Leveraging Traceability between Code and Tasks for Code Review and Release Management

Nitesh Narayan, Jan Finis, Yang Li
Institute of Computer Science
Technical University of Munich
Boltzmannstrasse 3, 85748 Garching, Germany
{narayan, finis, liya}@in.tum.de

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

***Abstract*—The software maintenance process relies on traceability information captured throughout the development of a software product. Traceability from code to software engineering artifacts like features or requirements has been extensively researched. In this paper, we focus on traceability links between code and tasks. Tasks can be further linked to other artifacts such as features or requirements. In this paper, we present an approach for (semi-) automatic creation of traceability links between code and tasks. The core idea is to let the developers create the links themselves while they use a version control system. We use these traceability links to improve the processes of code review and release management. A prototype based on this work has been implemented and integrated into the model-based CASE tool UNICASE. We applied the developed prototype in the open-source project UNICASE itself and report about our significant experiences.**

Keywords*-*traceability; code review; release management; patch; branch.

I. Introduction

Software configuration management (SCM) is the discipline of managing the evolution of large and complex software systems to assist software development and maintenance processes [1]. According to the IEEE standard [2], SCM covers several activities such as identification of product components and their versions, audit, review, as well as change control (by establishing procedures to be followed when performing a change). Practicing SCM in a software project has several benefits, including increased productivity, better project control, identification and fixes of bugs, and improved customer satisfaction [3]. Especially in projects with increased complexity, efficient handling of SCM requires tool support. Standard SCM tools exist for various activities e.g. version control systems (VCS). However, other SCM activities still lack proper tool support because of the involved traceability challenges, especially the review of changes during a code review and the building of a software product during release management. In this paper, we present an approach for (semi-) automatic creation of traceability links between code and tasks to improve the processes of code review and release management by providing tool support. Tasks represent a unit of work, which describe changes to be performed to the code or new developments and they are used in many software development projects. In the remainder of the paper, we use the term *work item* instead of task to avoid misunderstandings with the term task used in requirements engineering.

The paper is structured as follows: in Section II, we provide background information. In Section III, we describe the processes of code review and release management and benefits from using traceability links between code and work items. The approach is presented in Section IV and the prototype implementation is shown in Section V. In Section VI, we describe our experiences in using the prototype in the open-source project UNICASE. Related work is presented in Section VII and a discussion and future work conclude the paper in Section VIII and Section IX, respectively.

II. Background

Traceability links between requirements and work items have previously been researched in the MUSE model (Management-based Unified Software Engineering) [4], which integrates the system model and the project model. The system model describes the system under construction, such as requirements, features, use cases or UML artifacts. The project model describes the on-going project, such as work items, the organizational structure, sprints or meetings. The MUSE model is implemented in the model-based CASE tool UNICASE [5] [6], which we use to implement our approach.

The presented approach in this paper is dealing with the (semi-) automatic creation of traceability links between work items and code. Previous studies have shown that links between system elements and project elements provide useful information for the work (by shortening the navigation paths of the developers) and that based on such links system elements are kept more up-to-date [7]. Thus, we are extending the MUSE model by introducing traceability to code. Various development activities can benefit from the traceability links between requirements, work items and code. In this work, we concentrate on code review and release management activities.

III. Code Review and Release Management

This section explains the processes of code review and release management. We discuss how these two processes could be improved by using traceability links between requirements, work items and code. Furthermore, we specify requirements for an approach improving these two processes.

A. Code Review

Code review is a process where a team member reviews code written by another member to ensure quality and consistency, as well as to share knowledge in a software development project. It assists in improving the quality by identifying defects at an early stage. However, code review by its nature is a labour-intensive process. This is further affected by the lack of effective tool support. The problem is not overwhelming tool complexity, as the goals of such a tool are rather simple. Goals of such a tool are:

- Enable the reviewer to quickly transfer the changes to be reviewed to his local workspace as well as highlight the changes so that s/he can review the interplay of the changes with the entire code base.
- After the code review, it should allow the reviewer to add a review summary to the associated work item.

A specific question during requirements validation is to ensure that the implementation of every requirement or feature is reviewed. Thus, one needs to be able to associate changes in the code to its corresponding work item in the project management documents. Therefore, work items themselves are associated to a requirement or feature as well as to the changes in the code. This requires extended tool support to aid the code review process.

