Chair of
Software Engineering

Master's Thesis

Automatic Version Control System for Distributed Software Development

by

Sandra Weber

March 2012–September 2012

Supervised by

Prof. Bertrand Meyer, Dr. Martin Nordio, Hans-Christian Estler

Software Engineering Group, Department of Computer Science, ETH Zurich

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Software engineering nowadays is distributed. The companies outsource and the individual teams stretch over multiple countries. To be able to work together it's essential to be aware of the progress of the other project members. Most software projects use version control systems although it is time consuming and regularly introduces conflicts. Staying aware of the progress and resolving conflicts is even more challenging in distributed software development.

This thesis introduces an automatic version control system for CloudStudio, a web-based IDE optimized for real-time collaboration. The goal is to provide seamless and continuous change awareness as well as version control without requiring any interaction, enabling the developer to fully focus on the task at hand.

Committing, sharing and conflict resolution can be fully automated and done without user interaction. Still, the user benefits from all advantages of a version-controlled repository. The project repositories are exported to enable the use of another IDE with a CloudStudio project.

The inconspicuous, real-time change awareness serves as a conflict detection. Using a line-wise content model with a version tree for each line, conflicts can be prevented and automatically resolved. The line-based model is kept synchronized with the version-controlled file. Updates merged into the repositories without using the web-based IDE are handled, too.

Contents

Chapter 1

Introduction

1.1 Distributed Software Development

A successful software engineering company nowadays is often spread over multiple locations or has an offshore software production. The teams have to work across borders as well as the differences of cultures and face the challenges of distributed software development.

In this situation communication and collaboration are of utmost importance [23, 11]. The design of a well defined API, for example using contracts as discussed in the paper by Nordio et. al. [26], becomes essential.

The effect of distribution on software development have been researched from different angles [15, 14, 23]. Espinosa et al. [14] looked at the impact of time zones on the performance during software development. During the DOSE [25, 24] university course, Nordio et al. [23] studied the effect of time and cultural differences on the communication within the teams. Possible tactical approaches to face global software development are discussed by Carmel et al. [12].

1.2 CloudStudio

The most important tool for a developer is the Integrated Development Environment (IDE). There are already several IDEs like EiffelStudio or Eclipse. While they are optimized for software development, they lack the support of communication and collaboration needed in distributed software development. To fully collaborate the team members need to be able to work in a common environment. The IDE no longer has to be a personal tool requiring each developer to have his own installation. A common programming

environment is automatically adjusted and aligned with the one used by the teammates.

CloudStudio [22] is a cloud-based IDE enabling collaboration in a team using a common environment. All tools needed by a team member in a distributed software development team are provided.

CloudStudio not only offers the standard functionality of an IDE, but also enhances collaboration during activities such as pair programming or a code review. The new automated configuration management system and the real-time change awareness enable a better compatibility between different software components developed by different team members.

CloudStudio highlights the own modifications in the editor, as well as makes the user aware of the changes of other team members, enabling interaction at real time. This feature improves collaboration and anticipates conflicts. The web-based IDE's verification components offer static as well as dynamic functionality. It is possible to automatically proof the Eiffel code collaboratively programmed using AutoProof [29, 21]. During runtime the CloudStudio project can be verified using AutoTest [20, 30]. The communication over CloudStudio is simplified by integrated tools like a chat box.

A further feature that will be added to the web-based IDE in the future is AutoFix as researched by Pei et. al [27]. It's also planned to integrate the testing and proofing similar to the method described in the paper by Tschannen et. al [28].

1.3 Version Control System

Version control systems (VCS) are used in almost any software project with multiple team members. Teamwork requires sharing files. In software engineering VCS are the approved solution for managing text files and releases.

During the last few years distributed VCS like Git or Mercurial became increasingly more popular. With the ability to work independently of a server and a centralized repository, the projects gain flexibility when branching and merging.

Whether you choose a centralized or a distributed system, version control is a time-consuming, non-trivial activity. The cycle of sharing content takes multiple operations, commit, pull, push, merge and resolve. This thesis proposes a solution to reduce the time overhead introduced by the standard version control systems. The version control activities are simplified and automatized while conflicts are avoided and resolved using change awareness.

CloudStudio will rely on a distributed version control system to manage multiple repositories and will provide access to the project repositories us-

ing a HTTP connection. Within the web-based IDE the user will have all the benefits of a version control system as possible while all version control operations can be performed automatically. The CloudStudio IDE also supports tasks, represented by branches, to enable separated development with minimal sharing in between two different tasks. However using the change awareness the developers will be implicitly in the loop about the changes on the other tasks.

1.4 Change Awareness & Conflict Detection

Conflicts in the source code are part of the version control routine in any software project. In distributed software development conflicts are even harder to resolve. Awareness of the changes of the team members is crucial if the developers do not meet face-to-face every day during the coffee break.

There has been extensive research in the area of change awareness and collaborative conflicts in version-controlled software engineering projects. Multiple papers present approaches how to resolve conflicts and be fully aware of the work of other team members. The occurrences of conflicts and the performing of speculative version control operations was discussed in the paper by Brun et. all [9]. As described in their research, conflicts can be separated into two categories, textual and higher-order. A higher-order conflict indicates merging will lead to an error during building or testing, where as the textual category refers to conflicts when merging the source code. In their further work [10] a tool named Crystal is introduced, which provides awareness to conflicts on the project level. The tool Syde [16] provides a direct integration with Eclipse and uses the abstract syntax tree (AST) to detect conflicts and apply change awareness on the syntax level. It was used to investigate the conflict detection in a user study [17]. AST-level conflict detection compared to the textual analysis requires a program to compile, at least partially, before the analysis can start. The line-wise approach chosen in this thesis can detect conflicts while the user is typing. The change awareness remains independent of the language and the compiler. CollabVS [18] is an extension of Visual Studio providing file-level awareness and a tool for conflict resolution. Jazz [19] is implemented as Eclipse plugin and shows simple change awareness by highlighting changed lines without special means of resolving conflicts. FASTDash [8] is a team dashboard providing file-level awareness of the activities in Visual Studio project. Since it's suggested the team is assembled in a single room with the dashboard projected onto one of the walls, it is not a suitable tool for distributed software development.

The focus of this thesis is on textual clashes. It will provide an approach

on preventing and resolving textual conflicts using change awareness with an underlying distributed VCS and additional meta data about the history of changes. However using CloudStudio it will also be possible to detect a higher-order conflict.

1.5 Motivation

The extended change awareness of the CloudStudio IDE will be able to detect conflicts of both categories, textual as well as compiling or testing conflicts. The history of versions of a line are saved in form of meta data and are used to point the developer to potential conflicts while programming. This additional information about how the changes are related plays a vital role when resolving the conflicts automatically. A user of CloudStudio can see the changes of other team members in real-time. Even if he chooses to hide the changes of some users, the editor will indicate missing lines. Higher-level conflicts can be avoided by manually compiling, proofing and testing with changes of other users, without the need to share the modifications as well as the resulting conflicts first with the project repository.

With the new automated version control of the web-based IDE the sharing no longer is an effort. Using change awareness the developer will be able to see the modification of other users and be alerted to possible conflicts immediately. Thanks to the automation the collaboration will be improved by regularly sharing the code of team members working on the same task without any interaction on their part needed.

The underlying distributed version control system (DVCS) enables the developer to work on CloudStudio projects using an IDE of his choice. Yet in a situation like a code review or a pair programming session where tight collaboration is needed, the team can still benefit from the collaborative tools provided by the web-based IDE.

1.6 Goals

The goal of this thesis is to enhance CloudStudio with a new version control system that allows connecting and working on projects using a native IDE. The version control in the Cloud IDE will be mostly automated. All the same the user benefits from the advantages of a VCS for example by rolling back, going to a previous revision or consulting the revision log.

Using change awareness the user will be sensible to the changes of other developers. He will see the modifications in real-time, if he wants to. But

when compiling only the committed alterations of others are considered such that no additional compilation errors will be introduced. This is further endorsed by automatically committing the modifications if the user successfully compiled.

To enable fine-grained version control and awareness, a line-wise approach is chosen. Most programming languages rely on separation of lines to improve readability and sometimes even for syntactical purposes. By choosing a line-based model, the granularity and the benefits are maximized.

The server maintains meta data of the past versions of each line. This representation is synchronized and kept up-to-date with the version-controlled content. The conflicts within the web-based IDE will be resolved automatically using the additional information provided by the intermediate model.

The version control system provides access to the CloudStudio projects without using the web-based IDE. The user is authenticated with his account and is able to clone the project repository on the CloudStudio server to use it with a development environment of his choice. If a repository on the server is updated from the outside, the changes will immediately be displayed to the team members working with the web-based IDE. The adaption of the meta data may not be as accurate as when using CloudStudio due to the lack of information. However the changes will be applied iteratively for each intermediate commit to ensure deriving a change history as fine-grained as possible based on the limited information available.

To test and to analyze the implemented automatic version control system and the change awareness a case study is planned and specific use cases are conducted. The results are discussed in the section 4.

Chapter 2

Approach

To extend CloudStudio with an automated version control system, the existing backend had to be replaced.

In the previous deprecated design of CloudStudio the content of a file as well as the folder structure have been saved in an SQL database. The file content and all the changes of the users were saved in a simple, text-based format in one of the database tables. This approach will be completely abandoned and replaced by multiple repositories managing the files. The replacement was also a chance to refactor the code of CloudStudio and improve the modularity.

Using version-controlled repositories will allow the users to program in another IDE than CloudStudio. It has further advantages such as the possibility to use version control operations like rollback and having a back-up and the history of each of the files. The downside is that CloudStudio has to deal with multiple versions which requires merging to implement the change awareness. How this problem is approached is described in detail in the section 2.3.

2.1 Architecture

The CloudStudio server provides several services, which can be invoked asynchronously by the the CloudStudio clients. Each service has multiple methods, the client can call. There is for example a project service with methods to create a new project or to get all existing projects. Each service instance on the server is user-specific and will interact directly with the backend of the server or query the database to perform its purpose.

The backend is the connection to the file system and the distributed version control system. To improve abstraction and modularity the backend is

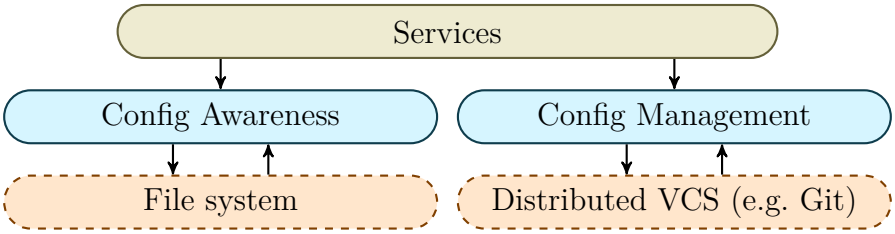


Figure 2.1: Architectural Overview

split into two parts as shown in the figure 2.1. The configuration management connects CloudStudio to the version-controlled repositories. Its implementation depends on the distributed VCS used. Whereas the configuration awareness interacts with the file system to save files and get the current file version while being unaware of the version control system. It can only operate on the version of the file that currently is in the working directory of the repository.

2.2 Configuration Management

2.2.1 Terms

Each VCS has its own terminology. Some of the terms used in this report are adopted from the distributed version control system Git [7]. Other notions are specific to CloudStudio's simplified version control.

- *Repository* A repository is the folder of the VCS containing its meta data and, if the repository is not bare, the working directory. The meta data consists of all needed version control information like the different revisions, the branches and the configuration.

  The terms *user repository* and *main repository* are referring to the architecture of the repositories as explained in section 2.2.3.

- *Working directory* The working directory contains the checked-out version of the files. The files can be modified without using the VCS. They also will be influenced by certain version control operations like a rollback.

- *Staged changes* The VCS is unaware of modifications made in the working directory without its help. By staging the changes, they are added to the index of the VCS and they will be automatically included in the next revision.

- *Uncommitted changes* The modifications that differ from the head revision of the working directory are denoted as uncommitted. This equals the staged as well as the unstaged changes.
- *Revision history* The commits leading to the head revision make up the commit history. A revision may have more than one parent if it's a result of a merger of multiple revisions. The revision history can therefore be more complex than a linear list of revisions.
- *Head revision* The head revision is the last commit in the revision history. Usually this will also be the most recent commit, unless the head is moved by going to a different revision.
- *Branch* A repository can have multiple branches. Each branch has its own head revision. When the repository is created, it already has one branch denoted as the default or master branch.
- *Current branch* The current branch is the branch checked out in the working directory.
- *Commit* By committing a new revision is created in the repository with either all the modifications or only the staged changes depending on the options used.
- *Share changes* Sharing refers to exchanging and merging revisions with the main repository. Depending on the circumstances, sharing changes includes a previous commit. The term is represented by a chain of operations in Git and can also be compared to the notion of "update" used by centralized VCS like SVN.
- *Merge* is used as a synonym for sharing changes in this report.
- *Hook* A hook is a script saved in the meta data folder of the repository. It is executed by the VCS before or after a certain event happens, such as committing or receiving a new revision

2.2.2 Distributed Version Control System

The distributed version control system (DVCS) are more flexible compared to the centralized VCS and can be used independently of a central server. Although having a server would not be a problem when integrating a VCS into CloudStudio, the flexibility of DVCS would have been greatly missed. With a DVCS it is possible to merge between any two repositories whereas

a centralized VCS focuses on sharing with the central repository. The design chosen, described in section 2.2.3, relies on this ability of the distributed VCS.

