the pattern defining *Query* must be provided be documentor, because it (indirectly) specifies the targets of the given comments.

- The *Documentation* entity only serves as a container for all entries and is already represented as the whole documentation document that is written with the conceived language.

- *Documentation Entry* and *Comment* obviously need to be specified by the author by the means of the proposed language.
- *Pattern Occurrences* are derived from a *Query* at tool runtime. As a matter of course, they are not present in the conceived documentation language.

Now, as all involved entities are identified, we can further work on the design of our documentation languages. In theory, every formal language $L$ follows a specific grammar $G$ – or in other words, $G$ as a *metalanguage* of $L$ and describes $L$. EBNF [1] is another language that can be used to define $G$. The contribution of this chapter is the definition of $G$ — a (*context free*) grammar of the proposed documentation language.

To define how a valid documentation document will look like, we will use a graphical EBNF representation describing the grammar of our language. The graphical representation is somewhat better to read as a (more) cryptical textual notation. Figure 11 is a straight forward conversion of the entity relationship diagram previously seen on figure 10. We just transfered the *Documentation*, *Documentation Entry*, *Query* and *Comment* entities to corresponding EBNF non-terminals preserving the relation multiplicities identified in chapter 3.3.

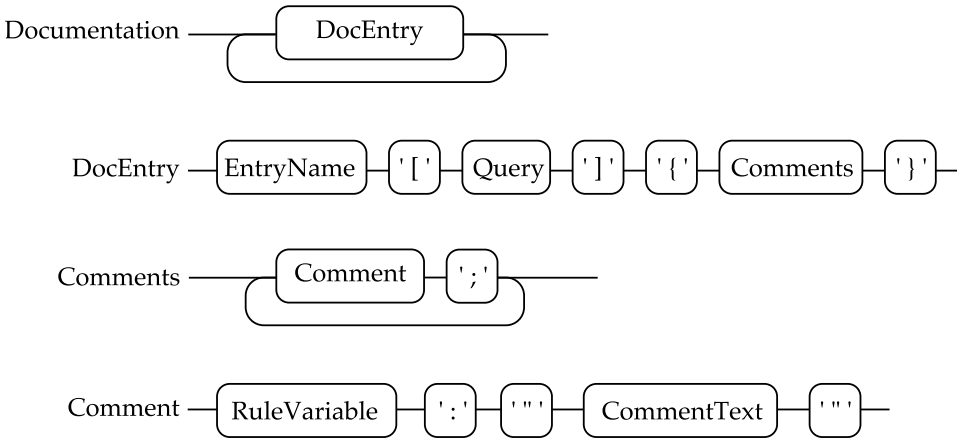


Figure 11: Documentation language in a graphical EBNF notation

Let us now further consider the *Query* nonterminal occurring in the second lane. There is no EBNF expression given, which further defines how valid queries are specified. The reason for this fact is, that we prefer to embed a rule definition that is given by a chosen logic programming language. So the *Query* nonterminal can be unsterstood as a placeholder

[1] abbr. for: **E**xtended **B**ackus **N**aur **F**orm

for a particular query language definition. However, an EBNF definition for the Rule language employed in this work can be found in [DV98].

Due to the fact that EBNF is a language to define *context free* grammars, it is not possible to express the important relationship between a variable inside a rule [1] and its relating comment inside the "Comments" nonterminal. Therefor, the following paragraph explains the semantics by instantiating the grammar to a simple example. Consider, for instance, a simple *Proxy* design pattern given by a structure according to figure 12.

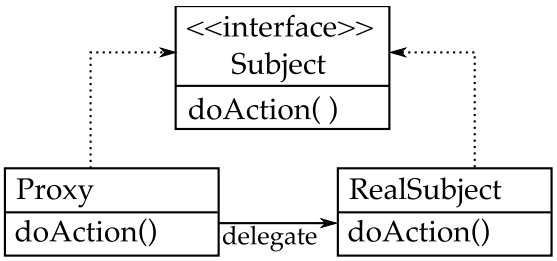


Figure 12: A Proxy pattern in a UML class diagram notation

To properly document all occurrences of such a structure in the source code, one need to identify all pattern participating parts and their corresponding relations between each other. Listing 1 gives a first example for the usage of our proposed language.

```
1  Proxy [ interface(?Subject),method(?Subject, ?AbstM),
2          implements(?Proxy, ?Subject),
3          implements(?RealSubject, ?Subject),
4          method(?Proxy, ?PM), method(?RealSubject, ?RSM),
5          signature(?PM, ?AbstM), signature(?SM, ?AbstM),
6          field(?Proxy,?Delegate),type(?Delegate,?Subject),
7          calls(?PM, ?SM, ?) ]
8  {
9    * :       "Provide a surrogate or placeholder for
10             another object to control access to it";
11   Proxy :   "maintains a reference that lets the proxy
12             access the real subject %RealSubject%";
13   Delegate:"A reference to the real subject
14             %RealSubject% in a proxy pattern";
15   RealSubject:"defines the real object that the proxy
16             represents";
17   Subject :"defines the common interface for
18             RealSubject and Proxy";
19   PM :      "This method inside %Proxy% makes a proxy call
20             to %RealSubject%";
```

[1] even if we former assigned an expression to Rule nonterminal

```
21   RSM :    "This method gets proxy calls from %Proxy%";
22 }
```

Listing 1: An example documentation entry - written according to grammar presented in figure 11

Regarding listing 1, two distinct parts are cognizable. Starting with the "Comments" area surrounded by the curly braces (lines 8-22); it is well cognizable how one can easily attach textual explanations to the different collaboration participants. The relationships between these participants are preliminarily specified inside the "Query" section between the squared brackets on lines 1 - 7. In this work, TyRuBa is used as the logic meta programming language of choice. A list of TyRuBa specific predicates and their semantics can be found in the appendix A. By a convention, variables need to be marked with a question-mark prefix. It is not necessary that every employed variable in the "Rules" section must be documented in the following "Comments" area (for instance, we have not covered the method inside the parent interface). The *Kleene Star* " * " , occurring in line 9, is a special construct to provide a general comment that will cover all collaboration participants in source code and can be used to provide some general informations about the collaboration independently from its current structure. This corresponds to the relation $R_7$ identified in section 3.3 that presented a domain model of the proposed approach.

Another interesting aspect is the possibility to refer to another participating element inside a comment by just using the role part name wrapped between two percentage signs. At tool runtime, these placeholders become automatically substituted with the current bounded source code element with respect to each particular pattern occurrence.

In the example, the given pattern structure definition is rather general, since it describes the pattern structure solely by its relations. However, the documentor can easily confine the result set of pattern occurrences by using some more explicit criterions. For example, by appending the predicate *name(*`?RealSubject, Display`*)*, all given comments are only assigned to collaboration participants, whose proxy delegates to a class named `Display`. Even further, it is also possible to use a regular expression as a name pattern. So by using *re_name(*`?RealSubject, /Display$/`*)* as an additional predicate, the query engine fetches all patterns having a concrete subject ending with "Display" (e.g. MobileDisplay). Since, TyRuBa internally uses the *Jakarta Regexp* [1] library, the *re_name(*`?Elem, /RE/ `*)* is a really powerful instrument to adjust the generality of a documentation entry.

[1] `http://jakarta.apache.org/regexp/`

Usually, more general rules without any explicit constraints (such as class- or method names) result in a larger result set. Thereby, the probability that comments overlap increases. Motivated from this issue, the following subchapter primary cares about the scenarios when there is more than one comment per observed program construct available.

## 3.5 Rating of documentation entries

When querying the repository for matching documentation entries, it might often be the case that multiple documentation entries are available for the same context. Since we are thankful for every given hint and comment — at a first look — this might not be a problem. However, this situation is quite not optimal, since some documentation entries are more important than others but may be overlooked if they are not highlighted. Similar to a Web-Search engine, only the first listed results are particularly observed.

In such scenarios, it might be helpful for a developer to obtain all matching documentation entries ordered according to their relevance. This in turn requires a ranking heuristic for a rating procedure that evaluates every matching collaboration. Hence, in this chapter we want to elaborate a yet simple rating mechanism, that assigns relevance-points to all matching documentation entries that are used as a sorting criterion.

A similar discussion was already taken in [Tan03]. The paper presents a tool called *XSnippet* that mines for sample code (called snippets) depending on the current context. They report that "ranking heuristics are critical to ensure that the best-fit code samples appear as the top snippets". They distinguish between two distinct categories which are presented within the following sections.

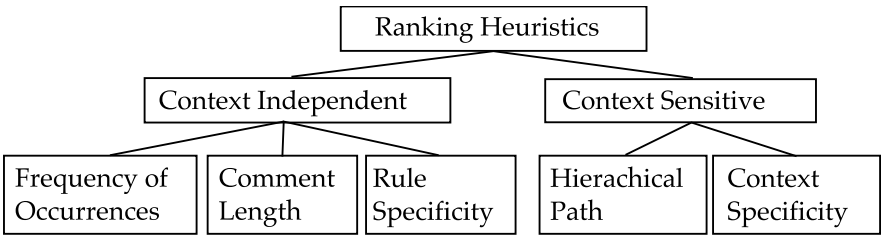


Figure 13: A Taxonomy of Ranking Heuristics

### 3.5.1 Context Independent Heuristics

Here, only the general (static) properties of the elements (to be ranked) without any relations to the matching context are considered. In [Tan03] these properties are "Snippet Length" and "Frequency of Occurrence". In our work, both approaches are employed — but in a somewhat modified

way since the context of code snippets differs slightly from the one of documentation. Additionally, a further property was gathered during the work on this section.

**Ranking by Frequency of Occurrence** Regarding the quantity of occurrences of a pattern inside the search domain as a relevance criterion is "a common notion, used in many diverse domains" [Tan03]. Consider, for example, a very general documentation entry that matches nearly every context, then this entry should be ranked below a more special one that describes a seldom pattern.

**Ranking by Comment Length** In our approach, the most extensive comment (regarding quantity of characters) that matches the context is higher rated as others. This idea is based on the assumption that longer comments usually provide more informations than shorter ones.

**Ranking by Rule Specificity** A lot of knowledge about the relevance of a given documentation entry can be gathered from it collaboration query that structural describes the pattern. We have actually identified two influences on that *Grade of Specificity*.

The first one is concerned about the chosen predicates. Regarding the core predicates of the employed logic programming language, some relations describe an element more specific than others. Primary the *name(?Elem,?Name)* respectively *re_name(?Elem,?NamePattern)* [1] and new introduced *rbd(?Elem, ?SourceMarker)* (to explicitly address special marked program constructs) predicates allow a very definite pattern specification.

The other factor regards the predicate coverage of the employed logical variables. Since every predicate imposes a relation on the involved elements, the more occurrences we count of a certain variable in different relations, the higher is the *Grade of Specificity* for that variable. The overall index can be calculated by dividing the amount of predicates through the amount of distinct variables employed in a pattern query.

By combining both ideas, one can determine an index that seems to be an adequate sorting criterion.

### 3.5.2 Context Sensitive Heuristics

So far, only the static properties of a documentation entry are observed. However, the grade of matching to the actual context remains uncovered.

[1] re stands for regular expression

Obviously, these fields of heuristics are more important than the ones discussed in the previous section. During a brief analysis, two distinct approaches are elaborated.

**Ranking by Hierarchical Path** In this work, it has been discussed how to define comments for source code elements in a collaboration. Retrieving a comment that matches the context, however, was not yet covered. Concerning our approach, we have chosen to make comments hierarchical, in a way that a comment for a particular element also holds for all its containing elements [1], thus building a cascade. Consider, for example, a textual explanation for a certain class — then this comment should also fade in on every containing element such as fields or methods and so on, since they are parts of the documented class. This meets the first requirement we set up at the initial requirements report in chapter 3.1. A motivation for this decision was to increases the probability that there is a comment for a certain source code element available.

Regarding the relevance aspect, items that are fetched indirectly through a cascade down the hierarchy are usually less relevant. Consider, for example, a comment for a parent package and another that exactly covers one particular field matching the context (the cursor position inside the editor), then the package comment is obviously less relevant as the field comment and should be ranked below.

**Ranking by Context Specificity** Similar to the same discussion in the section about Context Independent Heuristics, this approach also calculates a *Grade of Specificity* but here only for the matching context, which is internally represented as a logical variable bound to that context. If we filter out all predicates employing this variable, we are able to calculate a *Grade of Specificity* on that subset analogously to considerations taken in section Context Independent Heuristics.