B. Release Management

Another important activity in SCM is the release management process. This process involves deciding on which configuration a product is released and which features it includes. During the release management it needs to be verified that the code, from which the release is actually built, includes all features or requirements the release should embody.

Most of the existing literature fails to highlight this activity and its importance in release management. Van der Hoek et al. [8] even describe the release management as a “poorly understood and underdeveloped part of the software process”. Instead, it is generally assumed that a configuration of code already exists and that it contains all required content. Thus, two important aspects are:

- Is the implementation of each feature or requirement, which are part of this release, included in the code? Is the implementation not finished, or finished but only stored in the local workspace of the implementor?
- Are there any other changes in the code e.g. undocumented bug fixes or accidentally introduced changes?

If it can be validated which features or requirements are included in the code of a release, the release management process can further benefit from extended functionalities provided by the tool. For example, assembling the code for the release autonomously. By specifying the base version of code and a set of features or requirements to be included into the release, the system should be able to merge the implementation of all features or requirements into the code, ignoring already included features or requirements.

Other activities can also benefit from tracking the changes in the release. During release management, there is a need to capture a list of features and included bug fixes in the form of a change log for every release. This log is usually shipped with the product and/or published to inform users about changes. However, the change log could be assembled automatically as it is possible to identify the list of features or bug fixes implemented in the code.

C. Leveraging Traceability between Code and Work Items

For the purpose of improving the processes of code review and release management, we capture all changes made to the code in a so-called *change package*. The change package is *created* by accumulating the changes a developer performs in the code within the context of a certain work item. Hence, every change package is associated with only one work item.

During a code review, the reviewer needs to quickly transfer all changes to be reviewed to his/her local workspace. Suppose the reviewer has to review some changes in the code. For this task, s/he gets assigned a work item to be reviewed. The original work item was linked by a developer to a change package containing all changes that s/he performed to the code. So, while viewing the work item, the reviewer could quickly *apply* all the changes in the change package to his/her local workspace. After applying the changes, the reviewer could start immediately with the code review. The code review process would be improved due to automatic transfer of all changes to the local workspace and reduced setup time. Previously, these tasks had to be performed manually.

For the release management process, the base version of code and a set of features or requirements to be included into the release need to be specified. Using our approach, it is possible to merge all the change packages for every work item associated with the selected features or requirements over the base version of the code, ignoring already included change packages. Furthermore, it needs to be *validated* whether a change package is already included in the given code. Additionally, one could assemble the change log for the release automatically, as it is possible to identify the set of features, requirements or bug fixes implemented in the code. The release management process would benefit from automated assembling of code and change log creation. Previously, these tasks had to be performed manually, as well.

D. Requirements

Based on the ideas above on how to improve the processes of code review and release management, we have identified the following three requirements for a change package:

1) *Change package creation:* A change package must be producible from the current changes the developer has in his/her workspace.
2) *Change package application:* A change package must be applicable to a given set of code files.
3) *Change package validation:* It must be possible to validate if the changes in a package are already applied to a given code configuration.

In the following section, we discuss how to satisfy these three requirements to improve the processes of code review and release management.

IV. Approach

This work proposes an approach to (semi-) automatically establish traceability links between code and work items. Our approach links the exact changes which were made in the code to a work item. We assume that developers only work on one work item at a time and do not switch between different work items. The *core idea* for (semi-) automatically creating traceability links is letting the developer build the links himself/herself while using a VCS within the project. Whenever s/he finishes the implementation of a work item, the developer does not immediately commit the changes to the repository. Instead, before the commit, s/he orders the system to create a *change package* containing all changes in the code and associate it to a work item (see Figure 1).

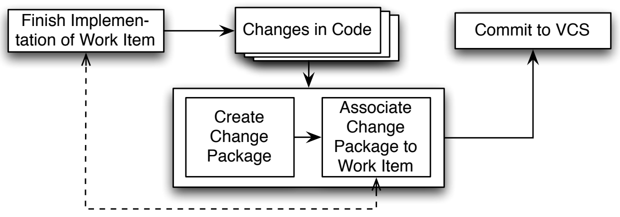


Figure 1. Process of creating Change Packages

The following subsections describe the proposed approach in detail with regard to two different ways of change package representation: a patch and a branch.