Git and Mercurial are two commonly used distributed version control systems in software development. To decide which DVCS is suited best for the goals outlined in this thesis, multiple criteria were considered:

- Functionality
- Advanced features
- Integration with CloudStudio
- Installation

The functionality is similar for both distributed version control systems especially when looking at the standard features. Qualitatively comparing the two, Git's functionality used to exceed Mercurial's, but Git can also be more difficult to use. Both systems provide hooks to add additional behavior, for example after committing. As described in [6] Mercurial used to keep its revision history immutable whereas Git provides functionality to rearrange the revision graph, so called rebasing. The same article also comments on the difference between their approaches on branching. However, during the evolution of the systems the gap closed between their functionality.

To interact with the version control system, the CloudStudio server has to be able to connect to the repository and perform operations. Also it would be possible to use a connector in another language than Java, a Java API provides a tighter integration with the CloudStudio server and better support. For both version control systems multiple APIs exist which allow interaction with the repository. There even is an implementation of Git in Java named JGit [5] which has been originally developed as part of the Eclipse plugin for Git. JGit has the advantage of a full integration of Git in Java with an object-oriented design. Operating on the repository is done by simply invoking a method and the returned objects can be used without further transformation. Additionally, concerning the third criteria, there is no supplementary effort to install Git on the server. Only when using the Apache HTTP web server to make the repositories directly available without using the web-based IDE, an installation of Git is necessary.

Due to this analysis it was decided to implement the automatic version control using JGit [5], the Git implementation in Java. However the design has been kept modular, such that it is possible to add support of Mercurial or another distributed VCS with little effort.

The integration of Git during the thesis has supported the results of this analysis. JGit offers most of Git's functionality. An implementation of Git compared to a simple API offers the benefit of being able to debug deep into the Git functionality. With thorough knowledge of Git and its objects [13], JGit can be extended. Some features that Git has are missing in JGit. For example stashing the uncommitted changes is not available. This influenced some design decisions and the implementation. Furthermore JGit will not call the hooks of the repository. Being aware of JGit specific behavior it was possible to take it into account and adapt it where necessary.

2.2.3 Architecture

A CloudStudio project has multiple repositories, a bare main repository and a repository for each user assigned to the project as shown in the figure 2.2. The main repository is bare because its files do not need to be changed and committed. It is the focal point of the architecture.

The user repositories aren't bare. They have a working directory containing the checked-out files that can be altered using the configuration awareness shown in figure 2.1 to access the file system. The user will share new revisions with the main repository and not directly with other user repositories. This architecture allows each user to have his own version of the project files. There will be no need to synchronize the access to the files.

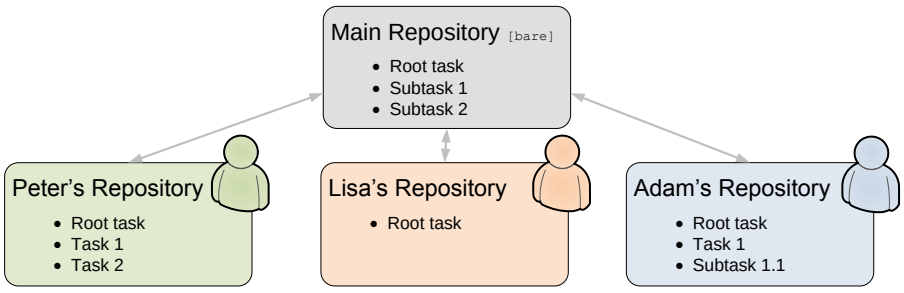


Figure 2.2: Example of the repositories in a project with three users

Additionally each project can have multiple tasks that are arranged in a tree and can be assigned to one or more users.

The root task is created implicitly. Any user in the project is automatically assigned to the root task. A new task can be create as a sub task of the root or any existing task. The creator is automatically assigned to the new task. Any assigned user of the task can add new users.

The tasks are represented by branches in the repositories. The default branch represents the root task and exists in every repository. By creating a task, a new branch is added in the main repository and checked out by the user repository. The repository of a newly assigned user will checkout the branch lazily as soon as the user wants to work on the task.

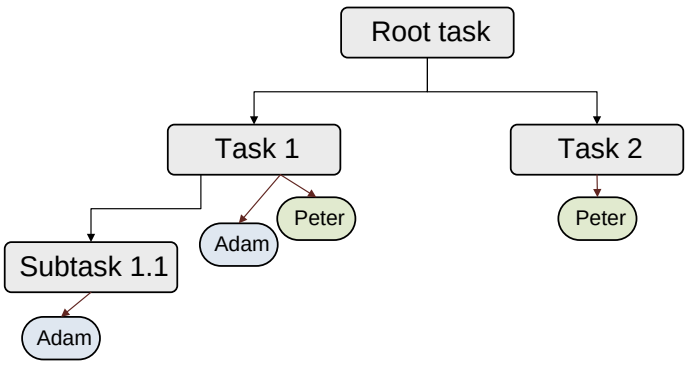


Figure 2.3: Example of a task tree (matching the example of figure 2.2)

If a user shares the changes, the modification will be merged into the main repository and his current branch. Due to the automatic version control the other user repositories are also updated, if the according option is enabled.

2.2.4 Functionality

The functionality of the CloudStudio configuration management is separated into the manual and the automatic features. The following list shows the manual features the user can access using the web-based IDE:

- *Commit* By performing a commit the user can select what files he wants to commit and a new revision with the given commit message and the modifications of the selected files is created.
- *Share changes* Sharing changes will merge the head revision of the user and the main repository. If there are any uncommitted changes, they will be committed before the merger. Any conflicts will be automatically resolved.
- *Rollback* A rollback can be executed in two different modes, everything or only uncommitted. If everything is rolled back, the working directory will be reset to the head revision of the main repository. Otherwise only the uncommitted modifications are reverted.

- *Show Log* Shows a log of the user repository with all its revisions.
- *Go to revision* Using the log it is possible to go back to a certain revision. Any uncommitted changes as well as the revision in between will be lost.

The automatic abilities of the CloudStudio configuration management are triggered by certain events. The action they perform, is similar to the manual features available. There are two automatic features:

- *Automatic commit*: If the user has successfully compiled the code in his working directory, an automatic commit is triggered. All modifications will be committed using a generated commit message. This feature can not be disabled.
- *Automatic share*: Automatically sharing executes a merger between the main and the selected user repository. There are two ways how it can be triggered. In the first case when another user shares new changes. The other team members working on the same task will be selected for an automatic share. The second way to trigger the merger is by an automatic commit.

  The automatic sharing can be disabled by the user in the web-based IDE. The setting is saved as part of the user information and not reset when logging out.

2.2.5 Access from the Outside of CloudStudio

One of the goals of this thesis is to make CloudStudio projects accessible without using the web-based IDE. The Apache web server fulfills this task by making the Git repositories available over HTTP. It runs as a service on a different port than the CloudStudio Java web server and forwards the Git communication accordingly. The Apache server is configured such as to restrict access to each project and require authentication. With the CloudStudio password being SHA-1 encrypted it can be used to devise password files without degrading the password security. The restriction of each repository is regulated separately. Only the user himself can create a copy of his user repository whereas all project users are admitted to the main repository.

Access to the user repository is possible thanks to the properties of a distributed VCS. With a centralized VCS only the main repository could be used to create a copy of the repository on the user's PC. However there are additional points to consider when cloning a repository with working directory as it is the case for the user repositories. The working directory will not

be updated automatically by Git when a new revision is pushed. Additionally the version control system will not force the outside source to resolve conflicts with uncommitted changes. These problems can be resolved by using hooks, which are called before or after certain events occur like receiving an update. The implementation of those hooks is explained in section 3.3.3.

The Apache server can not enforce access restriction of the branches. The CloudStudio task represented by a branch can be assigned to users. Yet when connecting from the outside all team members are able to access any task by cloning the main repository. Although the risk could be avoided by not allowing access to the main repository at all, there would be serious disadvantages. It is beneficial to the use of the change awareness to push changes from the outside to the main repository. Modifications pushed to the user repository are still perceived as the user's changes and presented not very differently from the uncommitted modifications. The changes will still be in the user repository and, even so they are visible to others, the modifications are not shared with the team and the main repository. It is recommended to clone the main repository when working on a project using Git and another IDE like for example EiffelStudio.

The Apache's access restrictions are part of the server configuration. For a new configuration to take effect, a restart of the server is required. The CloudStudio server will tell the Apache service to restart whenever the configuration has changed. Since the restart command of the Apache HTTP server is used, the availability of the server is hardly influenced. However it may take a few minutes before the project will be available over Git. It is further necessary to open the project first in CloudStudio.

2.3 Intermediate Representation

The information of the file system and the version-controlled repositories is collected in a intermediate representation. This representation is sent to the CloudStudio client which will use it to display the file content and the folder structure.

The information provided by the VCS is limited. It is possible to get a history of the file based on the commits. However for the fine, line-based granularity targeted by this thesis the available information is not enough. How is it possible to differentiate between a user changing an existing line or him deleting it and adding another line instead? During the implementation of the change awareness and the automatic version control it became imperative to have additional data concerning the past version of each line. Knowing the detailed history for each line is the key to correctly display the

merged file content including the changes of other users. It enables the automatic version control system of CloudStudio to find reasonable resolutions of conflicts without any user interaction. It is a significant decision to have an intermediate representation with more data than the VCS can provide. It has made the design and implementation much more complex and error-prone. However there is no way to offer the same functionality without it.

It would have been possible to implement a line-based approach only considering the commit history of the file. By iterating over each commit the modifications of a user can be shown in a certain granularity depending on the frequency of the commits. However the difference between two revisions of a file still doesn't yield the accuracy of a editor designed to maintain a line-wise version model. Only if committed regularly and not just once a day the repository can provide a fine-grained history. The intermediate representation is a merged version of the content of the main and the user repositories. It integrates all the branches as well as the uncommitted changes. The merging is no longer a problem as the assembled and merged model is constantly maintained and kept up-to-date. To recreate the full abstraction out of the version control system and the file system would be time consuming since it would be necessary to iterate over all past revisions. Nevertheless in some cases this approach is used in this thesis. When there is only limited information and no better precision available, it's best to iterate over the commits to create a change history. This is for example the case when receiving changes from the outside sent by a user using the HTTP connection to the Git repository instead of the web-based IDE. It is also applied when manually switching to a previous revision, because the intermediate model itself is not under version control.

2.3.1 Tag

The folder structure as well as the content of each file may differ for the repositories and their branches. The different version have to be merged together to provide the user with awareness of the modifications of other users. To denote whether a file, a folder or a line exists for a certain user and task a tag is used. As a file can exist for multiple users and tasks, it will also have multiple tags.

It proved to be useful to add information about whether a line is committed and if it has already compiled, which will be further referred to as version control status of a tag. The version control status is used during compiling with changes of other users. To prevent introducing further compilation errors the CloudStudio server will only consider modifications that are committed or have compiled previously.

In this thesis often when talking about tags it will only be mentioned whether the item exists for the user or the main repository. If not mentioned otherwise the notion of the main repository means the main repository and the root task. Since the main repository has no working directory, a tag of the main repository is always committed. When talking about the user's tag without mentioning a specific task, we usually refer to the user repository and the current task. The version control state in the user repository can be uncommitted, committed or committed and compiling.

2.3.2 File Content

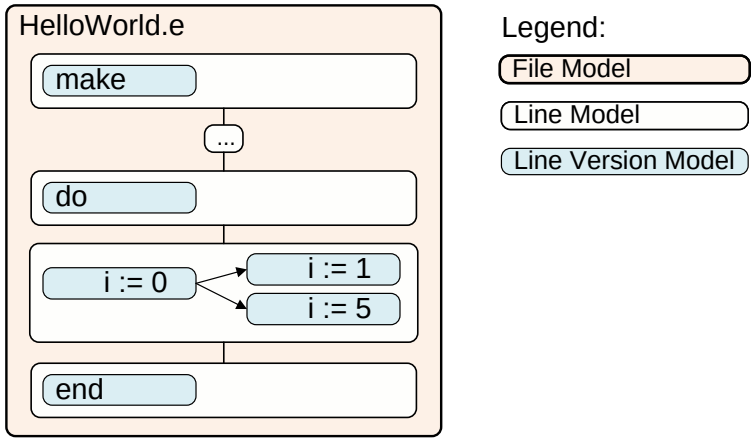


Figure 2.4: Example of a file with multiple lines and line versions

As reasoned before, the new CloudStudio editor will keep track of the file content using a fine-grained model based on lines. To represent the file content of one specific file, the text of the file is split into lines. Each line has a content and can exist for one or more tags, where each tag represent a different version of that file. To identify a line, a randomly generated integer is assigned as unique identifier. This has the advantage that the client can create new lines and send them as an update to the server. The unique identifier are a convenient way to avoid complicated synchronization. The lines are aligned in a list, where each line points to its predecessor and its successor using the unique identifier. If a new line is inserted, the pointers are relinked to add the line in between. All the lines together make up the combined abstraction of all versions of the file. The combined file model is serialized and persisted. It has to be continuously synchronized with the content in the repositories.

Lines will be regularly changed. To detect conflicts and merge the content of different versions of the file successfully a representation with multiple line versions is required. This approach will allow the editor to still be aware which line versions belong to the same line. In the editor as well as the merged content, at most one version of each line will be included and displayed.

The line versions are arranged as a tree as shown in figure 2.5. Each line has a root line version. All other versions are children of the root. The line version of the main repository and the root task is denoted as main line version. A line version has two attributes, its content and its tags. The version can be associated with zero or more tags, meaning the content of the given user and task is equal to the content of the line version model. A deleted line will be represented as a line version without content. This way deletion and absence are distinguishable. Two empty line version with no tags in the line version tree will cause one of them to be removed to keep the tree from growing indefinitely.

