Diplomarbeit

coTag:
Code tagging and similarity-based retrieval for eclipse using myCBR

Daniel Bahls

Oktober 2008

Betreuung: Prof. Dr. Andreas Dengel
Dr. Thomas Roth-Berghofer

Arbeitsgruppe Wissensbasierte Systeme
Prof. Dr. Andreas Dengel

Technische Universität Kaiserslautern
Fachbereich Informatik

Daniel Bahls
Gottlieb-Daimler-Str. 69
67663 Kaiserslautern

Kaiserslautern, den October 20, 2008

Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

(Daniel Bahls)

Contents

Abstract

During their professional life developers re-encounter many tasks and problems that have already been solved in the past. Mostly, when trying to reuse former solutions, just a few helping lines of code are demanded to be copied to the currently edited document. But finding it takes time or is not even possible. The code tagging plug-in *coTag*allows annotating code snippets in the open source IDE *eclipse*. Using the similarity-based search engine of the open-source tool *myCBR*, the user can search not only for exactly the same tags as offered by other code tagging extensions, but also for similar tags and, thus, for similar code snippets. In addition, *coTag* tries to retrieve suitable snippets for the current situation by taking into account the developer's work context.

Following a personalised approach, users can modify relations between tags in order to adapt *coTag*'s retrieval results to suit personal needs. Action explanations give details about the outcome of a query and reveal inappropriate tag relations, which are by default determined via trigram matching to compensate typing and spelling errors.

Being part of a community, developers share common goals, common interests and work on joint projects. As to support the exchange of coding knowledge, *coTag* allows for exchanging and collaborative tagging of code snippets. Since communities share a similar perspective on things, modified tag relations can be shared for collaborative similarity modelling. Moreover, author information is provided to explain the provenance of code snippets, tags or similarity customisations.

In this work, we analyse the nature of tags applied to code resources in order to find out what information is relevant for developers. As we view tagged snippets as solved cases, where tags and context information describe programming tasks and the code fragment itself describes the respective solution, *coTag* is based on a Case-based Reasoning model. Finally, we analysed user acceptance and evaluated whether this tool helps developers achieving shorter development times and higher quality of code.

Part I

Introduction

Chapter 1

A programmer's dilemma

During their professional life developers work in many projects, use many Application Programming Interfaces (APIs) and programming languages. Thereby, they re-encounter many tasks and problems that have already been solved in the past. But especially in the domain of programming, the accurate way of using a specific API, coding in a certain language or a certain algorithm can be hard to remember. Mostly, just a few helping lines of code are demanded to be copied to the currently edited document. One often remembers the fact that a similar situation has already been solved, but the respective piece of code cannot be found or is not even available on the file system anymore.

1.1 Problem description

There are many portals with tutorials, introductions, and best practices regarding APIs and languages out there on the Web, which are continuously used to solve programming tasks. Even though the quality of advice is good, they are lacking personalised retrieval functionality. As to find good coding examples under the high amount of resources, queries must be specified precisely—not with one's own words but common public terminology. In addition, the retrieval algorithm does not take into account the questioner's working context, as for example the libraries and languages used in daily work. It can take quite a while to retrieve a good example, that holds a solution to the given problem.

Personal coding experience is a valuable resource, which can hardly be compared with public know-how. It can be very specific and may have been made in a very special working context. If you think of projects, that are not available to the public, or that do not attract enough attention to be explained thoroughly, an intelligent system providing access to the personal coding experience base could fill that gap. Furthermore, the context of creation and logical structure of a code fragment, which has been elaborated by oneself, is already known. This personal connection may help saving time and energy for the reuse of the code.

Becoming acquainted with new programming languages, APIs and other tech-

nologies takes a great part of a programmer's daily work. If the own experience does not contain the required knowledge, a developer has to use other resources—Web tutorials, books and further documentation—to acquire missing information. This in general requires a lot of time and money. But there is also the possibility to exchange knowledge with other developers. Collaboratively working on a project, in a particular field or for the same company, developers use a shared set of modules and have a common view on technologies. Asking a colleague helps in many cases. Thus, coding experience of other group members may be used to expede the process of learning the essential. Moreover, the social exchange of know-how helps cultivating the frequent use of best practices, which leads to a high quality of code.

At present, the usual way to access formerly created pieces of code is based on the exploration of the personal workspace or file system. The respective file can be found by either having a clearly organised folder structure and a long memory or by performing a full text search. While the former alternative is effortful and limited to handle only a small amount of resources, the latter one demands formulating a query by keywords or a regular expression and requires much computation time to complete. Besides, the severe shortcoming of this approach is the fact that a workspace does not represent the personal experience base. Code changes continuously, as it is being developed further. Some expressions, passages or even files or modules sometimes vanish entirely, as they lose their purpose. Interfaces, protocols and semantics of a component can evolve and may no more contain the original piece of code the user is looking for. Additionally, the exchange of coding experience is complicated and happens for most people only by working on the same project.

All in all, one can say that personal coding experience is quite underappreciated at the moment and should gain more support. Having pointed out the reasons to take action, we developed a personal knowledge management system in this thesis to overcome this lack. Eventually, we hope to make a contribution for developers to produce higher code quality and reduce development times.

1.2 Prerequisites

The reader should be familiar with the concept of an IDE. In particular, it is recommended to be familiar with the IDE *eclipse*. Although it is not necessary to be conversant with the programming language Java 5[1] which was used to implement *coTag*, experience in the field of computer programming is very helpful in order to estimate the value of this work. Although the technique of tagging does not involve difficult ideas, being familiar with systems that allow for collaborative tagging is considered helpful. Since the tool retrieves code snippets with the help of similarity measures, being experienced in the field of Case-based Reasoning is advantageous. A comprehensive introduction can be found at [Lenz *et al.*, 1998].

[1] `http://java.sun.com`

However, we give an introduction to all used techniques which should suffice to comprehend this work.

1.3 The structure of the thesis

The thesis is divided into four parts. Part I begins with a description of the problem we address with this thesis. Chapter 1 illustrates the shortage of a tool that helps accessing personal coding experience and community knowledge. We develop the vision of the *coTag* system in Chapter 2, where the formulated problems from Section 1.1 are transferred to system requirements. Subsequent to a discussion about related work in this field, we consider the fundamental design decisions to our approach.

Part II details the concept behind our approach. Since we use tags to describe code resources, Chapter 3 surveys the kinds of tags that are generally used for code resources. This helps us elaborate and evaluate an approach for automatic tag generation and gives us ideas for the design of the CBR model in Chapter 4. After a concise introduction to general terms and concepts of Case-based Reasoning, we explain the used attributes to describe a code resource and detail the retrieval technique used in *coTag*. The chapter closes with a technique to allow users for applying personal preferences to the retrieval technique. Chapter 5 focusses on communities and how far knowledge exchange is supported.

Part III of this thesis commences with a walkthrough of the main functionalities of *coTag* from an end-user perspective in Chapter 6. It explains step by step how users can acquire code snippets, pose queries and how snippets are presented. As the scrutinisation of a retrieval result is quite complex, a separate chapter is dedicated to this use case. At the beginning, it describes how action explanations give details about the retrieval process. The chapter closes by introducing an interface to equip the system with a personal understanding of tag relations. To understand the system architecture behind, Chapter 8 reveals the components of the *coTag* IDE plugin, the *coTag* server and their communication. It describes *eclipse*, *myCBR* and major internal modules of *coTag*.

Part IV concludes the value of this work. The evaluation of *coTag* comprises an analysis of a six-week test run within a researchers work group and an inquiry based on a questionnaire, which is explained in Chapter 9. Finally, this work closes with a summary and outlook.

Chapter 2

Managing personal coding experience

Delivering the *right information* at the *right place* at *the right time* is the main goal of knowledge management. If there was a tool that helps formulating the problem quickly and easily, finds the desired piece of code reliably, and presents it after a few seconds right inside the developers working place, i.e., in his or her Integrated Development Environment (IDE), then the requirements of good knowledge management are met. This thesis addresses the task of building such a system.

The next section develops the vision of the *coTag* system. It lays open some essential issues directly following from the nature of the problem we want to solve. Moreover, the formulation of further goals leads to additional requirements. Other ideas and approaches of similar kind are compared thereafter in Section 2.2. In the last section, we determine the outlines of our approach, where the fundamental design decisions are discussed. It also gives a concise survey of the chosen techniques and solutions.

2.1 The vision

In our opinion, a good knowledge management system speaks the language of the user and supports his understanding of things. The user must be allowed to formulate queries with his own words, whereas the results must fit as good as possible to the user's estimation of relevance. This raises the need to learn his personal information model [Sauermann *et al.*, 2008].

Besides the personalisation issues, we demand further that the retrieval algorithm is able to take the user's work context into account. This issue needs some clarification, because the notion of context in general is very fuzzy. Interesting could be the currently edited code as to compare it with code fragments of the experience base. But we must have in mind, that the user did not solve his programming task, yet. So there may be no code at all which will be part of the

future solution. However, at the moment of query submission, we probably have the information on which kind of component the questioner is currently working. Whether the user is currently working on a GUI component, a data base connection, an IO module or others can be a valuable information for the retrieval engine to improve its quality.

In addition, it should compensate spelling or typing errors.

Furthermore, the system must be able to explain its behaviour. As the user does not understand the outcome of a query, a short an precise report must be available enabling the user to comprehend how the result came up. Being informed about the background of the outcome, the user may disagree with the system's way of query processing. But rather than settling for the things as they are, the user may want to tell the system how things can be improved. The moment after an explanation gives a good opportunity to configure the settings involved in the result.

The entire coding experience should be covered by this tool. This means in particular, that it should provide access to any formerly created source code and be applicable for as many kinds of project as possible. The coding experience should also be made available to other developers.

2.2 Related work

Many approaches to support the reuse of software artefacts have been developed over the last years. Each of them varies slightly regarding the motivation and purpose, and has its very own implementation. We want to introduce some related ideas in the following and make clear how they differ from ours.

An interesting concept has been followed by [Gomes & Leitão, 2006] who built a suggestion system for the domain of software design. It applies a company's own development experience as a case base of a CBR system. Therefore, it has been integrated into the commercial UML tool Enterprise Architect[1] and assists every software designer of the belonging company in generating new UML diagrams. Although, it covers a bunch of questions respective software design, it cannot answer language specific ones, since it doesn't comprise coding details. Also, the practical usage of a foreign API cannot be explained, because its UML diagrams are not available ad hoc.

A similar approach was followed with CIAO-SI [Nkambou, 2004], which is based on Case-based Reasoning techniques. At the beginning of a new development project CIAO-SI suggests software artefacts (models, documents, source code) that have been used in past projects. Therefore, the developer must formulate a query that consists of the respective application domain and additional software characteristics. With the use of CASE[2] tools, the resulting artefacts can be adapted to the requirements of the new project. CIAO-SI assists developers in the complete application design phase. It considers the outlines of the planned project in a

[1] `http://www.sparxsystems.de/`

[2] Computer Aided Software Engineering

macroscopic level of detail. In contrast to this, our intention is to show small code snippets or passages of source code documents in a much smaller context and to provide light-weight assistance for individual developers in day-to-day use.

Quite complementary to the above is the following approach [Storey *et al.*, 2006]. With TagSEA[3] Storey et al. aim at a better documentation and navigation of source code by enriching bookmarks with meta-data such as provenance and social tags.[4] TagSEA allows sharing of these among project teams. But their goal was to provide a more sophisticated use of bookmarks within source code, not to answer questions about the usage of API's or programming languages. Hence, it provides only a quite simple user interface to find the desired bookmarks based on exact match filters rather than on information retrieval techniques. Further, it restricts browsing to workspace files only, which means that no foreign code can be taken to answer a question.

The IDE *eclipse* already comes with a snippet view that allows storing pieces of code as a template with a name and description under an arbitrary category. The snippet organisation equals a folder structure known from file systems, i.e., the snippet can only belong to one and only one category. The plug-in was probably not meant for large amounts of snippets, hence there is also no retrieval support. This means in particular that categories must be well arranged in order to still get the full picture. Finding a certain snippet means finding the right path in this structure. This binds the user to a single pattern of thought and makes categorising a hard and effortful issue. As a consequence, the user is held back from archiving all interesting snippets of his workspace, which in turn results in an even sparser usage of the functionality at all.

Another very powerful tool to search foreign pieces of code is provided by google[5]. Queries can be formulated by regular expressions and other google specific patterns. A very fast search engine returns syntactically suitable results from a huge collection of source code. Although the knowledge base is very large, it does not contain a user's personal experience knowledge. Hence, any queries concerning private modules cannot be covered. As mentioned earlier, public portals and search engines force also the subordination to common public terminology and understanding of relevance. It just does not support the user's personal information model.

A more personalised approach is followed by DZone snippets[6], which allow developers to upload code snippets to their website. The publisher can tag the snippets with keywords, which can be used to retrieve them again. Being logged in as a registered user, one can browse through the own snippets. This tag-based approach allows users to organise their personal coding knowledge and supports the user's own vocabulary. But this personal snippet repository is accessable only via the Web. A browser must be used to insert and access the snippets. Hence,

[3] `http://tagsea.sourceforge.net`

[4] Their bookmarks are called *waypoints* following the metaphor used in navigation systems.

[5] `http://www.google.de/codesearch`

[6] `http://snippets.dzone.com`

meta information like programming language, document type, operating system or APIs in use must be entered as a tag manually, because the tool is not integrated in an IDE. Furthermore, it cannot handle private code snippets that are not meant for the public. In addition, the tag-based retrieval is not capable of handling spelling errors.

2.3 Outline of our approach

As we mentioned earlier, the workspace or file system does not represent the experience base due to the evolution of code and the removal of unused projects. Consequently, we have to provide not only an indexing strategy, but also a way to back up code. Generally spoken, providing access to any previously created source file is a difficult issue. Versioning systems like subversion or Concurrent Versions System could solve the problem of making the whole coding experience base available. But this approach requires additional hard disk space, needs maintenance, and the exact point of time for code submission is not clearly defined. Furthermore, retrieval functionality would also be problematic, as it had to include all versions of all files.

So we switched to the idea, that the user must somehow denote the pieces of code he deems valuable for reuse. As a consequence, only selected fragments of code must be stored, which reduces the size of the required archive enormously. In addition, the search space of the retrieval engine is limited to only those code pieces, that itself have been denoted as valuable for reuse. So, the quality of the results cannot be too bad. The disadvantage is, that the user must a-priori realise, that he may want to reuse a certain code fragment and make an effort to mark it, although he is most likely busy on something else at this moment. Nevertheless, this approach seems to be a good compromise between making available the entire experience, delivering good retrieval results efficiently and feasibility at all. A marked piece of code will from now on be referred to as *code snippet*.

After a code passage was marked as a code snippet, it could be retrieved again with the help of common Information Retrieval (IR) algorithms based on its content. But we have stated that one of our goals was the support of the user's personal information model. Especially letting him formulate queries with his own words—and yet delivering good results—requires additional knowledge for finding out the relevance between a query and a snippet. Hence, the piece of code must rather be annotated than denoted. With the evolution of the Web towards Web 2.0, one technique for information description called *tagging* has become very popular. Even though, tagging often lacks semantical foundations, its acceptance in social domains such as bookmark sharing[7] or photo exchange[8] among many others proves its practical usefulness. Having in mind the simple search and find interfaces and intuitive navigation among digital resources, we strive for the goal to bring these

[7] `http://del.icio.us`
[8] `http://flickr.com`

capabilities also to the domain of programming.

Given these preconditions, the whole use case becomes clear: A code snippet can be interpreted as a solution to a problem that was solved in the past. When a good solution to a particular problem was elaborated, it may be annotated and added to the experience base. Eventually, if the user encounters a problem that has once already occurred in a similar way, a query can be posed to retrieve the solutions to similar problems. This can be seen as a typical Case-based Reasoning (CBR) scenario, whereas a solution to a problem is drawn directly from the solution of a similar problem, which has already been solved.

A remaining issue is the question of retrieval. In CBR, similarity measures are used to predict the usefulness of a case for a given query. As to continue the discussion about the utility of regular IR techniques for our purposes, it remains to say that we have also to consider the work context as another aspect in addition to the tags. Therefore, the CBR point of view seems to fit better to the whole problem. But we may regard IR methods for the design of the tag similarity measure.

The delivery of the *right information* at the *right time* is a quality that can only be measured afterwards when the system was built. It must be designed carefully in order to pursue these goals. But a more direct control respecting the *right place* is given beforehand. We can say that the questioner will most probably be sitting in front of his work station and busy with the implementation of a particular component. Furthermore, since most developers use an IDE to solve their programming tasks, we chose an IDE for the right place, which was eclipse in our case. *eclipse* is a widely used and powerful open-source plug-in framework. It supports a great variety of programming languages and file types by providing convenient tools and editors, that bring syntax colouring and content assistance. Implementing our knowledge management system as a plug-in for *eclipse* has the advantage, that a lot of information can be gained about what the user is currently working on, the results of a query can be presented in a suitable manner, and the tool is always available during programming.

Part II

Theoretical considerations

Chapter 3

Code tagging

Web 2.0 software shows that there are many people out there, who love to tag photos, bookmarks and videos as long as tagging is easy and provides benefits. To facilitate tagging of code, we want to provide a mechanism to propose suitable tags, for which we must first understand what information is relevant for developers. Hence, we analyse the kinds of tags used to characterise code resources in the following.

In Section 3.1 we transfer known categories of tags to our domain top down. An empirical study about the nature of tags applied to code resources is presented subsequently. Eventually, we establish and evaluate an approach to generate useful tags automatically.

3.1 Categories of tags

[Golder & Huberman, 2006] introduced seven categories for tags given to general resources. By transferring these categories to the domain of code tagging, we want to get ideas what information could be relevant for users. In the following, we list the single categories and reflect on them with respect to our application domain:

1. **"Identifying what (or who) it is about"**
   A tagged piece of code describes a certain algorithm, which solves a tricky task, which is frequently used or which gives information about how a certain API can be accessed.

2. **"Identifying what it is"**
   The resource to describe with tags is in our case a section of a text document, which is of a particular programming language, document type and part of a certain project. Implicitly, it is also more or less designed for a certain platform[1]. These properties apply also for the snippet.

[1] e.g. Microsoft Windows, linux, mobile platforms such as symbian or similar

3. **"Identifying who owns it"**
   In this case, we regard the tagger as the owner of the resource. He may not be the author of the whole document, but he is the one who deemed the code snippet valuable for reuse and thus is the creator of the extracted snippet.

4. **"Refining categories"**
   The meaning behind a refining category tag can be very different and is hard to analyse in advance. For example, it could refer to a particular concept of the user's subject area which we do not know.

5. **"Identifying qualities or characteristics"**
   These tags represent a certain classification done by the tagger. Tags of this kind could be for example *configuration*, *web*, *database* or *prototype*.

