

# Traceability in a Fine Grained Software Configuration Management System

Martin Eyl<sup>1</sup>(✉), Clemens Reichmann<sup>1</sup>, and Klaus Müller-Glaser<sup>2</sup>

<sup>1</sup> Vector Informatik GmbH, Ingersheimer Straße 24, 70499 Stuttgart, Germany  
Martin.eyl@vector.com

<sup>2</sup> Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany

**Abstract.** Traceability between artefacts from different domains (e.g. requirements management or test data management) is important in the software development process. Therefore modern application lifecycle management solutions support traceability links between these artefacts. But the support of traceability links into the source code is still very rudimentary or does not exist at all, although the source code is of central importance. Traceability links between artefacts in a repository and source code can break very easily when changing the text. To solve this problem we store the source code as Abstract Syntax Tree (AST) in a repository. A special editor for the source code, which supports refactoring, makes robust traceability between the AST artefacts and other artefacts possible. The repository provides the version history of all AST artefacts including their traceability links for a better understanding of changes over time. This paper introduces an implementation of such a system based on Eclipse.

**Keywords:** Traceability · Fine grained software configuration management system · Abstract syntax tree

## 1 Introduction

The importance of traceability between development artefacts created during the software development life-cycle is well understood and incorporated in numerous software development standards [1]. Traceability, in particular requirements traceability, has received quite a lot of attention from the research community [2–7]. Traceability links from the requirement into the source code are crucial and can help to answer among others the following questions [6]: Where can I find the source code, which implements this requirement? Do all functional requirements have a concrete implementation? Why has this source code been developed? But not only requirements traceability requires links into the source code. There are more important use cases: Which defects have been fixed in this “if-statement” in the source code (Change Management)? Where can I find the source code for the automated tests which verify this requirement (Test Data Management)? Is there any additional documentation or a UML diagram for this class available (Design and Documentation) [8]?

Some commercial application lifecycle management solutions (e.g. Polarion Software [9] or Rational Team Concert [10]) integrate several domains in one application

and in one repository. These applications support the creation of traceability links between artefacts of different domains e.g. between test data management and requirement management. But the support of traceability links into the source code is still rudimentary. Very common is the possibility to link a requirement to a set of source code files which has been changed for the requirement. At most traceability links to classes [7] are supported. The reason is that the smallest unit stored in the source code configuration management systems is a file which usually contains a class. A traceability link into the text is difficult to maintain because by changing the text the link can break.

A Java class contains fields and methods. A method contains statements like a “for loop” or an “if statement”. At least for all of these artefacts traceability links should be possible. In addition the expectation towards a software configuration management system is that the complete history of these artefacts including the traceability links can be retrieved. Fine granular traceability links are needed because of the following reasons:

- The traceability between a test case and a method of a unit test in the source code is important because of the following reason: If the test case is linked with a requirement (test case tests the requirement), it is possible to find all test methods in the source code which verifies a certain requirement.
- Usually not only one requirement is implemented in a Java class. This would be too restrictive. Different methods might be linked to different requirements. Over time after several changes and refactorings (for example moving statements to another class) there is the need for traceability links between requirements and single statements otherwise we would lose the information why this statement has been developed.
- The software developer is not only interested in the last reason for change of the statement but also the developer wants to know all requirements and defects which caused a statement to be changed over the complete history regardless of how often the statement has already been moved in the source code between different classes and methods.
- Documents (e.g. an activity diagram or an image) which are linked to only parts of a class can provide additional information for a better understanding of the source code. For example a presentation explaining an algorithm can be linked to the “for loop” which implements the algorithm.

In order to support this kind of fine granular traceability links into the source code and to benefit from the links the following concepts are needed:

1. A consistent metamodel which allows us to define traceability links between different artefacts and the source code. This can be achieved by defining a metamodel for the source code.
2. An editor which allows us to change the source code without losing the traceability links into the source code.
3. Very good support for creating and maintaining the traceability links to keep expense low for the developer.
4. Visualization of the traceability links during the development of the source code in the source code editor.

These concepts have been implemented in a prototype called Morpheus. Morpheus is an extension for the Eclipse Integrated Development Environment (IDE) [11]. We did not want to develop a new IDE and therefore one major goal was that the software developer still can use all the powerful features of a standard IDE including the management of the projects and the Java editor.

## 2 Metamodel

For all domains in the software development process it is useful to define a metamodel with meta classes like requirement, test specification, test case, or ticket and traceability links between these meta classes. Often the following two solutions are used to store traceability links from the artefacts of these domains into the source code:

1. The full qualified name for example in Java the package name, class name and the method name are stored with the artefact. If the developer for example renames the method, all traceability links to this method have to be searched and updated. This can be quite expensive and complex. Also it is not possible to store links to source code which does not have an explicit name for example a “for loop” or an “if statement”. As well it is difficult and costly to follow the link from the source code to the artefact.
2. The directory name, the file name and the line number are stored with the artefact. But this information is only valid for a certain version of the file (which also has to be stored with the artefact) because with every change in the file the line number can get invalid. So, we can only follow the traceability link to a certain version of the file in the history and actually we have no information where the link points to in the current version of the file.