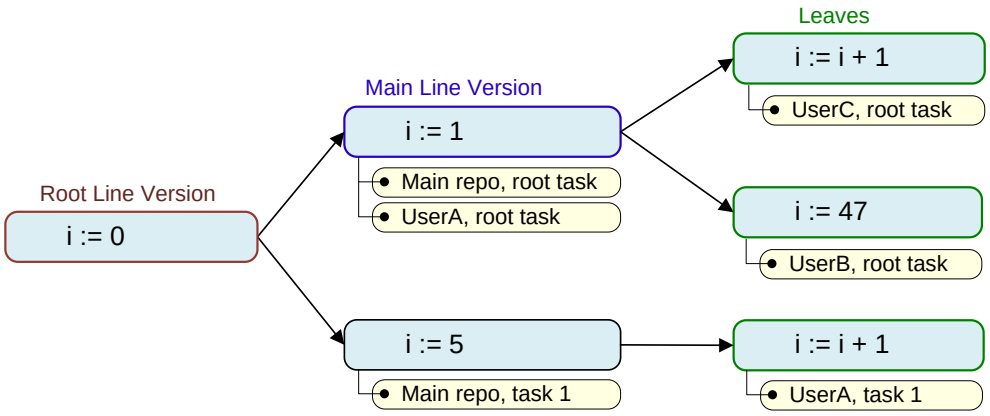


Figure 2.5: Example of a line version tree with the associated tags

Keeping the line versions in a tree instead of a simple set improves the conflict detection. The line is in a conflict if there is more than one leaf. If there are multiple leaves, the editor will always prefer displaying the own version before displaying other ones. In case the own line version is not a leaf it will display one of its children. When editing a line in the editor, even one displaying another version than your own, it will create a new version with the displayed version as parent. This allows multiple users to interactively alter the same line without introducing a conflict assuming all users enabled seeing the changes of the other team members. It also requires the period

between edits to be long enough to allow each client to send and receive the update from the server.

The web-based IDE allows the user to choose which users and tasks he wants to see in the editor. By doing so, some line versions in the tree are no longer considered since no tag of the version matches the chosen users and tasks. It will change which nodes are leaves in the version tree and accordingly a different line version is displayed in the editor. The conflict detection is independent of the displayed tags. It will always consider all users and tasks to alert the developer to potential conflicts.

2.3.3 Folder Structure

The folder structure consists of a tree of folders with files as leaves. Equivalent to the lines in the file, the files and folders are annotated with tags to identify for which users and tasks they exist. Additionally a flag for each file and folder indicates whether the file has been deleted somewhere. Using this flag tells the automatic conflict resolution to delete the file, if the file has been removed in one of the two revisions to merge. The user will be automatically warned if he is working on a file with the deletion flag set to prevent conflicts. CloudStudio can not exactly differentiate between deletion and absence of a file for each revision like it is possible for lines as described in the previous section. However the deletion flag provides the ability to know if a file has been deleted. When the flag is set, sooner or later the deletion will be shared and the file will be removed for each team member. The user is alerted to this conflict and can restore the file to prevent its deletion. Restoring will remove the deletion flag and prevent the file from being deleted during a merger.

Keeping a history of the folder structure is not as crucial as maintaining the intermediate model of the file content. It is possible to scan the repositories and each of their branches for all files and folders. However experience and performance tests proved it to be too time consuming. The scanning can take up to several seconds for small projects and will scale linearly to the amount of files. Additionally to keep track of deleted files the commit history has to be iterated. To solve this performance issue and to implement supplementary functionality to restore deleted files, the folder structure is persisted as well.

2.3.4 Conflict Detection and Resolution

Based on the described intermediate models and the change awareness it is possible to resolve all conflicts automatically. The resolution strategy used during sharing with the main repository is the same as the one used in the

editor or when compiling with the changes of other team members. Due to this uniformity the merged code can be compiled, tested and proofed. It also helps the user to implicitly understand how conflicts are resolved during development. The web-based IDE provides tools to alter the result of the merge. Additionally the user can simply edit the line, if the current merged content doesn't provide the result he hoped for.

The advantage of the automatic conflict resolution lies the fact that the user does not have to interact to merge the changes. The implemented automatic share feature depends on this, as it will automatically merge new changes of the main repository into the user repository as well as the other way around. With the goal to ease the usage of the version control, such a process should not depend on the interaction of the developer.

Conflicts are resolved by analyzing the two revisions of the source and the destination. In case of the folder structure, if the file is deleted for one version, it will be deleted in the merged version as well. The only way to prevent this is to restore the file.

The merger of the file content is done by merging all the lines. If a new line has been created, it will also be added in the merged content. A deleted line will be removed, unless it is a conflict. Conflicts in a line can be dealt with automatically using the line version tree as shown in figure 2.5. If one of the line versions is a child of the other, the conflict is resolved by taking the most recent version. If the line versions are not directly related, the user's version is returned rather than the version of the main repository. This approach enforces the merger of new content into the main repository. The user can influence the outcome of the share proactively by using the functionality of the web-based IDE. It is possible to switch to the main revision or to a conflicted version in the tree. This strategy of conflict resolution is based on the assumption, that the user will use the change awareness. If the user chooses to disable all change awareness and ignore the conflict annotations described in the section 2.4.1, the automatic conflict resolution may not provide the expected result.

2.4 Change Awareness

The automatic version control system is implemented based on change awareness, which makes the awareness one of the main goals of this thesis as well as a means to the end. Without the change awareness it is not possible to implement automatic conflict resolution as described in the previous section 2.3.4, which is essential to an automatic configuration management. The user has to be aware of possible conflicts and be able to compare his own revision to

the versions of others. This approach is already used by the standard version control system. However the intent of this thesis to make the user continuously aware and help him to proactively resolve conflicts is new. It is the logical step when trying to automate and ease the time consuming version control activities.

Both, the editor and the explorer, use change awareness to help the user be unobtrusively conscious of the modifications of other team members. Their content is updated in real-time with updates from the server if another user changed something.

2.4.1 Editor

The editor used in CloudStudio is based on the Ajax.org Cloud9 Editor [2], short ACE. It comes with support of multiple languages and was extended and integrated into CloudStudio during a semester project taking place simultaneously with the first half of the thesis. By adapting the displayed gutters, a color notation has been added. A hover text will display for whom the line exists and what other versions there are.

The user may not always want to see all the changes from other developers and tasks. He can disable them and they will no longer be directly visible. However an arrow in between two lines will indicate hidden lines. It also won't affect the conflict detection. A line marked as conflict, will still be marked red, whether the conflicting version would be displayed or not.

If the user changes the file content, the intermediate file model is updated by interpreting the delta provided by the ACE editor to adapt the line versions. The updated lines are sent to the server and the server redistributes them to the other clients. All users will see the updated file within seconds.

2.4.2 Explorer

The old file explorer of CloudStudio didn't include any change awareness. Using refactoring the modularity of the explorer has been improved to make it almost language-independent.

The files and folders displayed are colored according to their tags. More information about the file is supplied in an extra dialog. The message box shows whether the file exists for the user and the main repository.

The folder structure displayed in the explorer will be updated automatically, for example if another user adds a class. Similar to the update of the file content, the added file will be redistributed over the server to all CloudStudio clients.

Chapter 3

Implementation

CloudStudio is implemented in Java and runs on a Tomcat server. The automatic version control uses JGit [5] to interact with the Git repositories. To allow users to connect without a CloudStudio client, an Apache web server provides access to the repositories via HTTP. The CloudStudio server saves the user and the project information in a MySQL database accessed using the Java database connectivity (JDBC).

The web client is programmed in Java using the Google Web Toolkit [3] and is then compiled to JavaScript. The client communicates with the server asynchronously. The data passed is represented as models which are serialized for transmission. An event-based approach is used for the server to communication with the client using a long-polling server push. The editor of the client is based on the ACE [2], Ajax.org Cloud9 Editor. It is integrated into GWT using the AceGWT library [1]. The CloudStudio client is optimized for the browser Google Chrome.

The UML diagrams displayed in this chapter may be simplified compared to the actual implementation to improve the comprehensibility.

3.1 Services and Events

The old version of CloudStudio already included several services handling each a specific area, such as the user, the project or the file management. There is one instance of each service for each logged in user. Most services had to be refactored and adapted to the new backend, but their functionality remained the same.

Two new services have been added. The `ConfigManagementService` offers the client the ability to interact with the version control system. The change awareness and the combined file models integrating all versions of a file

can be accessed using the `ChangeAwarenessService`. When a service needs to invoke a method of the backend, the `ConfigManagementFactory` is called to get a backend controller encapsulated by the `ConfigAwarenessSystem` or the `ConfigManagementSystem` interface.

To inform the CloudStudio clients about any updates from the server, events are sent using long-polling to implement a server push. The client will open up a request which the server will not answer immediately but wait until there are events to push. If the request closes with a timeout, the client will just reopen another one. This method is often used in web applications, since not all browsers fully support new substituting technologies like Web Sockets.

Three new events are introduced with the new backend to send updates to the client, the `ChangedModelsEvent`, the `UpdatedCombinedFileModelEvent` and the `ReloadFolderStructureEvent`. Those events are used to provide the CloudStudio user with real-time updates of the progress of his team members and continuous change awareness. The `ChangedModelsEvent` is triggered, if some intermediate models have been changed. The event will send the modified models to the client. It is used when multiple files and folders are affected, for example when performing a version control operation such as a commit. The `UpdatedCombinedFileModelEvent` refers to one changed file and is used when user is typing in the editor. The event will include the set of modified lines to enable the CloudStudio client to update its file model. The third event is the `ReloadFolderStructureEvent`. It is rarely used and helps when it isn't possible to tell which folder structure entries have been changed. When triggered, the client will reload the whole folder structure of the IDE explorer.

3.2 Models

3.2.1 Tag Model

The tag model is used to uniquely identify the revision the line version or the file is associated with. To accomplish this it contains the information about the user and the task. Based on the user the repository can be identified. `null` refers to the main repository. The task name of the tag equals the branch name in the repository.

Additionally the tag has a `VersionControlStatus` with one of the following three values:

- `UNCOMMITTED` The item has been changed, but not committed.
- `COMMITTED` The item has been manually committed.

- `COMPILING` The item has been automatically committed when the user compiled successfully.

The values of the `VersionControlStatus` can be ordered by recency as shown in equation 3.1. If an item is annotated as `COMMITTED`, but has no other tags of the same user and task, it implicitly exists for the more recent value `UNCOMMITTED` as well. To check whether an item exists for a specific tag, the tags of the item are iterated. First the loop will look for a tag model with the according `VersionControlStatus` and then if it can't be found, it will check for any less recent values. This approach is based on the assumption that deleting an item can be denoted by a specific value such that deletion will result in an entry with a more recent tag similar to a change.

$$\text{Recency} : \text{UNCOMMITTED} > \text{COMMITTED} > \text{COMPILING} \tag{3.1}$$

To find an exact match between two `TagModel` instances the user, the task name and the `VersionControlStatus` have to be compared. In many situations it's enough to look at the user and the task as for example to check if the line exists for the logged-in person and his current task.

The tag model appears in both, the class diagram of the file content in figure 3.1 and the diagram of the folder structure in figure 3.2.

3.2.2 Models of the File Content

The file content is modelled by three classes as shown in the figure 3.1. The file specified by its path is represented by a `CombinedFileModel` instance containing a map of lines. Each line is a `LineModel` object with a randomly generated integer as a unique identifier. Using this identifier the line points to its predecessor and its successor. A line has multiple line version arranged in a tree. Each line version represented by the `LineVersionModel` class has a content and a list of tags for which this content applies.

The `CombinedFileModel` is, as its name says, a combination of all versions of the file. It assembles the file content of each repository and each branch and differentiates between whether the content was committed or not. The model is persisted by saving the serialized object in the project folder. The `CombinedFileModel` is initialized when the file is first opened in the web-based IDE or if it is altered in a commit from an external repository. The initialization is based on the fact that when initialing the file contents are all the same for all versions of the file. As the model is persisted the challenge is no longer the initialization, but to keep it synchronized to the content in the version-controlled repositories as discussed in section 3.4.