6. **"Self reference"**
   This kind of tags is used to distinguish personal resources from those of other users. In our case, self reference can have two different meanings. It can give information about who created the snippet which is in our understanding the tagger. Or it can also mean that the resources with this tag have something to do with the personal workspace of the tagger. As we emphasise on the distinction between personal and public snippets we want to provide additional options to search for snippets tagged by the user, or to search for snippets whose original document is part of the user's workspace.

7. **"Task organizing"**
   The purpose of such tags is to group resources respecting a particular task. Transferred to the domain of code tagging, a certain tag could be attached to all pieces of code that deal with a special programming task, like for example *serverSetup*.

The kinds of information we identified by systematically analysing the different categories of tags will be regarded again in Chapter 4 where the CBR model for the application is designed.

3.2 Tag nature study for code resources

In contrast to Section 3.1, where we looked at some properties of the tags top down, we now want to get some ideas bottom up by analysing the kinds of tags that can be observed for code snippets. On a web site called *DZone snippets*[2] developers can upload pieces of code and describe them using free text and tags. The style of resource publishing resembles much the way of flickr[3] as a tagged resource is published by the tagger himself and only the publisher is allowed to attach tags to it. In September 2008, their repository comprised $4,090$ pieces of code with an

[2] `http://snippets.dzone.com`
[3] `http://www.flickr.com`

average of 4.4 tags attached. As all code snippets have been annotated completely manually without any kind of tag proposition, this collection is a valuable resource to examine what information is considered of importance by code taggers.

A first impression can be gained by looking at the tag cloud presented at their home page (Figure 3.1). Apparently, a high ratio of the tags is dedicated to the used

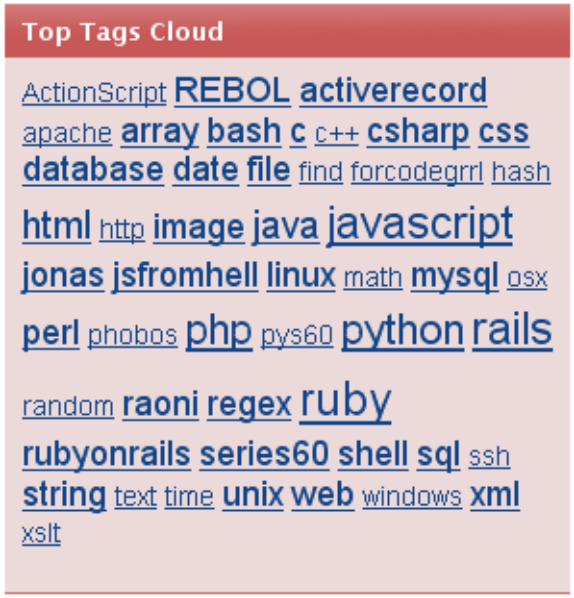


Figure 3.1: Most frequent tags at *DZone snippets*

programming language, the document type and operating system. But although these are the most frequently (re-)used tags, the majority of tags does not belong to the mentioned categories as illustrated in the Figure 3.2. This is due to the fact that the number of programming languages, document types and operating systems is strongly limited whereas the information about it is very significant for a published piece of code. Furthermore, a high ratio of the tags can be found in the snippet itself and is part of the code. Note that all document types that imply the usage of a certain programming language are assigned to the category *Programming language*. Document types such as *XML* or *CSV* are assigned to the category *Other document types*.

The most frequent tags of the remaining fraction are illustrated in Table 3.1. Because this fraction is composed of a great variety of tool names, interfaces, libraries and fuzzy concepts, it was not further subclassified. Interesting to see was also that only a few tags are used for self reference, whereas some are used to address other users (e.g. the tag *forcodegrrl*). It is further to mention that most of the tags were used only once. In average, the $13,974$ different tags were reused about 1.3 times. The nature of the reused tags differs slightly from the nature of the single tags, as many of the single tags were created by typing errors.

Table 3.1: Unclassified tags ordered by number of occurrence

(65) regex	(18) programming	(13) subversion
(61) web	(18) raoni	(13) testing
(59) sql	(18) search	(12) Treetop
(54) forcodegrrl	(17) dom	(12) acm
(47) phobos	(17) jonas	(12) bookmarklet
(44) database	(17) script	(12) http
(43) series60	(16) Typo3	(12) icpc
(39) mysql	(16) servers	(12) list
(32) apache	(15) TypoScript	(12) simple
(30) array	(15) command	(12) test
(29) string	(15) jpg	(12) valladolid
(28) algorithm	(15) url	(11) email
(28) math	(14) example	(11) model
(24) images	(14) file	(11) parse
(21) activerecord	(14) helper	(11) sysadmin
(21) convert	(13) capistrano	(11) validation
(21) random	(13) code	(10) SQLServer
(20) files	(13) firefox	(10) audio
(20) rake	(13) pattern	(10) browser
(19) jsfromhell	(13) prototype	...

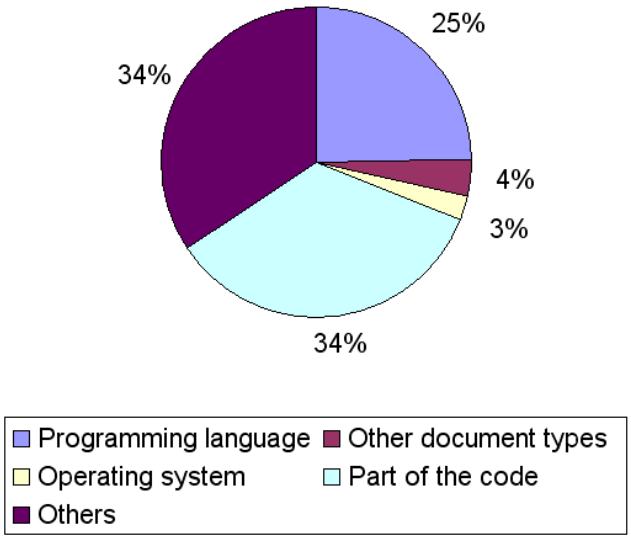


Figure 3.2: Frequency distribution of tags at *DZone snippets*

3.3 Tag proposition

In the previous sections, we analysed the kinds of tags in the domain of code tagging. Knowing approximately what tags could be relevant for a piece of code, we will now try to generate them automatically.

The information about the used programming language, document type or operating system can be acquired easily, as we will integrate our application in an IDE. This information will not be treated as a tag but as a separate attribute of the code snippet as explained later in Chapter 4. In the following, we focus on the generation of the content-based tags first and discuss the generation of tags, that are not part of the code subsequently.

For each snippet at *DZone snippets*, we tried to generate those of its attached tags that are also contained in the code. We measured then how many of the generated tags were in fact used as one of the snippet's original tags. First, tokens were generated by separating the code fragment at every punctuation character[4]. In average, 1.2% of these tokens occurred in the tags. The control statements being removed from the proposals let this ratio increase to 1.8%. In return, the ratio of content based tags being contained in the propositions was 79% and 78%. The situation is illustrated in Figure 3.3.

The examination yields that automatic generation of content-based tags is a difficult issue. Although higher rates are probably achievable with the help of additional background knowledge, we do not believe that this approach leads to an acceptable solution for automatic tagging. It seems that taggers make a certain se-

[4]Punctuation characters: !"#$%&'()*+,-./:;<=>?@[\]^ _`{|}~ and whitespaces

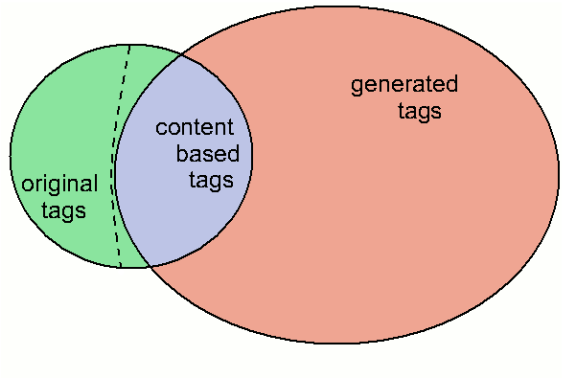


Figure 3.3: The left circle represents the actual tags of the snippet, of which a great fraction is also part of the code. The right ellipse represents the generated tags.

lection from a set of suitable tags, to emphasise only the relevant concepts. Hence, we must recognise that tags represent a very special kind of knowledge, that cannot be generated by automatic procedures at present.

Nevertheless, we could try to use our approach for text completion proposals. One big problem is that too many proposals are generated from the code, which are not used as tags. If the first letters of a tag are given, the chance to propose a suitable tag could be higher. The idea is to show an unobtrusive list of word completions when the user started typing a tag. We examined this idea again with the help of the *DZone snippets*. For each tag that appeared in the proposals, we counted the number of possible completions after each character. Figure 3.4 displays the average numbers of all snippets. In average, more than 50 proposals were given for a code fragment. After the first character is typed, the number of suitable proposals is reduced to five in average. With the next character this number is reduced further.

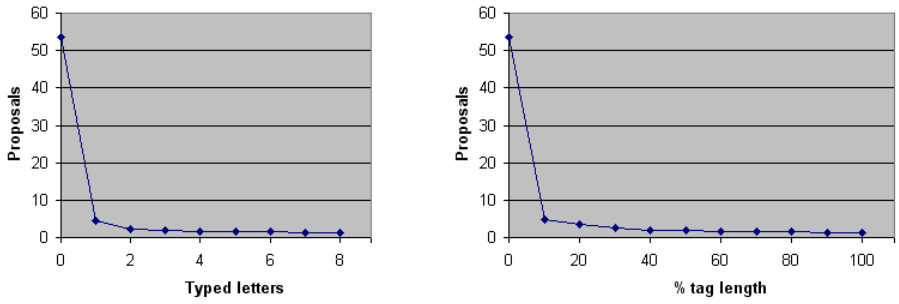


Figure 3.4: The left diagram shows the average number of proposals per number of typed letters

The tags that are not part of the code originate from the taggers mind who is able to classify the code fragment on a more abstract level having the overview of

its purpose and meaning. It seems difficult to determine such tags in a computational way. But with the help of the tag nature study in Section 3.2, we learned that there are frequently reused tags. This gave use the idea to include already used tags in the proposals. Furthermore, since many of the reused tags refer to known concepts in the domain of programming (e.g. *regex* or *database*), we want to include also words of public terminology. As it seems not helpful either to use all words of a dictionary for tag proposition, we decided to focus on community terminology only, which also supports our bottom up approach of personalisation.

Chapter 4

Case-based Reasoning

The system we want to build can be understood as a CBR application. When Eva, our example user, is looking for a piece of code she developed in the past, she must describe her current programming task with some keywords. Added some context information, the query for the system is created. This *problem description* is then used to search the experience base for code snippets that were used to solve a similar problem. If the returned snippets are suitable, Eva can copy and adapt them to build the solution for her task. Thus, tagged snippets can be seen as solved cases, whereas the tags and context information describe the programming task and the code fragment itself describes the respective solution.

As we want to take account of the work context during retrieval, we introduce a definition of it in Section 4.1. Fundamental terms and concepts of CBR needed to describe our model are explained in Section 4.2. Having everything we need to design the model, we define the attributes we use to describe a code snippet (Section 4.3) and detail the measures to determine the similarity between two problem descriptions subsequently (Section 4.4). The last section describes how the retrieval technique can be personalised to meet the user's understanding of similarity.

4.1 What do we mean by work context?

Context plays an important role in the field of computer science. But it is a very abstract term and must be defined precisely for every field of application. A generic definition which is frequently cited and also suits to our understanding of context is given by [Dey & Abowd, 2000]:

> *"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*

As we want to find an abstract description about the code which can be regarded in retrieval, the only entity we want to characterise is the piece of code. The *situation* of a code fragment can be understood as its *environmental embedding* and could be described by an enumeration of the modules it is working with. Thus, our definition of work context is the following:

> *A user's work context is defined by the set of item classes in use.*

Note that a work context does not regard the items themselves but only their classes. For example, the work context of a person who is picking apples from a tree could be described like this:

`apple` `ladder` `tree` `basket`

Figure 4.1: An illustration of a person's work context who is picking apples from a tree

The apples are represented qualitatively by a single class entry. The quantitative information could be used to define weights as a kind of relevance indication. But staying with our example, this raises the following question: *Does it make a difference for the work context, whether 20, 100 or 300 apples have been picked?* As finding an answer is problematic, we decided to leave it to a qualitative description of used item classes. As to illustrate how a work context looks like in the domain of programming, a few examples are given in the following.

Example 1: Peter is working on his ANT[1] file. He defines a build task by specifying the folders to be copied, the sources to be compiled and the temporary files to be deleted again. The context items identify the used ANT tasks:

`mkdir` `destdir` `copy` `delete`

`dir` `srcdir` `fileset`

`javac` `classpath` `todir`

[1] Another Neat Tool, `http://ant.apache.org`

Example 2: Eva is developing a GUI widget that allows dropping of dragged file objects to append it in a list. She therefore uses the classes of the JDK's Swing framework[2]. Her work context is the following:

```
java.util.List
javax.swing.JPanel
java.awt.dnd.DropTargetListener
java.awt.dnd.DropTargetEvent
java.awt.dnd.DropTarget
java.awt.datatransfer.DataFlavor
java.awt.datatransfer.UnsupportedFlavorException
```

Determining the work context of a developer depends strongly on the kind of document being edited. In Eva's case, the context items are Java classes, while the context items in Peter's case are ANT tasks and their properties. Thus, *coTag* needs a variety of *context extractors* to determine a developer's work context as to provide contextual information for each situation. In case a context extractor is not given for a particular document type, retrieval is carried out by tags only. More details on calculation of similarity are given in Section 4.4.

4.2 Introduction to Case-based Reasoning

Case-based Reasoning in general is a quite comprehensible kind of reasoning. As the name suggests, the reasoning is based on a collection of solved cases, which are mainly composed of a problem that has occurred in the past and its solutions. A case may further contain additional information like, for example, context or provenance. The reasoning process can be explained with the aid of the CBR cycle introduced by [Aamodt & Plaza, 1994] and illustrated in Figure 4.2.

At the beginning, the problem to solve must be formalised by a problem description. Basically, this represents the query posed at the system for which a solution is demanded. From another perspective, this problem description is part of a new case for which the solution part must yet be elaborated. Based on the assumption that similar problems have similar solutions, the CBR system compares the cases of its case base with the query respecting their problem similarity. As the retrieval of the most similar cases has finished, their solution parts are reused to compose a solution for the query, for which additional adaption knowledge is required. The generated solution is then revised and possibly adapted in collaboration with the questioner. Eventually, the found solution can be applied to the actual problem and the CBR system enhances its case knowledge by retaining the newly solved case.

[2]Java Development Kit, `http://java.sun.com/javase/6/docs/`

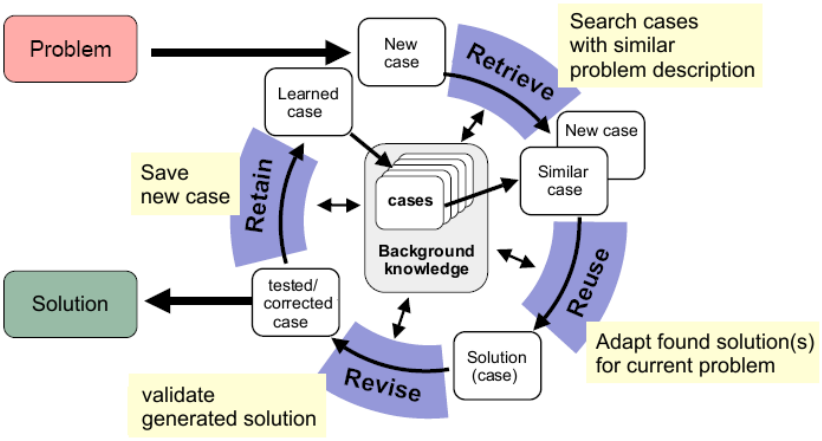


Figure 4.2: CBR cycle [Aamodt & Plaza, 1994]

The results of a retrieval are determined in two steps. First, all *filters* defined in the query are applied to the search space as to exclude inappropriate cases in advance. Secondly, the similarity is calculated for each remaining case as a kind of relevance estimation, which is used to establish a ranking. A filter specifies an allowed value (or a set of allowed values) for a certain attribute. Applying it to a set of cases yields a subset which contains exactly those cases whose attribute value corresponds to (one of) the specified filter value(s). Filters can be applied independently for each attribute. The application of more than one filter to a set of cases yields the intersection of all single subsets.

Moreover, the knowledge of a CBR system can be divided into four knowledge containers [Richter, 1995]. The *vocabulary* defines the language in which problems, solutions and all other information of a case can be described and provides the basis for the other containers. As we want to use structural CBR[3] with a flat hierarchy, the vocabulary is composed of a set of attributes, each specifying a particular set of allowed values. The *case base* represents the collection of solved cases. With our kind of vocabulary, a case base can be described with a table where each column represents one of the attributes and each row describes a case. Thus, a cell entry represents the value of the column's attribute for the case defined by the row. The *similarity measures* define metrics to determine the similarity between two problem descriptions. As in structural CBR problems are described using a set of attributes, it is convenient to apply the Local-Global Principle [Burkhard & Richter, 2001]. Local similarity measures are designed to determine the degree of similarity with regard to one single attribute. To aggregate all resulting local similarity values to a total one, a global similarity measure is used which technically could be a weighted sum for example. The *adaption knowledge* provides the in-

[3]There are also text, dialog and ontology based CBR systems (see [Lenz *et al.*, 1998]).

formation to transfer the solution(s) of the retrieved cases to the new problem. It comprises adaption rules and operators to assemble a solution for the query in an automatic or conversational way.

Considering the goals of our application, some parts of the cycle are not required. Adapting a retrieved code snippet automatically to solve the current programming task of the developer is hardly feasible, because programming tasks can be very complex and impossible to specify precisely with tags and context items. As we will not implement adaption rules, the idea is to present the most similar cases and let the user complete the task of solution generation. In addition, learning new cases in an automatic way is also left out at the moment, although it could be interesting for future work. The main part we want to focus on is the similarity-based retrieval. We use further the metaphor of a case to describe a tagged piece of code, because it suits well to the usage context of the application. In the following, we will introduce the vocabulary and similarity measures used in our application.

4.3 The vocabulary

In this section, we define the attributes we use to describe a case, i.e. a code snippet. By now, we already know that a case contains a set of tags and another set of context items for the problem description. Furthermore, it must contain the piece of code itself as the solution part. The mentioned information is obviously essential as it is necessary for the retrieval itself. But in addition, there are also other aspects to consider such as provenance or additional context information as to cover user interests and maintenance issues. Furthermore, we want to take into account the relevant kinds of information identified in Chapter 3. Table 4.1 outlines all the attributes we use and describes their value type and category. Some of the attributes are well-suited for filtering, for example *document type*, *project name*, *file path*, *author ID* or *creation date*. In the following, we will introduce them one by one and describe their purpose in detail.

Tags Technically, a tag is a sequence of letters and whitespaces. As a case may be described with more than one tag, the value of this attribute is always a set of strings.

Context items According to the discussion in Section 4.1 we describe a work context as a set of context item identifiers. As we use character sequences for the identifiers, the value type of the attribute is again a set of strings.

