Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Integration of Version Control Systems in a Tool for Distributed Pair Programming

Andreas Haferburg

Matrikelnummer: 3601910

andreas.haferburg@fu-berlin.de

Betreuer: Karl Beecher

Eingereicht bei: Prof. Dr. Lutz Prechelt

Berlin, 18. November 2010

Abstract

Saros is an Eclipse plug-in for distributed pair and party programming. A much requested feature for Saros was the integration of version control systems (VCSs).

This thesis explores the possiblities of integrating VCSs into the feature sets of Saros. The initial implementation enables users to automatically replicate the most common VCS operations like update or switch. When one user performs such an operation, these actions can be automatically distributed to the other participants of the session, thus extending the preexisting synchronization mechanism of Saros from only the file contents to the underlying state of the VCS.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 18. November 2010

Andreas Haferburg

Contents

Introduction

Chapter 1 explains some basic terms, laying out the groundwork for this thesis. Chapter 2 describes the list of requirements, and how they were determined. Chapter 3 presents a number of global design decisions, while chapters 4 and 5 describe the design and implementation in more detail. The last chapter evaluates what was done, and how it could be extended further.

Chapter 1

Fundamentals

1.1 Pair Programming

Pair programming is the practice of agile software development where two programmers team up for a software development task, sharing the same computer. The programmer operating the mouse and keyboard is called the *driver*, while the other person is called the *navigator* or *observer*. While the driver has to focus on what is being typed at a given moment, the observer can sit back and consider the bigger picture. It is encouraged that the two participants switch roles after a period of time.

[CW01] lists a number of advantages of pair programming over the conventional way:

Continuous code reviews Pair programming can be regarded as an extreme form of peer reviews: The driver is continuously reviewed by the observer, which means that many mistakes get spotted as early as possible, lowering the defect rate.

Design quality The constant dialog leads to better designs, and shorter, more maintainable code.

Learning By working closely together, programmers learn from each other. When regularly switching teams, the knowledge of each individual programmer can spread to the entire team. A side effect is a lowered project risk: Staff-loss is less threatening to project success if multiple people are familar with each piece of the code.

Problem solving Bouncing ideas off of each others speeds up the process of solving problems.

Satisfaction People enjoy work more.

Distributed pair programming or *remote pair programming* is a variant of pair programming where the participants are in different physical locations, while using a software tool to connect them over a network, typically the internet. There are different levels of tool support for distributed pair programming. On the most basic end are screen-sharing applications, which have the disadvantage of providing no interaction while requiring high bandwidth. More interaction is possible with remote desktop applications like VNC. There are also collaborative real-time editors, which enable editing the same file at the same time over a network connection, and plug-ins for IDEs[1] which add facilities to support pair programming to an existing application.

1.2 Eclipse

Eclipse is an IDE for Java software development. The framework is extensible through a plugin system. A large number of plugins exists, providing support for other programming languages, including C, C++, Perl, PHP, Python, and Ruby.

[1] Integrated development environment, a software application for software development

1.3 Saros

Saros is a plug-in for the Eclipse IDE.[Dje06] Its core feature is to support pair programming by replicating a project remotely and in real-time.

To initiate a session, a user selects a project, and invites other people to the session. The user initiating the session is called the host, who is also the only participant able to invite others. After accepting the invitation, the invitee can either choose to use an existing project, or to create a new copy of the incoming project. The host automatically sends a list of all the files in the project and their checksums to the invitee. If the invitee used an existing project, the checksums of the existing files are compared to the files of the host, and only the files that are not already present are requested. The host will then create and send a ZIP archive of the requested files to the invitee.

Saros supports the roles used in pair programming, despite the fact that typically all the participants have their own keyboard. By default, the host is the exclusive driver, and any invitee starts out as an observer. Only drivers are allowed to make any changes on the shared project. The host has the ability to reassign roles. Any participant of the session can be assigned the role of driver, observer, or exclusive driver. The latter command will automatically demote all other drivers to observers.

Any modification of a file in the shared project is replicated in real-time. When a driver edits a file, Saros detects the location of the change and the characters added or removed, and automatically applies the same changes in the editors of all other participants.

Other features include awareness information, follow mode, chat, and VoIP, the details of which don't need further description here since they are irrelevant for this thesis.

1.4 Version Control Systems

Version Control is the management of changes to files. Every version of a file over the course of its history is identified by the *revision number*, *revision id*, or simply *revision*.

Version control is an integral part of today's software development process. Features like automatically merging different changes of the same file are an important tool for any project where multiple developers work concurrently on the same code base.

When managing a set of files with a version control system (VCS), the user starts by creating an empty repository on a server. The files are then *imported* into the repository to create the first revision. The local copy of the managed files is called the *working copy*. If a project already exists in a repository, the user can perform a *check-out* in order to create a working copy from the repository.