Surely it would make a lot more sense if everything including the source code could be defined as one consistent metamodel. Then all traceability links could be handled in the same way whether the traceability link points into the source code or not. The solution is to integrate the Abstract Syntax Tree (AST) of Java into the entire metamodel. The complete model including all requirements, test cases, tickets and all AST artefacts can then be stored into one data backbone. The goal is to define one metamodel for all aspects of the software development process with traceability links between the meta classes. The links are bidirectional and can be traced with little effort.

For Morpheus we used the Meta Data Framework (MDF) of PREEvision [12–14]. MDF is based on the OMG’s Meta Object Facility (MOF) Standard [15]. MDF provides an editor for defining the metamodel and to generate the Java source code for the model. We also use the data backbone of PREEvision to persist the model. The client of PREEvision is based on Eclipse and so several plugins of PREEvision are used to load and store the model and to visualize the model in the graphical user interface. Also parts of the metamodel of PREEvision are reused (e.g. requirements or tickets).

## 2.1 Metamodel Generator

The Java AST metamodel can be derived from the Java Language Specification of Oracle [16]. By doing this manually many errors can creep into such a metamodel. To avoid this we developed a metamodel generator, which allows us to generate a MDF metamodel from the Backus-Naur Format (BNF).

Eclipse has a lot of functionality regarding parsing and processing of Java source code. Eclipse provides a Java AST and an AST parser, which creates an AST from a source code file. To use these and other functions, it is beneficial if the metamodel is very similar to the Eclipse AST. Therefore we extended the metamodel generator so that it is also possible to process the Eclipse Java AST classes as input and to generate a suitable metamodel. With this additional feature of the generator it is now possible to use the MDF metamodel with the Eclipse functionality.

## 2.2 Traceability into the Source Code

In the first version of Morpheus we have considered the following use cases in the metamodel.

**Test Data Management.** The *TestSpecification* describes how functional requirements must be tested in order to ensure the quality of the software (see Fig. 1). It is created in natural language regardless of the test implementation. Typically, the *TestItems* include different use cases and they are linked to the corresponding *Requirements* and *Tickets*.

A *Ticket* can be a change request or a defect. To ensure that a defect will not reappear in the next releases and to increase the test coverage of the automated tests it makes sense to create a test for a defect. A *Requirement* represents not the complete requirement specification but only one single requirement within a specification. The requirement has to be specific and it is written in natural language. Usually the requirement is a functional requirement which can be expressed directly in source code.

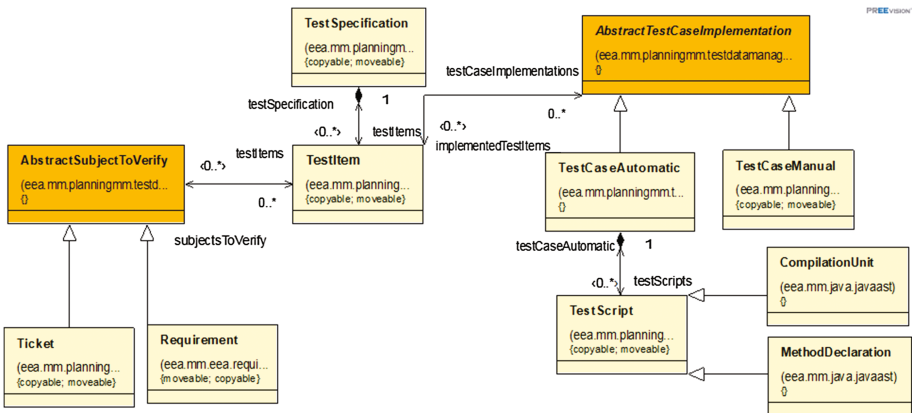


Fig. 1. The metamodel for test data management

The *Requirements* can be hierarchically structured in requirement packages which build up a complete requirement specification.

Test implementations are provided for the specified test items, which can be brought to execution. A test implementation is a manual test which is executed by a human tester (*TestCaseManual*) or an automated test (*TestCaseAutomatic*). The automated tests can then be linked to a compilation unit (e.g. unit test class) or a method of a compilation unit. The meta classes *CompilationUnit* and *MethodDeclaration* are AST artefacts. The *MethodDeclaration* is contained below a *CompilationUnit*. If the software developer renames or moves a test method to another unit test class the traceability link to the test case and therefore also to the *Requirement* or *Ticket* survives. The *TestCaseAutomatic* is linked to the *CompilationUnit* or the *MethodDeclaration* because the complete test class or the test method can be executed during automated test execution.