Code snippet We argued in Section 1.1 that source files move and code evolves. Furthermore, the snippets must be exchangeable among community members. As a consequence, the content of the tagged piece of code must be part of the case and is stored with this attribute. Since the resources we provide tagging for are textual, the value type is String.

Table 4.1: Case attributes

Attribute	Value type	Category	Set by
Tags	String (multiple)	Problem description	user
Context items	String (multiple)	Problem description	coTag
Code snippet	String	Solution	user
Additional info	String	Solution	coTag
Document type	String	Provenance	coTag
Project name	String	Provenance	coTag
File path	String	Provenance	coTag
Author ID	String	Provenance	coTag
Creation date	Long	Provenance	coTag
Rating	Float	Maintenance	user
Rating count	Integer	Maintenance	coTag

Additional info Whereas the context items make possible to process contextual information in a computational way, the purpose of the additional info attribute is to give any kind of information about context and environment for humans. The information about the used operating system is stored with this attribute. It can hold further information about the libraries and APIs used in the code fragment for example, but also other information is imaginable.

Author ID During the inspection of a piece of code that has been published by another member of the communitiy, it may be interesting to know who was the author. By author we mean the person who initially tagged the particular piece of code and created the case. The ID is unique for every member of the community. Storing details about the author in an additional data base avoids overhead within snippet data and makes updating an easy task.

Document type This attribute is used to describe the kind of document the snippet was acquired from. As we found out in Section 3.2, the information about the used programming language or document type is quite important[4]. Furthermore, we can provide filters to search only for particular document types.

Project name Knowing the original project a snippet is coming from makes possible to share project specific programming knowledge more easily, as we can again apply filters to retrieve only the snippets of a certain project. In addition, this information can be interesting for the user as a kind of provenance detail.

File path The path to the original source file is stored with this attribute as to provide provenance information and allows for further pruning of the search space

[4]Note that the used programming language can be inferred from the document type.

to snippets acquired from a particular directory. Moreover, it makes possible to access the original document of a snippet again as to look for its original environment or to compare their contents respecting code evolution.

Creation date This attribute holds the time of snippet acquisition. Its value describes the point of time in which the piece of code was tagged and committed to the case base. The time is coded as the number of milliseconds that have passed since midnight of January 1st in 1970 in terms of the UTC[5].

Rating For reuse and maintenance issues, we provide another attribute to tell users about the quality of the code fragment. In contrast to most other attributes, this one cannot be determined at acquisition time but specified as a kind of user feedback in the moment of snippet presentation like for example after retrieval. Although this information is more interesting in a multi-user scenario, it can also be used to denote quickly a personal opinion of a personally tagged piece of code. As the quality is given as a rating, the value type is a float with a range from one to five, whereas five means high and zero means low quality. The advantage of this kind of quality coding is that it can be processed computationally and respected at retrieval. Contrarily, it has the disadvantage that the reason for a particular rating cannot be given. Thus, a developer does not know why for example a snippet is low-rated. However, this attribute is just meant to provide user feedback and to support maintenance.

Rating count As the rating is given only by a single number, another attribute is needed to give information about the total number of ratings. Because every user has only one vote per snippet, this attribute is not needed for private code snippets. But as soon as a snippet is published, every community member can rate it. In order to calculate the average of all ratings, we must store the total number of ratings.

4.4 Similarity measures

Whereas filters are used to exclude inappropriate cases from the search space in advance, the similarity measures are used to determine the relevance of the remaining cases and establish a ranking. In general, background knowledge is needed to determine the similarity between two concepts. If no background knowledge is given at all, similarity is maximal in case two concepts are identical and zero otherwise. Such a similarity measure applied for tag-based retrieval would result in an exact match behaviour that can be seen at the mentioned TagSEA application [Storey *et al.*, 2006] and popular websites such as delicious[6]. In order to compensate spelling and typing errors, it is suitable to derive similarity syntactically.

[5] Universal Time Coordinated

[6] `http://delicious.com`

Table 4.2: Examples: similarity between two tags

Tag 1	Tag 2	Similarity
SWING	*AWT*	Syntactically very different, but semantically similar ⇒ customise this similarity
JScript	*JavaScript*	Syntactically and semantically similar ⇒ no customisation necessary
SWING	*XING*	Syntactically very similar, but semantically different ⇒ customise this similarity

If semantics are given, the conceptual relatedness can be exploited for similarity calculation. In addition to the similarity between two single concepts, a way to determine the similarity between two sets of concepts is needed in our case.

In the following, we describe the tag similarity measure and continue with the measure for the context items. Finally, we show how the global similarity is determined.

Tag similarity

The similarity measure for tags comprises two parts: the similarity of two single tags and the similarity of two sets of tags. In order to support semantical relations between tags, we view tags as symbols (instead of strings) having their very own meaning. Accordingly, the similarity value of two tags is looked up in a symbol table which is filled incrementally by the user (see Section 4.5). The main scenarios are illustrated in Table 4.2. Although the trigram similarity measure is capable to deal with spelling and typing errors and in some cases it can even be used to reason from syntactical to semantical similarity, it yields wrong values in many cases. To overcome this lack, we provide means to override trigram similarities with values customised by the user.

However, as the tags can be entered freely, they will not be found in the table in many cases and especially not in the beginning. In such a situation, when the similarity obviously cannot be determined semantically, *coTag* determines the similarity in a syntactical way by applying trigram matching [Manning & Schütze, 1999].

Having defined the similarity assessment for two tags, we now define the similarity assessment for two sets of tags. For each query tag, the most similar case tag is determined as its partner. Thereby, the case tags can be reused, which means that two query tags may have the same partner. Unused case tags are ignored. Eventually, the arithmetic mean of all partner similarities is returned for the final similarity value. Mathematically, it formulates as follows:

$$\text{sim}_{Set}(\vec{q}, \vec{c}) = \frac{1}{N_q} \sum_{i=1}^{N_q} \max_{j=1,\cdots,N_c} \text{sim}_{Single}(q_i, c_j)$$

where $\vec{q}$ and $\vec{c}$ represent the tag vectors of query and case, sim_{Set} and sim_{Single} are the two parts of the similarity measure, and N_q, N_c are the respective numbers of tags in query and case.

Context similarity

For the calculation of similarity between two work contexts many approaches can be imagined. To keep it simple, we do not apply any background knowledge about the given context items and their relations between each other. In contrast to the tag's similarity measure, we do neither provide trigram similarity nor the option to edit similarity relations. Thus, the only thing we can tell is whether two context items are identical or not.

Inspiration can be gained by thinking of the TF-IDF[7] metric [Salton *et al.*, 1983]. Therefore, the notion of *term frequency* must be understood as a weighing of the context items given by the work context. But as our definition of work context does only provide qualitative but no quantitative information about the used context items, weights about the relevance of the particular items are not available. The *inverse document frequency* in the transferred meaning could give information about the significance of the context items. If a particular item is part of the work context of many code snippets, it is probably less significant than a context item that plays a role in only a few snippets. Eventually, we could apply weights for the context items. But the inverse document frequency must be calculated each time before a retrieval is made or cached and updated in a smart way. Hence, we keep this discussion in mind for future work but will not follow it in this thesis any further.

Since no weights are available for the context items and the items itself can only be compared with respect to equality, the similarity must be calculated using the two item sets only. We can count how many of the items are contained in both sets. In order to normalise this value to the range $[0, 1]$ we divide this number by the total amount of items involved, which is formalised by the following formula:

$$sim = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

Global similarity

So far, the measure to determine the similarity between two sets of tags and the measure to determine the similarity between two work contexts are defined. A last similarity measure is needed that yields one overall similarity value for given

[7] Term frequency - inverse document frequency

tag and context similarities. It is arguable which similarity is more relevant for the overall similarity. In this work we consider both values equally relevant and calculate an unweighted mean.

Furthermore, we provide a flexible way of similarity calculation that depends on the query. If users decide to exclude the work context from the query, the global similar is calculated only by means of tag similarity. In return, the global similarity is calculated only by means of context similarity if no tags are included in the query. If neither the tags nor the work context is given, the cases of the search scope are returned unsorted.

4.5 Similarity customisation

In general, users do not bother to configure a system if the benefit does not compensate the effort. If further the interface to edit similarity relations is too complicated, the feature will not be used. Anyway, the user will not customise any similarity values as long as the retrieval works fine. But if the retrieval yielded irrelevant cases or did not yield relevant ones, a more sophisticated similarity measure is demanded. As mentioned before, to provide a mechanism for unobtrusive and incremental improvement of the tag similarity measure is one of the main goals of this work. We think, this is the right moment to attract the user's attention to similarity customisation.

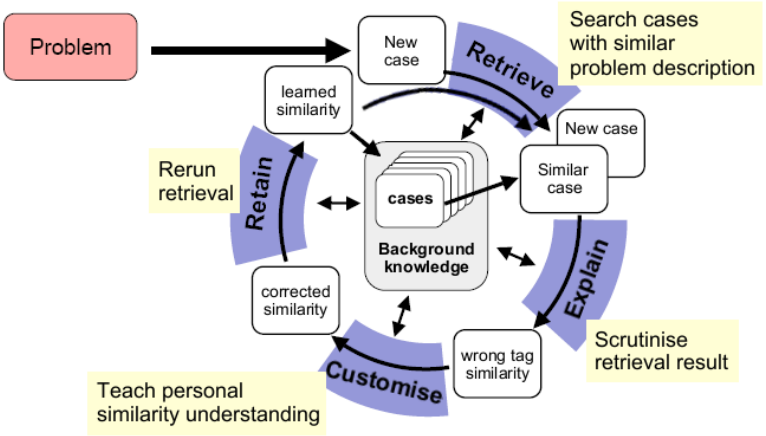


Figure 4.3: Modified CBR cycle from a similarity container perspective

The user needs to know first why the system assigned a wrong rank to the case, as to determine the wrong tag similarity. With the help of action explanations, which are further explained in Chapter 7 the system becomes *scrutable* [Tintarev & Masthoff, 2007] and lets the user find out what went wrong. Eventually, when the misleading tag similarity relation is found, the user can customise it to teach the

system his understanding of tag similarity. For the same query posed, the system can now return more appropriate results due to the improved similarity measure.

With a closer look, this workflow can be understood as a derivation of the original CBR cycle (cf. Figure 4.3 and Figure 4.2). The *retrieve* step returns cases that are similar to the query by the system's definition of similarity. If the user is puzzled or unhappy with the ranking of a particular case, the system can *explain* how it was retrieved. In case an inappropriate tag similarity is discovered, the user can *customise* it to improve retrieval quality. Eventually, the system *retains* this new piece of knowledge. The user may rerun the retrieval with the same query as a suitable solution to the problem may not be found yet and obtain different results. In contrast to the original CBR cycle [Aamodt & Plaza, 1994], which describes knowledge acquisition with respect to the knowledge container *case base*, our modified cycle describes knowledge acquisition with respect to the knowledge container *similarity measures*.

Chapter 5

Experience exchange

Communities share common goals, common interests and work on common projects. Their members exchange information when they meet at their place of work, at conferences or even in their spare time doing sports and the like. While they share common interests and follow common goals, they use a similar set of tools, libraries and algorithms. In addition, they share a similar perspective on things and use their own terminology, which consists of words with syntax, semantics and inter-relations. To enable knowledge exchange with *coTag*, we support exchange of code snippets and provide further solutions to share similarity and tag knowledge. Moreover, author information is provided about every publisher to explain the provenance of code snippets, tags or similarity customisations.

Despite of many shared commonalities, an individual member may have a different view on specific points, though. In our application, this can manifest in a different opinion about a similarity relation between two tags, the suitability of tags attached to a code snippet or the utility of a snippet at all. According to our guiding principle to continuously support the personal information model of the single user, we elaborated means to deal with these conflicts.

The first section gives a survey to the coTag Community Knowledge Repositoryand compares it to a market place. The exchange of code snippets is illustrated in Section 5.2. It lays open how snippets can be published, rated and added to a personal collection. Section 5.2 reveals and resolves the conflicts that occur during the exchange of similarity understanding. How already used tags are shared among community members for tag proposition is described in Section 5.4. Finally, the chapter closes with a short description about the available author information.

5.1 The coTag Community Knowledge Repository

A market place is a meeting point for the exchange of a variety of products. Some people visit market places to obtain, some to offer products, where these roles may change over time. Think of flea markets or ebay[1] for example. Furthermore, mar-

[1] `http://www.ebay.com`

ket places differ more or less in their range of available articles, which are often related to a certain subject area and imply a certain context depending on the clientage. In a store for water sports equipment, all offered mp3 players are waterproof for example. By offering a specialised range of products or being localised in a particular region, the market adapts to a particular clientage. By enhancing the number of customers addressed, the offered products do in many cases not meet the special demands of particular individual customers anymore due to the need to serve the masses. Many of these aspects can be transferred to the domain of knowledge exchange.

While a market place serves as a transshipment point for products, a coTag Community Knowledge Repositoryserves as a transshipment point for knowledge, whereas the exchanged kind of knowledge depends on the community. [Liebowitz & Beckman, 2000] defined a knowledge repository as an:

> *"... on-line computer-based storehouse of expertise, knowledge, experience, and documentation about a particular domain of expertise. In creating a knowledge repository, knowledge is collected, summarized, and integrated across sources."*

A coTag Community Knowledge Repositoryis a knowledge repository for the exchange of code snippets, author information and similarity understanding of one arbitrary community (Figure 5.1). To address a certain community, the repository

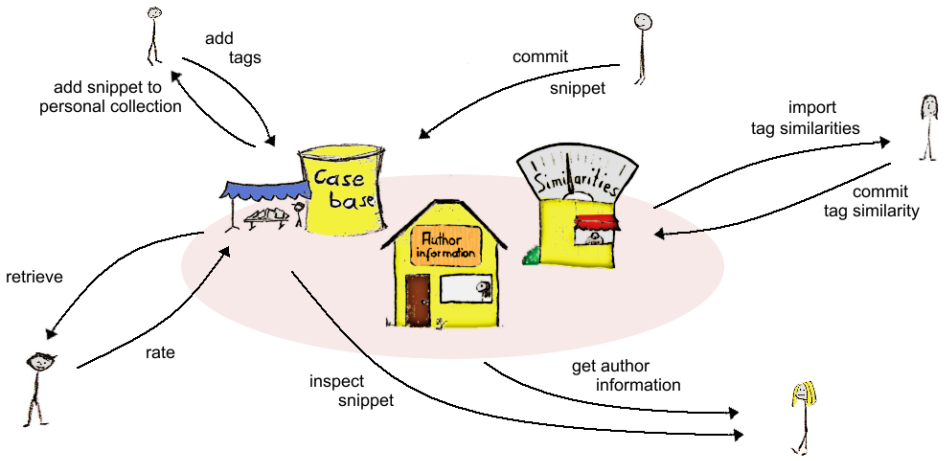


Figure 5.1: The coTag Community Knowledge Repository as a market place

must be accessible in a certain way and offer 'knowledge products' of a particular subject area. In return, a repository that is located at a particular spot may attract certain developers and support the advancement of a (new) community.

5.2 Code snippet exchange

The central purpose of experience exchange in *coTag* is to let developers share coding knowledge within their community. Therefore, personal code snippets can be published via the coTag Community Knowledge Repository, for which a unique author ID is needed. In the moment of publishing, an exact copy of the snippet is made for the repository and added to the community's public case base. In return, the members of the community can pose queries to retrieve the public snippets, whereas the personal similarity measure is used.

The user can rate the snippet as to give feedback about its quality or utility. This may give a hint for other users whether this code is eligible for reuse or not. As every community has exactly one vote per snippet, a snippet can have many ratings. Thus, the repository has to keep track of the submitted ratings and provide information about the number and average of a snippet's ratings to the public. This information could further be useful for maintenance in a way that low-rated snippets are removed after a while for example.

In addition, we allow users for adding snippets from the repository to their personal collection. This brings the advantage that interesting snippets can be accessed without a network connection to the repository. And in case the user participates in more than one community, it is not necessary to remember the snippet's repository to find it. In the moment a snippet is added to the personal collection, the user can *personalise* the snippet by customising the attached tags. By adding the new tags also to the snippet in the coTag Community Knowledge Repository, we enable *collaborative tagging*.

5.3 Similarity exchange

The idea of similarity exchange is to commit customised similarity relations to the coTag Community Knowledge Repositoryand import those of other community members. Storing the customised similarity relations of every community member via the repository is quite simple. The question is how they can contribute to the similarity measure of a single member. We want to apply the community's similarity customisations to the tag similarity measures of every member while respecting his own similarity customisations. Thus, we cannot import a tag similarity relation from the repository if the respective user already customised this value on his own. Furthermore, we must still allow him to override an imported similarity value if he disagrees with it. As we see, conflicts occur inherently when there is more than one opinion about the similarity relation between two tags. To make it possible to import these conflicting similarities into the measure of a third user who has not yet customised the respective tag similarity, these similarity values must be merged to a single one. We currently do this by calculating the average.

The resulting behaviour of these decisions is as follows. The tag similarity measure is now composed of three layers of different priorities. The basic similar-

ity measure is given by the trigram algorithm. The imported similarities from the repository override the trigram similarities, whereas the arithmetic mean is used to resolve conflicts. The most dominating layer form the similarity customisations of the local user, which override the similarity customisations from the repository.

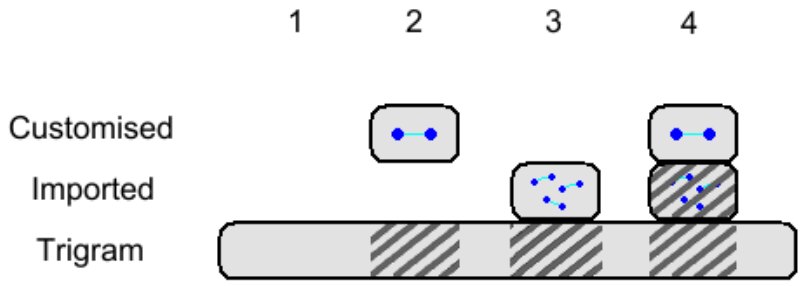


Figure 5.2: Priorities for tag similarity overriding

Figure 5.2 illustrates the four different cases that can occur. The first one describes a situation in which neither the user nor a community member customised the similarity value and the trigram similarity is used. For the second case, the similarity was defined only by the user and overrides trigram similarity. In the third situation, not the user but one or more members of the community set the similarity value which again overrides trigram similarity. The last case describes a situation where three different similarity values are available, but only the one from the user is regarded.

5.4 Tag exchange

As described in Section 3.3, we want to propose already used tags when a piece of code is being tagged. For this purpose, the coTag Community Knowledge Repositoryindexes all tags of the published snippets by provenance. This allows *coTag* clients for requesting them by their project, folder or file provenance.

5.5 Author information

Every snippet has an attribute to give information about its author. Although it is not necessary for the personal use of the system, the unique author ID is demanded in the moment a snippet is being published. Everyone who retrieves this snippet has then the option to look up information about the author at the repository. As explained later in Chapter 6, we offer only the author's name and an image. Further interesting information could be for example the URL to his homepage, company or e-mail address, phone number and so on.