After changing files, the user can *commit* these changes to copy the new versions of the files from the working copy to the repository, thereby creating new revisions. Other users can copy these new versions from the repository to their working copy by performing an *update*. A list of all the commits and their authors is available, providing information on who changed which part of the code base. The commit log can also be used as a summary of all the changes that were made during releases.

An alternative way of sharing changes is to create a *patch* instead of committing. A patch is a simple text file created by the VCS that contains a set of changes. Patches can then be handed to other developers, for example as an e-mail attachment, who can apply the patch in order to replicate the changes it contains in their working copy. This technique is useful when the author of the changes doesn't have commit rights, or for pre-commit reviews.

When creating a new file, the user can choose to *promote* it in order to add

it to version control, or to have the file ignored by the VCS. If the file was promoted, it will be copied into the repository with the next commit, and have its first revision number assigned to it.

The *base version* of a file is the version which was checked out from the repository, and the most recent commit in the repository is called the *head revision*. Typically, a user checks out the head revision, so initially the working copy and the base version are both identical to the head. When the user makes local changes to the working copy, the base version can be used to determine exactly what changed locally. If another user commits changes to the repository, the revisions of the base version can be used to determine what changed in the repository.

There are different approaches to handle the problems which arise from concurrent access to the same resource. Some VCSs only support file locking, which means that when a developer checks out a file, no one else has write access to the file until that developer commits the new version or releases the lock.

However, most modern VCSs allow multiple developers to edit the same file, and provide facilities to automatically *merge* (or *integrate*) concurrent changes. When multiple users modify the same file, the first user to commit will always succeed. However, a commit is rejected if the VCS determines that the base version of the committer differs from the latest version in the repository. In this case, the committer is required to first update to the head in order to merge changes in the repository into the working copy. This prevents users from unintentionally overwriting changes from others.

A merge is the application of two different sets of changes (or *diffs*) to the same file. Typically, merging text files is automated by the VCS. By comparing two versions of a file, the VCS determines which lines were added, removed or modified. Multiple changes are then transformed against each other in order to create the merge result. For example, if one diff adds two

new lines after line number 4, and another diff replaces the lines 7 and 9, applying the second diff after the first requires the line numbers of the second diff to be incremented by two (because of the two new lines).

Sometimes the VCS is unable to perform the merge automatically because of a *conflict*, e.g. when the same line was replaced by different lines. Another common example is when the file type is not supported by the VCS, for example when different changes to a binary file like an image have to be merged. In these cases, the user has to manually *resolve* the conflict.

Most VCSs also support *branching*, enabling users to track multiple branches of development in parallel. The *trunk* or mainline is the main version of development. Users can create a new branch from either the trunk or another branch to fork off a new line of development. A developer can e.g. work on an experimental feature in a separate branch, without the risk of destabilizing the main line of development. Another common practice is to create a new branch for every new release of a software. While the addition of new features happens mostly in the trunk, the release branch is still available for bug fixing to quickly create intermediate releases, completely independent of the main development. The VCS typically provides a way to *switch* the working copy to a different branch. Any subsequent commit will then only change the current branch in the repository. A branch can be merged back into the trunk or any other branch, which means that the changes from the branch will be integrated with the changes that were made to the other branch in the meantime.

1.4.1 Subversion

Development of Subversion started in 2000, with the goal of replacing and improving the most widely used VCS at the time, Concurrent Versions System (CVS).

One feature of Subversion is the fact that revision numbers are repository-

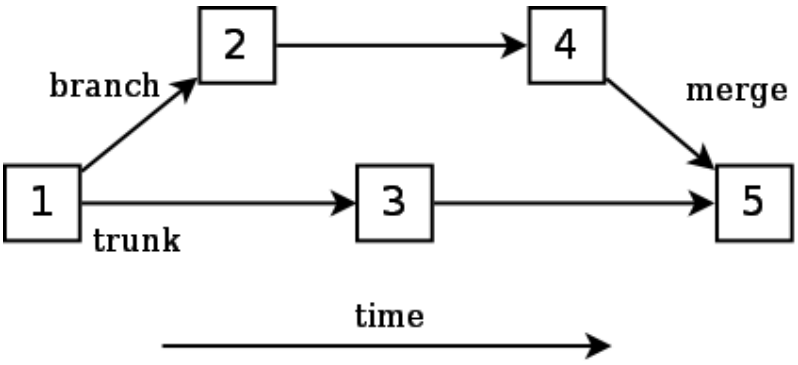

Figure 1.1: Branching and merging

wide. After each successful commit, every resource in the repository is assigned a new revision number, as opposed to only the files that were changed. This approach to revision numbering has the advantage that it's very easy to identify and refer to versions of the entire project.

Another feature are atomic commits. Some older VCSs like CVS or Microsoft SourceSafe exhibited low tolerance to network or hardware failure during commit operations. If a commit operation failed, it was possible to leave the repository in an undefined state, or even lose data. The term atomic is used in the sense of database transactions, where a rollback is performed after a failed transaction to recreate the state before the commit attempt.