Traceability links can be traced in both direction and so it is possible to find all automated test code for a requirement or ticket and it is possible to see which requirement or ticket is tested by a certain test method or test class.

**Requirement and Change Management.** The meta class *ASTNode* is the base class of all AST artefacts in the metamodel (see Fig. 2). When the developer changes the source code, already existing AST artefacts are changed or deleted and new AST artefacts are created. This set of AST artefacts is put into the *ChangedArtefactSet* which is connected to a *Requirement* or *Ticket*. Therefore traceability between the reason for change (requirement and ticket) and the changed source code is possible in both directions.

With every change of an AST artefact a traceability link to a requirement or ticket is created. There can be more than one requirement because the source code can be relevant for different requirements. For the requirements all test methods can be identified. All this information together (source code – requirement – test method) can be used to determine which automated tests should be executed when source code (AST artefacts) has been changed. This test selection strategy can be very useful during Continuous Integration (CI) to determine which automated tests shall be executed for the committed source code [17].

**Documentation.** The source code can be documented by comments in the source code text. But the software developer can only use characters. Some formatting is possible with HTML tags but editing is difficult. It is not possible to use images, diagrams,

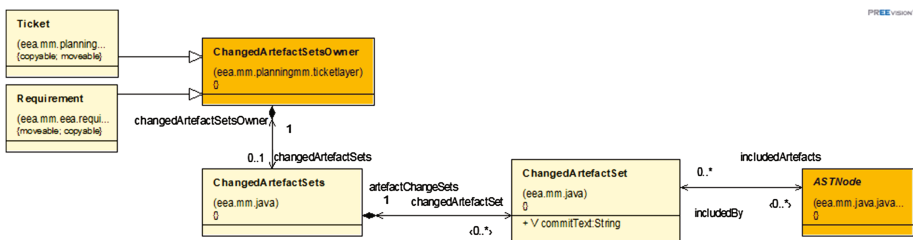


Fig. 2. The metamodel for requirement and change management

spreadsheets, or presentations. Traceability links between any kind of document containing documentation for the source code and the source code itself is a substantial improvement. Design documents can also be linked to all relevant AST artefacts.

The metamodel of MDF has the special meta class called *FileAttachment* which makes it possible to store files with the model in the data backbone (see Fig. 3). The meta class *FileAttachment* supports any document format, e.g. WinWord, Powerpoint or Excel document. The file attachment can be placed in a *SourceCodePackage* or in a *FileAttachmentPackage* contained in a *SourceCodePackage* independent from the AST artefacts. The software developer can then link the file attachment with any AST artefacts via the meta class *ModelContext* to express the relevance of the file attachment for this source code. Additional documentation stored in a *FileAttachment* can be useful for any AST artefact for example a class, a method, a statement or a field declaration. Moving the source code to another class will not break the traceability link.

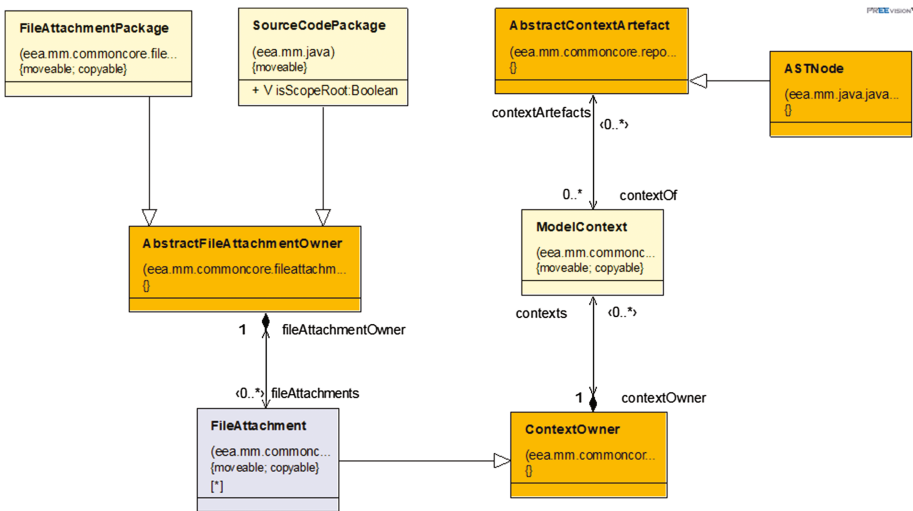


Fig. 3. The metamodel for documentation

### 2.3 Implications of the AST Model

The following implications have to be considered when using an AST model instead of text.