Now, as a catalogue of different ranking heuristics was elaborated, the importance of each ranking rule need to be assessed to provide better results. Since the relevance of a documentation cannot computed exactly and additionally depends on the intents of the documentation writer, the following consideration are quite informal and based on common sense ideas. Because of this fact, the proposed procedure can be regarded as an optimization but not as a solution for the present problem.

Tansalarak et. al in [Tan03] already reported that "context-sensitive heuristics provide better ranking for the best-fit code snippets than the context-independent heuristics." This assumption should be also preserved when providing relevance points for documentation entry.

[1] subtree of the AST representation with the matching node as its root element

Obviously, it is meaningful that a context matches directly and not through an parent element like a parent package or class. So, we decide to assign ten points to every documentation entry that fetches in a direct manner. Second, the Grade of Specificity of the context matching program construct inside the pattern role flows in by a factor of two.

Regarding context independent approaches, the documentation entry holding the longest comment for the current context additionally receives two points. Ranking the amount of pattern occurrences is not that easy because there might be significant variations in these amounts. Therefore, entries are ordered by their occurrence count and – based on this – divided into five buckets. Subsequent, they are rated from one up to five points according to their position in that bucket order. Finally, the calculated Grade of Specificity amount of the collaboration rule gets added to the overall relevance score.

## 3.6 Conclusion

This chapter conceived a novel approach in the area of internal software documentation allowing to document source code in context to a collaboration on the interface level. By using an established model in the area of object collaboration, we have introduced our ideas in chapter 3.2.2. Technically, the approach is based on a separation of programming code and textual comments and uses a formal pattern description to apply a loose couple relation between both concepts.

All involved elements and their relations to each other are conceptually introduced using a domain model as presented in section 3.3. Based on that model, the subsequent chapter derived a language definition allowing to write down documentations following the ideas of this work. Finally, some optimization aspects are discussed inside section 3.5 by elaborating a yet simple heuristic that can be used to order matching documentation entries by their relevance.

Footing on these considerations, the following chapter will design and implement a prototype as a plug-in for a common IDE.

# 4 Tool Implementation

To actually benefit from the new possibilities of this novel documentation approach, this chapter cares about turning the concepts elaborated during the previous chapter into practice. The goal is to directly assist the developer when browsing and writing code with a source code editor, so we choosed to implement the tool as a plug-in for the Eclipse IDE which can be regarded as a de factor standard environment (not only) for Java development. In addition, the platform is known for its good extensibility through the plug-in concept.

This chapter covers the tool implementation and integration by outlining some particularities, discussing some important design decisions and presenting the features of the final tool. The final plug-in itself is named *RuBaDoc*, which is an abbreviation for **Ru**le **Ba**sed **Doc**umentation.

For an introduction, the first section briefly presents the Eclipse platform as the technical foundation of the proposed tool.

## 4.1 The Eclipse Platform as a foundation for RuBaDoc

"Eclipse is a universal tool platform - an open, extensible IDE for anything, but nothing in particular"[1].

A center concept of Eclipse is the plug-in, which can be understood as the fundamental building block. To cite from the project website [2]: "Eclipse is the sum of its constituent plug-ins". Due to the fact that all plug-ins have to follow a certain specified assembly, extending the platform is somewhat comfortable. Further, Eclipse itself highly eases the procedure of writing such extensions with a plug-in as well. The plug-in development environment (in short PDE) is an environment that generates a bunch of scaffold documents by using a set of forms prompting for all relevant plug-in information. This eases the process of writing a plug-in manifest that defines the plug-in environment such as dependencies, used platform extension points and the functionalities that this extension contributes to the platform. Hooking into Eclipse is achieved through the concepts of extension points that allow a loose coupled connection between the platform core and the custom plug-in code. Eclipse exposes a lot of different extension points that can be extended or customized.

A detailed overview about available extension points and the Eclipse architecture is somewhat beyond our scope, so that we now want to leave this introduction and further care about the usage of the given mechanism.

[1] taken from `http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html` on 2007-11-20

[2] `http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html` on 2007-10-11

## 4.2 The RuBaDoc Eclipse Plug-in

This whole section is primary concerned about examining behavioral and structural issues of RuBaDoc and how to properly hook into Eclipse to offer access to the documentation repository. In general, there are two sorts of concerns consisting of visualizing and managing (create, retrieve, update, delete) documentation entries.

In general, the RuBaDoc extension builds on top of the JQuery Eclipse plug-in [1] developed at the University of British Columbia (`http://www.ubc.ca`). JQuery is an environment for source code queries based on a logic programming language called TyRuBa and is required to realize the Code–Comment relation. Within this prototypical implementation, it is not planned to make this query mechanism exchangeable. This could be a topic for future work.

This chapter is outlined as follows. At first, we discuss the behavior of the tool with respect to non functional requirements such as performance. Subsequent, we will care about structural aspects of the plug-in by elaborating and discussing an application model as a basis for a later implementation. The class diagram also shows how the whole architecture is decomposed into packages and explains their particular responsibilities. Section 4.2.4 is concerned about the interchange format of RuBaDoc and primary discusses how to consult documentation documents written according to the grammar defined in section 3.4. The last four subchapters mainly care about visible aspects of the documentation plug-in and how they are realized using the appropriate Eclipse extension points.

### 4.2.1 The Consistency of the Code–Comment Relation

The idea of this thesis is to fade in documentation entries context sensitive. In conclusion every context change event forces an update of the currently visualized entries. By term ‘context’ we understand the program construct currently covered by the cursor in the editor of the IDE. Typically, the cursor position in the editor is transient and a context change event occurs rather frequent. Since the relation between comment and source code is unidirectional from comment to code, checking for matching entries requires to iterate over all registered documentation entries to look for intersections.

A documentation entry matches the context if at least one of it variable binding – that satisfies the goal – directly or indirectly (i.e. hierarchically) equals it. To diagnose a ‘context match’ of a documentation entry, the collaboration defining query need to be executed in order to fetch the compatible variable bindings representing program constructs involved in the collaboration.

[1] available at: `http://jquery.cs.ubc.ca/`

Due to the fact that our approach bases on *Logic Programming*, such a query is not evaluated directly on source code, but on a logical representation of it which is called the fact base. This fact database only represents one snapshot of the project workspace. Hence, every source code modification need to be pushed to the fact base in order to be consistent. In our scenario, one single source code document usually comprises a lot of logical facts that need to be represented in the fact base. So, big software systems result in enormous fact bases, that become unhandy for reasoning procedures. Because of the dependency of a query from the fact base, reevaluating a query requires a prior refresh of the fact base. Considering the perceptible time and memory resources required by both procedures, it is technically not realizable to evaluate collaboration queries every time documentation entries are requested.

Therefore, compromises concerning the consistency aspect of the Code—Comment relation need to be taken, since this seems to offer the biggest optimization potential. Or in other words, the relation need not to be up-to-date at any time. In particular, a deferred approach will be applied where all documentation entries are initially charged with their corresponding source code mappings.

In conclusion to that, each time source code gets manipulated by the developer, some documentation entries may become inconsistent in a way that they cover collaborations that no more exists. Therefore the Code–Comment relation need to be refreshed on certain triggered events. To look for such events, this section examines common use cases and thereby additionally covers the behavior of the tool.

Figure 14 depicts an UML Use Case diagram that is a foundation for the following discussions.

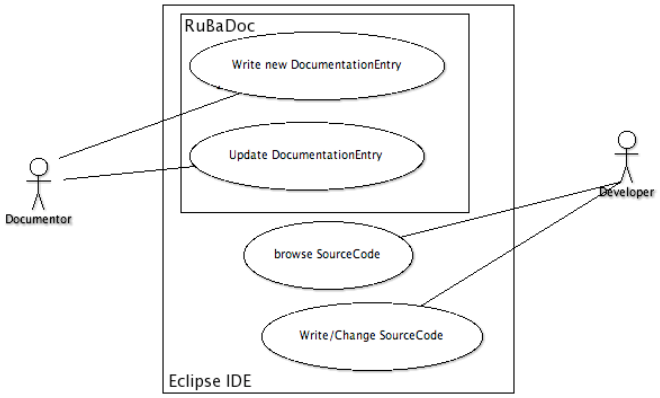


Figure 14: Use Case Diagram of the working scenario

Subsequent, we address each interaction scenario and describe the internal behavior of the RuBaDoc plug-in.

**Initialization** At first, we will discuss the startup protocol of the extension. In short, this use case is mainly about building a fact base, loading documentation documents automatically from hard drive and fetching all pattern occurrences internally (i.e. evaluating all queries).

The whole initialization process is somewhat time consuming and memory intensive. It will start by doing a structural analysis transforming the whole specified working set into logical facts. Subsequent to this transformation, the tool identifies and reads in documentation files following a certain file type convention. Our designed suffix is `.rbd` which is an abbreviation for **r**ule **b**ased **d**ocumentation.

The tool is designed to read in all found `.rbd` documents and merges them internally to one big repository. This makes it possible to structure files by covered packages, language or something else. In addition, a desirable side-effect of this procedure is, that it allows developers to write their own documentation to their contributed modules. Later, other team members can easily include these explanations by just placing the file in their work directory.

After parsing all entries and building up an internal representation, the query rule of each documentation entry of the repository becomes evaluated. In this deferred approach, all pattern occurrences will be pre-retrieved from the working set source code and attached to the corresponding documentation entries at startup. Later, when responding to a context change event, it is only need to consider this cached occurrences and thus avoiding a cost intensive query evaluation. A drawback of this decision is that tool startup time will increase somewhat.

**Editing/Exploring Source Code** Editing and exploring source code may trigger a context-changed event that demands adapted comments inside the documentation view. As already mentioned, by the term *context* we refer to the element currently covered by the editors cursor. In our implementation, qualified names are chosen to represent a context internally. These URIs locate a certain source code construct by its hierarchical path. Every time this context changes, all currently visualized documentation entries need to be updated.

Due to the fact that a context-changed event occurs rather frequent, we will work on cached results. Hence, code that was written after a cache refresh does not affect the documentation. This is not dramatic, since these changes might be present in the short term memory of the developer and therefore demand no further explanation. In the opposite case, deleting code results in orphan documentation, that indeed waste memory but usually does not mislead the developer. However, it is problematic if code was deleted or edited (in a way that

breaks the rule of the collaboration) that was previously a unique part of a certain collaboration. In this case, other participants are covered with documentations about a collaboration that no more exists.

For such scenarios, we will demand an action that forces a complete refresh of the whole fact base prior to a reevaluation of all Code–Comment relations. A partial refresh of only the affected relations is unfortunately not possible, since the impact of a particular source code modification on a documentation entry is not easily predictable, respectively only by a query reevaluation.

**Adding a new Documentation Entry** Besides of writing new documents using the language specified in chapter 3.4, we decide to provide some higher level tools that are more ergonomic than sole text files laying on the file system. In particular, we think about a form dialog that prevents common mistakes and supports the task of documenting by outlining a common path.

Since documentation entries are independent of each other, a refresh of the deferred occurrences of the other entries is not required after adding this new entry. Section 4.2.8 further discusses some implementation issues about the input form dialog.

**Editing a Documentation Entry** Analogous to the "New Entry" dialog described above, there should also be a dialog that makes the procedure of editing documentation entries more comfortable. Considering the performance aspect of the proposed plug-in, there are two cases of editing a documentation entry. In a first case, there will be only changes to the textual explanations or labels. This scenario can be handled by simply adjusting the objects properties. In a second scenario, when also the collaborations logical rule was modified, we need to remove all gathered collaboration instances and force a following rule reevaluation. Because of the independence between different documentation entries, further synchronization actions are not required.

Within this section, we pointed out different behavioral aspects of the proposed tool. The chapter presented a compromise between consistency (referring the Code–Comment relation) and performance. The allocation of documentations to source code needs not to be up-to-date at any given time. In conclusion, a mechanism is needed to trigger a complete refresh of these relations.

### 4.2.2 Architecture model

The following architecture model combines the core model entities, previously defined in the domain model on figure 3.3, with classes primary con-

cerned about environmental aspects (such as visualization, read/store and so on). The model is structured in layers represented via packages. These layers are ordered according to their information flow bottom-up. So, components (may) depend on information from other components below. The concrete responsibilities of each package will be discussed afterwards. The class model depicted on figure 15 provides an overall view over the RuBaDoc architecture.