In Subversion, the branch operation is implemented by simply using the copy operation, while relying on a naming convention. The main development branch is located in the directory `/trunk`, while branches are located in subdirectories of `/branches`. In order to create a new branch from the trunk called "test", the `/trunk` directory is simply copied to `/branches/test` using the svn copy command. The copy command will not actually duplicate data, it only creates directory entries that point to an existing tree, similar to hard links in the Unix world[PCSF08]. Only the changes commited to the branch will require new space. This means that the creation of branches (or any copy) is an operation constant in both time and space.

There are two plug-ins designed to integrate Subversion into Eclipse: Subclipse and Subversive. Since Subclipse is used for the Saros project, it was the natural first candidate for integration into Saros.

1.5 The Eclipse Team API

The Team API defines a collection of interfaces that clients can implement to add support for a version control system to Eclipse. Following Eclipse terminology, these plugins will be called *Team providers*.

One of the features provided by the Team API is the association of a Team provider with a project. There is at most one provider associated with a project at a given time, and the association is always for the entire project, not just a sub-directory. A project that has a Team provider is called *managed*.

Another feature is a callback mechanism that enables the Team provider to provide decorations for the labels of files and folders, such that VCS specific information can be displayed in other parts of Eclipse, like the revision number, the current branch, or the name of the author that committed the latest version.

It should be noted here that the interfaces are not designed to access the functionality of the VCS itself, but only to provide the necessary means of communication between the plugin and Eclipse to integrate the VCS into Eclipse. For example, while the Team API provides a method to update a resource to the latest revision, there is no provider-independent way to update to an earlier revision, let alone switch it to another branch. For this reason, the Team API is only of very limited use to this work, and most of the time, direct interaction with concrete Team providers is necessary. Unfortunately this means that access to the source code (or detailed technical documentation) of the Team provider is required to add support for the

corresponding VCS. For open source projects this is not a problem, but Saros will probably never support proprietary software like Perforce.

Chapter 2

Requirements Analysis

Saros was completely unaware of version control systems. In fact, one of the implicit assumptions in Saros was that the only way to modify a file is by using an editor in Eclipse. But there are at least two other ways: External editors and other plug-ins, most notably Team providers.

A Team provider is merely an interface to a version control system. To determine the functional requirements, the functionality provided by the version control system has to be inspected. In the context of VCSs, a file has two additional attributes: The *resource URL*, which is the location of the resource within the repository, and the *revision number*. The resource URL specifies not only the path of the resource within the project, but also the branch the resource is located in. Both attributes are assumed to be represented as Strings. In the following, we will call the values of these attributes the *Version Control state* of the resource, or shortly the *VC state*. The VC state uniquely identifies the base version of the file, which in turn determines the outcome of almost all the operations of the version control system. For example, to determine the local changes of files, the version control system compares the working copy to the base version. When performing code reviews, this diff is the starting point for developers. Distributed code reviews

using the VCS are only possible if all the participants of a session have the same working copy *and* base version.

Version control systems have only a small number of operations that can alter the base version of resources: Checkout, add, remove, copy, commit, update, and switch. To synchronize the state of the project between all participants in a session, Saros must be able to replicate the VC state of all resources, thereby replicating the base version of these resources. Only if the base version of the project is replicated, the outcome of certain VCS operations like e.g. a merge will be the same across all participants.

Other functionality of version control systems like merge, revert, or apply patch only alter the file contents. Assuming that file content modifications are replicated anyways, these operations can safely be ignored, because the source of these modifications (an external program as opposed to the user) doesn't matter. Since the operations branch, and tag only modify the state of the repository, they can also be ignored.

2.1 Functional Requirements

1. Synchronization of all file content modifications.

   Whenever the content of a file in a shared project is modified, the content changes must always be replicated. This includes the modifications of files by Eclipse editors, plug-ins, and outside editors.

2. Synchronization of the VC state during the invitation.

   After the invitation process is finished, the invitee's copy of the project has the same resources as the host's copy. In addition to the relative paths and file contents, the VC state must be replicated for every file and folder in the project.

3. Replication of Connect/Disconnect operations to/from an VCS repos-

itory.

In Eclipse, a user can disconnect a project under VC from the repository. Replication means that when a driver disconnects the project, the software must disconnect the local copies of the project of every other participants.

4. Replication of update and switch operations.

   Update means that the revision number changes, but not the resource URL, while switch means that the resource URL changes. If a driver updates a resource in a shared project, the same operation must be executed automatically by the software on the local copies of the other participants.

   Note that technically, an SVN update operation could also be executed with an SVN switch operation by using the same resource URL. However, not all VCSs have a switch operation, and it might be confusing to a user if Saros performs a switch to replicate an update of a driver. For this reason, an update should always be replicated by an update.

5. Replication of promoting and ignoring a file.

   When a driver promotes a file in order to add it to the VCS, the corresponding copy of the file on the other participants must also be promoted automatically.

