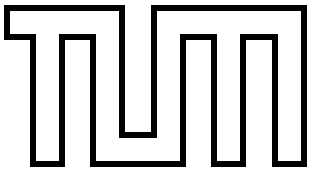


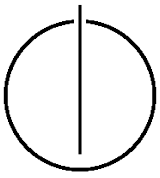
FAKULTÄT FÜR INFORMATIK

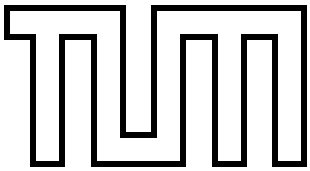
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

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

Jan Finis





FAKULTÄT FÜR INFORMATIK

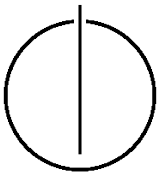
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

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

Einsatz von Nachvollziehbarkeit zwischen Quellcode und Aufgaben für Code Reviews und Freigabemanagement

Author:	Jan Finis
Supervisor:	Prof. Bernd Brügge, Ph.D.
Advisors:	Maximilian Kögel, Nitesh Narayan
Submission Date:	May 18, 2011



I assure the single-handed composition of this master's thesis only supported by declared resources.

Sydney, May 10th, 2011 Jan Finis

Acknowledgments

First, I would like to thank my adviser Maximilian Kögel for actively supporting me with my thesis and being reachable for my frequent issues even at unusual times and even after he left the chair. Furthermore, I would like to thank him for his patience, as the surrounding conditions of my thesis, like me having an industrial internship and finishing my thesis abroad, were sometimes quite impedimental.

Second, I want to thank my other adviser Nitesh Narayan for helping out after Maximilian has left the chair. Since he did not advise me from the start, he had more effort working himself into my topic than any usual adviser being in charge of a thesis from the beginning on.

Third, I want to thank the National ICT Australia for providing a workspace, Internet, and library access for me while I was finishing my thesis in Sydney.

Finally, my thanks go to my supervisor Professor Bernd Brügge, Ph.D. for enabling me to write my thesis at his chair and for providing his Software Engineering book which helped me a lot during the writing of this thesis.

Abstract

The establishment of traceability links from source code to software engineering concerns like features, requirements, and tasks (the so-called *code-to-concern* traceability) is a challenging field of research. The approach of this thesis links *tasks* in the software development project to their resulting *changes in the source code*. In contrast, most existing approaches link other concerns, like requirements satisfied by the code, to the source code. By linking code changes to tasks, a more finely grained traceability is established. Since tasks can be linked to other concerns, this approach can also be used to establish traceability to these concerns as well.

Another point which sets this work apart from previous ones is that the proposed links are maintained manually. In contrast to this, most of the existing approaches use heuristic methods to compute links. Due to their statistic nature, these approaches are rather unreliable. Numerous processes exist which cannot deal with this unreliability and thus cannot be realized with heuristic approaches. Two of them are elaborated in this thesis.

One process benefiting from the proposed links is found in the field of release management: By maintaining links between tasks and their resulting implementation, it is possible to determine which tasks are implemented in a given source code. By linking a release to a set of tasks which should be implemented for this release, the source code from which a release is to be built can be checked against these tasks. Thereby, it can be ensured that a release contains all features which are planned for it. In addition, a changelog can be generated by assembling the description of all included tasks. Ultimately, if the source code changes associated to a desired task are missing in the source code of a release, it is possible to automatically merge them into it.

Another process benefiting from the proposed links is code review: When the results of a task are to be reviewed, a link to these results can be used to quickly transfer the respective changes to the reviewer's machine.

Version control systems like Subversion or Git already store the change history of a software development project. Thus, using them to establish the links is a promising approach. In this work, two different approaches using version control systems for the establishment of code-to-task links are proposed: One uses patches whereas the other one uses branches in the repository to store the set of changes to be linked to a task.

The two approaches are applied to the two aforementioned processes in the fields of release management and code reviews. A tool is designed which performs the mentioned tasks. A prototype of this tool, which is integrated into the UNICASE CASE tool for Eclipse, is developed. Finally, the two approaches are compared and the prototype is evaluated.

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Objective . . . 3
- 1.3. Related Work . . . 5
  - 1.3.1. VCS Repository Mining . . . 6
  - 1.3.2. Source Code Traceability . . . 6
  - 1.3.3. Release Management . . . 8
  - 1.3.4. Code Review . . . 8
- 1.4. Scope . . . 9
- 1.5. Outline . . . 9

2. Preliminaries **11**
- 2.1. UNICASE . . . 11
- 2.2. Version Control Systems & Related Concepts . . . 12
  - 2.2.1. Version Control . . . 12
  - 2.2.2. Version Control Systems . . . 15
  - 2.2.3. Patches . . . 19
  - 2.2.4. Merging . . . 20
- 2.3. Popular Version Control Systems . . . 23
  - 2.3.1. Early VCSs . . . 24
  - 2.3.2. Subversion . . . 25
  - 2.3.3. Git . . . 26
  - 2.3.4. Further Modern VCSs . . . 30
- 2.4. Change Package Representation . . . 30
  - 2.4.1. Patches as Change Packages . . . 32
  - 2.4.2. Lightweight Branches . . . 32
- 2.5. Data Dictionary . . . 34

3. Requirements Elicitation **37**
- 3.1. Functional Requirements . . . 37
  - 3.1.1. Data Model Related Requirements . . . 37
  - 3.1.2. Change Package Related Requirements . . . 38
  - 3.1.3. Repository Related Requirements . . . 39
  - 3.1.4. Further Requirements . . . 40
- 3.2. Non-Functional Requirements . . . 40
- 3.3. Scenarios . . . 41
  - 3.3.1. Scenario “Project Setup” . . . 41
  - 3.3.2. Scenario “Development Workflow” . . . 42

3.3.3. Scenario "Release Building" . . . 42
3.4. Use Cases . . . 43
3.4.1. Use Case *Setup Stream* . . . 44
3.4.2. Use Case *Create Release* . . . 45
3.4.3. Use Case *Assign Work Items to Release* . . . 46
3.4.4. Use Case *Check Release* . . . 47
3.4.5. Use Case *Build Release* . . . 49
3.4.6. Use Case *Resolve Conflict* . . . 51
3.4.7. Use Case *Collect Changes* . . . 52
3.4.8. Use Case *Assign to Work Item* . . . 53
3.4.9. Use Case *Review Change Package* . . . 54
3.4.10. Use Case *Commit Change Package* . . . 55

4. Requirements Analysis **57**
4.1. Entity Objects . . . 57
4.1.1. CASE Objects . . . 57
4.1.2. Revision Handling Classes . . . 59
4.1.3. Release Checking Classes . . . 60
4.2. Boundary & Control Objects . . . 60
4.2.1. Change Package Related Objects . . . 61
4.2.2. Release & Stream Setup Related Objects . . . 62
4.2.3. Release Building Related Objects . . . 63
4.3. Dynamic Behaviour . . . 64
4.3.1. Release Checking . . . 64
4.3.2. Release Building . . . 66

5. System Design **69**
5.1. Design Goals . . . 69
5.1.1. Robustness & Reliability . . . 69
5.1.2. Time & Manpower . . . 69
5.1.3. Adaptability . . . 70
5.1.4. Utility . . . 70
5.1.5. Integration with Existing Systems . . . 70
5.2. Subsystem Decomposition . . . 71
5.2.1. Architecture Overview . . . 71
5.2.2. Selection of Off-The-Shelf Components . . . 71
5.2.3. Subsystems . . . 72
5.2.4. Components (Plug-Ins) . . . 74
5.3. Hardware/Software Mapping . . . 75
5.4. Further System Design Decisions . . . 76
5.5. Boundary Conditions . . . 77

6. Object Design **79**
6.1. The Eclipse Modelling Framework . . . 79
6.2. Adaption of the Analysis Object Model . . . 80
6.3. Extension Points . . . 82

6.4. Displaying Model Elements . . . 84
6.4.1. JFace Viewers . . . 85
6.4.2. EMF Label Providers . . . 88
6.4.3. Decorating Images and Text . . . 91
6.5. Command Execution . . . 93
6.6. Adaption to Different VCSs . . . 94

7. Conclusion **97**
7.1. Evaluation . . . 97
7.1.1. User Acceptance . . . 97
7.1.2. Comparison of the Two Approaches . . . 99
7.2. Summary . . . 100
7.3. Future Work . . . 100

Appendix **101**

A. User Manual **103**
A.1. Setting up a Stream . . . 103
A.2. Creating a Release . . . 106
A.3. Collecting Changes and Creating a Change Package . . . 106
A.4. Applying a Change Package to the Local Workspace . . . 108
A.5. Applying a Change Package to the Remote Repository . . . 109
A.6. Assigning Work Items to a Release . . . 109
A.7. Checking a Release . . . 110
A.8. Building a Release . . . 112

Bibliography **115**

List of Figures **123**

Chapter 1.

Introduction

The introduction of this thesis starts with the motivation, followed by the objective describing what exactly is the goal of this thesis. Afterwards, related work is presented and finally a structural overview of the remainder of this thesis is given.

1.1. Motivation

In Software Engineering, the Software configuration management (SCM) is the discipline of managing the evolution of large and complex software systems [1]. According to the IEEE standard [2, 3], SCM covers functionalities such as identification of product components and their versions, audit and review (quality assurance functions), status accounting (recording and reporting the status of components and change requests), and change control (by establishing procedures to be followed when performing a change). Practicing configuration management in a software project has many benefits, including increased development productivity, better control over the project, better identification and bug fixes, and improved customer goodwill [4]. To handle the ever increasing project complexity, the efficient handling of SCM requires tool support. There is good tool support for many tasks of the SCM. Standard SCM tools include bug and issue trackers (e.g. Bugzilla [5]), build tools (e.g. Make [6] or Apache Ant [7]), and version control systems (e.g. CVS [8], SVN [9], Git [10]). Other SCM activities still lack tool support. Two of them, the code review and the releases management, are in the focus of this thesis.

One task of the SCM is the software or code review[1]. Code review is a process or meeting during which source code written by a developer is presented to a reviewer (often another developer of the project) in order to check the source code for errors or other issues. In general, reviews are believed to provide a cost-effective means for discovering defects in any document produced during the software life cycle. They can provide a great increase in both productivity, by reducing development time, and product quality, by removing more defects than is possible without using reviews. However, reviews are, by their very nature, a labour-intensive process [11]. Because of this, tools for supporting the review process are very important. However, only few tools exist as of now. Obviously, the problem is not overwhelming tool complexity, as the tasks of such a tool are rather simple: It has to enable the reviewer to quickly transfer the changes to be reviewed to his machine in a form that allows running the code. The tool should also highlight where which changes

[1] Note that there will be no distinction in this thesis between code reviews, inspections, and walkthroughs. This is because all of them need the features described and it is irrelevant for our purposes which type of review is actually carried out.

were performed. After the review, it should allow the reviewer to approve or decline the changes and save this decision in the project management documents. Version control systems (VCSs) are a common SCM tool which allows to retrieve a set of changes. Thus, this part of the tool support for reviews already exists. However, the problem is the missing traceability between the project management documents, which contain the task to be reviewed and the VCS containing the implementation of the task. A good tool aiming at supporting the review process would enable the reviewer to simply state which task he wants to review. The system should make all arrangements to present the changes without accessing the VCS directly.

Another important sub-discipline of the SCM is release management. In the course of project development, release management is responsible for deciding in which configuration a product is released and which features it includes. Van der Hoek et al. [12] describe release management as a "poorly understood and underdeveloped part of the software process". This is especially true for one task: Making sure that the source code from which the release is built really includes all features the release should embody. However, most literature about release management does not even mention this task. Instead, it is assumed that a configuration of source code already exists and certainly contains all anticipated content. Not even a common term exists for this task, yet. In the remainder of this thesis, it will be called *release content control* (RCC): RCC is the task of defining which features are desired in a specific release, identifying which features are contained in the source code configuration from which the release will be built, and checking if these two sets of features match. The following important questions arise in conjunction with RCC:

1. Is the implementation of each desired feature of this release included in the source code? Or is it still not finished, or is it finished but only on the local hard disk of the developer who implemented it?

2. Are there any other changes in the source code, which are not included in the list of features? Examples could be undocumented bug fixes or accidentally introduced changes.

3. Was the implementation of every feature reviewed and tested thoroughly?

Although more and more effort is spent on the research on release management, only minor interest is put into the research on the first two questions in particular. Once a tool can decide which features are included in the source code of a release, one can demand even more: The tool should be able to build a specific release by assembling the source code of the release autonomously. By specifying the base version of source code and a set of features to be included into the release, the system should be able to merge the implementation of all features into the source code of the base version. Of course, features which are already included in the base version are omitted in this step. Another feature, which is easy to implement if RCC tool support is used, is the tracking of changes in the release: The release management must keep track of which new features the user can anticipate in this release and publish a list of these features. This is often accomplished in the form of a *changelog*. The changelog contains the list of changes included in the release and is usually shipped with the product and/or published online. By reading the changelog, a user working with an older version can decide if it is worth updating to the

latest version. If RCC is used, the changelog can be assembled automatically by the system, as the system is able to identify the set of features contained in the source code.

The implementation of a tool providing RCC support depends on the possibility to determine whether a feature is contained in the source code. To achieve this, traceability between the source code and the features is needed. Thus, tools for the review process support and the RCC support depend on the following challenge: The establishment of traceability links between source code and *concerns*[2] like features, tasks, or requirements (the so-called *code-to-concern traceability*). The fact that tools providing RCC support are hardly available may be founded in problem that traceability links which are reliable enough to enable RCC are hard to establish.

Many approaches exist to establish code-to-concern traceability. Most use heuristics like latent semantic indexing [14], vector space indexing, or probabilistic indexing [15]. The problem with all these techniques is that they cannot provide links which are reliable enough for RCC: There is a certain, yet too big, likelihood for omitting links or yielding wrong links. Thus, reliable code-to-concern traceability is a topic which still requires research. As shown above, it has at least two important applications in the field of software configuration management and may have many more.

1.2. Objective

In this thesis, we propose two techniques to establish code-to-concern traceability (i.e. traceability between project management concerns like features, bug reports, requirements, and tasks and the source code which implements them). In contrast to many previous works about this topic, our approach links the exact changes which were made in the source code related to a specific concern. This is achieved by using a version control system which tracks the changes. We apply the techniques to two specific software configuration management applications: The support of the code review and the release management process. A tool which uses the techniques to aid these processes is designed and a working prototype covering all important use cases is implemented. The tool is to be integrated into UNICASE [16]. This is a computer aided software engineering (CASE) tool which embodies a unified model, allowing to manage all aspects of software engineering in one model and to maintain traceability links between model elements from different sub-disciplines. By linking tasks with source code changes, the gap between management artifacts and concrete implementation is narrowed.

To be precise, the traceability link to be established is between *changes* in the source code and *work items*. A work item is the generalization of any task or set of tasks in the course of a software development project. Thus, this work actually focuses on *code-to-task traceability*. Changes in the source code are to be related to a work item by an *is-the-result-of* relation. Work items can be linked to other artifacts like requirements or use cases to reach full traceability between code and various software engineering concerns.

The tool allows to specify releases and their desired content. This is accomplished by selecting which work items, or more precisely the resulting changes thereof, are to be included in a selected release. By using the is-the-result-of relation and the assignment

[2] Robillard defines *concern* as "any consideration that can impact the implementation of a program"[13]. Almost any software engineering artifact has impact on the implementation, so this is a very general notion.

of work items to a release, the link between the software development domain (source code changes) and the software configuration management domain (releases) can be established. The core idea to generate the links is letting the developer build the links himself, using the version control system of the project: Whenever he finishes the implementation of a work item, the developer does not directly commit the changes to the repository. Instead, he orders the system to collect the changes and associate them to the work item. The system then collects the changes made in a so-called *change package* and attaches this package to the chosen work item in the unified model.

To establish the traceability links, the system must provide the following three services which operate on change packages: A change package must be creatable from the current changes the developer has in his workspace, the changes in the package must be applicable to a given set of source files, and it must be possible to determine if the changes in a package are already applied to a given source code. Finding an appropriate representation of change packages which allows efficient implementations for these three services operating on it is the core research part of this thesis.

Using the change packages and the three services, the proposed tool is able to verify, which work items assigned to a release already have their changes contained in the source code at hand. Thus, it can inform the release manager which change packages need to be applied to the code before the release can be published. The tool goes one step further by automating the process of assembling a release: The tool autonomously applies the changes associated to work items, if these changes are not yet contained in the source code. The ultimate goal is to assemble the source code for a release by just pressing a "build"-button and letting the system perform all the work.

The second main purpose of the tool is to support the review process. For this purpose, it is assumed that source code is reviewed and tested on the work item scale, i.e. the source code changes belonging to one work item are to be reviewed an tested[3] at the same time. The tool supports retrieving the source code changes associated to a work item and applying these changes to the local workspace of the reviewer. With the applied changes, the reviewer can instantly start reviewing and testing these changes.

As already mentioned, the tool represents change packages and implements the services operating on them by using a version control system. Since these systems already offer features like extracting and applying changes, maintaining a version history, and fetching the source code in a specific configuration from the repository, they are predestined to be used for tracking changes in the source code. In addition, version control systems are also widely accepted and used in the vast majority of existing software development projects.

The core of the prototype is kept generic: The way the change packages are actually represented and the services are implemented is factored out to adapter plug-ins. These plug-ins connect the core plug-in with the version control system. Thus, the different approaches to representing change packages using different version control systems can be exchanged and adapted to the version control system used for a specific project.

The prototype is developed as part of the UNICASE project, which is developed and maintained at the chair for applied software engineering at the TU München. The UNICASE project is an open source CASE tool for the Eclipse [17] platform. It features a unified

[3] Of course, tests on a larger scale, like integration tests and system tests, are required as well. These, however, are not in the scope of this tool.

model to embody data from different fields of software engineering. The prototype plugs into this model and enhances it by adding all the model elements necessary for release management and review tasks. By using the Eclipse platform, the tool is seamlessly embedded into the different integrated development environments (IDE) Eclipse offers. A prominent example is the Java development tools (JDT) environment for Java. In addition, the tool uses the team provider interface of Eclipse to identify and access the version control system used for a project and is thus able to decide which adapter plug-in to use for a given project, based on which VCS is registered as team provider for this project.

After the prototype is developed, its functionality is to be evaluated. For this reason, a small user study to test the acceptance of the prototype is to be conducted. Finally, the two approaches will be evaluated and compared.

In the remainder of this thesis, the terms *tool*, *system under development*, and *prototype* all refer to the release management and review automation tool developed within the scope of this thesis.

1.3. Related Work

To cut things short: No previous work covers the full spectrum of this thesis. The reason is that this thesis explores theoretical mechanisms of achieving code-to-task traceability while also providing a concrete application for the traceability. Related work covering these aspects is mainly split into two parts: The first part covers the fundamentals and techniques for maintaining traceability links without elaborating very concrete applications. The second part researches the possibilities to aid or improve the review and release management process. These works, however, are often rather high level surveys not proposing specific techniques but rather identifying main challenges in these fields. They also propose possible solutions, but often only on a very coarse scale without elaborating a detailed solution.

The following four fields in which research is done are related to the work of this thesis. While the first two cover fundamentals of maintaining traceability and using version control systems, the second two focus on aiding the concrete applications.

VCS Repository Mining This field of research focuses on mining and utilizing data from the repositories of version control systems.

Code Traceability Here, research is done on different ways of relating source code to other development artifacts like requirements and use cases.

Release Management This field examines and tries to improve the release management process.

Code Reviews This field deals with code reviews and their benefits to software engineering. It includes processes and tool support for aiding their efficient realization.

The reason why no work covers the full spectrum of the content of this thesis may also lie in the fact that using a unified model with traceable links between different software engineering and project management entities is rather uncommon. Because this work is

built on UNICASE, which embodies the philosophy of unifying the different models found in software engineering, it is natural to it to combine the different fields.

The research in the field of code traceability is the most important one for our approach, because maintaining links between source code changes and work items constitutes the most complex problem. If these links can be realized efficiently, the rest of the approach turns out to be just a well thought-out usage of them. Therefore, the most related work mentioned here is from this area.

1.3.1. VCS Repository Mining

The mining of different data from the repositories of version control systems is a huge topic in the field of data mining. Most work done in this field, however, focuses on retrieving statistical information. Although the retrieval of statistical information is natural to the data mining community, it is not relevant for the approach of this thesis: Our approach uses version control systems to establish more reliable traceability links. In contrast, statistical data is not able to provide links which are reliable enough for the proposed appliances. Since most of the work is only partly relevant, few sources are to be mentioned here. Instead, at this point reference is made to Kagdi et al. [18], who conducted a very detailed survey providing an overview and identifying important papers in this field of research.

Zimmermann et al. [19] used version histories in CVS to establish links between different changes in source code and built a recommender system for developers. Although the results are promising, they are still only statistical and thus yield missing links and false positives. Similar approaches, which also use version data to establish links between source code entities, were made by Gall et al. [20, 21].

Fischer et al. [22] link version control systems with release data, but go the opposite direction than 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.

1.3.2. Source Code Traceability

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

A very simple, yet effective approach is presented by Storey et al. [23, 24, 25] embedding the links directly into source code comments, which can be read by their proposed tool TagSEA [26]. They use tags in comments to refer to other artifacts or points of interest. They go further into the direction of this thesis by creating links between tasks and source code locations [27]. This is accomplished by connecting their tool with MyLyn [28], which can be used to express tasks. They connect source code to MyLyn tasks by using special tags.

Markus et al. [14, 29] establish links between source code and the corresponding documentation. In contrast to the aforementioned approach, they do not rely on any kind of

explicit link markers but try to recover the links automatically. This offers the great advantage that it is also applicable for legacy projects. They use the so-called latent semantic indexing (LSI, a machine learning technique) to extract parts of the semantics of the source code and the documentation. By using similarity measures operating on these semantics, locations in the source code can be linked to locations in the documentation.

A lot of research in this field has been conducted by Antoniol et al. Their first approaches links source code locations to object-oriented design artifacts [30, 31, 32] and track the links over software releases [33]. In their following contributions, they also establish links between code and free text documents (like user documentation and requirement specifications) [15]. Finally, they also treat code-to-concern traceability [34, 35, 36]. Like Markus 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 and source code identifiers and comments. Antoniol et al. also compare the performance of these methods in [15]. Additionally, they claim the poor performance of latent semantic indexing techniques, as applied by [14, 29]. Approaches using similar techniques are made by Zhao et al. [37, 38] and De Lucia et al. [39].

A very interesting approach is first made by Wilde et al. [40]. They use test cases and code coverage tools to map source code locations to features: By associating a test case to a feature, the lines executed by this test case can be associated to the feature. This is called execution-trace analysis. Of course, a line executed by more than one test case can belong to various features. Eisenberg et al. [41] use a similar approach and focus on weighting the influence of code locations onto features appropriately, thus retrieving more precise results. Markus et al. [42] combine their aforementioned approach using LSI with the execution-trace analysis, thereby increasing accuracy in comparison to previous results.

Grechanik et al. [43] use program analysis, run-time monitoring, and machine learning to establish traceability links between source code locations and use case diagrams. In their approach, the developer explicitly defines a small set of starting links, with the system then trying to infer further links. They implemented their technique as an Eclipse plug-in.

Use-case-to-code traceability is achieved by Omoronyia et al. [44]. In addition, they also trace which developer links together which artifacts. Their approach is based on tracing the operations carried out by a developer. Thus, their approach is also able to identify which developer is involved in the realization of a specific use case. This approach is more similar to ours than the aforementioned ones, because it tracks operations (i.e. changes), similar to our approach. Their contribution shows that tracking changes displays some advantages over the other approaches. For example, relating a developer to the source code and use cases is almost impossible with the other approaches, but very easy if changes are tracked.

Robillard et al. [45, 46, 13] contribute by suggesting a representation of source code location information linked to a specific concern. While most other approaches link lines of code, files, or changes to concerns, they introduce so-called concern graphs as a representation of source code locations linked to a concern. They concluded that their representation is compact, simple, expressive and thus appropriate for describing a code location linked to a concern.

1.3.3. Release Management

Van der Hoek et al. [12] identify basic issues in release management and develop a tool to aid the release management process. However, their approach does not include assembling and building components to be released. It is merely a database to which releases, components to be released, their dependencies among each other, and meta data can be added. They also identify the challenges of release management for component-based software[47].

Michlmayer et al. [48, 49] investigate the specific challenges posed by release management in open-source projects. Erenkrantz [50] does similar research evaluating the challenges in popular open source projects like the Linux kernel and the Apache HTTP server.

Saliu and Ruhe et al. [51, 52, 53, 54] contribute research in the field of release planning, especially for evolving systems which are built incrementally. This field is concerned with the selection and assignment of features to a sequence of consecutive product releases in order to meet the most important technical, resource, budget and risk constraints. They propose a release planning framework and evaluate it against previous approaches to release planning.

1.3.4. Code Review

The review process was introduced as "software inspection" by Fagan [55] in 1976. A good starting point for review technologies and their progression are the surveys by Laitenberger et al. [56, 57] and Aurum et al. [58].

The benefits and impacts of software reviews were examined by Laitenberger [59], Siy & Porter et al. [60, 61, 62], Kemerer et al. [63], and Mäntylä et al. [64]. Siy & Porter focus on the costs and benefits of reviews, on the parameters which may alter the performance of reviews, and on identifying in which situations reviews may yield value. Laitenberger also examines the cost-to-benefits consideration while comparing reviews to structural testing techniques. Kemerer et al. assess the impact reviews have on software quality. Finally, Mäntylä et al. examine which types of errors are commonly found by code reviews.

Review processes are examined and proposed by Dunsmore et al. [65, 66, 67], and Rigby et al. [68]. While Dunsmore et al. propose processes especially for reviews of object-oriented source code, Rigby et al. examine review processes in open-source projects.

Brothers et al. [69, 70, 71], Harjumaa et al. [72], and Belli et al. [73] contribute to tool support for the review processes. Brothers et al. propose the ICICLE tool — a tool for reviewing C code. ICICLE 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. propose a web based tool. It features the distribution of the document to be inspected, annotation of it, searching of related documents, a checklist, and inspection statistics. Belli et al. describe an approach for the automatic handling, checking, and updating of check lists used in reviews.

Macdonald et al. make further contributions focusing on tool support: They start with a survey about existing tools [11] and then describe a generic process notation which can be used to model a review process and use this model as input for a review support tool

which then works with the defined process [74]. Next, they discuss the features which could be provided by computer support for inspection and the gains that may be achieved by using such support [75]. They finish their research with an updated survey, which in detail compares the existing solutions for review management [76].

1.4. Scope

The aim of this thesis is to propose a tool to support the release management and review process in the ways described in Section 1.2. Only the aspects of the processes which are mentioned therein are to be supported by the tool. All other aspects of these processes are out of the scope of this thesis.

For reviews, this means that this thesis will not cover planning of reviews, review processes, evaluation of review benefits and impacts, and tool support which covers other aspects than the one mentioned in the objective.

For release management, this means that this thesis will not cover build management[4], release planning, delivery and deployment of release content, advertising of releases, and release techniques.

Concerning version control systems on which the approaches in this thesis are built, only centralized topologies with one global repository are considered. Because distributed version control systems allow this topology as well, this is *not* a restriction to the VCS used, only to the VCS topology a project uses (cf. Section 2.2.2). Although the approach of this thesis would probably also suit distributed topologies, this was not further investigated.

Due to the limited time scope of this thesis, the long term evaluation of the prototype in real software projects cannot be part of it. However, this definitely is a source for future work (cf. Section 7.3).

1.5. Outline

In this thesis, two approaches for establishing source-code-to-task traceability will be presented. A prototype applying these concepts to the release management and review process will be designed and evaluated.

Chapter 2 starts with elaborating topics which are the basis for the prototype. First, the UNICASE tool, on which the prototype will be built, is explained. Next, version control concepts and important version control systems are presented. Finally, the two approaches for establish code-to-task traceability links, which are the main theoretical contribution of this thesis, are presented.

Afterwards, Chapter 3 features the requirements elicitation for the tool to be created. Here, the functional and non-functional requirements are presented. In addition, three comprehensive scenarios describing the requirements of the system are shown, followed by the use cases which were extracted from them.

After the requirements elicitation, Chapter 4 analyzes the gathered requirements. A static model containing the entity classes of the problem domain is deduced. Subsequently,

[4] Although the term "building a release" is also used in the remainder of this thesis, it does not refer to build management, i.e. compiling and packaging sources. Instead, it refers to assembling the source code configuration which will be the input for the build management.

this model is enriched by adding boundary and control classes for the execution of the use cases. Finally, the dynamic model of some use cases is depicted.

Chapter 5 covers the system design of the prototype. The chapter starts with the design goals which will influence the following design. Afterwards, the general architecture of the prototype is shown. Based on this architecture and the design goals, the prototype is decomposed into subsystems which are then mapped to components. These software components are in turn mapped to hardware. Finally, further system design topics like the definition of the global control flow, exception handling, and boundary use cases are described.

Next, Chapter 6 covers the object design of the prototype. The focus is laid on presenting interesting parts of this design instead of covering it in whole. First, a short introduction about using the Eclipse Modelling Framework (EMF) [77] is given. The final extension for the unified model of UNICASE, as modelled with EMF, is shown. Afterwards, the concept of Eclipse extension points is depicted. Finally, strategies for the handling of commands working with the unified model, the appropriate displaying of model elements, and the adaption to different version control systems are presented.

At the end of the thesis, Chapter 7 draws a conclusion of the work. Here, the two approaches are compared, followed by an evaluation of the prototype. Ultimately, the content is summarized retrospectively and topics for future work are presented.

Chapter 2.

Preliminaries

This chapter presents and further describes topics which are the basis for the approach of this thesis. The chapter starts with a presentation of the CASE tool UNICASE into which the prototype will be integrated. Next, an overview over version control including the main concepts and a description of version control software is given. This presentation of the theoretical fundamentals is followed by a survey of the most widely used version control systems. Afterwards, the two approaches of this thesis are elaborated. Finally, a data dictionary is presented, which defines the most important concepts used in the remaining chapters.

2.1. UNICASE

As mentioned in the introduction, UNICASE is a CASE tool realized as a plug-in for the Eclipse platform. Its aim is to integrate the products and documents of different development activities into one unified model, thus allowing cross referencing between them. For example, a use case can be linked to the functional requirements related to it.

UNICASE consists of a client which allows the graphical creation and editing of the unified model. It also provides different tools for viewing the model from different perspectives, thus allowing different monitoring activities to be executed efficiently. The second part of UNICASE is the server, which is called EMF store. The server allows to share UNICASE projects. Once a project is shared, the server provides version control of the project model, allowing to view the revisions of the model and also reverting changes done to it. Thus, the EMF store can be treated as a version control system for EMF models.

Currently, the unified model of UNICASE embodies models for the following development activities:

Requirements Modelling Functional and non-functional requirements can be specified and use cases can be described accurately.

UML Modelling The plug-in supports UML diagrams like class or use case diagrams.

Bug and Feature Tracking UNICASE provides support for tracking bug reports, issues, and feature request.

Integrated Project Management UNICASE allows the specification of a work breakdown structure, iteration planning, project status visualization and limited review support: Different types of tasks and groups of tasks can be specified and assigned to developers. Additionally, a reviewer can be assigned to a task. The status of the task can

be tracked and the priority and estimated effort can be set. Different views visualize the progress of the tasks.

The prototype is developed as an extension for UNICASE, adding support for release management and extending the support for the review process. The unified model is extended by adding the model element types for concepts relevant to release management (like releases and change packages). The concept of work items already exists in the unified model. The work item concept is enhanced by adding the is-a-result-of association, which links work items to change packages and allows the association of work items with releases in which the products of these work items should be contained. The exact model changes are shown in Section 6.2.

2.2. Version Control Systems & Related Concepts

The proposed approaches for representing changes are built upon version control systems. To provide a basis for concepts used in these approaches, this section will explain basic version control concepts. This includes an introduction of version control in general, version control systems as an implementation of version control, the description of the patch concept, and finally a more detailed insight into the process of merging in version control systems.

2.2.1. Version Control

Version control, *revision control*, *software versioning*, or just *versioning* is a sub-discipline of software configuration management. According to Leon [4], version control is the act of tracking the changes of a particular file [or project] over time. Its task is to manage different versions of the source code of a project. A *version* of a project depicts a snapshot of all source code files at a given time and configuration. Close to the term of a version is the term revision: The oxford dictionary defines revision as "a revised edition or form of something" [78]. In the field of *version control* the term *revision* refers to a version which was created by altering another version. Here, the term revision is often used as a synonym of the term version. Since every version is created by altering another version (with the first version being created by altering a virtual, empty version), the use of the term revision for any version is reasonable.
Keeping versions has the following advantages:

- The source code of published versions (releases) and other important versions is archived so any release can be re-built if necessary.
- Destructive changes in the source code (like the accidental deletion of parts of a file's content) can be amended by recovering an older version, even if the destructive changes are already stored in the repository.
- The changes implemented at a specific moment by a specific person can be reproduced. This requires very fine grained versioning.

- The existence of more than one development *branch* can be managed. For example, while a new major release is already under development, changes to the current release (e.g. bug fixes) can still be introduced and published, which is quite common for larger projects.

Development Histories & Revision Graphs

In the simplest case without multiple development branches, the developers work only with the latest version. Thus, any newly created version is the successor of the latest version at that point. In this case, a version has always only one direct successor, with the exception of the latest version which has no successor yet. An expressive method for displaying version histories graphically is a *revision graph*. The graph represents the version history by treating versions as nodes and the "is-successor-of" relation as edges. The nodes are usually ordered by creation time to keep the graph clearly laid out. In the easy case of a linear development, where the latest revision is always created from one predecessor, such a graph would be a linear series of nodes. The upper part of Figure 2.1 shows such a revision graph.

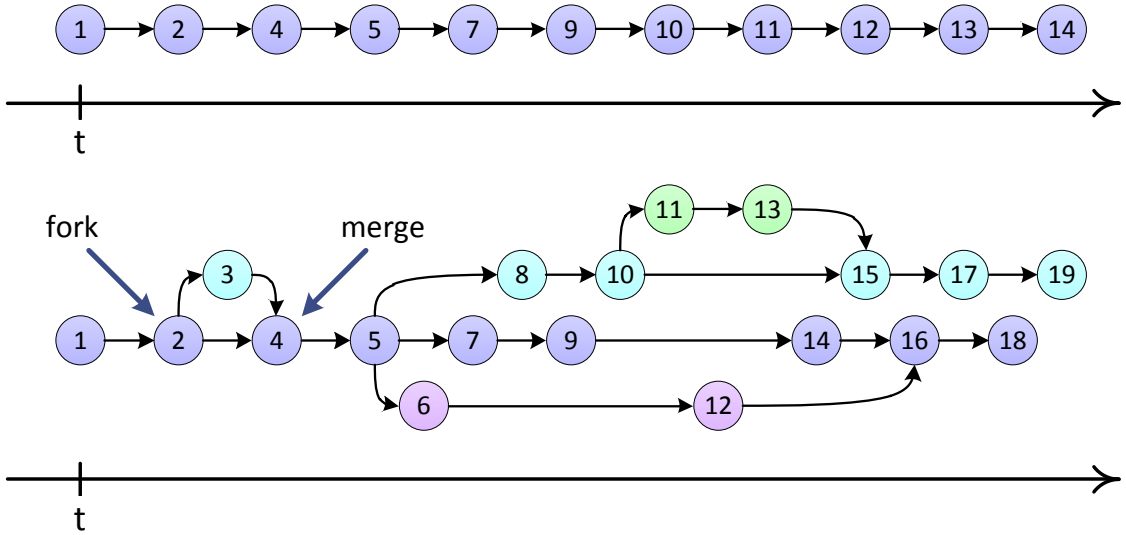


Figure 2.1.: Revision graphs for linear development (top) and with branches (bottom).