This simplified UML class model only shows the core entities that constitutes the tool. There is an amount of other classes that mostly deal with issues concerning JQuery/TyRuBa, JavaCC, or Eclipse that are currently not visible and somewhat beyond the current scope. In following description, the tasks and responsibilities of each implemented package will be reported.

**de.tud.inf.st.rubadoc.io** The `io` package is responsible for storing and reading documentations to respectively from the file system. A facade object for that facility is `DocumentationIO`, whose interface exposes methods to serialize the current documentation repository to hard disc and also import them back to runtime objects. Other auxiliary classes, mostly concerned about the parsing procedure, are omitted, since they are generated using third-party tools. Chapter4.2.4 further deals with the implementation issues of the documentation parser.

**de.tud.inf.st.rubadoc.model** The `model` package consists of the core entities, that were already identified in the domain model in figure 10. The `Query` object encapsulates the access to the underlying resoning facility (here JQuery/TyRuBa). To keep the core model entities small and clean, we decided to realize additional behavior by external classes following a *Visitor* design pattern. The `DocumentationRepository` refers to the previously presented `Documentation` entity, and is implemented as a *Singleton*, since there should be only one such a repository at one time.

**de.tud.inf.st.rubadoc.rating** The `rating` package is conceptually located between the `model` and `controller` package and can be seen as an access port to the result set of matching documentation entries. The package is concerned about rating documentation entries according to their relevance. The aggregate of rated documentation entries can be uniformly accessed in a desired order via an *Iterator*.

**de.tud.inf.st.rubadoc.controller** According to the *MVC Architectural Pattern*, the controller packages cares about responding and processing to respectively of events triggered by interactions with the development environment. In particular, these events are: changing the editor context (cursor), adding a new documentation entry, editing or removing a documentation entry and forcing a rule base refresh.

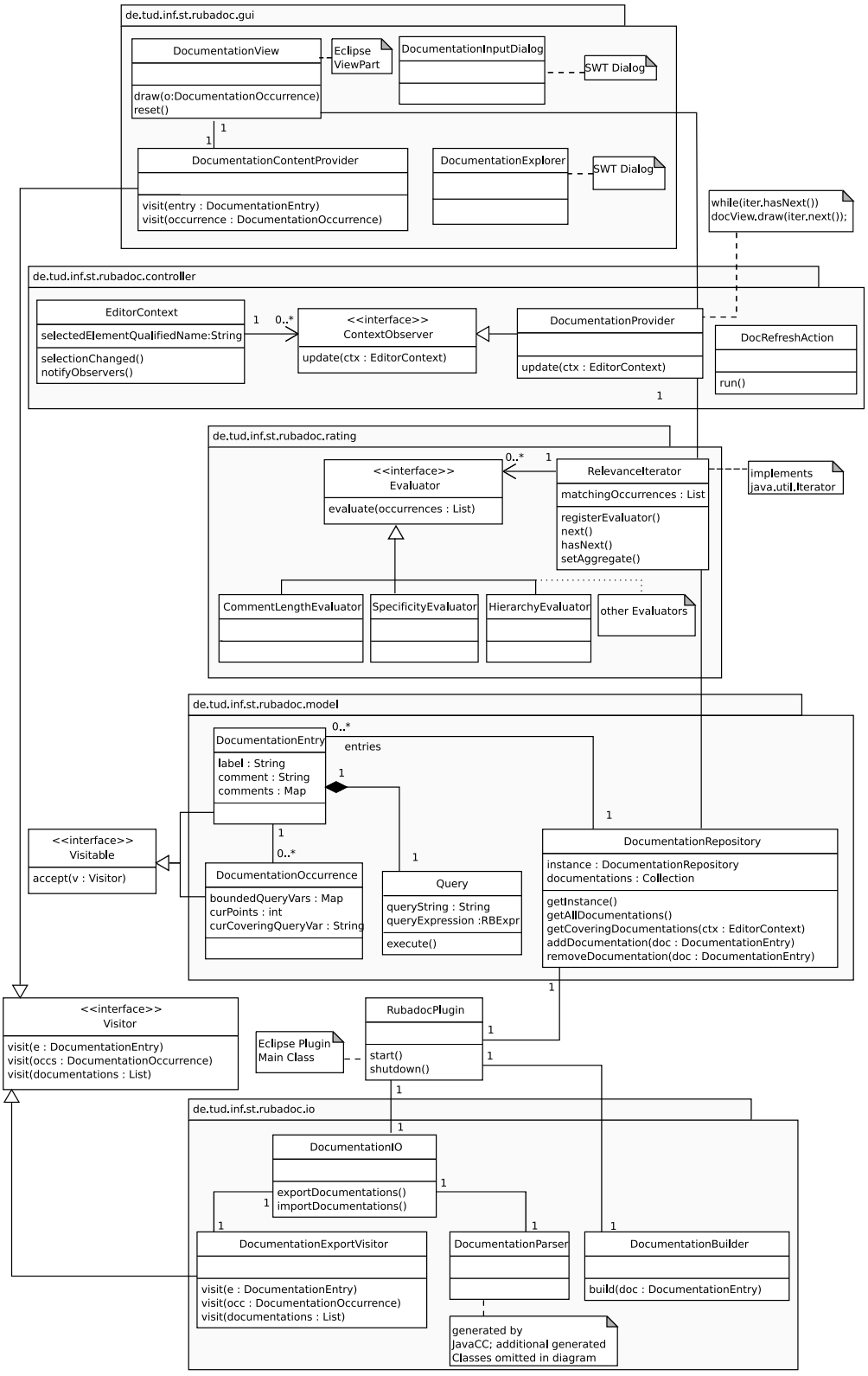


Figure 15: UML class diagram of the RuBaDoc core architecture

**de.tud.inf.st.rubadoc.gui** The `gui` package is responsible for visualizing the model objects and additionally provides forms to manipulate model properties. These classes strongly depend on the Eclipse environment, thus they implement the exposed extension points. The concrete display elements are topics of the chapters 4.2.5, 4.2.7 and 4.2.8.

### 4.2.3 JQuery Integration

In this work, JQuery was chosen as the underlying reasoning environment. The JQuery language

> "Is a logic (Prolog-like) query language based on TyRuBa. TyRuBa is a logic programming language implemented in Java. The JQuery query language is defined as a set of TyRuBa predicates which operate on facts generated from the Eclipse JDT's abstract syntax tree." [1]

Causes for this decision include:

- JQuery is an Eclipse plug-in too. Hence, some functionality could be easily reused.
- With a history of (now) six years of continuous release cycles, the query environment can be regarded as good maintained
- The employed logic programing language TyRuBa implements some optimizations such as tabling (also known as lemmatization or memoization) [2] which is a kind of a cache mechanism that prevents repeated executions of already evaluated relations. In addition, TyRuBa is able to reorder literals in a query and uses indexing mechanisms to optimize the query execution time.
- TyRuBa is a typed programming language, so it can provide more comprehensible error messages.
- JQuery is freely available as an Open Source project under a BSD License.

During the implementation we employed the following ground features of JQuery:

- The environment to build up a fact base
- The interface to TyRuBa for evaluating a query on this fact base

[1] taken from `http://jquery.cs.ubc.ca` on 2007-12-14
[2] `http://www.cs.sunysb.edu/~warren/xsbbook/node2.html`

- And finally a trigger-able action that forces a complete fact base refresh.

We do not need all visualization aspects such certain views or dialogs the plug-in originally contributes to Eclipse.

Further, JQuery was modified to more fit our needs. For the purpose of collaboration mining one do not need access to the entire AST. So, the procedure that transforms Syntax Trees to fact base predicates gets modified in a way to minimize the amount of generated predicates. Minimizing the amount of predicates results in three desired side effects:

1. Since a lot of nodes are bypassed when traversing the AST, the time needed for plug-in initialization can be reduced.

2. Less memory resources are required due to the reduced fact base.

3. Queries can be evaluated faster, because there are less predicates to consider.

Because of the following reasons, these listed predicates will be suppressed:

**Marker Predicates** All different kinds of markers such as Tasks, Bookmarks, Compiler Errors or -Warnings will be not represented in the fact base since they do not provide any design information.

**Source Locations** Due to the fact that the current exact source code position of an element is a very transient information, the modified procedure will abandon such information in order to decrease the amount of fact base predicates

**JQuery Labels** JQuery additional inserts static textual informations to the fact base, such as "method getName() reads name" and similar. Since these informations can be also derived at a later time when needed, they will be suppressed. As measured using a project with nearly 20 kLoCs this reduces the size of the fact base by one-third.

**Java Core** The procedure should not further index the used JDK libraries. For example, if someone uses java.lang.String then he is usually not further interested about structural details of that used class. Also it makes less sense to document the JDK core files since the source code is usually not visible.

Besides these reductions, a new core predicate gets added to the language. By using `rbd(?Elem,?Ident)` the documentator is now in a position to additional specifiy queries using *Source Code Markers*. For that purpose, the rbd predicate considers doclet annotations following the form of

`@rbd Ident`. Actually, there should be also support for JavaDoc tags in JQuery, but in the current version the corresponding code is commented out, due to the JLS3[1] update. Therefor, we need to implement our own predicate support. Fortunately, JQuery has already designed special mechanism to declare new predicates (specify predicate arity and expected types). Additionally, to employ the predicate, we need to hook in into the Source Code AST traversal to absorb the special formated doclet nodes. To explain the semantics of the new predicate, we will discuss a short example depicted in listing 2.

```
1  package de.sample;
2  /**
3  * @rbd SpecialClass
4  **/
5  public class DummyClass implements IDummy {
6          private Node accessPort;
7
8          public int dummy(int b) {
9                  accessPort.getChild(1);
10                 return b;
11         }
12 }
```

Listing 2: A simple Java Class

Based on listing 2, it is now possible to cover all collaboration with a class annotated with `SpecialClass`. Listing 3 demonstrates the usage of the new predicate by a simple example.

```
1  RbdTagCollab [ calls(?CallerM,?Method,?),
2          method(?Class,?Method),rbd(?Class,SpecialClass)]
3  {
4          CallerM : "This method calls a class that is
5                     annotated with SpecialClass";
6          Method  : "A method inside SpecialClass that
7                     gets a call from %CallerM%";
8          Class   : "This is a special class annotated
9                     with SpecialClass";
10 }
```

Listing 3: Documenting a simple Collaboration using the `@rbd` tag

Here, covered collaborations are restricted to have a class with a "SpecialClass" rbd annotation that receives a method call from an other class. This approach follows the ideas presented in section 2.2.1 and allows a very strict Code–Comment relation using a unique identifier as a tag value.

To measure the impacts of these changes JQuery was enriched by additional logging code. To get an insight to the procedure of transforming an

[1] Java Language Specification, Third Edition

AST to logical facts, listing 4 lists a subset of the predicates generated after consulting the class definition depicted in figure 2.

```
child(de.sample::Package, de.sample#DummyClass.java::CU)
class(de.sample%.DummyClass::RefType)
modifier(de.sample%.DummyClass::RefType, public)
implements(de.sample%.DummyClass::RefType,
        de.sample%.IDummy::RefType)
rbd(de.sample%.DummyClass::RefType,SpecialClass)

field(de.sample%.DummyClass#accessPort::Field)
child(de.sample%.DummyClass::RefType,
        de.sample%.DummyClass#accessPort::Field)
modifier(de.sample%.DummyClass#accessPort::Field,private)

method(de.sample%.DummyClass#dummy(int)::Method)
child(de.sample%.DummyClass::RefType,
        de.sample%.DummyClass#dummy(int)::Method)
modifier(de.sample%.DummyClass#dummy(int)::Method,public)
methodCall(de.sample%.DummyClass#dummy(int)::Method,
        de.sample%.Node#getChild(int)::Method,
        "=Proj/src<de.sample{DummyClass.java(25,22,6)"
        ::ca.ubc.jquery.ast.SourceLocation)
returns(de.sample%.DummyClass#dummy(int)::Method,
        int::Primitive)
params(de.sample%.DummyClass#dummy(int)::Method,
        [int::Primitive])
```

Listing 4: fact base representation of Listing 2

To evaluate the modifications done, a sample project (JHotDraw) consisting of approximately 20.000 lines of code was consulted. Using an untouched version, we measured 69.886 generated facts taking 15.716 ms of time. After implementing the mentioned changes, the JQuery analysis produces 42.852 fact base predicates taking now 12.957 ms in time. In conclusion, the new fact base is now only 61 percent of the former size and takes now 82 percent of the time to be build up. However, based on diverse measurements, the time need to evaluate a query on that fact base could not be decreased. This may be due to the optimization employed in TyRuBa that may bypasses unneeded predicates.