A. Patches as Change Packages

A patch is a file containing a set of changes between two versions of the code. The changes are stored in a specified format (e.g., unified diff) which allows to apply the changes contained in them to files (e.g., code), thus reproducing the patched version. Patches can be created and applied by almost all VCSs and even without a VCS, common programs like `diff` and `patch` under Unix can be used to create patches. The following mechanisms are used to fulfill the requirements for a change package:

1) *Change package creation:* A change package is created by creating the patch file.
2) *Change package application:* A change package is applied by applying the patch.
3) *Change package validation:* This is where patches reach their limits: It is rather difficult to check if a patch is already included in a given code. If the code was not changed afterwards, a simple check for the changes in the patch file can yield a result. However, if the code was changed afterwards (which is the more common case), comparing the content and the changes in the patch will not yield a result. Thus, relying only on a comparison of the patch content with the current file content is not suitable.

While the first two requirements are straightforward, the last requirement is challenging. There exist several possible approaches to implement the last requirement. For example, one is keeping a list of patches applied onto the code and linking this list with the version history in the repository. The problem with this approach is that it only works if all patches are applied using the system which tracks their application. If a patch is applied using common tools like the `patch` Unix command or the commands provided by the used VCS, this patch will be un-tracked.

A prototype for the patch representation of change packages was implemented. It is based on Subversion. Because it only uses the functionality of patch creation, it can also be adapted to other VCSs. The prototype currently does not support the check whether a patch is contained in the current version of the code. Therefore, the *patch-based approach can only support the code review process, but not the release management process*.

B. Lightweight Branches as Change Packages

VCSs ensure that no changes, besides the ones introduced by commits (and reliably logged), are done to the repository. Thus, a good approach for providing change package validation is to make them reside directly in the repository. By tying the representation closer to the repository, more of its tracking features can be leveraged.

The obvious choice for a change package representation which resides in the repository is a branch. A branch represents changes done to the code since the revision from which the branch started to diverge. The three required properties could be implemented for branches as follows:

1) *Change package creation:* A branch is created and changes are committed to this branch.
2) *Change package application:* A branch can be applied onto another branch of the repository by merging it into the other one.
3) *Change package validation:* By checking the revision graph, it can be deducted whether a branch has been

merged into another one (see details below). However, the repository has to support a revision graph to allow this approach.

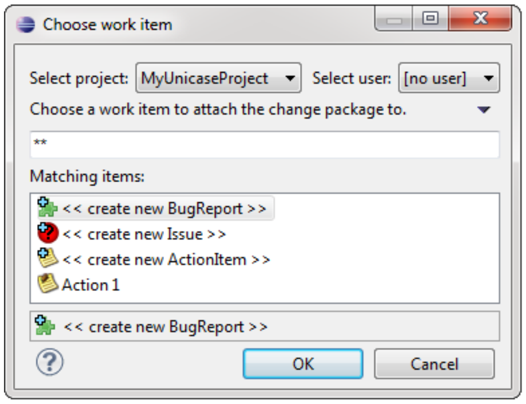
The basic idea of checking if a change package is already merged into a given branch is using the revision graph and performing a backward search (i.e., a search from a revision into the direction of its predecessor revisions). The search starts from the head revision of the given base branch which is to be checked for included change packages. If a revision identifying a change package branch (hereinafter called *indicator revision*) is found, a positive answer is given. If the search does not yield the indicator revision of a change package, a negative answer for this package is given. The representation of the indicator revision depends on the VCS. If the branch head of a merged-in branch is kept, this branch head can be used as indicator revision and stored in the change package. Otherwise, for example, the first commit on the branch can be used.

A drawback of this approach is that it restricts the VCS to be used. First, the VCS must support branches. This is no big restriction as most modern VCS do support this feature. The next limitation is more severe: The VCS must support a revision graph which reveals all predecessors of a revision which was created by merging. Subversion, for example, is not able to deliver this information and is therefore unsuitable for this approach. Finally, the branches must be *lightweight*: Since one change package is represented by one branch, numerous branches will exist concurrently in the repository. A branch is considered lightweight, if a large number of branches can be created without reducing the performance of the system and without taking too much space in the VCS. Additionally, the creation and merging of branches should be fast and the merge algorithms used should be sophisticated. It must be able to resolve most conflicts automatically, because this is cumbersome and error-prone.

