Tracing Requirements and Source Code during Software Development

Alexander Delater, Barbara Paech
Institute of Computer Science
University of Heidelberg
Im Neuenheimer Feld 326, 69120 Heidelberg, Germany
{delater, paech}@informatik.uni-heidelberg.de

Nitesh Narayan
Institute of Computer Science
Technical University of Munich
Boltzmannstrasse 3, 85748 Garching, Germany
narayan@in.tum.de

***Abstract*—Traceability links between requirements and source code are often created after development. This reduces the possibilities for developers to use these traceability links during the development process. Additionally, existing approaches applied after development do not consider artifacts from project management, which are used for planning and organizing a project. These artifacts can serve as a mediator between requirements and source code. In contrast to these existing approaches, we present an approach that creates traceability links between requirements and source code as the development progresses by incorporating artifacts from project management. In this paper, we make two key contributions. First, a Traceability Information Model integrating requirements, source code and artifacts from project management. Second, an approach for the (semi-) automatic creation of traceability links between artifacts from the Traceability Information Model achieving traceability between requirements and source code during the development process. We identified a catalog of information needs of developers from literature regarding requirements, source code that realizes these requirements, and work done by co-workers implementing these requirements. The presented approach satisfies the information needs of the developers during the development process, while keeping the traceability links up-to-date.**

***Keywords*-traceability; requirements; source code; software development; information needs.**

I. Introduction

Traceability information supports the software development process in various ways, amongst others, program comprehension, change management, software maintenance, software reuse and prevention of misunderstandings [1]. Traceability between requirements and source code has been extensively researched in the past years and much progress has been made in this field. Because the manual creation of traceability links between requirements and source code is cumbersome, error-prone, time consuming and complex [2], a major focus in research is on (semi-) automatic approaches. Existing approaches use various techniques, e.g., information retrieval [3] [4], execution traces [5], static/dynamic analysis [6], subscription-based or rule-based link maintenance [7] or combinations of them [8]. However, all these approaches do not use artifacts from project management, but such artifacts, e.g., sprints and work items, are widely used in software development projects nowadays. Thus, the first key contribution of this paper is a Traceability Information Model (TIM) integrating requirements, source code and project management artifacts.

Traceability links between requirements and source code are often created after development [9] using the aforementioned approaches. This reduces the possibilities for developers not only to use their project knowledge to improve the quality of the traceability links, but also to use the traceability links during software development and maintenance. Therefore, we argue that traceability links between requirements and source code should also be created during the software development process and not only after development. Thus, the second key contribution of this paper is a (semi-) automatic approach for creating traceability links between artifacts from the TIM achieving traceability between requirements and source code during the development process.

Additionally, while creating traceability links between requirements and source code, the information needs of the developers during development should play a major role. The importance of such information needs is presented by Ko et al. [10] for collocated software development teams, and Sillito et al. [11] on questions raised during a program change task. We identified a catalog of information needs of developers from the contributions of Ko et al. and Sillito et al. regarding requirements, source code that realizes these requirements, and work done by co-workers implementing these requirements. The presented approach satisfies the information needs of the developers during development while keeping the traceability links up-to-date.

The paper is structured as follows: Section II provides background knowledge about a model unifying system development and project management that we built upon, and the subset of artifacts from this model that we focus on in this work. Section III defines a model for source code representation and introduces the TIM integrating artifacts from system development model, project management model and source code model. Section IV introduces an approach to (semi-) automatically create traceability links between artifacts in the TIM. Section V provides a fictional example project to highlight the benefits of the presented approach. Section VI introduces a catalog of information needs and how they are satisfied using the artifacts and relations from the TIM created for the example project. Section VII describes related work and Section VIII provides a discussion of the presented contributions. Finally, Section IX summarizes the contributions and discusses future work.

II. Background

This section provides background knowledge about a model unifying system development and project management, and the subset of artifacts from this model that we focus on in this work.

A. MUSE Model

In software development projects, two different types of models are used for abstraction: the *system model* and *project model* [12]. Artifacts from the system model describe the system under construction, such as requirements, components or design documents. Artifacts from the project model describe the on-going project, such as work items, developers, sprints or meetings. These two models have already been integrated within a model called MUSE: Management-based Unified Software Engineering [12].

While the MUSE model describes the system under development and its project management, it does not provide traceability to the source code. The MUSE model is implemented in the model-based CASE tool UNICASE [13], which is a plugin for the Eclipse integrated development environment (IDE) and is developed in an open source project [14]. For this work, we build upon the MUSE model and extend it with a new *code model* to support traceability to the source code. The code model is introduced in Section III.