6. Dependencies on Team providers are optional.

   Providing support for additional VCSs will introduce new dependencies, either on a Team provider plug-in, or on some other interface to the VCS. The Saros plug-in must be able to run even if these dependencies are not satisfied. When trying to access a missing dependency, the software must handle any errors (e.g. a ClassNotFoundException) gracefully, and provide information to the user on how to resolve the problem.

7. Access to the repository is optional.

   There must be a fall back mechanism if a participant can't access the repository. Possible reasons includes wrong or missing credentials, server failure, or network problems.

   The requirements 2-5 don't have to be met when there is no access to the repository. In this case, the other requirements of Saros still have to be met, especially the replication of file contents.

8. Support for VCSs other than SVN, and Team providers other than Subclipse.

   Additional candidates are Mercurial, git, and CVS. Choosing a Team provider is mostly personal preference, so support for Subversive in addition to Subclipse would be desirable.

2.2 Non-Functional Requirements

9. The replication of VCS operations doesn't block the user interface.

   During the execution of a VCS operation it should still be possible to read all the shared files. It should also be possible to have write access to resources which don't conflict with the modified resources, for example resources of another project, or resources that aren't under version control.

10. VCS operations are replicated as soon as they are started.

    When a driver executes a VCS operation, the safest way to handle it is to wait until the operation has finished, and only then issue the replication of the operation. This means that all the other participants would have to wait idly until the driver is done. The reason for waiting is that the operation might fail, or that the driver cancels it, hence we

might call this way of handling VCS operations pessimistic.

A more optimistic approach would be to assume that the operation is going to finish successfully, and send out requests for replication as soon as the operation has started. This means that all the participants would execute te VCS concurrently, which is most efficient. However, it also requires us to think about handling the cases where the operation doesn't finish. If, for example, the host cancels the operation, the peers would also need to cancel it, or roll back the changes, possibly only in part.

A more subtle problem is a race condition when operating on the latest version (head). If e.g. the driver instructs the VCS to update to the head version, and someone else commits a newer version before the peers execute their update commands, the peer would update to a different version. The problem here is that 'latest version' can change its value, and that this value is unknown to the driver at the time when the command is sent to the server. In this (admittedly improbable) case, the peers that replicated the operation need to ascertain that the operation executed was in fact the operation intended by the driver.

Chapter 3

Solution Design

While the next two chapters focus specifically on the design and implementation of the replication of the VC state, this chapter describes some additional design decisions that guided the implementation task.

As a side note, it should be noted that the implementation of the first requirement (synchronization of all file content modifications) was almost trivial. The preexisting `SharedResourceManager` class is responsible for the detection and replication of modifications to the files of a project. It sends messages in the form of `FileActivity`'s to the other participants, for example if a driver renames or creates a new file. The `SharedResourceManager` class already detected the file modifications from external sources, and the only change required was to send a FileActivity in this case instead of ignoring it.

3.1 Milestones

To break down the development process into manageable parts, several milestones were identified, ordered by dependency, importance, and the expected effort.

1. Synchronization of the VC state during the invitation.
2. Detection of resource URL changes (switch).
3. Replication of switch operations.
4. Detection of connecting to/disconnecting from repository.
5. Replication of connect/disconnect operations.
6. Detection of revisions changes (update).
7. Replication of update operations.
8. Use of SVN console to replicate operations immediately.
9. Fallback mechanism.
10. Replication non-blocking.

Due to time constraints, only the first seven milestones were completed.

3.2 Optional dependency on Team providers

Saros must not have any hard dependency on a Team provider. If usage of a Team provider is requested, the software must still work properly if the dependency is not present. The dependency on a Team provider has two aspects: The plug-in dependency and the class dependencies.

Every Eclipse plug-in has a list of plug-in dependencies, which consists of the identifiers of all the other plug-ins that this plug-in depends on. Before loading a plug-in, Eclipse verifies that all the dependencies are installed and have been loaded successfully. The Eclipse framework also offers the possibility to mark a plug-in dependency as optional, which means that the dependent plug-in will be loaded even if the dependency cannot be resolved.

On the class level, what must be avoided is an uncaught `ClassNotFoundEx-`

`ception` thrown by the Java class loader when trying to access the classes of a Team provider unsuccessfully. This is achieved by using the adapter pattern. For every Team provider there is an adapter class, and the only way to access it is through an abstract interface. For example, the class SubclipseAdapter is the only place in the code that accesses the Subclipse plugin. Every interaction with the Team provider, be it requests for information or operations, is encapsulated by the corresponding adapter class.

Each adapter inherits from the abstract class `VCSAdapter`, which is the only public interface of the VCS module. This base class uses the common Team API, for example to determine the revision of a resource. More importantly, it also has the static method `getAdapter`, which is the only way to instantiate a concrete `VCSAdapter`. This is enforced by setting the visibility of the concrete adapter classes to package private, such that only the class `VCSAdapter` can access them.