### 4.2.4 Documentation Parser

Writing a parser that is able to read in documents written in our designed language by hand, is error prone and not flexible enough to answer possible changes that may occur in the future. Instead of a hard coding approach, we decide to generate our parser automatically, which seems to be a proven solution.

The parser generator of choice is *JavaCC* [1] that is a popular parser generator written entirely in Java. The project is available as open source (*BSD* license) and has a long history (roots until 1996). In addition, there is a lot of documentation, tutorials and examples available.

As an input, for the procedure of a parser generation, JavaCC needs (of course) a grammar that describes the language of the documents we later want to read in. Such a grammar file defines the relations between a set of tokens that constitutes our final language, by using the Backus Naur form (BNF).

In our work, we just transform the visual designed grammar (depicted in figure 11 on page 39) to a textual notation and merge it with some additional Java code that is needed to build up an internal representation of the written concepts. An excerpt from the final JavaCC document is depicted in listing 9 in appendix B.

### 4.2.5 Eclipse Documentation View

The context sensitive documentation view is the most important visible component of the plug-in, because it displays all matching documentation entries. The view is responsible to visualize the important properties of a `DocumentationEntry` object including:

- The name of the documentation entry
- The subject of the documentation (the element that is documented)
- A general comment covering the whole collaboration
- A specific comment that solely covers the current context of the collaboration
- A list of collaboration participants (each with a specific comment and additional informations concerning the relationship between the participant and the collaboration)

Technically, the component uses the `org.eclipse.ui.views` extensions point and directly inherits from `ViewPart`. Since their might be multiple matching collaboration, the informations are packed into *Group* containers, so that each one represents one `DocumentationEntry`.

Possible the most important part of such an entry for the developers might be the participants list, thus it refers to associated source code positions by using hyperlinks. This is a major contribution for exploring a software project consisting of a lot of interdependencies. Besides these hyperlinks, the participant explorer component is able to reason about why

[1] `http://javacc.dev.java.net/`

the element is fetched as a participant for the present collaboration. In particular, it translates the subset of predicates, that impose a relation on the observed participant, to a natural language. Therefore, besides of the comments given by the author, the collaboration is able to document itself.

The screenshot on figure 16 shows the *documentation view* right hand to the source code editor in Eclipse. Due to the limited page dimensions of this report, common views such as `package` explorer or `outline` are omitted in the screenshot.

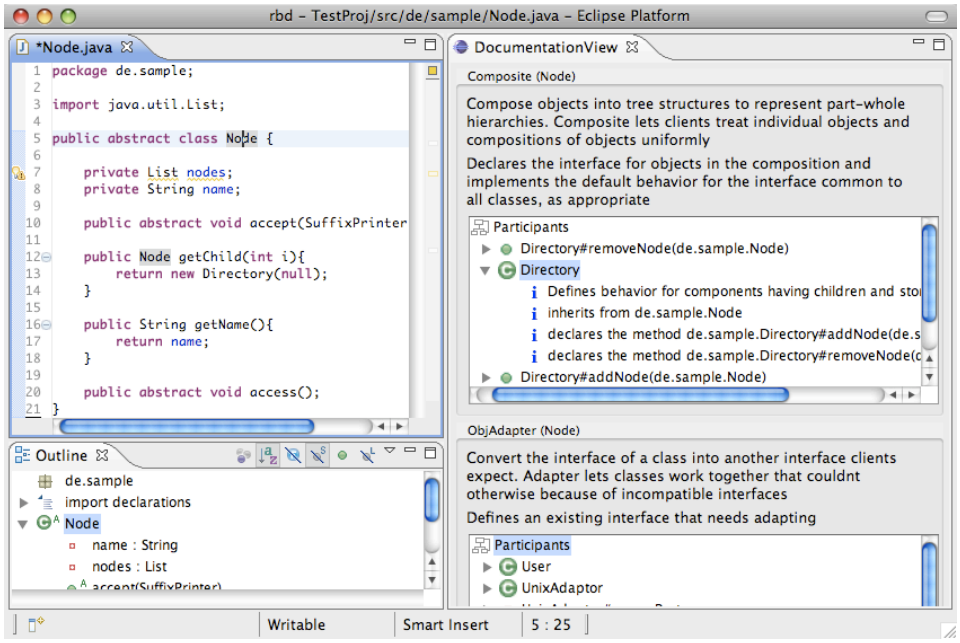


Figure 16: An screenshot of the Eclipse Documentation View

In the scenario depicted in figure 16, the developers cursor currently covers the `Node` class definition. So the actual context is `de.sample.Node`, which is the qualified name of the selection. Depending on that context the plug-in retrieves two matching documentation entries; namely an *ObjAdapter* and a *Composite*. After the documentation entries name, the matching element is printed in parentheses. This information is useful since we also allow an indirect matching (may be a comment for the package `de.sample`). The first label inside a documentation group-box describes the pattern as a whole, whereas the second prints a comment for the matching role part. Note, how comments are just taken over from the GoF book [GHJV95]. For the general collaboration comment, the *Intent* part was uses, whereas the different members get their explanations from the *Participants* section of the pattern catalogue. In screenshot, the `Directory` participant of file system tree structure (recurser node in composite) was inspected. The documentation plug-in now aggregates all information that

can be resolved. These are manual given explanations (the first blue '`i`') as well as automatically generated information (the remaining blue '`i`'s) from the given collaboration query.

### 4.2.6 Relevance Ordering Mechanism

All pattern matching documentation entries are not directly delegated to the eclipse view, but indirectly via a relevance iterator. We applied the homonymous design pattern from the GoF book that "provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation" [GHJV95]. The concrete iterator object is designed to do this traversal with respect to the given relevance points.

The action of providing points to resulting documentation entries is done preliminary by a set of evaluator objects. The whole mechanism was implemented with *extensibility* in mind. New rating mechanisms should be able to hook in easily to adjust the rating as a criterion for the relevance iterator. Technically, each concrete rating component applies an *Objectifier* pattern, thus implementing a common interface that imposes an `evaluate(DocumentationOccurrence occurrence)` method. Because of this, evaluators can be chained and thereby employ a flexible and extensible rating mechanism.

### 4.2.7 Documentation Explorer

The documentation explorer is the center administration tool responsible to browse and manipulate all known documentation entries from the repository. It uses a two frame layout consisting of a tree structure on the left side and an edit form for the documentation entry currently selected by the browsing tree. Besides of editing the objects properties, the form also allows to force a reevaluation of the collaborations formalization rule.

Figure 17 shows a screenshot of this component after selection a particular `Occurrence` inside the tree structure. One can see the resolved bindings for the used logical variables inside the *Query* input field above. If we prior selected the parent documentation entry inside the tree, the view container provides means to edit the comments for the particular collaboration members (logic variables).

### 4.2.8 Documentation Input Dialog

Writing documentation entries by hand using the documentation language proposed in chapter 3.4 might be unhandy and error prone. When writing the documentation by hand, the documentor always needs to keep track of all used logic variables, to be able to later attach a comment to each of them. If the role names in the rules section differ from those in the comment section, the given comments are lost in nowhere.

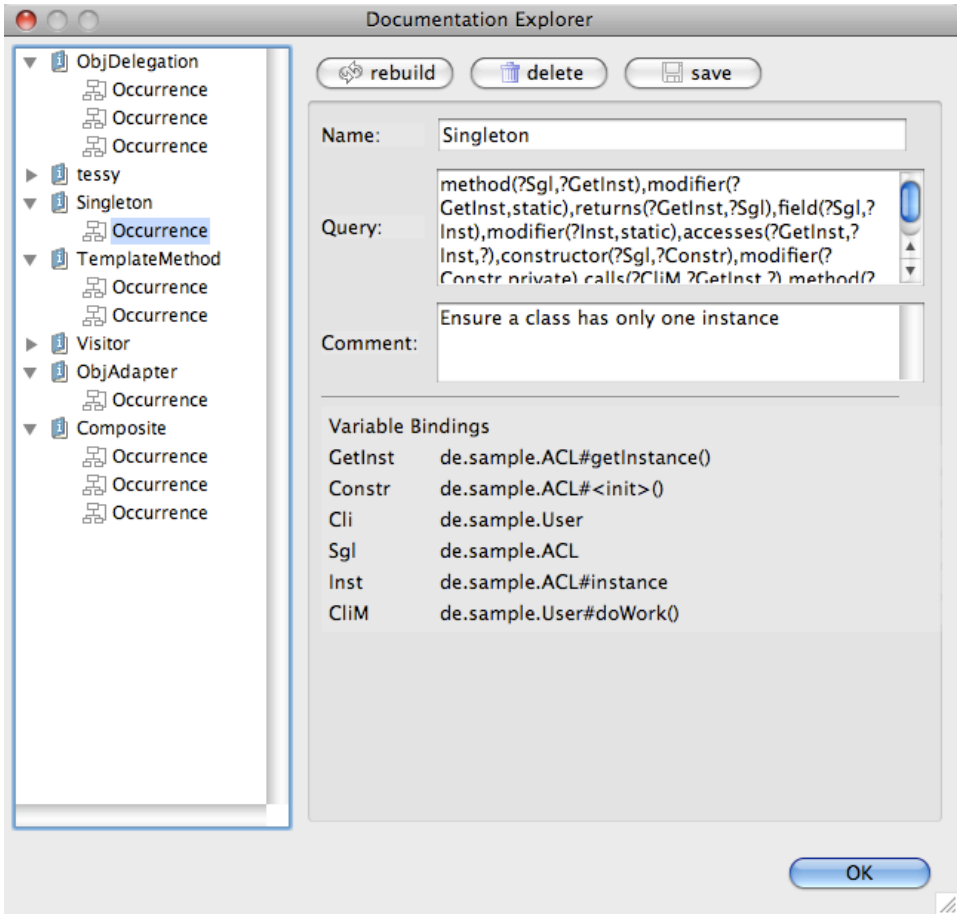


Figure 17: Screenshot of the Documentation Explorer after selecting a concrete occurrence from the tree

Major to this, the documentor needs instant feedback when writing the collaborations formalizing rule. For instance, if he uses a predicate which is not know, the engine should inform the writer as fast as possible.

In addition, we have not planed to force the documentation writer to learn a new domain specific language. It is commonly known that writing documentations was never really considered popular.

The procedure of adding a new documentation entry (using the dialog) is twofold. In a first step, the form asks for general informations; which are: the collaborations name, a logic rule defining the static structure and a general comment that covers the whole collaboration. After providing these informations, the system/dialog is prepared to extract all participants and automatically generating a new form asking for their particular comments.

Final to this procedure, the plug-in fetches all code fragments qualifying themselves for this new documentation entry. Technically, the query becomes evaluated and all instances of the described collaboration will be attached to the created documentation object. In a last step, the item gets added to the repository and the documentation text file (interchange document) becomes automatically synchronized. The final result of the input dialog is depicted on the screenshot 18.

The screenshot was taken after providing a valid rule, thus he dialog is now asking for textual explanations for the participating collaboration members.

## 4.3 Conclusion

This chapter has discussed some important implementation issues of *RuBaDoc*. The result is an Eclipse plug-in that directly assists the developer on browsing and writing source code. Internally, the extension is based on the JQuery plug-in that allows to select source code that matches a logic query. Footing on this facility, RuBaDoc builds an environment for software documentaion allowing to manage and visualize Documentation Entries.The next chapter will demonstrate the final plug-in in different case studies.