One system that actively advertises its ability to maintain a large number of lightweight branches is Git. It also incorporates the use of sophisticated merge algorithms which reduce the amounts of conflicts propagated to the user. Thus, Git was chosen for the prototype implementation of the lightweight branch representation of change packages. In contrast to the patch-based prototype, the branch-based one is able to support all required three requirements. Thus, the *branch-based approach can support both processes of code review and release management*.

V. PROTOTYPE

A prototype of the presented approach has been implemented and integrated into UNICASE. After performing changes to the code, the developer selects a work item for the created change package (see Figure 2). For the code review, a reviewer gets assigned a work item to be reviewed. S/he can easily apply the linked change packages to his/her local


Choose work item

Select project: MyUnicaseProject
Select user: [no user]

Choose a work item to attach the change package to.

**

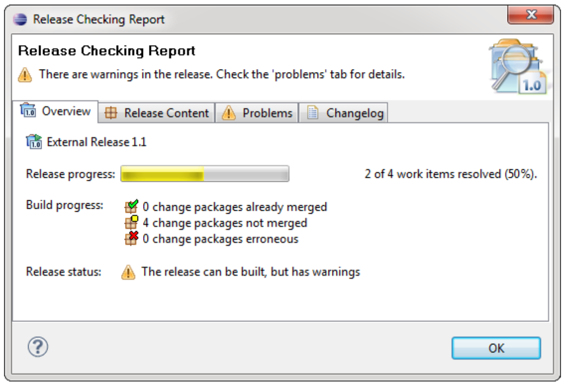
Matching items:

<< create new BugReport >>
<< create new Issue >>
<< create new ActionItem >>
Action 1

<< create new BugReport >>

Figure 2. Choosing a Work Item to associate to a Change Package

workspace. This process is eased in UNICASE by providing user-specific change notifications [9]. Once the developer finishes his/her work, the reviewer gets notified whether s/he can start with the review process. After the review, s/he can add a review summary to this work item and share it with the developer.


Release Checking Report

Release Checking Report

There are warnings in the release. Check the 'problems' tab for details.

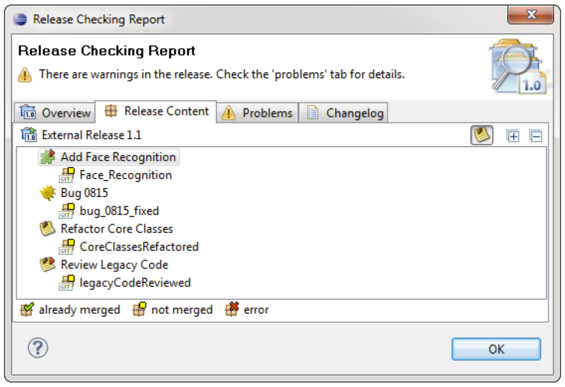
External Release 1.1

Release progress: 2 of 4 work items resolved (50%).

Build progress:
0 change packages already merged
4 change packages not merged
0 change packages erroneous

Release status: The release can be built, but has warnings

Figure 3. Release Management Support in Prototype: Overview


Release Checking Report

Release Checking Report

There are warnings in the release. Check the 'problems' tab for details.

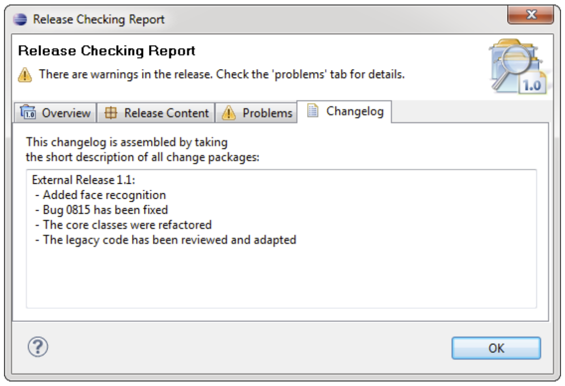
External Release 1.1

Add Face Recognition
    Face_Recognition
Bug 0815
    bug_0815_fixed
Refactor Core Classes
    CoreClassesRefactored
Review Legacy Code
    legacyCodeReviewed

already merged   not merged   error

Figure 4. Release Management in Prototype: Release Content

In Figures 3-5, an insight into the prototype implementation in UNICASE for the support of the release management