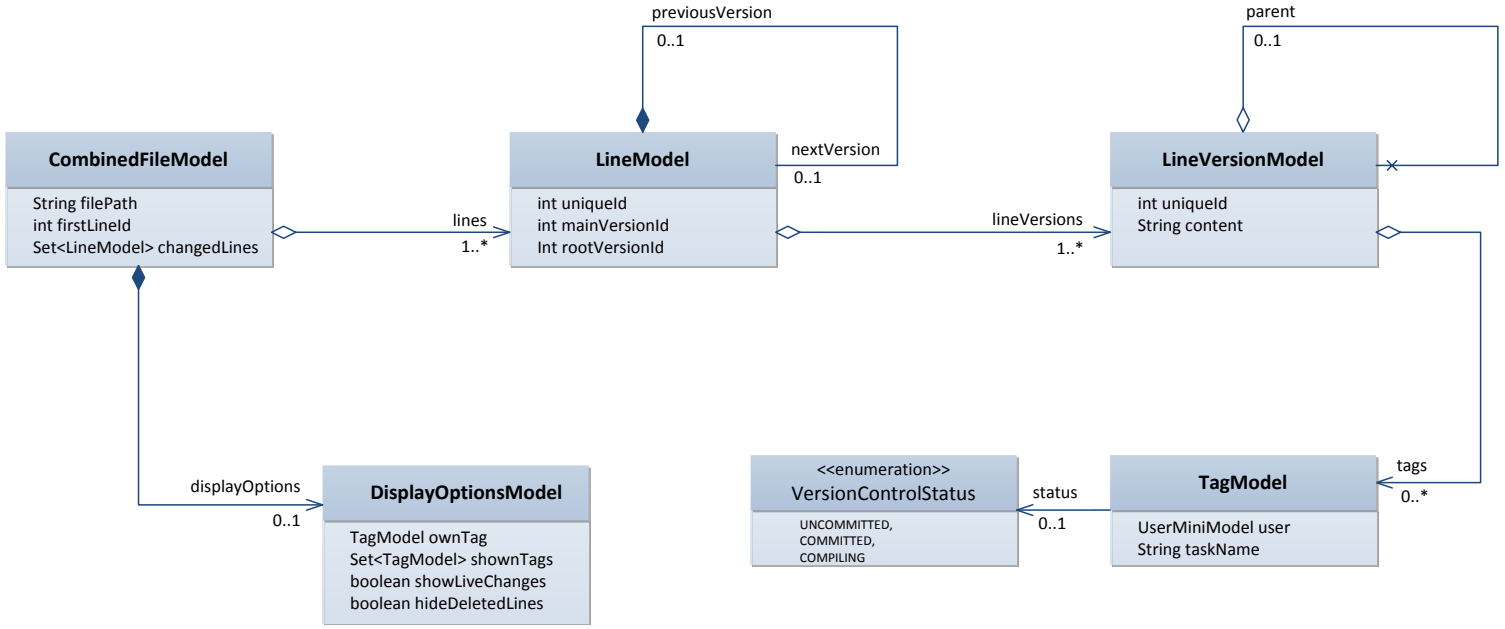


Figure 3.1: Class diagram of the models of the file content

The `CombinedFileModel` has contains an `DisplayOptionsModel` instance. It is used to assemble all options and settings of the CloudStudio editor and is also used in other contexts. The attributes of the `DisplayOptionsModel` are:

- `ownTag`: The own tag identifies which line version has priority and is displayed, if there are multiple versions in a conflict.
- `shownTags`: The shown tags restrict what content is displayed.
- `showLiveChanges`: If the live changes are disabled, only tags with a status of value `VersionControlStatus.COMMITTED` or `COMPILING` are displayed. Uncommitted changes are ignored for everybody, except the user and his current task represented by `ownTag`.
- `hideDeletedLines`: Hiding deleted lines will prevent lines from being displayed, if the content of the displayed version is `null` meaning the line has been deleted. By disabling it, the line versions with content `null` are ignored.

Using those options, the `CombinedFileModel` can determine the content to display in the editor. The same method is also used when saving back the own content into the working directory of the user or when compiling with the modifications of the other users. Using the file model and its options it is also possible to translate a `LineModel` to a line number and vice-versa. All those methods related to the displayed content depend on the options. Lines added by another user may or may not be displayed depending on the setting of the `DisplayOptionsModel` and whether the user is part of the shown tags.

The `CombinedFileModel` keeps a map of the lines of the file indexed by their unique identifier as well as the unique id of the first line. It declares methods to insert lines or to delete them which can be used by the CloudStudio server as well as the client. By inserting a line the new `LineModel` will be added to the map of lines of the `CombinedFileModel`. If a line has been completely deleted for each and every version of the file, the `LineModel` is removed from the map. During insertion as well as deletion the pointer of the line and its neighbor are updated such that the lines will form a bidirectional list. The line directly affected by the modification will be added to the set of changed lines of the `CombinedFileModel`.

On the client side when the user is editing, the `CombinedFileModel` is kept up-to-date by listening for the on-change event of the ACE editor. The triggered method will receive a delta of the editor as a parameter. The delta contains all information needed to recreate the modification in the file model. The delta is interpreted and the modified lines are added to the set of changed

lines maintained by the `CombinedFileModel`. After adapting the file model, the text of the editor as well as the change awareness notations are updated if necessary.

Each modification in the client triggers the timer for sending an update to the server if it's not already started. The use of the timer prevents updates to be more frequent than each second, which would be inefficient. To inform the server about the modifications in the client the set of changed lines maintained by the `CombinedFileModel` are used as parameter to invoke the update method on the `ChangeAwarenessServer`. The server processes the update and redistributes it to the other users as well. By making use of the unique identifier of the `LineModel` it is possible to insert the new line correctly. It is first identified by its predecessor and if the previous line was not found, the server will try to find its successor to place the new line before it. This allows multiple users to synchronously change a file without content being lost.

3.2.3 Models of the Folder Structure

The folder structure is a tree where the files are arranged as leaves. The files are represented by `FileMiniModel` instances and the folders by `FolderModel` objects. Each item of the folder structure, as shown in the class diagram in figure 3.2, has a path and a name. The existence tags associated with the item determine for which version the file or folder exists, whereas the deletion tags indicate an item has been deleted in the repository for the given branch.

The explorer of the web-based IDE may not only show existing files and folders. Depending on the programming language used it will also list artificial entries which have no respective item in the project folder, for example the libraries in Eiffel. Additionally some folders are immutable and can't be deleted. For the Eiffel projects the IDE explorer uses on additional classes like the `ClusterModel`, the `LibraryModel` and the `PrecompModel` to represent the folder structure.

Based on the type of the file or folder and its other attributes, the icon displayed in the CloudStudio IDE can be determined, as well as the operations the user can perform on the item like deleting or adding a new class.

The items of the folder structure, all inheriting from `FileSystemModel`, further have a change type. By using the folder structure annotated with the change type, the commit dialog in the CloudStudio IDE can display which files have been added, modified or deleted.

Just like the model of the file content, the folder structure is persisted. It could be reconstructed and initialized from scratch whenever a user opens the project, but the process would be too time consuming.

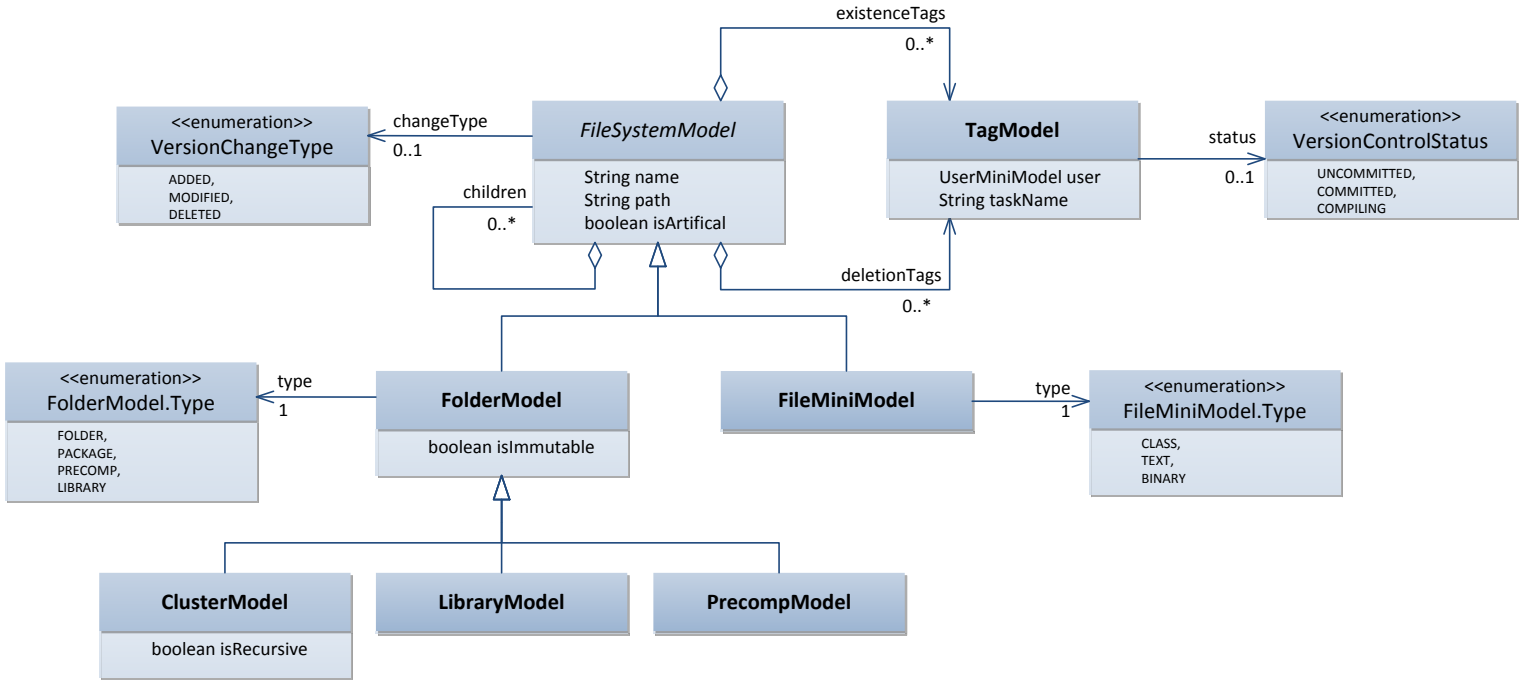


Figure 3.2: Class diagram of the models of the folder structure

3.2.4 Task Model

The task model is a representation of all branches of the project repositories as a tree. Each task has one or more users assigned to it as shown in figure 3.3. Those users are allowed to switch to the task, create new sub tasks, add another user to the task or even delete the task for everyone. As the branches representing the tasks are an unstructured list and not a tree, it's essential the tree of `TaskModel` objects is persisted. The model is serialized and saved in the project folder.

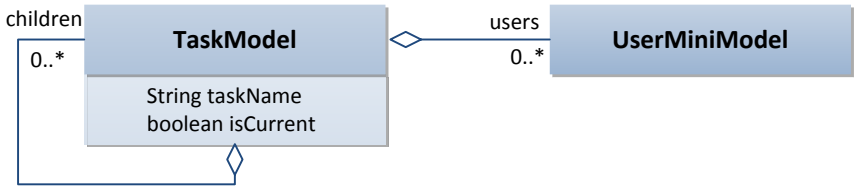


Figure 3.3: Class diagram of the task model

The tasks can be managed using the web-based IDE. Each project has its own task tree. Using the dialog to manage the tasks, as shown in figure 3.4, the user can easily create new tasks and switch to another. Creating a new task will add a new branch to the user repository starting with the revision of the parent task. The created branch will be merged into the main repository.

Figure 3.4: Screenshot of the task management in CloudStudio

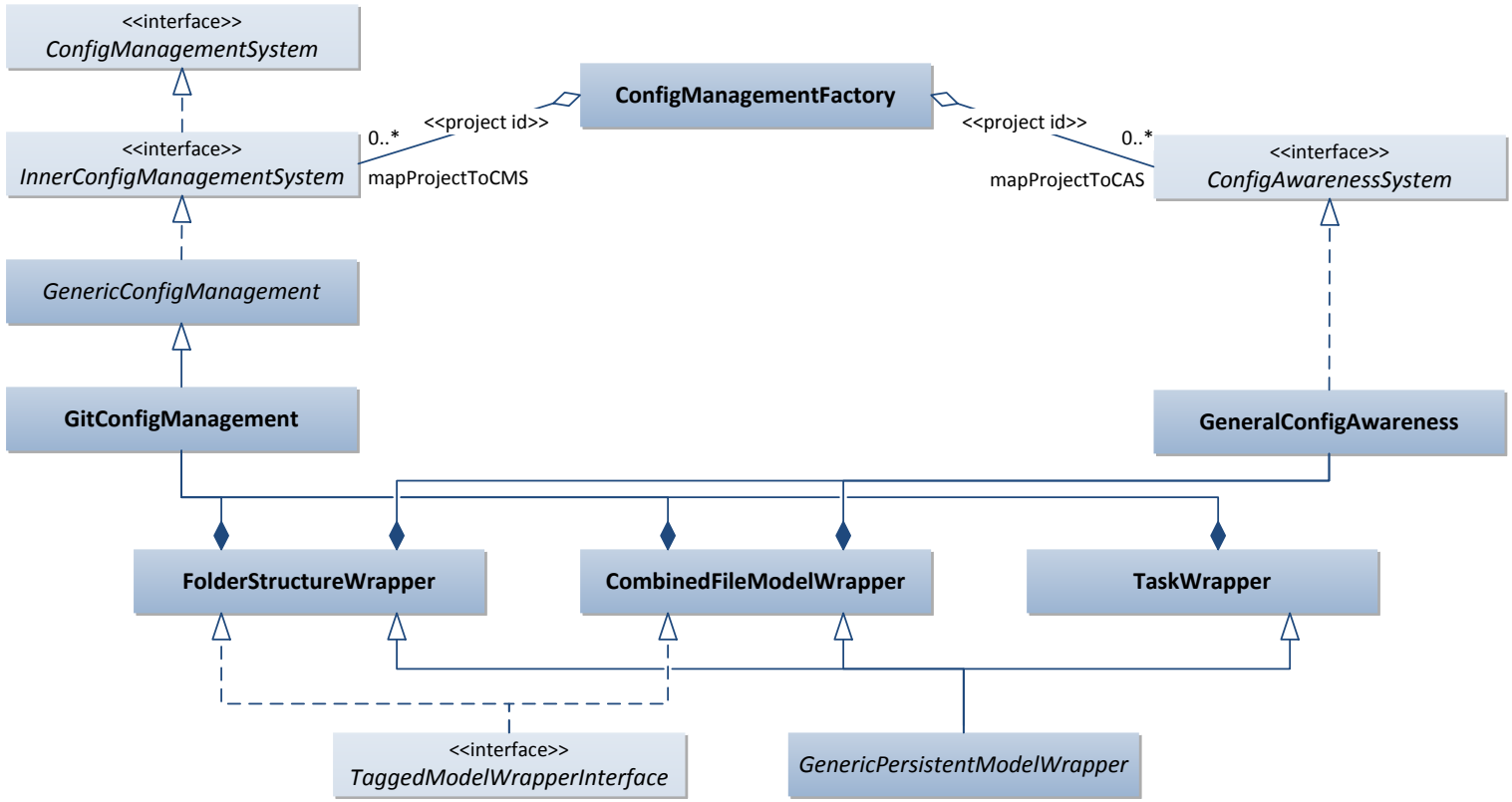


Figure 3.5: Class diagram of the backend

3.3 Backend

The backend can be accessed using one of the two interfaces as shown in figure 3.5. The `ConfigManagementSystem` provides a connection to the underlying version control system. The implementation of this interface depends on the distributed VCS used. The `ConfigAwarenessSystem` is the access point to the file system. Its implementation is independent of the version control system. Designed using the factory pattern, a project-specific instance implementing either interface can be acquired using the `ConfigManagementFactory`.