B. Focus on Subset of Artifacts

The MUSE model supports a large amount of artifacts. Therefore, we focus on a subset of artifacts that are required by the information needs of the developers regarding requirements and co-workers implementing these requirements. From the system model, we focus on the artifacts of *feature* and *functional requirement* representing requirements at different levels of detail. A feature is an abstract description of a requirement, and it is detailed by one or more functional requirements. From the project model, we focus on the artifacts of *developers*, *work items* and *sprints*. Work items represent a unit of work and are the task descriptions used in software development projects (we use the term work item instead of *task* to avoid misunderstandings with the term task used in requirements engineering). They can describe work for new implementations and bug fixing. As they are the basis of the daily work, they are regularly kept up-to-date [15]. Developers are assigned to work items. Sprints are used to organize work items in work packages and they provide a time frame to realize the work items.

III. Modeling Code and Traceability Information

In this section, we define the representations of source code that extend the MUSE model by a new code model. Furthermore, we define a TIM integrating the artifacts we focus on from the MUSE model and the representations of source code from the code model.

A. Code Model

The code model contains file-based and change-based representations of source code. We chose to use these representations because they are widely used in software development projects and are independent of any programming language. This is supported by a comprehensive literature survey by Kagdi et al. [16]. For file-based representations, we focus on *code files* containing source code. For change-based representations that are supported by a version control system (VCS), we focus on *revisions*. Revisions themselves contain changed code files. Other representations would be possible, e.g., class, method or interface. However, not all programming languages support these artifacts, reducing the applicability of the code model. Table I shows the different representations of source code and their attributes.

Table I
Attributes of Representations of Source Code

Type	Attributes
Code File	fileName, projectName, pathInProject
Revision	date, author, number, repositoryUrl, pathInRepository, commitMessage, changedCodeFiles [added, modified, or deleted]

In the following, we describe the reasons for choosing these attributes. For code files, we require the attributes *fileName*, *projectName* and *pathInProject* to locate them in a project. For revisions, we require the attributes *date* and *author* to tell when and by whom the revision was created. We also need the attributes *number*, *repositoryUrl* and *pathInRepository* to reliably locate the revision in a VCS. Moreover, the attribute *commitMessage* is required to describe the changes contained in this new revision. This comment is usually written by the author of the revision and is optional. The most important information of a revision is stored in the list *changedCodeFiles* and each artifact in this list has the same attributes as the artifact code file. Moreover, each changed code file in the revision has a *state* [added, modified, or deleted] that shows if the code file was newly added, existed before and was only modified or was deleted in the revision.

B. Traceability Information Model

Traceability in a project should be documented in and driven by a Traceability Information Model [17]. A basic TIM consists of two types of entities: traceable artifacts and traceability links between these artifacts. It also defines which types of artifacts are intended to be traced to which related artifact types and by what type of traceability link. We define a TIM that realizes the artifacts from the MUSE model that we focus on and extends it by the two representations of source code. The TIM (see Figure 1) shows all artifacts that we want to connect: system model artifacts (feature, functional requirement), project model artifacts (sprint, work item, developer) and code model artifacts (code file, revision).

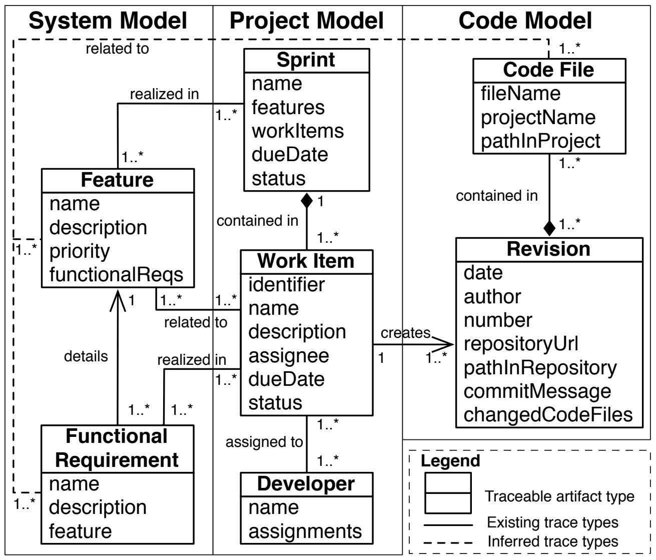

Figure 1. Traceability Information Model integrating system model, project model and code model

Figure 1 depicts the core traceable components. A feature is realized in a sprint and is detailed in one or more functional requirements. Functional requirements are realized by work items. A work item must have one or more linked functional requirements. A feature can be related to a work item, e.g. during bug fixing. Work items are contained in a sprint and are assigned to developers. One work item can create one or more revisions. A revision contains one or more changed code files. All these traceability links between the artifacts are represented as straight lines in Figure 1. All artifacts from the TIM can be found in common software development projects.

The central artifact is the work item, as it connects the artifacts from the system model to artifacts from the code model. Using work items, we can achieve traceability between requirements and source code by inferring traceability links. An inferred traceability link between two artifacts is derived from all artifacts in between these two artifacts. For example, a functional requirement is realized in one or more work items, and a work item creates one or more revisions containing code files. Thus, we can infer traceability links between the functional requirement and the code files. The inference process and the used algorithm is explained in detail in Section IV. The inferred traceability links are represented as dashed lines between features, functional requirements and code files in Figure 1.