This changelog is assembled by taking
the short description of all change packages:

External Release 1.1:
- Added face recognition
- Bug 0815 has been fixed
- The core classes were refactored
- The legacy code has been reviewed and adapted

Figure 5. Release Management Support in Prototype: Changelog

is provided. The tool is able to show the progress of the build process as well as all already merged, not merged or erroneous change packages (see Figure 3). The release content tab shows all included work items in this particular release and their linked change packages (see Figure 4). The change log is automatically created (see Figure 5).

VI. Application of Prototype

We used the prototype during one sprint (4 weeks) in the development of the UNICASE project. In the following, we report on the processes of code review and release management with and without using the prototype as well as our experiences.

A. Code Review

The code review process *without* using the prototype was as follows: a developer got assigned a work item describing his/her work for implementation. After implementation, the developer created manually a patch file containing all changes. Then s/he send the patch via e-mail to a reviewer for code review. The reviewer downloaded the patch and applied it manually to his/her local workspace. Afterwards, the reviewer had to find the work item belonging to the patch. After the reviewer had reviewed all changes and agreed to all performed changes, s/he committed the changes to the VCS. Finally, the reviewer had to write a review comment to the work item indicating that the changes have been reviewed and applied.

The code review process *with* using the prototype was as follows: a developer got assigned a work item describing his/her work for implementation. After implementation, the developer selected the assigned work item (see Figure 2). If no work item existed for the performed changes, e.g., for a hot fix, the developer just created a work item on demand. After that, the changes were committed to the VCS in a separate branch. A change package was created containing a link to a branch and it was linked to the selected work item. The developer marked the work item as *done* and assigned a reviewer to it. The reviewer was automatically notified that new code changes were waiting for review using user-specific change notifications [9] in UNICASE. S/he opened the work item and selected the option to automatically download and apply the linked change package. All changes were automatically fetched/pulled from the branch in the VCS and applied to the local workspace of the reviewer. After reviewing all changes, the reviewer committed the changes to the main trunk of the VCS. Finally, the reviewer had to write a review comment to the work item indicating that the changes have been reviewed and applied.

B. Release Management

The release management process *without* the prototype was as follows: A release manager went through the list of all the work items and their linked features/requirements that the release should embody. Subsequently, the release manager went through each work item based on their priority and verified if their is a corresponding commit from the developer in the VCS assigned to the work item (switching back and forth between two different tools). Often, s/he also noticed that the patches were not applied to the branch from where the new release has to be made but rather at other place. In this scenario, s/he would have created a diff and applied it to the release branch that was checked out in his/her local workspace. If s/he was unsure whether a commit belongs to a work item, s/he first tried to contact and confirm with the assignee of the work item. Once the branch is ready by including all the code changes, the release manager committed all the local changes to the VCS and moved on to do the release manually or using a build server.

The release management process *with* the prototype was as follows: A release manager created a release model and added all the features/requirements the release should embody. Once s/he was done with a VCS checkout of the latest version for the release branch, s/he selected the “Build Release” option provided in the prototype. The prototype applied all change packages linked with every work item that are related to the features/requirements included in the release model automatically. Next, the prototype presented a release checking report to the release manager that showed a summary of the release process with various information, e.g. whether all the work items are resolved (Figure 3), all change packages merged to the release branch (Figure 4) and the automatically created change log from the work item descriptions (Figure 5). Once the release manager selected to do the release, all the change packages were merged and committed to the VCS repository.

C. Experiences

We noticed several advantages while using the prototype. First, the developers were able to automatically assemble all code belonging to the sprint and its planned work items and check if code could be assembled successfully. Second, if problems occurred during the release, these problems were reported. An overview about already merged, not merged or erroneous change packages helped the developers looking into the specific change packages. Third, the change log was automatically assembled. We learned that the presented approach significantly increased the productivity of our development team, e.g. the release management process using the prototype is now up to 4 times faster than before (30 minutes with and 2 hours without prototype).