When trying to instantiate a concrete adapter class which depends on a missing dependency, the class loader will throw a `ClassNotFoundException`. In this case, the `getAdapter` method will simply catch the exception and return `null`. Since the adapter class is the only place in the code accessing the Team provider, it is safe to assume that this exception can only be thrown during the execution of the `getAdapter` method.

3.3 User Interface

Before the implementation of a fallback mechanism (milestone 9), there are two possible modes of operation: Either Saros uses VCS operations whenever possible, which requires every participant to be able to execute these operations successfully, or Saros completely ignores VCS operations. To select the mode, the host has to choose between “Share project...” and “Share project with VCS support” when starting a session.

If the first of these options is selected, Saros behaves just as it used to before the integration of VCSs. If the latter option is selected, Saros tries to replicate VCS operations. However, if one of the participants is unable to execute a VCS operation for whatever reason, it is possible that the project enters an inconsistent state. In this case, the participant would see an error message, but the host would not be automatically informed that something went wrong, even though it's the host's responsibility to select the proper mode of operation.

Once a proper fallback mechanism is in place, there is no more need for the user to select a mode of operation. VCS operations would be used if possible, and if any problem occured while trying to replicate an operation, Saros could automatically fall back to the old way of using file and folder activities.

3.4 Other requirements

The integration of version control systems in Saros consists of the replication of the VC state of the shared resources. The replication of state can be broken down further into the two parts initialization and updates. The next two chapters describe the design and implementation of these two parts.

Chapter 4

Synchronization of the VC state during the invitation

We will first take a closer look at the existing invitation process, the changes that needed to be made to it, and how the requirements were finally implemented.

During the invitation, the FileList class is used to determine the current state of the project on the host side, which is then sent to the invitee. If the invitee chooses to use an existing project, another FileList is constructed for the state of the local project. The invitee then compares the two FileLists to determine which resources already exist in the local project, and if they have the correct checksums. Files are only requested from the host if either they don't exist, or if the checksums don't match.

For the synchronization of the VC state during the invitation, FileList was the obvious choice to store the additional information, since it contained most of the other information used in the invitation process. The next section describes several changes that were made to improve the extensibility of the FileList class.

4.1 FileList Refactoring

The FileList class had multiple responsibilities: Storing a serializable list of files and their checksums, comparing two file lists, and storing the comparison result. In revision 2231 and earlier, the list of files and their checksums was stored in a Map<IPath, Long> called "all". To store the result of a comparison between two FileLists, there were four additional fields called "added", "removed", "altered", and "unaltered". The first two were used if the path exists in only one of the two lists, and the other two indicate whether or not the checksums match. These four fields also had the type Map<IPath, Long>, even though the checksums aren't needed for a comparison result. In fact, the other field is called "all" because it is supposed to contain all entries from added, altered, and unaltered as the class invariant. To simply store a list of files instead of a diff between two lists, the data is stored in the field unaltered.

After computing the difference, the invitee needs to request a list of files from the host. This was implemented by simply sending the entire diff to the host, who would then use the paths of only the altered and added entries to prepare the list of files to send to the peer.

This design was lacking when it comes to separation of concerns. The same class was used for a list of files and the comparison result between two lists of files, and there is no clear distinction between the two. Since every class should only have a single responsibility, the comparison functionality was moved to a separate, new class `FileListDiff`, and the fields that store the comparison result are now of type List<IPath>. FileList now only has a single map to store the checksums of files, called entries. To request a list of files, the invitee now constructs and sends a new FileList to the host, which also means that a FileListDiff doesn't need to be serializable.

4.2 Synchronization of the VC state

If the shared project is managed, it first needs to be connected to the repository. This might require a checkout or connect operation, which are provided by the class `VCSAdapter`. The parameters for these operations are the repository URL, the path of the project in the repository, the revision number, and the ID of the Team provider. These parameters were added to the FileList class.

After the project is properly connected, the VC state of every resource must be synchronized. To transmit this data, the class FileListData was added, which not only stores the checksum of a resource, but also the VC state. After the FileList refactoring, only the type of one field (instead of five) had to be changed, from Map<IPath,Long> to Map<IPath,FileListData>.

Since the VCS operations are recursive, the synchronization requires a preorder traversal of the resources, starting with the project. For every resource, the current VC state is compared to the state of the host, and if there are differences, either a switch or update operation is performed to synchronize the VC state.

Since it is possible that some resources were deleted locally, a revert operation must be performed before the synchronization step. After all, a precondition for switching or updating a resource is that it exists. This revert operation will recreate any deleted resources from the base version. The downside is that it might undo some local changes that are also present at the host, for example from a previous session.

When the invitation process is finished, the local project of the invitee will be an exact replica of the hosts' project, with not only the folder structure and file contents being replicated, but also the VC state of every resource in the project. The next chapter describes how the VC state is kept in sync during a session.