IV. Tracing Requirements and Source Code during Software Development

This section presents a (semi-) automatic approach for creating traceability links between artifacts from the TIM during the software development process, especially between requirements and source code using work items.

We presume that the following situation is present in the software development project. First, there exists a list of features and detailing functional requirements. Second, a project manager has planned the realization of the features in sprints and s/he has broken down the realization of the functional requirements into work items for the developers in the software development project. Third, the work items are already assigned to developers, e.g. manually by the project manager or using an approach by Helming et al. for semi-automatic assignment of work items [15]. For the presented approach, we assume that all artifacts from the TIM are available in one integrated environment supporting traceability links between all artifacts. Such an integrated environment can be supported by the model-based CASE tool UNICASE [13].

A. Capturing Traceability Links

Figure 2 depicts the process of capturing traceability links and every activity is described in detail in the following. The core idea of creating traceability links between artifacts of the TIM is letting the developers create these links themselves. First, the developer selects a work item from his/her list of assigned work items and tells the system that s/he starts implementing source code. While working on the work item, all features or functional requirements the developer looks at during implementation are automatically captured by the system, meaning that the system logs these types of artifacts while a developer opens them during implementation. The developer can look at the linked features or functional requirements of the work item or look at other artifacts of these types to get a better understanding during implementation. After finishing the implementation of a work item in the source code, the developer tells the system that s/he has stopped implementation.

The developer does not immediately commit the changes to the VCS. Instead, before the commit, s/he has to validate two lists of artifacts: one list of all changed code files in the source code, and another list of all captured features or functional requirements that s/he looked at during implementation. While the former is standard in software development and already supported by any VCS, the latter represents additional work for the developers. This validation is necessary to only create relevant traceability links between the work

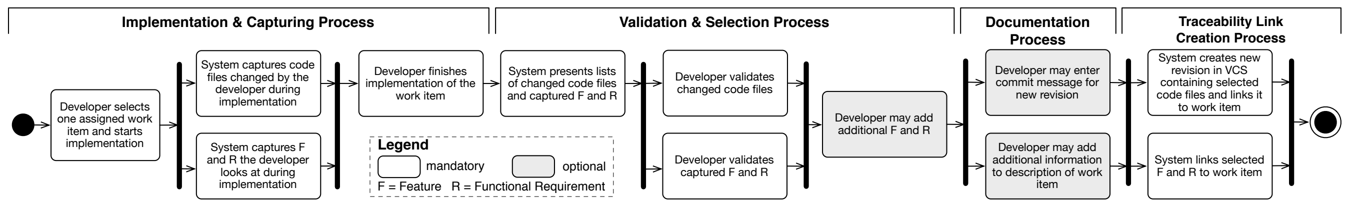
Implementation & Capturing Process
Validation & Selection Process
Documentation Process
Traceability Link Creation Process
Developer selects one assigned work item and starts implementation
System captures code files changed by the developer during implementation
System captures F and R the developer looks at during implementation
Developer finishes implementation of the work item
System presents lists of changed code files and captured F and R
Developer validates changed code files
Developer validates captured F and R
Developer may add additional F and R
Developer may enter commit message for new revision
Developer may add additional information to description of work item
System creates new revision in VCS containing selected code files and links it to work item
System links selected F and R to work item
Legend
mandatory
optional
F = Feature R = Functional Requirement

Figure 2. Process of Capturing, Validating and Creating Traceability Links (UML Activity Diagram)

item and these artifacts. For example, a developer can look at a functional requirement during development, which is not directly involved in the implementation, but related to the work item. During validation, a developer removes unrelated artifacts from the list. Furthermore, an optional activity for the developer is to select additional features or functional requirements that are related to the work item, but that s/he has not had a look at during implementation. Two other optional activities are to enter a commit message for the new revisions or add additional information to the description of the work item.

After validating all artifacts and optionally adding further features or functional requirements, the developer selects to commit all information to the VCS. The system then creates a new revision containing only the selected code files. The work item is linked to the newly created revision, and the attribute *identifier* of the work item is inserted at the end of the commit message of the revision to achieve bi-directional traceability between work item and revision. Moreover, the system links all validated and selected features and functional requirements to the work item.

B. Inferring Traceability Links between Requirements and Source Code

The created traceability links are used to infer links between requirements and code, specifically between features, functional requirements and code files. The Algorithm IV.1 for creating inferred traceability links is executed when the status of a work item is changed by the developer from *assigned* to *done*. The algorithm connects in a brute force manner all linked requirements (features, functional requirements) of a work item with all the code files in the linked revisions of the work item. The statement *workItem.getLinkedRequirements()* returns all features and functional requirements. The statement *workItem.getRevisions()* returns the linked revisions of a work item sorted by attribute *date* in ascending order. This is important because the algorithm applies change operations to the artifacts which need to be in the order they occurred. If the algorithm identifies already existing links, it does not create them again. If a code file was modified, its information consisting of the attributes *fileName*, *projectName* and *pathInProject* is updated for each linked requirement. If a code file is deleted, the link to the requirement is removed.