Once multiple development branches are introduced, a version can have more than one successor. This is the case if a new development branch is started from that version. A branch starting at a specific version (which belongs to another branch) is also said to *branch*, *diverge* or *fork* from the other branch at that version. The changes done to two development branches can be recombined, which is usually called *merging* the two branches. The term "merge" is used because merging is what has to be actually performed: Merging two branches is achieved by calculating all changes on the two branches since the version at which they forked. The changes of one branch are then combined (merged) with the ones of the other branch to achieve the recombination of the branches. If branches are used, the revision graph becomes a directed acyclic graph (DAG). This is shown at the lower part of

Figure 2.1: A version has an additional successor for each additional branch which forked from it. A version originating from a merge of two or more branches has as many predecessors as branches were merged. Because the "is-successor-of" relation is consistent with the temporal order of versions, no circles can exist (thereby yielding a DAG).

An example for more than one development branch is UNICASE, which is developed on an internal and an external branch. The latest changes are commonly added to the internal branch. From this branch internal releases are built and tested internally (hence the name "internal"). The external branch contains only features which have proven to work flawlessly. From this branch external versions are released to the public.

Note that versioning does not necessarily mean that a version control system is used. However, version control systems are indeed used in most modern projects. Some operations like merging two development branches are very complicated without a version control system (since they require the calculation and merging of changes which appeared since the divergence of the branches). For linear version hierarchies without branches, simply copying all artifacts of the project to a safe destination to create a version would be a possibility to accomplish versioning without using any tool support, but it is cumbersome and uses much disk space if many versions are maintained. Thus, versioning without tool support is very uncommon, and even small private software development projects often rely on version control systems.

Repositories

One of the most essential items for any software development project is a repository. Although today the term is often used in the context of version control system, a repository in general is just a place where the versions of source code of a project are stored. Even if no version control system is used, a repository usually exists to allow collaboration between developers. It can be a folder on an ftp server or even on the local machine of one of the developers. The repository has the following functions:

- It is the safe spot for the versions. If the data on a local machine of a developer gets lost, then he can re-fetch the source code from the repository. Thus, the repository should be protected against hardware failure and other causes of data loss. This can be achieved by mirroring the artifacts on different machines, creating regular backups and using redundant hardware (e.g. RAID [79]).
- The repository is the main source for collaboration between developers as changes to a file are shared with other developers by updating this file in the repository.
- New developers joining the project or established developers working at a new machine can obtain the latest versions of the source code from the repository.
- The code in the repository is used to build releases.
- Depending on the type of the repository, it can offer additional services like versioning. This is especially the case for repositories managed by version control systems.

Although it is assumed here that a project has only one repository, other forms of development with more than one repository exist as well. While this case is quite unknown for commercial products, it is used quite frequently for open source projects.

2.2.2. Version Control Systems

A Version Control System (VCS) is a tool for software development which controls the different versions of the source code. Of course, non-source code files can be versioned by a VCS as well. However, since source code files are the most common files found in VCS repositories, the remainder of this thesis will only use the terms *source files*, *source code* or simply *code*. Whenever one of these terms is used, it actually refers to any possible file to be versioned. The basic mechanism of each version control system is that it features one or more repositories in which the revisions are stored. The copy of the source code the user has on his local machine and to which he applies the changes is called the *working copy*. The basic operations of a version control system consist mainly of transferring data from the repository to the working copy or vice versa: The user can retrieve revisions from the repository and upload the changes in his working copy to the repository, thus creating a new revision.

There is a set of basic commands which is more or less supported by every version control system: The creation of a new revision in the repository by uploading the changes in the working copy is usually called *commit* or *check-in*. A developer commits his work after he has implemented and tested a feature. In the distributed case, where the developer has a private repository on his own disk, he can commit during the implementation of a feature even if the code contains errors. Most version control systems allow to reproduce every single revision which was committed to the repository. However, some systems allow the pruning of older revisions which are no longer needed. The operation which retrieves a certain revision from the repository and writes the data to the working copy is usually called *check-out* or *fetch*. Most newer version control systems allow the divergence to different branches and the merging of those, as described in section 2.2.1. Some of the systems maintain a special main branch called *trunk* while others treat each branch equally. For one branch, the latest version is usually called the *head* revision of this branch. Most of the time, developers are only interested in checking out the head revision of a branch to retrieve the latest changes. Doing so is called *updating*. Most version control systems also provide utilities to compare to revisions, thus identifying the changes made. Another feature provided by most version control systems are *tags*. A revision can be tagged with a specific name to find it later on in the large number of revisions. Thus, tags are used to mark special revisions, e.g. the ones from which releases were built.

Repository Data Representation

The naïve algorithm of creating a new revision in a repository would be to transmit all files to the repository and save a complete copy of all files. This, however, is unacceptable, because it would require a tremendous amount of disk space and the commit would take a very long time if the bandwidth of the connection to the repository is limited. Therefore, version control systems use mechanisms to reduce the amount of space and network traffic required.

One possibility to reduce traffic and disk space usage is to store only the differences between two revisions, which is done by the subversion VCS, for example. The differences between two successive revisions are called *delta* in this context. There are two directions for representing versions as deltas: In the case of *forward deltas*, the differences from the

predecessor to the successor are saved and the first version in the repository is saved as plain text. A version can be constructed by starting at the first version and walking the revision graph to the desired version. While walking, the forward deltas are applied one by one. The other direction can be implemented using *reverse deltas*. In this case, the most recent version of each branch is saved as plain text and reverse deltas depicting the changes from successor to predecessor are used to retrieve older versions. The reverse delta approach has the advantage that the latest version, which is needed most frequently, can be reconstructed without applying any deltas. The drawback of this method is that the latest version saved in plain text has to be updated with each commit. In contrast, the forward approach always saves only the first version, which never changes, as plain text.

Another approach to reduce disk space is to store the files not by name but by content: By creating a hash from each file and saving the file using the hash as file name, two files containing exactly the same content will not be saved twice, even if they reside in different revisions or directories, or have different names. Consequently, all files that have not changed between two successive revisions do not have to be saved twice. This approach is used by Git, for example.

To ensure data integrity, most version control systems provide guarantees known from database management systems. For example, most version control systems guarantee parts of the ACID (atomicity, consistency, isolation, durability) properties for any transaction (like commits): Either a commit is executed thoroughly and successfully, or the commit is aborted and no change is made to the repository (atomicity). No two people can commit concurrently, or if they can, the system guarantees sequentially consistent semantics (isolation). Some version control systems also protect against other threats to data integrity like malicious changing of data.

Centralized vs. Decentralized

Version control systems can be divided into two categories: If one central repository is stored on a globally accessible server, the VCS is labeled *client-server* or *centralized*. The abbreviation is *CVCS* (centralized version control system). If each developer usually has a local repository on his machine, the VCS is called *peer-to-peer*, *distributed*, or *decentralized*. The abbreviated term *DVCS* (distributed/decentralized version control system) is also quite common. In the distributed case, data is exchanged by pulling or pushing data from one repository to another one. Finally, there are also *local* version control systems which do not support networking. However, systems of this type have become obsolete; all recent version control systems support networking in some manner.

Figure 2.2 shows the conceptual differences between centralized and distributed version control systems. In the centralized case on the left, there is one global repository for a project. Developers retrieve the latest version by checking out or updating from this repository. Once they are done with their changes, they commit back to the repository. In the distributed case on the right, each developer maintains his own local repository. Data is transferred between developers by either *pulling* data from another developer or *pushing* data to another developer. The pull strategy is more common because it requires only read permissions on the other developer's repository while pushing would require write permissions. As shown in the figure, dedicated repositories can exist in the distributed case as well, but there is no need to have one, or a project can have more than one.

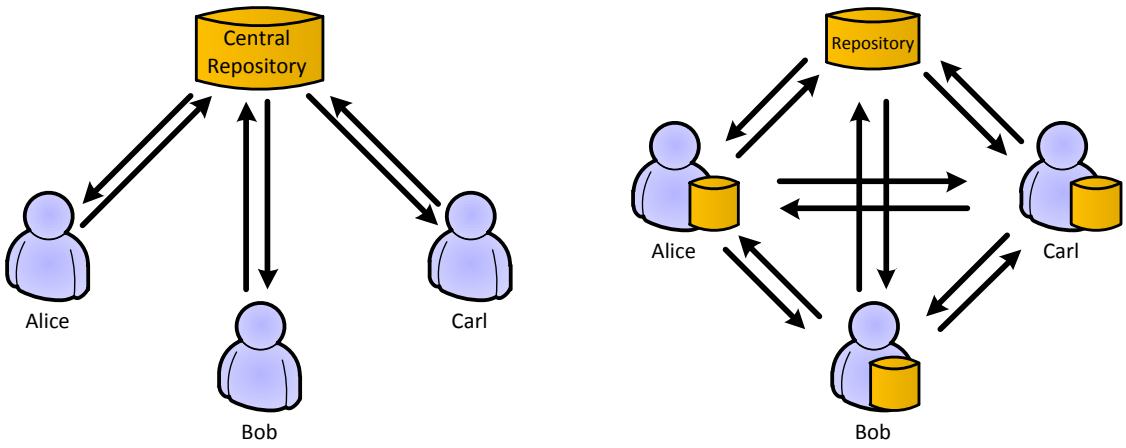


Figure 2.2.: Centralized (left) vs. distributed (right) version control.

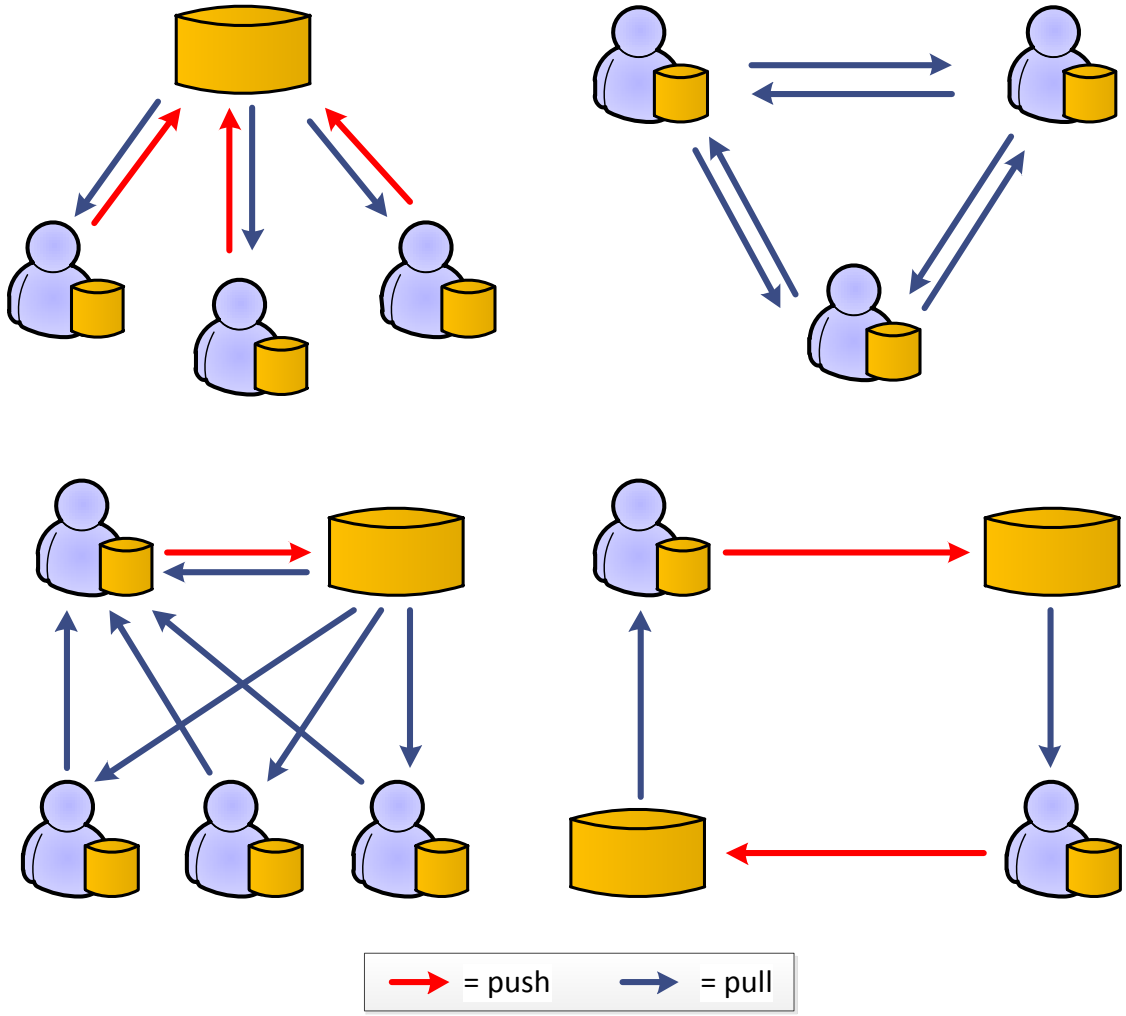


Figure 2.3.: Different topologies when using a DVCS.

The distributed case allows arbitrary topologies for a project. Some of them are exemplified in Figure 2.3. Pull operations are shown as blue arrows and push operations as red arrows.

In the upper left example, the centralized case was emulated with one central repository to which every developer pushes and pulls. This shows that the centralized case may even be regarded as a special case of the distributed one. But still, developers have their own local repositories, which allows them to use version control even if they are offline.

Another topology, shown in the upper right corner of the figure, is the pure peer-to-peer case where no central repository exists and a loose network of developers pulls from each other, which is often used in smaller open source projects.

For larger projects, the topology shown at the lower left corner may be used: A hierarchy of developers exists. Although only one hierarchy level is shown in the example, more may exist. The developers higher in the hierarchy retrieve work from developers lower in the hierarchy by pulling from them. They incorporate the changes into their local repository from which the next hierarchy level pulls and so on. The developer(s) at the topmost level have write access to a global repository and push their changes to the repository. From this repository, developers update their working copies to the latest version.

In the final example, as shown in the lower right corner of the figure, each developer has an own private local repository and a dedicated global repository. Other developers have read permissions for the global repository. The developer does changes to his local repository. Once he wants other developers to see his changes, he pushes them to his global repository from which other developers can pull the changes.

The cases shown here are only examples; any imaginable project structure can be constructed with distributed version control systems.

While the centralized version control systems like CVS and Subversion were more prominent in the 1990ies, most recent version control systems are distributed. According to Alvis and Sillito [80], there is a trend of moving from centralized to distributed version control systems, especially in open source software (OSS) development. By examining different OSS project communities, they identified the following reasons for this trend:

First class benefits to all users Even developers who do not have commit access in an OSS project can track their changes while CVCSs often caused non-committers to open parallel central repositories to track their changes.

Simple automatic merging DVCSs usually have a better support for automatic merging and thus allow the easy maintenance of many branches which can be merged back easily into the trunk later on.

Improved support for experimental changes With the cheap local branches offered by most DVCSs, experimental changes can easily be made by creating an experimental branch. This is unimaginable with CVCSs like Subversion.

Support disconnected operation A big advantage a local repository offers is that most operations can be performed locally. This increases responsiveness due to missing network delay and allows to access the repository without an Internet connection available.

2.2.3. Patches

A very simple form for collaboration between developers are patches. A patch is a file depicting the changes made to one or many source files. Because it is usually very small in size, it can be sent by email and is thus very handy to quickly exchange modifications to the source code. This form of collaboration is particularly often found in open source projects where not every contributor has direct access to the central repository. Such projects usually have a publicly available mailing list to which patches can be sent. If, for example, a contributor without commit access fixes a bug, he emails a patch containing the bug-fix to a committer of that project or uploads it to the project's bug tracking system. The committer can then apply the patch onto his version of the source code, review it, maybe make additional changes, and then commit the changes to the repository. It was also the method of choice in the days before most version control systems were invented, since patches can be created with basic terminal commands in Unix systems which exist since the 1970ies. Although many version control systems exist today, the method is still widely used in open source projects [81].

Patches are usually created by saving the so-called *diff* (short form of difference, the name emerges from the correspondent Unix command) of two files, which is also called *diffing* the two files. Usually the two files to be diffed are the unmodified *base* version of a source file and the version containing the changes. There are many programs which can create such a diff file and then apply the changes to another file. The most basic tools for this task are the Unix commands `diff` to create and `patch` to apply the diff file. To ease the collaboration of different programs, a standardized format, the so-called *unified diff* format, is often used [81]. More sophisticated patch formats (like the unified diff format) save contextual information with the changes and are thus able to apply the patch meaningfully to a file which differs from the base file from which the patch was created. If the differences are to big, however, the patch application can yield undesired results or fail completely.

Instead of exchanging patches, another approach would be to send the complete changed files. However, using patches has numerous advantages:

- A patch can also be applied to files that are slightly different from the base file from which the patch was created. This is the most important advantage, because the developer reviewing a patch can apply it onto his version of the source code, even if it differs from the ones of the patch-creator. This is often the case if the developer himself also introduced changes to this file, or the patch-creator used an older version of the file.
- The patch file directly shows which parts of the code were changed, so the reviewer can concentrate on them. This is particularly useful when it comes to large files.
- Programs applying patches can also revert the application. Thus it is easier for the developer reviewing the patch to apply it to his local copy of the code, test it, and revert the application if the changes are flawed.
- The unified diff format also includes the file name of the file to be patched. This allows to apply the patch more conveniently because it enables the applying program to identify the file to patch automatically.

- The file size of all changed files would be bigger than the corresponding patch (this issue is rather insignificant considering the disk space and bandwidth of today's computers and networks).

Most version control systems contain the functionality to create patches depicting the changes between a base version of a file in the repository and the current version of that file in the working copy. Thus, patches can be used easily in addition to the usual version control mechanisms.

2.2.4. Merging

When multiple versions of source code are to be combined, the contents of each file, or rather the differences between the two files, have to be combined. This is called *merging* the files. The most obvious situation in which source files have to be merged is when two development branches are merged. However, even without having multiple branches, merging must often be done by a version control system. When a user updates to the latest revision of the branch he is currently working on merging may be necessary: If more than one user is working concurrently in the branch, they can add changes to the same files. Upon updating, the changes the other users have already committed are received and have to be merged with the changes the local user has in his workspace. Thus, merging is an important mechanism for every version control system which supports more than one user (which every modern VCS does), even if the system does not support multiple branches.

The merging of two versions of a file is the process of combining the differences in both files to create a new version containing all aspects of both versions. Depending on the amount and complexity of differences, and on the amount of additional information available (like the knowledge about a base version from which both versions emerged), the merging can be done automatically by an algorithm or must be done partly by a human. The latter is the case if the files are either so different that a merge algorithm cannot even find a common pattern, which happens very rarely, or if the algorithm is unable to combine the differences in the files because it cannot decide how to combine them. An example for the second case, which happens quite frequently, would be itwo text files containing totally different content in one line. An algorithm cannot determine which of the versions is the correct one or if both versions have to be combined somehow, so it surrenders and cedes the choice to the user. This case is called a *merge conflict* or simply *conflict*. The files and the differences in them are then said to be *conflicting* or *in conflict*. The system performing a merge usually tries to resolve each difference between both files automatically. So, even if some differences in the file are in conflict, others can be merged by the system.

There are two conflicting goals for a merge algorithm: It should be able to resolve most differences automatically without escalating a conflict, but it should never resolve any differences producing unintended, wrong results. The second goal is very important, because developers performing a merge tend to think that everything went right if no conflict occurred. If an erroneous merge was done, this could introduce bugs in the code which are hard find, as they were not introduced by a programmer but by the system while performing the merge. So, the key feature of a good merge algorithm is to resolve many differences automatically while ensuring that almost no wrong results are produced.

Two-Way Merge

The naïve merge approach compares two versions and tries to combine all differences. Because only two versions are involved, this type of merge is called *two-way merge*. It is used by older version control systems. It performs comparably bad because it either generates a large amount of unintended results or produces many conflicts. The reason for this is that it is not clear which version has to be used in the case of differences. The following example shows in which simple situations a two-way merge fails:

Version 1

```
class C{
}
```

Version 2

```
class C{
    // Line added
}
```

Figure 2.4.: Two versions of a source file

Consider the two versions of a Java source file shown in Figure 2.4. The only change is that a line with a comment was added in the second file. A naïve two-way merge without a preference for any file must guess if it is correct to leave out the line or add it. Guessing wrong will lead to an unintended result, so it is better to propagate the decision to the user, i.e. creating a conflict. This conflict can be resolved automatically by stating which version "wins" in such situation. Such a decision, however, will not lead to correct results in most cases, because both files may contain changes which should be used. Another approach would be to always prefer adding a line over removing it. But, this will introduce wrong results in cases where a line should be deleted. Thus, there is no general strategy which can accurately merge two files without either much help from the user, which is undesirable, or additional information. The following strategy relies on such information.

Three-Way Merge

A more sophisticated algorithm for performing merges is the *three-way merge*. As the name suggests, three versions of a file are used to perform the merge: The two versions two be merged and the base version from which both versions originated. In the case of a branch merge in a version control system, the base version can always be obtained by following the branches backwards until the revision where they diverged is reached. The file in this revision is used as the base version. This approach produces less conflicts because it can often make a decision based on the content of the base version.

Version 1

```
class C{
}
```

Version 2

```
class C{
    // Line added
}
```

Base Version

```
class C{
}
```

Figure 2.5.: Two versions of a source file emerging from a common base version

Again, consider the example source files from Figure 2.4 with an added common base

version, as shown in Figure 2.5. In this case, the three-way merge can safely decide that it is correct to add the line with the comment. Since the base version did not contain the line, the line must have been added in version 2 while version 1 has not been altered. So, the simple addition, change, or removal of one or more lines can often be resolved automatically by a three-way merge. However, if both versions contain changes in the same places, a three-way merge must fail as well and produce a conflict.

Version 1

```
class C{
    // Foo
}
```

Version 2

```
class C{
    // Bar
}
```

Base Version

```
class C{
}
```

Figure 2.6.: Conflicting versions

Consider the versions in Figure 2.6. In this case, both versions added a line with different content to the base version. The algorithm cannot decide if it is correct to use either of the versions or add both lines. Thus, the system will report a conflict and the user has to decide.

Although two general strategies for merging — the two-way and three-way merge — were discussed, no concrete algorithms were presented, as this would go beyond the scope of this introduction. However, a large number of algorithms exist. Whether a version control system supports easy branching depends much on the quality of its merge algorithm. Without a good merge algorithm, each branch merge may raise many conflicts which have to be resolved by the user performing the merge. Resolving conflicts is no easy task because it requires knowledge about the contents of both versions of the file to decide how the conflicting changes have to be combined.

Conflicts

Whenever the differences in two or more versions of a file have to be merged, a conflict can occur. A conflict depicts the situation in which the system is not able to decide how to merge a specific difference. In this case, the system stops the merge process and asks the user to merge the differences manually.

As shown above, less sophisticated merge strategies like the two-way merge often produce conflicts. More sophisticated strategies like the three-way merge are able to resolve more potential conflicts automatically by examining additional information like a common base version. A common situation where a conflict occurs and cannot be resolved automatically by a merge algorithm is that two users edited the same parts of a file concurrently. Another case where the merge system is not able to perform the merge is if two files differ from each other so much that no common structure can be found. But this occurs only very rarely because files to be merged usually emerge from the same base file. Therefore, they usually do not contain totally different content.

In case of an unresolvable conflict, the responsibility of a good merge system is to provide the user with the exact location of the conflict and the content of both versions at that location. It is then the user's tasks to combine the content of both versions manually. The most common way for showing a conflict to the user is by inserting so-called *conflict*

markers into the file. These are textual insertions depicting details about the position of the conflict and the conflicting differences. Consider the source code versions which were shown in Figure 2.6. Here, the action causing a conflict is the insertion of the comments `//Foo` and `//Bar` into the same line. Assume the two versions are managed using Git and stored on the branches `Version1` and `Version2`. If the user orders to merge these branches, Git will detect the conflict and insert conflict markers yielding the content shown in Figure 2.7.

```
public class C {
<<<<<<< refs/heads/Version1
    //Bar
=======
    //Foo
>>>>>>> refs/heads/Version2
}
```

Figure 2.7.: Result of merging version 1 and version 2 of Figure 2.6

The conflicting line was replaced by conflict markers wrapping the two conflicting versions of that line. Git surrounds a conflict by multiple < and > signs followed by the names of the merged branches (which are implicitly prefixed with `refs/heads/` in Git). The contents of the two branches are separated by = signs.

So the user has the conflicting lines replaced by the conflict markers comprising the conflicting contents. He can now combine these contents and remove the conflict markers to resolve the conflict. Afterwards, he must explicitly inform the version control system that the conflict was resolved. Because conflict markers are plain text (and thus could be added to a file manually), the system cannot rely on just checking the absence of conflict markers. In Git, the developer uses the `add` command to mark the conflict in a file as resolved. Other version control systems use similar mechanisms.

Although the manual resolution of a conflict seems quite simple on first sight, it is indeed a very tricky operation. To combine the differences meaningfully, the developer must know the purpose of the lines which conflict and must know how the two versions alter this purpose. For large projects, this can be quite a problem because not many persons know the whole source code of the project. Consequently, a merge should always be executed by the persons who worked on the branches to be merged. Merging branches leads to incorporating changes to the source code. Thus, even if both branches are tested thoroughly, their merge can introduce new bugs and has to be tested again. Due to all these reasons, resolving conflicts is a very dangerous operation and should be averted whenever possible. It is very important to use a good merging algorithm that brings up as few conflicts as possible, while not producing erroneous results.

2.3. Popular Version Control Systems

In this section, the most popular and historically important version control systems are described in more detail. Especially, Git and Subversion, which were examined in the scope of this thesis and on which the prototype is based, are described in more detail.

2.3.1. Early VCSs

Version control systems have been existing since the 1970s. However, the first systems did not contain any form of networking and offered version control locally on one machine. This limited the collaboration support of this tools significantly. Hence, version control systems were not as popular as today. This changed with the release of the first version control system with network support: CVS.

Source Code Control System

One of the first systems to be considered a version control system is the Source Code Control System (SCCS) [82] written 1972 by Mark J. Rochkind at the Bell Labs. Although it was developed long ago, it contains many features of modern version control systems. For example, it already contains limited access control, is able to reconstruct any recorded revision, logs the date and user who made a specific change, and allows to state a reason for a change (comparable to the so-called *commit message* in recent version control systems) [83]. SCCS stores revisions using forward deltas (cf. Section 2.2.2). It is a local VCS and does not support collaboration between different machines. Therefore, it has become obsolete and is not used widely anymore. Its file format is still being used by some modern version control systems like the proprietary products TeamWare [84] and BitKeeper [85].

Revision Control System

The Revision Control System (RCS) [86] is a version control system written by Walter F. Tichy in the early 1980s [87]. It is realized as a set of Unix commands. RCSs tracks single files, which makes the tracking of whole projects cumbersome. RCS stores revisions as reverse deltas in order to improve performance, because it is assumed that newer versions (especially the latest version) are needed more often than older ones. RCS explicitly supports branches. However, RCS stores branches as forward deltas to save disk space (as each branch head would have to be saved as plain text in the reverse approach). This decreases the performance for retrieving revisions on branches: Starting from the head revision of the trunk, the reverse deltas must be applied back to the revision where the branch forked from the trunk and then applying the forward deltas in the branch until the desired revision is reached. This design decision was made because working with branches was very uncommon at the time RCS was developed. The focus was laid on fast retrieval of the trunk revisions. RCS introduces locking of important files, which prevents other users from changing them while volatile changes are made. Like SCCS, RCS is a local VCS and not widely used anymore. It was mainly superseded by CVS, which was originally based on RCS, but added network support.

CVS

The Concurrent Versions (or Versioning) System (CVS) [8] started out as a set of UNIX shell scripts written by Dick Grune in 1986 [88]. The scripts used the commands of RCS as version control primitives and built a networking layer on top of them. It introduced the concept of a central repository and a local copy on the developer's machines. In contrast to earlier VCSs, this imposed the need for operations to merge code from the repository

into the local version and vice versa. In 1989, Brian Berliner wrote a C version of CVS which is still used today, calling it CVS II [89]. Although this version was not built on RCS anymore, it still used the storage format of RCS.

CVS is able to version whole projects. This is accomplished by using the approach of keeping a separate RCS history for each file while providing commands which operated on many files simultaneously. The introduction of networking and the ability to easily manage whole projects, paired with CVS being free software, quickly made it a de-facto standard in open source software development [90]. Although CVS is used less often today in favor of more recent version control systems, it is still used for many projects, especially in the open source community [91]. CVS was particularly superseded by Subversion which was created with the aim to become CVS's successor.

2.3.2. Subversion

Subversion (SVN), now Apache Subversion [9], was initially created as open source project at CollabNet in the year 2000. The explicit goal of SVN was to create a version control system which operated much like CVS but fixed bugs, limitations, and design flaws of CVS [90]. In November 2009, it became an incubation project of the Apache Software Foundation and was accepted as Apache top level project on February 17, 2010.

An SVN repository consists of one directory tree starting at a specified root. Different projects are usually stored in the same SVN repository by creating sub-folders in the repository's root folder. Instead of versioning single files, a revision in SVN may represent a whole directory tree of files. SVN uses global revision numbers to identify revisions, starting from zero and increasing the revision number by one for each successive commit. Only files which are actually changed in a commit receive the new revision number of the commit, others retain their old ones. The revision number of a directory is calculated as the highest revision number of its contents. The reason why unchanged files retain their old revision number after a commit is because a commit does not include an update in SVN. Thus, these files could have been changed by other people in the meantime. If their revision number was updated, their content should be updated as well to maintain consistency. Since no update is performed during commit, they are not updated. Once an update to the latest revision is done, all files' revisions are update to the latest revision number even if they have not changed. This is done to flag these files as up-to-date. Since trees have a revision, too, it is very easy to describe a specific revision of the whole project by stating the revision number of the root directory of the project. In CVS, in contrast, a revision of a project can only be specified by stating a certain date and then searching the revision for each file which was the latest at that date. The improved support for versioning of whole projects is one of SVNs key features over CVS.

SVN stores meta data for each folder in a hidden sub-folder called `.svn`. This folder also contains the base version of each file as yielded by the last check-out or update operation. While this increases disk usage, it allows the execution of some commands locally instead of having to query the remote repository, which would introduce network delay. For example, the changes in the working copy can be calculated locally, while CVS must ask the repository to collect the changes. Also, local changes can be reverted without querying the repository.

SVN is a client-server based version control system: The working copy is linked to ex-

actly one remote repository. The networking can be done via an own SVN protocol, HTTP, or SSH+SVN to use secure shell encryption [92]. The repository uses either the Fast Secure File System (FSFS) or a Berkeley Database (BDB) as back-end to store the content. Both approaches use deltas to save disk space. The details are based on the used back-end [93]: In case of FSFS, forward deltas are stored. This means, to retrieve a revision deltas must be applied starting from revision 0 until the desired revision is reached. Because this would cause time consumption linear to the number of revision, so-called *skip deltas* are used: Instead of saving the deltas to the direct predecessor revision, the deltas to an earlier version are stored. Which earlier version is chosen is calculated from the revision number in a way that ensures that at most a logarithmic amount of deltas has to be applied to reach an arbitrary revision. This approach is similar to the one of skip-lists. If BDB is used as back-end, reverse deltas are stored. The skip delta approach is also used here to ensure logarithmic time consumption.

SVN does not offer the concept of branches or tags directly. However, it is able to support them by using the `copy` command. This command makes a so-called cheap copy of a file or directory in the repository. A cheap copy starts out as just a symbolic link to the copied directory. Once changes are done to the copy, these changes are stored as deltas only. Thus, although the cheap copy is shown as an own directory with all the contents of the source directory in the repository, it does not take the space of a complete copy (at least not in the repository, only in the working copy). A branch can be created by cheap-copying the whole project folder to another location. Developers who want to work on the branch check out the destination where the files were copied to and introduce their changes on this destination. Since the resulting copy is not explicitly marked as branch, a default directory structure is recommended to identify branches and tags: The recommended project structure in an SVN repository is to have a directory for each project in the root directory of the repository. In this directory, the folders `trunk`, `branches`, and `tags` are created. The `trunk` folder contains the main development branch of the project. A branch is created by cheap-copying the content of the `trunk` folder into a new subfolder in the `branches` directory, having the branch's desired name. A tag is created the same way, but copied into the `tags` folder. Thus, a tag in SVN is conceptually the same as a branch with the exception that nobody should commit to this tag (which would make it a branch). Access control mechanisms can be used to prevent users from committing to the tags directory. To check out a tag or branch without having to check out the whole directory (which would also take additional disk space), SVN provides the `switch` command to replace the working copy with the specified directory or revision. A shortfall of the branching mechanism in SVN is that it does not track branch merges. Version 1.5 supports the tracking of merges but allows it only for repositories which were already created with version 1.5 or higher — legacy repositories cannot be upgraded to support merge tracking[94].

2.3.3. Git

The development of the Git [10] version control system was started by Linus Torvalds in April 2005 after the proprietary version control system BitKeeper, which was used for the Linux kernel at that time, was no longer usable free of charge for all developers due to a license change. Torvalds said that he had designed Git using a "what would CVS never ever do"-approach [95]. Thus, it can be anticipated that most concepts of Git are different

from those of CVS and its successor Subversion. The most obvious difference is that Git is a distributed version control system. There are no privileged server nodes. Instead, each user has an own repository on his machine and can make this repository publicly available so other users can pull content from it. Despite the decentralized model, many open source software projects have an "official" repository from which releases are made and which developers use for updating their working copies [94].

A Git repository is a hidden folder named *.git*. It is usually found in the topmost folder of the working copy. Git identifies each data object (revision, tree, file) with a 40 byte SHA-1 hash of its content. Git does not store files by their names or paths but by their content: Two files with the same content have the same hash and are therefore stored as one blob (binary large object, Git's representation for file content). The advantage is that renaming or even moving a file to another folder still tracks this file as the same file, while other version control systems enforce the usage of special commands to keep track of renaming or moving [96]. Git supports compression to reduce the repository size. Garbage data piles up in a Git repository until it is explicitly collected. Such garbage can arise from deleted branches, for example. While it is unimaginable in a CVCS like SVN that data is irrevocably deleted from the repository, this is quite common in the distributed case.

Instead of having steadily increasing revision numbers like in SVN, a revision (or *commit*, as a revision is often called in Git) explicitly saves its previous and successive versions. Global revision numbers are generally not very suitable for distributed VCSs because it is hard to keep them consistent for the different repositories exchanging commits with each other.