Chapter 5

Updates of the VC state during a session

This chapter describes how the VCS operations connect to/disconnect from a repository, update and switch are replicated. The replication of these operations can be broken down into three separate parts. When a driver performs such a VCS operation, Saros must first be able to detect and identify exactly which operation was performed, and which parameters were used. These parameters must then be transmitted to the other participants to instruct them to replicate the operation. Lastly, the other participants must be able to execute the operation.

5.1 The Eclipse Resource API

To detect and respond to resource modifications in the workspace, Eclipse plug-ins register an IResourceChangeListener which will receive IResourceChangeEvents.[Art04] Most of the time, only a small fraction of the resources changes.

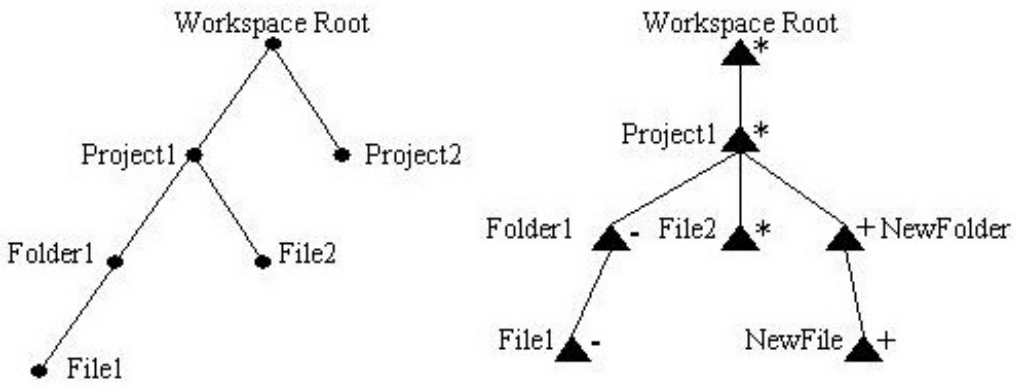

Figure 5.1: On the left is the folder structure of an example project. On the right is the structure of an examplary IResourceChangeEvent for the deletion of Folder1, the modification of File2, and the creation of a new folder and file in Project1. Note that the event doesn't contain any reference to the Project2, since it was unchanged. (Figure taken from [Art04])

In order to keep the event processing proportional to the number of the changes, as opposed to the size of the workspace, an IResourceChangeEvent is the smallest subtree of the project file tree containing the workspace root, and all the the changed resources.

An event consist of `IResourceDeltas`, every one of which corresponds to one resource in the workspace. A IResourceDelta carries information on exactly what changed in a resource between two discrete points in time. If, for example, a file was moved within the workspace, the event node corresponding to the new path of the file would also contain the old path. Other types of information include file content changes[1], and changes to the sync info of a resource, which includes the VC state.

Eclipse also provides an implementation of the visitor pattern for `IResourceDeltas`. Plug-ins can implement the `IResourceDeltaVisitor` interface,

[1]To be exact, this flag really means that the file's time stamp changed, which would also be set if the file was merely touched.

which can be used to visit all the nodes in an `IResourceDelta` tree in a preorder traversal.

The Resource API also provides a number of flags for each resource. Files can be marked as "derived", which means that they don't contain original user created content, but that they were created by some tool. For example, a compiler typically creates some artifacts, like .class files in case of the Java language. Derived resources are ignored by Team providers, since they can easily be recreated from the source code. Also, since they are not particularly interesting to a user, they are not displayed in the Package Explorer.

Most VCSs use files or folders in the working directory to store meta data about the managed resources. For this purpose, resources can be marked as "Team private". These resources are normally ignored by other plug-ins, including all of Eclipse's default plug-ins.

5.2 Detection of VCS Operations

When receiving a notification that the VC state changed, we need to be able to determine exactly what changed. Unfortunately, the Resource API only provides a simple flag to indicate that some part of the SyncInfo changed, but there's no way to tell which parts exactly. Therefore, the VC state of every resource in the project needs to be stored. When a change occurs, the new value can be compared to the previous value to determine if it changed. If the resource URL changed, the operation was a switch. If only the revision changed, but not the resource URL, the operation must have been an update.

The next question is where to store this information. The ConsistencyWatchdogServer already stores a list of checksums for every document which is opened in an editor, so it would be possible to extend this class to store more information, and on all the resources in the project. However, it serves an entirely different purpose, namely to provide a way for the other clients in the

session to determine if their document states are still in sync with the host. Also, the watchdog mechanism is only concerned with documents, while the detection of VCS operations must handle all files in the project. For these reasons it was decided against touching this part of Saros.

5.3 SharedProject refactoring

There was no class in Saros that stores project specific information, so a new class had to be created. Unfortunately, the class name SharedProject was already taken.

Originally, Saros was designed to share only one project. Before I started this work, efforts were made to support more than one project per session. However, the implementation is quite incomplete, and frequently there is no clear distinction in the code between the concepts "session" and "shared project", because they used to be the same. An example is the class SharedProject, which really didn't represent a single project, but the entire session. So this class needed to be renamed, and "SarosSession" was chosen as the new name.