Part III

Implementation

Chapter 6

coTag use cases

This chapter illustrates *coTag*'s functionalities from an end-user perspective and groups them in a few main use cases. Step by step, we explain how users can acquire code snippets from the editors of *eclipse* (Section 6.1), pose queries to retrieve snippets from their local or community repository (Section 6.3) and how snippets are presented (Section 6.4). We show further how to demand tag proposals and how to configure *coTag* (Section 6.2).

A use case is a description of a system's behaviour as it responds to a request that originates from outside of that system. Use cases describe the interaction between a primary actor (the initiator of the interaction) and the system itself, represented as a sequence of simple steps [Sommerville, 2004]. Although they are preliminarily used for requirements analysis, we use them here to explain how the system can be used. For the *coTag* system, three main use cases can be distinguished: one for case acquisition, one for retrieval and one for preference configuration (Figure 6.1).

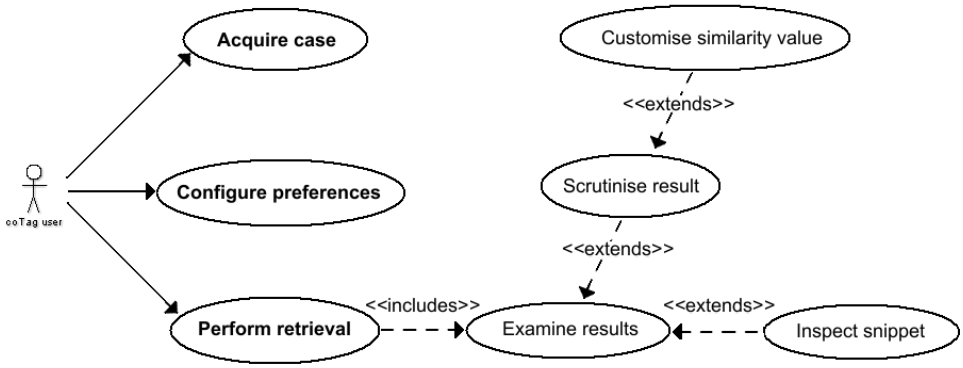


Figure 6.1: The use cases of *coTag*

The first use case begins at the moment the user deems a code passage valuable for reuse. Concentrating on working at a certain task, the user needs a quick

and easy way to tag the desired piece of code and continue with his work again. There are three kinds of initial situations for code tagging. In the first situation, an interesting piece of code was just finished editing and its developer already has the feeling the code could be interesting for reuse. In another situation, the developer is searching for a particular piece of code inside his workspace to reuse it for his current taks. After it was found, the piece of code is tagged to find it faster next time. The last kind of starting situation, the developer wants to kick off or update the *coTag* experience base with some useful snippets.

While the first use case aims at enhancing the experience base, the second one has the goal to bring the actual benefit of the *coTag* system to the user. In its initial situation, the user wants to consult the system as to achieve a particular programming goal. It comprises the composition of a query and the examination of its results, whereas code snippets must be presented appropriately and similarity calculation must be explained. Lastly, as we want to support the step from explanation to intervention, it also includes similarity customisation (see Section 7.5).

The last use case deals with the administration of the *coTag* settings. It includes the configuration of the server connection, personal information, such as tag proposition and similarity exchange. Both former use cases may be extended by this one in case the user wants to publish a snippet or query for snippets on a server, for which the connection has not been configured, yet.

6.1 Case acquisition

As the user encounters a code passage he deems valuable to register with the *coTag* system, the following steps are required. The respective piece of code is specified by selection. The entry *coTag this!* in the context menu[1], opens the *coTag capture dialog* depicted in Figure 6.2. It shows the selected piece of code in the scrollable text area at the bottom, which can be used for review or to carry out minor changes to the code snippet. The user is prompted to enter some characterising tags in the text field at the top of the window. The tags are separated by whitespaces, but there is also the option to use quotes to define multi word tags.

Tag proposition, as described in Section 3.3, is activated automatically half a second after the last key stroke, if the character left to the cursor is not a whitespace (Figure 6.3). In addition, tag proposals can be demanded directly by pressing the key combination CTRL + SPACE.

The proposals are grouped into 5 categories, which can be flipped by pressing the mentioned key combination again:

1. Content-based tag proposals
2. Personal and community tags attached to the respective document

[1] In order to provide quick access, the dialog opens also with the key combination CTRL + ALT + ENTER. The same combination can be used to close it again.

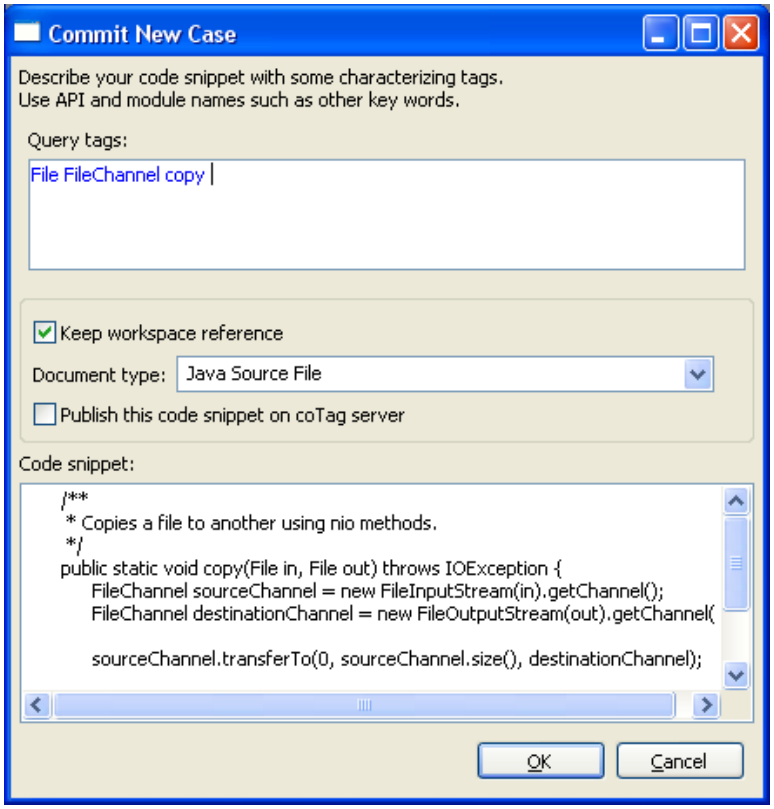


Figure 6.2: Dialog for case acquisition

3. Personal and community tags attached to the documents of the respective folder

4. Personal and community tags attached to the documents of the respective project

5. all personal and community tags

In the first moment the propositions show up, the proposals of the first category are shown.

The user can specify further, whether the snippet should be published on the *coTag* server, which type of document this piece of code comes from, and whether a reference to the original source file should be kept. Although the document type is determined automatically already, we want to provide the option to take control over it. Keeping the workspace reference means that the original file and line number the tagged code snippet was derived from can be found again (more on this in Section 6.4 and Section 8.1). The OK button becomes selectable when at least one tag is entered and closes the acquisition procedure with its selection. Both check boxes hold their state of selection for the next time.

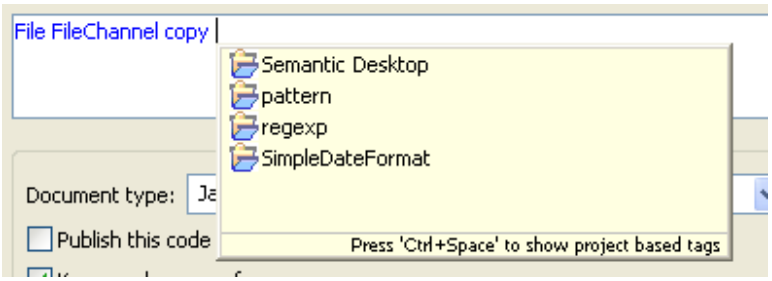


Figure 6.3: Tag proposition based on tags of the folder

In case the tagger wants to publish his snippet and the server connection is not yet configured properly, a dialog as shown in Figure 6.4 pops up to tell the user what information is missing to publish the snippet. With the closing of this window the preference page is opened automatically to enter the missing information.

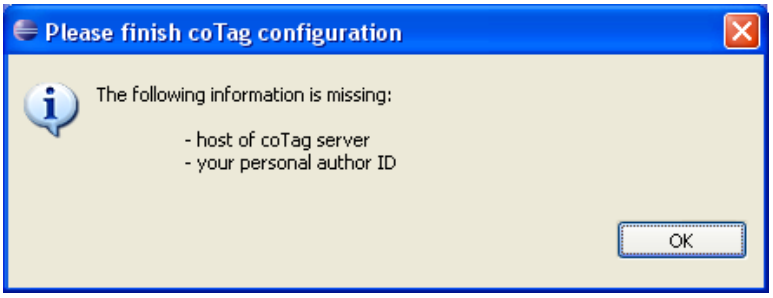


Figure 6.4: Dialog explaining a server connection failure

6.2 Preference configuration

A couple of settings can be customised by the user. Some of them are required, for example to enable the exchange of knowledge via a *coTag* server while others are not no technical prerequisite but offered to apply personal preferences.

The screenshot displayed in Figure 6.5 shows that the preference page is grouped into three categories: server connection, retrieval options and author details.

The technically required information to connect to a *coTag* server consists of the server's host name only, which can be entered freely in the respective text field. The author ID, which is conceptually necessary for server requests, is an information about the user and is hence grouped with the fields for his name and depiction. By pressing the *calculate* button a dialog pops up and demands the user's e-mail address (Figure 6.6) in order to calculate a unique MD5[2] [Rivest, 1992]

[2]Message-Digest Algorithm 5 is a widely used cryptographic hash function with a 128-bit hash

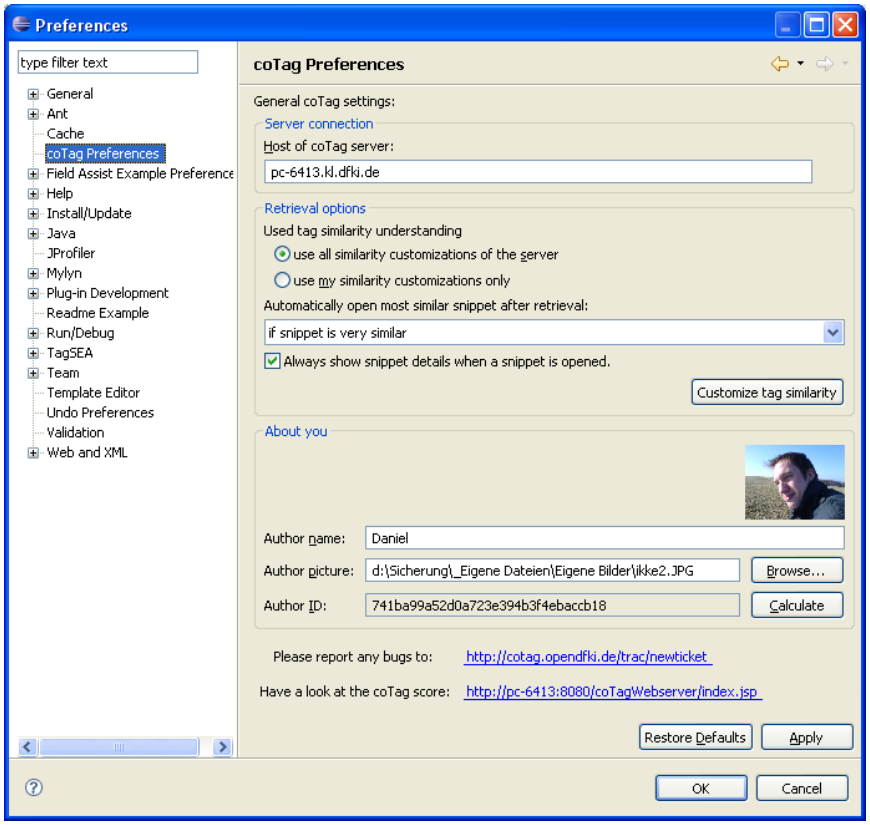


Figure 6.5: The *coTag* preference panel

sum. Because users may hesitate to disclose this private piece of information, the purpose is explained and the user is assured that no further use of this e-mail address will be made. In contrast to the author ID, the name and the path to an image file can be entered.

We always want to support both, the *private* and the *collaborative* use of *coTag*. Hence, it is allowed to turn off similarity imports in order to reject all similarity customisations contributed by others. If this option is selected, the user's own similarity customisations only are regarded to build the tag's similarity measure. If he decides later to import foreign similarity values again, he can turn on this feature again. As all similarity customisations are logged at the server, a full rebuild of any user's similarity measure can be made at any point of time for both options. Moreover, all tag similarities can be inspected or modified with the *general similarity editor* described in Section 7.5.

Another issue concerning the retrieval is the question wether to show the content of the most similar snippet automatically or not. On the one hand, we want to reduce the number of clicks from query specification to snippet presentation. On

value.

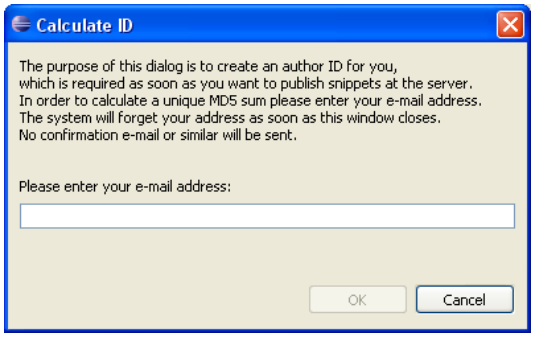


Figure 6.6: Calculator panel for the author ID

the other hand, if the best result of a retrieval is yet very unsimilar to the query, the snippet is probably not helpful and may be disturbing if it is opened as a matter of routine. Thus, we introduced a minimum similarity value to tell whether it is worth opening automatically or not. But as the value of this threshold is debatable and depends strongly on personal preferences, we provided a combo box for configuration. By default, this value is set to "very similar".

Two web links are offered at the bottom of the configuration panel. The first one is directed to the ticket system at the project's home page and can be used to report bugs and recommendations. The second one points to the *coTag* evaluation page (Section 9.1).

6.3 Retrieval

As soon as the developer encounters a programming task for which he wants to reuse a code snippet, *coTag*'s query view can be used to search for private snippets or to examine the snippets of the community (Figure 6.7). In the first place, the query is defined by the tags entered in the upper text field. As with the acquisition dialog, the delimiters are again whitespaces and tags can again be quoted to let them include delimiters. All the tags attached to the snippets reachable within the specified search scope are used as proposals. The way they are offered reminds at the text completion proposals of web browsers or the tag proposals given at the query fields of popular tag based web sites[3].

Furthermore, as *coTag* supports context-sensitive search, a small checkbox is provided to include the current text selection of the active editor as context description. The context extractor (Section 8.2) for the respective document is used to identify the context items involved in the selected text. If the checkbox is unselected or a text selection is not given, the retrieval is based on tags only. In the following, we demonstrate the basic functionality with the help of a simple query consisting of the words *logging*, *init* and *configuration*.

[3] see for example `\url{http://www.youtube.com}`

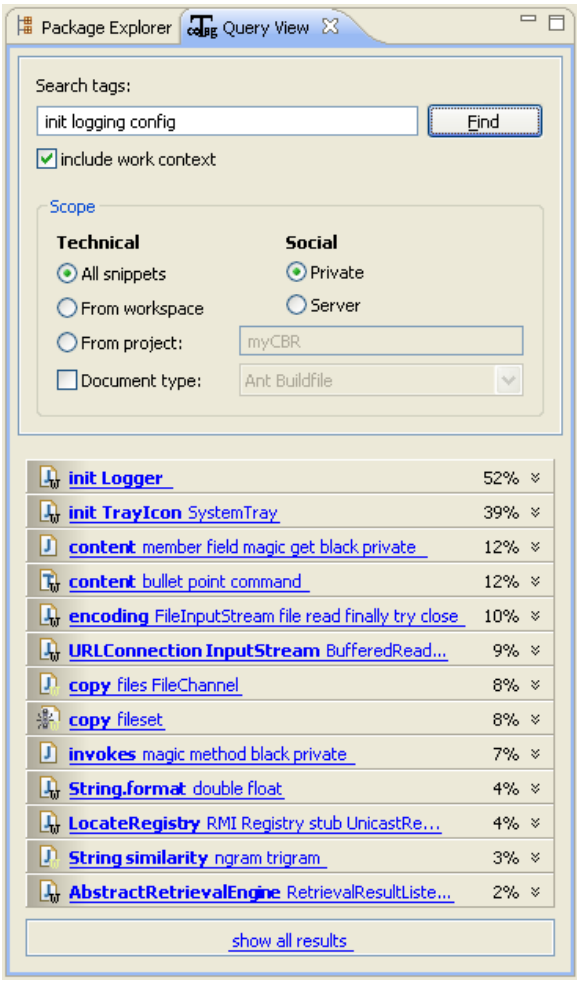


Figure 6.7: *coTag*'s query view showing an example query and its results

While the query tags and context are used to determine the ranking of the retrieved code snippets, the search scope defines the set of snippets chosen for this search, also known as the search space. Two independent scales can be distinguished to define the scope: a *technical* and a *social* dimension. At the present time, the social dimension is defined by two possible values. While the *server* option addresses the query at the snippets at the coTag Community Knowledge Repository, a query in *private* mode returns snippets of the personal collection only. A third option could be imagined to restrict the search scope to a set of chosen authors. The technical dimension determines the set of snippets by their relatedness to workspace and project. By selecting *all snippets* all available snippets are used. The *workspace* option includes only those snippets that have their origin in a file available in the local workspace. The last technical search option allows for filtering the snippets of a particular project. At last, independent from

both search dimension is the option to filter snippets by their document type. This combo box is always pre-selected with the type of the currently edited document, but it can be configured manually also.[4] All search scope options are combined in AND mode, meaning that the intersection of all single snippet sets is chosen as search space.

By pressing the *Find* button, *coTag* starts processing the query.[5] If the option to include the work context is selected, the context items are determined from the current text selection first. As this may take a few seconds, an animated icon appears behind the context checkbox to lay open *coTag*'s state of query processing. As soon as all information is obtained, the query is submitted to the retrieval engine of *myCBR*. Before the result list is shown in the lower part of the query view, another animated busy icon is used as a place holder.

The result of the query is shown as a list. Each row represents a retrieved snippet and is divided into three parts. To indicate its document type the respective icon is shown on the left, which is already provided by the *eclipse* platform. It is further decorated with a small "w" if the snippet's original source file is available in the workspace. Moreover, if there is even a workspace reference available, a golden "w" is shown. On the right side, the total similarity to the query is displayed in percentage. The small arrows give the hint that the list behaves like an accordeon component and shows detail information when they are triggered. In our case, the provided details give action explanations about the result [Roth-Berghofer & Cassens, 2005], which are described thoroughly in Chapter 7. The tags of the snippet are displayed in the middle of the row and use the most of the available space. They are sorted descendingly from left to right respecting their similarity to the matched partner (as explained in Section 4.4). All partnered tags are painted bold to show that they were assessed to be similar to the query in a certain way, while the remaining tags are displayed modestly. All in all, they are rendered as a link to indicate that the snippet's content appears as soon as the user clicks on them.