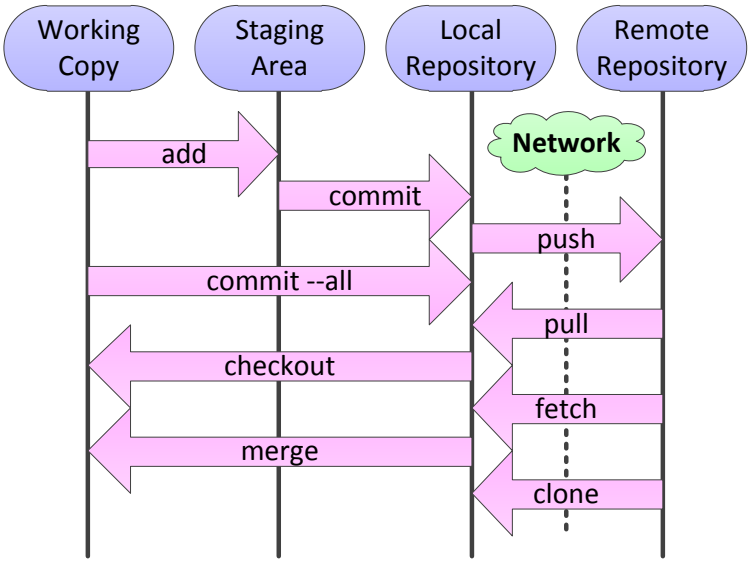


Figure 2.8.: Data transferring Git commands

Git, and DVCSs in general, support commands for committing and retrieving data from the local repository (similar to CVCSs) and for exchanging data with other repositories. Git provides even more: It has a so-called *staging area*, which is a cache for file changes before they are committed. Figure 2.8 shows the basic commands of Git transferring data between different locations. The `add` command transfers changes in one or more files to

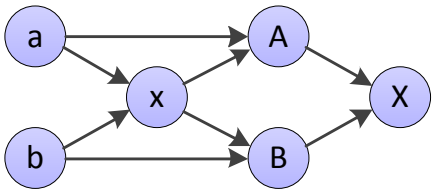


Figure 2.9.: Criss-cross merge

the staging area. The `commit` commands commits all changes in the staging area to the local repository. By selectively adding files to the staging area, a so-called selective commit can be done[1]. It is also possible to automatically add all changed files to the staging area before committing (`commit -all`), so the usual workflow of other VCSs, which do not have a staging area, can be emulated[2]. The `checkout` command can be used to retrieve any commit, tag, or branch from the repository and place it in the working copy. The `merge` command is used to merge two branches and transfer the result to the working copy. The `push` command is used to push specific branches to another repository, given that write permissions for the other repository are held. Three commands allow the retrieval of data from another repository to the local repository: `clone`, `fetch` and `pull`. `Clone` is used to create a new local repository by copying (cloning) the content of a remote repository. The `fetch` and `pull` commands allow to retrieve one or more branches from another remote repository. While `pull` tries to merge the retrieved branch into the currently active branch, `fetch` just retrieves and stores the branch without relating it to the currently active branch. Thus, a `pull` is basically a `fetch` followed by a `merge`. The `pull` command is usually used to update the current branch to the latest revision stored in a remote repository. In this case, it is desired that the branch retrieved from the remote repository is directly merged into the current working branch.

A feature Git shares with other distributed version control systems are so-called cheap local branches. The term "cheap" expresses that such branches can be created and merged quickly and easily and do not use much disk space. This allows using branches for many applications, which would be impossible with a centralized version control system. For example, a new branch can be used for a feature or bug fix that is being developed. Once the work on this feature is done, the branch is merged back into the main development branch. Another application is creating an experimental branch to have a sandbox for testing different changes. Figure 2.10 shows a small part of the revision graph of the JGit project which is self-managed with Git. As can be seen, numerous branches are created, concurrently edited, and merged. No project managed with a VCS which supports only non-local branches, like SVN, would look like this.

An additional property required to use branches conveniently is the existence of sophisticated merge algorithms which allow merging branches with a minimal number of conflicts to resolve. Git provides different merge strategies. For example, the *recursive* merge performs a three-way merge on two branch heads. While using a three-way merge

[1] This is also possible in other version control systems by explicitly specifying the list of files to include in the commit.

[2] Note that newly created files are not added by `commit -all`, only changed files. For new files, add has to be executed explicitly.

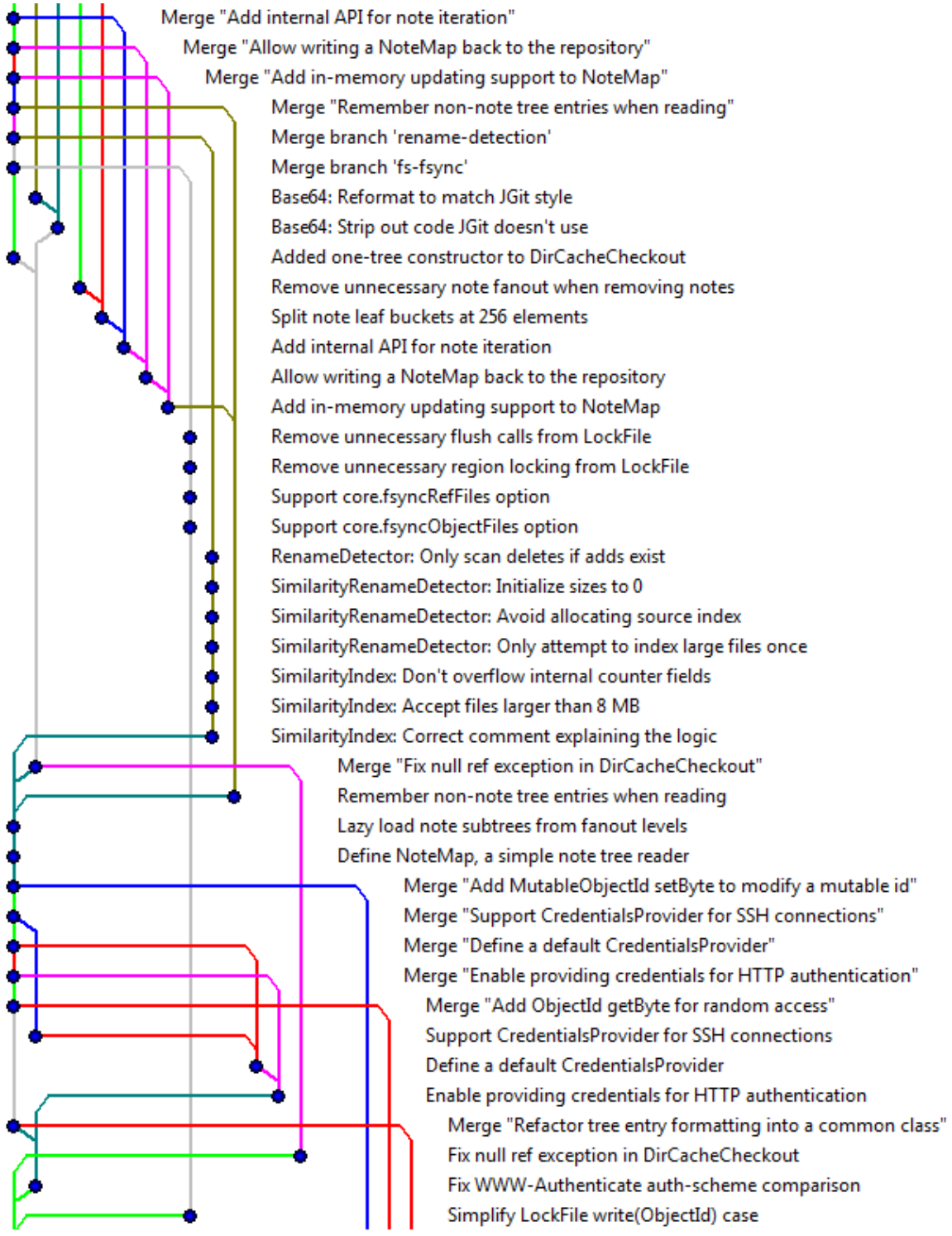


Figure 2.10.: Part of the revision graph of the JGit project

is a common feature in modern version control systems, the interesting part is selecting the common ancestor required for a 3-way merge. Figure 2.9 shows a situation, called "criss-cross merge" where it is not clear which revision to select as common ancestor [97]. Revisions **A** and **B** are to be merged to revision **X** while **a** and **b** are possible choices for a common ancestor[3]. However, using either of them could lead to a surprising result as the content of the other one would not be taken into account. To resolve this, the recursive strategy merges the candidate revisions **a** and **b** into a temporal revision and uses this revision as a basis for the merge

Another merge strategy used by Git is the so-called *octopus* merge which is able to merge more than two commits concurrently. Additional strategies like the *ours* strategy allow to merge two branches by using only the content of one branch ("our" branch, i.e. the currently active branch) and discarding the differences in the other. This can be used to alter the revision graph without merging the actual contents. For example, such merge can be used to display that a development branch containing a bug fix should no longer be merged into other branches, as it was replaced by another fix for the same problem. The different strategies applicable in Git make the lightweight branches more flexible and allow to use them in more situations.

2.3.4. Further Modern VCSs

Besides Git, many other version control systems were released in the last decade. Most of them are distributed systems. Some of the most important distributed open source VCSs [98] are Gnu Arch [99] (2001), Darcs [100] (2002), SVK [101] (2003), Monotone [102] (2003), Codeville [103] (2005), Mercurial [104] (2005), Bazaar [105] (2005), and Fossil [106] (2007). However, SVN and Git dominate the field: Figure 2.11 shows the amount of packages for the Debian Linux distribution using a specific version control system. Here, SVN and Git are by far in a leading position, followed — with a huge gap — by Bazaar. All other VCSs tend to zero, with the obsolete CVS still having the top position.

Although some proprietary solutions like Team Foundation Server [107] (centralized) by Microsoft and BitKeeper [85] (distributed) by BitMover Inc. exist, open source solutions have prevailed, because most of them are very mature projects supported by a large community and can thus equal or outperform the proprietary solutions. In addition, platforms which allow free hosting of projects exist for most major open source version control systems. This boosts their popularity even more. Examples are GitHub [108] for Git, Launchpad [109] for Bazaar, Google Code [110] supporting SVN and Mercurial, and Assembla [111] supporting SVN, Git, and Mercurial.

2.4. Change Package Representation

Before a prototype to aid the review and release management process can be designed, theoretical approaches of how version control systems can be used to reach the desired functionality have to be elaborated.

[3] Revision **x** is not a candidate because **a** and **b** are additional ancestors of **A** and **B**, respectively. Thus, **x** does not contain all necessary base information. If the direct edges **(a,A)** and **(b,B)** were missing, **x** would be a valid common ancestor.

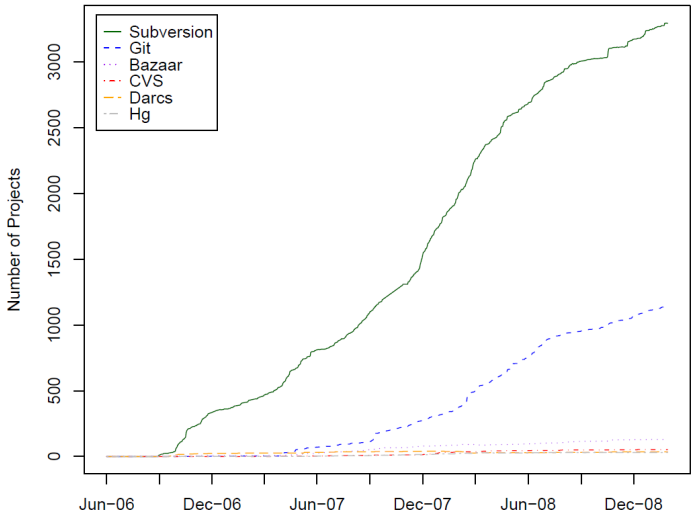


Figure 2.11.: Use of different VCSs in Debian packages [80].

The core concept in the center of our research is the `change package`, which contains a set of changes done to source code. The concepts used to represent a change package and services operating on it is crucial for the functionality of the tool to be developed. The main services needed in association with change packages are the following:

Change package creation: Changes done to the source code have to be collected and bundled in a change package.

Change package application: It must be possible to apply the changes contained in a change package to source files. This should even be possible if the source files are not identical to the ones from which the changes were collected, although this may introduce conflicts.

Change package check: It must be possible to determine if the changes contained in a change package are already applied to a given set of source files, even if these files were altered after the application. This service must be very reliable and must produce neither false-negatives nor false-positives, because it is used during release building to determine which change packages are yet to be applied to the source code to build a certain release. If the services return wrong results, a change package may be applied multiple times (which has no defined semantics and therefore may produce arbitrary, undesirable results) or not at all (which is equally undesirable).

If these three services are provided efficiently, the rest of the desired functionality can be realized rather easily. The possible implementation of these services greatly depends on the representation of a change package. Thus, the main challenge is to find a suitable representation for a change package. A representation is suitable, if a reasonable algorithm for the services working on this representation can be found.

In the course of this thesis two possible representations for change packages were elaborated. The first, earlier approach uses a patch as change package representation. The second approach uses branches in the repository of the version control system.

2.4.1. Patches as Change Packages

Using a patch to implement the concept of a change package seems reasonable on first sight: A patch is a file containing a set of changes, which quite precisely reflects the definition of a change package. To be suitable, an algorithm for the three aforementioned services, which works with patches, must be found. In the case of patches, following mechanisms could be used:

Change package creation: A change package is created by creating the patch file.

Change package application: A change package is applied by applying the patch.

Change package check: This is where patches come to their limits: It is rather difficult to check if a patch is included in a given source 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 source code was changed afterwards (which is the more common case), comparing the content and the changes in the patch will not yield a reliable result. Thus, relying on the content is not suitable.

While the concepts of the first two services are rather straightforward, the last service is quite challenging. There exist several approaches for the last service. One, for example, is keeping a list of patches applied onto the source 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 version control system, this patch will be un-tracked. Thus, the method is not perfectly suitable.

Weißgerber et al. [81] proposed another method for identifying which patches were applied to a specific revision of a CVS repository, which could also be suitable to track the patch application in other repositories.

In this thesis, however, the focus was laid on finding the best representation for change packages. Since a better alternative was found (see next section) no further investigations in finding a service for patch checking was done. However, a prototype for the patch representation of change packages was created. It is based on Subversion. Because it only relies on a small subset of Subversion's features (actually only the patch creation; the patch application is done by Eclipse if the patch is saved in unified diff format), it can also be adapted to other version control systems easily. The prototype does not implement the third service and thus only supports the use cases which do not rely on this service. Nevertheless, this prototype has some relevance. This relevance originates from the fact that the second, below-mentioned representation performs better but is limited to a small set of version control systems. The patch mechanism, in contrast, works for almost all version control systems.

2.4.2. Lightweight Branches

While patches allow the simple collection and application of changes, it is hard to derive a reliable mechanism to decide if a patch is already applied to a set of source files. Because patches perform quite well when providing the first two aforementioned services,

the challenge consists in finding a representation which is more suitable for providing the third service. Version control systems ensure that no changes, despite the ones introduced by commits (and reliably logged), are done to the repository. So, a good approach for providing reliable change package checking is to make change packages reside directly in the repository. By tying the representation closer to the repository, more of its tracking features can be used.

The obvious choice for a change package representation which resides in the repository is a branch. A branch represents changes done to the source code since the revision from which the branch started to diverge. The services could be implemented for branches in the following way:

Change package creation: A branch is created and changes are committed to this branch.

Change package application: A branch can be applied onto another branch of the repository by merging it into the other one.

Change package check: 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 information from which a revision was merged must be available.

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 version control system. 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 the first commit on the branch can be used, for example.

If optimized correctly, the performance of this approach is quite high, even for large revision graphs. A naive approach would be to walk the whole revision graph and check each visited revision against each change package, so it would be in $O(\|G\| \times \|C\|)$ (with $\|G\|$ denoting the size of the revision graph and $\|C\|$ the amount of change packages). By using a hash map from indicator revision to change package, the checking of one revision against all change packages can be done in constant time yielding $O(\|G\|)$. Even for large revision graphs with a million commits, this can be executed in a few seconds or less. In addition, the backward walking on a branch can be stopped whenever a revision is reached which is older than the oldest indicator revision of all branches. So the search has to be performed only on a the most recent parts of the tree. These optimizations enable an execution of the search in less than a second on very large graphs and even while checking against thousands of change packages.

A drawback of this approach is that it restricts the version control system to be used. While patches are supported by most version control systems, or can even be used without them, this approach puts some constraints on the version control system to be used:

First, the version control system must support branches. This is no big restriction as most modern version control system do support this. The next limitation is more severe: The version control system must support a revision graph which reveals two or more predecessors of a revision which was created by merging. SVN, for example, is not able to deliver this information and is therefore unsuitable for this approach (except the most recent versions). 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 taking too much space. Additionally, the creation and merging of branches should be fast and reliable and the merge algorithms used should be sophisticated. They must be able to resolve most conflicts automatically, because resolving conflicts is a cumbersome, error-prone work and would relativize the gain the system offers.

One system which actively advertises its ability to maintain a large number of lightweight branches is Git. It also advertises the use of sophisticated merge algorithms which reduces the amounts of conflicts propagated to the user. Thus, Git was chosen for the prototype implementation of the lightweight branch representation of change packages.

2.5. Data Dictionary

This section defines concepts which will be used in the remainder of this thesis, especially in the requirements elicitation and analysis. Since many of them are not defined uniformly in literature, it is important to consider the definitions provided here.

Repository A global location where the source code and other resources for a software development project are stored. The repository should be readable by all developers, so they can retrieve the latest source code to work on. Depending on the project, the repository may either be written by developers, or the developers must send their changes to designated persons which have access to the repository and add the developers' changes. In most modern software projects the repository will be accessed via a version control system.

Workspace The location where a developer stores his working copy of the source code for the development project. Commonly, this is a directory on the developer's local machine. The content of a workspace is obtained from the the repository, but may be different due to the developer making modifications to it.

Release A release is a distribution of software, documentation, and support materials. Only the process of assembling the final source code for the release is in the scope of this thesis. Further actions like compiling this code, packaging it and distributing it is not elaborated further. A release is associated to a set of work items. These items describe the tasks to be done to create the final source code for the release.

Stream A stream is an ordered series of releases. A release that follows directly after another one in a stream is called the successor of that release. For example, a software development project could consist of an "internal" stream which is used for internal testing releases and an "external" stream which is distributed to the public.

Change Package A change package is a set of changes. It must be "applicable", i.e. it must be possible to apply the changes in a change package to source files, even if these files do not fully resemble the files from which the changes were recorded. It must also be possible to revert the changes introduced by the application of a change package.

Work Item A work item is any task in the software development process. It is assigned to a specific person ("assignee") who should execute this task and has a "state" which shows how far the execution of this work item has advanced. This state is either "not executed yet", "executed but not reviewed yet", or "resolved" (= executed and reviewed). When work items are associated with a release, they commonly refer to implementing a new feature or fixing a bug. A change package can be assigned to a work item, which means that the changes in this package are the result of the execution of this task.

Chapter 3.

Requirements Elicitation

The first step for the development of the tool, which applies the two approaches shown in the previous chapter, is the requirements elicitation. Here, the functional and non-functional requirements are gathered by elaborating scenarios and use cases of the system.

3.1. Functional Requirements

The functional requirements describe the features and services which the system must provide to fulfill its use cases. While some functional requirements are tied closely to one use case (like the creation of change packages to the collect changes use case), others are required by almost all use cases (like the maintenance of the data model). The requirements are grouped into four main categories which will be explained hereinafter. Section 3.4, which depicts the use cases, will show which requirements relate to which use cases.

3.1.1. Data Model Related Requirements

This section depicts requirements related to the data model which has to be kept by the program. Since the viewing, browsing, and alteration of this model is a central aspect of the system, there are various requirements related to it.

FR 1: Data Model Maintenance

The main purpose of the system is to provide operations which use traceability links between different software engineering concerns like features, work items, releases, and streams. Therefore, the system must maintain a model of all these concerns. The different model elements should be organized in a tidy manner which allows to find a specific element easily even in big models. For example, a hierarchical organization of the elements can be used. Each model element must have a name and description to allow the exact specification of its meaning.

FR 2: Traceability Links Between Data

The data model must include traceability links between the various concerns. It must be possible to set these links. For example, it must be possible to assign work items to a release, associate a release to a stream, or attach a change package to a work item.

FR 3: Viewing of Data

One of the purposes of the system is to show the data model to the user. This allows to gain an overview or inspect details concerning different aspects of a project. Therefore, the system must provide ways to browse the data model and to search for specific elements in it. Particularly, it must provide ways to follow links between model elements.

FR 4: Creation and Alteration of Data

The system must allow the creation and alteration of various model elements. For example, it must be able to create releases and streams and alter their attributes like their name and description. It must also be possible to delete model elements, if an element was created accidentally.

FR 5: Data Persistency

The data model is being developed over time and has to be archived after the project is finished. Thus, the data model has to be kept persistently; it must still be available after restarting the tool.

FR 6: Data Collaboration Support

The aim of the tool is to track the work items and releases of whole software development projects. Since there are many people working in such a project which have to access the tool (release managers, reviewers, developers), the system must provide ways to ensure that all these people can work on the data model of the project. It must be possible to share the project data with other users who can read from and write to it. It must be ensured that concurrent access to the model does not result in a loss of data (e.g. lost update).

3.1.2. Change Package Related Requirements

Functional requirements related to change packages mainly specify services which work with them. The three services which have to be provided for a change package were identified in Section 2.4 and are reflected in the following requirements:

FR 7: Change Package Creation

The system shall provide the ability to create a change package. During this creation, the changes in the local workspace of the user have to be gathered and represented in the change package. Since change packages are part of the data model, the changes which they represent also belong to this model, so all data model requirements (sharing with other developers, persistency) apply to them as well.

FR 8: Change Package Application

The system must provide a service which applies the changes in a change package to a set of source files automatically. This must be possible even if these source files are slightly different to those from which the change package was created.

FR 9: Recognize Conflict

During the application of a change package a conflict can occur, since the source files to which the change package is to be applied can be different from those of which the change package was created. This is especially the case if the user has made modifications to these files in the meantime. The system must be able to detect this conflict during the application of a change package. It must then abort the application, insert conflict markers, and inform the user about the conflict.

FR 10: Change Package Containment Checking

It must be possible to check whether the changes in a given change package are reflected in a set of source files. This check must be very reliable and always yield a correct result. Although this is quite difficult to achieve, it is very important because it is the basis for the release building. Erroneous results will lead to change packages being omitted in a release or applied twice.

3.1.3. Repository Related Requirements

For various use cases, the tool must provide services which communicate with the repository to retrieve or upload changes or revisions, respectively.

FR 11: Find Local Source Code

Given a remote repository, the system has to be able to locate the source files in the local workspace which represent the working copy retrieved from this repository. Particularly, this includes that the system has to be able to decide whether a given directory contains source files belonging to a specific remote repository. This is important for the release building and change package application process, since it ensures that the changes are applied to the correct files.

FR 12: Find Remote Repository

Given a directory in the local workspace which is under version control and a set of possible remote repositories, the system must be able to compute the location of the remote repository to which the directory belongs. This is applied when creating a stream from this directory to link the stream to the correct remote repository.

FR 13: Fetch Release Head Revision

Before a release can be built by applying the change packages not yet applied, the head revision of the release must exist in the local workspace. Thus, the system must be able to retrieve this revision from the repository.

FR 14: Commit Changes to Repository

Once a release is built, the result has to be committed to the repository. Thus, the system must provide services to achieve this. Once such commit is done, the system must provide information to identify the revision created of this commit (e.g. by creating a tag or saving

the revision number). This is important because the respective information has to be linked to the release. This way, the built revision of the release can be found and accessed at any time.

3.1.4. Further Requirements

FR 15: Release Error Checking

During the checking of a release, different errors can occur. The specific set of errors which can occur depends on the change package representation and version control system. For example, if the lightweight branch approach (cf. Section 2.4) is used, a change package belonging to a release is represented as a branch in the repository. Here, it would be an error if a change package branch does not fork from the tree on which the release branch is located. All these errors which could possibly interfere with the release building process must be detected to prevent erroneous results during the building.

FR 16: Change Log Assembling

The system must be able to assemble a changelog for a given release. This changelog must include descriptions of all change packages associated to the release. The changelog will be presented to the user while checking and building a release.

3.2. Non-Functional Requirements

Since the purpose of the tool is to aid developers in frequently occurring operations, the *responsiveness* and *usability* were identified as two main non-functional requirements. Because the tool will be used to handle source code, which is a costly artifact of the software development process, *robustness* is required to ensure the integrity of the managed code, even in the presence of erroneous usage. Finally, *implementation and interface* of the tool are restricted because it is to be implemented as an Eclipse plug-in built on top of the UNICASE plug-in.

Responsiveness

Most of the operations the system provides must run quickly (less than 10 seconds) so they can be used in the usual software development process without creating noticeable delay. The building of a release can take longer since releases are not built as frequently as the other operations will occur. Nevertheless, this step should perform as quickly as possible as well.

Usability

The software must be intuitive and easy to use. It must provide functionality with as few user interactions as possible. The functionality must be provided where the user anticipates it and generally where the user "is" when he wants to access the functionality. For

example, when creating a change package, the user is normally in his software development view and wants the functionality to be provided there instead of changing to another program, window, or view.

Robustness

The tool will handle the source code of software development projects. Since the code is a very costly artifact, the tool must be robust against erroneous user input. This mainly means that the operation the system offers shall be somehow "undoable": It must be possible to revoke the changes incorporated by an operation like the application of a change package onto the local workspace.

Implementation and Interface

The system is to be built as an Eclipse plug-in. It therefore has to be written in Java. For backward-compatibility purposes, it must compile correctly with Java 1.5. It is to be integrated into the UNICASE plug-in. This means that its domain model should be stored in UNICASE projects and should make use of (and extend) the unified model already provided by UNICASE. For example, UNICASE already provides a `WorkItem` model element. This element is to be used to attach change packages to and to associate it to releases.

Furthermore, the plug-in must be built to work seamlessly with widespread version control systems. This ensures that it can be used in many software development projects relying on these systems for their version control.

3.3. Scenarios

The change tracking and release management workflow in the scope of this thesis can be described with three common scenarios: The setup of a software project, the process of assembling a release, and the common workflow of a developer.

The scenarios all happen in the same software development project. The development team consists of three members: The team leader *Helen* and the two developers *Tom* and *Simone*. The system to be developed is a WYSIWYG ("What you see is what you get") editor for HTML pages, i.e. the editor should not display the HTML source code but rather display the page like a browser would display it. The software is to be developed incrementally; many internal and external versions are to be anticipated. Therefore, the team wants to use tool support for the review and release management process.

3.3.1. Scenario "Project Setup"

At the start of the project, Helen wants to set up all components so the development can begin. She creates the repository in which the development resources (documents, source code, built products) will be stored for team collaboration. She creates the first stream for the project called "internal" to hold internal releases. Then, she sets up the first release in this stream called "internal v0.0.1". After some meetings with Tom and Simone in which the core features for the system under development are gathered, she decides which of these features are to be included in the first internal release. The system allows Helen to

save this decision. She chooses the features "basic user interface" for providing a basic user interface which is the most important feature for the first internal release. In addition, she chooses "caption support" and "table support" for supporting the HTML elements for captions and tables, respectively.

3.3.2. Scenario "Development Workflow"

Helen has assigned the task to design and implement the basic user interface to Tom. Simone is responsible for reviewing Tom's work. First, Tom designs and implements the feature. The system records the changes he does to the source code. After Tom has finished his implementation, he tells the system that he is done and that all the work he has done belongs to the feature "basic user interface". The system collects his changes and links them to the task of implementing this feature.

Simone meets with Tom to review his work together. The system helps Simone with the review by allowing her to retrieve the result of Tom's work quickly and to point out the changes he has done. Simone detects some severe bugs in Tom's code and tells him to fix them before the feature can be included in the release. She tells the system that the feature is not finished yet. Tom fixes the bugs and tells the system to assign his changes to the feature. Simone decides to review Tom's code alone this time. She finds some minor mistakes which she fixes herself. She tells the system that the feature is reviewed and can be included into the release. Since she thinks that this feature is crucial for the development of other features, she tells the system to immediately add this feature to the source code of the release in the repository.

3.3.3. Scenario "Release Building"

Helen checks the status of the release daily. The system shows her which features of the release are already implemented and reviewed. It also shows her which code changes for those features are already included in the repository and which are not. The day after Tom and Simone are finished with all features in the release Helen runs her daily check. The system tells her that the release is ready to be built. It also shows that the code changes for the "basic user interface" feature are already included in the repository. The changes for the other two features are not yet included. She decides to build the release right away and orders the system to build the release. After showing a summary of the release status again, the system begins building the release autonomously: The source code for the final release is assembled by the system; all the code changes for this release, which are not yet in the repository, are applied to it. However, the system encounters a problem while applying the not yet included code changes: The changes for the features "caption support" and "table support" both include a change in line 55 in the source file "Decorator.java". The system is unable to decide in which way to combine these changes, thus it reports this merge conflict to Helen. Helen combines the changes manually by editing the line in the file. She then tells the system that she has resolved the conflict and orders it to go on with the building process. After the system has finished merging all features into the source code of the release, it commits the updates source code to the repository. The system associates the release with the revision created in the repository to be able to retrieve it later on.

3.4. Use Cases

The tool to be created supports the change tracking and release management workflow in software projects. Thus, the actors interacting with the system are persons from the development team. The following three actors are important for the change tracking and release management workflow:

- The release manager is responsible for creating, building and releasing new versions of the product (releases). He is also responsible for deciding which changes are to be included in a release.
- Developers design and implement the system under development. They make changes to the source code and collect them in change packages.
- Reviewers review the change packages created by developers to check if they are correct and meet the quality demands. Reviewers can either be developers or dedicated persons in the project.

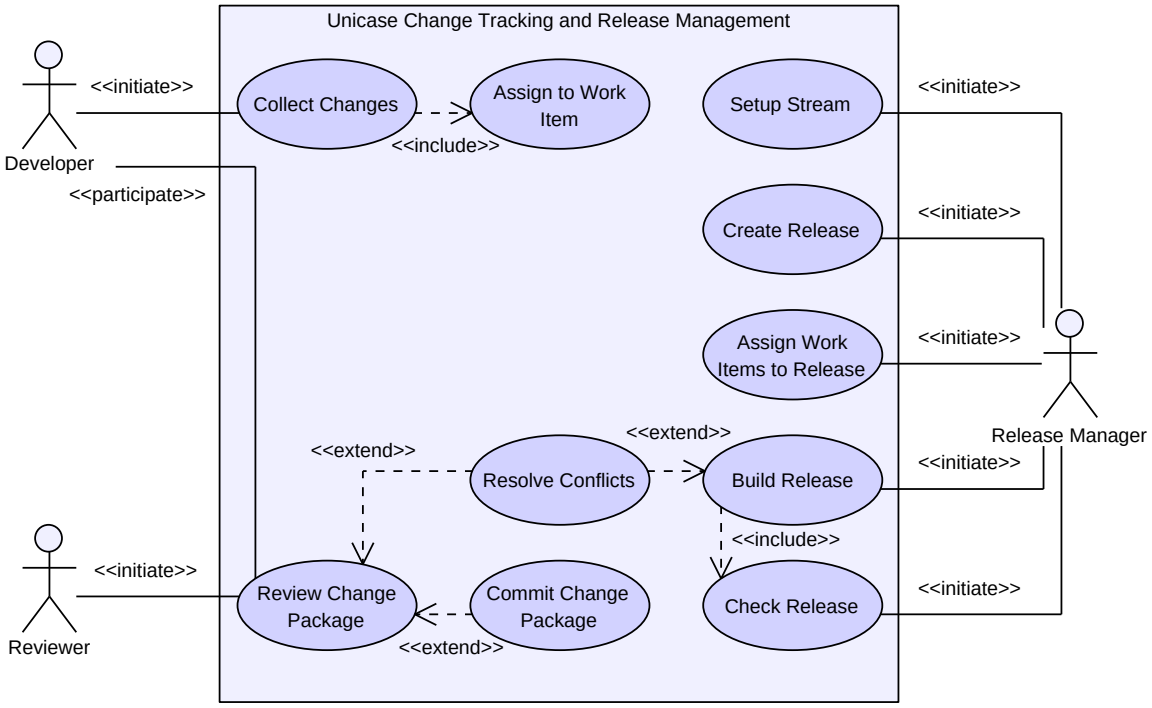


Figure 3.1.: Use case overview

Figure 3.1 shows a UML use case diagram depicting an overview of all use cases which are to be supported by the system. The use cases on the right are all related to the release management process and are thus all initiated by the release manager. First, the release manager must be able to setup a stream (use case *Setup Stream*). Then, he can create releases in this stream (use case *Create Release*). He can choose which work items are to be included in a release (use case *Assign Work Items to Release*). Later, when the developers

make progress with the work items he can ask the system for the status of the release (use case *Check Release*). When all work items for the release are resolved, the release can be built (use case *Build Release*). This includes checking the release status again. During the building process, conflicts can occur which have to be resolved before the building process can go on (use case *Resolve Conflicts*).

The system must track changes developers make to allow these changes to be inspected later on (use case *Collect Changes*). Changes collected by the system can then be associated with a work item (use case *Assign to Work Item*). This way, the changes get connected to a release (since releases include work items).

Finally, changes made by the developers must be reviewed. The system has to allow the reviewer to review a set of changes made by a developer (Use case *Review Change Package*). Therefore, the changes have to be applied to his workspace. This can cause conflicts which have to be resolved (use case *Resolve Conflicts*). If the reviewer finds no flaws in the changes and thinks that they should be added to the repository right away, he can do that right away (use case *Commit Change Package*).

The following sections describe the use cases in detail. Each section starts with a short introduction of the use case, followed by the details in tabular form. Afterwards, the participating objects of the use cases, which were identified using Abbott's natural language analysis [112], are listed. Finally, functional requirements related to the use case are listed. Note that functional requirements which are related to almost all use cases (like the maintenance of the data model or its persistency) are not included in these listings.

3.4.1. Use Case *Setup Stream*

In this use case, a new stream is created and configured. Since most projects will probably have only a few streams, this happens rather infrequently.

Use Case	Setup Stream
Actors	Initiated by release manager
Flow of Events	
User Steps	**System Steps**
The release manager selects a project in his local workspace and orders the system to create a stream for this project.	
	The system asks the release manager for a name for the stream and further details how to embody the stream in a repository. These details depend on the version control system used and thus cannot be stated generally here. The system also asks for a remote repository to link the stream with. It tries to infer one from the selected project and recommend the

The release manager provides the information the system asks for and confirms his decision.	usage of the inferred one. The system creates the stream and sets up all necessary data to embody it in the repository appropriately. It also links the stream with the selected or inferred remote repository.
Entry Condition	The release manager has the permissions to create a stream.
Exit Condition	The stream has been set up successfully.
Quality Requirements	The creation and set up shall take less than 30 seconds.
Exceptions	• The repository in which to embody the stream is not available. • The release manager enters invalid information (VCS dependent).

Participating objects: `Stream`, `Repository`
Related functional requirements:
- FR 2: Traceability Links Between Data
- FR 4: Creation and Alteration of Data
- FR 12: Find Remote Repository

3.4.2. Use Case *Create Release*

In this use case, a release on a selected stream is created. The term "created" means that the system is notified that this release exists. Neither is the code for the release built, nor are any work items or other information about the content of that release added to it. This is done later in the use cases *Build Release* and *Assign Work Items to Release*.

Use Case	Create Release
Actors	Initiated by release manager
Flow of Events	
User Steps	**System Steps**
The release manager triggers the system's "create release" function.	
	The system asks the release manager for a name for the release and for a stream to which

The release manager provides the information the system asks for and confirms his decision.	the release should belong. The system creates the release and associates it to the stream.
Entry Condition	The release manager has the rights to create a release and a stream already exists.
Exit Condition	The release has been created successfully.
Quality Requirements	The creation shall take less than 10 seconds.
Exceptions	None

Participating objects: `Release`, `Stream`
Related functional requirements:
- FR 4: Creation and Alteration of Data

3.4.3. Use Case *Assign Work Items to Release*

Work items are added to a release in this use case. This can be done directly after the release has been created, or later if the work items to be included are not known at the time the release is created. Of course, this use case can occur more than once per release. If, for example, work items are determined and added iteratively, or if new work items arise after the first set of items for a release has been added.