To be able to store the state of a single project, a new class SharedProject is used. Saros's job is to replicate a shared project, i.e. to keep copies of the project on the peers in sync with the local project. A SharedProject represents the state that these remote copies are supposed to be in. Whenever a driver detects a mismatch between the project and its corresponding SharedProject, we know that we need to send activities.

5.4 Replication of VCS Operations

To instruct the other participants of replicating a VCS operation, the pre-existing activity architecture of Saros is used. An activity is a message from

a driver that instructs other participants to perform an operation. An example would be the FileActivity, which is used for the creation, modification, deletion and renaming of files.

The class VCSActivity was created to hold all the parameters to replicate the operation remotely. It implements the IResourceActivity and IActivity interfaces. VCSActivity objects are created exclusively by the SharedResourcesManager. When a change to the VC state of a resource in a shared project is detected, the appropriate VCSActivity is created and sent to the other participants.

The SharedResourcesManager is responsible for executing VCSActivities to replicate the VCS operation. When a VCSActivity is executed, a simple ProgressMonitorDialog is used as the GUI, not only to provide feedback for the user about the type of operation and its progress, but also to block the Eclipse GUI, which prevents the user from creating any other activity.

The SharedResourcesManager is also resposible for the other kinds of IResourceActivity: FileActivity and FolderActivity. FolderActivities are used for the creation and deletion of folders, and FileActivities are used for the creation, deletion, and modification of files. Typically, every VCS operation results in the modification of some resources of a project, which in turn triggers the creation of multiple file or folder activities. If all of these activities were executed by other participants, some changes might get applied twice.

Before the implementation of a fallback mechanism, these file or folder activities are simply not sent to the other participants. But the important part is to detect if one activity is part of another activity, as it is needed for the implementation of a fallback mechanism as well. For this purpose, a method `includes` was created in the VCSActivity class to test if the activity includes another one. If, for example, an update operation would modify one file and create another, three activities are generated by the SharedResourcesManager: A file modification `m`, a file creation `c`, and the update operation `u`. In

this case, the following statements would evaluate to `true`: `u.includes(m)` and `u.includes(c)`. Only those operations are sent to the other participants which are not already included in other activities.

Chapter 6

Results and Evaluation

6.1 Requirements Evaluation

As stated above, due to time constraints not all the requirements mentioned in chapter 2 could be met. The design and implementation of the requirements 1 and 6 were described in chapter 3, chapter 4 dealt with requirement 2, and chapter 5 described the realization of requirements 3 and 4.

This leaves a number of tasks unfinished:

- A fallback mechanism in case a participant is missing a dependency, or doesn't have access to the repository. (Requirement 7)
- Support for more Team providers. (Requirement 8)
- To run VCS operations in the background, without blocking the GUI from read access. (Requirement 9)
- To start the replication of a VCS operation as soon as it begins, not only after it finishes. (Requirement 10)

The fallback mechanism is certainly the most important, since it would make the software more robust, and the user interface easier to understand.

6.1.1 Fallback mechanism

The best way to keep access to the repository optional would be to use redundant activities. For example, when an SVN update adds a file, the driver would send one activity for the update operation, and another one for the creation of the file. The peer would first try to replicate the SVN update by executing the update operation. If this fails for whatever reason, the peer could then simply execute the FileActivity to create the file. However, if it doesn't fail, the FileActivity would need to be discarded, to avoid applying a change twice. This is particularly important for TextEditActivities, which don't describe an absolute state, but only a relative state change of a document.

A VCS operation typically consist of multiple file and folder activities. To express this relationship, an activity a is called *redundant*, or *included* in another VCSActivity v, if executing v successfully has the same effect on the content of one of the modified resources as executing the activity a. Currently, redundant activities are simply prevented from being transmitted to other participants. To support sending these redundant activities, there needs to be a way to indicate that an activity is included in a VCSActivity. After executing the VCSActivity, the result needs to be inspected. If the execution failed, every activity which was included in the VCSActivity needs to be executed, otherwise they can be discarded. For this reason it seems best to bundle all the activities that are included in a VCSActivity, and to send them as part of the VCSActivity.

Alternatively, a participant could try to execute a VCSActivity, and only if it fails request the contained activities from the host. This solution would however require a lot of effort to manage the data involved, since every driver executing a VCSActivity is required to store the included activities until receiving notice from every participant that the original VCSActivity has finished executing. While the first solution of sending redundant data certainly is more wasteful when it comes to data transfered, it would be much

easier to implement.

6.2 Problems Encountered

A number of problems only became clear during the later stages of this work, and remain to be solved.

6.2.1 Problems with multiple drivers

An aspect that was ignored so far is when there's more than one driver in a Saros session. Since any driver is allowed to make changes to the project, conflicting commands might be issued by different drivers. This problem is handled well by Saros in the case of text edit operations. In [Rie08], the Jupiter algorithm was implemented, which enables concurrent editing of text documents.