Algorithm IV.1: INFERTRACES($workItem$)

```
$allReqs = workItem.getLinkedRequirements()$
$allRevisions = workItem.getRevisions()$
for each $rev \in allRevisions$
  $allCodeFiles = rev.getAllChangedCodeFiles()$
    for each $cf \in allCodeFiles$
      $state = cf.getState()$
        if $state = ADDED$
          for each $req \in allReqs$
            if $req.isNotConnectedTo(cf)$
              $req.addLinkTo(cf)$
        if $state = MODIFIED$
          for each $req \in allReqs$
            if $req.isNotConnectedTo(cf)$
              $req.addLinkTo(cf)$
            $req.getLinkedCodeFile(cf)$
                $.update(fileName, projectName$
                $pathInProject)$
        if $state = DELETED$
          for each $req \in allReqs$
            $req.removeLinkTo(cf)$
return $(workItem)$
```

V. EXAMPLE

We use a fictional example project to highlight the benefits of the presented approach and to support discussion. The example project is a Java application called *Movie Manager* that one can use to manage his/her movie collection. Users can add, modify and delete movies as well as rate them. The application supports importing data about performers (actor/actress) of a movie from an Internet movie database. Presenting all information about the artifacts in the project is beyond the purpose of this paper. Therefore, we only provide a list of used artifacts with short descriptions to support basic understanding. There are two features (F) and six detailing functional requirements (R) (see Table III). The project is planned in two sprints with feature F1 developed in Sprint 1 and feature F2 developed in Sprint 2. Amy, Bill and Carl are members of a team collaborating to develop the application and they have eight work items (W) (see Table IV). Amy is mainly focusing on the data objects within the application, Bill is responsible for the user interface, and Carl is doing bug fixing.

Table II
CHANGED CODE FILES AND CAPTURED TRACEABILITY LINKS OVER TEN REVISIONS OF MOVIE MANAGER

C1	C2	C3	C4	C5	R1	R2	R3	R4	R5	R6	F1	F2	Work Item	Dev.	Rev.	Commit Message
c					LB						CL		W1	Amy	1	Created Data Object 'Movie' #W1
m	c				CL	LB	LB				CL		W2	Bill	2	Implemented basic UI for listing Movies #W2
	m					CL	LB				CL		W2	Bill	3	Display and change information of Movie #W2
	m	c						LB					W3	Bill	4	Added basic Rating Control and added it to Movies UI #W3
		m						LB			CL		W3	Bill	5	Completed 5 star Rating Control #W3
			c							LB		CL	W4	Amy	6	Created Data Object 'Performer' #W4
	m		m							LB			W5	Bill	7	Display information of Performers for Movie, currently showing dummy data as import needs to be implemented #W5
		m						LB			CL		W6	Carl	8	BugFix for Rating Control #W6
				c					LB			CL	W7	Amy	9	Performer Import #W7
				m					LB			CL	W8	Carl	10	Bugfix for Performer Import #W8

c = created m = modified LB = Linked Before CL = Captured Link

Table III
FEATURES AND FUNCTIONAL REQUIREMENTS

Arti-fact	Description	Detailing
F1	Movie Management: Add, modify and delete movies as well as rate them	-
F2	Performer Management: Import performers from Internet movie database	-
R1	Users should be able to add and remove a movie from the list	F1
R2	Users should be able to display and change the textual information about a selected movie	F1
R3	Users should be able to display a list of available movies and select one from the list	F1
R4	Users should be able to rate movies	F1
R5	Users should be able to import textual information about the performers of a movie from Internet movie database	F2
R6	Users should be able to display textual information about the performers of a movie	F2

F = Feature R = Functional Requirement

Table IV
DEVELOPERS AND WORK ITEMS

Arti-fact	Description	Assigned To
Amy	Database Expert	W1 W4 W7
Bill	UI Expert	W2 W3 W5
Carl	Bug Fixing	W6 W8

Arti-fact	Description	Realizing	Sprint
W1	Create Data Object for Movie	R1	S1
W2	UI for Movies	R2 R3	S1
W3	UI Control for Rating	R4	S1
W4	Create Data Object Performer	R6	S2
W5	UI for Performers	R6	S2
W6	Bugfix for Rating Control	R4	S2
W7	Performer Import	R5	S2
W8	Bugfix for Performer Import	R5	S2

W = Work Item R = Functional Requirement

A number of code files are developed to achieve Movie Manager: Movie.java (C1), MoviesUI.java (C2), RatingControl.java (C3), Performer.java (C4), PerformerImport.java (C5). Table II provides an overview about the ten created revisions, created (c) and modified (m) code files, used traceability links (LB) from the TIM and captured traceability links (CL) during the software development (in the small example, there are no code files that needed to be deleted). Furthermore, all three developers have entered commit messages for each revision that roughly describe how they have modified the source code.