To improve the modularity of the code, wrappers are used providing special functionality to both the configuration management as well as the configuration awareness. The `TaskWrapper` is used to manage the tasks. To adapt the model of the folder structure the `FolderStructureWrapper` is called. The `CombinedFileModelWrapper` is responsible for the combined file models with their lines and line versions.

All wrappers extend the class `GenericPersistentModelWrapper` to manage their persistent models. The persistent data will be read lazily and the loaded model is kept per reference in the wrapper. The `FolderStructureWrapper` and the `TaskWrapper` keep only a single reference to the root of the tree whereas the `CombinedFileModelWrapper` has a map of models, potentially one for each file in the project. The wrapper of the folder structure and the wrapper of the file content additionally inherit the `TaggedModelWrapperInterface` which specifies the methods to keep the models synchronized with the version-controlled repository using the tags. The approach is explained in more detail in section 3.4.1.

3.3.1 Interfaces

The `ConfigManagementSystem` interface is the larger of the two interfaces. It can be split into two parts, the task management and the version control management. The implementation of the task methods operates on the task tree using the `TaskWrapper`. The following method signature are part of the task management in the interface:

- `addUserToProject(UserMiniModel user)`
  The user is assigned to the project. A new repository is created by cloning the main repository.

- `createTask(UserMiniModel user, String task, String taskParent)`
  A sub task is created as a child to the given parent task. If the user is not a member the parent task, he is not allowed to create a sub

task. The creator will be automatically assigned and switched to the newly added task. All content as well as the history is inherited from the parent task. Any uncommitted changes of the user repository will be lost.

- `addUserToTask(UserMiniModel user, String task, UserMiniModel newUser)`
  By invoking this method a member of the project will be assigned to an existing task. The user invoking the operation must already be part of the task to be allowed to add a new user. The branch will not be immediately added to the user repository, it will lazily be copied from the main repository when `switchToTask` is called by the new user.

- `switchToTask(UserMiniModel user, String task)`
  The user switches to the given task. Any uncommitted changes of his repository are discarded and the named task is being checked out in the working directory. If the user didn't work on the task before, he will pull it from the main repository.

- `deleteTask(UserMiniModel user, String task)`
  The task is being deleted for all the team members. A user is only allowed to delete a task, if he is assigned to it. Deletion will remove the branch from the main repository as well as from every user repository. Any changes not shared with another task will be lost.

- `String getCurrentTask(UserMiniModel user)`
  Get the current task of the user. When assigned to the project each team member starts working on the root task. The current task is changed by using `switchToTask`.

- `TaskModel getTask(String task)`
  Find and return the task model with the given name.

- `TaskModel getTasksByUser(UserMiniModel user)`
  Return the task tree and annotate the current task of the user as such.

- `boolean isValidTaskName(String task)`
  Check if the task name is valid and can be created. It will control if the name contains no illegal symbols and make sure there is no task with the same name.

The second part of the interface offers functionality of the version control system. The methods and return values are kept independently of the version control system used. There is certain advanced functionality that is not

listed here, that is available for the Git implementation `GitConfigManagement`. Currently it's possible to just cast the object and then access the Git-specific functionality. When an additional distributed VCS as for example Mercurial were to be added, it would make sense to add the methods in the general interface `ConfigManagementSystem`. This would require that the result types that currently are part of JGit are abstracted to conform any DVCS. The following functionality can be found in the interface currently:

- `commit(UserMiniModel user, String msg, List<FileSystemModel> files)`
  Commits the changes of the user repository with the given message. The methods accepts a selection of files as argument. It is recommended to use `getUncommittedFiles` to obtain them as the models have to be annotated with the `VersionChangeType`. If no files are selected and the argument is `null`, all changes are committed. Committing will change the tags with `VersionControlStatus.UNCOMMITTED` into tags with status `COMMITTED`.

- `autoCommit(UserMiniModel user)`
  Automatically commits the changes of the user with an generated commit message. This method will be invoked, if the user compiled the content of the working directory successfully. The tag with status `COMMITTED` or `UNCOMMITED` will be modified into `VersionControlStatus.COMPILING`.

- `shareWithCurrentTask(UserMiniModel user)`
  By clicking on share changes the user invokes this method. It performs a list of operations. First the user repository will pull new revisions from the main repository. Any conflicts will be automatically resolved and committed. Then the content of the user repository is pushed to the main repository. It will trigger automatic sharing for any user that has the option enabled.

- `shareWithTasks(UserMiniModel user, String targetTask)`
  Sharing between tasks will iteratively merge the revisions of the current task to its parent until reaching the targeted task and return in the other direction back. To do this, the repository will switch to the intermediate tasks, which will cause any uncommitted changes to be lost. The automatic sharing is triggered by this method as well.

- `FolderModel getUncommittedFiles(UserMiniModel user)`
  Get the tree of uncommitted files and folders. The files are annotated with the change type `VersionChangeType`, which can be either added, modified or deleted.

- `initialShareAndCommit(UserMiniModel user)`
  The initial share and commit is used to initiate a new project with its preexisting files. It will automatically add all files in the working directory to the version control system and commit them with a generated message. The new files are shared with the main repository.

- `rollback(UserMiniModel user, boolean everything)`
  Rolling back will reset the working directory of the user's repository. The option `everything` lets the user choose whether to get back to the last committed state or to reset to the latest revision of the main repository. Depending on the choice, the intermediate models will be updated. By just returning to the committed state, any tags of the user and his current task with the status `VersionControlStatus.UNCOMMITTED` will be deleted. If everything is rolled back, the tags of the main repository are copied to replace any tag of the user and his current task. This way when rolling back the meta data can be updated accurately, unlike when using `goToRevision`.

- `goToRevision(UserMiniModel user, String revision)`
  By going to specific revision the user can do even more than just `rollback`. It's possible to select any revision and switch the repository to its state. However the intermediate models can't switch their state, since they are not version-controlled. As such it is not possible to revert their state to just any revision. The solution is to undo each commit in between iteratively as explained in section 3.4.2. The accuracy depends on the size of the commits.

- `processExternalPush(UserMiniModel user, String task, String oldRev, String newRev)`

  This method is invoked by the RMI Listener when an external source is pushing new changes into the main or a user repository. The method parameters tell us which repository and which task are affected as well as the old revision and the new revision. The intermediate models are updated by iterating over the newly added commits as described in section 3.4.2.

- `List<RevisionModel> showLog(UserMiniModel user)`
  The method returns a linear list of revisions which represent the commit history of the user. Each revision contains a time stamp and a commit message. It can be uniquely identified by the revision hash.

Any method of the `ConfigManagementSystem` implementation affecting the intermediate models will trigger an `ChangedModelsEvent` sent to the CloudStudio clients to update their folder structure and files.

The additional interface `InnerConfigManagementSystem` provides methods to the backend classes that are not exported to the services of CloudStudio. It extends the standard functionality of the configuration management. The `GitConfigManagement` is the Git implementation of the `ConfigManagementSystem` interface. The VCS independent code as for example the updating the models is inherited from the class `GeneralConfigManagement`.

The `ConfigAwarenessSystem`, the second of the two main backend interfaces, is used to connect to the file system and operates on the working directory of the user repository. It contains methods to retrieve the folder structure and the combined file models. The following list shows most of its functionality, neglecting methods without significance:

- `List<FolderModel> getFolderStructure(UserMiniModel user)`
  Returns the folder structure, containing all folders and files annotated with their type. The tree is directly displayed in the explorer of the CloudStudio IDE.

- `String getFile(String filePath, UserMiniModel user, String task, boolean isUncommitted)`

  Get the file content of the user repository, or the main repository if the user is `null`, and the given task. The last parameter of the method determines whether the uncommitted content in the working directory or the content of the head revision is retrieved. The implementation will call the `ConfigManagmentSystem` interface to get the committed content.

- `saveFile(UserMiniModel user, String filePath, String content)`
  Saves the file with the given content. If the file doesn't exist, it will be created.

- `deleteFile(UserMiniModel user, String filePath)`
  Deletes the file identified by its path.

- `restoreFile(FileSystemModel file, UserMiniModel userMiniModel)`
  Restores the file, such that it will not be removed during merger. This method will not influence the working directory unless the file has just been deleted and the deletion has not been committed yet. It will change the way the file is merged. The restoring has an effect on all project members.

- `createFolder(UserMiniModel user, String path)`
  Creates a new folder.

- `deleteFolder(UserMiniModel user, String path)`
  Deletes an existing folder denoted by its path and all its sub folders and files.

- `existsPath(UserMiniModel user, String filePath)`
  Checks if the file specified by the path exists.

- `String prepareCompile(UserMiniModel user)`
  Prepares compilation of the user's content. The method will return the path to the working directory.

- `String prepareCompileWithChanges(UserMiniModel user, List<TagModel> userTags, List<TagModel> taskTags)`

  Prepares compilation with the changes of other users and returns the path to the merged files. The list of users and tasks shown are passed as a parameter. A temporary folder will be created and the merged content of the files, similar to the content displayed in the editor, will be saved into the folder. Only tags with status `COMMITTED` or `COMPILING` will be considered, such as to avoid compilation errors introduced by the live changes of the other users.

- `CombinedFileModel getCombinedFileModel(String path, UserMiniModel u)`
  Returns the file model identified by its path.

- `updateCombinedFileModel(String filePath, Set<LineModel> changedLines, UserMiniModel user)`
  Updates the file model with the changed lines. The method will also save the changed content of the file to the file system and trigger the `UpdatedCombinedFileModelEvent` to send the update to the other users.

3.3.2 Access over HTTP and Git

The Apache HTTP web server enables the user to work on CloudStudio projects with an IDE of his choice. It runs as a service. The CloudStudio server will restart the Apache server when the configuration has changed, for example when a new project has been created. The Apache server is optional and not required when running the CloudStudio server. It's also possible to add the access over HTTP later for an existing CloudStudio without any additional effort.

The configuration of the Apache web server required in order to integrate it with CloudStudio is kept as simple as possible. It's enough to insert a single line into the configuration file of the httpd process to include the generated CloudStudio configuration file. The CloudStudio Apache configuration will then again include the configuration of each project. All configuration files for the Apache are generated by the `ApacheAccessManager` class. To configure CloudStudio with Apache the `CLOUDSTUDIO_APACHE` environmental variable in the Tomcat context needs to be set to the path of the Apache installation. Further it's recommend to specify the HTTP address and the port of Apache in the `CLOUDSTUDIO_HTTP` context variable.

Listing 3.1: Configuration of the access to a CloudStudio Git repository

```
1  #Connection using git with login to the main repository
2  <LocationMatch "/ExampleProject/main/.*(git-(receive|
       upload)-pack)?">
3    AuthType Basic
4    AuthName "Access to the CloudStudio project"
5    AuthUserFile "C:/path/to/cloudstudio/
6                        ExampleProject/.config/main.users"
7    require valid-user
8  </LocationMatch>
9
10 #HTTP access with login to the main repository
11 <Directory "C:/path/to/cloudstudio/ExampleProject/main">
12   Options FollowSymLinks Indexes
13   AuthType Basic
14   AuthName "Access to the CloudStudio project"
15   AuthUserFile "C:/path/to/cloudstudio/
16                        ExampleProject/.config/main.users"
17   require valid-user
18 </Directory>
```

In the generated configuration file of a CloudStudio project the main repository and each user repository is listed. The block `LocationMatch` redirects Git calls. The `Directory` tag enables viewing the project directory and its files with a web browser. Both access methods are restricted by a repository-specific password file. The password file is generated using the user name and the password of the CloudStudio user.

The Apache HTTP server offers other modular approaches to extend the configuration which don't require a server restart. However the Git configuration uses the tag `LocationMatch` as shown in listing 3.1 and `LocationMatch` can only be used in the global configuration file.

A detailed description of how to setup the Apache server in Windows to export the CloudStudio projects can be found in the CloudStudio wiki.

CloudStudio offers the user the possibility to download a batch file to simplify the use of Git. The user can choose how long the script will wait between sharing with the CloudStudio server and whether to share automatically at all. If the automatic sharing is disabled, the user will have to hit enter when he wants to share and he is asked to enter a commit message.