1. For editing the AST model we need a special AST editor. It is not possible to convert the AST into text and to change the text in any text editor and to convert the text back to an AST because then we could lose traceability links to the source code (for more details see next section). Therefore we need a special text editor or a complete new AST editor.
2. The syntax of Java can change with every new Java version and so a new meta-model is needed for a new Java version. The source code of Java 1.8 cannot be

stored in the metamodel for Java 1.7. Usually the changes are backwards compatible so that the source code must not be changed and a migration of the model via a model to model transformation [18] is not necessary. The metamodel will only be extended.

3. Whatever the user enters in the editor must be convertible to an AST so that it can be stored in the model. Therefore it is not possible to store certain kinds of syntax errors in the model.
4. If the user temporarily comments source code out e.g. for testing purpose, traceability links could get lost. It is of course possible to save the comment in the AST, but the original AST artefacts with possible traceability links are then deleted. After removing the comment and thereby restoring the original source code the AST artefacts are newly created without any traceability links. For such a use case a special support is needed. Here is potential for improvement for Morpheus.

### 3 Java Editor

A special Abstract Syntax Tree (AST) editor is needed for editing the AST model. The Java Editor of Eclipse is very powerful with many functions like syntax highlighting, code assistant, quick fix, integrated debugging and more. So, we wanted to reuse this editor and only extend the functionality for editing the AST. We call this editor “Java AST Editor”.

The input and output of the Java editor is text. Therefore the AST has to be converted to text before editing and the text has to be converted to an AST after editing. Thereby the following problem has to be solved (shown as an example for renaming a method): If the developer renames a method and saves the source code in the AST model, a new method declaration is created and the existing method declaration is deleted including all traceability links to this method. During saving it is impossible to know whether the developer has renamed the method or deleted a no longer needed method and created a new one. This is determined by how the user has changed the text: The developer changes the name of the method in the text or the developer deletes the text of the method and starts writing a new method. Therefore we have to keep track of the AST artefacts in the text. This is accomplished by the following features of the Java AST Editor:

- During opening the Java AST Editor the AST artefacts from the model are converted to text. For each AST artefact the start position and the length in the created text is determined and stored in the editor as mapping information. So, the editor knows for each position in the editor which AST artefacts are located on this position.
- If the developer enters new characters or deletes characters, the start position and length of the AST artefacts have to be corrected. If the changed text is located at the position of the AST artefact, the length has to be corrected. For all other AST artefacts behind the changed position, the start positions have to be adapted.
- If the developer moves text in the editor (e.g. per drag and drop), removes text via the clipboard command cut or adds text via the clipboard command insert, the start position and length of all relevant AST artefacts have to be adapted.

- If the developer copies source code text into the clipboard via the clipboard command cut, not only the text itself is put into the clipboard but also all AST artefacts contained in the text and their start positions and lengths. If the developer pastes the text back into the editor, the developer can decide whether only the text or the AST artefacts shall be used. In the first case new AST artefacts are created. In the second case the list of AST artefacts stored in the editor is updated with the information from the clipboard. So, it is possible to move source code from one class to another without losing any traceability links to the moved AST artefacts.

When the text of the editor is saved, the text is converted back to AST artefacts. These AST artefacts are then merged into the model. Why cannot the new created AST artefacts from the editor just replace the original AST artefacts in the model? The reason is that the AST artefacts from the editor do not have any traceability links which might exist in the model and these traceability links have to be merged. MDF provides a powerful merge engine which is used for this purpose. The merge engine uses the mapping information of the editor (the AST artefacts from the model and their current position in the current text) so that the AST artefacts from the editor can be matched to the AST artefacts in the current model and so a merge can be executed. The AST artefacts in the current model are then modified during the merge.

With the Java AST Editor the developer can just edit the source code as before without knowing that the source code is stored as AST artefacts in a model. But the Java AST Editor cannot really know what the developer intends to do. Does the developer want to modify a method although the developer deletes the text of the method and creates a new one with the same name? Does the developer want to create a new method although the developer just changes the name of the method? So the developer must be aware that AST artefacts with traceability links are edited and the developer must understand when a method will be created, changed or deleted: As long as the developer is not deleting the text of the method and is only changing text, no new method is created and the current method will be changed. With this knowledge the developer can change the text in the proper way according to his or her intentions.

Via the clipboard it is possible to copy source code. In this case new AST artefacts are created. The developer can decide whether to take over the traceability links from the copied source code or not.

## 4 Traceability Link Creation and Visualization

In the first version of Morpheus the focus was on supporting the software developer for traceability link creation and visualization. For the roles test manager, product manager or project manager additional functionality and additional reports are possible and useful.