VII. Related Work

In the following, related work for code traceability, code review and release management is discussed.

A. Code Traceability

Maintaining traceability links between code and other artifacts is a challenging task and therefore a field of intense research. Most approaches relate structures in the code like classes, methods, modules, files, or lines of code to other artifacts like requirements or features. Our approach tracks changes instead of structures in the code, instead.

A very simple, yet effective approach is presented by Treude et al. [10] embedding the links directly into code comments, which can be read by their proposed tool TagSEA. They use tags in comments to refer to other artifacts. They go further into the direction by creating links between tasks (equal to our definition of work item) and code. This is accomplished by connecting their tool with MyLyn [11], which can be used to express tasks. They connect code to MyLyn tasks by using special tags. A major drawback of using tags in the source code is the overhead of keeping the tags updated. In a complex project with a large number of artifacts it gets difficult for a developer to recall which code files should be connected to which work item. In our approach, we use the abstraction of patches over individual code artifacts to overcome this issue. Fischer et al. [12] link VCS with release data, which is in contrast to our approach. They use the version history in the repository, in addition to bug tracking data, to automatically build the release history of a project and allow viewing and querying it to retrieve information about the evolution of the project. While they do post-mortem analysis, our approach is focused on creating links between code and tasks during development to benefit code review and release management.

Marcus et al. [13] establish links between code and the corresponding documentation. In contrast to the approaches mentioned above, they try to recover the links automatically using information retrieval, namely Latent Semantic Indexing (LSI). This is different to our approach, as we establish links between code and work items (semi-) automatically while the developers use a VCS. Qusef et al. [14] use their SCOTCH tool for dynamic slicing and conceptual coupling to recover traceability links between unit tests and tested classes. Antoniol et al. [15] link code locations to object-oriented design artifacts. Like Marcus et al., they establish all the links without additional information, by performing different static and dynamic analyses. They use, amongst others, vector space indexing and probabilistic indexing techniques (comparable to LSI). All these methods are based on the textual similarity of artifact content, code identifiers and comments. This results in unreliable traceability links, as the created traceability links cannot ensure a high precision as well as a high recall at the same time.

B. Code Review

Several researchers contributed to the tool support of code review. Brothers et al. [16] proposed the ICICLE tool, which embodies different functionalities aiding code reviews. Examples are a human interface for preparing comments on the code under inspection and hypertext-based browsers for referring to various kinds of knowledge associated with code inspection, thus achieving a certain degree of traceability. Harjumaa et al. [17] proposed a web based tool, which features the distribution of the document to be inspected, annotation of it, searching of related documents, a checklist, and inspection statistics. Belli et al. [18] described an approach for the automatic handling, checking, and updating of check lists used in reviews. A similar approach of improving artifact quality by distributed artifact inspection was presented by De Lucia et al. [19]. They focused on general aspects of the artifacts life-cycle and presented a distributed inspection process consisting of seven phases implemented in a tool called WAIT.

All these approaches are in contrast to ours, as we are able to transfer the changes to be reviewed automatically as well as providing traceability from the code over work items to other artifacts, e.g., features or requirements. There are further tools for code review which can do a lot of other things that our tool does not support. However, these tools do not support the functionality discussed here.

C. Release Management

Among numerous research work aiming at improving the release management process, we identified two major contributions related within the scope of our work. Van der Hoek et al. [8] identified the basic issues in release management and developed a tool to aid the release management process. In contrast to our approach, their approach does not include assembling and building components to be released. It is merely a database of releases, components to be released and their dependencies among each other. Saliu et al. [20] contributed research in the field of release planning, especially for evolving systems which are built incrementally. Our approach supports release planning as well, since work items linked to changes in the code are included in a release.

VIII. Discussion

The proposed approach works with certain constraints and assumptions. For example, it is also possible that a change belonging to a work item has been committed to the repository without the creation of a change package. This can happen if the code is committed with the sources of another change package, which happens if a developer was working on two work items and committed the changes for both in one change package only. However, this can be noticed by the developer if s/he realizes that one of the work items has a missing change package. The approach leaves the task of creating links between the code changes and the work items to the developer himself/herself. So, the approach suffers from mistakes a developer does while performing this activity (like linking code to the wrong work item). The general problem is that artifacts in human readable text or code can not be linked together with full reliability using any existing technique. Our approach of letting the developers create the links themselves as part of their usual work is expected to be better in comparison to automatic approaches.