6.4 Snippet inspection

How a code snippet is presented depends on the way it was acquired. If the user disabled the option to keep the workspace reference, the task is simple, because only the originally acquired piece of code is available. To show its content, *coTag* uses the default editor respecting its document type. Figure 6.8 shows a screenshot of the *eclipse* workbench revealing the content of the most similar snippet for our example query. because we do not support later editing of snippets, the editor's purpose is limited to their presentation and set to read-only mode. As the user is

[4] When a new code snippet of an unknown document type is acquired the list of known document types is updated.

[5] If the social search scope is set to *server* and the connection settings are not properly configured, *coTag* prompts to enter the missing information in the same manner as described in Section 6.1

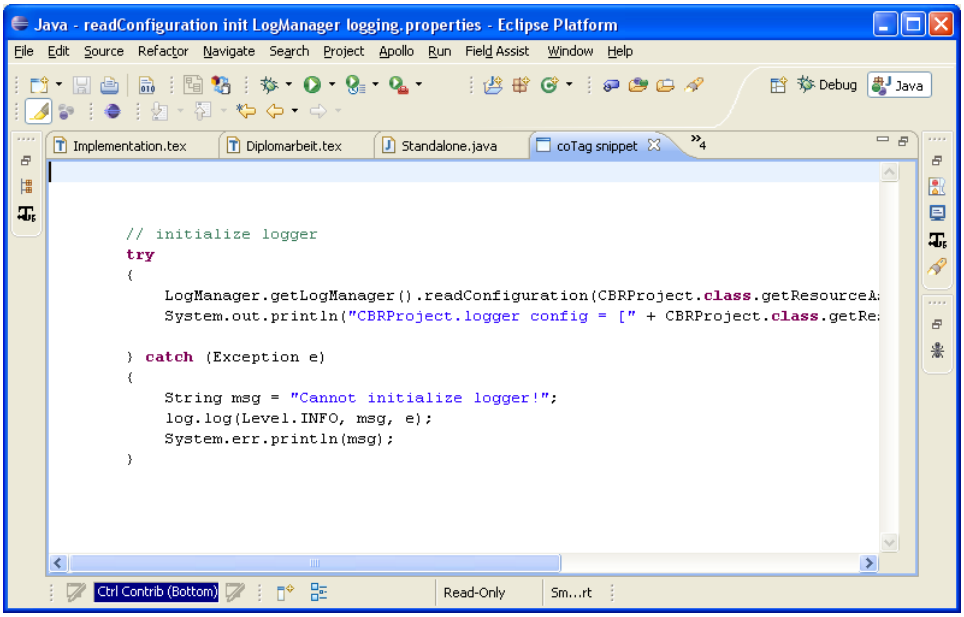


Figure 6.8: Code snippets are displayed with the default editor of their document type.

now enabled to inspect the snippet or copy it for reuse, the end of *coTag*'s retrieval use case is reached.[6]

But if the user chose the option to keep the workspace reference we have to cope with an inherent dualism stemming from the existence of two concurrent code snippet instances: the originally tagged piece of code which is part of the *coTag* experience base and the passage in the original document of the workspace. Because the latter one is part of a project and serves a functional purpose, it is possible that its content changes to suit other requirements. Additionally, as it is allowed to customise the acquired code fragment in the capture window (see Section 6.1), both instances may already be different immediately after acquisition. In such a case, the question is raised which piece of code is of higher value to the developer posing the query. The old piece of code has not changed and fits in any case to the given tags and context information. But on the other side, the advanced piece of code may be improved concerning bugs and errors or is somehow more state of the art while still serving exactly the same purpose. Since the answer to the question is not ours to give, we must offer both of them.

To tell whether or not both snippets are equal, we compare their syntax structures. *eclipse* provides a default parser for every document type, which we use to extract the tokens of each snippet, whereas comments and whitespaces are ignored. If both syntax structures are equal, the original source file is opened and the part

[6]The user can still inspect further snippets of the retrieval result, demand explanations, customise similarities or refine the query.

containing the code snippet is selected (cf. Figure 6.9). In case the snippets are

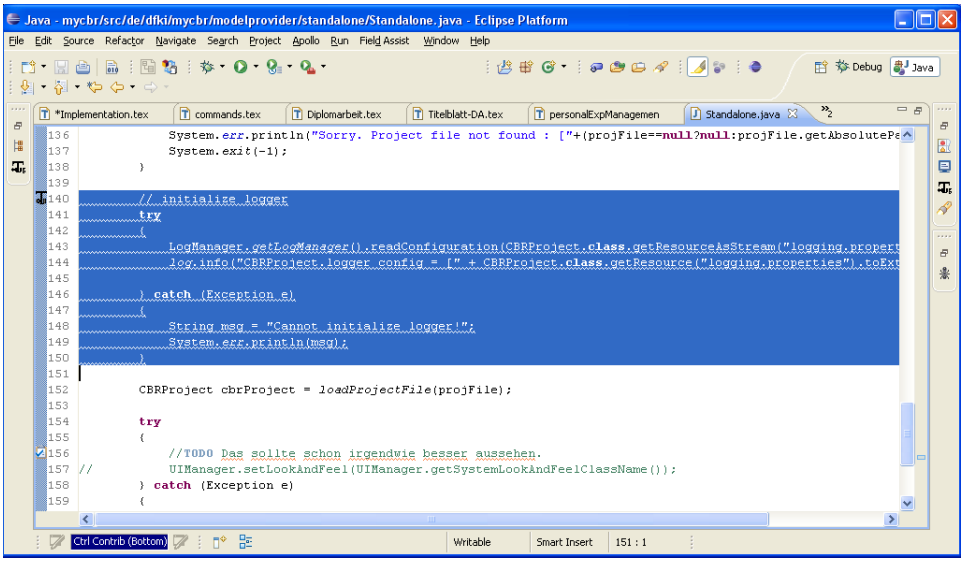


Figure 6.9: Snippet shown as a part of its original source file.

different, both are opened simultaneously in a comparison editor to lay open in which way they differ from each other (Figure 6.10).

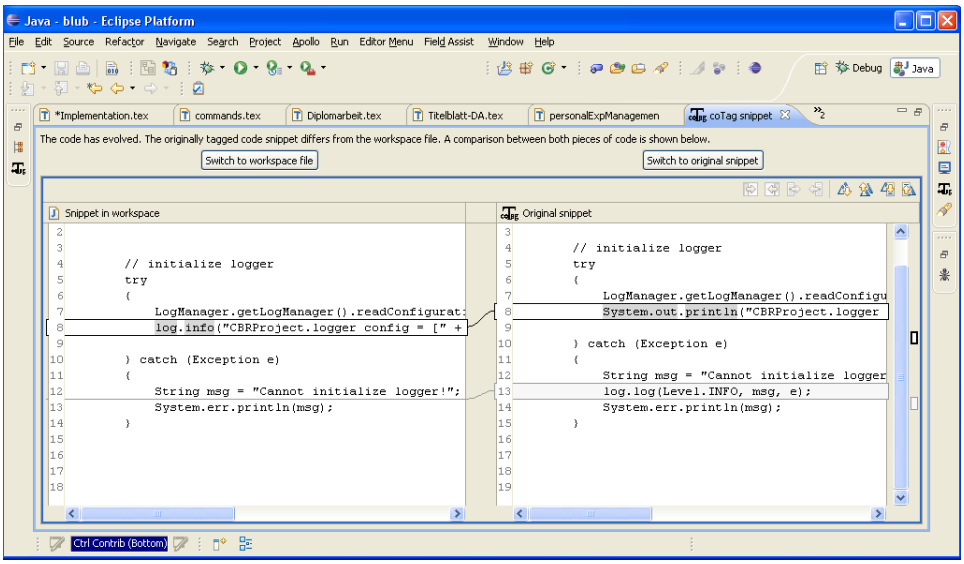


Figure 6.10: A comparison between an originally tagged snippet and its workspace representation

In addition to the code snippet itself, *coTag* provides another *snippet info view* for further details and options, which is automatically shown when a snippet is

opened. A lot of meta information is displayed on the right side, while some further options for snippet inspection and administration are offered on the left (Figure 6.11). The tags attached to the snippet are displayed as well as its provenance,

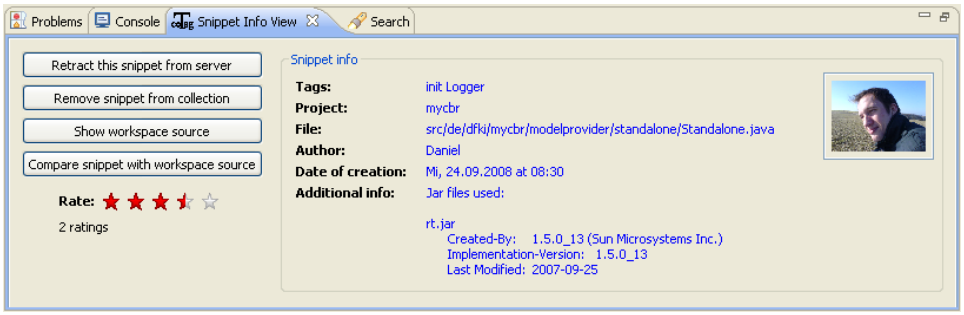


Figure 6.11: Additional meta information about the snippet

which includes the date of creation, some author information and the project name and project relative path of the file it was derived from. Since names are often not very meaningful for new community members, a picture of the author is presented at the upper right corner of the panel to convey a more catchy feeling for the snippet's origination. In spite of presenting the context items, which are used for similarity calculation and may not necessarily be human-readable, the contents of the additional info attribute are displayed here. Read more about this field in Section 8.2.

The topmost button on the left allows for afterwardly publishing a snippet of the private collection. In case this snippet is already published, this button offers to retract it from the server. In a similar way, one can remove a snippet from one's own collection with the second button.[7] If the snippet was retrieved via the *coTag* server, the button can be used to add it to the personal collection, which makes it possible to retrieve it again without being connected to the server and also allows for attaching one's personal tags to it. In the moment the button is pushed, the case acquisition window pops up for snippet personalisation and displays the tags of the original snippet in the tag's text field. Whereas the tags are allowed to be modified, all other meta data cannot be changed. For every way of usage the button labels are adapted to describe its associated action, of course.

The last two buttons offer different editors for snippet presentation. If the project name and the project relative path to the source file are available for the current snippet[8] and the file is available on the system, *coTag* offers to open this original document[9]. The last button provides the option to open the comparison

[7] Being the author of the snippet, its removal from the private collection implies also the removal from the server.

[8] In some cases, this information is not available, because the annotated code snippet was acquired from a document which has no representation as a file, e.g. another *coTag* snippet.

[9] This is not to be confused with the utilisation of code references, which point exactly to the lines of code the snippet was derived from.

editor as shown in Figure 6.10 explicitly. Of course, this is available only in case there is a workspace reference. As we described above, a comparison is presented automatically only if both snippets differ from each other syntactically whereas comments and whitespaces are ignored. Just in order to provide the user with all available options of inspection, we added this button to the panel.

The last feature of this panel is dedicated to giving quality feedback about the snippet itself. Following the style of youtube[10] the inspecting developer is allowed to rate the quality of this contribution. Every *coTag* participant registered at the server has exactly one vote per snippet, which is submitted by selecting one of the asterisks. The meanings of the asterisks translate to *poor*, *not helpful*, *worth knowing*, *pretty cool* and *awesome*, which is further explained with the help of tooltips. On the one hand, the red painted ones indicate the average of all votes. On the other hand, the asterisks can be used to submit a vote by clicking on one of them. Besides the community snippets, the private snippets can be rated also. A rating serves then as a private note about the snippet's quality.

[10] `http://www.youtube.com`

Chapter 7

Scrutinisation of retrieval result

In this chapter, we detail the implementation of the alternative CBR cycle introduced in Section 4.5. In order to scrutinise the outcome of a query, action explanations must be available, which present the details of the retrieval process. As a code snippet is retrieved with the help of similarity measures, all steps of similarity calculation are to explain. Thus, partner matching and single tag similarities as well as the similarity between two work contexts must be illustrated. Eventually, a simple interface to customise the similarity between two tags is needed.

Looking again at *coTag*'s query view (Figure 6.7) one can see two arrows at the right of each row in the result list. They indicate that there is additional retrieval information available. If the user clicks on them the accordeon list reveals an explanation panel below the selected row (Figure 7.1). It illustrates the result of

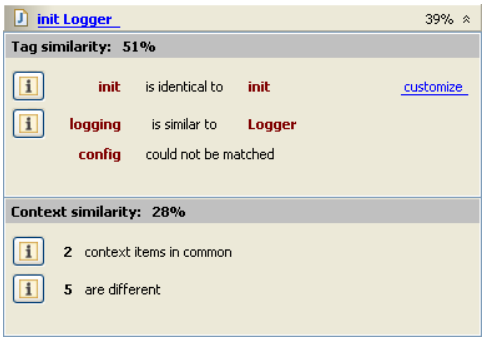


Figure 7.1: Explanation of set similarity between two sets of tags

the tag partner matching described in Section 4.4 as well as the context similarity if a work context was included in the query. In addition, each header provides a tooltip to describe briefly the way *coTag* calculates similarities in general. We do not detail the calculation of the arithmetic mean in the particular case, because the step from the given single similarities to the total one is very simple and is intuitively retraceable.

Harder to understand is the way how single similarity values are determined

for which the buttons on the left open the respective explanations. There are three different possibilities for the provenance of a single similarity value. The similarity can be determined syntactically by trigram matching, which in many cases has nothing to do with its semantic similarity—it may not even be obvious, why two words are syntactically similar (Section 7.1). The user may have customised a similarity relation in the past and may be puzzled, because he cannot remember (Section 7.2). Even more necessary are the explanations for imported similarities, because the user takes notice of them only when a retrieval is done. All this explanatory information is available due to the explanation support mechanism of *myCBR* [Bahls, 2008; Bahls & Roth-Berghofer, 2007]. To customise these values, one must activate the link at the upper right corner of the shown widget (Section 7.5).

7.1 Trigram similarity

Since we want to lay open what the trigram algorithm does, we must take another close look at it. The two given words are divided in a set of fragments, which have the length of three characters. The similarity is then calculated by the following formula:

$$sim = \frac{|A_1 \cap A_2|}{|A_1 \cup A_2|}$$

where A_1, A_2 are the two sets of word fragments. Hence, the degree of similarity equals the percentage of fragments both words have in common.

As to provide a simple explanation that is easy and effortless to understand, we want to bring both words involved face to face and highlight their commonalities in a visual way. But highlighting every common fragment as a whole may be confusing, so we decided to determine the degree of similarity contribution for all characters of both words, which makes a simple visualisation possible. By *degree of similarity contribution* we mean the amount of common fragments a character is part of, which is referred to as *hit count* in the following. Each character of each word participates in three different word fragments, which in turn consist also of three characters. For every matching fragment the hit count of all characters involved is increased by one. Eventually, all hit counts are divided by their maximum for normalisation.[1]

The explanation is then presented by colouring each character with respect to its normalised hit count by applying a certain colour gradient. Low hit counts result in a pale colour and high hit counts lead to a dark red colour of the character as can be seen in Figure 7.2.

[1]The maximal hit count possible depends on the given words and is neither fixed nor limited in advance. Just think of the words *googolplex* and *googolplexplex*.

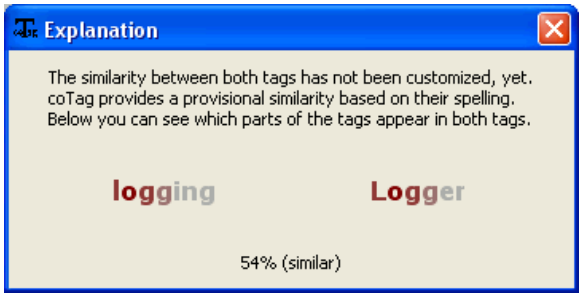


Figure 7.2: Trigram explanation

7.2 Customised tag similarities

When a tag similarity was customised by the local user, an explanation is most likely not demanded, because the similarity accords to the user's understanding of similarity. But if this is not the case, the customised similarity relation may be inappropriate for this particular retrieval. However, the system must be able to explain.

The provenance information we can provide comprises at least the date of customisation. We can tell further that it was the user who changed the tag relation.

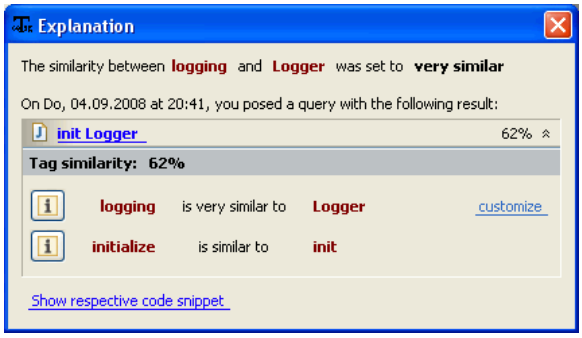


Figure 7.3: Explanation of a tag relation once customised by the local user

Since we applied similarity customisation as a feedback to a result, we can draw the connection between customisation and its related retrieval result. As the reason for customisation is probably founded in the outcome of the past retrieval, we illustrate the past situation with the help of a screenshot of the explanation at that time (Figure 7.3). The user has further the option to inspect the respective snippet.

7.3 Imported tag similarities

An imported similarity relation has its origin at another user's work place. On the one hand we claim that people participating in the community group share a

common understanding of things. On the other hand, people tend to have different opinions and different views on things, which was the motivation to follow a strictly personalised approach with this work. We try to overcome this trade-off situation with the help of explanations, again. Additionally, if the user disagrees, he can override the respective similarity value. As an explanation, we can tell who customised the similarity value, which may have been several users (Figure 7.4).

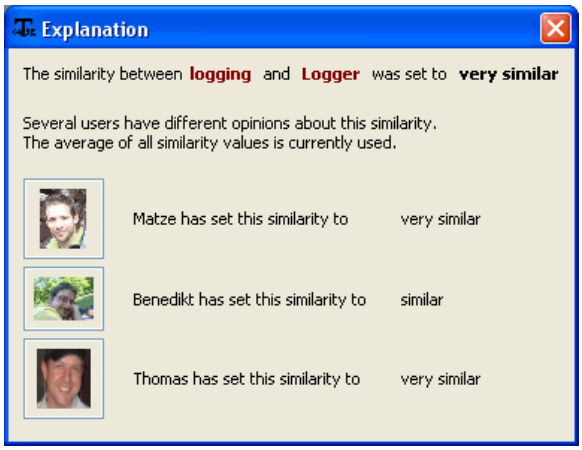


Figure 7.4: Explanation of imported similarity values

7.4 Context similarity

To explain the resulted context similarity, we simply tell the user how many of the query's and the snippet's context items are equal and how many are not. To understand the degree of contextual similarity it is not necessary to list the single context item identifiers, but if the user is interested, they can be demanded by the buttons on the left (see lower part of Figure 7.1).