Use Case	Assign Work Items to Release
Actors	Initiated by release manager
Flow of Events	
User Steps	**System Steps**
The release manager chooses a set of work items and a release and tells the system to assign these work items to the release.	The system associates the selected work items to the selected release. If a work item is already in the release, it is ignored.

Entry Condition	The release is not built yet.
Exit Condition	The selected work items have been added to the release.
Quality Requirements	The system must perform the assignment in less than 1 second on average.
Exceptions	None

Participating objects: `WorkItem`, `Release`
Related functional requirements:
- FR 2: Traceability Links Between Data
- FR 3: Viewing of Data

3.4.4. Use Case *Check Release*

After work items have been assigned to a release, the release manager can check the status of this release. He is provided with an overview over all possibly interesting properties of a release, like which work items have already been resolved or which work packages have already been added to the source code.

Use Case	Check Release
Actors	Initiated by release manager
Flow of Events	
User Steps	**System Steps**
The release manager asks the system to check a selected release.	
	The system fetches the latest revision of the resources belonging to the release from the repository and stores it in the local workspace.
	The system processes the release and its resources and calculates all necessary data.
	The system shows an overview over all important properties and problems of the release. A list of all properties to be displayed is shown below.
The user browses the shown information.	

Entry Condition	A release exists.
Exit Condition	The user has been informed about properties of the release.
Quality Requirements	• The system must show as much information as computable, even when an error occurs (e.g. the repository cannot be reached). • The system must detect and show any problem that could prevent the release from being built. • The system must give hints about how to fix problems it finds. • The shown information must be well structured and presented. • The system check must take less than 5 seconds (excluding the network delay for retrieving the content from the repository which may take very long but cannot be avoided).
Exceptions	The content of the release cannot be retrieved from the repository.

Participating objects: `Release`, `Stream`, `Revision`, `Resource`, `Repository`, `LocalWorkspace`, `ReleaseProperties`, `Problem`, `WorkItem`, `ChangePackage`
Related functional requirements:
- FR 3: Viewing of Data
- FR 2: Traceability Links Between Data
- FR 10: Change Package Containment Checking
- FR 11: Find Local Source Code
- FR 13: Fetch Release Head Revision
- FR 15: Release Error Checking
- FR 16: Change Log Assembling

The properties and problems of a release to be displayed must include the following items:

- An overview of the release progress. The release progress is monitored by showing how many of the included work items have already been resolved.
- An overview of the build progress. This is monitored by showing how many change packages associated to work items of the release are already contained in the head revision of the stream in the repository to which the release is assigned.
- A short summary about the status of the release. The status can be one of the following:

  In progress Not all work items of the release have been resolved. It cannot be built yet.

Ready to be built All work items have been resolved and no errors and warnings were detected.

Building discouraged All work items have been resolved and no errors were detected. However, warnings were found and thus the building of the release is discouraged.

Erroneous Errors were found. The release cannot be built, even if all work items have been resolved.

Built The release is already built.

- A detailed structural view. This view must show which work items are included in the release and which change packages belong to them. The status of each work item (whether it is already resolved) and change package (whether it is already contained in the repository) shall be displayed as well.
- A changelog. The changelog is assembled by taking all descriptions of included change packages.
- A list of problems, i.e. warnings and errors. Since the warnings and errors that can occur depend on the version control system used for accessing the repository and tracking local changes, they cannot be specified here. This main rule must hold: All conditions that would make the building process impossible or would make it yield erroneous results must be detected and presented as errors. All conditions that might yield to an erroneous build result should be displayed as warnings.

3.4.5. Use Case *Build Release*

Once all work items in a release are resolved, the release can be built. The term "building" refers to the process of applying all change packages to the source code of the release. This yields the final source code which can be used to compile and publish the release.

Use Case	Build Release
Actors	Initiated by release manager
Flow of Events	
User Steps	**System Steps**
The release manager orders the system to build a selected release.	
	Includes use case *Check Release*. I.e. the user is once again shown the overview of the release properties. If a problem is revealed, the building process must be aborted.
	The system allows the release manager to choose build details, e.g. to choose the name

The release manager chooses the build details and confirms his decision to build the release.	of a tag in the repository to identify the built revision. Further details depend on the version control system. The system automatically builds the release: Each change package associated to the release, which has not yet been applied to the source code of the release, is applied to it in the local workspace. If a conflict occurs, the program informs the user about it and pauses the building process. After the release manager has resolved the conflict (**includes** use case *Resolve Conflict*), the building process is continued. The system commits the assembled source code of the release to the repository. The revision assigned to this commit by the repository is associated to the release as "built revision". In addition, it can also be saved in the repository by creating a tag. The system informs the user about the success of the building process.
Entry Condition	The release is ready to be built.
Exit Condition	The release is built successfully.
Quality Requirements	• The building process must take less than 30 seconds per applied change package plus maximum 5 minutes for other steps. • The building must run autonomously as long as no conflict occurs.
Exceptions	• The repository cannot be reached. • The release or its source code contain a problem which prevents the release from being built (cf. use case *Check Release*). • A conflict occurs while applying the change packages.

Participating objects: `Release`, `ReleaseProperties`, `Problem`, `Repository`, `Revision`, `ChangePackage`, `Resource`, `Conflict`

Related functional requirements:

- FR 8: Change Package Application

- FR 9: Recognize Conflict
- FR 14: Commit Changes to Repository

3.4.6. Use Case *Resolve Conflict*

While building a release or applying a change package to the source files in the local workspace, a conflict can occur. This conflict has to be resolved before the change package application or the release building, respectively, can go on.

Use Case	Resolve Conflict
Actors	Initiated either by release manager or reviewer
Flow of Events	
User Steps	**System Steps**
The user generates a conflict by ordering the system to apply a change package which cannot be merged automatically to the source files in the local workspace.	
	The system recognizes that a conflict has occurred during the application, aborts the merge process, and informs the user accordingly.
The user resolves the conflict and notifies the system about this fact.	
	The system continues the action which was interrupted due to the conflict.
Entry Condition	A change package is being applied to the local workspace.
Exit Condition	The conflict has been resolved and the system has been notified about this fact.
Quality Requirements	The system must point out where exactly the conflict occurred (files, lines).
Exceptions	None

Participating objects: `Conflict`, `ChangePackage`, `LocalWorkspace`, `Resource`
Related functional requirements:
- FR 9: Recognize Conflict

3.4.7. Use Case *Collect Changes*

The developer who implements a feature or fixes a bug wants the system to track the changes he incurs. These changes can then be assembled to a change package and associated with a work item in the use case *Assign to Work Item*.

Use Case	Collect Changes
Actors	Initiated by developer
Flow of Events	
User Steps	**System Steps**
The developer modifies the resources in his local workspace.	
	The system tracks the changes performed by the developer.
Once the developer has made all changes for a specific work item (i.e. he fully implemented a feature or fixed a bug), he notifies the system to collect them.	
	The system asks for a name and location for the change package.
The user enters the requested information and confirms his decision.	
	The system creates a change package, collects the changes in the work space and assigns them to the created package.
Entry Condition	The developer has the source code in his local workspace so that he can modify it.
Exit Condition	A change package has been created and contains the collected changes.
Quality Requirements	• The tracking of user changes must not reduce the responsiveness of the system while making modifications. • The creation of the change package must take less than 10 seconds. • The system must track the user all the time. It shall not be necessary to start the tracking explicitly.

	• The changes in the change package must be represented in a way which allows to apply it onto their machines (cf. Section 2.4 for possible representations).
Exceptions	The change package cannot be created due to a problem specific for the change package representation. For example, if branches are used, the VCS can raise an exception during the creation of the branch.

Participating objects: `Resource`,`LocalWorkspace`,`WorkItem`,`ChangePackage`
Related functional requirements:
- FR 4: Creation and Alteration of Data
- FR 7: Change Package Creation

3.4.8. Use Case *Assign to Work Item*

After the developer has created a change package, he selects the work item to which the changes in the package are related to and assigns the change package to this work item.

Use Case	Assign to Work Item
Actors	Initiated by developer
Flow of Events	
User Steps	**System Steps**
The developer notifies the system to assign a selected change package to a work item.	
	The system presents the developer a selection of all existing work items. It also gives the developer the possibility to create a new work item.
The developer selects a work item to attach the change package to.	
	The system assigns the change package to the selected work item.
Entry Condition	The user has created a change package and a work item exists.
Exit Condition	The selected change package is assigned to the selected work item.

Quality Requirements	The system must support the user in the decision which work item to assign the change package to. This can be achieved by appropriate sorting, searching and recommendation mechanisms.
Exceptions	None

Participating objects: `ChangePackage`, `WorkItem`
Related functional requirements:
- FR 2: Traceability Links Between Data
- FR 3: Viewing of Data

3.4.9. Use Case *Review Change Package*

For quality assurance purposes, any source code should be reviewed before it is added to the repository. Therefore, one or more reviewers check the source code. The developer can participate in such review to explain his code and answer questions. The system should support this by allowing the reviewer to inspect and test the changes contained in a change package.

Use Case	Review Change Package
Actors	Initiated by reviewer
Flow of Events	
User Steps	**System Steps**
The user ensures that he has the version of the source code in his local workspace on which he wants to apply the change package to be reviewed.	
The reviewer notifies the system to apply a specific change package.	
	The system applies the change package's contents to the resources in the local workspace of the reviewer. If a conflict occurs, the program informs the user about it. After the release manager has resolved the conflict (**include** use case *Resolve Conflict*) he can go on.

The reviewer checks the changes. He informs the developer about problems he finds. If no problems exist, he sets the associated work item to "resolved" and thus allows it to be added to the repository. If the reviewer is of the opinion that the change package should be committed to the repository directly, he can do this (**includes** use case *Commit Change Package*).	
Entry Condition	The developer has finished a work item and has enabled it for being reviewed.
Exit Condition	The reviewer has reviewed the change package and set its status to "resolved".
Quality Requirements	• The application of the change package to the workspace must take less than 10 seconds. • The system must support the reviewer by providing ways to highlight the changes in the workspace.
Exceptions	The reviewer's workspace is in a state which does not allow the application of the change package (depending on the change package representation).

Participating objects: `ChangePackage`, `Resource`, `LocalWorkspace`, `Release`

Related functional requirements:
- FR 2: Traceability Links Between Data
- FR 3: Viewing of Data
- FR 8: Change Package Application
- FR 9: Recognize Conflict

3.4.10. Use Case *Commit Change Package*

After checking a specific change package, a reviewer can decide to commit it to the repository right away. This is in contrast to applying the changes when a release is built. Direct commit of a change package can be useful if other change packages rely on code in this change package.

Note that the changes are committed to the repository location corresponding to the revision on which the change package was applied. For example, if a version control system with branches is used, the commit will be on the currently checked-out branch.

Use Case	Commit Change Package
Actors	Initiated by reviewer
Flow of Events	
User Steps	**System Steps**
After having reviewed a change package, the reviewer notifies the system to commit the changes in his local workspace to the repository.	
	The system commits the changes to the repository.
Entry Condition	The system has applied a change package to the local workspace of the reviewer
Exit Condition	The changes are committed to the repository.
Quality Requirements	The commit to the repository must take less than 30 seconds on average.
Exceptions	• The repository cannot be reached. • The changes cannot be committed (depending on the version control system).

Participating objects: `ChangePackage`, `LocalWorkspace`, `Repository`, `Release`
Related functional requirements:
- FR 12: Find Remote Repository
- FR 14: Commit Changes to Repository

Chapter 4.

Requirements Analysis

This chapter contains the analysis of the requirements which were identified previously. The section starts with the static domain model containing entity objects identified in the use cases. Afterwards, boundary and control objects are added. Finally, the dynamic behaviour is exemplarily described.

4.1. Entity Objects

The entity objects are the core data model identified from the requirements elicitation. They depict concepts from the solution domain. The concepts were divided into three main parts which will be explained one by one. These are the following:

CASE This package contains classes which are to be included in the CASE data model and have to be included into the CASE documents. They form the link to the rest of the CASE system.

Revision Handling This package contains the classes for tracking the changes in the local workspace, the communication with the repository and the versioning of the resources. The classes in this package will probably not be handled by the tool itself but by the version control system with which the tool collaborates.

Release Checking Here, data which is generated during the checking of a release is contained. Thus, the classes in this package are transient and need to exist only during the checking.

4.1.1. CASE Objects

The CASE package is depicted in Figure 4.1. The class depicting a development branch in the software development project is the `Stream`. Streams are associated to a so-called `RepositoryStream`. While a stream models a set of consecutive releases, the repository stream depicts the place of the stream in the repository. Such repository stream has a `RepositoryLocation`. This location contains information where to find and how to access the repository in which the repository stream is contained. There are two classes of the revision handling package to which the objects in the CASE package have links. These conceptual links are all indirect (they won't be simple associations in the resulting object design). They depend on the final implementation of the revision handling. A repository stream is linked to a `Revision` which depicts the head revision of this stream in the repository. A repository stream is not associated with a `Branch` class, although this

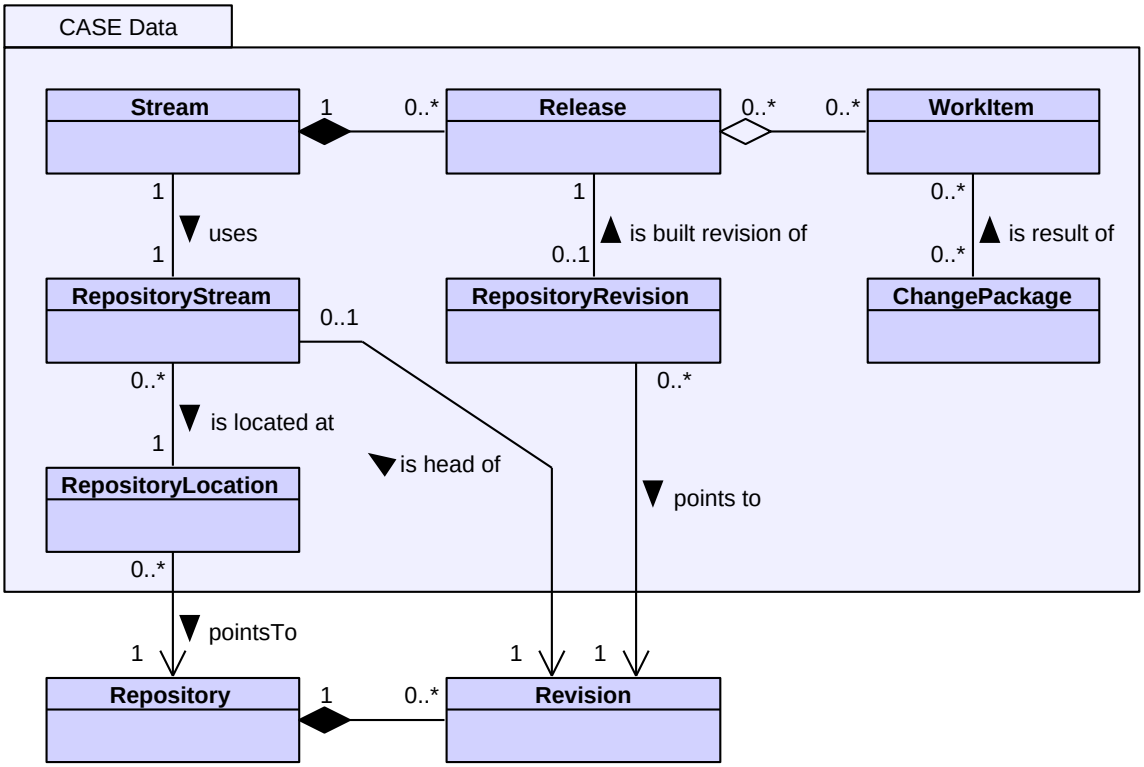


Figure 4.1.: The entity classes of the CASE package

would be a natural implementation of a stream in most version control systems. However, this would restrict the solution to using a versioning system which provides branches. By demanding only a head revision for a stream, the final implementation can also use systems that do not support the concept of branches. A repository location points to a `Repository`. This link is also indirect and can be realized for example by saving the URL of a repository accessible over the Internet. By having the link chain `Stream` → `RepositoryStream` → `RepositoryLocation`, it is possible to exactly determine where the latest code for a specific stream can be found and to retrieve this code. This is important for building the release.

The core class for the release management is the `Release`. A release always belongs to exactly one stream. A stream in return often contains a growing number of releases. The source code of a release which is not built yet can be obtained by following the link chain `Release` → `Stream` → `RepositoryStream` → `RepositoryLocation`. The head revision of the stream will contain the latest resources which can be used to build the release. Once a release is built, the revision which contains the assembled source code is saved as a `RepositoryRevision` and associated to the release as "built revision". A repository revision, as the name states, is loosely linked to a revision in a repository. Such link can for example be established by saving the revision number (SVN) or the commit hash (Git). By keeping this link to the revision, the code of a built release can be retrieved from the repository at any time.

If only the aforementioned classes are used, there is no traceability from a release to the

source code changes it should contain. This link is established over `WorkItem`s. A work item is a dedicated task or set of tasks to be accomplished in the software development process. A release may contain a number of work items. If a work item is contained in a release, this means that the result of the work must be included in the final source code of this release. The code changes which are the result of a work item are gathered in a `ChangePackage`. Once a change package is created, it is attached to the corresponding work item. The work item acts as a bridge between the tracked code changes and the release management. It is also important for code reviews: The attached change package allows the system to present the reviewer the attached code changes. If all reviews for a work item are concluded, its status can be set to "resolved". Once all work items associated to a release are resolved, the release is ready to be built and published.

4.1.2. Revision Handling Classes

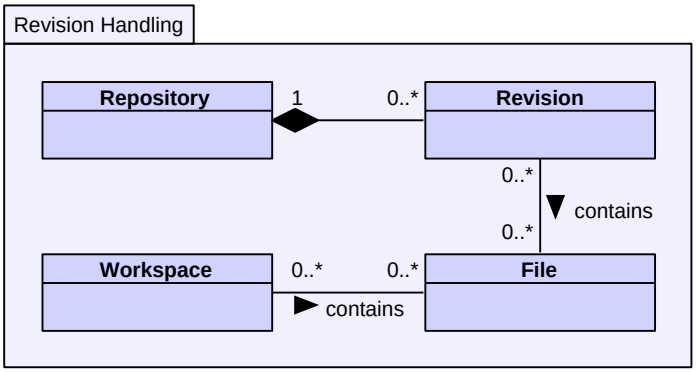


Figure 4.2.: The entity classes of the revision handling package

The revision handling package is shown in Figure 4.2. The classes in this package reflect the change tracking and versioning tasks which will probably be executed by dedicated version control systems in the final implementation. Only the basic properties of a version control system were assumed here. This allows for a wide variety of version control systems to be used or even to implement a version of the tool which does not require a version control system, but fulfills the tasks in this package itself. For example, not even branches in the repository were included in the model.

The starting point for the versioning is a `Repository`. This is a central storage for the source code of a project. It contains `Revision`s which depict versions of the `File`s (source code and other files to be versioned) of a project. There is also the `LocalWorkspace` in which the developer has his working copy of the project files. He performs changes in this workspace which are tracked by the system. The system can commit the files in his workspace to the repository, thus creating a new revision in the repository containing the files. A revision can also be pulled from the repository, replacing the files in the workspace of a developer with the ones stored in the revision.

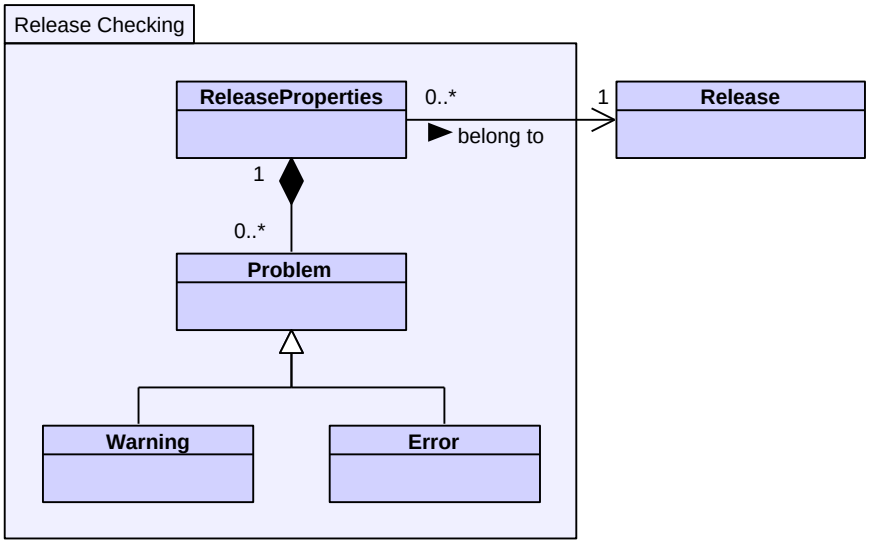


Figure 4.3.: The entity classes of the release checking package

4.1.3. Release Checking Classes

The classes in the release checking package which are depicted in Figure 4.2 model transient data which is created during the checking of a release and then presented to the release manager. The `ReleaseProperties` class contains all data generated during the checking of a specific release. It has an association to the `Release` which was checked to generate the properties. It contains `Problem`s that were found during the checking. Problems can either be `Error`s which prevent the release from being built or `Warning`s which do not prevent but discourage the building of the release. Other content of the release properties depend on the version control system used and thus cannot be modelled here.

4.2. Boundary & Control Objects

The boundary and control objects are used to operate on the entity objects to enable the work flow of the identified use cases. The boundary objects form the interface which allow the user to interact with the system. They send the commands from the user to the control objects. These objects are usually created by the boundary objects at the beginning of a use case. They create, query and modify the entity objects to form the system behaviour in a specific use case. They return the results to the boundary objects which present them to the user.

For most use cases, exactly one boundary and one control object were modeled to handle this use case. An exception to this is the `VersioningManager`. It was chosen as a dedicated control object which controls all functionality usually associated to a version control system. Thus, the `VersioningManager` must access the `LocalWorkspace` and track changes in it. Furthermore, it must be able to commit to and fetch from a `Repository`. Although accessing repositories and monitoring the local workspace are two different tasks, they were initially combined in the `VersioningManager` since accessing the repository

always also embodies reading or writing resources from or to the local workspace, respectively. Another exception for the one-control-per-use-case rule is the missing control object for the *Resolve Conflict* use case. Since this use case needs almost no control functionality of the system (the user has to resolve the conflicts manually and then notify the system), this control is included in the two control objects for the use cases which extend the resolve conflict use case.

This section shows which of the boundary and control options interact with which entity objects. It is divided into three subsections, each of them showing and explaining the boundary and control objects used for a set of use cases which access similar entity objects. Only the entity objects which are accessed by the boundary and control objects directly are shown in the class diagrams. Other classes of the entity model are omitted. Also transitive dependencies are omitted for readability purposes. In the class diagrams used, the boundary and control objects are annotated with their respective stereotypes. The entity objects' stereotype is omitted. Thus, a class without a stereotype is considered an entity object.

4.2.1. Change Package Related Objects

In this section, all use cases related to the work with change packages are explained. A class diagram depicting the related classes is shown in Figure 4.4.

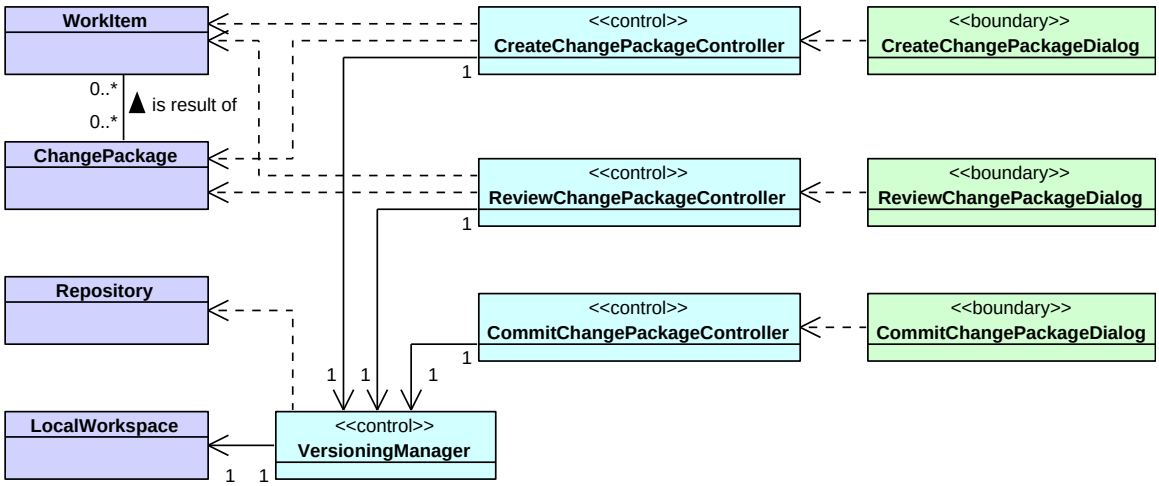


Figure 4.4.: Change package related classes

All control objects have an association to the `VersioningManager` class. As already mentioned, this control class handles all actions related to the local workspace and the communication with the repository. It has access to the `LocalWorkspace` object representing the workspace so that it can alter and query it. Actions performed by the versioning manager include retrieving the changes from the workspace, applying changes onto it, committing its content to a repository, or fetching resources from a repository.

The `CreateChangePackageController` and `CreateChangePackageDialog` classes are the control and boundary classes handling the use cases *Collect Changes* and *Assign to Work Item*. Since the latter use case is included by the former one, these two can be

handled by one set of classes. The association to the `VersioningManager` is needed to gather the changes from the workspace. Besides this association, the controller is dependent to the `WorkItem` and the `ChangePackage` class since it creates the change package and attaches it to a selected work item.

The `ReviewController` and `ReviewDialog` are responsible for the *Review Change Package* use case. The controller needs access to the `VersioningManager` to apply the changes in the change package to the local workspace. It also has dependencies to the `WorkItem` and the `ChangePackage`: The change package is the one selected to be reviewed. The "resolved" status of the work item connected to the change package has to be set if the changes are accepted by the reviewer.

The `CommitChangePackageController` and `CommitChangePackageDialog` handle the "commit change package" use case. The controller has access to the `VersionManager` since it needs to commit the resources in the `LocalWorkspace` to a `Repository`.

4.2.2. Release & Stream Setup Related Objects

The boundary and control classes related to the creation and setup of streams and releases are displayed in Figure 4.5.

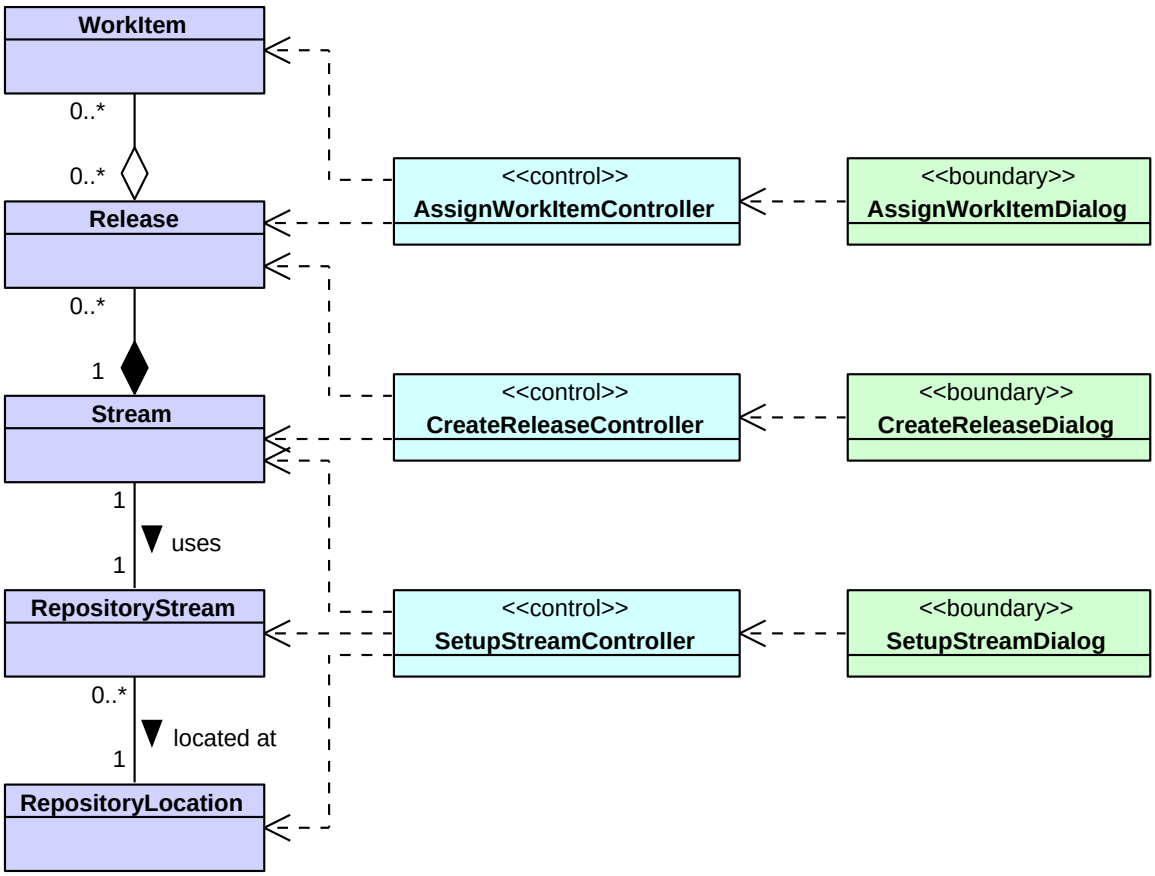


Figure 4.5.: Release & stream setup related classes

The first use case for a development project is *Setup Stream* which creates and configures a new stream. The boundary and control classes for this use case are the `SetupStreamDialog` and `SetupStreamController` classes. The controller creates a `Stream` and an associated `RepositoryStream`. In addition, it will either create or reuse a `RepositoryLocation` for the `RepositoryStream`.

The next use case is *Create Release* to add a new release to a stream. The control class for this use case is `CreateReleaseController`, the boundary object is `CreateReleaseDialog`. The controller depends on the `Release` and `Stream`, since it creates a `Release` object and needs a `Stream` to add the newly created release to.

The final use case in the release setup process is *Assign Work Items to Release*, handled by the `AssignWorkItemController` and `AssignWorkItemDialog`. The controller needs a `WorkItem` and a `Release` to assign the former to the latter.

4.2.3. Release Building Related Objects

The building of a release contains the actual *Build Release* use case and the *Check Release* use case which is included in the building process but can also be triggered separately. The class diagram for these use cases is depicted in Figure 4.6.

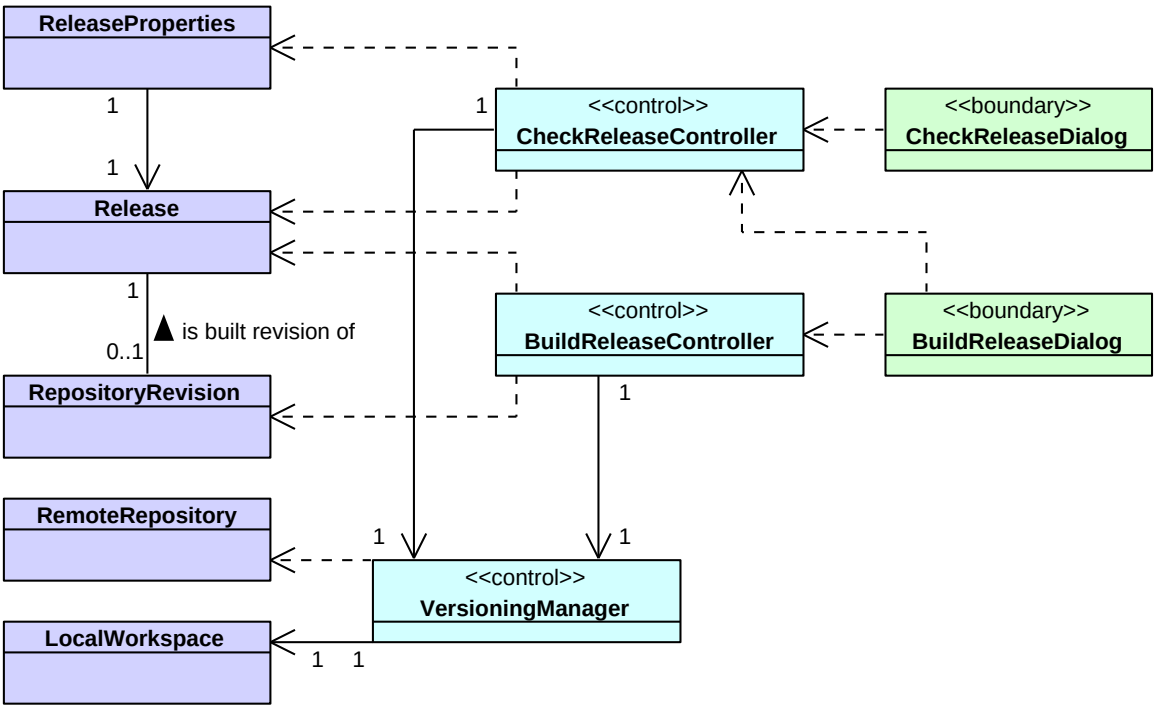


Figure 4.6.: Release building related classes

Checking a release is controlled by the `CheckReleaseController`. The boundary object communicating with the user is the `CheckReleaseDialog`. The controller needs a `Release` to be checked. The check produces a `ReleaseProperties` object. The controller also needs a link to the `VersioningManager` to fetch the latest revision of the

release's stream from the Repository to the LocalWorkspace and to receive information about the changes in the local workspace.

The release building is controlled by the BuildReleaseController. Here, the boundary object is the BuildReleaseDialog. In addition to the build controller, the dialog also uses a CheckReleaseController to perform the mandatory checking before building. The VersioningManager is associated to the build release controller, since building a release includes merging the change packages of the release in the LocalWorkspace and then committing the result to the Repository. The controller has dependencies to the Release built by it and the RepositoryRevision, which is created to tag the built revision of the release in the repository.

4.3. Dynamic Behaviour

This section describes the dynamic behaviour of selected use cases, namely the ones with the most complex dynamic behaviour: Checking and building a release. The behaviour of other use cases is rather trivial and can easily be inferred from the use case descriptions.

4.3.1. Release Checking

Before a release can be built, it must be checked. In addition, checks can also occur without the intention to build the release. Figure 4.7 shows a sequence diagram depicting the dynamic behaviour of the *Check Release* use case. Note that this is the version where the *Check Release* use case is included in the *Build Release* use case. Thus, the release manager communicates with the BuildReleaseDialog, not with the CheckReleaseDialog.