The team used the presented process (see Figure 2) during development. In the following, the creation of revisions 1 and 2 is shortly explained. All other revisions were created in the same way. Work item W1 was assigned to Amy and she had to implement the data object for storing movies. First, she looked at the linked functional requirement R1 of her work item to get a better understanding of the attributes of the data object. She started implementing Movie.java (C1) and looked at F1 for the feature description. She finished implementation and validated and confirmed all captured links to F1 and R1. Next, she entered a commit message and the system created a new revision with the new code file C1.

Work item W2 was assigned to Bill and he was supposed to implement a user interface for listing the movies. Thus, he first looked at the linked functional requirements R2 and R3. Bill looked during implementation at feature F1 because it was already linked to R3. Furthermore, he looked at R1 because this requirement was also linked to F1. Bill used

Table V
INFORMATION NEEDS ON REQUIREMENTS DURING DEVELOPMENT ACTIVITIES WITH USED TRACEABILITY LINKS

Nr.	Information Need on Requirement	Development Activity	Used Traceability Links
1.	What is the program supposed to do?	Implementation, Program Comprehension	F-R, F-W, R-W, F-C*, R-C*
2.	Why was this code implemented this way?	Program Comprehension	C-Rev
3.	What have my co-workers been doing?	Change Awareness	F-W, R-W, W-D, W-S
4.	Which code is involved in the implementation of this feature?	Maintenance	F-C*, R-C*
5.	To move this feature into this code, what else needs to be moved?	Change Management	F-C*, R-C*
6.	What will be the impact of this change?	Change Management	F-R, F-W, R-W, F-C*, R-C*

F = Feature R = Functional Requirement W = Work Item S = Sprint D = Developer C = Code File Rev = Revision * = inferred

the inferred traceability link from F1 to Movie.java (C1) to change Movie.java because it missed an attribute that Amy forgot to implement, and created the code file MovieUI.java (C2). The inferred traceability link was created after Amy changed the status of her work item from *assigned* to *done*. He finished implementation and validated and confirmed all captured traceability links to R1, R2, R3 and F1. Finally, he entered a commit message and the system created a new revision with new code file C2 and modified code file C1.

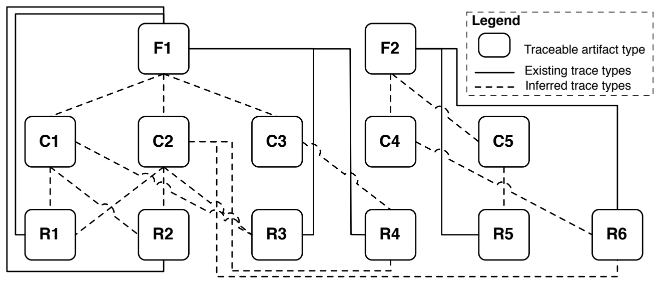

Figure 3. Existing and Inferred Traceability Links

After the completion of each work item, traceability links were inferred using the presented algorithm (see Algorithm IV.1). In revision 10, this resulted in the traceability links between features, functional requirements and code files shown in Figure 3. The straight lines show the traceability links that existed before. Furthermore, the inferred traceability links are shown as dashed lines.

VI. INFORMATION NEEDS ON REQUIREMENTS

Developers have various information needs during the software development process. Ko et al. [10] identified 21 and Sillito et al. [11] identified 44 information needs, respectively. From these 65 information needs represented as questions, we have identified those which are asked by developers during software development focusing on requirements, code that implements these requirements, and work done by co-workers related to these requirements. We looked through all information needs and used the following criteria for identification: a) mentioning terms that are related to requirements, e.g., *feature*, *concern*, *behavior* or expressions like *supposed to*, b) mentioning the term *impact* in conjunction with a changing requirement, and c) mentioning terms like *developer* or *co-worker*. We have identified six information needs (see Table V, Nr. 1-3 from Ko et al. and Nr. 4-6 from Sillito et al.) that met these criteria. All other information needs are rather specific for implementation and do not focus on requirements, e.g. reproducing a failure during bug fixing or understanding execution behaviour. We defined in Table V for each information need on requirements during what development activity the information need occurs and the used traceability links.

A. Identified Information Needs on Requirements

In the following, we explain how these information needs of developers can be satisfied by employing the TIM (see Figure 1), the captured traceability links from the process (see Figure 2) and the inferred traceability links (see Algorithm IV.1) for the example project mentioned in Section V.