7.5 Similarity customisation

Given the action explanation for a retrieval result, the user may disagree with *coTag*'s similarity understanding. The customisation button on the right of the explanation panel gives the opportunity to edit the tag similarities as shown in Figure 7.5. Unless they are syntactically equal, all similarities between two matched partners can be modified with the help of a drop-down list. Although in *myCBR* the similarity could be set to any value in the range between zero and one, we decided to offer only a few values, which correspond to a particular numerical value (Table 7.1). The advantage is that their meaning is clear. As soon as a new value is selected in the list, a similarity update is committed to the *coTag* system (see Section 8.3).

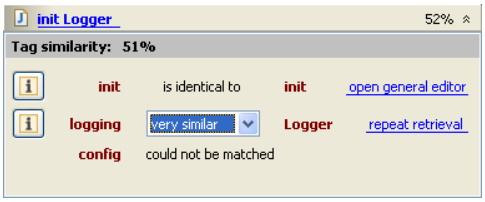


Figure 7.5: Customisation of tag similarities involved in a particular result

Table 7.1: Translated similarity values

Translation	Similarity value
unsimilar	0.0
partly similar	0.25
similar	0.5
very similar	0.75
identical	1.0

Another advantage of this kind of similarity editing is to know the context in which the similarity was edited. An adjustment of a similarity value does not only provide the information that a similarity relation between two tags is set to a new value. Available is also the information which query and which case are currently given. Under this aspect, similarity editing can be seen as feedback to the shown results of a certain query. One finds out why a suggested case fits or fits not to the given query. A screenshot of the action explanation is made to provide this information when the user demands an explanation for the customised similarity value again as described in Section 7.2.

Not only the degree of similarity for the already matched tags can be modified. Also the other relations between query and case tags can be introduced with the help of a general similarity editor (Figure 7.6). Note that if the general similarity

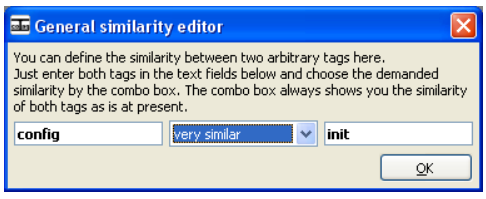


Figure 7.6: All tag similarities can be inspected and customised with the general similarity editor.

editor was opened from the preference page, no context information is available and the explanation is limited to the date of customisation.

Chapter 8

System architecture

coTag consists of two main software components: The local client, which is integrated into the IDE, provides the user interfaces and manages the personal knowledge base, whereas the *coTag* server is used to manage and provide interfaces for the coTag Community Knowledge Repository.

The IDE chosen for integration with *coTag* is called *eclipse* and is a widely used and powerful open-source plug-in framework. It supports a great variety of programming languages and document types by providing convenient tools and editors, that bring syntax colouring and content assistance. Together with its good software design and the great amount of developer information on the web, *eclipse* represents a good IDE to develop *coTag* for.

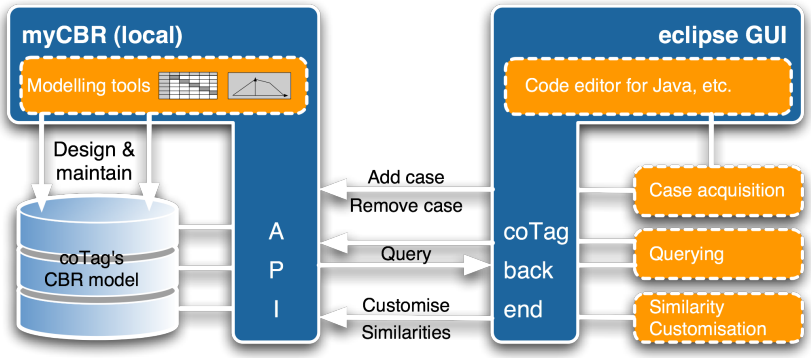


Figure 8.1: Architecture of the *coTag* client

For the CBR part, we use the open-source tool *myCBR*[1] which is a plug-in for the ontology editor Protégé[2] and provides a set of editors to easily design a structural CBR model. Furthermore, it offers an API for retrieval functionality and

[1] http://mycbr-project.net/
[2] http://protege.stanford.edu/

the insertion of new cases. In addition, *myCBR* supports a variety of explanations [Bahls, 2008], which is a fundamental precondition for the use with *coTag*.

The architecture of the *coTag* client is illustrated in Figure 8.1. It dissects the plugin installed at the developers *eclipse* distribution and describes its internal communication. The *coTag* backend serves as a mediator for the communication between the user interface and *myCBR*. It stores the acquired code snippets and provides the similarity measures such as the tag similarities, which are updated whenever a similarity value of a tag relation is customised. In addition, it holds provenance explanations for all customised and imported similarity values. As *myCBR* provides user interface for similarity measure modelling and case inspection, it can be used further to maintain *coTag*'s CBR model. The front end consists of three different UI elements for case acquisition, querying and similarity customisation, using the standard widget toolkit (SWT) of *eclipse*, which are explained in Chapter 6 and Chapter 7. The query view creates a *CotagQuery* object (see Section 8.3) after the *find* button was pressed, and forwards it to the *coTag* backend, which opens the *myCBR* project to pass it further to the CBR system. After the results of a query are returned from *myCBR*, the *coTag* backend transforms them into *CotagSnippet* objects and passes them to the query view. The Case acquisition UI is accessible via the context menu of a text selection.

Figure 8.2 illustrates the complete architecture including the *coTag* server. While the upper part depicts the local plugin as described before, the lower part represents the server module which manages and provides interfaces for the coTag Community Knowledge Repository. Furthermore, the depicted modules can be grouped by columns, whereas the left part represents the *myCBR* backend and the right part shows the specific modules for the *coTag* application. Whenever the user publishes a snippet or customises a similarity value, the *coTag* client submits it to the server. Any kind of communication is done by synchronous RMI[3] calls. For retrieval, the server applies the user's personal similarity measure. Although the similarity customisations of all users are logged at the server and can be used to rebuild the personal similarity measure of every user, a server query includes the table of customised similarity values. As with the *coTag* client, retrieval at server side is also done by *myCBR*. For the import of customised similarity values, the *coTag* client checks for similarity updates at the server as soon as a private or server query is posed.

Providing workspace references for acquired code snippets to allow users for inspecting their original code passage in the workspace file requires a sophisticated mechanism due to evolution of code. Therefore, we implemented a certain coTag *marker* which is explained in Section 8.1. To determine the work context for a snippet or a query as defined in Section 4.1, we use particular *context extractors* which are explained in Section 8.2. The general software components which are used by server and client are described in Section 8.3. Specific components to handle the tasks of the *coTag* server are explained in Section 8.4.

[3]Remote Method Invocation, see `http://java.sun.com`

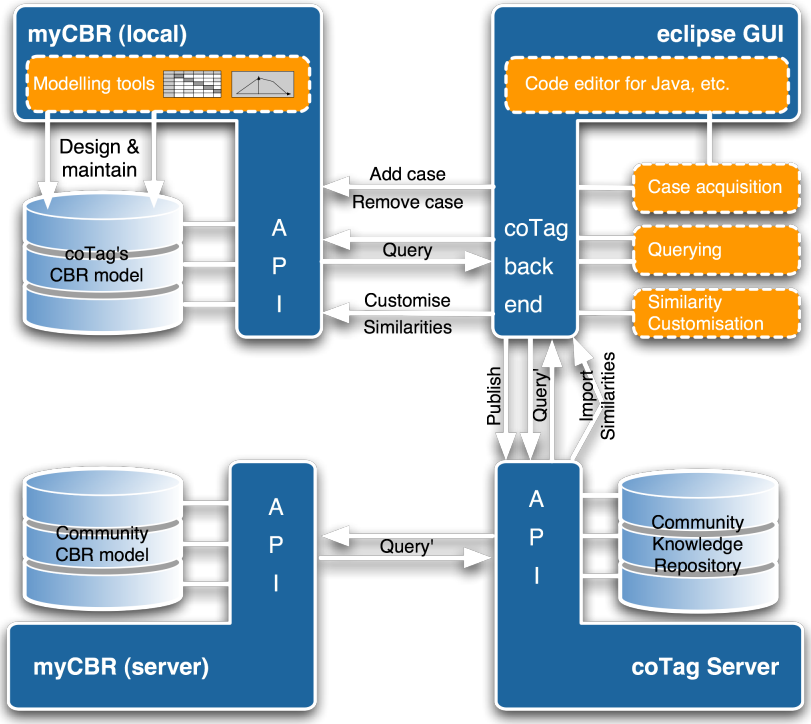


Figure 8.2: Overall architecture

8.1 The *coTag* marker

The option to keep workspace references for an acquired code snippet (explained in Section 6.1 and 6.4) involves a sophisticated mechanism. The task is to know the exact lines of code within the source document from which the tagged code snippet was acquired. Simply storing the respective character offsets or line numbers is not enough, because the source file may be developed further causing the original lines of code to be moved, changed or even deleted. The only proper solution, that does not involve the application of vague heuristics, is given by keeping track of these lines whenever the document is edited.

*eclipse*provides a marker framework allowing us to specify the beginning and the end of a text fragment within any kind of *eclipse* editor. It automatically manages to update their position, cares about their persistance and further allows us to define an additional attribute, which is needed to store the ID of the code snippet it belongs to. When a code snippet is retrieved, *coTag* checks whether the original source document is available and whether a marker with the snippet's ID is available. If this is the case, the character positions of the beginning and the end of the code fragment can be read.

In addition, *eclipse* enables us to display an unobtrusive icon at the vertical

ruler the editor at the beginning line of the respective text position (see Figure 6.9), which is shown also if the user just steps by. If the icon is clicked with the mouse, the lines of the tagged code snippet are selected. Hovering over brings up a tooltip to reveal its tags. As a last feature, the context menu contains an entry to open the snippet in the *snippet info view*.

Unfortunately, *eclipse* does not yet support the exchange of these markers in a team. Being currently limited to tagging of code *snippets*, this feature would further enable users to collaboratively tag *sections* of their shared resources. We made a try at bridging this gap but recognised quickly that this task demands great care and effort, because of the different versions of the distributed source documents. The TagSEA plugin avoids these difficulties by inserting special comments in the content of the code [Storey *et al.*, 2006]. Other TagSEA users of the team obtain the marker eventually when their TagSEA plugin analyses the code. We declined to implement this approach as we want to share markers for every kind of document including those that do not allow inserted comments. A future approach will probably be based on the integration with the team support of *eclipse* and the exchange of specific files that contain all information about the markers in a shared project.

8.2 Context extractors

In order to determine the work context for a code fragment that is being used as context input for the query or that is being captured with *coTag*, procedures are needed to identify the *context items*. As they depend strongly on the document type and programming language, a small framework is designed to implement new and access available context extractors. In addition to the work context, the component is also used to provide the human readable content for the attribute *Additional info*. At the moment, *coTag* provides context extractors for Java classes and XML documents, which are described in the following.

Context items of a piece of Java code are the classes of all objects, methods and constants used in the selected code fragment. As identifier, we use their full qualified names. The additional information for the user contains the path, the version and creation date of the JAR[4] files used. This includes also the information about the used version of Java.

As XML is a very generic document type that does not allow to draw conclusion about its purpose, it is hard to determine the context items. For future work, it could be interesting to analyse its DTD[5] file. As an abstract solution, the XML context extractor returns the names of the tags and properties used in the selection. Additional information is not provided at this generic level. *eclipse* gives further information about its usage, which could be used to provide a more specific solution. For example, ANT build files are described in XML. Their context items

[4] A *Java archive* is a packaged collection of Java classes that can be used as a runable program or library file

[5] *Document Type Definitions*, `http://www.w3.org/XML/`

should include also the content of the usual tags like *target* or *depends*.

8.3 General components

SimilarityUpdate When the user customises a similarity value, all associated information is stored within this value container: both tags and the new similarity value. It represents an atomic similarity knowledge unit, which is used to assemble the whole similarity measure of a user. They are directly applied to the tag similarity measure of the local CBR system. In addition, they are committed to the SimilarityManager of the coTag Community Knowledge Repositoryif the server is accessible.

CotagQuery Although *myCBR* already provides a general query component to retrieve cases from an arbitrary *myCBR* project, we developed a specific one for *coTag* to make all available features explicit and transparent. It allows for defining the query tags, the context items, the two dimensions of the search scope and the additional filter for the document type.

CotagSnippet As with the CotagQuery component, the *myCBR* cases are wrapped with an interface to provide simpler access to the attributes as defined in Section 4.3. Furthermore, it gives information whether it was retrieved from the personal collection or the coTag Community Knowledge Repository.

TagManager To propose suitable tags for a given document, we need an interface to obtain personal and social tags depending on the selected proposal category (cf. Section 6.1) There are two kinds of TagManager components. The local one is used to administer the personal tags at the client side, while the other one is used at the server side to manage community tags. In case the server is not accessible, the community tags are left off.

Activator This is the heart of the *coTag* plugin and represents a central access point for all other components. Every *eclipse* instance has exactly one such object. It provides the interfaces to acquire and retrieve snippet, to customise similarity values, to get access to the TagManager, etc.

8.4 Server components

Identity Manager An identity is a container to hold author information, which in our case is the name, an image and the author ID. The IdentityManager is a component to administer the author information of the community. It provides access to Identity objects for a given author ID or allows for modifying image or author name.

SimilarityManager This component stores the similarity customisations of all community members. When a user customised a similarity value, the SimilarityManager is notified and appends an entry in a similarity customisation history file. An update is applied for the similarity measures of the other users as soon as they access the server. With the help of time stamps, the SimilarityManager finds out whether a user needs an update or not. If a user customised a similarity value in a moment when the server was not accessible, the similarity update is fetched next time the user poses a query at the server.

RatingHandler For personal ratings, the case's *rating* attribute can be used directly. But for the coTag Community Knowledge Repository, a RatingHandler is needed, as every community member has only one vote per snippet. When a rating is submitted, it updates the attributes *rating* and *rating count* of the affected case immediately.

Part IV

Conclusions

Chapter 9

Evaluation

As stated in the introduction, we wanted to achieve shorter development times and a higher quality of code in the first place. We therefore assumed that this goal can be reached by improving the access to personal coding experience. We wanted further to support the user's language and understanding of things as to yield retrieval results that are relevant respecting the personal point of view. In addition, our goal was to improve retrieval quality by taking into account the user's work context. Supporting the exchange of knowledge with other community members, our goal was to help developers learning and solving their tasks. We wanted also to build a retrieval mechanism which is tolerant to spelling and typing errors. Eventually, the system should be able to explain the outcome of a query.

For the evaluation, the *coTag* software was handed out to thirteen developers at the Knowledge Management group of the German Research Center for Artificial Intelligence in Kaiserslautern. A *coTag* server was installed on the institute's intranet. Until publishing deadline of this thesis, six weeks remained the users for using the system. After the test period, we analysed the log files of the test persons and conducted interviews to get information about the system's utility and the value of its single features.

Some preparations were made before the actual evaluation phase began, which are explained in Section 9.1. Section 9.2 points out how we evaluated user acceptance of *coTag*. The results of a questionnaire are presented in Section 9.3. The conclusions are drawn in Section 9.4.

9.1 Preparations

As to avoid software related issues causing biases in the evaluation results, a thorough verification of the software quality was required. Not only reliability of the single components under separate inspection is required but also their safe communication processes and state of configuration. Thus, we tested the software by simulating a multi-user scenario including load tests and simultaneous requests at the server. More details on this test are given in Appendix A.

In order to kick off the communities experience base, we started a contest to attract the developers to acquire code snippets and use the system. Besides ranking, additional gimmicks such as *snippet of the day* and *most frequent tags* were published on an internal website. The rules were simple. The publisher of a code snippet earns 100 points as long as he did not publish more than ten snippets. In order to avoid spam, no points were given anymore after 10 snippets were published. In addition, whenever a snippet was rated, its author gained or lost a certain amount of points depending on the rating (see Figure 9.1).

Figure 9.1: The rules for the contest

We had the idea to reward the winner of the contest at the end of the evaluation period as to offer a higher incentive. But we realised that such a measure would bias the evaluation results too much. Users must be attracted to use the system by its actual benefit. We assumed that just by publishing such a ranking the participants would be motivated enough and talk about the tool with other colleagues. We hoped this would help the tool to establish itself at the workgroup.

9.2 User acceptance

Every test user was given an introduction to the tool, which included a walkthrough of the main use cases *preference configuration*, *snippet acquisition*, *retrieval* and *scrutinisation of result* including *similarity customisation*. During the introduction, most test users gave positive feedback on the visual design of the user interface. Neither positive nor negative remarks were given about the handling of the user interface. The trainees quickly understood how the tool can be used. During the test period, none of the users reported any handling related issues. Some minor software related problems occurred because of the use of different Java versions, but we did not receive any other problem or bug reports.

To estimate the degree of user acceptance, we looked into the user's log files.

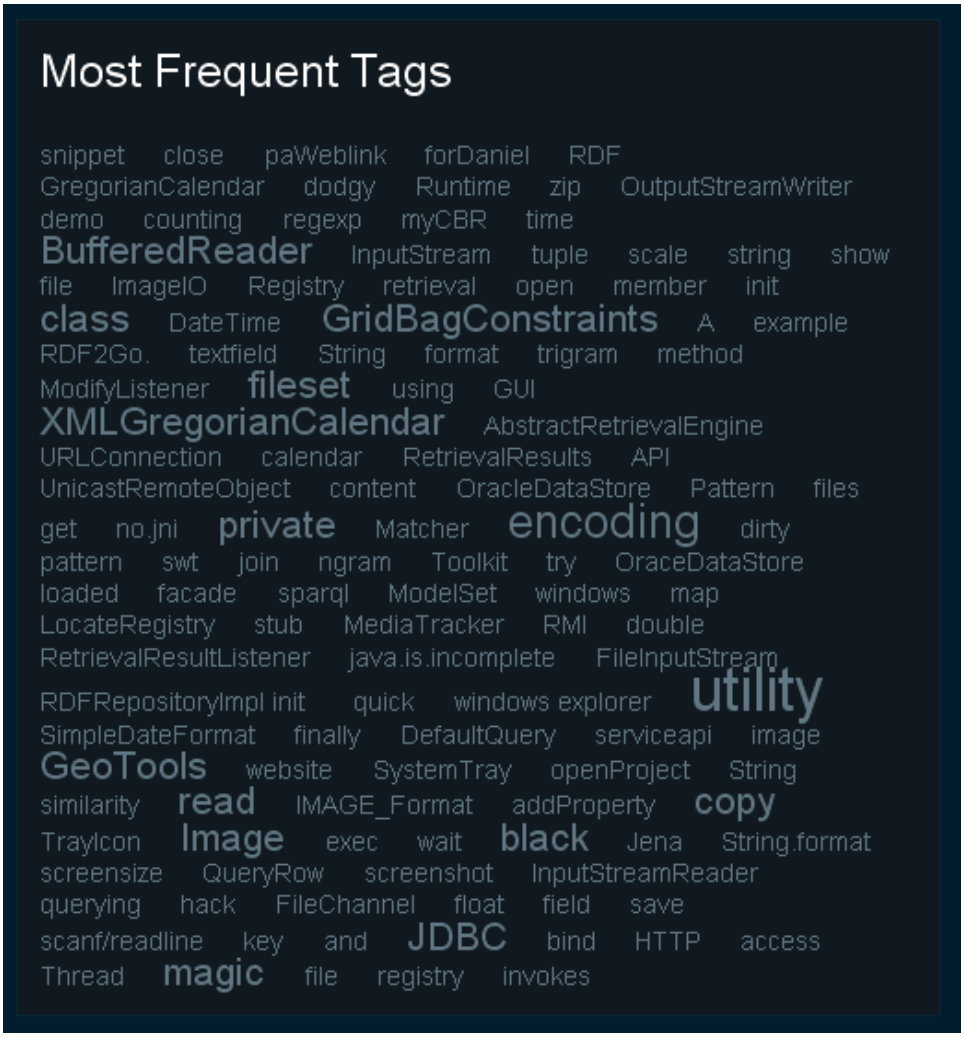


Figure 9.2: Most frequent tags at the coTag Community Knowledge Repository

High activity respecting querying and case acquisition could be determined during the first week of the test period. As was expected in the light of the brief evaluation period, the maximum number of cases acquired by a single user was thirteen, while in average it was 3.9. The number of snippets published at the server was 38. The number of queries posed at the server was 76. Seven similarity relations were customised.