The use case starts with the user choosing to build a release (build(release)). The BuildReleaseDialog creates a CheckReleaseController and orders it to check the release. The controller first asks the VersioningManager to fetch the latest revision of the source code of the release from the repository. The manager retrieves the repository (and other information like its Stream) from the Release to be built. It then transfers the latest revision to the local workspace and returns the control to the controller. Next, the controller asks the VersionManager to calculate the merge information of the release. The merge information of a release contains the information which change packages are already merged in the source code of the release. To compute this information, the VersionManager needs the change packages contained in the release and therefore queries the Release object for them. After having received the merge information, the controller creates a ReleaseProperties object in which the results of the checks will be stored. Now, the controller performs the different checks and gathers their results. Which checks are to be performed is implementation dependent. After the checks are completed the results are stored in the previously created ReleaseProperties object. These results also contain the merge information. Every problem that is revealed during the checks is also stored in the properties. Finally, the properties, which are now filled with the problems and other checking results, are returned to the BuildReleaseDialog. This boundary object presents them to the user. The controller object is no longer needed and therefore destroyed[1]. The release manager is done after inspecting the changes and can close the

[1]Note that the "destroy" message is not shown here because it is not important and would only increase the

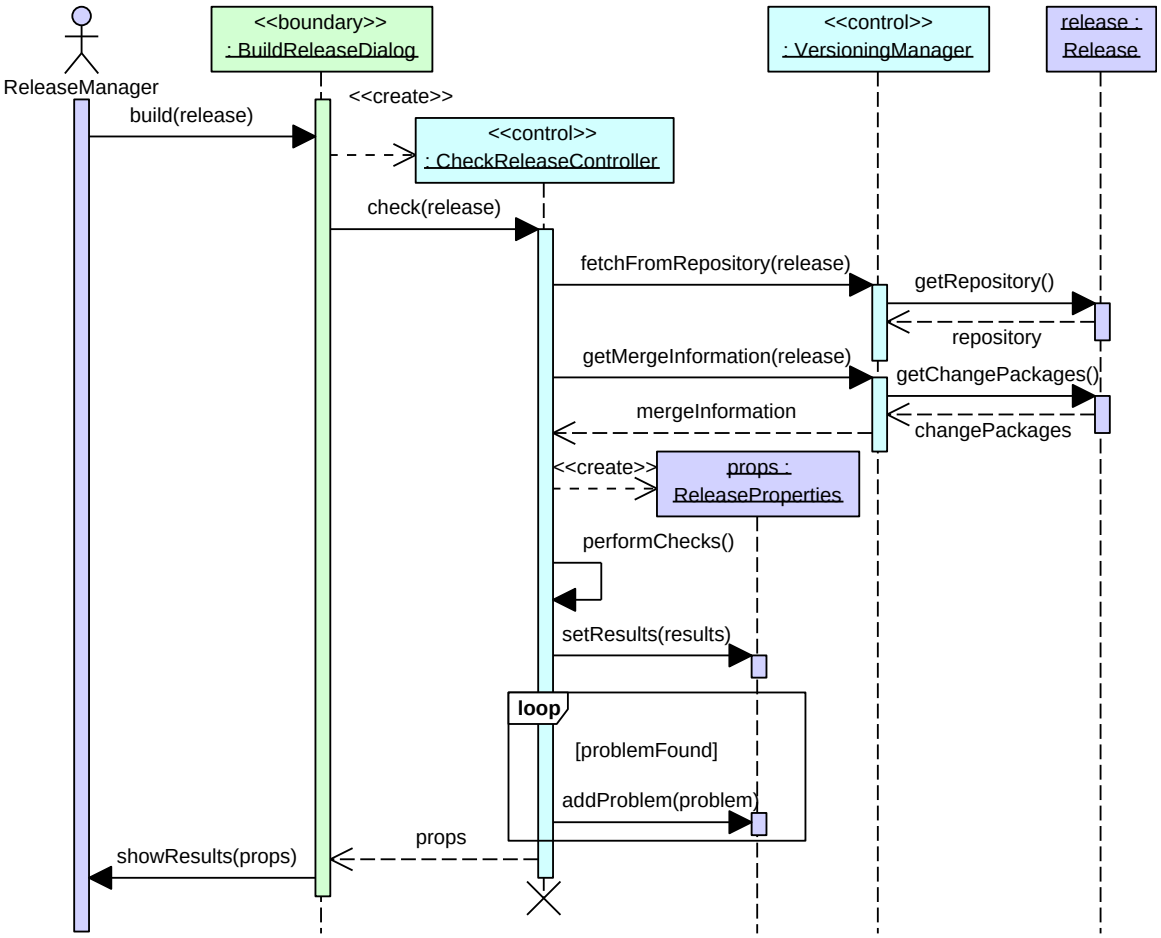


Figure 4.7.: Dynamic behaviour of the *Check Release* use case

dialog, if he only wants to check the release without building it. In case of the *Build Release* use case, the release manager can now choose to build the release if no errors occurred. The next section shows how the *Build Release* use case continues.

4.3.2. Release Building

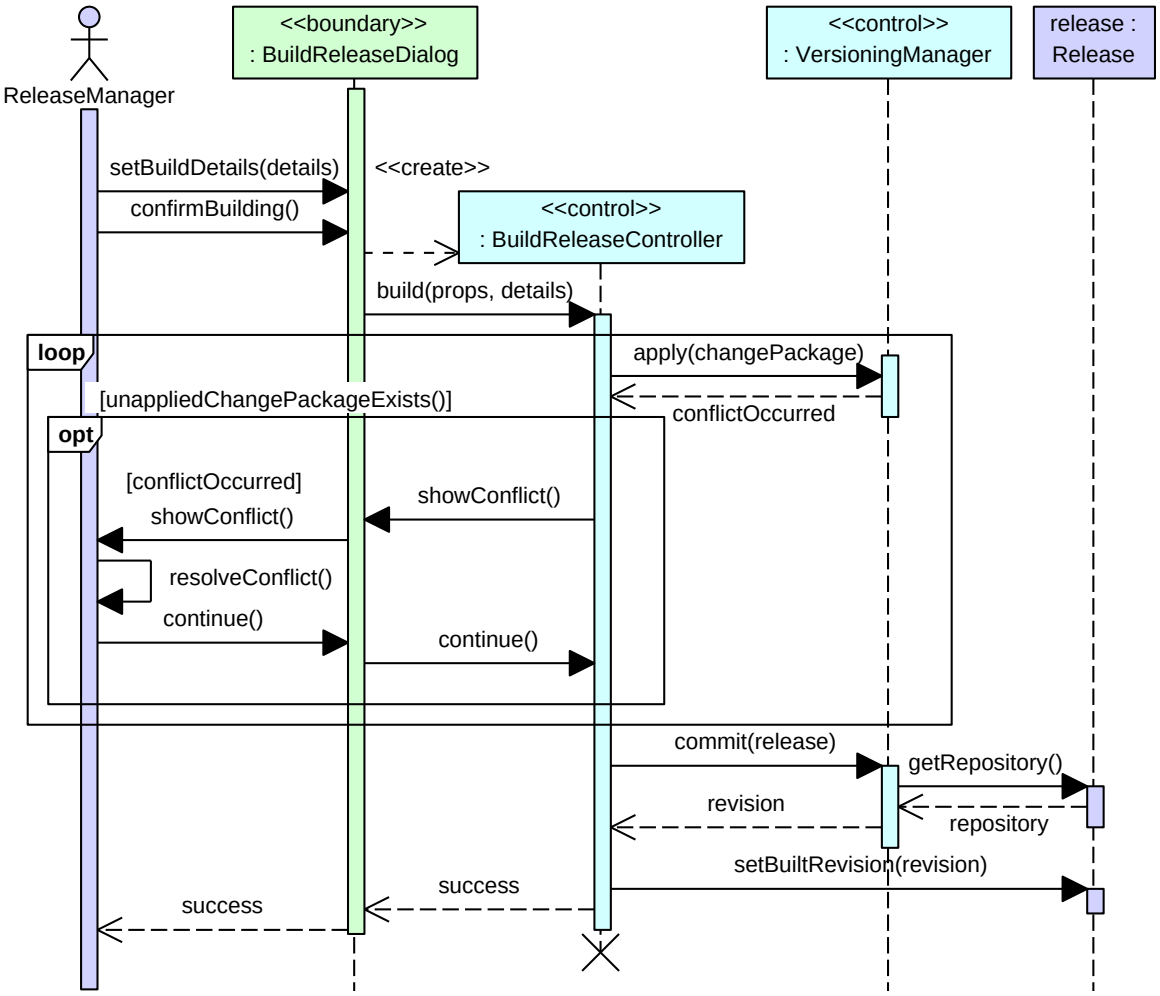


Figure 4.8.: Dynamic behaviour of the *Build Release* use case

The *Build Release* use case starts with the *Check Release* use case as described in the previous section. After the release manager has inspected the changes and no errors were found, he can decide to build the release. Figure 4.8 shows the dynamic behaviour from this point on. If the release manager decides to build the release, he first enters the build details into the `BuildReleaseDialog` and then confirms his decision to build the release. The dialog creates a `BuildReleaseController` object to handle the rest of the use case. The dialog sends the build message to the controller. This message contains the build details entered by the user and the release properties (`props`) which were computed

complexity of the diagram

before, during the release checking. In particular, these properties contain a link to the release itself and to the merge information, depicting which change packages have not yet been applied to the source files in the local workspace. The controller now processes each un-merged change package one by one: The `VersioningManager` object is ordered to apply the change package to the local workspace. If a conflict occurs during the merging of the changes contained in the package, the controller is notified about this fact. It calls the dialog to present the conflict to the user. Once the user is informed about the conflict, he must resolve it and then order the dialog to continue the building process. Once all change packages are applied, the workspace contains the final version of source files for this release. The controller orders the versioning manager to commit the content of the workspace to the repository. The version manager queries the release for the repository to which to commit the resources. Once the versioning manager has committed the resources to the repository, it receives an identification for the revision of this commit from the repository (here only called `revision`). It sends the revision to the controller, which stores it in the release as built revision. In addition, it can also create a tag in the repository to mark the revision of the release. This fact was omitted in the diagram because it is not assumed that the repository must support tagging. The release building is now finished and the controller notifies the dialog about the success. The dialog in turn notifies the user. The use case is finished and the controller is destroyed.

Chapter 5.

System Design

After the previous chapter has analyzed the requirements in a solution independent manner, this chapter begins with the prototype's design. It starts with a definition of the design goals which will have influence on the following design. Next, the subsystem decomposition is elaborated: First, the general architecture of the prototype is explained, followed by the actual decomposition into subsystems. Finally, the subsystem decomposition contains the mapping of the subsystems to components. Afterwards, the previously identified components are mapped to hardware. The chapter ends with a summary of further decisions concerning system design. Here, topics like persistent data management, access control, global control flow, and boundary conditions are described. These topics play only a minor role for the prototype, as solutions for most of them are already provided by the host environment (Eclipse and UNICASE).

5.1. Design Goals

The design goals embody properties which the design must ensure. Most of them have been inferred from non-functional requirements. Others originate from the time scope of this thesis and from the desire to make the tool easily adaptable to different version control systems.

5.1.1. Robustness & Reliability

The tool works with the source code in the software development process. It merges versions of source files, fetches source files from the repository and commits others to the repository. Source code is an expensive artifact and thus has to be protected. The design must ensure that the source files handled with the tool are by no means destroyed or damaged. The integrity of the data in the repository must be kept at all costs. In addition, it has to provide mechanisms to undo accidental changes, like the application of a change package to the workspace comprising the wrong version of the files or the building of a release with missing action items.

5.1.2. Time & Manpower

The size of the prototype is limited to the prescribed time of a master's thesis, which is six months. During this time, the thesis itself and the implementation of the prototype has to be finished. Due to the nature of a master's thesis, it has to be performed by one person; no further designers may be employed.

5.1.3. Adaptability

The tool is to be designed for collaborating with different version control systems and must be able to support different change package representations. This goal emerges from the fact that even the prototype supports two version control systems and change package representations, as stated in Section 2.4. In addition, the prototype should be designed to allow the easy implementation of support for additional version control systems. The need for this arises because the prototype will become a new feature of UNICASE. As the aim of UNICASE is to be a widely applicable CASE tool, it must be usable in as many development projects as possible. Since different development projects work with different version control systems, the tool has to support as many version control systems as possible. The tool and UNICASE are open source, so teams wanting to use them for their development projects can even implement the support for their version control system themselves. In addition, if new, more sophisticated version control systems are developed, the tool can quickly be adapted to work with these systems, thus staying up-to-date.

5.1.4. Utility

The utility design goal emerges from the two non-functional requirements "usability" and "responsiveness". The tool is useful only if users can use it quickly and efficiently. If the program does not respond quickly enough or requires much effort to learn and use, the gain it offers by supporting reviews and release building will be worth less than the time its appliance consumes. Most operations should be local. Network communication should take place only when absolutely necessary as it reduces the responsiveness. In addition, this design goal enforces the careful design of the user interface, to be as intuitive as possible. The functionality should be located in the UI where the user expects it.

5.1.5. Integration with Existing Systems

As already mentioned, the prototype is to be integrated into the UNICASE plug-in for Eclipse. Consequently, the prototype has to consist of one or more plug-ins. Therefore, the prototype must be written in the Java programming language and can make use of all the features, frameworks and toolkits the Eclipse platform offers. The prototype's user interface must extend the existing Eclipse user interface. This is achieved by adding buttons, menus, views and dialogs at appropriate positions.

The prototype's CASE data model must be integrated into the unified model of UNICASE, because the model elements of this plug-in (like releases, change packages, and streams) have to be stored in UNICASE projects. The data model used by UNICASE is developed model driven. It is modeled using the Eclipse modeling framework (EMF). The java code for the model classes is generated from the model by EMF. To integrate additional classes into this model, the additional classes also have to be modelled and generated with EMF.

The user interface of the tool should be included into the UNICASE user interface seamlessly. This should be done by adding buttons to the model element editor of the UNICASE perspective and pop-up menus when appropriate model elements are shown in the editor. For example, when a release is shown in the model element editor, a button for checking

and building the release should be shown in the menu bar of the editor.

Besides the integration with Eclipse and UNICASE, the prototype also should make use of common version control systems. To allow the support of different version control systems, a mechanism for adding support for additional systems must be provided by the prototype (cf. design goal "adaptability"). The version control system plug-ins to be supported are Subclipse [113], which is an implementation of the SVN versioning system, and JGit [114] / EGit [115], which together form an implementation of the Git versioning system. The prototype should make use of as many desirable features these plug-ins offer as possible. An example for this would be to show their respective conflict resolution views if a conflict occurs, which depicts details about the location of the conflict and aids in the resolution process.

5.2. Subsystem Decomposition

The aim of the subsystem decomposition is the division of the system into replaceable subsystems with defined interfaces, dependencies, and responsibilities. The aim is to create subsystems which focus on one task (high cohesion) while not introducing too many dependencies between subsystems (low coupling). With low coupling, changes in one system are less likely to enforce changes in other systems, thus increasing the maintainability of the prototype. High cohesion also increases maintainability by setting clear and sharp responsibilities per subsystem. If each subsystem has exactly one clearly defined purpose, then developers can understand the system faster and make changes to the correct subsystem.

5.2.1. Architecture Overview

The basic architecture of the prototype is a three layer architecture: The bottom layer is the data model which has no dependencies on other systems, but most other systems query and alter it. The middle layer is the business logic layer containing classes responsible for the functionality of the plug-in. The business logic accesses the model layer. The topmost layer is the user interface (UI). As a basic design principle, the business logic should never depend on the user interface. Instead, the user interface only calls the business logic to access functionality. The observer design pattern may be used to allow notification from the business logic to the UI without introducing dependencies. The UI also accesses the model directly to display parts of it, so this is an *open architecture*, as defined by Rumbaugh et al. [116].

5.2.2. Selection of Off-The-Shelf Components

The integration with existing systems as a design goal has a large influence on the selection of off-the-shelf components. Since the tool is to be integrated into the Eclipse platform and the UNICASE plug-in, many components like the persistency management, the access control and the version control of the CASE documents is already provided by the host environment, so no own components for these tasks have to be designed.

The aim of the tool is to provide functionality by using features of version control systems, so the management of the repository and the changes in the workspace is also handled by an off-the-shelf-component: the version control system.

The handling of the unified model of UNICASE and the repository and workspace management carried out by the version control systems already embody two important aspects of the required functionality. The rest of the functionality, i.e. the data model, the controller classes and the user interface, does not exist as a reusable off-the-shelf component. Thus, it has to be self-implemented.

There are use cases which are thoroughly implemented by the chosen off-the-shelf components. Therefore, the classes identified for them in the requirements analysis will not be reflected in the final system and object design. The following use cases are provided by off-the-shelf components:

Use case *Create Release* Since a release will be modelled as an element of the unified model, its creation can be accomplished with UNICASE, as it supports the creation of any model elements belonging to the unified model.

Use case *Assign Work Items to Release* The model element editor of UNICASE is able to assign work items to releases.

Use case *Commit Change Package* After changes are applied to the local workspace of a developer, they can easily be committed to the repository by using only functionality of the version control system (i.e. the `commit` command).

5.2.3. Subsystems

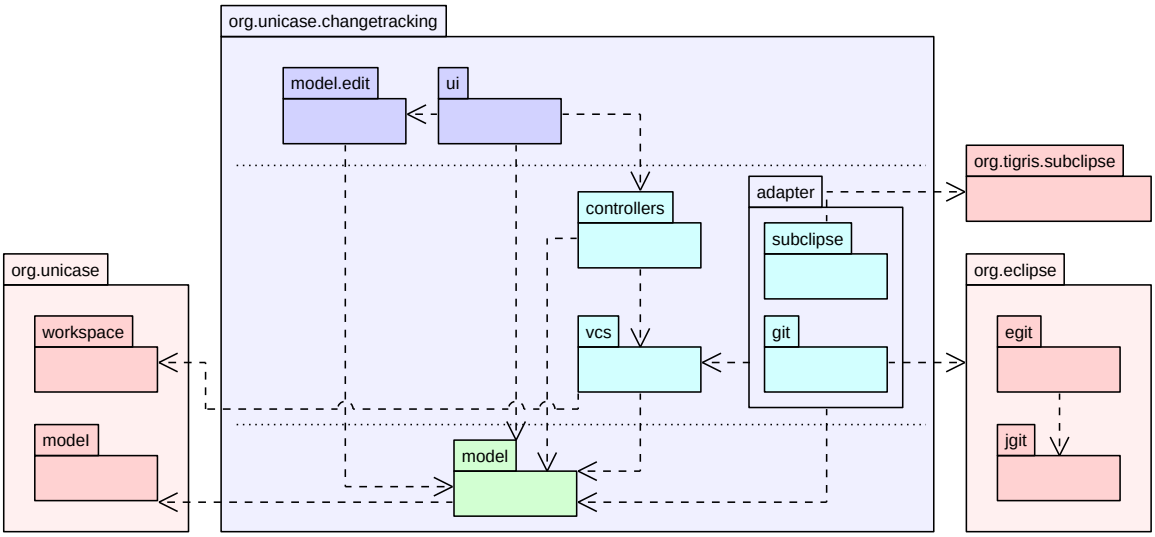


Figure 5.1.: The subsystem decomposition

Figure 5.1 shows a UML package diagram depicting the subsystem decomposition of the prototype. It also shows the dependencies on off-the-shelf components. The dependencies on packages of the Eclipse platform are not shown, since this would complicate

the diagram. In addition, these dependencies are not important as the Eclipse platform is the host environment for the plug-in and thus its classes can be regarded as a globally accessible standard library.

The system boundary for the prototype is the `org.unicase.changetracking` package (shown in light-blue). All packages inside belong to the prototype. The other packages depict off-the-shelf-components (shown in red). The `org.unicase` package depicts systems of the UNICASE plug-in, which is the basis for the prototype. The package `org.tigris.subclipse` is the Eclipse version of the SVN version control system. The packages below in the `org.eclipse` package are the Eclipse Git implementation. The dotted lines in the main package separate the three architectural layers. The data model subsystem is shown in green, the business logic subsystems are shown in turquoise, and the UI subsystems in blue.

The lowest layer in the prototype consists of the `model` package. It contains the CASE data model which consists of classes generated by the Eclipse modelling framework. This data model is built upon the UNICASE data model and thus has a dependency on it. The `model.edit` package is also generated by EMF and contains classes for editing and viewing the model.

The `vcs` package contains all classes related to handling the version control system used by the prototype. However, it only contains classes which are common to all version control systems. To reach the design goal of adaptability to different version control systems, it is obvious that there have to exist classes which access one special version control system. The adapter design pattern is used to provide uniform interfaces for all version control systems (cf. Section 6.6). The `vcs` package defines the interfaces which have to be provided by the adapters for the various version control systems and contains code to access these adapters appropriately. This plug-in therefore acts as a façade for the version control system: All other packages should communicate only with the `vcs` package for issuing version control commands (despite of the different version control adapter packages). All adapters for version control systems reside in the `adapter` package. This package depends on the `vcs` package, because the adapters implement the interfaces defined in it. The two adapters to be implemented for the prototype are the ones for the Subclipse SVN implementation and the JGit/EGit Git implementation. They are stored in the `subclipse` and `git` subsystems, respectively. They have dependencies on their respective version control implementation because they access it directly. The `vcs` package depends on the `model` package because it uses model elements (like change packages) to read and store information. It also depends on the `workspace` package of UNICASE because it uses the UNICASE command framework (cf. Section 6.5) to issue version control commands. This framework is located in the `workspace` package.

The handling of the control flow of the use cases is provided by the `controllers` subsystem. All control classes reside in this package. This subsystem is dependent on the `model` because the controllers create, alter and query model element classes. The subsystem also has a dependency on the `vcs` subsystem because it triggers different version control functionality depending on the use case to be executed.

The topmost layer of the prototype is the `ui` subsystem which contains the user interface. It creates and starts the different use case controllers and is thus dependent on the `controllers` subsystem. Its dependency on the `model` and the `model.edit` package originates from its need to display model elements.

5.2.4. Components (Plug-Ins)

Since the prototype is built for the Eclipse platform, it consists of plug-ins. A plug-in is the smallest indivisible unit which can be deployed independently to a hardware device. Thus, a plug-in corresponds to a component on the lowest level in the hardware/software mapping.

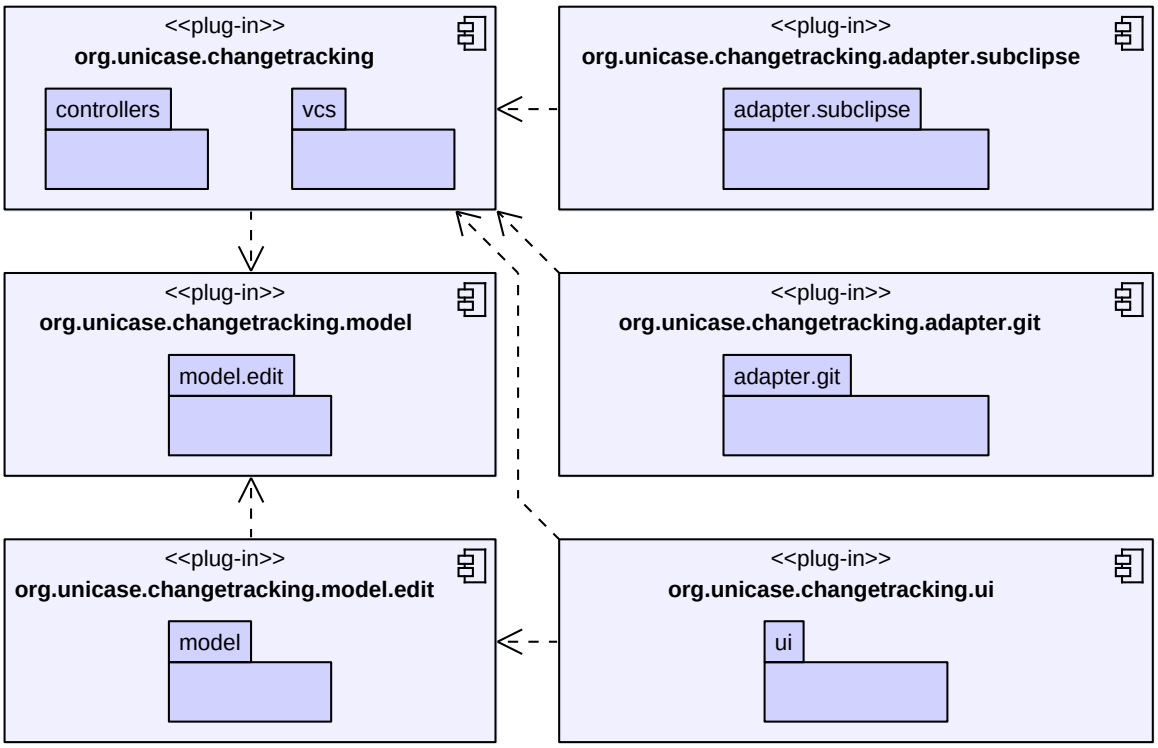


Figure 5.2.: The plug-ins of the prototype

Figure 5.2 shows a UML component diagram depicting the plug-ins of the prototype and their dependencies. The subsystems which are mapped to a plug-in are shown as packages inside the component. The components are flagged with the «`plug-in`» stereotype to emphasize that they are realized as Eclipse plug-ins. All plug-in names start with the common prefix `org.unicase.changetracking`. The remainder of this section will therefore only use the suffixes which are not shared by all plug-ins. Thus, `model` plug-in hereinafter refers to the `org.unicase.changetracking.model` plug-in. Eclipse plug-ins can reexport the dependencies they have, allowing other plug-ins which depend on them to use their own dependencies. Thus, transitive dependencies are not shown in the diagram and also do not need to be specified in the implementation. An example for an omitted transitive dependency would be the dependencies from the adapter plug-ins to the model plug-in. This dependency is already satisfied by depending on the `org.unicase.changetracking` plug-in.

Basically, the allocation to plug-ins is based on two rules: The plug-ins should be separated into user interface (UI), business logic, and domain model code, comparable to the model view controller architecture. In addition, generated code should be separated

from hand-written code, which increases the maintainability because programmers know in which parts of the code they must look for bugs and which parts are generated and thus probably bug-free.

To separate the generated code from the hand-written one, the `model` and `model.edit` subsystems must be located in a separate plug-in, as they are both generated. They could be put into one plug-in, but this was not done because the `model.edit` plug-in contains classes for viewing the model elements and thus has dependencies on the Eclipse UI. Therefore, it can be seen as an UI component and should not be mixed with the domain model classes. In addition, it is a recommendation of the Eclipse platform that code which has dependencies to the Eclipse UI packages should not be merged with code without such dependencies. For Eclipse and UNICASE, this is especially important for reusing plug-ins for client server purposes: Servers usually do not have a graphical user interface and therefore do not need any of Eclipse's UI classes. Having no dependency on the heavyweight UI framework of Eclipse allows the server to run with less resource usage. The server, however, needs the same data model as the clients it communicates with. If data model classes are mixed with user interface dependent classes in the same plug-in, this would add a undesired UI dependency on the server code.As for UNICASE, the UNICASE server (the EMF store) needs the model plug-in to be able to store classes defined in it. If the `model` code was in the same plug-in as the `model.edit` code, the EMF store would require this combined plug-in which would introduce an undesired dependency on the whole Eclipse UI framework. This is the reason why the `model.edit` code *must* be in its own plug-in, separated from the `model` code.

For the same reason, the plug-in containing the business logic (`org.unicase.changetracking`) was separated from the user interface code (`ui` plug-in). In contrast, there was no reason to separate the `controllers` subsystem from the `vcs` subsystem as they both are part of the business logic and can be deployed together.

Although the two adapters are part of the business logic, they must be located in separate plug-ins. This claim originates from the core idea of the adapters: A user should not be required to have all existing adapters, but only those for the version control system he works with. If all adapters were in the main plug-in this would introduce dependencies on all supported version control systems: A user installing the prototype would have to install all supported version control systems, which is not desirable at all. Thus, each adapter must be a separate plug-in which can be installed independently of the main plug-ins and other adapter plug-ins.

5.3. Hardware/Software Mapping

The software components elaborated in the previous section have to be mapped to hardware which runs them. Figure 5.3 shows a UML deployment diagram depicting the runtime configuration of the machines used by the tool. Components of the prototype are shown in blue and off-the-shelf components are shown in green. The `org.unicase` prefix at the beginning of the component names was omitted. The machines where the tool is mainly used are the developer machines, i.e. the PC a developer is working with. All components of the plug-in must be installed on this machine. The plug-in does not do any networking with other machines. Instead, this is done by the off-the-shelf components.

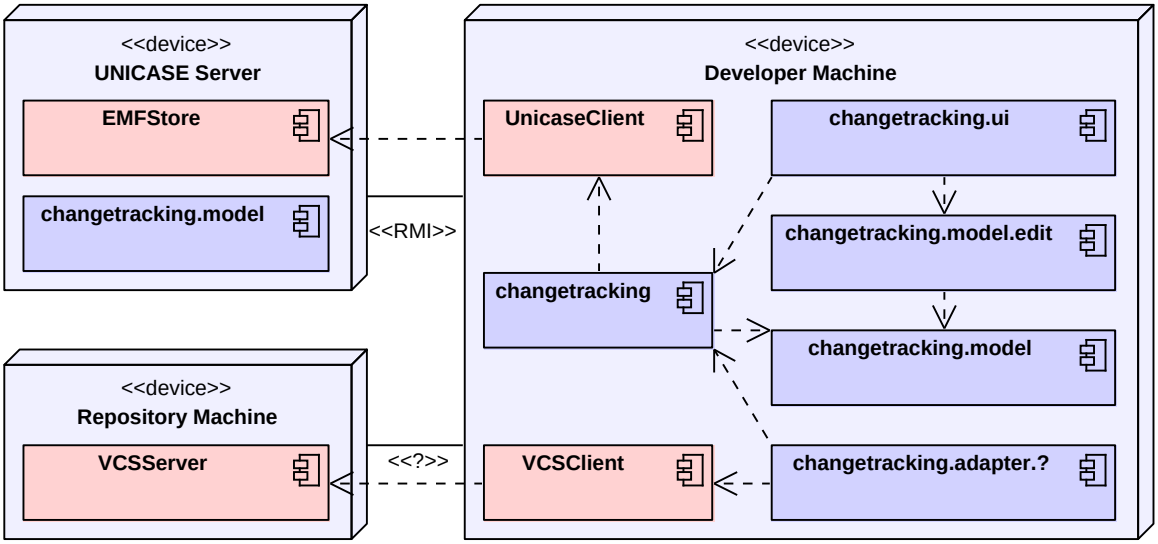


Figure 5.3.: The hardware/software mapping

The UNICASE client plug-in on which the prototype is dependent communicates with a UNICASE server. The server component of UNICASE — the EMF Store — allows the sharing of the UNICASE models and thus the collaboration with other users. The client uses Java's remote method invocation (RMI) to communicate with the server. The `model` plug-in must be installed at the UNICASE server, because the EMF store must have access to the classes of all model elements received from the client. Since the plug-in registers its additional model element classes via an extension point (cf. Section 6.3), the EMF store has no direct dependency on the `model` plug-in, but it still must be installed.

The communication between the version control system client and the server housing the repository is entirely held by the version control system, which is accessed by the appropriate adapter. The adapter is represented by the `changetracking.adapter.?` component. The name was chosen to symbolize that this must be an adapter matching the version control system in use. The «?» stereotype on the connection between the VCS server and the client symbolizes that the protocol depends on the VCS and thus cannot be universally specified here.

5.4. Further System Design Decisions

Most of the further system design decisions are dictated by the host environment, i.e. Eclipse and UNICASE. This section elaborates how the different aspects are handled by the host environment. In the cases where the prototype can handle an aspect itself, the decision about how to handle it is taken. These case are, however, rather rare.

Persistent Data Management

The persistence of data allows the system to maintain its state even if it is shut down and restarted, which is very important for this prototype. In Section 4.1, three groups of

data were identified: The content of CASE documents, the version control data, and the data acquired during release checking.

The release checking data is only needed transiently; it can be discarded after the check is finished. Thus, it does not have to be saved persistently. The version control data like revisions, branches, or changes to the local workspace must of course be kept persistently, but this is managed by the respective version control system, so the prototype does not have to perform any actions for the persistence management of this data. Finally, the CASE document content must also be saved persistently — a CASE document which vanishes after the system is shut down is of no use. The CASE document data resides in the model of UNICASE, which also handles its persistence autonomously (using serialization to an XML representation). The prototype does not have to trigger the persistence of objects, since this is all done by UNICASE. This means that even though different data has to be kept persistently, it is all handled by off-the-shelf components.

Access Control

Just like the persistent data management, access control is handled by the host environment. The EMF store handles the access rights of users reading models from it or committing changes to a model to it. The server of the version control system handles access control to the repository where the source code is stored.

Global Control Flow

Since the prototype runs on the Eclipse platform, the decision about the global control flow is basically out of its scope. The platform dispatches the handling of events like the pressing of a button by the user. The basic mechanism used by the Eclipse platform is an *event-driven* process in which the GUI is periodically queried for events and the corresponding event handlers are called by the platform when an event occurs.

5.5. Boundary Conditions

Most boundary use case are controlled by the Eclipse platform. The startup and the shutdown, for example, are out of the scope of the plug-in. Configuration use cases either affect the CASE data or the version control data. Configuration of the CASE model data is already supported in various ways by the UNICASE plug-in. Model elements and projects can be created, destroyed and altered. Configuration of version control data is provided by the respective version control system.

The only boundary use case within the scope of the prototype is the handling of exceptions. Even here, however, the prototype can rely on sophisticated mechanisms provided by Eclipse, UNICASE, and the version control system. UNICASE, for example, handles the corruption of model data. The version control system usually keeps track of the version control data and verifies that no corruption or malicious changes are introduced. In addition, it usually provides atomicity semantics for any action performed to the repository. This means that the prototype does not have to treat cases where data integrity could be harmed by, for example, a loss of connection to the repository while a commit is carried out or data is fetched to the local workspace. Hardware and network failures are also handled

at lower levels.

Since the use cases for the system are rather short actions, and all important data is already protected at lower levels, the best exception handling strategy for the prototype in most use cases is to abort the use case and show a respective error message to the user. Eclipse already provides a mechanisms for this: An exception message is shown in an error message box. In addition, the stack trace of the exception, which is available in the Java Virtual Machine, is saved in the so-called error log view, which can be accessed by advanced users, developers, or administrators to retrieve details about the exception in order to fix the problem.

Chapter 6.

Object Design

This chapter features the object design of the prototype. However, not all aspects of the object design are shown, since listing all of them would exceed the scope of this thesis. Instead, the chapter focuses on interesting parts of the design, challenges and their solutions, and the usage of framework functionality of Eclipse and UNICASE.

The chapter starts with a short introduction of the Eclipse Modelling Framework (EMF) which is used to implement the domain model using model driven development. Afterwards, the final domain model as modelled with EMF is shown and explained. The next section elaborates the Eclipse extension point concept, which is extensively used by most Eclipse plug-ins, including UNICASE and the prototype. The following section covers different aspects of a design challenge of the prototype: The appropriate displaying of model elements in various situations. The next challenge covered is the execution of commands which work on EMF models while providing "undo" and "redo" support. Finally, the approach to adapt the prototype to different version control systems is shown.

6.1. The Eclipse Modelling Framework

The data model of UNICASE, and consequently also the data model of this prototype, is modeled with the Eclipse Modeling Framework. The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor [77].

Modelling with EMF is very close to UML modelling because the EMF meta model *ECore* is an implementation of a subset of the UML meta model MOF (Meta Object Facility). ECore contains only the most important parts of the meta model like classes and their associations. It is therefore comparable to EMOF (essential MOF), which is the core of the MOF meta model. The reason is that the goal of EMF was to provide a implementation of MOF. However, many unimportant features, which are barely used and would only increase complexity, were left out, resulting in the ECore meta model.