### 4.1 Test Data Management and Documentation

**Traceability Link Creation.** PREEvision provides a view called “Model View” (1) (see Fig. 4) which shows all artefacts of the currently loaded model. This view is

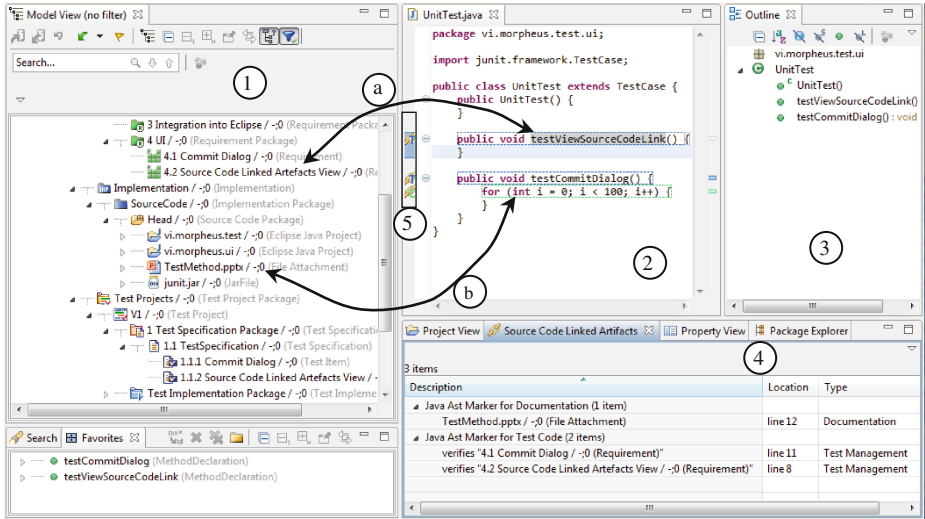


Fig. 4. Support of drag and drop to create links

integrated into the Eclipse Integrated Development Environment (IDE). The Java AST Editor (2) knows for every cursor position the corresponding Abstract Syntax Tree (AST) artefact. Therefore drag and drop into the editor or out of the editor is possible.

If the developer wants to document that the newly created method “testViewSourceCodeLink” verifies the requirement “Source Code Linked Artefacts View”, then the developer drags the requirement from the model view into the editor and drops it onto the method (a). Alternatively the developer can also drag the text with the name of the method onto the requirement in the model view. It is also possible to use the “Outline” view (3) as drop target or as a starting point of a drag and drop operation. After dropping the requirement on the test method declaration Morpheus creates automatically a test specification (if not already existing), a test item and a test case in the model and links all the artefacts including the method declaration. Of course the developer can also use an existing test item or test case for the drag and drop operation.

A similar drag and drop operation can be executed with the file attachment with the difference that the file attachment can be dropped on any AST artefact and not only on a method declaration (b). Again both directions from model view into the Java AST Editor or from the Java AST Editor into the model view are possible.

**Traceability Link Visualization.** For the traceability link visualization in the Java AST Editor the marker functionality of Eclipse has been used. The traceability link markers are shown in the “Source Code Linked Artefacts” view (4) and on the marker bar (5) in the editor area (see Fig. 4). By double clicking on a traceability link marker in the view the according source code text is highlighted and the linked artefact in the model view is selected. By clicking on the icon in the marker bar a list of all linked artefacts is displayed in a popup window and a tool tip for each artefact provides additional, detailed information (see Fig. 5). Double clicking on the linked artefact in

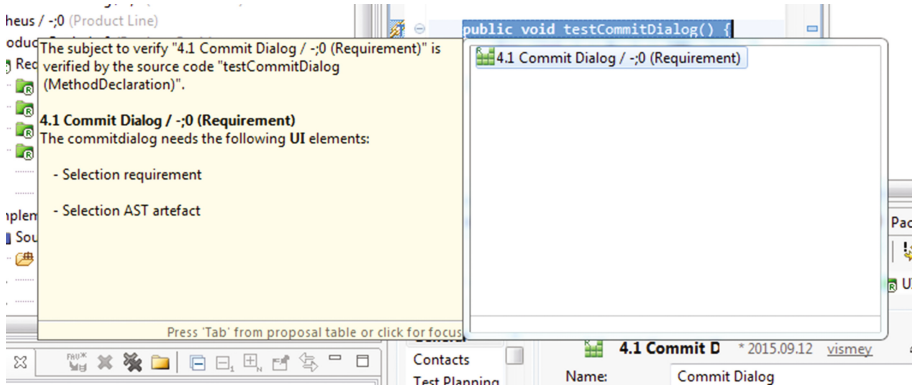


Fig. 5. Popup window showing linked requirements

the popup window selects the artefact in the model view and in the case of a file attachment opens the file in the according editor.

The relevant information (requirement, ticket or document) for the method or the current line of source code is directly available without switching to a different application and without searching for the information. The deletion of a traceability link can be done via a context menu in the “Source Code Linked Artefacts” view.