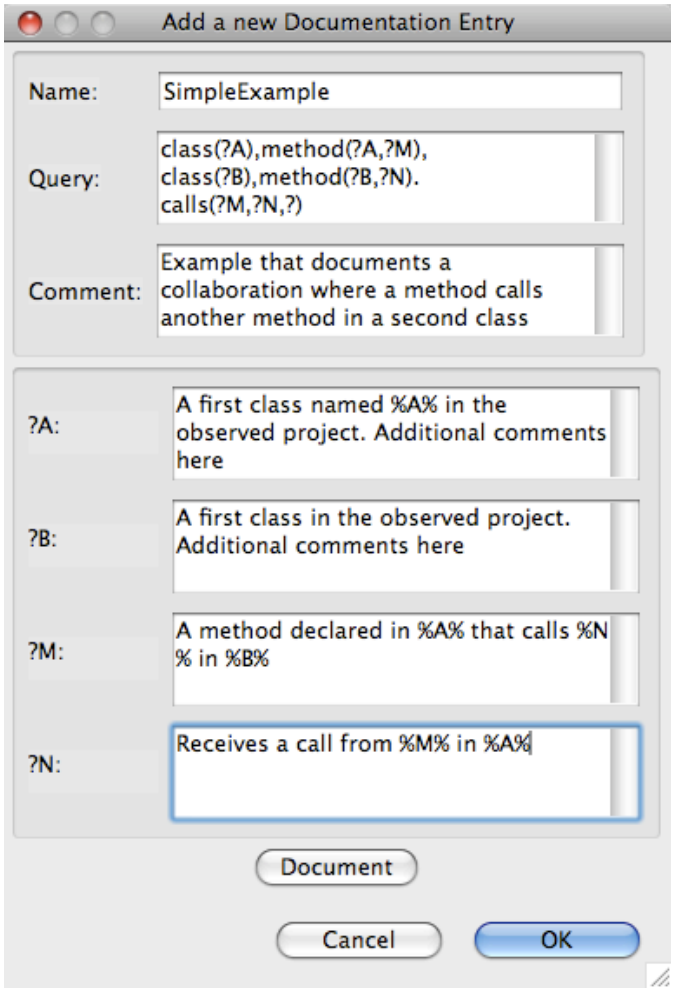


Figure 18: Dialog for writing a new Documentation Entry

# 5 Evaluation

Final to this work, the concepts elaborated so far will be proven in practice. In order to evaluate the capabilities of the RuBaDoc Eclipse plug-in this chapter examines three different case studies. The observed software projects strongly differ in their LoC (*Lines of Code*) amplitude ranging from 3 kLoC up to 150 kLoC. As a criterion for this discussion the study will document three different collaboration aspects of the software systems under study and examines quantitative benchmarks as same as the accuracy of the results.

The first case study comprises the RuBaDoc plug-in itself (only the core without any third party code). With somewhat above 3 kLoC, the projects admeasurement is quite small and should therefor not cause any performance problems. As a second project, the JHotDraw [1] source code was chosen, since it is known for its pattern richness. Finally, the study regards the source code of the Apache Tomcat Servlet Container [2], that was chosen to inspect the scalability aspect of the the underlying reasoning facility.

During the following experiments, the measurement results were obtain on a system with the following configuration:

- Apple MacBook Pro (2.2 GHz Core2 Duo Processor) running Mac OS X 10.5.1
- 4 GB of main memory
- Java 1.5
- Eclipse 3.3 Europa Edition (started with VM argument: -Xmx2048M)
- JQuery 3.1.14 (released: August 30, 2007)

To inspect the source code metrics of the observed projects, the Metrics Eclipse plug-in (version 1.3.6) [3] was used with the default settings.

## 5.1 Evaluation Criterions

This section briefly presents the three different collaborations that we want to cover during the analysis. They range from a rather general collaboration up to a somewhat more elaborated design pattern. Additionally, the following presentations also act as a tutorial for the proposed documentation language, since the written documentation entries will be briefly explained. The meanings of the employed TyRuBa predicates can be found in appendix A.

[1] available at: `http://www.jhotdraw.org`
[2] available at: `http://tomcat.apache.org/`
[3] available at: `http://metrics.sourceforge.net`

**Object Delegation Collaboration** The first use case starts with a quite simple example. In the scenario, the aim is to document an object collaboration where an object gets delegated to another, that in turn invokes a method on the given object. The whole relationship looks similar to the *Template Class* pattern, if we understand the delegated element as a *Hook Object* and the invoking method as a member of a *Template Class* definition. Additional, all participants that act as a client (triggering the procedure by calling the delegating method) should also be covered. The final documentation entry is depicted on listing 5.

```
1  ObjDelegation [ arg(?TM, ?D),
2          calls(?TM, ?CM, ?),
3          method(?D, ?CM),
4
5          method(?Client, ?DM),
6          calls(?DM, ?TM, ?) ]
7  {
8          *  : "In this collaboration a method invokes a
9                method on an object given as a param";
10         D : "This object acts as a Hook in a Template-
11               Hook constellation with method %TM%";
12         TM: "This method works on %D% that was
13               given as a param";
14         Client: "This is a Client that makes use of a
15               Template-Hook pattern";
16 }
```

Listing 5: Documenting a general object collaboration

The query section of the documentation entry (line 1 to line 6) captures all collaboration involved program constructs as logical variables and defines their relations to each other. The variable binding concept allows one to specify inter-relationships [1] between a set of different predicates. So, the first three predicates are connected via using the `?TM` respectively `?CM` variable to express that a method (`?TM`) with an argument of type `?D` (line 1) calls a method on the given argument (line 2-3). Somewhat decoupled, the last two relations cover the client role of that collaboration that triggers the procedure by delegating the object.

After having formally described the pattern, comments can be easily attached to a subset of the logical variables used inside the formalization. The variables represent parts of an object role that are members of the observed collaboration. Finally, the plug-in is now in a position to fetch this documentation entry if at least one of the variable substitutions that satisfies the query intersects with the current context (direct or indirect).

[1] a relationship between relationships (predicates)

**Design Pattern: Singleton** After having discussed a collaboration in general, this section presents how to cover a concrete Design Pattern known from the GoF book [GHJV95]. The Singleton Pattern – in its structure – is rather simple, since it consists of merely on single class. Similar to the previous entry, additional clients that make use of the singleton instance should also be covered with comments.

```
1  Singleton [ method(?Sgl,?GetInst),modifier(?GetInst,static),
2          field(?Sgl,?Inst), modifier(?Inst,static),
3          accesses(?GetInst,?Inst),returns(?GetInst,?Sgl),
4          constructor(?Sgl,?Constr),modifier(?Constr,private),
5
6          calls(?CliM,?GetInst,?),method(?Cli,?CliM) ]
7  {
8          *       : "Ensure a class has only one instance";
9          Sgl     : "defines the Instance operation
10                   %GetInst% that lets clients access its
11                    unique instance";
12         Inst    : "an unique single instance of %Sgl%";
13         GetInst: "a global point of access to %Inst%";
14         Const   : "No one instead of %Sgl% itself
15                     should call this constructor";
16         Cli     : "This class may act as a client for
17                    a Singleton pattern";
18 }
```

Listing 6: Documenting a Singleton Pattern

Note, how the key (structural) issues of the pattern are transformed to a logical representation (references in bold face).

1. A Singleton has a "operation that lets clients access its unique instance" ([GHJV95]) **line 1, line 3 and line 6**
2. A Singleton has a static field keeping the unique instance **line 2**
3. A Singleton class has a private constructor: **line 4**

**Design Pattern: Visitor** The last example was intended to examine the *regexp* engine (Jakarta Regexp) used in TyRuBa. A pattern that relies to a naming pattern is the *Visitor* with its *accept* and *visit* methods. As observable, explanations for the collaboration members are just taken form the *Participants* section of the GoF book ([GHJV95]).

```
1  Visitor [ implements(?ConcrV,?Visitor),
2          method(?Visitor,?VisM),re_name(?VisM,/^visit/),
3          method(?Va,?AccM),arg(?AccM,?Visitor)
4          (re_name(?AccM,/^accept/);name(?AccM,visit)),
5          calls(?AccM,?VisM,?),
6          arg(?VisM,?Visitable),subtype*(?Visitable,?Va) ]
7  {
8          * : "Represent an operation to be performed
9               on the elements of an object structure";
10         Visitor : "declares a Visit operation for each
11            concrete class in the object structure";
12         ConcrV : "implements a fragment of the algorithm
13            defined for the corresponding class of object
14            in the structure; provides the context for the
15            algorithm and stores its local state";
16         Va  : "Defines an Accept operation that takes
17            visitor as an argument";
18         AccM:"The accept method inside a Visitor pattern";
19         VisM:"The visit method inside a Visitor pattern";
20 }
```

Listing 7: Documenting a Visitor Pattern

The query between line 1 and line 6 can be interpreted as follows:

1. A concrete Visitor implements an interface that specifies some `visit` methods: **line 1 and line 2**
2. A visitable element in a object structure has an `accept` method with a `visitor` as argument type: **line 3 and line 4**
3. The `accept` method calls the `visit` visit method with a visitable element as argument: **line 5 and line 6**

Note that in line 2 and 4 we take some assumption about the concrete name of the *visit* and *accept* method of the *Visitor* pattern. Obviously, in general the pattern is not dependent on the concrete names of these methods. If they are named in a different way, than the collaboration might be still represent a Visitor pattern. However, this thesis assumes that the person writing the documentation is the same person that also implements the source code covered by the written documentation. Therefor, the actor has some design knowledge and knows about the conventions he applied when implementing the pattern.

## 5.2 RuBaDoc Eclipse Plugin

It this scenario, the source code of the final plug-in is documented by itself. The following evaluation only covers the core modules of RuBaDoc located

| Package | LoC | Size | avg Class/Interface size |
|---|---|---|---|
| `rubadoc` | 85 | 2 | 42.5 |
| `rubadoc.io` | 1306 | 13 | 100.5 |
| `rubadoc.gui` | 965 | 17 | 56.8 |
| `rubadoc.model` | 303 | 4 | 74.8 |
| `rubadoc.controller` | 210 | 7 | 30.0 |
| `rubadoc.rating` | 119 | 6 | 19.8 |
| **sum** | **2988** | **49** | **61.0** |

Table 1: Source Code metrics of the RuBaDoc plug-in

inside the `de.tud.inf.st` package that are implemented during chapter 4. So, any third party code (such as from JQuery or TyRuBa) does not affect the considerations. Table 1 provides some source code metrics from the systems under study.

The superior amount of LoC's inside the `rubadoc.io` package results from the use of a Parser Generator as discussed in section 4.2.4. In particular, the documents generated by the JavaCC (the Parser Generator of choice) environment comprises nearly 900 lines of code.

With an overall amount of 2988 lines of code, RuBaDoc can be seen as a rather small sized software project. As expected, the numbers gathered during different measurements are quite satisfying. The initial startup procedure that structural analyzes the whole project Working Set and transforms the containing elements to corresponding fact base predicates takes less than 3 seconds in time which is not that perceivable. Based on logging code, the RuBaDoc project source code produces 5.119 fact base entries. So, every source code line approximately produces 1.7 fact base predicates. Since most of the statements of a method body (`if` statements, `loop` statements, operations, assignments or similar) are not represented in the fact base, the ratio of "fact base predicates per fact base representable line of code" is considerable higher.

After having discussed, some general benchmarks, we now care about documenting the collaboration presented in section 5.1. As measured during the case study, the time needed for evaluating one single query on such a fact base is negligible and far below a hundredth of a second (concrete numbers can be found in the conclusion section of this chapter). All gathered occurrences of each of the collaborations match those known so far and are evaluated manually. In addition, the overlapping of the Visitor and the ObjectDelegation collaboration was detected and presented in a desired order by the ranking mechanisms.

The screenshot depicted in figure 19 show a matching Documentation

Entry that describes a *Singleton* Design Pattern. As observable, the current Editor context is `de.tud.inf.st.rubadoc.DocumentationRepository`. Regarding listing 6, the context equals with a substitution of the `Sgl` logic variable and is therefor fetched from the repository.

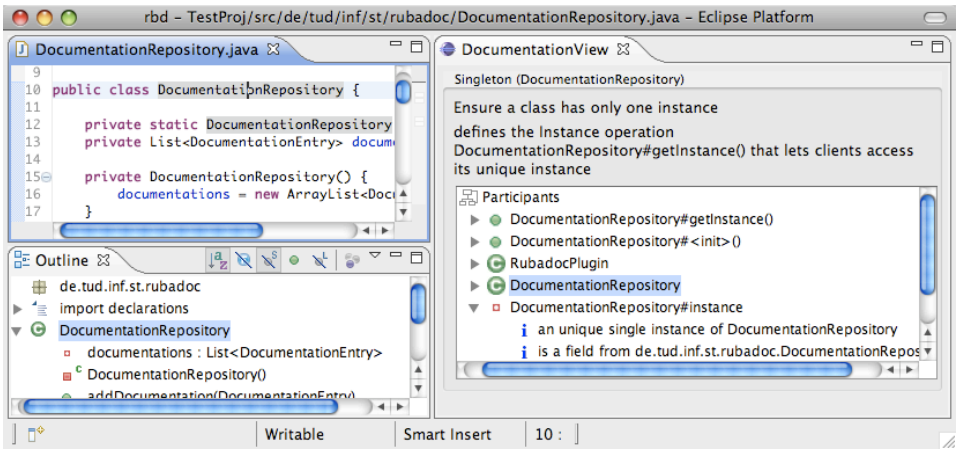


Figure 19: Screenshot of the Documentation Entry described in Listing 6

So in conclusion, considering the performance aspect (for small sized software projects) has turned out satisfactory.