1) What is the program supposed to do?: The features and functional requirements define what the program is supposed to do. As a work item needs to have a relation to functional requirements and can be related to features, an assigned developer can use the linked artifacts during implementation and program comprehension. For example, Amy knows during implementation what attributes the data object for movies requires since the functional requirement R1 is linked to her work item W1. However, she forgot to implement one attribute in revision 1; so, Bill had to change the data object again in revision 2. Furthermore, if a developer is interested in the purpose of a code file during program comprehension, s/he can use the inferred traceability links from the code file to the features and functional requirements. For example, if Carl is interested in the purpose of C3 (RatingControl.java), he can use the inferred traceability links to F1 and R4 that were created when Bill finished the work item W3 in revision 5.

2) Why was this code implemented this way?: Starting from the code files, a developer can look at the linked revisions. The commit messages may contain information concerning why the code was implemented this way. For example, Bill decided to implement the Rating Control with a 5 star rating and documented his decision in the commit message of revision 5. Documenting these decisions as artifacts of type rationale would be part of future work.

3) What have my co-workers been doing?: Since all work items are contained in a sprint and assigned to developers, a developer is able to see on what features or functional requirements his/her co-workers will be working on or have been working on in the past, which is supporting change awareness. Furthermore, a developer is able to see the co-workers that have previously worked on the same feature or functional requirement. Using this information, s/he can seek further knowledge from these co-workers. For example, Carl can see that Bill has worked previously on the Rating Control and he can ask him for advice during bug fixing.

4) Which code is involved in the implementation of this feature?: A feature is detailed in functional requirements. A developer can use the inferred traceability links from features and functional requirements to code files to quickly identify code that is involved in the implementation of a feature. For example, Carl can see that code files C4 (Performer.java) and C5 (PerformerImport.java) are involved in feature F2 (Performer Management) during bug fixing described in W8. This enables to identify not realized features and functional requirements as well as the progress of their implementation.

5) To move this feature into this code, what else needs to be moved?: 'Moving a feature' means that an entire feature with all its detailing functional requirements and realizing code can be moved from one development project to another project. As one feature is connected to detailing functional requirements, and these artifacts are connected by inferred traceability links to code files, related code files can be identified during change management. For example, the code files C1, C2 and C3 are related to feature F1 through their relations to the requirements R1, R2, R3 and R4 (see Figure 3). Therefore, if a feature needs to be moved, all its related functional requirements and the realizing code files can be easily identified. However, this may require additional code files to be moved that are required by the to-be-moved code files. Additional work on integrating the moved code files in the new environment may be necessary, as well.

6) What will be the impact of this change?: If a feature or a functional requirement need to be changed to reflect changed customer demands, all related artifacts maybe affected by this change can be identified easily during change management. For example, suppose R4 is changed to support a different rating, it can be identified that W3, W6 and C3 are maybe affected by this change. Affected work items can be identified, e.g. if a change in a feature or functional requirement is comprehensive, the planning of the realization in the work items needs to be adapted. An initial set of code files can be identified potentially affected by this change. The changes in the code files can result in additional changes in other code files. The initial set of code files can be a starting point for detailed change impact analysis.

B. Frequently Unsatisfied Information Needs

Many of the frequent information needs are problematic, because the searches for this information are often unsatisfied and have long search times. It is of particular interest that the most difficult information needs to satisfy are questions regarding requirements and co-workers working on these requirements [10]. Ko et al. have identified seven most frequently unsatisfied information needs, from which three are exactly the same information needs 1-3 from Table V that met our criteria. For example, searches for the information need *1. Why was the code implemented this way?* resulted in 44% of unsatisfied searches and a maximum of 21 minutes of observed search time.

One of the most frequently sought and acquired information by a developer includes what co-workers have been doing, which corresponds to the information need *3. What have my co-workers been doing?*. To determine who to ask, developers often identify co-workers by inspecting commit logs, but such information is not always accurate [10]. Our approach helps developers determining co-workers who have worked on the same requirements as themselves in the past to seek further information.

VII. Related Work

Approaches related to our work can be divided into two groups: approaches achieving traceability between requirements and source code after development and approaches capturing traceability links as we do during development.

A. Traceability between Requirements and Source Code

In [18], a general overview about requirements traceability is provided. As the manual creation of traceability links between requirements and source code is error-prone, time consuming and complex [2], research focuses on (semi-) automatic approaches. Existing approaches create traceability links between requirements and source code using various techniques, e.g., information retrieval [3], [4], [19], [20], [21], execution-trace analysis [5], [22], [23], static/dynamic analysis [6], subscription-based or rule-based link maintenance [7] or combinations of them [8], or only create links between work items and code [24]. However, no approach uses artifacts from project management to create traceability links between requirements and source code, as we do with our approach using work items.