The Java classes generated by the framework are very flexible. For example, they strictly separate interface from implementation. Each model class becomes a Java interface and a class implementing it. The client code should never access the implementation but only work with the interfaces. This provides for features which are usually impossible in Java, like multiple inheritance (which is allowed by UML and thus shall also be allowed by EMF). Since client code should not access the implementation directly, it may not create

model elements directly (as this would require a call to the constructor of the concrete implementation). Instead EMF makes use of the abstract factory pattern and provides generic factory classes for the model elements. It also supports many reflective aspects like obtaining the classes contained in a package or the operations and attributes of a model element class. The additionally generated classes for viewing model elements are extensively used by UNICASE and this plug-in, as shown in Section 6.4.2. The framework also supports the command-based editing of models, which enables support for tracking changes and undo/redo support. This feature, which is also used by UNICASE and this plug-in, is explained later in Section 6.5.

6.2. Adaption of the Analysis Object Model

The classes which should be added to the unified model of UNICASE must be modelled using EMF. As a basis for the EMF model, the analysis class model of the CASE entity objects was used. This model, which was shown in Figure 4.1 on page 58 and explained in Section 4.1.1, contains the classes which should be modelled using EMF and thus included in the UNICASE meta model. However, some small refactorings have to be applied to this model to make it suitable for UNICASE.

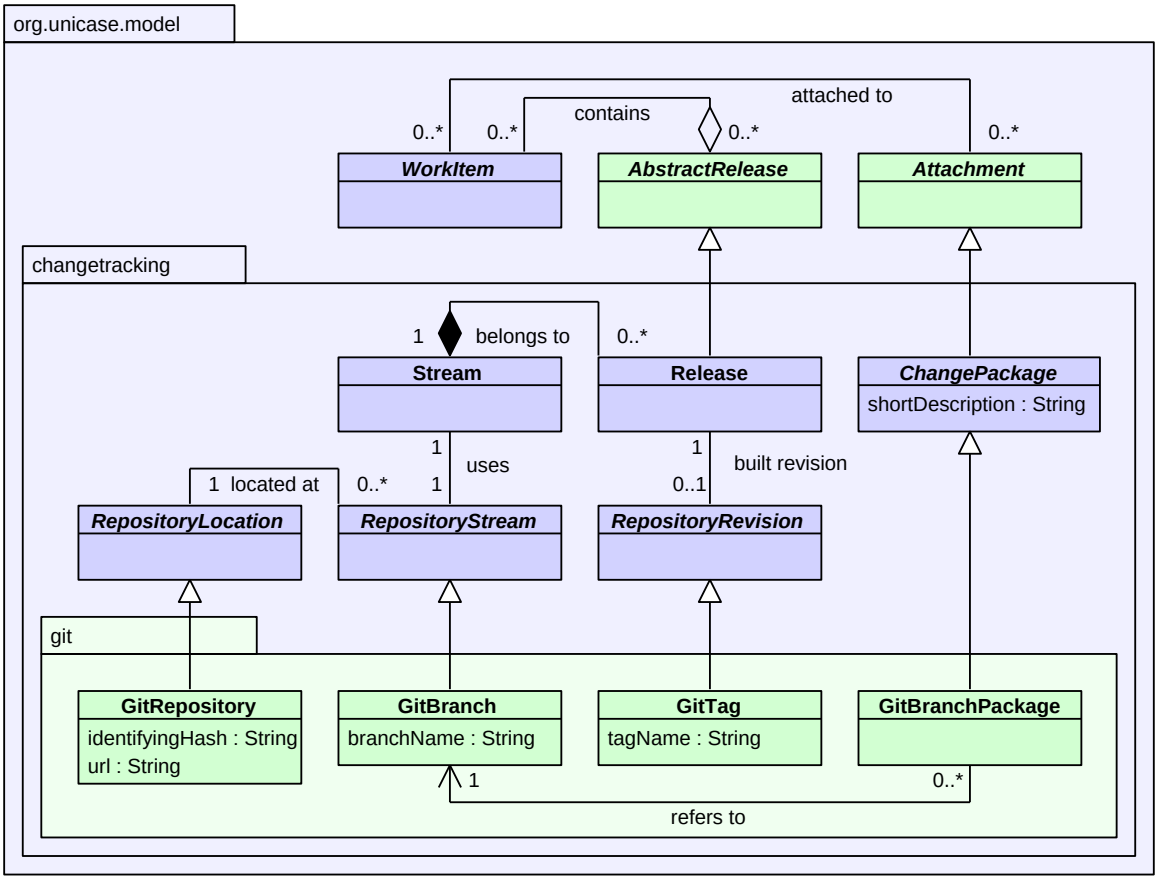


Figure 6.1.: EMF data model

Figure 6.1 shows the model after the necessary refactorings. In addition, a specific package for the Git adapter is included. Classes that were newly introduced in the model are highlighted in green color. All other facts of the model have remained basically the same as in the analysis model, with the only exception that `ChangePackage` received a `shortDescription` attribute, which is used for creating the changelog for releases and some classes were flagged to be abstract (*italic* font).

All model classes reside in the `org.unicase.model` package and its subpackages. The classes which have to be newly introduced for the prototype are located in the `changetracking` subpackage. Note that all classes in this diagram are implicitly subclasses of `UnicaseModelElement` from which any model element used in a UNICASE model must be derived. Thus all classes have basic attributes inherited from unicase model element. The most important are a `name` and `description`. Also note that all attributes shown are implicitly private, but EMF also generates `get` and `set` methods, so they can be read and written.

The first change which can be seen from this model is that the `WorkItem` is no longer inside the change tracking package, but in the usual UNICASE package[1], because the work item is already part of the unified model, as UNICASE already supports the modelling of various types of work items. In the unified model, the work item class is the abstract super class of any type of task or even a set of tasks.

The second change is that the change package is no longer directly connected to the work item. Instead, change packages now extend the `Attachment` class provided by the unified model. The model allows to add attachments to any model element. This refactoring was done so the attachment mechanism can be reused instead of adding another association to the work item class. This helps to keeps work items simple.

The final refactoring conducted is that work items are no longer directly associated to `Release`s. Instead, an abstract class `AbstractRelease` was introduced in the UNICASE core model. The reason for this design decision is that the UNICASE core model may not be dependent on the change tracking model, since the change tracking plug-in should be an optional extension of UNICASE. Introducing a dependency would make UNICASE unusable without the plug-in. Another approach would be to add the whole change tracking package to the UNICASE core. This, however, is undesirable as it would blow up the core model unnecessarily. Thus, the best way is to create a small interface between the core classes of UNICASE and the change tracking classes without introducing dependencies from the core to the change tracking package. This interface is the `AbstractRelease` class. As the inheritance does not introduce a dependency from the superclass to the subclass, the core is no longer dependent to the change tracking plug-in. The introduction of the `AbstractRelease` class and the addition of the association from the work item to the abstract release are the only two changes to the core model. All other functionality is handled inside the change tracking model, which resides in the optional plug-in.

Finally, the diagram displays the model package of the Git version control system adapter. This package is only shown as an example for a model package which any adapter must provide. Adapters must provide all these classes, because different version control systems use different information for their implementations of the abstract concepts of repositories, revisions, streams, and change packages. Thus, an adapter model must pro-

[1] Actually, the work item is in a `task` subpackage which is not shown here for simplicity reasons.

vide concrete subclasses of the abstract classes `RepositoryLocation`, `RepositoryStream`, `RepositoryRevision`, and `ChangePackage`.

For the `git` package, the implementation of a repository location is called `GitRepository`. It uses a URL to identify the location of the remote repository and a so-called *identifying hash*. This is a string representation of the SHA-1 hash of the earliest commit in the repository. It is used to check whether a local repository corresponds to a remote repository. This is important because many local repositories can exist in the workspace of a developer, and the plug-in must identify the one corresponding to a remote repository when performing actions like building a release. The implementation used by the Git adapter to represent a stream in the repository is the `GitBranch`. It contains the name of the branch, which is sufficient to find the branch in the Git repository. Finally, the `GitTag` class is an implementation of a repository revision. It contains the tag name which can be used to uniquely retrieve the revision of that tag. Another Git implementation of a revision could be a revision class which stores the SHA-1 hash of a revision, which would also be sufficient. However, it was chosen to use a tag because the tag name offers better readability. The Git implementation of a change package is the `GitBranchChangePackage` which is linked to a branch. This branch stores the changes contained in the package.

6.3. Extension Points

The Eclipse platform consists of many plug-ins. It uses the OSGi implementation Equinox to establish a dynamic component model in which components can be added or removed even during runtime. A component in OSGi is also called *bundle*. A bundle is a normal jar-file containing compiled Java classes and resources. It contains additional manifest information stating onto which other bundles this bundle depends and which Java packages of this bundle are visible for other bundles. An Eclipse plug-in is an extension of the bundle mechanism: A plug-in is an OSGi bundle with additional content. The most important additional content is an XML file called `plugin.xml`. In this file, the plug-in declares extension points it offers and extensions for points of other plug-ins.

The Eclipse platform is designed to maximize extensibility. The extension point mechanism allows plug-ins to contribute to the functionality of other plug-ins without introducing a dependency from the extended plug-in to the extending one.

A plug-in which should be extensible by other plug-ins must define an extension point in its `plugin.xml` file. The details about the extension point are modelled as an XML Schema and also included in the plug-in. In order to extend this plug-in, another plug-in must specify an extension for the advertised extension point in its `plugin.xml`. It must provide details to the extension point which must correspond to the XML schema defining the extension point. Thus, the schema defines which information an extending plug-in must provide to extend the point. The extending information can be ordinary data like strings or numbers. However, most of the time it is the fully qualified name of a Java class.

The core concept of passing classes to extension points is very close to the *dependency injection* design pattern, which aims to relocate the instantiation of objects to an external framework. This breaks the dependency which the creating class must have on the class of the object it intends to create[2]. Instead of directly creating a new instance (`new X()`

[2]The factory method design pattern breaks this dependency, too, but introduces a dependency on the factory

in Java), it asks the framework to provide an instance implementing a specific interface. Dependency injection is a core concept of plug-in based platforms, because it allows to write code which will later instantiate classes that do not even exist at the time the code is written, since no more dependency on the instantiated class exists in the code.

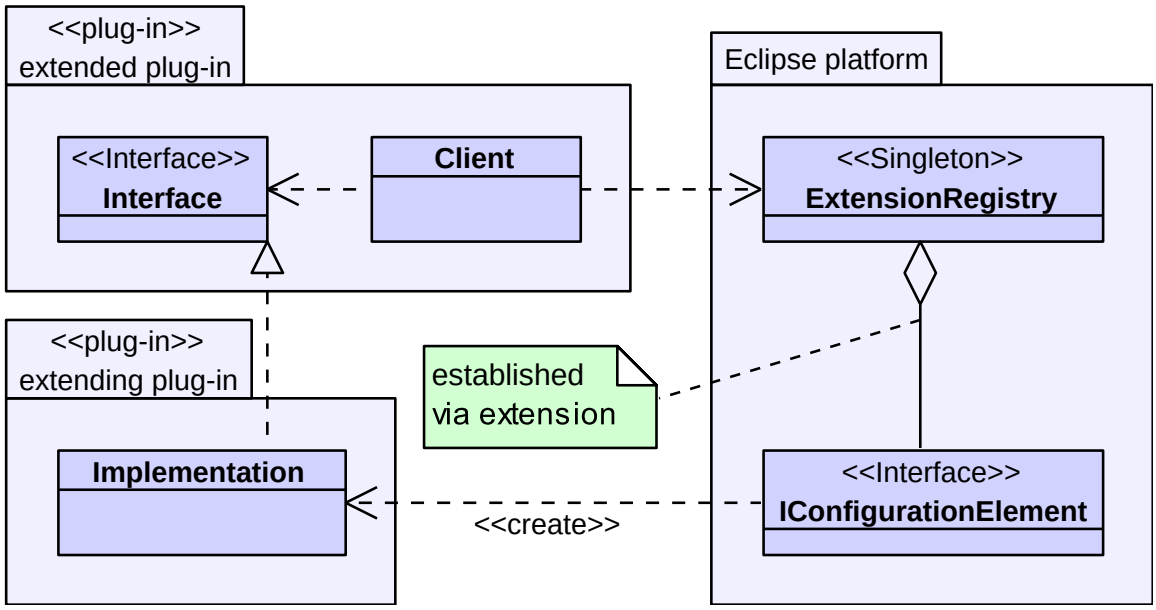


Figure 6.2.: Extension point usage

The concept of extending plug-ins by passing classes via extension points is shown in Figure 6.2. The figure shows a UML class diagram in which the different plug-ins are depicted as UML packages with a «`plug-in`» stereotype. The Eclipse platform itself is also shown as a UML package. The extended plug-in usually contains the code working with the extensible data. This code is here shown as the `Client` class. Like in the dependency injection pattern, this code must instantiate an object of a class which implements a certain `Interface`. The extending plug-in provides an `Implementation` of this interface. To avoid the dependency on the extending plug-in, the client code cannot instantiate the implementation class directly. Instead, it defines an extension point and requires in the extension point definition that each extension must provide a class which implements `Interface`[3]. The extending plug-in provides the `Implementation` class. It therefore has a dependency on the extended plug-in. This dependency, however, is desired. Only vice versa, from the extended plug-in to the extending plug-in, is undesirable. The extending plug-in defines an extension in its `plugin.xml`. An XML directive could look like this (provided the extension point is named `X`):

```
<extension point="X">
     <cls class="XImplementation" />
</extension>
```

class which in turn has the dependency on the class to be instantiated. Thus, the dependency still exists, but is transformed into a transitive one.

[3] In addition, the class may also be required to extend a certain class or more than one interface.

This code specifies that the extension point named `X` is to be extended with the class `XImplementation`. The Eclipse platform contains a singleton class called the `ExtensionRegistry`. It reads the `plugin.xml` files of all currently installed plug-ins and thus has information about the extension points and corresponding extensions. To receive an instance of `Interface`, the `Client` asks the extension registry for any extensions to the extension point `X`.

The extension registry maintains a set of `IConfigurationElements` for each extension point. Each element contains the information about one extension of this point. The configuration elements provide a method to create an instance of the class specified in the extension. The client code chooses from the configuration elements it receives from the registry and calls one of them to create the instance.

The Eclipse platform provides various extension points used by the prototype plug-in. Also the prototype defines an extension point for providing adapters for different version control systems. This way, adapters to new version control systems can be implemented as dedicated plug-ins and added without modifying the prototype itself. Further information about this mechanism is shown in Section 6.6.

6.4. Displaying Model Elements

Displaying the EMF model elements is a central part of the user interface of the plug-in. The model elements are presented in different structured views for the various use cases. For example, in the details page of the check-release-wizard the work items and change packages contained in the release are displayed in a tree-like structure, where the change packages are considered child-nodes of the work items to which they belong. Figure 6.3 shows an example release containing three work items, each having one attached change package.

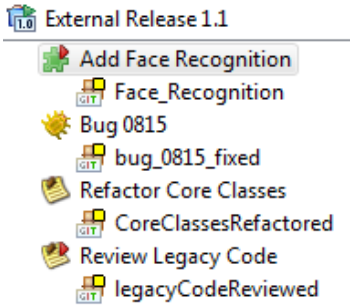


Figure 6.3.: Hierarchic displaying of model elements

Another structural view of model elements is a flat list. Such list can be used when one or model elements are to be selected out of a set. For example, in the *Create Change Package* use case, the work item to which the change package is to be appended has to be chosen. In the dialogs used to choose these model elements, all available model elements are displayed in a list. In addition to structured views, single model elements are displayed at various locations in the user interface.

To achieve a good user experience, model elements are displayed with their name and

an icon matching their type (see again Figure 6.3). By choosing appropriate images for the newly modelled element classes, the user is able to quickly identify what kind of model element he is seeing. For the change package, a package like icon as known from Java-packages in the Eclipse Java IDE is chosen, for example. In addition to newly created model elements, the plug-in must also be able to display existing model elements of UNICASE, like the different work items, which already have a dedicated icon and label.

While the model elements are displayed with label and icon, the icon is overlayed with small additional images — so-called decorations — to display some details about the model element. For example, a release is overlayed with a tick symbol if it is already built.

Concerning the classes used to display and decorate model elements efficiently, three major challenges were identified and solved using features of the Eclipse platform and the Eclipse Modelling Framework. The following subsections will elaborate the mechanisms used to solve each of these challenges in detail. The challenging problems are:

1. Displaying structures of model element like the above-mentioned trees and lists while keeping a sharp separation of the model and the user interface code.

2. Generically providing labels and icons for any type of model element. Displaying existing UNICASE model elements without writing additional glue code.

3. Overlaying decoration images over the model element icons.

As a basis for all user interface components, the standard widget toolkit (SWT) as provided by the Eclipse platform is used. SWT is a versatile user interface toolkit which allows to assemble user interfaces from different parts, the so-called widgets. Examples for basic widgets are `Button`s, `ScrollBar`s, or windows (called `Shell`s in SWT). While other user interface frameworks for Java like Swing are purely written in Java, SWT's core consists of platform-dependent C code which accesses the native user interface components of the respective operating system. SWT therefore runs faster than pure Java frameworks and SWT widgets resemble the look-and-feel of native applications for the respective operating system better than their Java pendants. This comes with the cost that an own library must be provided for each operating system on which an SWT application is to be used. Due to the widespread use and maturity of SWT, libraries exist for all major operating systems. A small problem is that these libraries are not 100% compatabible to each other, since different operating system user interfaces provide different sets of features. When using MacOS, for example, buttons always have a fixed height, while other operating systems allow arbitrary height settings. The Eclipse platform is built on SWT and provides useful add-ons like the JFace UI toolkit.

6.4.1. JFace Viewers

The displaying of model elements in different structural views like lists, tables or trees is achieved by using the JFace toolkit provided with the Eclipse platform. This toolkit provides different `Viewer` classes for displaying data in different structures. The main purpose of these viewers is to provide model-based content adapters for SWT widgets.

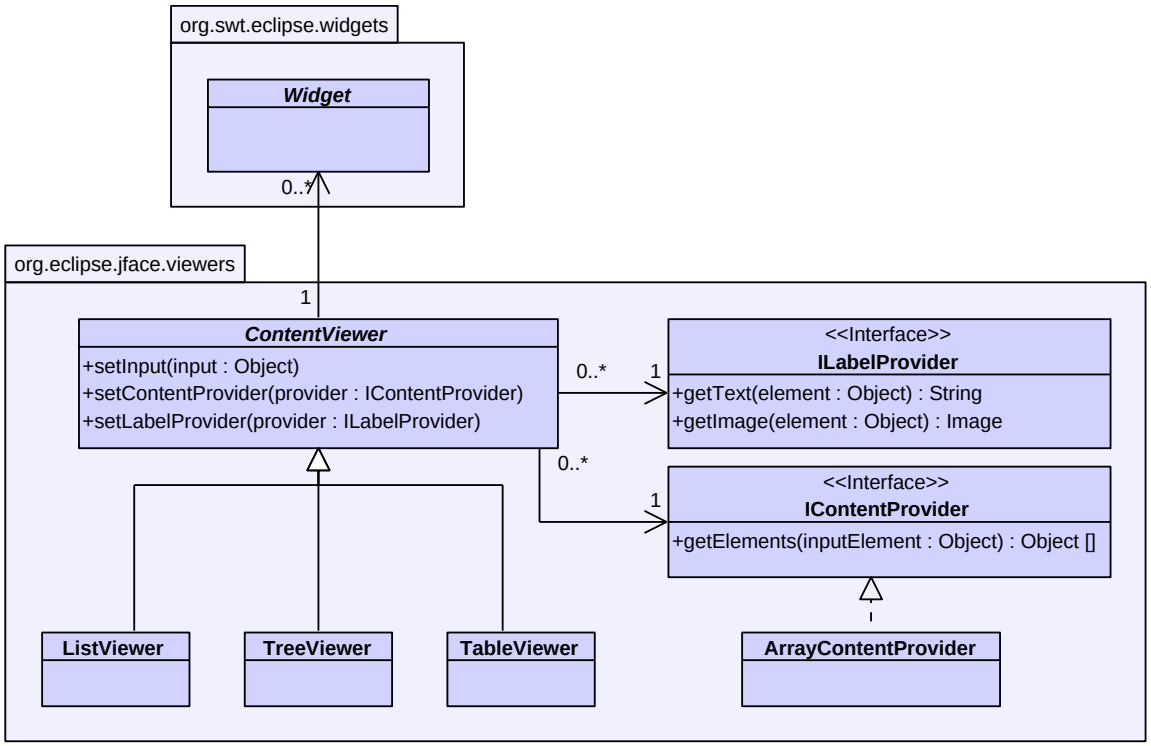


Figure 6.4.: The JFace viewers

Figure 6.4 shows a class diagram depicting the most important classes of the JFace viewers. The abstract base class of the viewers is the `ContentViewer`. An `IContentProvider` and an `ILabelProvider` must be assigned to a viewer. Then it can be populated with an input. The viewer sharply separates the different tasks to be performed to display an input model: The content provider is responsible for extracting the structure to be displayed out of the model. The label provider is then responsible for providing a text and icon for each object of the input model which was extracted by the content provider.

The typical usage of a viewer is to assign a label provider and content provider upon its creation by using the correspondent `set` methods. Then, the viewer can be populated with an input model by calling the `setInput` method, taking the model as parameter. The parameter type is `Object`, depicting that any type of model can be set as input[4]. Once the `setInput` method is called, the viewer asks the content provider to extract the structure from the model by calling its `getElements` method with the input model as parameter. The content provider will then extract the structure and return all elements, which are to be displayed in the viewer, as an array.

Note that the class diagram was simplified: There are more methods which content providers must support to fully populate different viewers. For a tree viewer for example, the content provider must also implement a `getChildren` method to extract the child nodes of a certain tree node from the model. In order to keep the focus on the basic concepts, this, and other details, will not be discussed further. For lists and tables, the

[4]This could have also been solved more elegantly with type parameters and generic classes. However, the JFace toolkit should also be able to run with older Java versions which do not yet support generic classes.

`getElements` method is sufficient.

The content provider must be aware of the inner structure (i.e. the class) of possible input models, since it only receives an `Object`. Thus, a corresponding content provider must be implemented for all classes to be used as input. JFace provides basic content providers for widely used input models. For example, the `ArrayContentProvider` displayed in the diagram assumes that the input model is an array. It simply casts the input variable to an array and returns this.

The `getElements` method of the content provider returns an array of objects. This array depicts the elements which will be displayed in the viewer. For example, a list viewer will show each array element in one list item. The table viewer will display each element in a table row.

Since the structure of the input is now present to the viewer, only one task is missing before the viewer can actually populate the underlying SWT widgets: Text and icon have to be provided for each element to be displayed. This is done by the label provider associated with the viewer. The viewer calls the `getText` and `getImage` methods, repeatedly handing each of the input elements as parameter. It is the label provider's task to examine the input element and return a matching label (like the name of the element, for example) and icon, respectively. Since the methods receive the input elements as bare `Objects`, the label provider must be implemented to match the possible input classes. After the viewer has gathered all the images and texts for the different input elements, it can display them appropriately by creating and altering a set of SWT `Widgets` it consists of.

There are three major viewers in the framework, as already mentioned and depicted in the class diagram: The `ListViewer` shows the input elements in a plain list. Each list item has its own line. The `TreeViewer` shows the input as a tree-like structure. Finally, the `TableViewer` shows the elements in a table. Each row corresponds to a model element. The table can have many rows, and each row can be assigned a different label provider to show various information about the input elements.

Advantages of the Viewer Concept

Viewers depict model-based wrappers of SWT widgets. SWT in contrast only provides, widgets which do not allow to specify an input model and a way to display it. Instead, they can only be populated with bare strings and icons. This especially harms the traceability of user input: If the user selects an entry in an SWT list, the system can only tell which line number was selected. If the client code needs the actually selected object, it has to retrieve the object itself. This is usually done by maintaining mappings from line number to associated input object. This is cumbersome, and the resulting client code is bug-prone. In addition this mapping code will probably be written repeatedly, thus introducing redundant code. The JFace viewers provide the traceability from the user input back to the model, since the viewers are populated directly with the model instead of strings.

Another advantage is the sharp separation of concerns: Instead of decoding the data in the model itself, the viewer delegates the different tasks to the providers: The structure is decoded by the associated content provider and the decision how to display each element of the input is taken by the label provider. This high degree of delegation makes the viewer concept very flexible: By using different content providers, a data-structure can be displayed in completely different ways. Thus, two different views of the very same model

can be displayed without having to alter the model or viewer code.

By using different label providers, elements can be shown in different ways. Especially the icons can be altered by different providers. An example for this is the details page of the check release wizard: A label provider is used which decorates the change package icons showing whether they are already merged in.

The framework can be extended easily by creating separate viewer classes. For the implementation of the plug-in, for example, a basic "one item" viewer was implemented, which only displays one object by showing an icon and the text next to each other. A simple content provider was written for this viewer, which assumes that the input consists of only one element.

6.4.2. EMF Label Providers

The JFace viewer concept, which makes use of label providers to display text and icons for input models, was elaborated in the previous section. The data model of this plug-in is the one of UNICASE which is modeled using EMF. To display UNICASE models with a viewer, a label provider must exist which is able to provide an icon and text for all possible model elements a UNICASE model can contain. Usually each model element class has its own icon and the name of the model element is used as text.

The challenge is to create a label provider for all possible elements contained in the unified model. The main problem is that these classes can come from different sub-plug-ins which are not necessarily known at the time the plug-in is implemented. Luckily, EMF provides a way to construct label providers for arbitrary EMF models. By using an extension point, the actually supported set of model elements can be extended dynamically, thus allowing existing code to support models that will be created at a later stage.

Concepts

The foundation for these versatile label providers is that EMF generates a basic label provider for each model element class. This label provider is only able to provide text and icons for this very class. EMF classes are always contained in a package. For each of these packages, EMF generates an `AdapterFactory`. Adapter factories combine the abstract factory design pattern with the adapter design pattern. In the adapter pattern, an adapter is a class that provides a specified interface by wrapping a class which does not provide this interface by default. This is often used to adjust the interface of legacy code to match the interface of newly created code. In this case, an adapter is used to provide the `ILabelProvider` interface to EMF model element classes. The adapter is created by an adapter factory. Such a factory provides an `adapt` method which takes an object (the adaptee) and a type stating to which interface the adaptee should be adapted. If possible, the factory then creates and returns an adapter for the specified adaptee, adapting it to the specified interface.

The adapter factory which is generated by the EMF framework for each package is able to take an instance of each of the classes in its package as adaptee. It can adapt it to various useful interfaces. The `ILabelProvider` interface is one of them. When asked to adapt one of the classes in its package to a label provider, it looks up the generated label provider class corresponding to the adaptee's class. It then creates an returns an instance of this label

provider class. This mechanism already allows to generically retrieve a label provider for any model element contained in a package. However, a UNICASE model consists of model elements from different packages, so some more effort has to be applied to generate a label provider for any given model element class in any given package.

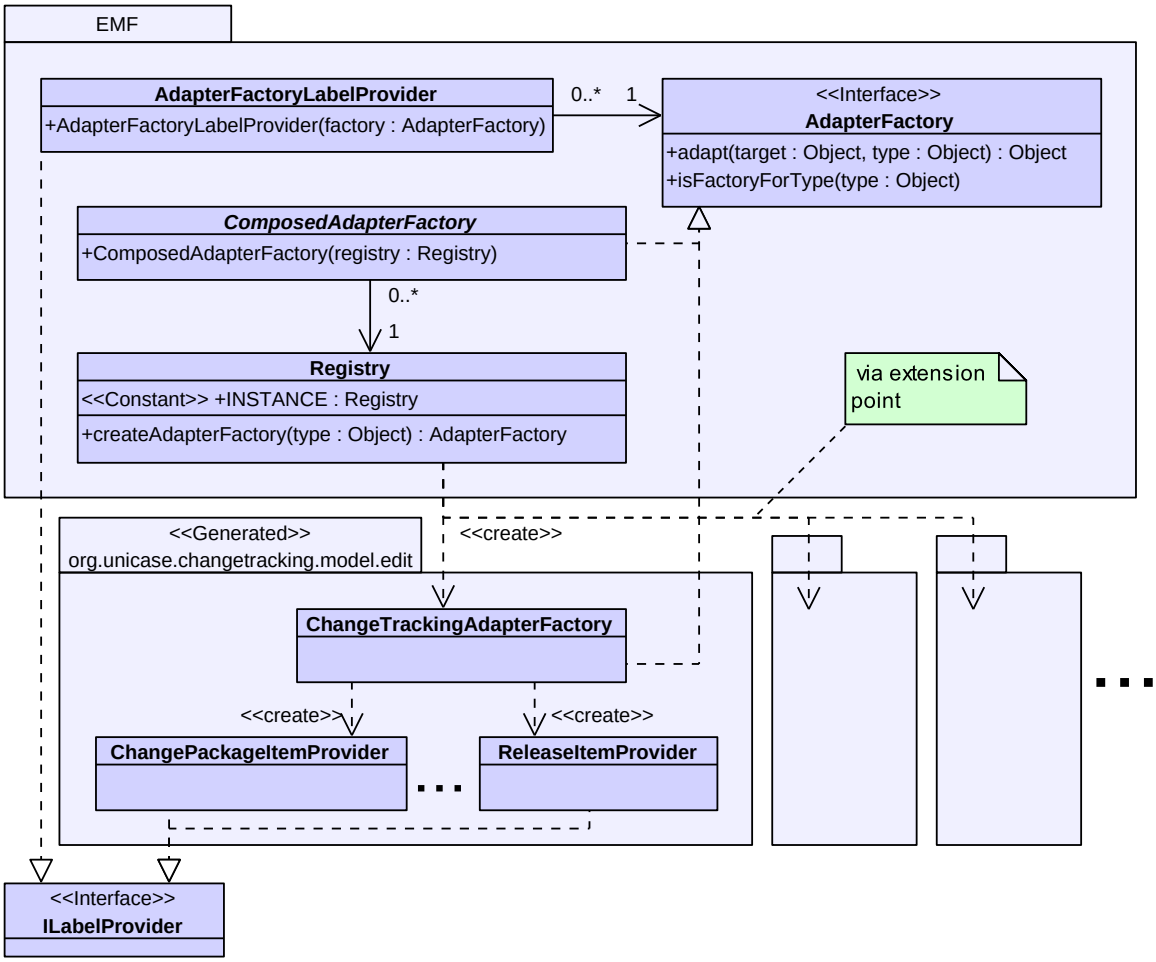


Figure 6.5.: Classes for using EMF label providers

Figure 6.5 shows a UML class diagram depicting all classes necessary to provide generic label providers. The lower packages depict the different model element packages included in the unified model (or other EMF model). The code for all these packages is generated by EMF. The package for the classes of the prototype (`org.unicase.changetracking.model.edit`) is shown in more detail. The «`generated`» prototype was used to depict that all classes in the package are generated by EMF. It is called the edit package of the corresponding model package. While the model package contains the classes depicting the model elements themselves, the edit package contains useful classes for working with the model elements.

The edit package contains a label provider for each class in the corresponding model package. Here, the provider for the release class (`ReleaseItemProvider`) and the change package class (`ChangePackageItemProvider`) are shown exemplarily. The periods be-

tween them depict that there are many other providers in the package. They are not called label provider but *item provider* because they do not only implement the `ILabelProvider` interface but other interfaces, too. The package also contains the aforementioned adapter factory, named after the package name (`ChangeTrackingAdapterFactory`). This factory, which implements the `AdapterFactory` interface, is able to adapt all model element classes of the change tracking package to interfaces like the `ILabelProvider` interface by creating and returning the appropriate item provider. Besides an `adapt` method, the `AdapterFactory` interface also provides a method `isFactoryForType` which returns whether the factory can adapt to a given type.

In addition to the change tracking package, there are many more packages included in the UNICASE EMF model. In the diagram, the periods right of the packages depict this fact. All packages which should be included in the model are registered by creating extensions to the extension point `org.eclipse.emf.ecore.generated_package`. The class reading from the extension point is the `Registry` class. This singleton class is able to create the appropriate adapter factory for a EMF model element whose package was registered at the aforementioned extension point. This is accomplished by querying the model element for its package and then creating the factory of this package, if the package is among the registered ones.

The registry can be wrapped by a `ComposedAdapterFactory`. This class, which also implements the `AdapterFactory` interface, is created from the registry[5]. Its `adapt` method asks the registry to create an appropriate adapter factory and then forwards the call to this factory.

Finally, there is the `AdapterFactoryLabelProvider` which wraps an adapter factory while providing the label provider interface: It is created with an adapter factory and maintains a link to this factory. Whenever one of its `get` methods is called, it calls the factory's `adapt` method which provides the corresponding label provider. The `get` call is then forwarded to this label provider.

Dynamic Behaviour

Since different concepts are involved in the generic use of label providers, a quite complex dynamic behaviour evolves. To generate a generic label provider, the client code must first create a `ComposedAdapterFactory` from the `Registry`. Then, an `AdapterFactoryLabelProvider` can be created, wrapping the adapter factory. This is the generic label provider which is able to provide text and images for all EMF model elements.

Figure 6.6 shows an UML communication diagram depicting the dynamic behaviour calling the `getImage` class for a model element of the class `Release`. The call is made to the previously created `AdapterFactoryLabelProvider` (1). First, the label provider asks the `ComposedAdapterFactory` to adapt the release object to a `ILabelProvider` (2). The adapter factory in turn orders the `Registry` to create the adapter factory corresponding to a release (3). The registry looks up the package of the `Release` class (the change tracking package) and creates and returns the corresponding `ChangeTrackingAdapterFactory` (4). The composed adapter factory now forwards the `adapt` call to the `factory` it received from the registry (5). This factory creates the label provider for the

[5] It can also be created by directly specifying a list of other adapter factories. Since this is not important for the prototype, it is omitted here

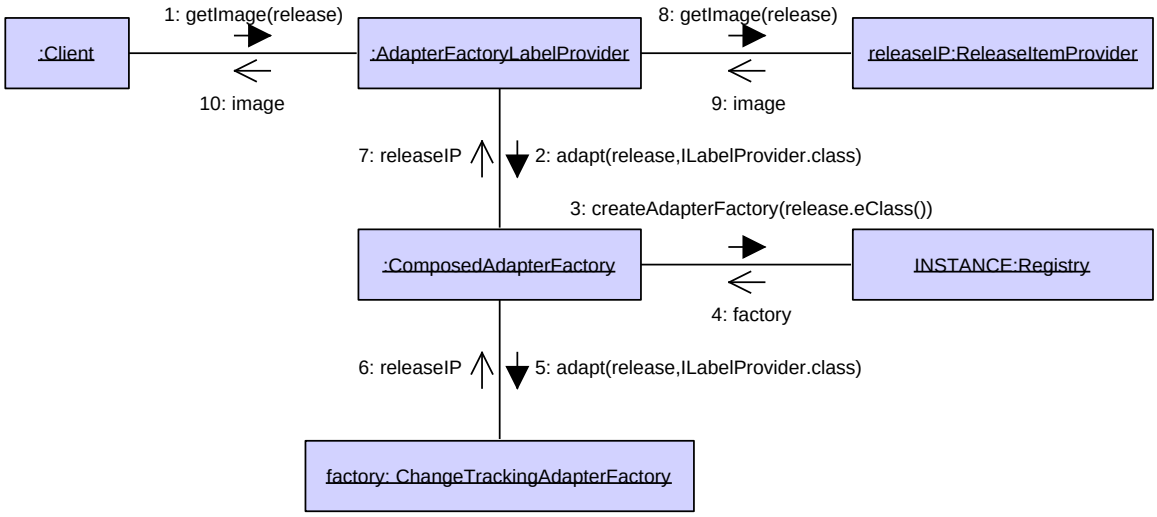


Figure 6.6.: Dynamic behaviour of EMF label providers

release class (`ReleaseItemProvider`) and returns it to the (6) composed factory, which also returns it to the adapter factory label provider (7). This label provider now forwards the `getImage` call to the label provider it received from the factory (8). The release label provider creates and returns the image for releases (9) and the adapter factory label provider returns this image to the client (10).