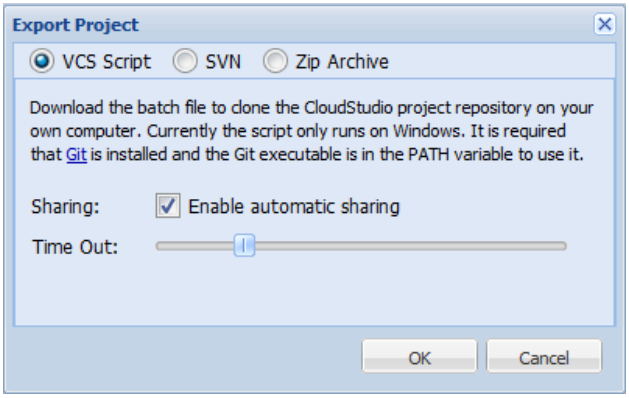


Figure 3.6: Screenshot of the dialog to export the Git batch script

Once downloaded the script can be started and will ask the user for his password. All other information like the project name, the user name and the HTTP address of the CloudStudio server are included in the script. If the user supplied the correct password the main repository of the project will be cloned. After the copy of the repository has been created the script will start a loop to share the modifications after the specified time period with the CloudStudio server. Instead of a regular update the user can also manually initiate one by hitting enter.

3.3.3 Git Hooks

Git hooks [13] are scripts that will be called when a certain event occurs. JGit, the Git implementation in Java, ignores these hooks. They will not be invoked if for example a commit is done using JGit. The CloudStudio implementation utilizes this fact to its advantage. Usually it's not needed to trigger the hooks if the CloudStudio server executed the operation using JGit. The only case where it's necessary, is after a commit. The CloudStudio server will manually execute the post-commit hook.

The hooks are created dynamically by the `GitHooksCreator`, one set of hooks for each repository. There are three types of hooks used:

- *Post-commit hook* The post-commit event is triggered after a successful commit. The hook performs an update of the server information, such

that the information published using the HTTP server is up-to-date. The post-commit hook, unlike the other hooks, will be invoke manually by the server after a commit in CloudStudio.

- *Pre-receive hook* The pre-receive hook is called before receiving a new revision, meaning somebody pushed into the repository. Since JGit will not call the hooks, the pre-receive event is only triggered when a user using Git instead of the web-based IDE pushes commits over the HTTP connection into the repository.

  The pre-receive hook is only used in the user repositories. It automatically commits any changes. This forces the external source invoking the push to resolve any conflicts; even those with the uncommitted changes. It also enables the post-receive hook to include the new modifications into the working directory without needing to stash the commits which is currently not supported in JGit.

Listing 3.2: Post-receive hook for project id 12 and user id 5

```
1  #!/bin/sh
2  echo "Post-Receive Hook"
3  unset GIT_DIR
4  # Only in a user repository: Reset the working directory
5  git reset --hard
6  git update-server-info
7
8  # Call the PostUpdateHook to invoke the RMI method
9  cd C:/path/to/cloudstudio/.config/hooks/
10 while read oldrev newrev ref
11 do
12   java com.cloudstudio.server.configMngm.hooks.
         PostUpdateHook 12 $oldrev $newrev $ref 5
13 done
```

- *Post-receive hook* The post-receive hook is triggered after the pre-receive hook and the receive event. The hook will be called after the new revisions have been received and only if the push was successful.

  The goal of the post-receive hook is to update the models on the server. Additionally in case of a user repository the working directory needs to be reset to the newest revision as described in line 4 and 5 in listing 3.2. Thanks to the pre-receive hook there will be no uncommitted changes lost when resetting.

To inform the CloudStudio server about the new revisions Java remote method invocation (RMI) is used. The server creates and registers an RMI listener which then again will be called by the post-receive hook. The project id and the user id if it's a user repository will be passed as arguments. Additionally the method takes the new and the old revision identifier as well as the reference which can be used to calculate the branch name as parameters. The server will update the intermediate models by iterating over all commits as described in section 3.4.2.

3.4 Synchronization of the Models

3.4.1 Update by Tags

In most cases the models of the folder structure and the file content can be updated relative to a different revision.

There are four operations used to update the persistent models by tags summarized in the `TaggedModelWrapperInterface`. Except for deletion which has only one parameter, all other methods take two arguments, the source and the destination `TagModel`. It's important to differentiate whether the operation will be applied to all tags with the same user and task or if the `VersionControlStatus` also has to match. This is referred to as exact matching. If not mentioned otherwise, only exactly matching tags are considered in the operation.

- *Copying* will simply duplicate the appearance of the source tag for the destination tag. The destination tag is added if the source tag exists and the destination tag is deleted if the source tag does not exist.
- *Copying passively* is similar to copying. However it will not delete the destination tag if the source tag doesn't exist. It is used when committing.
- *Deleting* will remove any occurrences of the given tag. If the parameter for only exactly matching is enabled, the `VersionControlStatus` of the tag has to match as well in order to be deleted.
- *Merging* is the most complex one of the four methods. The content of the source tag will be merged into the destination tag. To do so, only the source respectively destination tag with the most recent, but committed `VersionControlStatus` is considered. The tags with status `UNCOMMITTED` are ignored, since the version control system will not share uncommitted changes during a merge between two repositories. How the content is

merged depends on the implementation of the interface. The method will only affect the destination tag; the source tag remains unchanged.

The method to copy is for example used when a user is added to the project. He will start with the content of the main repository. To update the models all tags of the main repository are copied with the user's own tag as destination. Copying passively is used during a commit. For example when committing manually the tags with `VersionControlStatus.UNCOMMITTED` will be copied passively to `COMMITTED`. The existing `COMMITTED` tags will not be deleted, if there is no `UNCOMMITTED` tag. The method to merge is applied when sharing with the main repository. The committed tags of the user are merged with the tags of the main repository.

In most cases the implementation of the `ConfigManagementSystem` will use the methods of the `TaggedModelWrapperInterface` to keep the models of the file content and the folder structure synchronized with the version-controlled repository. Section 3.4.2 describes situations and the approach if a relative update is not possible.

3.4.2 Update by Iteration over the Commit History

There are cases where the new head revision is not related to the previous commit or the revision of the main repository. It's not possible to update the models by tags as described in the previous section. When only the information of the version control system is available, an update by iteration over the commit history is applied.

Currently only two situations require an update based on the commit history. The first case is when going to a different revision which is not the predecessor of the current one. The second situation is when new modifications are pushed from an external source directly to the Git repository. When switching the revision all commits in between the old and the new head revision have to be undone. The undoing is similar to processing the revisions backwards instead of forward as during the update after a external push.

Both folder structure and file contents are persistent and need to be synchronized to the state of the repositories.

The model of the file content is line-based. As there is no additional information in the VCS about which line has been deleted or which line has been changed, it is best to keep the commits as small as possible to reproduce the actions of the user. Instead of calculating the direct difference between the new and the old head revision the procedure will iterate over each intermediate commit as described in listing 3.3. The iteration of the commit will interpret each commit as a single packaged user interaction. All

changes are integrated into the model and the accumulation of all applied deviations will result in the new head revision.

The folder structure does not require fine granularity as the line-wise file content does. It isn't updated iteratively, but by calculating the direct difference between the new and the older head revision as described in listing 3.4.

Listing 3.3: Java Pseudo Code: Update of the file content

```
1  //Iterate over each intermediate commit
2  for (Commit previousCommit : commitHistory) {
3
4    for (Diff diff : getDiff(previousCommit, nextCommit)) {
5
6      if (diff.getChangeType() == ChangeType.MODIFY) {
7
8        for ( i = 0; i < diffOutput.length; i++ ) {
9          String line = diffOutput[i];
10
11         // Prefix '-': The line has been removed
12         if (line.startsWith("-")) {
13           int a = findNextAddedLine(diffOutput, index);
14           if (a != -1) {
15             cfm.changeLine(rowNr, ownTag, diffOutput[a])
16             diffOutput[a] = null;
17             rowNr++;
18           } else
19             cfm.deleteLine(rowNr, ownTag);
20         }
21
22         // Prefix '+': The line has been added
23         else if (line.startsWith("+"))
24           cfm.insertLine(rowNr, ownTag, line);
25
26         // No prefix
27         else
28           rowNr++;
29       }
30       cfmWrapper.saveModel(cfm);
31     }
32   }
33   nextCommit = previousCommit;
34 }
35 }
```

Listing 3.4: Java Pseudo Code: Update of the folder structure

```
1  // Only consider the diff between source and destination
2  for (DiffEntry diff : getDiff(user, srcCmt, destCmt)) {
3
4    switch (diff.getChangeType()) {
5
6      // New, renamed or copied file
7      case ADD, RENAME, COPY:
8        folderWrapper.addFile(newPath, ownTag);
9
10     // Deleted or renamed file
11     case DELETE, RENAME:
12       folderWrapper.removeFile(oldPath, ownTag);
13
14       if (!folderWrapper.existsSomewhere(oldPath))
15         cfmWrapper.removeCFM(oldPath);
16       else
17         cfmWrapper.deleteTags(oldPath, ownTag);
18
19     // Changed File
20     case MODIFY:
21       if (isEiffelProjectSettingFile(path))
22         triggerReloadFolderStructureEvent();
23   }
24 }
```

Listing 3.3 and 3.4 simplify certain aspects and are written in pseudo code which will not compile. They capture the structure and the approach of the actual implementation. However the implementation may slightly differ from it and the pseudo code neglects some special cases.

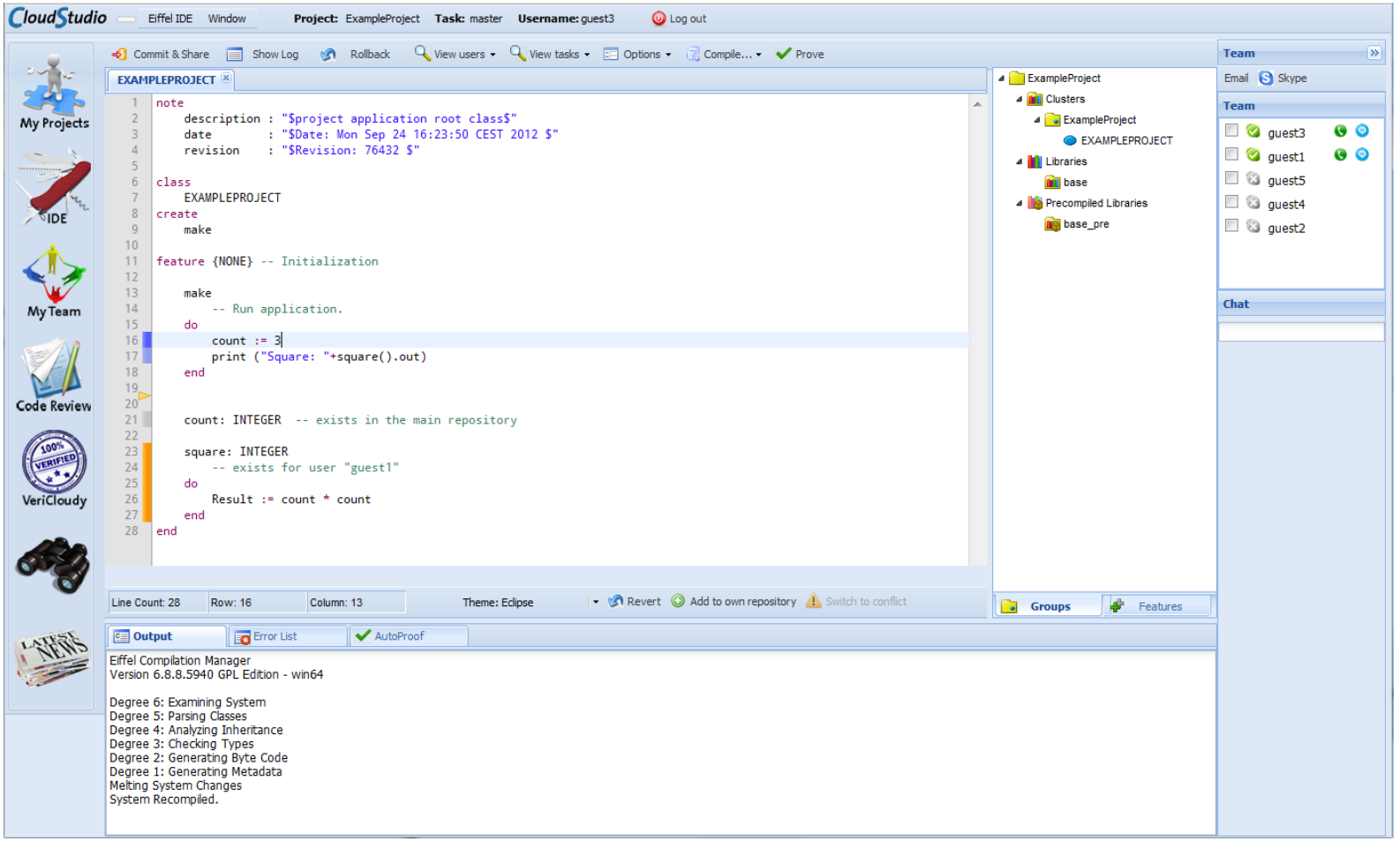


Figure 3.7: Screenshot of the CloudStudio IDE

3.5 CloudStudio IDE

3.5.1 Folder Structure in the Explorer

The explorer of the CloudStudio IDE enables the user to be aware of the changes of his team members. A specific color schema will highlight the added and deleted files as well as the files that don't yet exist in the user's repository. More information on where the file exists can be accessed using the information button in the right-click menu.