B. Capturing Traceability Links

An approach similar to ours for the automatic capturing of links was presented by Omoronyia et al. [25]. They have achieved traceability between use cases and source code. In contrast, our approach supports features and functional requirements. Their approach is based on tracing the operations carried out by a developer called navigation trails. However, this approach requires an elaborate model with rankings of navigation trails to derive the most relevant links. Rankings of links are currently not supported by our approach. Thus, in future work we want to analyze whether the availability of work items can support this ranking.

Their approach is also able to identify which developer is involved in the realization of a specific use case, which is also supported by our approach. The contribution of Omoronyia et al. shows that tracking changes displays some advantages over the other approaches. For example, relating a developer to the source code and requirements is almost impossible with the other approaches, but very easy if changes/operations are tracked, like in our approach. Furthermore, their approach does not support work items from project management and revisions in a VCS. However, such artifacts are widely used in software development projects nowadays. Therefore, our approach is more easily applicable in practice compared to the approach by Omoronyia et al.

Omoronyia et al. [25] also claim to satisfy certain information needs. However, they did not use a structured method to identify these information needs like we did and only proposed those that were satisfied by their approach. Furthermore, their information needs are not based on project management and co-workers within the project. The benefit of our approach is that we can satisfy these information needs of developers during the development process.

VIII. Discussion

Egyed et al. [26] investigated the effort of recovering traceability links between requirements and code after development. In general, these traceability links were recovered by project members who were not directly involved in the realization of a particular requirement, but knew the code base. Our approach distributes the effort of creating traceability links over all developers actively participating in the project while they perform their implementation work. Using our approach, the developers are now involved in the traceability process, they can use their expertise and project knowledge to create reliable traceability links and these links also help them to satisfy their information needs during development. As a developer benefits not only from these traceability links himself/herself, but also his/her co-workers, we expect that they are better motivated to create and validate traceability links during software development.

Additionally, one might ask: ”Why is (manually) creating links between requirements and work items, and between work items and code files less complex compared to existing work on linking requirements to source code directly?”. We argue that our approach is less cumbersome and error-prone than manually creating direct links between requirements and code, because the only manual work is to establish initial links between work items and requirements (which is typical for issue management) and to validate the automatically captured links (which should be easy as the links refer to the work just finished). Creating direct links manually requires the developer to keep every relationship in mind.

In the current approach, developers might make mistakes when adding non-related features or functional requirements to a work item. However, this risk is reduced since we let the developer validate all traceability links before they are created. It has been shown that humans were better at validating links as opposed to searching for missing links [27]. This strengthens our approach of letting the developers validate the links going to be created instead of recovering links or searching for missing links. The additional work of the developers introduced by validating traceability links and manually adding additional ones is considered as small, compared to the effort to establish traceability links after development using various approaches mentioned before.

Currently we are developing tool support based on UNICASE, which is a plugin for the Eclipse IDE. The Eclipse IDE supports various programming languages through additional plugins, e.g. Java, C++, Python etc. By integrating UNICASE and Eclipse with plugins for VCSs like Subversion or Git, a comprehensive tool environment can be provided supporting developers while they perform various development activities. By using these plugins, file-based as well as change-based representations of source code can be accessed. We looked at various research tools, e.g., TagSEA [28], and commercial tools, e.g., IBM Rational Team Concert [29]. Some of these tools do support all the elements that we have (requirements, work items, code). However, our tool would provide, unlike all other tools, complete traceability between all these elements as well as (semi-) automatic linkage of requirements and code.

IX. Conclusion and Future Work

In this paper, we presented an approach for tracing requirements and source code during software development to satisfy information needs of developers regarding requirements during development. We defined a TIM that integrates requirements, source code and artifacts from project management. We also presented an approach for the (semi-) automatic creation of links between artifacts from the TIM.

In this work, we only focused on information needs of developers. However, we are aware that also information needs of other project participants can be satisfied with the created links, e.g., of project managers or requirements engineers, which is subject to future work. Furthermore, we are aware that the algorithm for inferring links is very basic and might create a lot of links. Therefore, we will investigate possibilities for more advanced inference algorithms, e.g. an algorithm providing a relevance ranking for each link based on the change history of the artifacts connected by the link, to identify relevant links from the large set of inferred links. Currently we develop tool support based on our approach. Once the tool is finished, we will empirically evaluate the approach in the UNICASE project itself and apply it in various case studies. We will compare our approach to existing baseline approaches w.r.t. precision and recall.