One important issue was to find out to what extent the users could *benefit* from the *coTag* system. We assumed that although it may be interesting to just have a look on a retrieved code snippet, in most cases the user would like to copy (a part of) it to the clipboard and paste it somewhere into his currently edited code. Thus, every time the content of a code snippet was copied to the clipboard of a developer's work station, we logged the event. During the evaluation period, we

logged 15 of these snippet hits at three different users.

Interpreting the acquisition of snippets respecting user acceptance needs some consideration, because many of them were acquired at the beginning of the test phase in order to try out the features of *coTag*. In addition, during the first days, the test persons wanted to participate in the contest. After the first week, the frequency of snippet acquisition decreased but still happened occasionally.

The interpretation of a query respecting user acceptance is difficult. It can express that the user was working on a task for which he wants to reuse or look at a similar code snippet for which the system was actually built. But it can mean also that the questioner just wanted to play around to understand the behaviour of the tool, which was mostly the case during the first days of the evaluation period. In case a query was posed at the coTag Community Knowledge Repository, it may be that the user was just curious about snippets of other users. In addition, one cannot conclude that a query is posed whenever a snippet is demanded. As none of the test users had more snippets in the case base than *coTag* can display in the query view, all personal cases were visible after a retrieval. Thus, some users did not even pose a new query to retrieve a snippet and just selected the suitable ones from the ranking of the last one.

The total number of different tags was 127, where the most frequently used tag occurred four times. The tagging style of the users resembled the tags found with the tag nature study in Section 3.2 (see Figure 9.2). Interesting to see is that many of the tags, which describe a particular module, are written in correct case (e.g. *BufferedReader* or *XMLGregorianCalendar*), which is probably due to the use of content-based proposals.

9.3 Questionnaire

To find out to what extent *coTag* achieves the goals of this work, we designed a questionnaire. Nine participants of the evaluation team were interviewed to collect further comments and remarks to the questions.[1] Note that in many cases, the asked person could not give a clear answer. Thus, we could not count a selection for one of the muliple-choice answers. In addition, many of the given answers were based on opinions and estimations but not on experience. The details to the questionnaire can be found in Appendix B. In the following we summarise the answers and the remarks given to the questions.

Accessing coding experience The answers to the questions about how often a code example could be helpful were quite uniform. Answering how often a code example is demanded during four hours of work was generally problematic, as it depends strongly on the current task. When the user works in an unfamiliar environment with unfamiliar libraries, code fragments are demanded for purposes

[1] Unfortunately, not all of the thirteen test users were available for the interview.

of learning. When the user works in a familiar environment, code fragments are demanded for reuse. For the purpose of learning, code snippets appear to be demanded more frequently than for the purpose of reuse. However, in average the developers estimated that a code example would be helpful approximately four times within four hours. The test persons stated that they also search for an example in the moment it could be useful, unless finding a suitable snippet is not likely or too time-consuming. It was difficult to get clear answers with regard to the distinction between longing for a suitable snippet and actually searching for one. All asked persons declared that they prefer personally created code over code examples found on the web. A quite interesting result was that most of the developers involved want to reuse code fragments as often as possible.

Most of the interviewees stated that they lack of a tool to access their personal coding experience. They use several techniques to find personal code fragments again. The documents containing frequently reused code are often accessed directly without searching, as the developer knows exactly where it can be found. If the location of the containing document is not known, many developers perform a full text search within the workspace. For this purpose, some of the persons keep old and inactive projects in their workspace. One user stated that full text searches take too much time and demanded a tool to perform them more quickly. Two test persons already used a simple tool to store and retrieve code fragments, which resembled a small data base in both cases.

Does *coTag* help? Most test persons stated that *coTag* helps accessing the personal coding experience. Some persons mentioned that the disadvantage of *coTag* is—as we already mentioned at the design decisions in Section 2.3—that cases must be acquired by hand before results can be returned.

Many of the asked persons stated that *coTag* helps exchanging coding knowledge. But four persons did not want to make a statement because they need more time to see whether it would help or not. One person said it could be helpful to convey tricks and special know-how for specific situations. Another one remarked that it was necessary to see how it works with large experience bases.

The question whether it helps solving programming tasks faster was difficult to answer for most of the interviewees. They stated, that it certainly helped when a suitable snippet was found. Especially for specific programming tasks dealing with closed-source libraries as well as in unfamiliar environments, *coTag* could provide benefit by the exchange of know-how and speed up development times. But searching for it takes time and interrupts the current workflow. If one or more queries are posed but no suitable snippet is found means wasting time.

The opinions about whether *coTag* helps improving code quality or not were quite different. Most persons stated that just by using *coTag*, the quality of code would not improve. But if a developer tried to pursue best practices and submitted only code fragments of good quality, *coTag* would help improving code quality. Some of the persons asked think *coTag* helps to improve code quality in situations

where several persons work on a project that uses closed-source libraries. In such a situation, it could help to look at the code of other team members.

Many different opinions were expressed about the completeness of information provided by a single code snippet. While a high fraction stated that a *coTag* snippet provides all required information to solve a task, other persons argued that it does not include enough information about dependencies and other context information. Some mentioned that developers would not provide enough documentation inside the snippet.

Retrieval In general, the interviewed developers stated that *coTag*'s query component helps to find code snippets quickly and reliably. But most of them also remarked that they were not sure about retrieval quality at great experiences bases.

While most test persons were excited about our approach to support similarity-based retrieval, some were sceptical. One person preferred exact match search, because he assumes that the precision of the retrieval worsens for large repositories and queries with more than one key word.

coTag's tag-based retrieval of code snippets was estimated helpful. Some of the interviewees stated that tagging provides a good solution to characterise code, thus, tag-based retrieval is also helpful. Others mentioned that taggers who acquire a snippet have a very personal view which makes snippet exchange a problematic task. One person stated that tags are not enough to characterise code fragments and snippets should be retrievable also by class and method names (in case of Java code).

Our approach to take into account the user's work context was deemed very interesting and helpful. Since context-based retrieval was tested only by way of trial and not during daily work, the developers asked could only estimate its value. One reason could be that the users were flooded with too much other information about *coTag*'s features during the introduction to the tool, and did not remember the context feature when a snippet was needed. In addition, people are used to enter key words in a text field to specify a query, rather than making a text selection in the editor, and performed only tag-based search as a matter of routine. But the general opinion about this feature was quite positive. Note that in many cases, we had to exclude many answers in the statistics of the questionnaire because they were based on estimation rather than on experience. One person would like to apply such kind of search for the retrieval of source files from the workspace.

The question about retraceability of a particular retrieval result aimed at the utility of *coTag*'s explanation support. Although most of the asked persons did not make frequent use of it, they liked the availability of explanations and their visual presentation. Some stated that they just did not need any further explanations about the results, which is just in line with our argumentation. Explanations should be available when asked for.

Code tagging and tag proposition The technique of tagging was deemed valuable especially for code snippets that cannot be retrieved easily by their content, because they lack of significant key words (e.g. SQL or SPARQL queries). But many persons mentioned also that tagging interrupts the current workflow, is time-consuming and causes cognitive load.

Tag proposition was considered helpful. Furthermore, one person mentioned that it helped avoiding typing errors and propagating a uniform vocabulary within the community. Both, content-based tags and already used tags, were considered helpful for proposition.

Collaborative similarity customisation While most developers liked the idea of trigram matching to compensate typing and spelling errors, the opinions were divided on similarity customisation to support semantical connections between tags. Some test persons deemed similarity customisation too time-consuming to be beneficial, even though one needs only a few clicks for generating a link between tags. One person mentioned that he must be really bored until he gets the idea to customise tag relations. Some of them saw it as a good solution to improve retrieval results. Others would appreciate an additional similarity layer of predefined values, gained from WordNet[2] or similar. But the tag nature study from Section 3.2 showed that most tags are not derived from natural language. Then again, most developers liked the idea of collaborative similarity modelling in a community. Even though the interviewed persons could not give profound answers based on experience, most of them considered similarity exchange helpful, while some were afraid that customisations at the system of another community member could affect the own retrieval quality in a negative way.

Snippet feedback Some developers expressed the opinion that comments at a snippet are not relevant for the choice and reuse of a snippet. Others mentioned that they are important to give detailed feedback. Some persons guess that snippet rating could be an interesting feature as soon as *coTag*'s case base is very large and the community is somewhat bigger. The information could be interesting for maintenance. Additionally, one person stated that the top-rated snippets were very interesting for him, as they were new and interesting to him. But others stated that it is hard to express the utility of a snippet with a single rating. It demands too much information compression. In addition, it is a very subjective value.

Additional remarks Although the only purpose of the *coTag* website was to motivate users during the evaluation phase, some test persons found it useful for the *coTag* system in general. But the web page is quite provisional and is missing some features. Users remarked that it would be convenient to navigate through the snippets by selecting tags of the tag cloud. Furthermore, they were missing a kind of history that shows the newest snippets.

[2] `http://wordnet.princeton.edu`

One person was concerned about privacy issues and used *coTag* only in personal mode. Even though users can decide whether to publish a snippet during the acquisition, some users were afraid to make a wrong selection by accident.

9.4 Conclusions

Respecting the evaluation goals, we can draw the following conclusions. For those developers who consider tagging as a useful technique for characterising resources, *coTag* provides a convenient way to acquire and retrieve personal and community code snippets. In combination with similarity customisation, developers are enabled to apply their personal information model.

At least for private use, *coTag* can help to speed up development times. Under certain circumstances, *coTag*'s support of snippet exchange makes an additional contribution to achieve programming goals faster. But this depends strongly on the size and structure of the community and the tagging behaviour of the single community members.

Although the improvement of code quality does not automatically follow from using the tool, *coTag* can be used to support best practices. It depends only on the snippets inserted into the experience base. If the tagger takes care of acquiring only high-quality snippets, the reuse can improve the quality of the code. The same applies for snippet exchange within communities, which probably needs more control and maintenance for this purpose. In return, code quality can also worsen, if *coTag* users apply snippets of bad quality to their program code.

On the one hand, the trigram-based retrieval technique is able to compensate spelling and typing errors. On the other hand, the trigram similarity measure determines wrong similarities for semantically different but syntactically similar tags (and vice versa). By providing similarity customisation, users could apply their own understanding of tag relations and (probably) improve retrieval quality. Furthermore, it is to mention that this feature is too time consuming for some users, while others consider the benefit worth the effort.

A long-term evaluation could help understanding the utility of *coTag* more thoroughly, as some questions are still not clarified satisfyingly. Especially for the exchange of knowledge within a community, it remains to examine how the tool works on great knowledge bases. It would be interesting to measure precision and recall for retrieval especially in connection with similarity customisation and exchange. In addition, context-based retrieval still needs further evaluation, as its utility was estimated only on the basis of hypothetical considerations. The retrieval quality could just not be measured because the search space was too small.

Finally, the system is able to give satisfying explanations about a retrieval result, the provenance of snippets and customised similarity values.

Chapter 10

Summary and Outlook

In this thesis, we have developed *coTag*, a tool for software developers to access personal programming experience efficiently. Users can tag code passages within the editors of the IDE *eclipse* and retrieve them again using the QueryView widget. In addition, *coTag* acquires also context and a lot of additional information about the snippet in the moment of acquisition, which can also be used for retrieval parameters. The similarity measures of *myCBR* are used to perform a retrieval, and its explanation support is used to make a particular result transparent. While *coTag* uses trigram matching to compensate lexical errors in queries and snippet tags, it allows further for similarity customisation. This enables users to apply semantical relations between tags and improve retrieval results respecting the user's personal understanding of things. Via a *coTag* server, code snippet can be exchanged to share coding knowledge with community members and customised similarity values can be shared as to enable collaborative similarity modelling.

We analysed the nature of tags in the domain of code tagging in order to elaborate and evaluate an approach for tag proposition. We further used this information to find a suitable description scheme for code snippets.

Finally, we evaluated this tool with respect to our development goals during a test period of six weeks with the help of thirteen developers, a questionnaire and an analysis of the log files. Although it was not comprehensive enough to draw quantitative conclusions, we could get a lot of feedback and many ideas about improvements and further development in a qualitative way. Nevertheless, there are people who gain personal advantage, whom *coTag* helps improve code quality and helps achieving programming goals faster.

For the future development of *coTag*, we recommend the following:

Enhance background knowledge The similarity measure should be further enhanced with general background knowledge about the domain. As an example, the relation between *image* and *icon* or *class* and *interface* should be set to *very similar* from the beginning. As the tags in the domain of code tagging are in most cases not part of the natural language (see Section 3.2), it is probably not useful to apply

WordNet or other dictionary related approaches. The tags are rather related to code content. Thus, it could be interesting to examine the utility of collocations gained from a great corpus of code resources for this purpose.

Similarity customisation One of the test persons had the idea to improve similarity customisation by analysing the posed queries of a certain time window. As the questioner is looking for a particular solution to solve a task, he may pose several queries to make a find. All key words of the posed queries are somehow related and could be used to offer proposals for similarity customisation. This idea should be further examined.

Comment on code snippets Developers should be allowed to attach comments to a code snippet as to give detailed hints about its practical usefulness and quality. During the interview, we learned that many developers have problems to reduce their feedback to a single rating on a scale of five values. In addition to using this rating value, the system should rather count how often a snippet was indeed reused as we did for the evaluation. This information could be easier to interpret, as it is more objective.

Community support People often participate in several communities. Thus, providing knowledge exchange via one single server is not enough. Upcoming versions of *coTag* should be able to manage a collection of servers. Therefore, it would be necessary to discuss to what extent the similarity relations imported from one server can be transferred to another and how conflicts between different communities can be resolved.

Privacy Since some projects underlie a confidential disclosure agreement (CDA) and users are nervous to publish parts of it by accident, it could be helpful if *coTag* provided some kind of *publishing lock* on the level of projects. If a project was locked for publishing, the option to publish would be disabled for any snippet acquired from project documents. This could put affected developers at lax and let them take advantage of the system without needing to be careful when new pieces of code are tagged.

We encourage researchers and developers to download[1] *coTag*, the server software as well as the *eclipse* plugin, for personal use or to develop it further. For feedback, additional ideas and recommendations, please contact us via the website.

[1] `http://cotag.opendfki.de/`

Acknowledgements The writer is indebted to Prof. Andreas Dengel for his thoughtful criticism, and his kindness and encouragement during the completion of this thesis. The author's greatest debt for personal and professional help is to his advisor Dr. Thomas Roth-Berghofer whose gentle, complementary and clear-sighted guidance has been available and immensely profitable at every stage of the thesis. Appreciation is also extended to Sven Schwarz for his constant feedback, to Christopher Tuot for his supporting ideas in community building, and Georg Buscher for his valuable advice for the questionnaire. The writer also offers his thanks to the evaluation team: Leo Sauermann, Kinga Schumacher, Manuel Möller, Gunnar Grimnes, Andreas Lauer, Christian Reuschling, Yingyan Zhang, Sven Regel, and Thomas Pikson for their time and consideration.

Appendix A

User simulator

Before the software was released and handed out to the evaluation team, it was tested extensively in a complex test setup, because we set great value upon high reliability of the system. Especially in distributed systems, errors are hard to detect due to their asynchronous nature of communication. The goal was to validate the correctness of the system's behaviour in a multi-user scenario including an immense amount of simultaneous server requests. Thus, we created a certain *user simulator module* and several dummy users with their own workspaces and *eclipse* workstations.

Three dummy users have been modelled for the test. Each of them uses a different *eclipse* version and configuration, and a special workspace was composed with the help of real projects from my personal workspace. Eventually, *coTag* records were collected manually within each workspace. All *coTag* repositories have been copied into another folder, which we refer to as *external coTag repository* from now on, and reset to its initial state again.

The user simulator module has the task to access an *eclipse* application, perform a set of user actions at the *coTag* plugin and validate their results. Moreover, it is capable to connect to several *eclipse* applications at the same time as to perform every simulation of user behaviour simultaneously. The *external coTag repository* provides all information necessary to frame reasonable user actions, that are applicable for the respective workspace. A survey of this test environment can be obtained in diagram A.1.

The choice of actions consists of the following types:

- **Submit a new case** Submitting a new *coTag* record is a very workspace specific action and requires information such as source file, code offset and used tags. So the cases from the external repository are used. The action succeeded if the record can be found in the tested dummy user repository and equals its external template.