IX. Conclusion

This paper proposed an approach for (semi-) automatically creating traceability links between code and work items by using VCSs. The created traceability links were used to improve the processes of code review and release management. The idea to automate the code review process was to use traceability links between code and work items to apply the code to the reviewers machine, highlight the changes and add the reviewers feedback to the work items. For the release management, the traceability links between code and work items were used to check which work items are contained in the code of a release and to build a release automatically by merging in the missing features or requirements.

The techniques and applications discussed in this paper, like the establishment of links between code and work items, are still not explored in depth. Therefore, there are a lot of possibilities for future research in this area. We are aware that our approach currently only allows developers to work at one work item at a time. Therefore, support for working on several work items at once is subject to future work. Furthermore, we want to study further possibilities for the application of our proposed approach. The work presented in this paper has been evaluated so far only with a small user group during one sprint in the UNICASE project. Thus, we plan to conduct a representative user experiment. A study where the prototype is used throughout a development project would be very beneficial to evaluate the benefits offered by the proposed approach to the code review and release management processes.

References

[1] Tichy, W.F. Tools for software configuration management. In Proceedings of the International Workshop on Software Version and Configuration Control, pp. 1-20 (1988)

[2] IEEE. IEEE standard for software configuration management plans: ANSI/IEEE std. 828-1983 (1983)

[3] Leon, A. Software configuration management handbook. Artech House, Inc. Norwood, MA, USA (2004)

[4] 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)

[5] 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)

[6] UNICASE Open Source Project. http://www.unicase.org/

[7] Helming, J., David, J., Koegel, M., and Naughton, H. Integrating System Modeling with Project Management - A Case Study. In COMPSAC 09: International Computer Software and Applications Conference, pp. 571-578 (2009)

[8] Van der Hoek, A., Hall, R., Heimbigner, D., and Wolf, A. Software release management. Software Engineering - ESEC/FSE, pp. 159-175 (1997)

[9] Helming, J., Koegel, M., Naughton, H., David, J., and Shterev, A. Traceability-Based Change Awareness. In MODELS 09: International Conference on Model Driven Engineering Languages and Systems, pp. 372-376 (2009)

[10] Treude C. and M.A. Storey. How tagging helps bridge the gap between social and technical aspects in software development. In ICSE 09: International Conference on Software Engineering, pp. 12-22 (2009)

[11] Eclipse MyLyn. http://www.eclipse.org/mylyn/ [retrieved: September, 2012]

[12] Fischer, M., Pinzger, M., and Gall, H. Populating a Release History Database from version control and bug tracking systems. In ICSM 03: International Conference on Software Maintenance, pp. 23-32 (2003)

[13] 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)

[14] Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., and Binkley, D. SCOTCH: Slicing and Coupling Based Test to Code Trace Hunter. In 18th Working Conference on Reverse Engineering (WCRE), pp. 443-444 (2011)

[15] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. Maintaining traceability links during object-oriented software evolution. Software: Practice and Experience, vol. 31, no. 4, pp. 331-355 (2001)

[16] Brothers, L.R. Multimedia groupware for code inspection. International Conference on Discovering a New World of Communications (ICC), pp. 1076-1081 (1992)

[17] Harjumaa, L. and Tervonen, I. A WWW-based tool for software inspection. In HICSS, Published by the IEEE Computer Society, pp. 379-388 (1998)

[18] Belli, F. and Crisan, R. Towards automation of checklist-based code-reviews. In ISSRE 96: International Symposium on Software Reliability Engineering, IEEE Computer Society, pp. 24-33 (1996)

[19] De Lucia, A., Fasano, F., Scanniello, G., and Tortora, G. Improving artefact quality management in advanced artefact management system with distributed inspection. Software, IET, vol. 5, no. 6, pp. 510-527 (2011)

[20] Saliu, O. and Ruhe, G. Supporting software release planning decisions for evolving systems. 29th Annual IEEE/NASA Software Engineering Workshop, pp. 14-26 (2005)