## 5.3 JHotDraw

A common subject in diverse case studies of current work in the area of pattern mining is JHotDraw, because it is considered as a "show-case for good use of design patterns" [MMvD07]. JHotDraw is a graphics framework for drawing and manipulating a variety of figures. One of the authors is Erich Gamma who is also known as a member of the '*Gang of Four*' [1].

For the following examination, we used version 6.0 beta1, which is available as Open-Source on the project homepage: `http://jhotdraw.org`. Source code metrics for that particular version are presented in 5.3. The study only covers the core modules of JHotDraw. Packages concerning the separate JUnit test cases are excluded.

Regarding these numbers, JHotDraw is almost seven times bigger than RuBaDoc. In contrast to that, with 44.667 generated predicates, the size of the underlying fact base increases by a factor of 9. So one might assume that the fact bases grows non-linear with respect to the amount of lines of codes. But what is more important — regarding usability issues — with a duration of 12.893 ms the initialization for this project takes only a fourfold of the times consumed by RuBaDoc (measured in section 5.2). Further, the time consumed when evaluating a certain query is far below a second which

[1] coll. synonym for the authors of [GHJV95]

| Package | LoC | Size | avg Class size |
|---|---|---|---|
| `standard` | 5.295 | 93 | 57 |
| `contrib` | 6.823 | 93 | 73 |
| `figures` | 3.046 | 45 | 68 |
| `util` | 2.830 | 36 | 79 |
| `application` | 735 | 1 | 735 |
| `samples.javadraw` | 643 | 10 | 64 |
| `framework` | 521 | 5 | 104 |
| `applet` | 372 | 1 | 372 |
| `collections` | 257 | 6 | 43 |
| **sum** | **20.522** | **290** | **71** |

Table 2: JHotDraw source code metrics

is somewhat astonishing regarding the complexity of the chosen example queries and the fact base consisting of nearly 45 thousand entries. This might be owed to the optimizations such as indexing and caching that are implemented in TyRuBa.

Evaluating the detection of the patterns previously formalized in section 5.1 "requires knowledge of all pattern occurrences in the target code base". Since we do not have this knowledge, we rely on the findings reported in related work. For the Singleton pattern, for instance, Tsantalis et. al in [TH06] reported two different instances in JHotDraw.

This equals with the results of our measurements. In particular one of these is `FigureAttributeConstant` "for accessing a special figure attribute" [1] and the other is the `Clipboard` as a heap for transient data. Unfortunately, [TH06] regards JHotDraw in version 5.1 so we can not guarantee that we have covered all pattern instances.

There is a similar issue with the Visitor design pattern. For version 5.1, [TH06] reported only one detected instance in JHotDraw. Our study recovers three Visitor instances. This might be a reasonable result, since there are actually three distinct classes implementing the `FigureVisitor` interface.

In conclusion to this examination, RuBaDoc is capable to document a mid-range software project up to 20 thousand lines of code.

## 5.4 Apache Tomcat

Final to this evaluation, a rather big software system should be considered to study the scalability aspect of JQuery respectively RuBaDoc. For this purpose, the source code of Tomcat 6 (version 6.0.14 in particular) developed by the Apache Software Foundation was chosen. "Apache Tomcat is the servlet container that is used in the official Reference Implementation

[1] taken from the corresponding JavaDoc entry

for the Java Servlet and JavaServer Pages technologies" [1]. Since the package structure is somewhat more extensive, we currently forego a detailed metrics listing like done before with RubaDoc respectively JHotDraw. The overall amount of LoC is 155.118 source lines distributed on 1.376 classes or interfaces. Transforming the whole project source code into a logical representation requires 152.023 ms in time, which is critical but barely acceptable since the procedure of building a fact base usually occurs not that frequent. Fortunately, it seems that the initialization-time grows linear by 1 ms per LoC, so a project with 1 MLoC presumably takes a quarter of an hour. But studies such as [KHR07, HVdMdV05] reported that "JQuery has been found to be unable to work on large systems, such as the Eclipse sources" [2].

However — considering a project of that size — a developer asking for documentation often only works on a subset of all the packages to implement a certain feature. Therefor he can reduce the working set to only cover the packages he is currently interested in. Doing so, the amount of generated fact base predicates may become handleable and allows a much more faster analysis.

Nevertheless, considering the huge amount of 241.664 generated predicates (without Working Set reduction), JQuery satisfies us with an average of two seconds per pattern query evaluation. Will will later discuss possible reasons in the conclusion section of this chapter.

Unfortunately, we have no concrete knowledges about all pattern instances in the Tomcat source code. However, the instances detected by our queries are correct proven by a manual investigation. In particular, three instances of the Singleton pattern were covered with documentation [3].

Recalling a Visitor pattern in tomcat – however – was not that unproblematic. The studies reported only one Visitor occurrence in the whole source code, which seems to be too sparse. Notably the compiler package was expected to have some more Visitor patterns to handle the node tree structure. A manual study supports this suspicion because eighteen different classes with a `visit` method that calls an `accept` method were counted. This leads to a following fact base investigation.

Obviously, JQuery does not recognize method calls on collection aggregates. Listing 8 gives an example of code snippets that currently can not be adequately represented.

[1] cited from `http://tomcat.apache.org`

[2] The Eclipse Core 3.1 consists of 974,527 lines of code

[3] since one of these classes has three different clients we counted five occurrences of the pattern as formulated in listing 6

```
1 public void visit(Visitor v) throws JasperException {
2         Iterator iter = list.iterator();
3         while (iter.hasNext()) {
4                 Node n = (Node) iter.next();
5                 n.accept(v);
6         }
7 }
```

Listing 8: Documenting a Visitor Pattern

The used `list` field in line 2 is a non type-parameterized `List` of Nodes elements. However, the information that the accept call on line 5 is on a `Node` object is absent in the fact base (although line 4 assign the collection element to a concrete `Node` type). Due to the missing possibility to detect calls on collections, a couple of Visitor instances remain uncovered. Obviously, this applies for all patterns that strongly rely on uniform aggregates such as the Observer pattern. Hopefully future versions of JQuery might solve this task of detecting calls on collection aggregates. Since this work only provides a prototypical implementation to demonstrate the concepts proposed so far, we currently might see over this limitation.

## 5.5 Conclusion

This chapter demonstrated the practical usage of the plug-in using three different sized software projects as a subject for a documentation. For the discussions, different collaborations ranging from rather general to more specific patterns were examined. The studies are not representative, because different projects from different domains with different coding styles are used to compare the impacts of a varying LoC amount.

The quantitative benchmark results are summarized in table 5.5 and further visualized in figure 20. As observable, the amount of generated facts and the time consumed therefor grows linear, hence good scalability properties can be assumed. This is not surprising since there are no considerable additional expenses when consulting an additional source code document. The analysis mechanism extracts design artifacts (such as `extends` or `implements` et.al.) as well as calls and accesses to methods respectively fields. Transforming these information to fact base predicates is yet fast compared to initial creation of an AST representation. This phenomena explains the correlation between the last to columns in table 5.5.

| Project | LoC | FB pred | FB time | pred/LoC | time/LoC |
|---|---|---|---|---|---|
| RuBaDoc | 2.988 | 5.119 | 2.876 ms | 1.67 | 0.96 ms |
| JHotDraw | 20.522 | 42.852 | 12.893 ms | 2.09 | 0.63 ms |
| Tomcat 6 | 155.118 | 241.664 | 152.023 ms | 1.56 | 0.98 ms |

Table 3: RuBaDoc/JQuery Initialization Benchmark

The results of the particular query measurements are summarized with the following three tables. To prevent fluctuations, each scenario was measured twice (M1 and M2). However, the results seem to be stable since only a few variations are noted. Within the following tables, Occ denotes the amount of occurrences of the collaboration in the project source code. This number needs not to be equal to the amount of collaboration instance because of the reasons given in section 3.3. As observable, all different projects report correct (as manual proven) occurrences for the described design patterns. Hence, it looks promising to elaborate a catalogue of commonly used design pattern that can be activated as a makro. The last column calculates the query time per kLoC allowing a uniform comparison of the result to the other projects.

| Pattern | LoC | M1[ms] | M2[ms] | $\phi$M[ms] | Occ | M/kLoC |
|---|---|---|---|---|---|---|
| RuBaDoc | 2.988 | 30 | 31 | 30.5 | 9 | 10.207 |
| JHotDraw | 20.522 | 819 | 823 | 821 | 195 | 40.000 |
| Tomcat 6 | 155.118 | 3494 | 3489 | 3491.5 | 1204 | 22.509 |

Table 4: Evaluation: Template Class

| Pattern | LoC | M1[ms] | M2[ms] | $\phi$M[ms] | Occ | M/kLoC |
|---|---|---|---|---|---|---|
| RuBaDoc | 2.988 | 5 | 5 | 5 | 4 | 1.673 |
| JHotDraw | 20.522 | 18 | 19 | 18.5 | 19 | 0.901 |
| Tomcat 6 | 155.118 | 2022 | 2063 | 2042.5 | 5 | 13.167 |

Table 5: Evaluation: Singleton

| **Pattern** | **LoC** | **M1[ms]** | **M2[ms]** | **$\phi$M[ms]** | **Occ** | **M/kLoC** |
|---|---|---|---|---|---|---|
| RuBaDoc | 2988 | 5 | 9 | 7 | 2 | 2.342 |
| JHotDraw | 20.522 | 39 | 47 | 43 | 4 | 2.095 |
| Tomcat 6 | 155.118 | 354 | 372 | 363 | 5 | 2.340 |

Table 6: Evaluation: Visitor

Detecting a more general pattern takes significantly more time, since the query engine can not cut out so many solution paths as in a more specific one. Since JQuery is able to reorder predicates of a query, the usage of a name or even re_name predicate strongly reduces evaluation time. Such as in the Visitor pattern, a lot of classes can be disregard, when they do not have a *visit* or *accept* method. As observable in figure 21, detecting a Visitor pattern takes far below a second of time and has a straight linear growth with respect to the lines of code of the target source code.

Nevertheless, the results of all query evaluation (concerning a quantitative benchmark) are satisfying, regarding the amount of considerable data. This might be a consequence improvements implemented in the reasoning environment. In particular, Kniesel et. al in [KHR07] reported the following optimizations that are implemented in JQuery' underlying TyRuBa system:

1. The underlying TyRuBa query engine makes use of tabling to avoid reevaluation of certain already evaluated predicates. Hence tabling (also known as memoization or lemmatization) can be regarded a cache mechanism that might speed up the execution of a query.

2. TyRuBa uses index mechanism to prevent a sequential search for facts in the fact base. Indices allow a very fast access to a certain information.

3. TyRuBa reorders literals to early reduce the search tree of a query evaluation. The reordering of literals is based on the mode declaration of the predicates.

Additionally, the adequate amount of available memory might have improve the result. Logic Programming language typically require a lot of memory when evaluating queries. The 4GB memory of the test system additionally speed up evaluation time. Measurements on a different test system with 1.6GHz Core Duo and 1024MB of memory reported a fourfold of the time to recover Singleton patterns in JHotDraw.

Further, the result depicted in table 5.5 are visualized in figure 20 to clarify the correlation between the lines of code and the amount of generated fact base predicates (thick black line) respectively the time consumed to build up the logical representation accordingly (thick grey line).