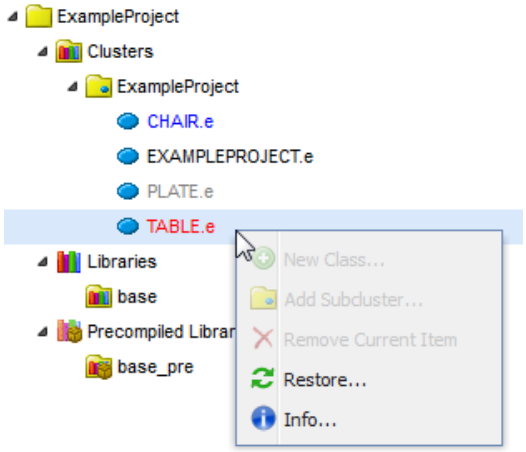


Figure 3.8: Screenshot of the CloudStudio explorer

The color schema consists of four settings with the following meaning:

- *blue* The file has been newly added. It doesn't exist in the main repository, most likely because the user hasn't shared his modifications yet.
- *grey* The file doesn't exist for the user and his current task.
- *red* The file has been deleted by somebody and hasn't been restored.
- *none (black)* The file exists for the user and his current task as well as in the root task of the main repository.

3.5.2 File Content in the Editor

The change awareness of the editor is much more complex than the awareness of the explorer. It enables the user to see the changes of other team members. As shown in figure 3.7 the file content displayed is the merged version of all the users and tasks that are currently viewed. Each line is annotated with

a color to implicitly inform the user about where the line came from. By hovering over the line number additional information about the line and the currently displayed content is displayed.

The user interface of the CloudStudio IDE enables the user to decide how much change awareness he wants. It's possible to select the displayed users and tasks. Per default all users respectively the root task and the user's current task are selected. If the developer selects a further task, it will automatically add any users of the task as well. For a tag to be considered, its task and user must be selected. The shown tags, calculated as the Cartesian product of the selected tasks and users, are saved in the `DisplayOptionsModel`. The content displayed depends on the shown tags.

As described in chapter 3.2.2 each line model has multiple line versions assembled in a tree. One of the leaves of the tree is displayed. By not showing all tags some nodes are ignored because they have no tag that would be displayed as well as no children with shown tags. This will change the leaves of the tree. Accordingly a different line version will be displayed in the editor. To find out whether the line is in conflict all tags are considered to warn the developer about conflicts.

The user is informed about missing lines by an orange arrow as in the example in figure 3.7. A line can be either missing because its displayed content is `null` indicating it has been deleted or there is no line version with a tag that would be displayed. By hovering over the adjacent lines the user can find out for whom the line exists. A line will only be fully removed and no longer indicated by an arrow if it has been deleted completely for every tag.

A color scheme is used to annotate the line in the CloudStudio editor to tell the user whether it has been changed by himself or by another team member:

- *blue* The line has been changed by the user. If the line has been committed, manually or automatically, it changes to *light blue*.
- *orange* The line has been changed by somebody else.
- *dark grey* The line exists for the main repository and the root task. If the line exists for a task different to the root task, it is annotated as *dark grey-blue*.
- *red* The line is in conflict. There are multiple leaves in the line version tree. The first conflicting version, preferably the one of the current user, is displayed.

- *none (grey)* The line exists for the user as well as for the root task and the main repository.

The status bar just below the editor visible in figure 3.7 enables the user to perform additional actions. He can add multiple lines to his own repository. By doing so, his tag will be attached to the line version currently displayed. Another button enables the user to go back to the version of the main repository. In this case, the tag is not just switched to the main version. Most likely the main version is not a leaf and just switching the tag would not result in the main version to be displayed. To enforce that the content of the main version is chosen over the other line versions, a new line version is created. The third status bar button is only enabled when the cursor is on a conflicting line indicated by a red color annotation. It allows the user to switch to a different conflicted line version. By doing so the user can resolve simple conflicts. After switching it's possible to further adapt the line content. This enables further conflict resolution as the new line version will be a child of the conflicted line version and will overwrite it in case of a merger.

3.5.3 Version Control

During the software development the CloudStudio IDE will automatically commit the changes if the user compiled successfully. Whether the new revision should also be automatically shared with the main repository, can be selected in the options. Additionally it's possible to select whether another user sharing with the main repository should trigger an automatic share.

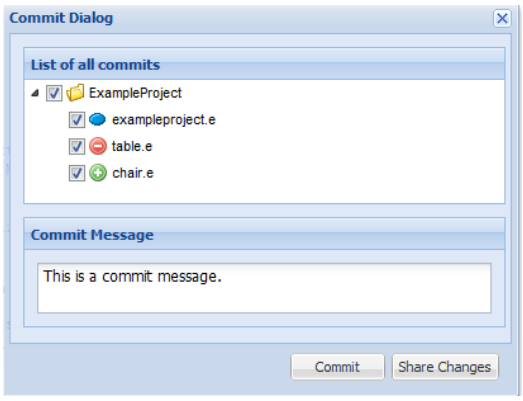


Figure 3.9: Screenshot of the CloudStudio editor

Although this thesis focuses on an automatic version control system, the user still has the ability to manually share or commit. The commit dialog as

shown in figure 3.9 allows the user to specify a commit message and select the modifications he wants to commit. The developer can also look at the commit history using the "Show Log" button, rollback or go to a previous revision.

3.5.4 Compile with Changes

When compiling in CloudStudio the developer can choose whether to just compile his own modifications or whether to include the user and tasks he has currently displayed in the editor. The compilation will only consider committed or compiling changes as not to further introduce compilation errors. The possibility to compile the result of a possible merge with the main repository and the selected users and tasks enables the developer to detect higher-level conflicts. The same functionality can be used when running, testing or proofing a CloudStudio project.

Chapter 4

Evaluation

The automatic version control and change awareness integrated into CloudStudio during this thesis has been evaluated based on different aspects.

The performance in CloudStudio has been investigated. In section 4.2 several typical use cases during collaborative software development have been run through and analyzed.

A case study has been designed and prepared. Due to the time limits of the thesis, the case study was only performed with a very limited number of participants. The task setting has been improved based on the received feedback. Further and more wide-spread trials were not possible due to time constraints, therefore not yielding results mature enough for publication.

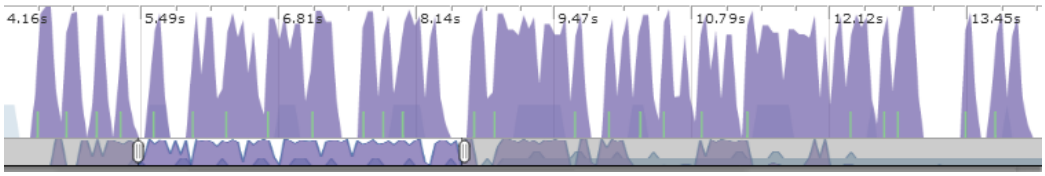
4.1 Performance Analysis & Optimization

The performance analysis focuses on the efficiency and properties of the new editor in the CloudStudio client as well as the communication between the server and the client. In the unoptimized version the complex line-based model of the file content requires the client to calculate the merged content each time the editor is updated which happens on almost every keystroke. The CloudStudio client sends frequent requests to the server. It also continuously keeps a long-polling request open required to enabled the server to push events to the client.

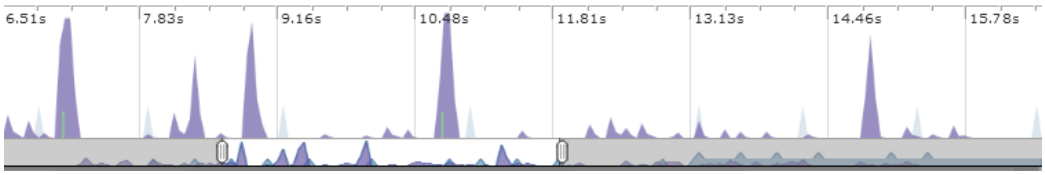
The performance has been evaluated using the Google Chrome browser. The measurements are conducted using the developer tools of Chrome and the Speed Tracer [4] to analyze the browser's performance and the network requests sent to the server. The tests are performed in a new CloudStudio project within the automatically created Eiffel class file. The measuring is started when the file is opened. Since the goal was to investigate the perfor-

mance during development the test setting was to program a simple method with a literal integer as a result value. To simulate common developer activity the code was written without premeditation and with naturally occurring typos in the code which were immediately corrected.

The ACE editor which the CloudStudio editor is built on sends frequent change events. The ACE change event contains a delta with the users modifications. The file content model needs to be updated with this delta. If the file model changes, it's sometimes necessary to reload the change awareness annotations and the content of the editor. The deltas received are small. Even for a user typing fast, it contains most likely only one or two letters.



(a) Unoptimized CloudStudio editor



(b) Optimized CloudStudio editor with minimized reloading on keystroke

Figure 4.1: Performance measured by the Speed Tracer [4]

Captured performance data of the unoptimized CloudStudio editor is shown in figure 4.1(a). The violet filled area indicates the responsiveness of the user interface of the CloudStudio IDE. Typing in the editor results in lags handicapping the developer. The text is only updated and displayed after the user almost finished typing the line, which encourages typos and errors in the code. The lags considerable degraded usability of the IDE.

Analyzing the JavaScript CPU utilization using profiling showed that the browser spent a lot of effort calculating the line version to display. The CloudStudio editor reloads the content and the change awareness after each change event to override the ACE editor's own interpretation of the changes. However reloading of the content theoretically is only necessary when deleting lines and also only when having the option to hide deleted lines disabled. Although during the development of the editor the complete content reload helped to discover any wrong adaption of the model, there is a lot of potential for optimization in minimizing the reloading. The colors and the hover text

of the change awareness have to be updated more frequently, but it can be neglected during editing on a single line. While inserting or removing text on the same line which is already annotated as newly changed by the user, the optimized editor will only update the model and trigger the timer to send aggregated updates to the server.

The performance of the optimized version of the editor measured by the Speed Tracer tool is shown in figure 4.1(b). Compared to the results of the unoptimized version it is noticeable that the browser no longer is overloaded with events to process. The spikes shown in figure 4.1(a) are results of each change event received. In figure 4.1(b) there are only a few spikes. By analyzing the log it is possible to determine their cause. The spikes result from entering new lines. As expected adding a new line will require reloading the colors and hover texts of the change awareness and will result in more calculation than events without any reloading at all.

In both performance measurements in figure 4.1 it's possible to observe the regular calls to the server to update the file model. They are indicated by the light blue filled areas. The updates of the model sent to the server occur as expected more or less each second.

4.2 Analysis of Specific Use Cases

During the second part of the evaluation two use cases with a high level of collaboration are conducted. The results are evaluated based on the ability of CloudStudio to operate in those situations. All use cases first describe the setup and the steps. If not mentioned otherwise the CloudStudio editor of each user applies the default setting which will display content of all other users working on the same task.

4.2.1 Use Case 1

- UserA programs a feature `square` in the main class of the project.
- UserB calls the feature `square` and compiles with the changes of UserA.
- UserB adds the feature `square` to his own repository and compiles without the changes of others.
- UserA notices a bug and fixes it.

During UserA programming the feature, UserB can instantly observe his progress in the CloudStudio editor. For UserA the feature `square` is annotated

with blue. UserB's editor displays the lines with the change awareness color orange. UserB includes a call to the newly programed `square` feature. When compiling with the changes of UserA no errors occur. UserB adds the feature to his own code by using the status bar button "Add to own repository". The lines change from orange to blue. For UserA the color of the lines remains unchanged. If UserB compiles without the changes of others no errors are introduced by the call to `square`. UserA changes a line in the feature, probably to fix a bug. UserB will see the changed line highlighted in orange. When compiling with the changes of UserA the bug fix will apply.

The users benefit from the advantages of having a version tree for each line. When UserB adds the lines to his repository, he will switch to the same version as UserA. If UserA changes a line, this will not introduce a conflict, but be recognized by the editor as a successor of UserB's version.

4.2.2 Use Case 2

- UserA alters a line.
- UserB also changes the same line.
- UserA also modifies the line.

When UserA changes the line, its change awareness color will change to blue for UserA and orange for UserB. UserB can see the changes of UserA and can alter them. If UserB changes the line, the new line version is displayed to UserA in orange. Again UserA changes the content of the line, which will switch the color back to blue for UserA respectively orange for UserB.

With full change awareness enabled no conflicts are introduced during collaborative activities like pair programming. This use case is an example for that. UserB can change the same line without creating a conflict because the latest version of UserA is displayed. The new line version will be a child of the displayed version and overrule it. However it will only work if both UserA and UserB can see each others changes using the change awareness of the CloudStudio editor.

4.3 Case Study

The case study is designed to analyze the usability of CloudStudo as well as the advantages of the new automatic version control system and change awareness. During this thesis the case study has only been performed on a

trial basis to refine the task setting and further improve the integration of the new backend into CloudStudio.

The case study can be performed in four settings with a team of two developers working on the same task set:

1. CloudStudio: Fully automatized version control with AutoShare
2. CloudStudio: Partially automatized version control without AutoShare
3. EiffelStudio with the batch script: Partially automatized version control
4. EiffelStudio with Git: Standard version control

Depending on the setting the participants either use CloudStudio, the web-based IDE with change awareness, or Git and EiffelStudio, the IDE for Eiffel. The options of the CloudStudio IDE allows to further influence the level of automation of the version control by enabling or disabling the automatic share. Also when using EiffelStudio the programming experience can be changed by using the batch script provided by CloudStudio to simplify the use of Git.