## 4.2 Requirement and Change Management

**Traceability Link Creation.** For the traceability between requirement and the source code implementing the requirement, links are needed between these artefacts. Similarly traceability links can be created between tickets and the changed source code which solves the ticket. Morpheus handles requirement and ticket equally. So, in this section the term requirement can also be replaced by ticket.

The effort to link these artefacts can be enormous if it is done after the development task has already been finished. The best time to create this traceability link is during the commit of the changed source code [7]. The commit stores the changes in the data backbone and makes them available for all users. The developer has to provide the reason for change by selecting the requirement which can be done in the commit dialog. Additionally a commit comment can be entered. If the developer works in parallel on several requirements, the developer has to select more than one requirement and has to decide which changed AST artefact belongs to which requirement. This can also be done in the commit dialog.

Mylyn [19] is used to simplify this task for the developer. Mylyn is a task management tool for software developers integrated into Eclipse. The tasks are usually imported from an application lifecycle management repository. In our case each requirement or ticket in the model represents a task and they are imported into Mylyn if these artefacts are assigned to the current developer. The developer can activate a task in

Mylyn at the beginning of his or her work. Mylyn keeps then track of all touched artefacts by creating a context for this task. During the commit Morpheus retrieves this information from Mylyn (each task with its touched artefacts) and determines the correct mapping between requirements and the relevant changed AST artefacts. If the developer uses Mylyn then there is no additional effort during the commit of the source code.

With every commit of source code a *ChangedArtefactSet* is created with the following information: author, date of commit, commit comment and all changed AST artefacts. Morpheus links the *ChangedArtefactSet* with the selected requirement or ticket.

**Traceability Link Visualization.** Eclipse provides the functionality to display an annotation bar in the editor area. In the Java editor this annotation bar is used to show information about the last modifications in the source code. This information is retrieved from the Software Configuration Management (SCM) system. Besides the author and the change date also the commit comment is displayed in a popup window. The color of the annotation represents the author (different colors for different authors) or the date (newer changes are displayed in a lighter color).

For Morpheus the annotation bar has been modified in such a way that the necessary information is now retrieved from the model. The traceability link from the source code to the requirement or ticket can now be used to retrieve title and content of these artefacts which are displayed in the annotation bar (1) and in a popup window (2) in addition to author, change date and commit comment (see Fig. 6). Different colors are used for different requirements or tickets.

The annotation bar of the Java editor can only show information about the last modification for a line in the source code text. Although it is possible to compare two

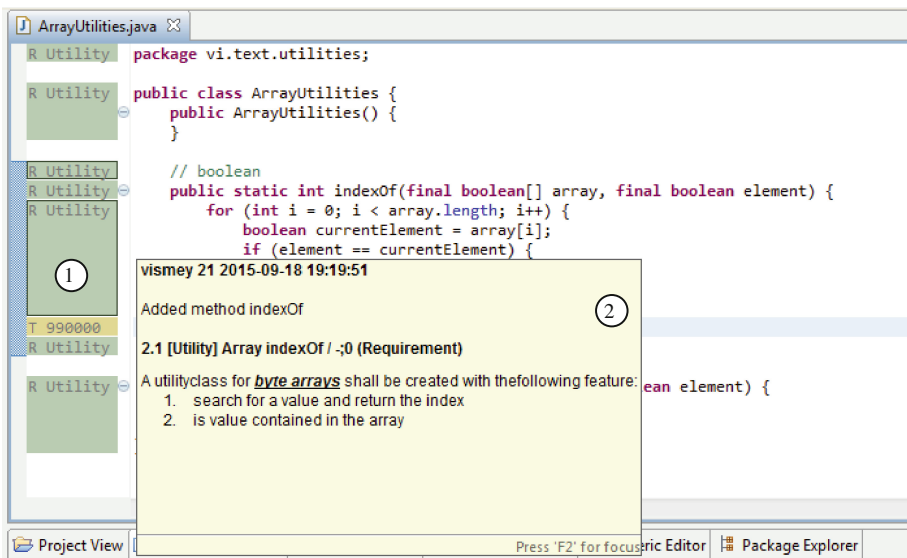


Fig. 6. Annotation bar showing history information of AST nodes

arbitrary file versions from the history, a single AST artefact in two versions cannot be compared. Eclipse is not able to find the matching AST artefacts because position and/or name in the file might have changed. A typically use case is the following: the developer wants to know when, why and who has changed the “for loop” in the last two years. By comparing different versions of the file the developer can try to find the “for loop” in the older versions of the file which can be very time-consuming and difficult. If the “for loop” has been moved from one file to another file then the developer has to check all committed files to find the “for loop”. The more changes (beside the change in the “for loop”) have been made in the different versions of the file the more difficult it is to find the relevant source code text.