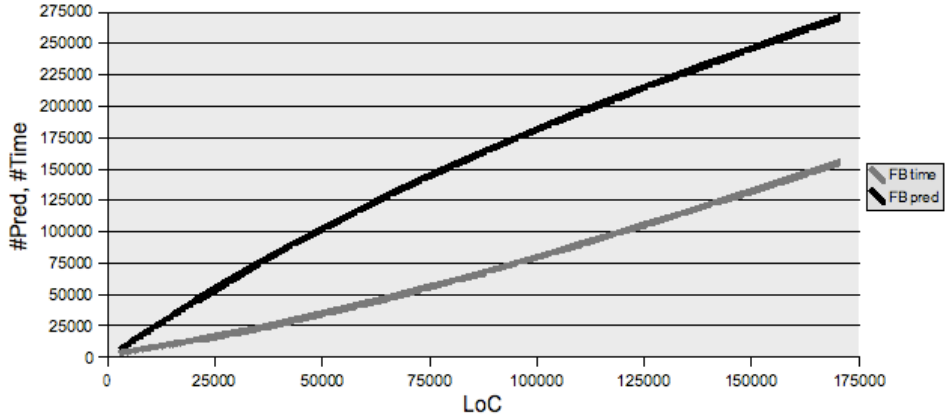


Figure 20: Time to build up a fact base with respect to an increasing LoC amount

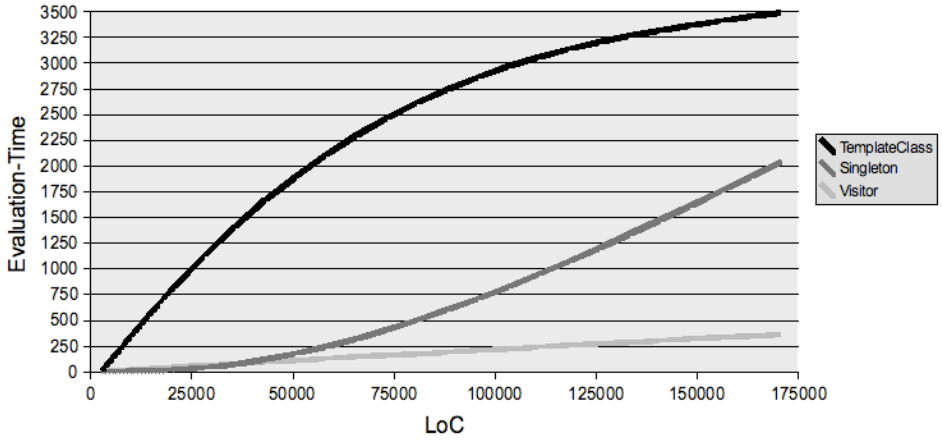


Figure 21: Measurements of the execution time of different collaboration queries

# 6 Conclusion and Future Work

This work presented a novel approach of an internal software documentation that aims to cover the collaboration aspect of a software system. Understanding the relationships between program constructs in todays software systems is crucial to get an overall view of the internal architecture. As analyzed in the problem specification (section 1.1), current means in the area of software documentation are not able to adequately cover these aspects of a software design.

This thesis demonstrated how a separation of source code and documentation aspects resolves the limitations currently prevailing when using inline comments to document a collaboration. But in contrast to existing work (such as [NAC+00, Wer07, Bar07]) that already realized such a separation, we employed an indirect Code–Comment relation that addresses subjects by their relations to other program constructs not directly by their position. At tool runtime, Documentation Entries fade in context sensitive. In case of multiple matching entries, this thesis elaborated a set of ranking heurists to sort documentation by their relevance.

Conceptually, this work bases on the ideas of the *Role Modeling* approach that is eminent when talking about object collaborations. A role model describes an object collaboration task by specifying the behavior of the participants and the relationships between them. The idea of this thesis is to document a role model with a documentation entry that attaches textual explanations to the involved members. In conclusion, comments only exists in context to a parent documentation entry. The relationships between Role Models and the proposed Documentation Model were explained in 3.2.2. We showed that by adopting this high level of abstraction, class level constructs become transitively documented when they conform to a documented object collaboration task. Further, we presented how this flexible Code–Comment connection solves the problems of *tangled* as well as *scattered* comments by applying an *M-to-N* relationship between both concepts.

However, *Role Models* are important at the analysis phase of an information system development to specify collaborative behavior that domain experts can easily validate. But when demanding documentation for a program construct, the collaboration is already implemented in source code and therefore not offhandedly identifiable. Therefore, we showed how to recover collaboration in source code using DataLog queries. Such queries formally describe the structure of a collaboration task with a set of relationships (namely predicates) between the involved elements (such as methods, fields, and so on). A set of these elements that belong to the same class definition represents an object role. Since we attached explanation to each

single element, we are working on a finer granularity than roles by documenting their constituent parts (such as role method, role fields, and so on). Basing on that model, we have designed a documentation language in subchapter 3.4 allowing to write down and share explanations.

Furthermore, the concepts formulated in this work are practically proven by a prototypical tool implementation in chapter 4. The Eclipse Platform provides an ideal foundation for this goal, because of its open plug-in architecture. In addition, the Eclipse IDE can be regarded as a de-facto standard environment (not only) for the Java development. Our extension builds on top of the JQuery plug-in which is a source code browser that allows to select program constructs by the means of a reasoning facility based on *Logic Programming*.

As a proof of concept, the tool has been tested on documenting the collaboration aspect of three different sized software projects. The case study in chapter 5 demonstrated the applicability or the plug-in. The benchmarks measured during these examinations approve the qualification of JQuery/TyRuBa as an underlying collaboration mining mechanism. The time needed to transform a workspace to a logical representation grows linear with respect to the lines of code. Due to the diverse optimizations implemented in TyRuBa, this holds for evaluation too [1].

The goal of the prototypical implementation was to demonstrate the ideas of this thesis in practice. At its current state, the tool is rather close coupled to the JQuery respecitvely TyRuBa project. It is considerable to ease this dependency to employ similar source code reasoning tools such as JTransformer [SRK07], CodeQuest [HVdMdV05] or even an own custom implementation.

In addition, the tool demands some further development to increase its usability. Currently, documentation only fades in when matching the context, thus only when the cursor inside the editor covers a programs construct that participates in a documented collaboration. However, it is also helpful to see all program construct in an opened source code document that are currently covered with documentation. One can think of special notification markers next to source line number inside the editor view or even special symbols in Eclipse' outline view.

[1] depending on the written query

Furthermore, one can think of an additional built-in documentation catalogue covering the most common collaborations. For this purpose, the formal description of the commented patterns need to be as general as possible to use them in such a universal way. Doing so, software projects may become documented automatically. In this scenario, developers profit from the use of RuBaDoc tool even if no one prior wrote a documentation for the project.

Another important issue, can be seen in the generation of an offline hypertext-based documentation. At the current time, making use of the written documentation requires Eclipse and the RuBaDoc plug-in. Nevertheless, one can also think of a hypertext based document that is easier to spread. Similar to the JavaDoc tool, all documentation entries and their covered occurrences could be extracted to generate a structured offline document. Instead of having class names as a browsing criterion (as in JavaDoc), one can think of the different collaboration as criterions for the left menu frame. Doing so, the documentation could be also published on a project website in is therefor not only accessible for developers using the Eclipse plug-in.

Development of the tool should be continued, since we personally believe that there is a significant need for such a kind of software documentation. There are barely no migration costs when documenting existing/legacy software systems afterwards. In addition, the target source code stays untouched when consulting a project to document, so there is no risk of unwanted modifications. Therefor it is never to late to applying this approach in practice.

# A Built-in JQuery Specific TyRuBa predicates and rules

This appendix list all of the JQuery/TyRuBa specific predicates that can be used to define the relationships inside a collaboration. The following tables are taken from the Documentation section of project homepage `http://jquery.cs.ubc.ca/documentation/appendix2.html`.

There are two different sorts of predicates. **Core Fact Predicates** "directly operate on the facts that are stored in the fact database". In other words, these are the same predicates that were also used when transforming the abstract syntax tree to a logical representation (the fact base). On the other hand, **Derived Predicates** (presented in the last table) "are useful predicates that have been derived from the core fact predicates using rules."

## A.1 Unary Core Predicates

| Predicate | Argument Type | Description |
|---|---|---|
| `cu(?X)` | CU | `?X` is a Compilation Unit |
| `package(?X)` | Package | `?X` is a package |
| `class(?X)` | RefType | `?X` is a class |
| `interface(?X)` | RefType | `?X` is an interface |
| `method(?X)` | Method | `?X` is a method |
| `constructor(?X)` | Constructor | `?X` is a constructor |
| `initializer(?X)` | Initializer | `?X` is an initializer |
| `field(?X)` | Field | `?X` is a field |
| `bookmark(?X)` | Bookmark | `?X` is a bookmark |
| `warning(?X)` | Warning | `?X` is a compiler warning |
| `error(?X)` | Error | `?X` is a compiler error |
| `task(?X)` | Task | `?X` is a task |

Table 7: Unary Core Predicates

## A.2 Binary Core Predicates

| **Predicate** | **Argument Type** | **Meaning** |
|---|---|---|
| `priority(?T,?P)` | Task, String | Task `?T` has priority `?P` |
| `name(?E,?S)` | Element, String | Element `?E` has name `?S` |
| `child(?Sup,?Sub)` | Element, Element | Element `?Sup` has a child `?Sub` |
| `extends(?C1,?C2)` | RefType, RefType | Class (or Interface) `?C1` extends Class (or Interface) `?C2` |
| `implements(?C,?I)` | RefType, RefType | Class `?C` implements Interface `?I` |
| `throws(?C,?T)` | Callable, RefType | Callable `?C` throws `?I` |
| `type(?T,?T)` | Field, Type | Field `?C` is of type `?I` |
| `modifier(?E,?S)` | Element, String | Element `?E` has modifier (i.e public, private, static, etc) `?S` |
| `arg(?C,?I)` | Callable, Type | Collable `?C` has an an argument of type `?I` |
| `returns(?C,?I)` | Callable, Type | Callable `?C` returns Type `?I` |
| `signature(?C,?I)` | Callable, String | Callable `?C` has signature `?I` |

Table 8: Binary Core Predicates

### A.3 Ternary Core Predicates

| Predicate Name | Argument Type | Description |
|---|---|---|
| `methodCall(?B,?M,?L)` | Block, Method, SrcLocation | Block ?B calls Method ?M at location ?L |
| `superCall(?C1,?C2,?L)` | Callable, Callable, SrcLocation | Callable ?C1 makes a "super" call to Callable ?C2 at location ?L |
| `constructorCall(?B,?C,?L)` | Block, Constructor, SrcLocation | Block ?B calls Constructor ?C at location ?L |
| `instanceOf(?B,?T,?L)` | Block, RefType, SrcLocation | Block ?B performs an instanceof test for type ?T at location ?L |
| `reads(?B,?F,?L)` | Block, Field, SrcLocation | Block ?B reads field ?F at location ?L |
| `writes(?B,?F,?L)` | Block, Field, SrcLocation | Block ?B writes to field ?F at location ?L |
| `param(?C,?T,?N)` | Callable, Type, Integer | Callable ?C has an argument of type ?T as its ?Nth argument |
| `tag(?E,?N,?V)` | Element, String, String | Element ?E has javadoc tag ?N with value ?V |

Table 9: Ternary Core Predicates

## A.4 Derived Predicates

| Predicate | Meaning |
|---|---|
| `type(?T)` | `?T` is a Type (a class, interface, or primitive) |
| `element(?E)` | `?E` is an element |
| `likeThis(?E1,?E2)` | Element `?E1` and Element `?E3` have the same name, but are not the same object |
| `strongLikeThis(?C1,?C2)` | Callable `?C1` and Callable `?C2` have the same signature |
| `child+(?E1,?E2)` | Element `?E2` has Element `?E1` as one of its ancestors |
| `package(?E,?P)` | Element `?E` is in Package `?P` |
| `constructor(?CL,?CO)` | Class `?CL` declares Constructor `?CO` |
| `method(?T,?M)` | Type `?T` declares Method `?M` |
| `subtype(?T1,?T2)` | Type `?T2` is a direct subtype of Type `?T1` |
| `subtype+(?T1,?T2)` | Type `?T2` is in `?T1`'s type hierarchy |
| `subtype*(?T1,?T2)` | Type `?T2` is in `?T1`'s type hierarchy (both can equal) |
| `field(?T,?F)` | Type `?T` declares Field `?F` |
| `inheritedField(?T,?F,?Inh)` | Type `?T` inherits Field `?F` from Type `?Inh` |
| `calls(?B,?C,?L)` | Block `?B` calls Callable `?C` at SourceLocation `?L` |
| `staticCall(?B,?C,?L)` | Block `?T` statically calls Callable `?C` at SourceLocation `?L` |
| `polyCalls(?B,?C,?L)` | Block `?B` calls polymorphic Callable `?C` at SourceLocation `?L` |
| `accesses(?B,?F,?L)` | Block `?B` accesses Field `?F` at SourceLocation `?L` |
| `inheritedMethod(?T,?C,?Sup)` | Type `?T` inherits Callable `?C` from Type `?Sup` |
| `overrides(?C1,?C2)` | Callable `?C1` overrides Callable `?C2` |
| `creator(?C1,?Ctor,?L)` | Class `?C1` is created by Block `?Ctor` at SourceLocation `?L` |

| | |
|---|---|
| `implements+(?C1,?I)` | Interface `?I` is implemented by Class `?C` or an ancestor of `?C` |
| `re_match(?RE,?S)` | String `?S` matches Regular Expression `?RE` |
| `re_name(?E,?RE)` | Element `?E` has a name that matches RegExp `?RE` |