The tasks designed for the case study require a lot of collaboration and coordination. To be able to fully evaluate the communication the participants are only allowed to use Skype to communicate. The CloudStudio server will log events like commit or share as well as the automatic version control operations. Further the log will highlight if a conflict has been detected in the model of the file content.

The questionnaire of the case study will help to analyze the usability of automatic version control and change awareness. It also gives the participant the chance to post feedback and suggestions.

To improve the task setting and check if the backend integrated during this thesis is ready for productive deployment, the case study has been performed in a small trial. The performance issues described in section 4.1 proofed to considerable degrade usability and had to be solved first. It further was noticed that the usability of CloudStudio is strongly influenced by the absence of tools like replace or automatic completion.

The task description and the questionnaire can be found in appendix A.

Chapter 5

Conclusion & Future Work

5.1 Conclusion

This thesis introduces an automatic version control system with proactive conflict detection and resolution based on change awareness and integrates it into CloudStudio, a web-based IDE. Using change awareness the developer sees the modifications of his team members in the editor in real-time. The chosen line-based model of the file content and the version tree for each line allow a detailed apprehension of the line versions and their relation. No false-positive conflicts are introduced when multiple users edit the same line while being aware of the changes of the others. Furthermore real conflicts are highlighted in the CloudStudio editor and only occur if the user ignores the change awareness annotations. The integration in CloudStudio also allows the detection of higher-order compilation conflicts by compiling with committed changes of the main repository and other users.

The line-based model is optimized for text strongly structured by line breaks. This is the case for the source code of most programming languages. However the editor is not ideal for standard text editing, especially if the line breaks are frequently rearranged. Compared to a syntax-aware file model the advantages of the line-based approach overweight the disadvantages. The code is not required to be able to compile to construct the model of the file content. The line-based editor is implemented completely independent of the language and the compiler.

The automatic version control system introduces by this thesis works without user interaction. Although the manual functionality of a VCS is still available, the developer can work in CloudStudio without performing a single version control activity. To achieve this any conflicts occurring will be resolved automatically. The CloudStudio configuration management does not

offer any additional tool to resolve conflicts. Although it would be possible to implement such a tool, it is not necessary. The change-aware editor shows the user the result of the conflict resolution and offers multiple possibilities to influence it. CloudStudio continuously helps the user to anticipate conflicts and resolve them on the fly.

It's also possible to use CloudStudio projects without using the web-based IDE. The repositories can be cloned using Git. CloudStudio even offers a script to simplify the process. When pushing new revisions from the outside, the CloudStudio server will update the models automatically with as much accuracy as possible.

5.2 Future work

The functionality integrated by this thesis can be easily extended. It is possible to further enhance the automatic version control system of CloudStudio and even widen its use to another IDE like EiffelStudio. Some of those possibilities are explained below.

5.2.1 Improvements of the CloudStudio IDE

A highly recommended extension of the CloudStudio IDE is to adapt the commit dialog to enable adding a message to a previous automatic commit. By being able to retroactively replace the generated message of the automatic commit, one of the few drawbacks of having an automatic version control system can be resolved. This could be implemented using the rebase command of Git to rewrite the commit history.

CloudStudio already enables the developer to detect compilation conflicts by compiling with the changes of others. It would be possible to further enhance the detection of higher-level conflicts as well as to automatize it. For example the project could be automatically tested after successful compilation using AutoTest [30, 20] while including the modifications of the other team members.

Currently the CloudStudio editor does not specifically support copying of lines. Copying may result in conflicts if a user edits the deleted lines. The model of the file content allows further optimization by rearranging the order of the lines instead of deleting and then adding them somewhere else. As rearranging lines can frequently occur in object-oriented programming, it may be a valuable feature although the copy&paste functionality is hard to adapt in the browser.

The choice of a line-based approach over a syntax-aware solution has many advantages. But there is also a lot of potential to integrate syntax awareness. Using change awareness the feature list and auto-completion can be complemented with the new features of other users.

5.2.2 EiffelStudio Plugin

As part of the thesis a simple batch script to export the project using Git has been added enabling the developer to use EiffelStudio or another IDE. To further extend the ability to work on the own IDE, EiffelStudio could be extended with a Git plugin. The plugin would simplify the use of Git for the developers and could also include some automation of the version control system.

Improving the support of working with EiffelStudio even further, it would be possible to export the models of the file content to enhance the plugin with functionality equal to the CloudStudio editor concerning conflict detection.

5.2.3 Version-controlled File Content Model

Currently the models of the file content are persisted but not version-controlled. This introduces a lack of information when switching to a different revision as stated in section 3.4.2. A solution to this problem would be to put the intermediate models under version control. This approach could further be useful for an EiffelStudio plugin working with the copy of a CloudStudio repository. However the models have to be treated carefully and merged correctly when a new revision is push into the repository.

Bibliography

[1] AceGWT - An integration of the Ajax.org Code Editor (ACE) into GWT. http://github.com/daveho/AceGWT.

[2] Ajax.org Cloud9 Editor. http://ace.ajax.org/.

[3] Google Web Toolkit (GWT). http://developers.google.com/web-toolkit/.

[4] Google Web Toolkit (GWT) - Speed Tracer. http://developers.google.com/web-toolkit/speedtracer/.

[5] JGit. http://eclipse.org/jgit/.

[6] Analysis of Git and Mercurial. http://code.google.com/p/support/wiki/DVCSAnalysis, Sept. 2012.

[7] Wikipedia - Git (Software). http://en.wikipedia.org/wiki/Git_(software), Sept. 2012.

[8] J. Biehl, M. Czerwinski, G. Smith, and G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 1313–1322, New York, NY, USA, 2007. ACM.

[9] Y. Brun, R. Holmes, M. Ernst, and D. Notkin. Speculative identification of merge conflicts and non-conflicts. Technical Report UW-CSE-10-03-01, University of Washington, 2010.

[10] Y. Brun, R. Holmes, M. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. *ESEC FSE, Szeged, Hungary*, 2011.

[11] E. Carmel. *Global software teams: collaborating across borders and time zones*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[12] E. Carmel and R. Agarwal. Tactical approaches for alleviating distance in global software development. *IEEE Softw.*, 18:22–29, March 2001.

[13] S. Chacon. Pro Git. `http://git-scm.com/book`.

[14] J. A. Espinosa, N. Nan, and E. Carmel. Do gradations of time zone separation make a difference in performance? A first laboratory study. In *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2007)*, pages 12–22. IEEE, Aug. 2007.

[15] H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer, and J. Schneider. Agile vs. structured distributed software development: A case study. In *Proceedings of the 7th International Conference on Global Software Engineering*. IEEE, 2012.

[16] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 235–238. ACM Press, 2010.

[17] L. Hattori, M. Lanza, and M. D'Ambros. A qualitative analysis of preemptive conflict detection. Technical Report 2011/05, University of Lugano, Sept. 2011.

[18] R. Hegde and P. Dewan. Connecting programming environments to support ad-hoc collaboration. In *Proceedings of the Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 178–187, 2008.

[19] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Introducing collaboration into an application development environment. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 21–24. ACM, 2004.

[20] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[21] M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In J. Vitek, editor, *TOOLS-EUROPE*, LNCS. Springer-Verlag, 2010.

[22] M. Nordio, H.-C. Estler, C. A. Furia, and B. Meyer. Collaborative software development on the web, 2011. arXiv:1105.0768v3.

[23] M. Nordio, H.-C. Estler, B. Meyer, J. Tschannen, C. Ghezzi, and E. D. Nitto. How do distribution and time zones affect software development? A case study on communication. In *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2011)*. IEEE, 2011.

[24] M. Nordio, C. Ghezzi, B. Meyer, E. D. Nitto, G. Tamburrelli, J. Tschannen, N. Aguirre, and V. Kulkarni. Teaching software engineering using globally distributed projects: the DOSE course. In *Collaborative Teaching of Globally Distributed Software Development - Community Building Workshop (CTGDSD)*, New York, USA, 2011. ACM.

[25] M. Nordio, R. Mitin, and B. Meyer. Advanced hands-on training for distributed and outsourced software engineering. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 555–558. IEEE, 2010.

[26] M. Nordio, R. Mitin, B. Meyer, C. Ghezzi, E. D. Nitto, and G. Tamburrelli. The role of contracts in distributed development. In *Proceedings of Software Engineering Approaches for Offshore and Outsourced Development*, 2009.

[27] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Code-based automated program fixing. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 392–395. IEEE, 2011.

[28] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM '11. Springer, 2011.

[29] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Verifying Eiffel Programs with Boogie. In *First International Workshop on Intermediate Verification Languages (BOOGIE 2011)*, 2011.

[30] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer. Stateful Testing: Finding more errors in code and contracts. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 440–443. IEEE, 2011.

Appendix A

Appendix: Case Study

TASK DESCRIPTION

WHAT YOU NEED TO KNOW

The tasks are performed on an existing project containing several classes modelling a library. A short description will help you get started. Your team partner received the same task sheet. Together you try to complete the tasks. Since CloudStudio offers no possibility to run the program, it's enough if you solved the task and it compiles.

The `LIBRARY` has a lot of books. Everybody can register for a `LIBRARY_CARD` which is needed to borrow an item. To do this, you can talk to a `LIBRARIAN`. With your `LIBRARY_CARD` you can reserve a book using one of the four `LIBRARY_PC`s. Once reserved the book can be picked up.

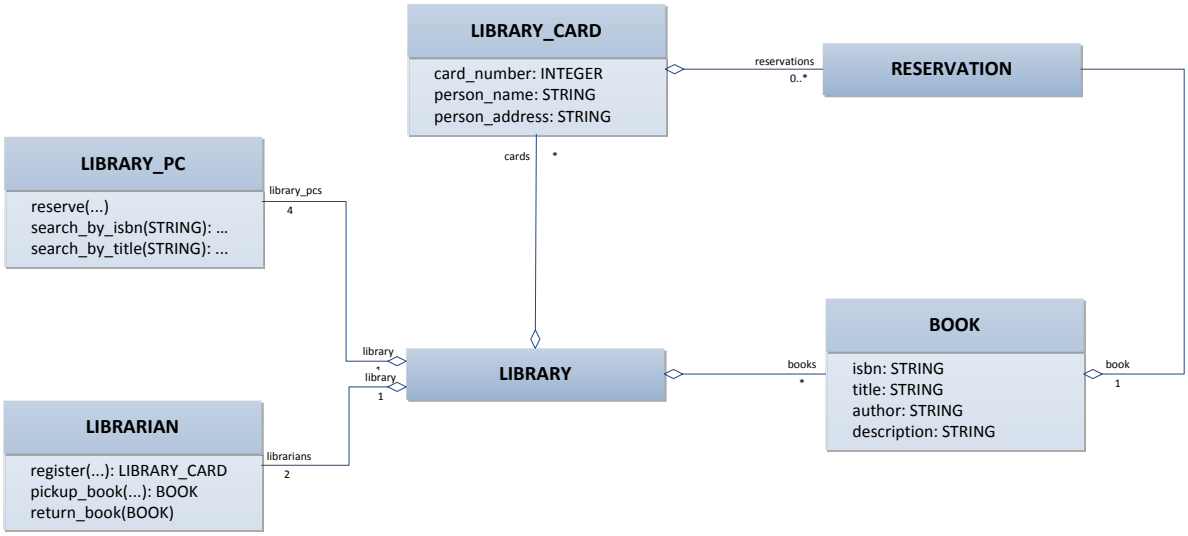


FIGURE 1 **CLASS DIAGRAM**

WHAT YOU NEED TO DO

TASK 1 - EXTENDING LIBRARY

The library plans to extend their selection of books with DVDs and games.

- Create a super class `ITEM`.
- A `DVD` has a title and a description as well as an actor. It possible to search for actors on the `LIBRARY_PC`.
- A `GAME` has a title as well and a description.

TASK 2 - REFACTORING

Refactor the existing code.

- Check the contracts and class invariants
- Use the super class `ITEM` where it makes sense.

QUESTIONNAIRE

Thank you for participating in the case study. Please answer the following questions and help us to further evaluate CloudStudio.

User Name: _____

Project Name: _____

EXPERIENCE

I'm experienced in developing with **Eiffel.**

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

I'm experienced in using distributed version control system like **Git or Mercurial.**

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

I'm experienced in using centralized version control system like **SVN.**

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

CLOUDSTUDIO

You can skip this section, if you didn't use CloudStudio.

What properties apply for the **change awareness** (seeing the changes of your partner in the editor)?

☐ helpful ☐ great ☐ confusing ☐ I ignored it

Which **feature of the change awareness** did you notice and find useful during programming?

☐ *hover text* (over line numbers)

☐ *colour in the explorer* marking the files

☐ *colour in the editor* marking the lines

What properties apply for the **automatic version control** (automatic committing and automatic sharing)?

☐ helpful ☐ great ☐ confusing ☐ I ignored it

Did **conflicts occur** during the case study?

○ none ○ sometimes ○ often ○ all the time

If so, how would you **describe the conflicts**?

☐ textual ☐ compiling (*) ☐ CloudStudio didn't behave as expected

☐ others:

(*) Compiling conflicts means the compilation fails when compiling with changes of your partner or when compiling after sharing.

How did you **resolve the conflicts**?

☐ m*anually* - editing the line in the editor yourself

☐ *automatic* - the conflicts resolved themselves

☐ by comparing to the *conflicted version* (hovering over the line number)

☐ by *deleting* the line

☐ by using the *status bar buttons* (revert, add to own repo. and switch to conflict)

COMMENTS

If you have any comments or suggestions, you can tell us now.

Send as Mail