Summary

In summary, EMF is able to provide generic label providers for model classes by the following mechanisms:

1. A generated adapter factory per package can create providers for model classes in the package.

2. The `Registry` registers all EMF packages which were advertised through the extension point. A `ComposedAdapterFactory` can access the registry to generate the adapter factory of the package a model class is contained in. Thus, the corresponding label provider can be created for any EMF model class.

3. The `AdapterFactoryLabelProvider` wraps an adapter factory to allow easy use of the factory as label provider.

6.4.3. Decorating Images and Text

The last challenge related to displaying model elements is the so-called decoration of the text and particularly the images. Decorating an image means overlaying small sub-images which show details about the model element. This concept is extensively used in Eclipse. The prototype uses decorations as well: A release, for example, is overlayed with a tick symbol if it is already built, otherwise with a “play” symbol (triangle facing right). Decorating text means altering this text. For example, some version control plug-ins prefix

the name of files with a ">" symbol if the local version of the file contains changes. In the prototype, the decoration of text was not used and thus will not be discussed further.

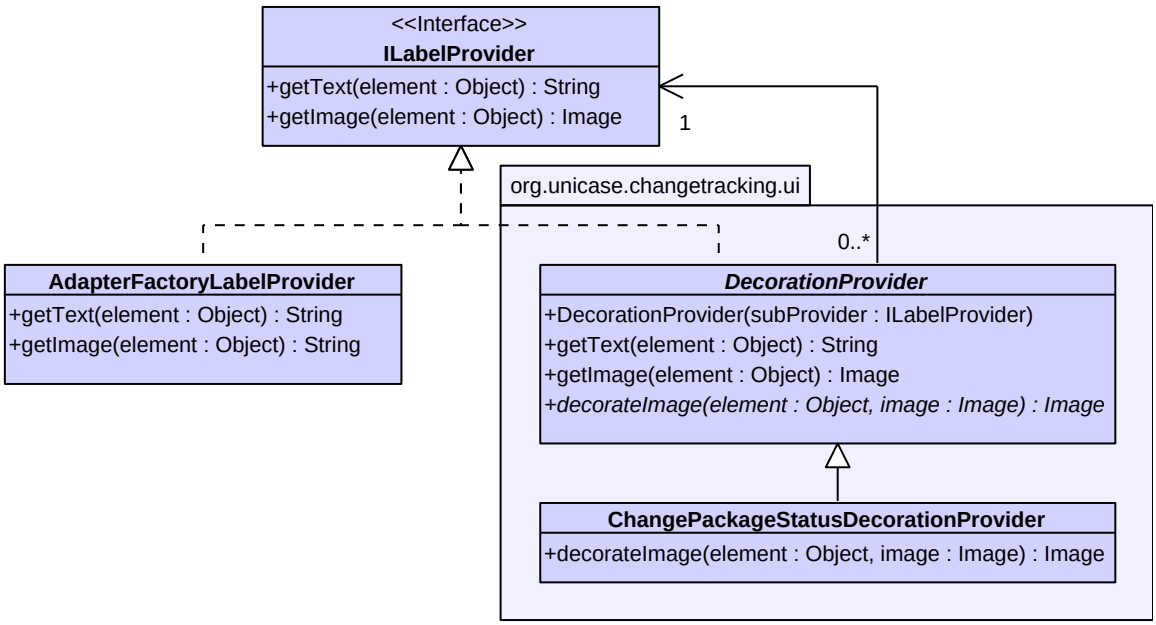


Figure 6.7.: Decorating images and text

The task of decorating images was accomplished using the similarly named decorator design pattern as shown in Figure 6.7. Displayed here are the already discussed interface `ILabelProvider` and its implementation `AdapterFactoryLabelProvider` which can be used to provide labels for arbitrary EMF objects. Another class implementing the label provider interface was created to achieve the decoration: the abstract `DecorationProvider`. Its constructor uses another label provider as parameter and keeps a link to this provider. As no decoration of text is desired, calls of `getText` method are directly forwarded to the linked label provider. The `getImage` method first fetches the basic image by calling the linked provider's `getImage` method. Then, however, it calls its `decorateImage` method to overlay additional images over the basic image. The result of the decoration is then returned. The `decorateImage` method is abstract, so the concrete decoration has to be implemented by concrete subclasses. An example subclass shown in the diagram is the `ChangePackageStatusDecorationProvider` class. It is used in the check release wizard to display the stats of change packages. As shown in Figure 6.8, they are either overlayed with a tick symbol if already merged into the release, with a yellow square if not yet merged, or with a red cross if they are erroneous.

Figure 6.8.: Package status icons generated by a decorator

6.5. Command Execution

UNICASE tracks changes applied to the model and supports actions like undoing them. The basis for tracking these actions is the command design pattern. Figure 6.9 shows a UML class diagram depicting the command pattern as provided by EMF and used by UNICASE and the prototype. A `Command` is an action which possibly alters the model. It can be `execute`d and possibly supports `undo` and `redo` operations. The `canX` methods indicate whether the feature `X` is available. For keeping a history of executed commands and for allowing to undo more than one command, EMF keeps a `CommandStack`. A command can be executed on this stack by passing it to the stack's `execute` method. The stack will trigger the command's execute method and store it for later undoing and history keeping. An `EditingDomain` manages a set of EMF models and the commands working on these models. For this purpose, it uses a command stack. Commands working on the models of an editing domain are usually created through the domain and executed on the domain's command stack. The `RecordingCommand` is an implementation of a command which automatically tracks the changes it does to models in the editing domain. EMF maintains a global editing domain which can be accessed via static get methods. In most cases it is sufficient to use this global editing domain.

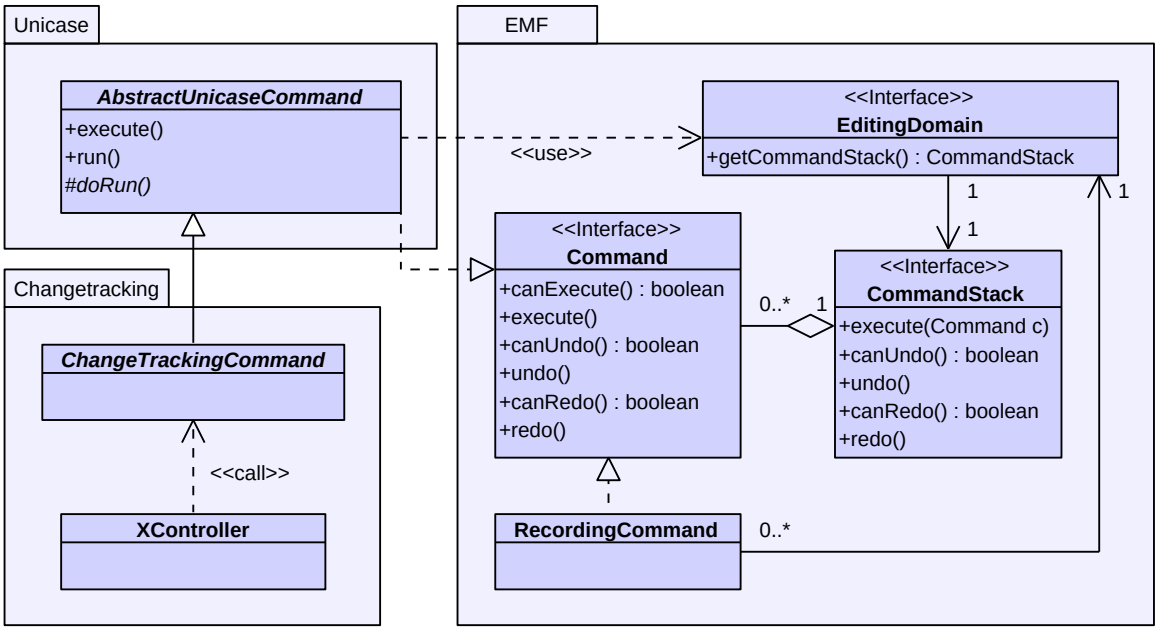


Figure 6.9.: Command execution using EMF

The base class for each command in UNICASE is the `AbstractUnicaseCommand` which is also an implementation of a `Command`. Its `execute` method calls the abstract `doRun` method. Thus, a concrete UNICASE command is bound to a specific receiver by appropriately implementing the `doRun` method which calls the receiver. However, the command often implements the functionality itself, without delegating to a receiver class. In this case, the command is no longer decoupled from the receiver. This disadvantage is of limited importance because this part of the command design pattern is not relevant for

UNICASE; what is important here is the tracking of changes a command does to the model. A UNICASE command also provides a `run` method. This method executes the command on the stack of the global editing domain of EMF. If this domain supports transactions, it will automatically wrap the command in a recording command, thus recording all changes and providing undo and redo support.

The change tracking prototype wraps most of its functionality into commands. For this purpose, the abstract `ChangeTrackingCommand` class was created as a subclass of the UNICASE model element class. The controller classes of the different use cases run concrete subclasses of these commands to execute the functionality of the respective use case. In the diagram, the class `XController` is a placeholder for any controller class of the use cases which calls the commands. The concrete implementations are provided by the version control system adapter component to achieve behaviour unique for the specific version control system. This is shown in the next section.

6.6. Adaption to Different VCSs

One of the design goals is to allow the collaboration with various version control systems. Consequently, the part of the functionality which is identical for all version control systems has to be separated from the part which is specific to each version control system. There must be a way to support additional version control systems without having to change the core code. To achieve this, a special plug-in is created for each version control system. This plug-in must contain an adapter class which wraps the version control system classes and provides a fixed interface to the core code. To support the addition of adapter plug-ins without changing the core code, the core plug-in provides an extension point to which the adapter plug-ins advertise their adapter classes.

Figure 6.10 shows an overview of the classes used to support various version control systems. All the classes shown in the `VCS Adaption` package reside in the core plug-in. The classes in the `Git` package are provided by the Git adapter plug-in. Other adapter plug-ins must provide similar classes.

The core class for the adaption is the `VCSAdapter` interface, for which each adapter plug-in must provide an implementation. It defines different methods to be provided by the adapter. There exists roughly one method per use case. In the diagram, the methods `createChangePackage` and `buildRelease` invoked in the equally named use cases are shown. The periods below these methods indicate that the adapter provides more methods for other use cases, which are of no importance here. The parameters for the methods were omitted as well. Of course, each method must take all important information as parameter. For example, the `buildRelease` method must be given the release to be built and a `properties` object which was obtained while checking the release. As shown in the diagram, the methods of the adapter class return a `ChangeTrackingCommand`. Since this class is abstract, all adapters must provide concrete implementation for each command the adapter provides. The core plug-in calls the adapter's method to obtain the concrete command object. This object is then executed by the core plug-in, as shown in the Section 6.5.

Before the client code (a controller class for one of the use cases) can execute a command for a certain use case, it has to obtain it from an adapter. For this purpose, it can use the

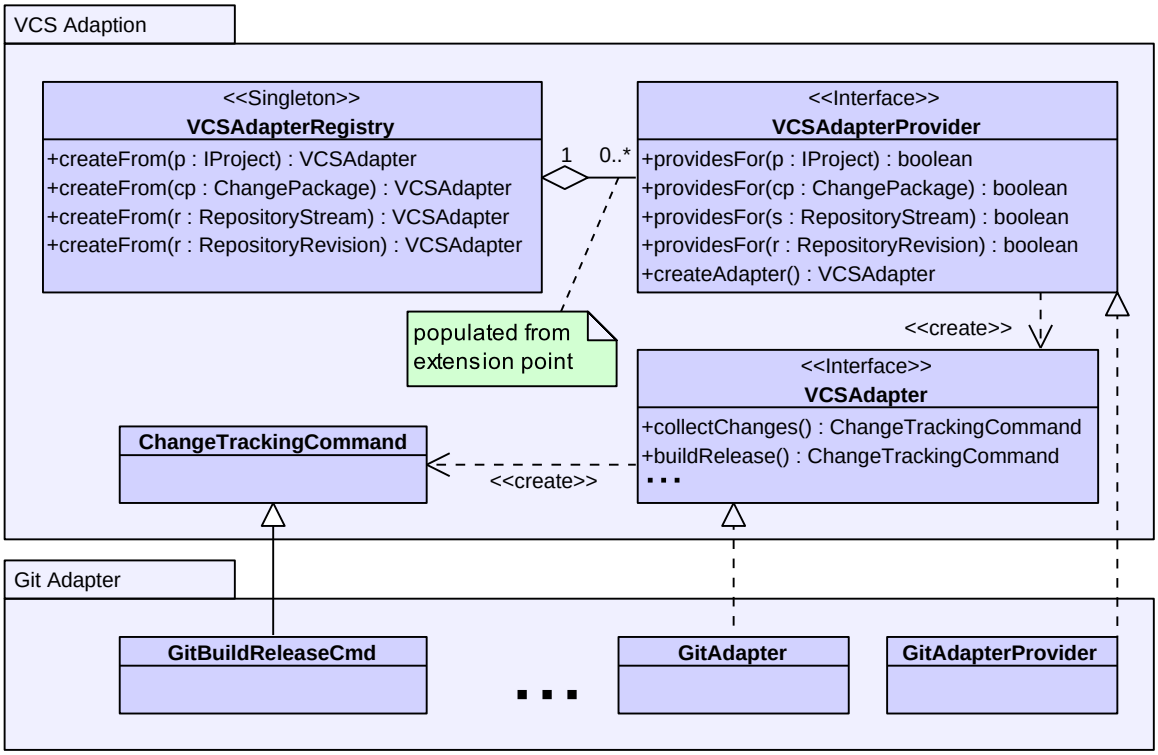


Figure 6.10.: Adaption to different VCSs

`VCSAdapterRegistry` singleton object. This object implements the registry which reads the extensions from the provided extension point (cf. Section 6.3). The extension point definition requires its extensions to provide a class which implements the `VCSAdapterProvider` interface. The registry keeps an aggregation of all these provider classes registered via the extension point.

The first method a `VCSAdapterRegistry` must provide is the `createAdapter` method. This is a factory method which creates and returns the corresponding adapter. In the Git adapter plug-in shown in the diagram, the `GitAdapterProvider` class would create an instance of the `GitAdapter` class. The other four `providesFor` methods are indicator methods. Given a workspace project (`IProject`), a change package, a repository stream, or a repository revision, the respective indicator method must return whether this adapter plug-in is able to provide an adapter for the given object. For workspace projects, an implementation of this indicator method will probably check which team provider is registered for this plug-in and return `true`, if the team provider matches the version control system adapted by this plug-in. For streams and repository revisions, an implementation will simply check whether the objects belong to the concrete subclass supported by this adapter. The Git adapter will check, for example, if a revision is an instance of `GitTag`, a repository stream is an instance of `GitBranch`, or a change package is an instance of `GitBranchPackage`.

The registry contains four `createFrom` methods corresponding to the four `providesFor` methods of the adapter provider. These methods ask all registered providers one by one if they can provide an adapter for the given object by calling their corresponding `pro-`

`videsFor` method. Once a provider returns `true`, the registry calls its `createAdapter` method to create an adapter and returns this adapter to the caller.

The reason for the existence of four `create` methods is that, depending on the use case, the correct adapter must be found, starting from either a change package, a repository revision, a repository stream, or a workspace project. In the *Create Change Package* use case, for example, the adapter must be determined by observing the workspace project from which the change package is to be created. In contrast, the *Review Change Package* use case starts with only a change package and must determine the adapter from this object.

The dynamic behaviour is the following: The client code (a controller class) calls the appropriate method from the registry to create an adapter. The registry asks all registered adapter providers. If one of them can provide an adapter, the registry calls its `create` method which yields the adapter. The registry returns the adapter back to the controller class. This class calls the adapter's method corresponding to the use case handled by the controller (e.g. `buildRelease` for the *Build Release* use case.). A command object is created and returned by the method. Finally, this command is executed by the controller to trigger the desired behaviour.

Every adapter plug-in must provide classes like the ones shown exemplarily for the Git adapter: It must provide a `VCSAdapterProvider` and register it at the extension point of the core plug-in. It must also provide implementations for the different use cases as subclasses of the `ChangeTrackingCommand` class. In case of the Git package, the `GitBuildReleaseCommand` is shown as an example of a command implementing the *Build Release* use case. The periods on its right suggest that there must be more classes in the plug-in for the other use cases. Finally, the plug-in must provide a `VCSAdapter`. This adapter must be created by the adapter provider of the plug-in. Its methods must create and return the respective command implementations.

Chapter 7.

Conclusion

After the previous chapters covered the design of the prototype, this chapter evaluates the results of this thesis. The evaluation starts with the results of a user experiment, which rates the anticipated user acceptance of the prototype. Afterwards, the two approaches for representing change packages are compared, and advantages and deficiency of each approach are highlighted. In the following, the content of this thesis is summarized retrospectively. The thesis finishes with the proposal of topics for future research.

7.1. Evaluation

Since this thesis had a theoretical part (approaches for change package representation) and a practical part (application of the approaches to aid release management and reviews), two different aspects have to be evaluated. For the practical application, the user acceptance is an important factor and was therefore evaluated by executing a small user experiment. For the theoretical part, the two approaches were compared, summing up their advantages and disadvantages.

7.1.1. User Acceptance

Tool support must be accepted by the users to yield any benefits. A factor which greatly influences the acceptance of the user is how complicated the usage of the tool is, how fast it can be learned, and how much time the different use cases take compared to the execution without tool support. The prototype performs well in all these areas: Most use cases are realized with only a few clicks, and no complex configurations have to be performed. For example, the *Apply Change Package* use case is usually only one click. In contrast, conducting the application without the tool would require to identify the correct revision/patch to be applied and then trigger the application manually, which is by far more work. The creation of a change package is only one click, too, followed by the selection of the work item to which the change package is to be associated and the entering of a description. This is not considerably more work than checking-in the code to a repository without using the tool.

The great advantage of the tool becomes obvious when checking and building a release. This process has been reported as very cumbersome by the person building the releases for the UNICASE project. With the tool, checking and building a release is basically only two clicks. In addition to the automatic building, the tool also provides a changelog which would have to be maintained manually without the tool and may not be accurate. Of

course, building a release may yield conflicts. And, as elaborated in Section 2.2.4, resolution of conflicts is a very complex task. However, this problem would also occur without the tool, so this is no inconvenience of the tool but of the process in general.

Experiment Setup

Five post graduate computer science students with background knowledge in the field of version control systems, but no knowledge about the tool, participated in a limited user experiment. Each of them was monitored separately while performing the following tasks: First, the participants were shortly explained all use cases, i.e. were told the purpose of the use case and its steps while performing them at a fully set-up eclipse installation with UNICASE, JGit/EGit, and the plug-in (with the Git adapter) installed. Afterwards, they were given a clean eclipse workspace with only an empty java project under Git version control and a UNICASE project in it. They were told to perform the use cases one by one. They were given the possibility to receive additional help by asking questions, but were told to use this possibility only if they are unable to continue on their own. Afterwards, they were asked the following questions. The possible answer choices are displayed in parenthesis:

- Is the tool more complicated than using the version control system directly? (Tool more complicated / Equally complicated / VCS more complicated)
- Do you think that the tool is useful if used in software development projects? (Very useful / Useful / Hardly useful / Not useful at all)
- If you had to decide if the tool is used in a software development project, would you decide to use it? (Yes / No / Depends on project or other factors)

After having answered the questions, the participants were asked to explain the reasons for their respective answers. In addition, they were permitted to talk freely about their opinions and experiences concerning the tool.

Results

All of the participants were able to perform all use cases without calling for additional help. Although they had to search the correct buttons to start the use cases for some time, because they were not familiar with the user interface of UNICASE, they were able to perform the use cases not considerably slower than the person who demonstrated the use cases (who is familiar with the tool and UNICASE).

All participants agreed that using the tool is not more complicated than using the version control system directly. In fact, two participants found the tool to be easier. When being asked about the reason for their decision, they described the dialogs as being tidier and having less unnecessary input fields than the ones of the version control system. When being asked to rate the tool, four participants rated the tool as very useful and stated that they would use it in software projects, especially bigger ones. One participant replied that he would not use it, because he saw no sense in release management and traceability from tasks to changes.

When talking freely about their opinions and experiences with the tool, the majority was positively surprised by the release building process, as it combined their previous changes in seconds, without additional input, and generated a changelog automatically. They assumed that this functionality is the one which might save the most time.

In conclusion, the user feedback was very positive and yielded high acceptance. However, it is obvious that the results of this experiment are not representative due to its very limited number of participants. In addition, students were selected whose opinion might differ from the one of experienced software engineers. Thus, this experiment can only be treated as a positive cue for the user acceptance of the tool. An experiment on larger scale, especially in a real project, would be necessary to receive more reliable results. This is a topic for possible future work.

7.1.2. Comparison of the Two Approaches

Two approaches for representing change packages were elaborated in this thesis (cf. Section 2.4). Both have their advantages and problems which are to be evaluated. The first approach was to use *patch files* as change package representation, the second one was to use lightweight branches (called *branches* in the remainder of this section).

A general problem is the challenge of establishing a reliable link between source code and work items. Both approaches, like probably every other approach, are not 100% reliable. This means that it can always be 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 source code is committed with the sources of another change package, which happens if a developer was working on two tasks and committed the changes for them in one change package instead of separating them. Although this problem exists theoretically, it is of minor practical relevance because the developer will notice the mistake when he sees that one of his tasks has no associated change package although he already implemented and committed it.

Both approaches assign the task of initially creating a link between the source code changes and the work items to the developer himself. Such an approach can never be secured against mistakes a developer makes while performing this task (like linking source code to the wrong work item). However, there is no reliable way to eliminate this error: Automatic techniques as described by others (cf. related work, Section 1.3) do not rely on the developer but are unable to produce reliable links, too. In fact, their error rate is much higher than anticipated error rates of reasonable developers. The general problem is that artifacts in human readable text or source code can never be linked together with full reliability. Consequently, our approach of letting the developer create the links might still be the better in comparison to automatic approaches, especially if recommendation techniques are added (cf. future work, Section 7.3).

Another fact which can affect the reliability of links is if changes are directly committed to the version control system without the creation of a change package. The approach using branches is more robust against this, because it allows to check if a commit exists which is not on a change package branch. The patch approach in contrast does not link the repository directly to the work items and thus is less applicable for ensuring reliable links. As a matter of fact, the support for checking the links was omitted in the prototype which uses patches. However, some possible approaches for implementing this functionality for

the patch approach were discussed in Section 2.4.

The most important advantage of the patch approach is that it is applicable for all version control systems which support patches (which almost all VCSs do). It is even applicable if no version control system but only tools for creating and applying patches like `diff` and `patch` are used.

The branch approach requires to have lightweight branches. It must support a high amount of concurrently running branches, since each work item is created on a separate branch. These branches should preferably be local. Otherwise, many use cases will take much time due to the network delay. Thus, the branch approach is limited to a set of VCSs supporting all these features. Especially the currently most widespread VCS Subversion is not suitable for this approach due to its lack of local lightweight branches. However, many of the modern distributed VCSs are suitable, so this restriction becomes less severe the more these modern VCSs are established and accepted.

In conclusion, both approaches seem suitable for implementing the desired functionality. The approach using branches seems more promising because it is tied closer to the version control system, thus allowing to maintain links more reliably.

7.2. Summary

This thesis proposed two approaches for maintaining code-to-task traceability links by using version control systems. These approaches were applied to the release management and code review process. The main idea for the code review application was to use the links to aid the review process by helping to apply the code to the reviewer's machine. For the release management, the links were used to check which work items are contained in a release and to build a release automatically by merging in the missing features.

The thesis began with a motivation for the problem and a related work survey in the different fields covered by the thesis. It continued with the elaboration of fundamental concepts the approaches build on — mainly version control systems and related concepts. Afterwards, two approaches for representing change packages, namely patches and lightweight branches, were proposed. After the theoretical concepts were elaborated, the design of a tool applying these concepts began. First, a solution-independent elicitation and analysis of the requirements for such tool was presented, followed by the design of a prototype for the proposed tool. Finally, the theoretical concepts were compared and the user acceptance of the prototype was measured in a user experiment. Concerning the concepts, it was concluded that the branch approach is better suitable as it promises more reliable links. Concerning the prototype, the user acceptance was high and the tool was rated positively.

7.3. Future Work

Many of the techniques and applications used in this thesis, like the establishment of code-to-task links, are still not explored in depth. Therefore, there is still much room for future research in this area. Five concrete topics for interesting future work were identified and are presented below.

Combination with Other Release Management Tasks

The field of release management is not explored as much as other fields of the software configuration management. Thus, a lot of work can still be done here. The prototype created in this thesis covers only one aspect of release management. An interesting approach would be to combine the techniques used in this thesis with other tasks of release management like build management and deployment, thus creating a one-click-to-release tool.

Improvement of Techniques

Another possible field of research is the improvement of the techniques used in this thesis. The patch approach needs a way to reliably perform the checking and building of releases. The branch approach cannot ensure reliable links either, as stated in the previous section. However, the unreliability is limited to incorrect usage. Thus, a promising approach could be the combination of the branch approach with a recommendation system using automatic techniques like latent semantic indexing (cf. related work, Section 1.3.2) to build links automatically. These links can then be used to run a sanity check against the link the user is trying to establish. If the automatically retrieved links do not match the link the user creates, he can be warned and other possibly correct targets for the link can be recommended.

Additional Techniques for Traceability Links

The two approaches used in this thesis are not the only possible ones. In addition to improving them, other techniques for maintaining source-code-to-task links (or links to other software engineering concerns) may be a field for future research. As most of the current approaches in the field of code-to-concern traceability focus on providing automatically calculated heuristics instead of more reliable links, there is still room for further research in this area.

Additional Applications for Code-to-Task Links

Once source-code-to-task links are established, there are many possible applications for them. The two topics elaborated in this thesis are examples that show the potential provided by such links. Further applications may yield very interesting and trailblazing results. Thus, the exploration of further applications is certainly a promising field of research.

Representative User Experiment

The user experiment undertaken in this thesis was very limited. A study where the release management and review support of this tool is used in a real project would be very beneficial to evaluate the true value offered by the application of code-to-task traceability to the field of reviews and release management.

Appendix A.

User Manual

This is a manual for using the implemented plug-in. Although the Git adapter is used here, the usage of the SVN adapter is similar. It is assumed that the usage of UNICASE and Git is well-known. The usage of the plug-in starts either in the UNICASE perspective, where model elements reside, or in a development perspective like the Java perspective, where changes from projects are tracked.

Use cases which begin in the UNICASE perspective are usually started by pressing a button in the model element editor of model elements provided by the plug-in. For example, the model element editor shows a `Check Release` and a `Build Release` button, when showing a release element. These can be pressed to start the respective workspace.

The use cases which begin in a development perspective are started by right-clicking a project which is under Git version control. The appearing context menu contains a submenu labeled `Unicase Change Tracking`, as shown in Figure A.1. This submenu contains buttons to start or resume the respective use cases.

A.1. Setting up a Stream

When a new project is set up, at least one stream has to be created before the rest of the plug-in can be used properly. This is done in the development perspective where the local workspace can be inspected (for example with the Package Explorer view). Before a stream can be set up, a project under Git version control must exist in the local workspace. A stream in UNICASE must always be associated to a specific Git branch. The currently checked-out Git branch will be associated to the stream to be created. Thus, make sure to check out the branch which should contain the stream before starting the stream creation. With the appropriate branch checked out, right-click the project and choose `Create Stream From Current Branch` from the `Unicase Change Tracking` menu. A window is shown where the location and name of the stream can be chosen, as displayed in Figure A.2. Here, you can browse all UNICASE projects in your UNICASE workspace to find an appropriate place for the stream. A stream model element is a usual UNICASE model element, so it can be placed in any leaf section.

Once a name and location are chosen, the plug-in searches for matching Git repository locations in the UNICASE project which was selected as a location for the stream. A Git repository location matches if it contains the same initial commit than the local Git repository to which the selected project belongs. If no such repository location exists, which will be the case if the UNICASE project is newly created, a dialog asking for the further proceeding is shown (Figure A.3).

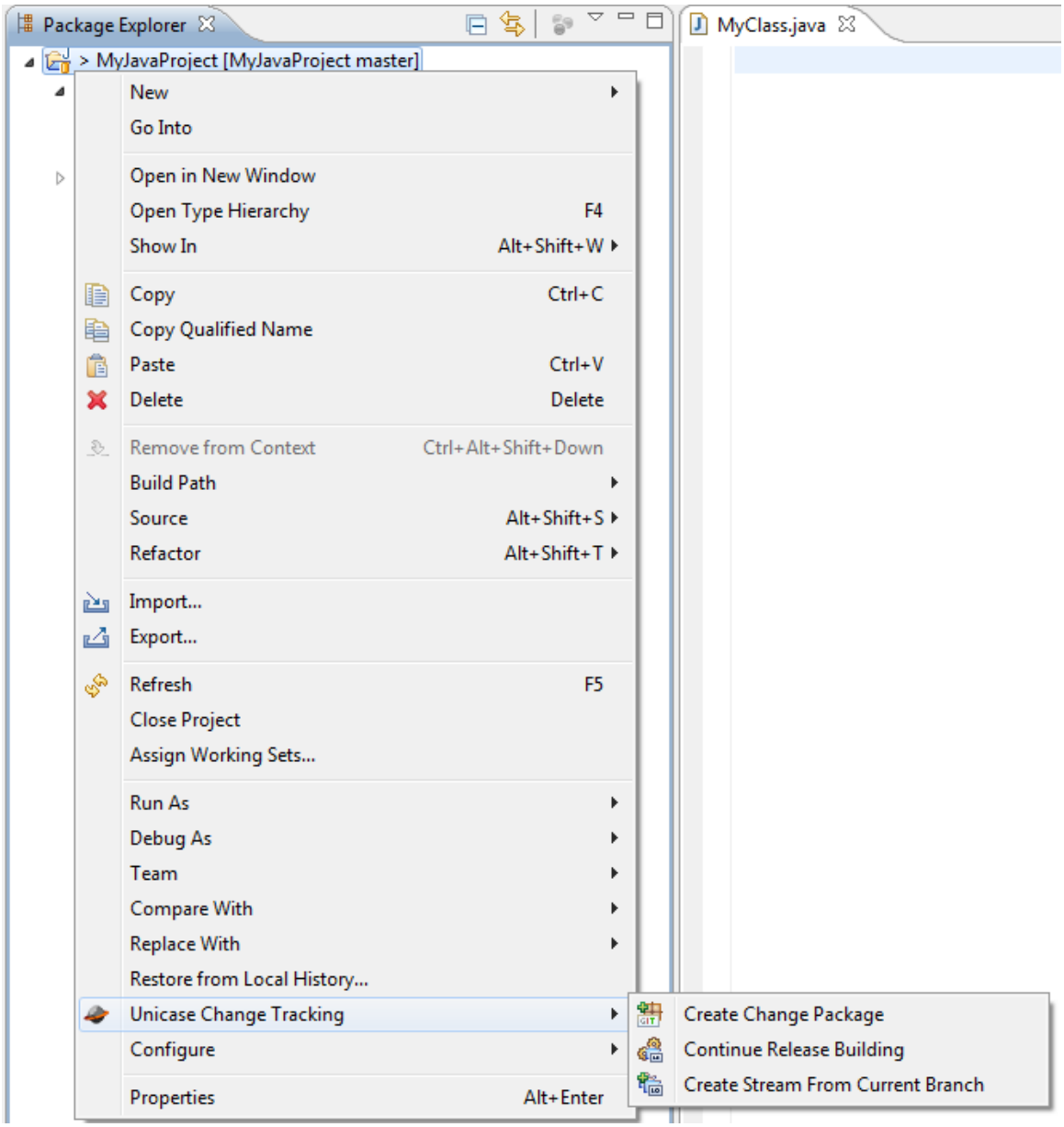


Figure A.1.: The “Unicase Change Tracking” context menu

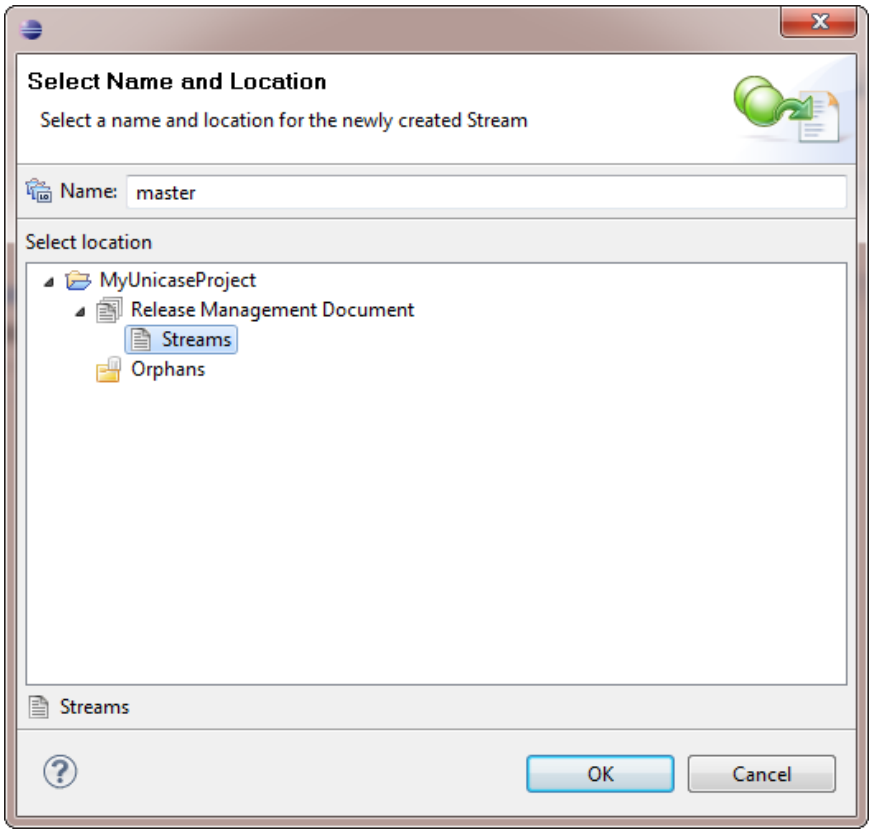


Figure A.2.: The "Select Location" dialog

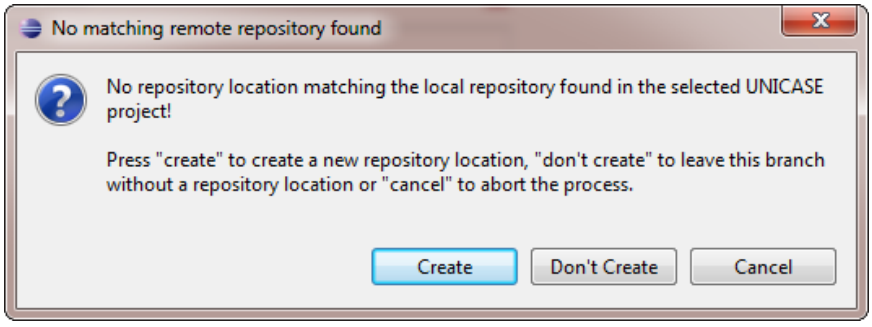


Figure A.3.: Further proceeding in case of missing repository location

The dialog provides three options: First, a new repository location can be created in the UNICASE project with the `Create` button. If this option is chosen, the system asks for the URL of the remote repository. This URL will be used for synchronizing the local repository with the remote one. Second, the stream can be left without a repository location. If this option is chosen, a location must be added manually later on. As long as no location is associated with the stream, most operations will fail. Finally, the whole stream creation can also be aborted with the `Cancel` button.