REFERENCES

[1] Egyed, A. and Grünbacher, P. Supporting software understanding with automated requirements traceability. International Journal of Software Engineering and Knowledge Engineering, vol. 15, no. 5, pp. 783-810 (2005)

[2] Spanoudakis, G. and Zisman, A. Software traceability: A roadmap. Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing, pp. 395-428 (2004)

[3] Hayes, J.H., Dekhtyar, A., and Osborne, J. Improving requirements tracing via information retrieval. International Conference on Requirements Engineering, pp. 138-147 (2003)

[4] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G. Recovering traceability links in software artifact management systems using information retrieval methods. Transactions on Software Engineering Methodology, vol. 16, no. 4, art. 13, ACM (2007)

[5] Eisenberg, A.D. and De Volder, K. Dynamic feature traces: Finding features in unfamiliar code. In ICSM 05: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 337-346 (2005)

[6] Antoniol, G. and Gueheneuc, Y.G. Feature identification: A novel approach and a case study. In ICSM 05: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 357-366 (2005)

[7] Maeder, P. and Gotel, O. Towards Automated Traceability Maintenance. Journal of Systems and Software, vol. 85, no. 10, pp. 2205-2227 (2011)

[8] Eaddy, M., Aho, A.V., Antoniol G., et al. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In ICPC 08: Proceedings of the 16th IEEE International Conference on Program Comprehension, pp. 53-62 (2008)

[9] Cleland-Huang, J., Heimdahl, M., Huffman Hayes, J., Lutz, R., and Maeder, P. Trace queries for safety requirements in high assurance systems. In REFSQ 12: Proceedings of the 18th International Conference on Requirements Engineering: Foundation for Software Quality, pp. 179-193 (2012)

[10] Ko, A.J., DeLine, R., and Venolia, G. Information needs in collocated software development teams. In ICSE 07: Proceedings of the 29th International Conference on Software Engineering, pp. 344-353 (2007)

[11] Sillito, J., Murphy, G.C., and Volder, K.D. Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng., vol. 34, no. 4, pp. 434-451 (2008)

[12] Helming, J., Koegel, M., and Naughton, H. Towards traceability from project management to system models. In TEFSE 09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, pp. 11-15. IEEE Computer Society (2009)

[13] Bruegge, B., Creighton, O., Helming, J., and Koegel, M. Unicase - an Ecosystem for Unified Software, In ICGSE 08: Distributed software development: methods and tools for risk management, pp. 12-17 (2008)

[14] UNICASE Open Source Project. http://www.unicase.org/ [retrieved: September, 2012]

[15] Helming, J., Arndt, H., Hodaie, Z., Koegel, M., and Narayan, N. Automatic Assignment of Work Items. In ENASE 10: Evaluation of Novel Approaches to Software Engineering, Communications in Computer and Information Science, vol. 230, pp. 236-250 (2011)

[16] Kagdi, H., Collard, M.L., and Maletic, J.I. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution, Journal of Software Maintenance and Evolution, vol. 19, pp. 77-131 (2007)

[17] Maeder, P., Gotel, O., and Philippow, I. Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice. In TEFSE 09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, pp. 21-25. IEEE Computer Society (2009)

[18] Dahlstedt, A. and Persson, A. Requirements interdependencies: State of the art and future challenges. In Engineering and Managing Software Requirements, Aurum and Wohlin (eds.) Springer, pp. 95-116 (2005)

[19] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering, pp. 970-983 (2002)

[20] Marcus, A. and Maletic, J.I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In ICSE 03: Proceedings of the 25th International Conference on Software Engineering, pp. 125-135. IEEE Computer Society (2003)

[21] Marcus, A., Maletic, J.I., and Sergeyev, A. Recovery of traceability links between software documentation and source code. International Journal of Software Engineering and Knowledge Engineering, vol. 15, no. 5, pp. 811-836 (2005)

[22] Egyed, A. A Scenario-Driven Approach to Trace Dependency Analysis. Transactions on Software Engineering, vol. 29, no. 2, pp. 116-132, IEEE (2003)

[23] Burgstaller, B. and Egyed, A. Understanding where requirements are implemented. In ICSM 10: Proceedings of the 26th IEEE International Conference on Software Maintenance, pp. 1-5 (2010)

[24] Anvik, J. and Storey, M.A. Task articulation in software maintenance: Integrating source code annotations with an issue tracking system. In ICSM 08: Proceedings of the 24th IEEE International Conference on Software Maintenance, pp. 460-461 (2008)

[25] Omoronyia, I., Sindre, G., Roper M., Ferguson J., and Wood, M. Use case to source code traceability: The developer navigation viewpoint. In RE 09: Proceedings of the 17th IEEE International Requirements Engineering Conference, pp. 237-242 (2009)

[26] Egyed, A., Graf, F., and Grünbacher, P. Effort and quality of recovering requirements-to-code traces: Two exploratory experiments. In RE 10: Proceedings of the 18th International IEEE Requirements Engineering Conference, pp. 221-230 (2010)

[27] Kong, W.-K., Huffman Hayes, J., Dekhtyar, A., and Holden, J. How do we trace requirements: an initial study of analyst behavior in trace validation tasks. In Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering, In conjunction with CHASE 11, pp. 32-39 (2011)

[28] TagSEA. http://tagsea.sourceforge.net/ [retrieved: September, 2012]

[29] IBM Rational Team Concert. http://www.ibm.com/software/rational/products/rtc/ [retrieved: September, 2012]