Morpheus knows the exact history of any AST artefact because not only the latest version is stored in the data backbone but also every committed version can be retrieved from the data backbone. Via the context menu in the annotation bar or directly in the editor the developer can receive detailed information about the history of an AST artefact (see Fig. 7). In the popup window every commit, where the AST artefact has been changed, is displayed in a list (in this example for the “return” statement) with the information who has changed it and when was it changed. A tool tip window contains information about the linked requirements and tickets with the reason for change.

Two entries in the list (two versions of the AST artefact) can be selected and can then be compared. For this the complete compilation unit, where the AST artefact is contained, is loaded from the data backbone and converted to source code text for both selected versions. The source code text can then be displayed in the compare editor of Eclipse. If the AST artefact has been moved from one compilation unit to another, two different compilation units are loaded and compared with the AST artefact in two different versions.

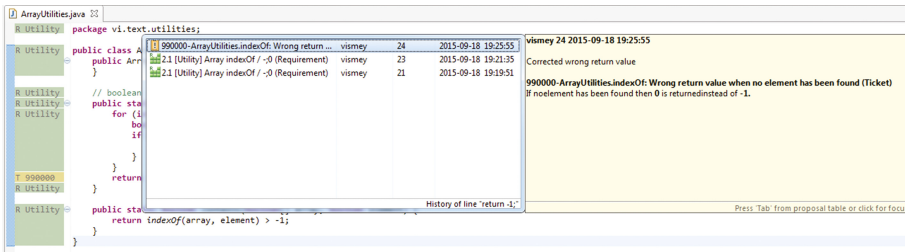


Fig. 7. Popup window showing the detailed history of the AST artefact “return -1”

## 5 Related Work

### 5.1 Fine-Grained Version Control Systems

All major and popular version control systems can only manage coarse grained data chunks (files). But there are several research projects in the field of fine-grained version control systems (for example COOP/Orm [20] or Sysiphus [21]) and some of them support source code.

MolhadoRef [22] is a refactoring-aware software configuration management system. It stores program entities (classes, fields and methods) in nodes, slots and attributes and records the refactoring operations that change them. A node represents a program entity and a slot holds the values. Attributes map nodes to slots. The recorded operations are replayed to transform one version to another. So, the history of refactored program elements can be tracked. The generic versioned data model of MolhadoRef allows storing programs in different languages. A concrete implementation for Java with an integration into Eclipse has been developed. MolhadoRef does not support AST artefacts below a method and provides no traceability functionality.

Stellation [23] is a software configuration system which supports fine (method level) storage granularity by using so called fragments. One key feature is the multi-dimensional program organization which allows multiple overlapping viewpoints of the fragments instead of having only one viewpoint which is dominated by the layout in source code files. Therefore Stellation provides a query language which allows the developer to search the repository for relevant fragments and present the result in a source-file like form. Stellation (as well as MolhadoRef) does not support traceability to requirements, tests or documentation and does not include any AST artefacts below a method.

## 5.2 Traceability

UNICASE Trace Client [6] is a tool which provides a traceability information model consisting of artefacts from requirements engineering, project management and source code and traceability links between these artefacts. The capturing of the traceability links between requirements, work items and code has been solved similar to Morpheus and is done during the commit of the source code into the software configuration management system. Refactoring of a class (rename, delete, split up or unite classes) is supported. UNICASE Trace Client supports only traceability links to classes. Links below class level are not possible.

## 5.3 Domain-Specific Language (DSL) Development Environment

JetBrains Meta Programming System (MPS) [24] is an environment for language engineering which allows to create own domain-specific languages. MPS can also be used for Java. The editor of MPS is called a projectional editor because the Abstract Syntax Tree (AST) is edited and not text. The developer is not completely free to enter any text. The entered text must be transformable to an AST. Every AST artefact has a unique identifier and refactoring is possible without losing the identity of the AST artefact. The development environment stores the AST of one software module in an XML file on the hard disc or in a software configuration management system (for example Subversion [25]). The history of an AST artefact is expensive to calculate and traceability links are not supported. The support of traceability links would be difficult because the AST artefacts exist only inside the XML files.

## 6 Conclusion and Future Work

In this paper we presented Morpheus as an extension of Eclipse which allows the software developer to create fine granular traceability links into the source code to support traceability between source code and requirements, defects, test cases and documentation. Morpheus supports refactoring of the source code (e.g. renaming or moving of source code) so that the traceability links will not break. This is achieved by integrating the Abstract Syntax Tree (AST) of Java into the model and by using a text editor which is aware of the AST artefacts. The model can be stored in a data backbone. With Morpheus integrated into Eclipse all features of Eclipse can still be used as usual.

We are currently working on the following improvements: Firstly, syntax errors in the source code stored in the data backbone shall be prevented. Without any syntax errors in the model the application can always be built and continuous integration will no longer fail because of syntax errors. Some types of syntax errors are not possible because the source code is stored as AST. But other syntax errors can even exist in an AST for example a method invocation with wrong parameter types.