- **Pose a query** The query tags can be collected from the local tag manager of the external repository. Even randomly created words can be used. But then,

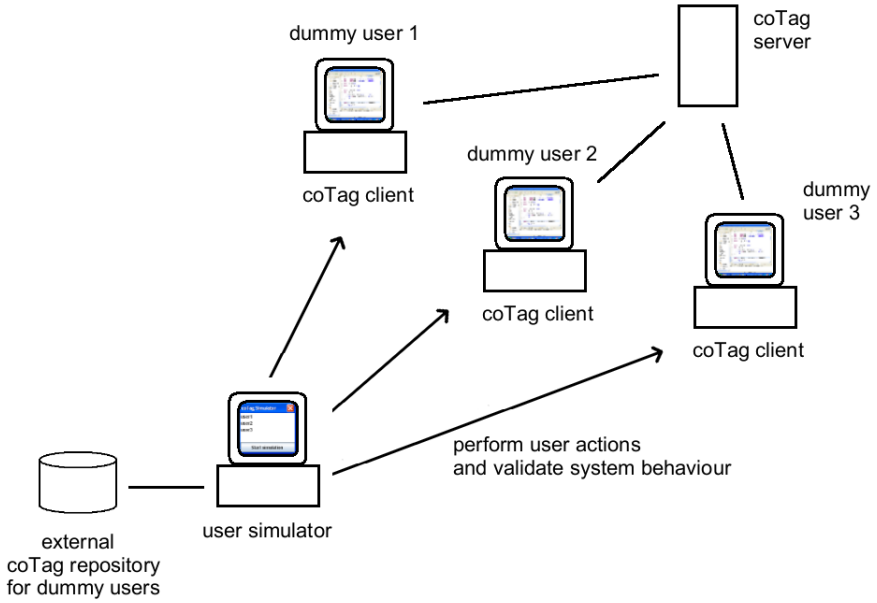


Figure A.1: Test environment

the task of validation is problematic. Hence, we prefer to use the tags of an already submitted case. So we can check whether the the respective case has maximal similarity to the query in the retrieval results.

- **Customise a similarity value** For similarity customisation, some arbitrary tags are chosen from the local tag manager, and a random similarity value is created. A SimilarityUpdate object is then committed to the respective *coTag* client. Validation is done by looking up the new similarity value for the tags.

- **Synchronise similarity measure** In this case, the external repository is not needed. After the synchronisation method of the *coTag* client was called, the new similarity measure is compared with the old one to check whether all locally made similarity customisations are still present.

- **Change user image** An image is chosen randomly from a particular image folder, which is installed at each client solely for testing purposes. This action is not validated automatically, because the support of images is not crucial. This action is supposed merely to make sure this feature does not cause system failure.

For a proper configuration of the test environment, it is necessary to make sure,

that the repositories of the *coTag* server and all *coTag* clients are set to their initial state. An ANT script was created for this task. During the startup of an *eclipse* application, the loaded *coTag* plugin looks for the Remote Method Invocation (RMI) nameserver at the server specified by the URL in the settings. If a *user simulator module* can be found, it submits user name and client reference. Depending on the user name, the simulator loads the user's *external cotag repository* and immediately schedules all actions to perform. As soon as the desired dummy *eclipse* applications are registered, the test run can be launched by pressing the respective button in the simulator window (Figure A.2).

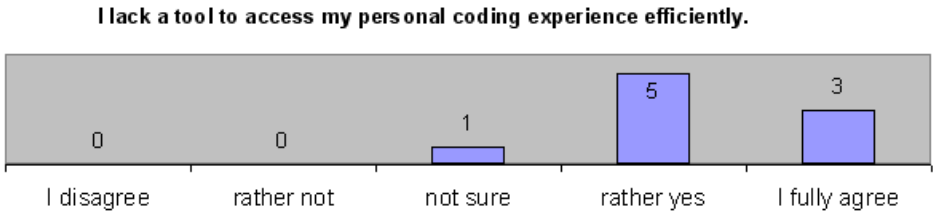
Figure A.2: Simulator window

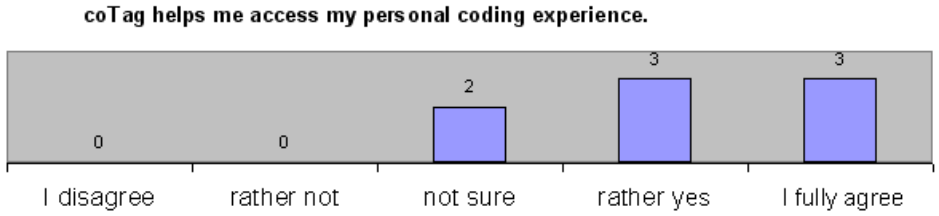
Done so, the simulator starts a separate thread for each registered *coTag* client, which immediately begins to work off its assigned schedule. The result of an action is verified directly after its execution. If an exception occurred or a verification failed, the simulator prints a message to Standard Out and stops all threads.

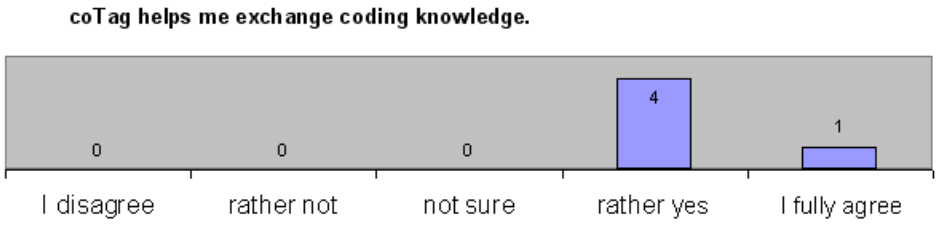
The action schedule of a user comprises the submission of all external *coTag* records, one hundred queries, one hundred similarity customisations, one synchronisation of the similarity measure and one image change.

Appendix B

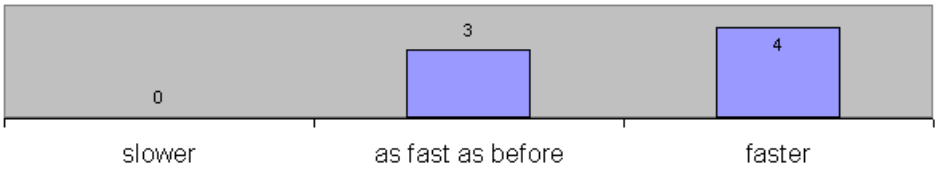
Questionnaire result



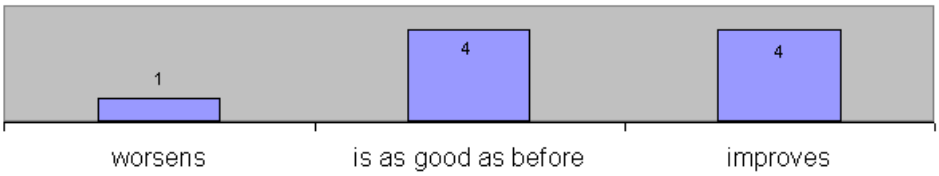




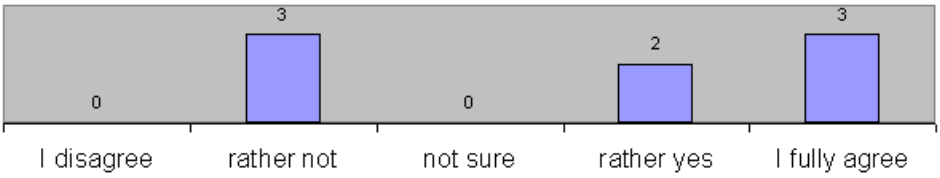
Using coTag I solve my programming tasks



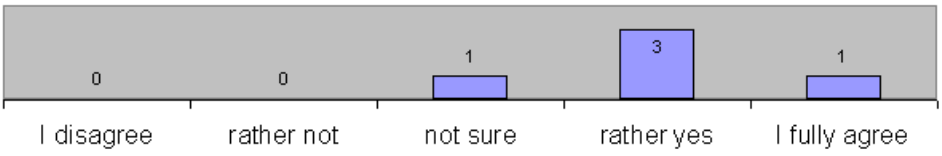
Using coTag the quality of my code



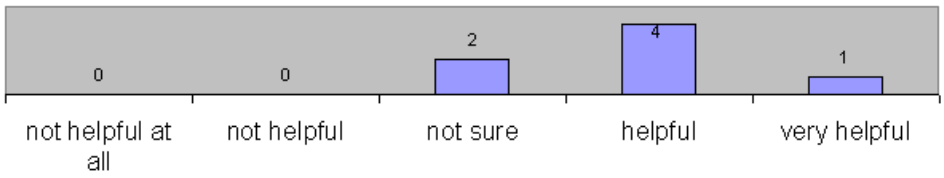
A code snippet provides all important information.



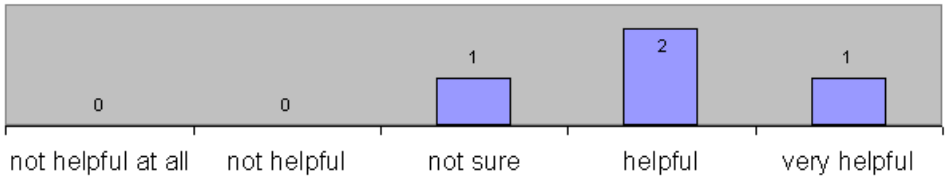
The QueryView helps me finding relevant snippets quickly and reliably.



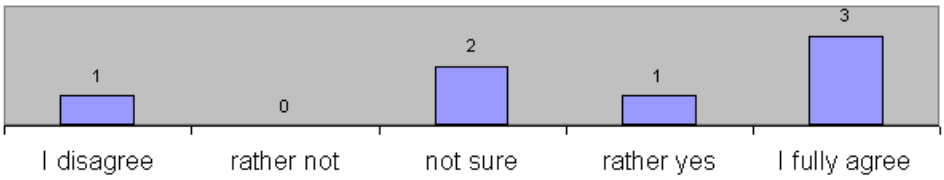
What do you think of coTag's tag-based retrieval?



What do you think of coTag's context-based retrieval?

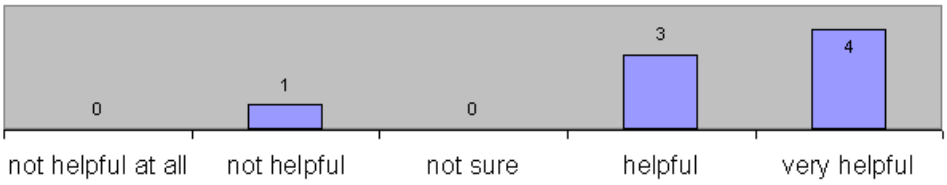


I understand the way coTag performs a retrieval in general.

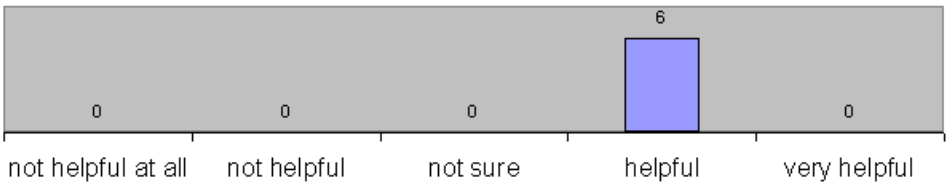


In particular, I understand how a particular result came up.

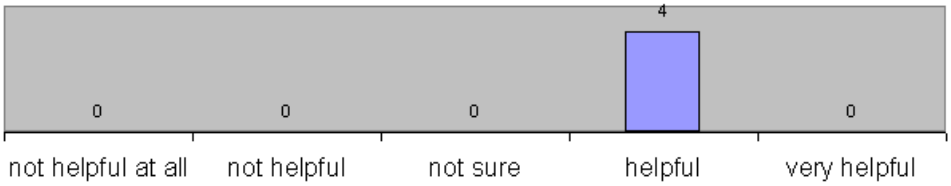
What do you think of the idea to describe code fragments with tags?



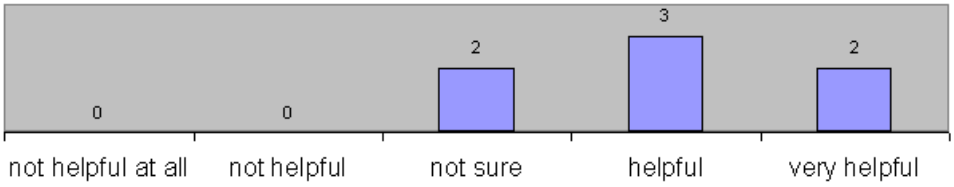
What do you think of coTag's content-based tag proposition?



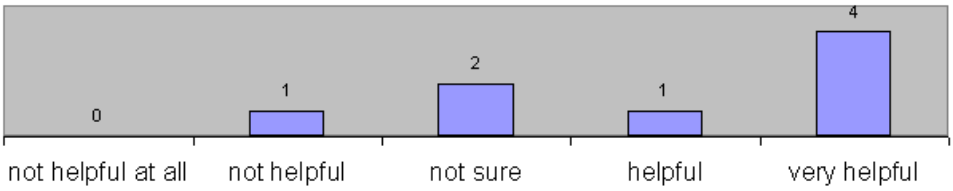
What do you think of coTag's proposition of already used tags?



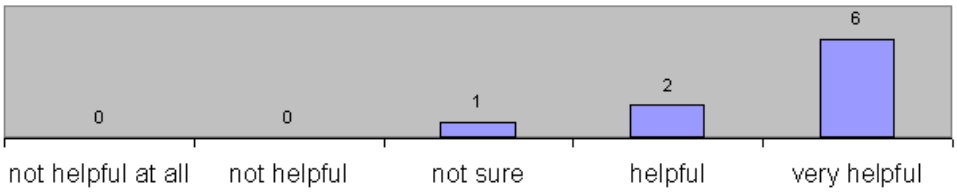
What do you think of coTag's similarity-based retrieval?



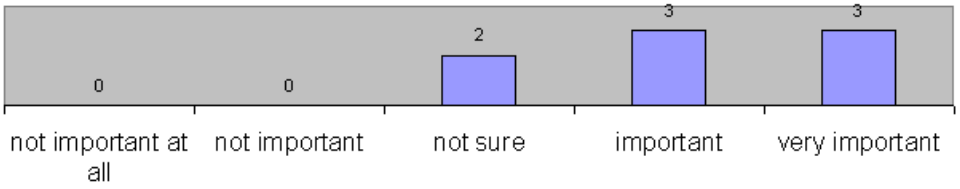
What do you think of similarity customisation?



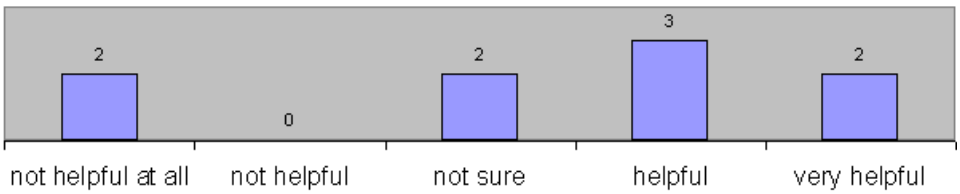
What do you think of similarity exchange?



Do you think, it would be important to comment tags?



Do you consider it helpful to rate snippets on a scale from 1 to 5?



Bibliography

AAMODT, AGNAR UND PLAZA, ENRIC. 1994. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, **7**(1), 39–59.

BAHLS, DANIEL. 2008. *Explanation Support for the Case-Based Reasoning Tool myCBR*. Project Thesis, University of Kaiserslautern.

BAHLS, DANIEL UND ROTH-BERGHOFER, THOMAS. 2007. Explanation Support for the Case-Based Reasoning Tool myCBR. *Seiten 1844–1845 aus: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence. July 22–26, 2007, Vancouver, British Columbia, Canada.* The AAAI Press, Menlo Park, California.

BURKHARD, HANS-DIETER UND RICHTER, MICHAEL M. 2001. On the notion of similarity in case based reasoning and fuzzy theory. 29–45.

DEY, ANIND K. UND ABOWD, G. D. 2000. Towards a better understanding of context and context-awareness. *Aus: Computer Human Interaction 2000 Workshop on the What, Who, Where, When, Why and How of Context-Awareness*.

GOLDER, SCOTT A. UND HUBERMAN, BERNARDO A. 2006. The Structure of Collaborative Tagging Systems. *Journal of Information Science*, **32**(2), 198–208.

GOMES, PAULO UND LEITÃO, ANDRÉ. 2006. A Tool for Management and Reuse of Software Design Knowledge. *Seiten 381–388 aus:* MOTTA, ENRICO, VAN HARMELEN, FRANK, UREN, VICTORIA UND SLEEMAN, DEREK (Hrsg.), *Managing Knowledge in a World of Networks 15th International Conference, EKAW 2006, Podebrady, Czech Republic, October 2-6, 2006.*, Band 4248. Springer Verlag.

LENZ, MARIO, BARTSCH-SPÖRL, BRIGITTE, BURKHARD, HANS-DIETER UND WESS, STEFAN (Hrsg.). 1998. *Case-Based Reasoning Technology: From Foundations to Applications*. Lecture Notes in Artificial Intelligence, Band LNAI 1400. Berlin: Springer-Verlag.

LIEBOWITZ, JAY UND BECKMAN, THOMAS J. 2000. *Knowledge Organizations: What Every Manager Should Know*. Boca Raton, FL, USA: CRC Press, Inc.

MANNING, C. UND SCHÜTZE, H. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press.

NKAMBOU, ROGER. 2004. Capitalizing Software Development Skills Using CBR: The CIAO-SI System. *Seiten 483–491 aus: Innovations in Applied Artificial Intelligence*. Lecture Notes in Computer Science, Band 3029. Springer Verlag.

RICHTER, MICHAEL M. 1995. *The Knowledge Contained in Similarity Measures*. Invited Talk at the First International Conference on Case-Based Reasoning, ICCBR'95, Sesimbra, Portugal.

RIVEST, R. L. 1992 (Apr). *The MD5 Message Digest Algorithm*. RFC 1321. `ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt`.

ROTH-BERGHOFER, THOMAS R. UND CASSENS, JÖRG. 2005. Mapping Goals and Kinds of Explanations to the Knowledge Containers of Case-Based Reasoning Systems. *Seiten 451–464 aus:* MUÑOZ-AVILA, HECTOR UND RICCI, FRANCESCO (Hrsg.), *Case-Based Reasoning Research and Development, 6th International Conference on Case-Based Reasoning, ICCBR 2005, Chicago, IL, USA, August 2005, Proceedings*. Lecture Notes in Artificial Intelligence LNAI, Nr. 3620. Heidelberg: Springer Verlag.

SALTON, GERARD, FOX, EDWARD A. UND WU, HARRY. 1983. Extended Boolean information retrieval. *Commun. ACM*, **26**(11), 1022–1036.

SAUERMANN, LEO, GRIMNES, GUNNAR UND ROTH-BERGHOFER, THOMAS. 2008. The Semantic Desktop as a foundation for PIM research. *Aus:* TEEVAN, JAIME UND JONES, WILLIAM (Hrsg.), *Proceedings of the Personal Information Management Workshop at the CHI 2008*.

SOMMERVILLE, IAN. 2004. *Software Engineering (7th Edition) (International Computer Science Series)*. Addison Wesley.

STOREY, MARGARET-ANNE, CHENG, LI-TE, BULL, IAN UND RIGBY, PETER. 2006. Shared waypoints and social tagging to support collaboration in software development. *Seiten 195–198 aus:* HINDS, PAMELA UND MARTIN, DAVID (Hrsg.), *CSCW '06: Proceedings of the 20th anniversary conference on Computer supported cooperative work*. New York, NY, USA: ACM. `http://doi.acm.org/10.1145/1180875.1180906`.

TINTAREV, NAVA UND MASTHOFF, JUDITH. 2007. A Survey of Explanations in Recommender Systems. *Seiten 801–810 aus: ICDE Workshops*. IEEE Computer Society.