The system creates the stream model element and a Git branch model element reflecting the chosen Git branch. The branch model element is created in the same location as the stream. The model elements get properly associated to each other and to the location, so that the stream is ready to contain releases. The exemplified result of a "Create Stream" operation (with creating a repository location using a GitHub URL) is shown in Figure A.4.

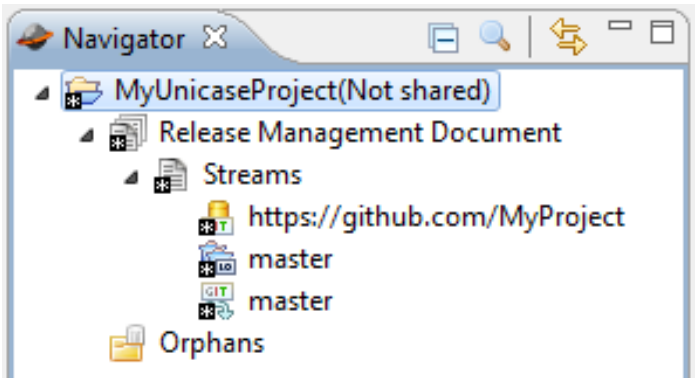


Figure A.4.: Result of a "Create Stream" operation

A.2. Creating a Release

The creation of a release is not directly supported by the plug-in. A release is a regular UNICASE model element. It can be created directly with UNICASE like other UNICASE model elements by right-clicking a leaf section, choosing `Create New Model Element...` and selecting `Change Tracking Release` in the displayed dialog. After the release is created, it must be associated with a stream. This can be done in the model element editor.

A.3. Collecting Changes and Creating a Change Package

The easiest way to create a change package is to check out the commit from which the changes are to be made (with the usual Git methods). Then, make the modifications to be contained in the package without committing them. Once you are done with the changes for one package (or earlier, if you want to save the changes in the remote repository), right-click the project in the package explorer and select `Create Change Package` from the `Unicase Change Tracking` menu.

Because creating a change package and attaching it to a work item is always executed together, the plug-in asks for a work item to attach the change package to. You are presented a dialog (Figure A.5) in which you can choose an work item to which the change package is to be attached. For faster filtering, you can choose the project in which the work item is placed and the user to which the work item is related. If a user is chosen, only work items that are either assigned to the user or are to be reviewed by the user are displayed. If `[No User]` is chosen, all work items in the project are displayed. You can also choose to create a new work item by selecting one of the «`Create new ...`» entries: A feature, a work item, or an issue can be created. If an entry for creating a work item is chosen, a dialog to select a name and a location for the newly created item is displayed, similar to the one for as depicted in Figure A.2.

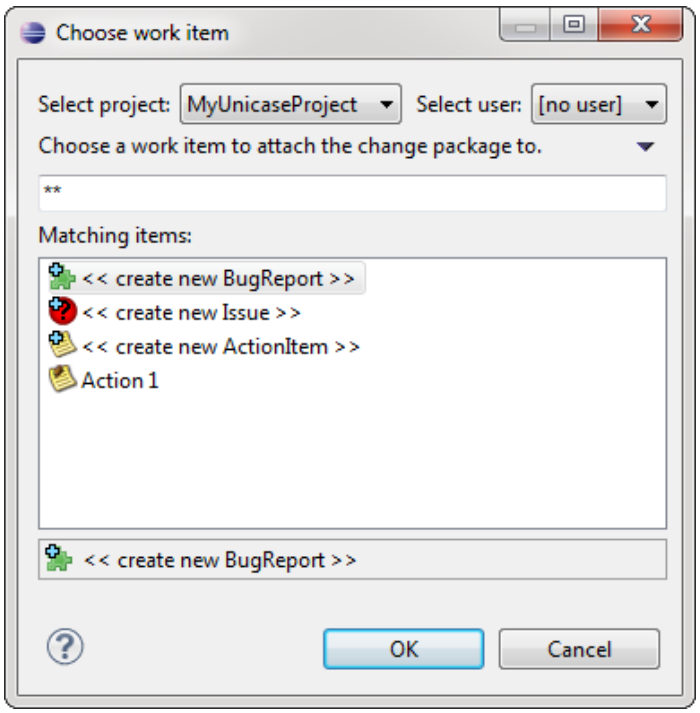


Figure A.5.: Choosing a work item

Once an item is chosen, a wizard for creating the change package pops up. On the first page (Figure A.6), the selected work item, UNICASE project and repository location are shown. The repository location is automatically looked up in the selected UNICASE project by searching for a repository location matching the local repository. If no repository is found, a new one can be created.

On the next page of the wizard (Figure A.7), you are asked to enter a name, short description, and long description for the change package. These will be used in the model element and the first commit on the Git branch that will automatically be created. The fields follow the conventions for Git commits: The short description should be a one-line description of the changes which will be displayed in the changelog and in the history view. The long description is a text which explains the contents more precisely. The name will be used as the name of the branch. Thus, it must be a valid Git identifier. You will be

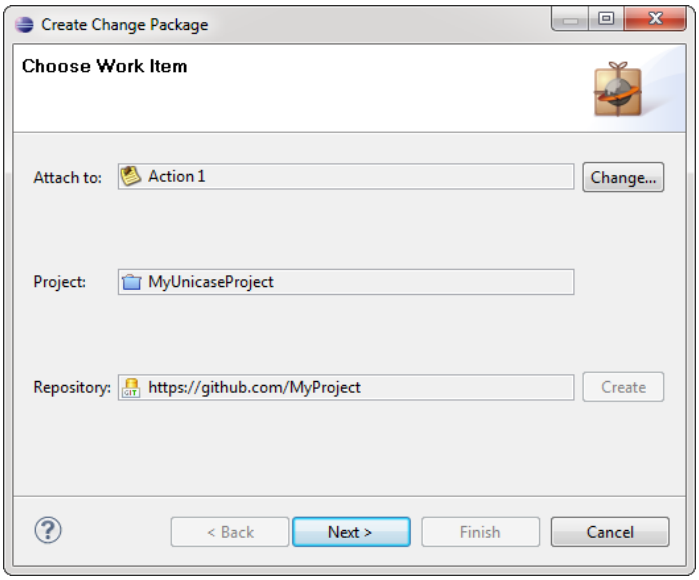


Figure A.6.: Change package creation wizard, page 1

shown an error message if the chosen name does not match this criterion. This is the last page of the dialog. Once you finish it, the change package will be created. The following steps are taken:

- A branch is created and all changes in the workspace and the staging area (i.e. all uncommitted changes) are committed to it.
- The new branch is pushed to the remote repository
- A model element for the change package and the Git branch are created and set up properly.
- The newly created change package is attached to the selected work item.

After a change package has been created, you can add additional changes to this package by committing to the associated branch. The latest commit on this branch will be used for applying the package later, so additional commits will also be applied. Thus, you can also create a change package before doing your changes and commit all your changes to it afterwards.

A.4. Applying a Change Package to the Local Workspace

To apply a change package to your local workspace, press the `Create Change Package` button. It can be found in the top toolbar of the model element editor when a change package is opened, or next to the link to a change package when a work item is opened. The change package will be applied to your local workspace. This works only if the workspace currently contains no changes and a local Git repository corresponding to the repository location of the change package is located in the local workspace.

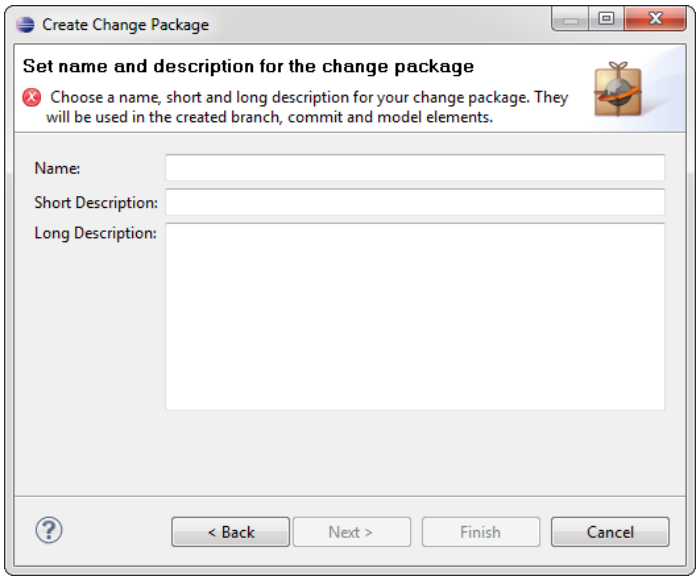


Figure A.7.: Change package creation wizard, page 2

If the Git repository is not found in the local workspace, a dialog is opened in which you can choose the further proceeding. One option is to select another location for the local repository. Choose this option if you have a matching local repository which is not located in the Eclipse workspace. You will be shown a file dialog in which you can choose the location of the local repository. Another option is to create a local repository by cloning from the repository location. The third option is to cancel the application process.

Once a local repository is found, the change package is applied to it. This is done by first pulling the branch associated to the change package from the remote repository and then checking out this branch.

A.5. Applying a Change Package to the Remote Repository

The plug-in does not provide functionality for applying a single change package to the remote repository, because this can be done easily directly with Git. Simply apply the change package to the local workspace, as described in the previous section, merge it with a desired branch, and push the changes to the remote repository.

A.6. Assigning Work Items to a Release

There is no support for assigning work items to a release in the plug-in, because this can be done with UNICASE. Work items can be assigned to a release in the model element editor. The release has a field “Included Work Items” to which the work items can be added.

A.7. Checking a Release

Checking a release reveals possible problems and visualizes the progress of that release. If no problems are revealed and all work items are resolved, the release is ready to be built. A release can be checked by opening it in the UNICASE model element editor where the buttons for checking and building a release can be found. Figure A.8 shows a release opened in the editor. Here, the button ( ) for checking a release, which is located in the top toolbar, is marked with the label **(C)**.

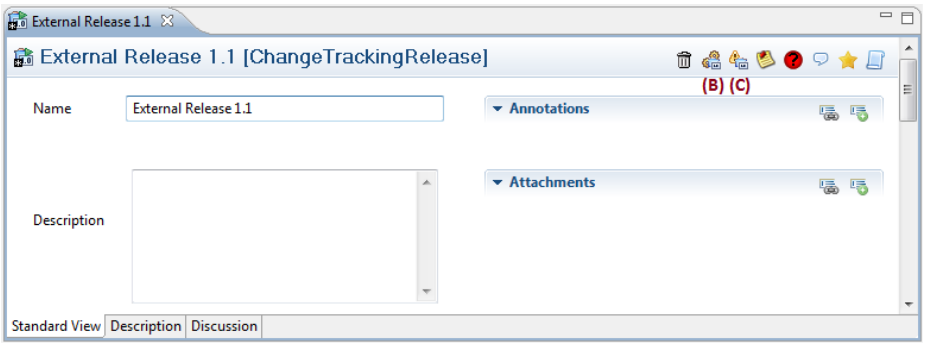


Figure A.8.: Editor showing a release.

Once the `Check Release` button is pressed the system asks for updating the local repository. If you agree, the system will fetch the latest content from the remote repository to ensure that you have an up-to-date copy in your local workspace. If you skip this update, the check will be done without updating first, but you will receive a warning reading that the results might be outdated. If you do not have the project in your local workspace, you will receive an error message. Once the update is finished or skipped, the system performs a full check of the release and then shows a window containing the results. This window is shown in Figure A.9. It contains four tabs which display different aspects of the release. Above these tabs is the status message. It shows the overall status of the release, i.e. if it is ready to be built or contains warnings or errors which would prevent the building process.

The first tab, shown in Figure A.9, contains an overview of the progress of the release. It contains the amount of work items assigned to this release and indicates how many of them are already resolved. The building progress is shown, as well, i.e. it is displayed how many change packages are already merged into branch of the release ( ) and how many still have to be merged in by a build operation ( ). The third category to which a change package can belong to is *erroneous* ( ). A change package is erroneous if for some reason the system cannot determine its state, or it cannot be found or processed properly in the local repository. In either case, you will find details about the problem in the *Problems* tab. Finally, the tab contains a short summary of the status of the release showing if the release is ready to be built.

The second tab, shown in Figure A.10, contains a tree view of the work items and change packages belonging to this release. The work packages are shown as child nodes of the work items to which they belong. The change package icons also show which packages are merged, which are not merged, and which are erroneous. By pressing the button,

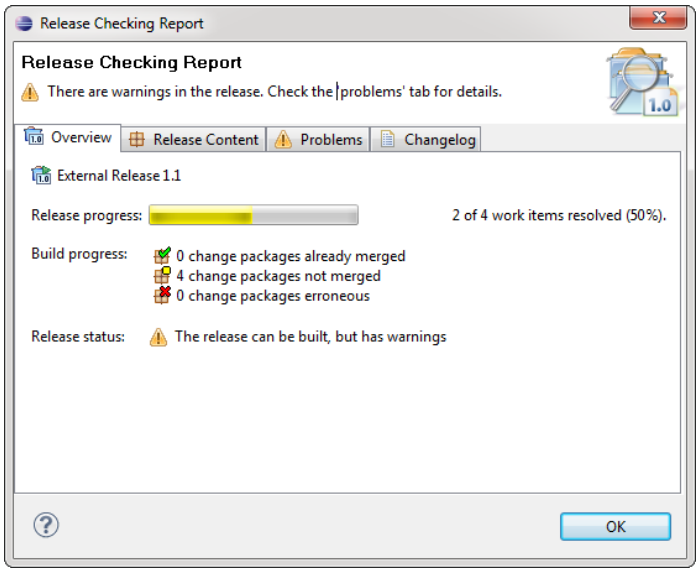


Figure A.9.: Release check window, “Overview” tab

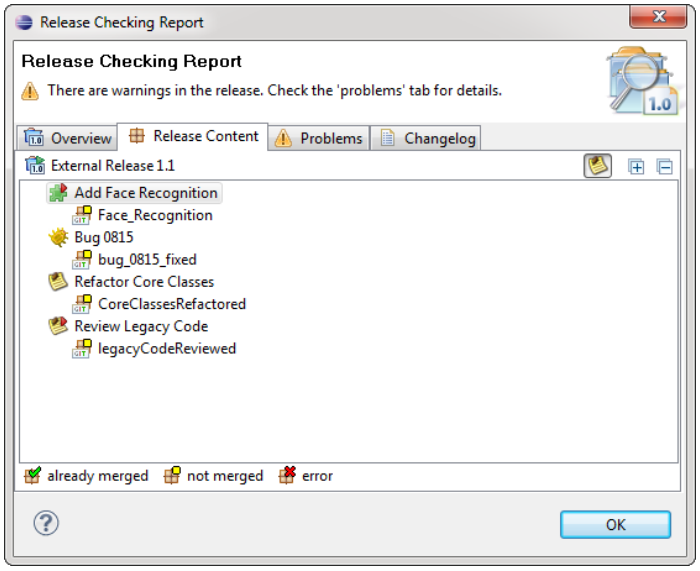


Figure A.10.: Release check window, “Release Content” tab

you can toggle between hiding and showing the work items. If the work items are not shown, only the change packages are displayed as a list.

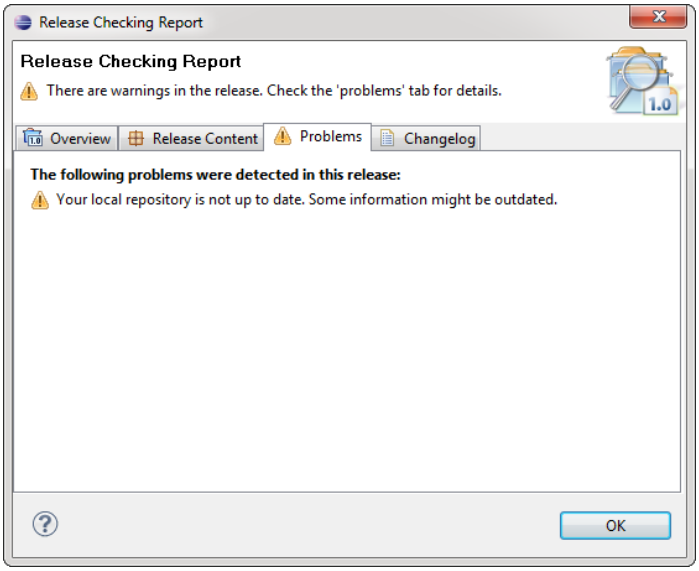


Figure A.11.: Release check window, "Problems" tab

The third tab, shown in Figure A.11, contains all warnings and errors that were revealed while the checks were performed. An error ( ) is a problem that prevents the release from being built. A warning ( ) is a problem that does not prevent the building process. However, the process could yield erroneous results. Therefore, it is discouraged to build a release, if it contains warnings. The tab lists all errors and warnings found and often also shows a hint how to solve them.

The fourth and final tab is shown in Figure A.12. It contains the changelog. The changelog is an automatically assembled text showing the short description of all change packages which were included. It arranges these descriptions in a list-like manner. You can copy and paste this text into the changelog file of your project or use it for other purposes.

A.8. Building a Release

Like the button for checking a release, the button for building a release ( ) is located in the top toolbar in the model element editor when a release is selected. In Figure A.8 showing a release in the editor, it is marked with **(B)**. Once it is pressed, you will be presented with the decision to update your local repository by fetching the latest updates from the remote repository. If you do not update, you will receive a warning since you might be working with outdated data. After the update is finished or skipped, the system performs a check of the release. When the check is finished, you will be presented the build release wizard. The first page of this wizard is the same as the check release dialog which opens up when you check the release. So you can recheck the project before building it. If errors were detected, the release cannot be built and you have to cancel the building process. Otherwise, you can switch to the next page where you can enter a name for the tag which

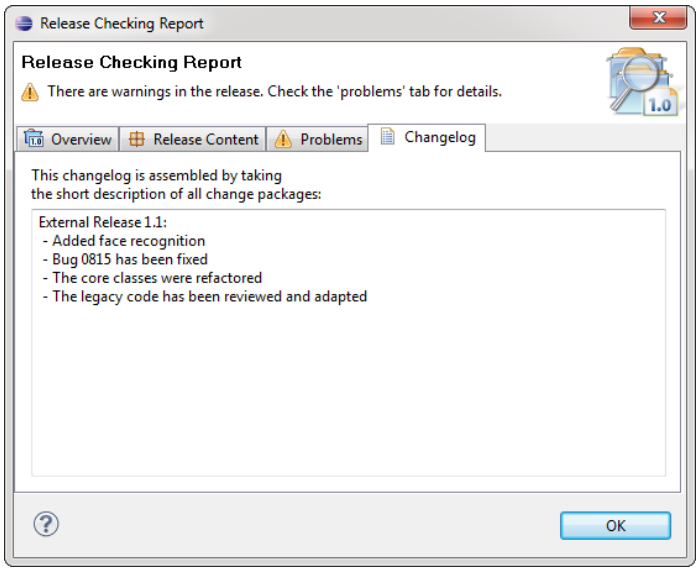


Figure A.12.: Release check window, "Changelog" tab

will be used to tag this release in the Git repository. Thus, it must be a valid Git identifier. Afterwards, you can finish the dialog which starts the building process.

The building process consists of the following steps:

- All change packages that are not yet merged into the branch of the release are merged into it one by one.
- A tag for the release is created in the Git repository.
- All changed data in the Git repository is pushed to the remote repository.
- A tag model element is created in UNICASE and attached to the release, and the release build data is set appropriately.

If a conflict occurs during the merge process, a dialog informing about the conflict is shown and the Java perspective is opened. You are now supposed to resolve the conflict. You can use all tool support of Git for resolving conflicts. Once you are done, add all conflicting files[1]. Once all conflicts are resolved, you can continue the release building. This is done by right-clicking the project that is built (or one of the projects, if the release contains more than one Eclipse project) and choosing `Continue Release Building` from the `Unicase Change Tracking` context menu. The system will continue to merge the remaining change packages. You will be informed once the building process is finished successfully. Now, you have the final code for the release in your workspace.

[1] Adding files is done by selecting them and choosing `add` in the Team menu. This is Git's standard way for marking a conflicting file as resolved.

Bibliography

[1] W.F. Tichy. Tools for software configuration management. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 1–20, 1988.

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

[3] IEEE. IEEE guide to software configuration management: ANSI/IEEE std 1042-1987. 1987.

[4] A. Leon. *Software configuration management handbook*. Artech House, Inc. Norwood, MA, USA, 2004.

[5] bugzilla.org contributors. Bugzilla. http://www.bugzilla.org.

[6] Free Software Foundation, Inc. GNU Make. http://www.gnu.org/software/make.

[7] Apache Software Foundation. Apache Ant. http://ant.apache.org.

[8] Free Software Foundation, Inc. Concurrent Versions System. http://savannah.nongnu.org/projects/cvs.

[9] Apache Software Foundation. Apache Subversion. http://subversion.apache.org.

[10] Git contributors. Git - fast version control system. http://git-scm.com.

[11] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. *case*, page 0340, 1995.

[12] A. Van der Hoek, R. Hall, D. Heimbigner, and A. Wolf. Software release management. *Software Engineering—ESEC/FSE'97*, pages 159–175, 1997.

[13] M.P. Robillard and G.C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):3–es, 2007.

[14] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135. IEEE Computer Society, 2003.

[15] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, pages 970–983, 2002.

[16] UNICASE contributors. UNICASE. http://unicase.org.

[17] The Eclipse Foundation. Eclipse. http://www.eclipse.org.

[18] H. Kagdi, M.L. Collard, and J.I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[19] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th international conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.

[20] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, page 190. Published by the IEEE Computer Society, 1998.

[21] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. 2003.

[22] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. 2003.

[23] M.A. Storey, L.T. Cheng, I. Bull, and P. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 195–198. ACM, 2006.

[24] M.A. Storey, L.T. Cheng, J. Singer, M. Muller, D. Myers, and J. Ryall. How programmers can turn comments into waypoints for code navigation. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 265–274. IEEE, 2007.

[25] C. Treude and M.A. Storey. How tagging helps bridge the gap between social and technical aspects in software development. 2009.

[26] Geeknet, Inc. TagSEA. http://tagsea.sourceforge.net.

[27] J. Anvik and M.-A. Storey. Task articulation in software maintenance: Integrating source code annotations with an issue tracking system. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 460 –461, 28 2008-oct. 4 2008.

[28] The Eclipse Foundation. Eclipse MyLyn open source project. http://www.eclipse.org/mylyn.

[29] A. Marcus, J.I. Maletic, and A. Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):811–836, 2005.

[30] G. Antoniol, A. Potrich, P. Tonella, and R. Fiutem. Evolving object oriented design to improve code traceability. In *IWPC*, page 151. Published by the IEEE Computer Society, 1999.

[31] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability for object-oriented systems. *Annals of Software Engineering*, 9(1):35–58, 2000.

[32] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability recovery: Selecting the basic linkage properties. *Science of Computer Programming*, 40(2-3):213–234, 2001.

[33] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-oriented software evolution. *Software: Practice and Experience*, 31(4):331–355, 2001.

[34] G. Antoniol, E. Merlo, Y.G. Guéhéneuc, and H. Sahraoui. On feature traceability in object oriented programs. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, pages 73–78. ACM, 2005.

[35] G. Antoniol and Y.G. Guéhéneuc. Feature identification: A novel approach and a case study. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 357–366. IEEE, 2005.

[36] M. Eaddy, A.V. Aho, G. Antoniol, et al. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *The 16th IEEE International Conference on Program Comprehension*, pages 53–62. IEEE, 2008.

[37] W. Zhao, L. Zhang, Y. Liu, J. Luo, and J. Sun. Understanding how the requirements are implemented in source code. 2003.

[38] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2):195–226, 2006.

[39] A. De Lucia, R. Oliveto, F. Zurolo, and M. Di Penta. Improving comprehensibility of source code via traceability information: A controlled experiment. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 317–326. IEEE Computer Society, 2006.

[40] N. Wilde and M.C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[41] A.D. Eisenberg and K. De Volder. Dynamic feature traces: Finding features in unfamiliar code. 2005.

[42] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 234–243. ACM, 2007.

[43] M. Grechanik, K.S. McKinley, and D.E. Perry. Recovering and using use-case-diagram-to-source-code traceability links. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 95–104. ACM, 2007.

[44] I. Omoronyia, G. Sindre, M. Roper, J. Ferguson, and M. Wood. Use case to source code traceability: The developer navigation view point. In *2009 17th IEEE International Requirements Engineering Conference*, pages 237–242. IEEE, 2009.

[45] M.P. Robillard and G.C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th international conference on Software engineering*, pages 406–416. ACM, 2002.

[46] M.P. Robillard. Tracking concerns in evolving source code: An empirical study. 2006.

[47] A. Van der Hoek and A.L. Wolf. Software release management for component-based software. *Software: Practice and Experience*, 33(1):77–98, 2003.

[48] M. Michlmayr. Quality improvement in volunteer free software projects: Exploring the impact of release management. In *Proceedings of the First International Conference on Open Source Systems*, pages 309–10. Citeseer, 2005.

[49] M. Michlmayr, F. Hunt, and D. Probert. Release management in free software projects: Practices and problems. *Open Source Development, Adoption and Innovation*, pages 295–300, 2007.

[50] J.R. Erenkrantz. Release management within open source projects. In *Proc. 3rd. Workshop on Open Source Software Engineering*. Citeseer, 2003.

[51] D. Greer and G. Ruhe. Software release planning: An evolutionary and iterative approach. *Information and Software Technology*, 46(4):243–253, 2004.

[52] O. Saliu and G. Ruhe. Software release planning for evolving systems. *Innovations in Systems and Software Engineering*, 1(2):189–204, 2005.

[53] G. Ruhe and M.O. Saliu. The art and science of software release planning. *IEEE software*, pages 47–53, 2005.

[54] O. Saliu and G. Ruhe. Supporting software release planning decisions for evolving systems. 2005.

[55] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[56] O. Laitenberger and J.M. DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.

[57] O. Laitenberger. A survey of software inspection technologies. 2002. `ftp://cs.pitt.edu/chang/handbook/61b.pdf`.

[58] A. Aurum, H. Petersson, and C. Wohlin. State-of-the-art: Software inspections after 25 years. *Software Testing Verification and Reliability*, 12, 2002.

[59] O. Laitenberger. Studying the effects of code inspection and structural testing on software quality. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 237–246. IEEE, 1998.

[60] A.A. Porter, H.P. Siy, C.A. Toman, and L.G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *Software Engineering, IEEE Transactions on*, 23(6):329–346, 1997.

[61] A.A. Porter and P.M. Johnson. Assessing software review meetings: Results of a comparative analysis of two experimental studies. *Software Engineering, IEEE Transactions on*, 23(3):129–145, 1997.

[62] H. Siy and L. Votta. Does the modern code inspection have value? In *ICSM*, page 281. Published by the IEEE Computer Society, 2001.

[63] C.F. Kemerer and M.C. Paulk. The impact of design and code reviews on software quality: An empirical study based on PSP data. *IEEE transactions on software engineering*, pages 534–550, 2009.

[64] M.V. Mäntylä and C. Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, pages 430–448, 2008.

[65] A. Dunsmore, M. Roper, and M. Wood. Systematic object-oriented inspection: An empirical study. In *ICSE*, page 0135. Published by the IEEE Computer Society, 2001.

[66] A. Dunsmore, M. Roper, and M. Wood. Further investigations into the development and evaluation of reading techniques for object-oriented code inspection. In *Proceedings of the 24th international conference on Software engineering*, pages 47–57. ACM, 2002.

[67] A. Dunsmore, M. Roper, and M. Wood. The development and evaluation of three diverse techniques for object-oriented code inspection. *IEEE transactions on software engineering*, pages 677–686, 2003.

[68] P.C. Rigby and D.M. German. A preliminary examination of code review processes in open source projects. Technical report, DCS-305-IR, University of Victoria, 2006.

[69] V. Sembugamoorthy and L. Brothers. ICICLE: Intelligent code inspection in a C language environment. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 146–154. IEEE, 1990.

[70] L.R. Brothers, V. Sembugamoorthy, and A.E. Irgon. Knowledge-based code inspection with ICICLE. *Innovative Applications of Artificial Intelligence 4: Proceedings of IAAI*, 92, 1992.

[71] L.R. Brothers. Multimedia groupware for code inspection. In *Communications, 1992. ICC'92, Conference record, SUPERCOMM/ICC'92, Discovering a New World of Communications., IEEE International Conference on*, pages 1076–1081. IEEE.

[72] L. Harjumaa and I. Tervonen. A WWW-based tool for software inspection. In *HICSS*, page 379. Published by the IEEE Computer Society, 1998.

[73] F. Belli and R. Crisan. Towards automation of checklist-based code-reviews. In *ISSRE*, page 24. Published by the IEEE Computer Society, 1996.

[74] F. Macdonald and J. Miller. Modelling software inspection methods for the application of tool support. 1995.

[75] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. Automating the software inspection process. *Automated Software Engineering*, 3(3):193–218, 1996.

[76] F. Macdonald and J. Miller. A comparison of computer support systems for software inspection. *Automated Software Engineering*, 6(3):291–313, 1999.

[77] The Eclipse Foundation. Eclipse Modeling Framework project (EMF). http://www.eclipse.org/modeling/emf.

[78] O.E. Dictionary. Oxford English dictionary online. *Mount Royal College Lib., Calgary*, 14, 2004.

[79] D.A. Patterson, G. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116. ACM, 1988.

[80] Brian de Alwis and Jonathan Sillito. Why are software projects moving from centralized to decentralized version control systems? *Cooperative and Human Aspects on Software Engineering, ICSE Workshop on*, 0:36–39, 2009.

[81] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *International Conference on Software Engineering 2008*, volume 2008, pages 0–11. Association for Computing Machinery, One Astor Plaza 1515 Broadway, 17 th Floor New York NY 10036-5701 USA, 2008.

[82] BerliOS. SCCS - the POSIX standard Source Code Control System. http://sccs.berlios.de.

[83] Mark J. Rochkind. The source code control system. In *IEEE Transactions on Software Engineering*, volume SE-1. December 1975.

[84] Sun Microsystems. Sun WorkShop TeamWare. [no longer available].

[85] BitMover, Inc. Bitkeeper - the scalable distributed software configuration management system. http://www.bitkeeper.com.

[86] RCS Maintainers. Official RCS homepage. http://www.cs.purdue.edu/homes/trinkle/RCS.

[87] W.F. Tichy. RCS — a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[88] Dick Grune. Concurrent Versions System, a method for independent cooperation. Technical report, IR 113, Vrije Universiteit, 1986.

[89] Berliner, Prisma Inc, and Mark Dabling Blvd. CVS II: Parallelizing software development, 1990.

[90] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, Inc., Sebastopol, CA, USA, June 2004.

[91] Brian Berliner. CVS. http://brianberliner.com/cvs/, 2011.

[92] Mike Mason. *Pragmatic Version Control Using Subversion*. The Pragmatic Programmers LLC., Dallas, Texas, USA, 2006.

[93] The Apache Software Foundation. Skip-deltas in Subversion. http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas.

[94] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, Prem Devanbu, A. E. Housman, and A Shropshire Lad. The promises and perils of mining Git, 2009.

[95] Linus Torvalds. Transcript of tech talk: Linus Torvalds on Git. https://git.wiki.kernel.org/index.php/LinusTalk200705Transcript.

[96] Jon Loeliger. *Version Control with Git*. O'Reilly Media, Inc., Sebastopol, CA, USA, May 2009.

[97] Branching and merging with Git. http://lwn.net/Articles/210045/.

[98] Ian Clatworthy. Distributed version control systems — why and how. http://ianclatworthy.files.wordpress.com/2007/10/dvcs-why-and-how3.pdf.

[99] Free Software Foundation, Inc. GNU arch. http://www.gnu.org/software/gnu-arch.

[100] Software Freedom Conservancy. Darcs. http://darcs.net.

[101] Best Practical Solutions LLC. The SVK version control system. http://svk.bestpractical.com.

[102] Monotone.ca contributors. Monotone. http://www.monotone.ca.

[103] Ross Cohen. Codeville. http://codeville.org. [currently unreachable].

[104] The Mercurial community. Mercurial SCM. http://mercurial.selenic.com.

[105] Canonical Ltd. Bazaar. http://bazaar.canonical.com.

[106] Fossil contributors. Fossil - simple, high-reliability, distributed software configuration management. http://fossil-scm.org.

[107] Microsoft Corporation. Visual Studio Team Foundation Server 2010. http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server.

[108] GitHub Inc. Secure source code hosting and collaborative development - GitHub. https://github.com.

[109] Canonical Ltd. Launchpad. https://launchpad.net.

[110] Google. Google code. http://code.google.com.

[111] Assembla, LLC. Assembla project workspaces to accelerate software teams, with issue tracking, GIT, SVN, and collaboration. http://www.assembla.com.

[112] R.J. Abbott. Program design by informal English descriptions. *Communications of the ACM*, 26(11):882–894, 1983.

[113] CollabNet, Inc. Subclipse. http://subclipse.tigris.org.

[114] The Eclipse Foundation. JGit. http://eclipse.org/jgit.

[115] The Eclipse Foundation. EGit. http://eclipse.org/egit.

[116] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. Object-oriented modeling and design. 1991.

List of Figures

2.1. Revision graphs for linear development (top) and with branches (bottom). . 13
2.2. Centralized (left) vs. distributed (right) version control. 17
2.3. Different topologies when using a DVCS. 17
2.4. Two versions of a source file 21
2.5. Two versions of a source file emerging from a common base version 21
2.6. Conflicting versions 22
2.7. Result of merging version 1 and version 2 of Figure 2.6 23
2.8. Data transferring Git commands 27
2.9. Criss-cross merge 28
2.10. Part of the revision graph of the JGit project 29
2.11. Use of different VCSs in Debian packages [80]. 31

3.1. Use case overview 43

4.1. The entity classes of the CASE package 58
4.2. The entity classes of the revision handling package 59
4.3. The entity classes of the release checking package 60
4.4. Change package related classes 61
4.5. Release & stream setup related classes 62
4.6. Release building related classes 63
4.7. Dynamic behaviour of the *Check Release* use case 65
4.8. Dynamic behaviour of the *Build Release* use case 66

5.1. The subsystem decomposition 72
5.2. The plug-ins of the prototype 74
5.3. The hardware/software mapping 76

6.1. EMF data model 80
6.2. Extension point usage 83
6.3. Hierarchic displaying of model elements 84
6.4. The JFace viewers 86
6.5. Classes for using EMF label providers 89
6.6. Dynamic behaviour of EMF label providers 91
6.7. Decorating images and text 92
6.8. Package status icons generated by a decorator 92
6.9. Command execution using EMF 93
6.10. Adaption to different VCSs 95

A.1. The “Unicase Change Tracking” context menu 104

A.2. The “Select Location” dialog 105
A.3. Further proceeding in case of missing repository location 105
A.4. Result of a “Create Stream” operation 106
A.5. Choosing a work item 107
A.6. Change package creation wizard, page 1 108
A.7. Change package creation wizard, page 2 109
A.8. Editor showing a release. 110
A.9. Release check window, “Overview” tab 111
A.10. Release check window, “Release Content” tab 111
A.11. Release check window, “Problems” tab 112
A.12. Release check window, “Changelog” tab 113