Secondly, we plan to avoid the effort for merging source code. When several developers change the same code, they have to merge their changes and this is costly and error-prone.

## References

1. Cleland-Huang, J., Gotel, O., Zisman, A.: *Software and Systems Traceability*, vol. 2. Springer, Heidelberg (2012)
2. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: *Proceedings of the First International Conference on Requirements Engineering*. IEEE (1994)
3. Pinheiro, F.A.C.: Requirements traceability. In: do Prado Leite, J.C.S., Doorn, J.H. (eds.) *Perspectives on Software Requirements*. Kluwer International Series in Engineering and Computer Science, pp. 91–114. Springer, Heidelberg (2004)
4. Bacchelli, A., Lanza, M., Robbes, R.: Linking e-mails and source code artifacts. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 1. ACM (2010)
5. Corley, C.S., et al.: Recovering traceability links between source code and fixed bugs via patch analysis. In: *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM (2011)
6. Egyed, A., Grunbacher, P.: Automating requirements traceability: beyond the record & replay paradigm. In: *17th IEEE International Conference on Proceedings of the Automated Software Engineering, ASE 2002*. IEEE (2002)
7. Delater, A., Paech, B.: Tracing requirements and source code during software development: an empirical study. In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE (2013)
8. Marcus, A., Maletic, J.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: *25th International Conference on Proceedings of the Software Engineering*. IEEE (2003)

9. Inc., Siemens: Application Lifecycle Management (ALM), Requirements Management, QA Management | Polarion Software (2016). <http://www.polarion.com/>. Accessed 30 July 2016
10. IBM: IBM - Rational Team Concert (2016). <http://www-03.ibm.com/software/products/de/rtc>. Accessed 30 July 2016
11. Foundation, Eclipse: Eclipse Neon (2016). <http://eclipse.org>. Accessed 30 July 2016
12. Vector Informatik GmbH: PREEvision – Development Tool for model-based E/E Engineering (2016). [https://vector.com/vi\\_preevision\\_en.html](https://vector.com/vi_preevision_en.html). Accessed 30 July 2016
13. Matheis, J.: Abstraktionsebenenübergreifende Darstellung von Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zur Ableitung von Sicherheitszielen nach ISO 26262. Shaker (2010)
14. Zhang, R., Krishnan, A.: Using delta model for collaborative work of industrial large-scaled E/E architecture models. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 714–728. Springer, Heidelberg (2011). doi:10.1007/978-3-642-24485-8\_52
15. OMG: OMG's MetaObject Facility (MOF) Home Page (2016). <http://www.omg.org/mof/>. Accessed 30 July 2016
16. Oracle: Java SE Specifications (2016). <https://docs.oracle.com/javase/specs/>. Accessed 30 July 2016
17. Eyl, M., Reichmann, C., Müller-Glaser, K.: Fast feedback from automated tests executed with the product build. In: Winkler, D., Biffl, S., Bergsmann, J. (eds.) SWQD 2016. LNBIP, vol. 238, pp. 199–210. Springer, Heidelberg (2016). doi:10.1007/978-3-319-27033-3\_14
18. Reichmann, C.: Grafisch notierte Modell-zu-Modell-Transformationen für den Entwurf eingebetteter elektronischer Systeme. Shaker (2005)
19. Kersten, M.: Eclipse Mylyn Open Source Project (2016). <http://www.eclipse.org/mylyn/>. Accessed 30 July 2016
20. Askund, U.: Configuration management for distributed development in an integrated environment. Lund University (2002)
21. Bruegge, B., Dutoit, A.H., Wolf, T.: Sysiphus: Enabling informal collaboration in global software development. In: International Conference on Global Software Engineering, ICGSE 2006. IEEE (2006)
22. Dig, D., et al.: MolhadoRef: a refactoring-aware software configuration management tool. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications. ACM (2006)
23. Chu-Carroll, M.C., Wright, J., Shields, D.: Supporting aggregation in fine grained software configuration management. In: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software Engineering. ACM (2002)
24. JetBrains: MPS overview (2016). <https://www.jetbrains.com/mps>. Accessed 30 July 2016
25. Collins-Sussman, B., Fitzpatrick, B., Pilato, M.: Version Control with Subversion. O'Reilly Media Inc., Sebastopol (2004)

Software Quality. Complexity and Challenges of  
Software Engineering in Emerging Technologies  
9th International Conference, SWQD 2017, Vienna,  
Austria, January 17-20, 2017, Proceedings  
Winkler, D.; Biffl, S.; Bergsmann, J. (Eds.)  
2017, XII, 189 p. 51 illus., Softcover  
ISBN: 978-3-319-49420-3