Table 10: Derived Predicates

### A.4.1 Custom Predicates

| **Predicate** | **Meaning** |
|---|---|
| `rbd(?Elem,?Mrk)` | Element `?E` has a source code marker (Doclet) `?Mrk` |

Table 11: Custom Predicates

# B The JavaCC Grammar File

```
SKIP : {
  " " | "\t" | "\n" | "\r"
}

TOKEN : {
  < Varchar: ["a"-"z","A"-"Z","%","*"]
       (["a"-"z","A"-"Z","0"-"9"," ","%",",",".","."])* >
| < Rules: ["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z","0"-"9"
       ,",",",";","?","(",")","/",".","%","^","\"","_","*"]
       )* >
| < LDocBound: "{" >
| < RDocBound: "}" >
| < LRuleBound: "[" >
| < RRuleBound: "]" >
| < CommentWrap: "\"" >
| < CommentSeparator: ";" >
| < VarCommentSeparator: ":" >
}

List<DocumentationEntry> Input() : {
  List<DocumentationEntry> docs =
       new ArrayList<DocumentationEntry>();
  Token name,rules;
  Map<String,String> comments;
}
{
 (
  name=<Varchar> <LRuleBound> rules=<Rules> <RRuleBound>
       <LDocBound>comments = DocElemComments()<RDocBound>
  { docs.add(
     new DocumentationEntry(name.image,comments.get("*"),
       new Query(rules.image),
       comments
     )
    );
  }
 )*
 <EOF>
 { return docs; }
}

Map<String,String> DocElemComments() : {
  String[] comment;
  Map<String,String> comments =
       new HashMap<String,String>();
}
{
```

```
48  (
49   comment = DocElemComment()
50   { comments.put(comment[0],comment[1]); }
51  )+
52 { return comments; }
53 }
54
55 String[] DocElemComment() : {
56   Token Dvar,Dtxt;
57   String[] comment = new String[2];
58 }
59 {
60   Dvar=<Varchar> <VarCommentSeparator>
61     <CommentWrap> Dtxt=<Varchar> <CommentWrap>
62     <CommentSeparator>
63  {
64   comment[0] = Dvar.image;
65   comment[1] = Dtxt.image;
66   return comment;
67  }
68 }
```

Listing 9: JavaCC input-document to generate a parser for the documentation language designed in figure 11

# C The RuBaDoc Project-CD

This appendix describes the content and structure of the CD that is attached to this report. It contains the four main folders: **rubadoc**, **documentation**, **report** and **bibliography** that are briefly covered in this section.

**The RuBaDoc Plug-In** The final Eclipse extension is enclosed as both – as deployable binary plug-in (`bin` folder) as well as the browsable source code (`src` folder). By placing the `.jar` from the `bin` folder into the eclipse/plugins folder, the extension is ready to use. The package structure of the `src` was already explained in section 4.2.2. Additional required libraries can be found in the `lib` folder.

**The Documentation** Due to this thesis, the projects source code was documented in two different ways. The `javadoc` folder provides a conversant API documentation that covers interface level constructs in a context independent way. The hypertext document was created by using the javadoc tool that extracts special annotated explanations from source code.

Based on the work of this thesis, the `rbd` folder contains a document that covers the collaboration aspects of RuBaDoc which were not adequately documentable with a doclet approach. The `collaboration.rbd` file can be consulted by the proposed plug-in to capture the relationship between certain program constructs.

Together both documents provide an adequate insight to the RuBaDoc architecture.

**The Report** All TEXdocuments that are needed to compile this thesis can be found in the report folder. The bootstrap document is DA.tex that defines formatting assertions and includes all chapters. All depicted illustrations can be found in the `figures` folder. In addition, the *SVG* sources for all drawings are enclosed in the `figuresSVG` folder.

**Bibliography** The folder contains the pdf-files of the referenced papers (as far as possible). To clarify the relationship between pdf-files and references inside the report, the bibliography keys in the thesis are used as as filenames.

## C.1 Initializing The Plug-In

To set up the documentation environment, the RuBuDoc plug-in demands two parameters. First – but optional – is a documentation document written in the language designed in section 3.4. The designed prefix for RuBaDoc documents is `.rbd`. The plug-in reads in all files following this

convention that are in the Eclipse Workspace directory (usually: [UserDirectoryPath]/workspace).

In order to build up the internal fact base, the plug-in requires a Working Set that bounds the search domain. In Eclipse, Working Sets can be specified using `Window / Working Sets / Edit ...` In the Dialog, one need to create a new Working Set. In the wizard, choose `Java` as Working Set type. Following, a form ask for `name` and `content` of the Working Set. Enter **RBD** as name, and select the sources as preferred. It is recommend to exclude external library, since these requires perceptible time and memory resources. Typically, only the source packages are selected as a Working Set (see figure 22).

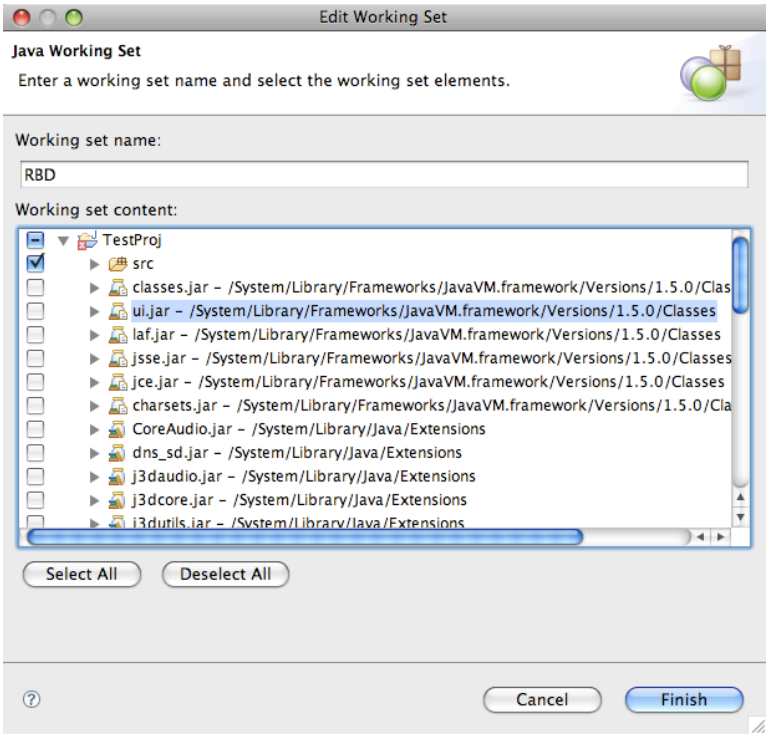


Figure 22: Specifying a Working Set for Documentation

## References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[AHE98] Amiram Yehudai Amnon H. Eden, Yoram Hirshfeld. LePUS - a declarative pattern specification language. Technical report, Department of Computer Science, Tel Aviv University, 1998.

[App00] Brad Appleton. Patterns and software: Essential concepts and terminology. *http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html*, 2000.

[Bar03] Aline Lucia Baroni. Design patterns formalization. Technical report, Ecole Nationale Superieure des Techniques Industrielles et des Mines de Nantes, 2003.

[Bar07] Andreas Bartho. Tutorial creation with deft. *Technische Universität Dresden - Department of Computer Science*, 2007.

[BC89] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[DE05] Jens Dietrich and Chris Elgar. A formal description of design patterns using owl. In *Australian Software Engineering Conference (ASEC)*, New Zealand, 2005. Massey University New Zealand.

[DJ04] Maja D'Hondt and Viviane Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 132–140, New York, NY, USA, 2004. ACM.

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, 1998.

[EMO04] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2004.

[Fer04] Len Feremans. Integrating JAsCo artifacts within the concern manipulation environment. Technical report, Vrije Universteit Brussel, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[Hed97] Görel Hedin. Language support for design patterns using attribute extension. *Lecture Notes in Computer Science*, 1357:137+, 1997.

[HOA+06] Elnar Hajiyev, Neil Ongkingco, Pavel Avgustinov, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Datalog as a pointcut language in aspect-oriented programming. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 667–668, New York, NY, USA, 2006. ACM.

[HVdMdV05] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. *CodeQuest*: Querying source code with datalog. In *Object-Oriented Programming Languages and Systems (OOPSLA) Companion*. ACM Press, 2005.

[KC02] A. Kacofegitis and N. Churcher. Theme-based literate programming. In *Theme-based literate programming*, 2002.

[KHR07] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 6, New York, NY, USA, 2007. ACM.

[KM01] Mira Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 354, Washington, DC, USA, 2001. IEEE Computer Society.

[Knu84] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

[KP96] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering*, pages 208–, 1996.

[LSF03] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practise. *IEEE Software*, pages 35–39, 2003.

[Mar01] Marius Marin. Formalizing typical crosscutting concerns. Technical report, Software Engineering Research Group, Delft University of Technology, 2001.

[MK06] Kim Mens and Andy Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 239–248, Washington, DC, USA, 2006. IEEE Computer Society.

[MM83] C McClure and J Martin. *Software Maintenance, The Problem and Its Solutions*. Prentice-Hall Inc., New Jersey 07632, 1983.

[MMvD07] Marius Marin, Leon Moonen, and Arie van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 758–761, Washington, DC, USA, 2007. IEEE Computer Society.

[NAC+00] Kurt Nømark, Max Andersen, Claus Christensen, Vathanan Kumar, Søren Staun-Pedersen, and Kristian Sørensen. Elucidative programming in java. In *IPCC/SIGDOC '00: Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation*, pages 483–495, Piscataway, NJ, USA, 2000. IEEE Educational Activities Department.

[Nør00] Kurt Nørmark. Requirements for an elucidative programming environment. *International Workshop on Program Comprehension*, 2000.

[NSW+02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, New York, NY, USA, 2002. ACM Press.

[PHW04] P.Tarr, H.Ossher, and W.Harrison. Pervasive query support in the concern manipulation environment. Technical report, IBM Research Report RC23343 (W0409-135), 2004.

[Rie98] Dirk Riehle. Bureaucracy. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 163–185. Addison Wesley, 1998.

[Rie00] Dirk Riehle. *Framework Design - A Role Modeling Appraoch.* PhD thesis, Swiss Federal Institute Of Technology Zurich, 2000.

[RWL96] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with objects: the Ooram software engineering method.* Manning Publications, Greenwich, CT, 1996.

[Smi02] David Smith, Jason McC.; Stotts. Elemental design patterns - a link between architecture and object semantics. Technical report, University of North Carolina at Chapel Hill, 2002.

[SRK07] Daniel Speicher, Tobias Rho, and Günter Kniesel. JTransformer - eine logikbasierte infrastruktur zur codeanalyse. *Universität Bonn, Institut für Informatik III*, 2007.

[Tan03] Kajal Tansalarak, Naiyana; Claypool. Mining for sample code. Technical report, Department of Computer Science, University of Massachusetss - Lowell USA, 2003.

[TH06] Nikolaos Tsantalis and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909, 2006. Member-Alexander Chatzigeorgiou and Member-George Stephanides.

[TL03] Toufik Taibi and David Ngo Chek Ling. Formal specification of design patterns - a balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.

[Usc96] Michael Uschold, Mike; Grüninger. Ontologies: principles, methods, and applications. *Knowledge Engineering Review*, 11(2):93–155, 1996.

[Ves99] Henrik Morck Mogensen; Kristian Ravn Tylvad;Thomas Vestam. DocSewer - a documentation tool (prethesis). Technical report, Institute of Electronic Systems - Aalborg University - Department of Computer Science, 1999.

[Ves04] Thomas Vestdam. *Tools, Patterns, and Experiments.* PhD thesis, Aalborg University, Department of Computer Science, 2004.

[Vol01] Kris de Volder. JQuery: A generic code browser with a declarative configuration language. Technical report, University of British Columbia, Vancouver BC, Canada, 2001.

[Wen03] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. *ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland*, 2003.

[Wer07] Jochen Wertenauer. codation - verbindung von code mit zusatzinformation. Diploma thesis, Institut für Softwaretechnologie, Universität Stuttgart, 01 2007.

## Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, January 31, 2008