However, other operations like the modification of files and folders can lead to problems in multi-driver sessions. For instance if one driver Alice renames a file A to B, while another driver Bob moves the file to a different directory at the same time. In this case, each Saros client would send a FileActivity to instruct the other client to execute the operation. When the client of Alice tries to execute Bob's file move, the source path can't be found, since the file has already been renamed, which would result in an error message. Similarly, when the client of Bob tries to execute the file rename, the file was already moved to a different directory, and a similar error will be displayed.

Similar concurrency problems exist with the VCS operations. An example is when one driver executes a VCS operation on a folder that another driver renames at the same time. Probably the worst case would be a situation like two concurrent update operations to different revisions. The problem here is that a concurrent execution of these conflicting commands wouldn't even be

prevented by an error.

In cases like these, Saros provides no facilities to handle or resolve concurrency issues, and the local copies of the project can become inconsistent. In [Jac09], the way Saros handles these issues is described as insufficient. When a file or folder activity is about to be executed, it is tested if the client is currently the exclusive driver in the session. If this isn't the case, a warning message is displayed, stating that executing the operation might result in an inconsistent state of the session.

This behavior is implemented by using facilities of the Team API, which enable plug-ins to inspect and veto changes to the file structure of the project. Unfortunately, these facilities can't be used to intercept and prevent the VCS operations by a Team provider. For this reason, a similar warning message can't be displayed for VCS operations.

In [Jac09] it was proposed to handle these issues by either asking the other drivers to allow an operation, or by extending the Jupiter algorithm to also handle file and folder operations. Since it is not possible to prevent the execution of VCS operations, the first of these suggestions doesn't apply to our problem, however it is conceivable that the second suggestion could also solve the concurrency issues of VCS operations. The advantage would be that the way that text edit activities are handled wouldn't have to be changed.

But as in the case of file and folder operations, for this first implementation of VCS integration it is simply assumed that there is only one driver at a time that can execute VCS operations.

6.2.2 Inconsistency detection

Saros only has limited support for inconsistency detection and resolution. Inconsistencies are detected by the "consistency watchdog", which is a thread running on the host. Every ten seconds, it scans every document currently

open in an editor, computes the checksum if the document is part of a shared project, and sends these checksums to the other participants. The host's checksums are then compared to the local checksums of the peers. In case of a mismatch, the peer can request all the inconsistent files from the host.

One problem with this approach is that inconsistencies aren't detected unless the file is opened in an editor. VCS operations are independent of editors, and most of the time, not all files modified by a VCS operation are opened in editors. On top of that, folder can't be opened in editors, so inconsistencies involving folders can't be detected or resolved.

In addition to inconsistencies because of different file contents, an inconsistency can also occur because the two copies of the file point to different files in a repository, i.e. because the VC state of the files differ. While this typically also means that the file contents differ, the resolution of the inconsistency would require different actions. If only the file content is different, having the host send the file is sufficient. But if the VC states differ, the proper way to resolve the inconsistency would be to perform a switch or update operation on the file. Only if this VCS operation fails, the software should resort to requesting the file contents from the host.

6.2.3 External VCS adapters

Currently, the implementation of a concrete `VCSAdapter` requires access to Saros' source code. The only way to instantiate a concrete `VCSAdapter` is by using a static method in the `VCSAdapter` base class, and every `VCSAdapter` is hard-coded in this method. This also means that in order to compile the Saros plug-in, every supported Team provider must be present, introducing a lot of dependencies.

A more flexible approach would be to put the concrete `VCSAdapter` in a separate plug-in. When the adapter plug-in is loaded, it would register its adapter with the `VCSAdapter` class. The identifier of the Team provider needs

to be associated with the adapter class, and the base class could store these associations dynamically in a map.

6.3 Conclusion

This thesis describes the integration of version control systems in Saros. Saros' replication mechanism of a project was extended to also synchronize the VC state of the resources in a shared project. The version control system is used automatically during the invitation, such that not only the file contents are synchronized, but also the underlying base version of the files. During a session, VCS operations like update and switch are automatically detected, and distributed to the other participants to keep the remote copies synchronized.

At the moment only Subversion is supported, but the extensible design allows to easily add support for more version control systems to the software. While limited time prevented all the requirements from being met, the most important aspects were implemented successfully, adding an often requested feature to Saros.

Bibliography

[Art04] John Arthorne. How You've Changed! `http://www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html`, 2004. [Online; accessed 25-09-2010].

[CW01] A. Cockburn and L. Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–248, 2001.

[Dje06] Riad Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung, 2006.

[Jac09] C. Jacob. Weiterentwicklung eines Werkzeuges zur verteilten, kollaborativen Softwareentwicklung, 2009.

[PCSF08] C.M. Pilato, B. Collins-Sussman, and B.W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2008.

[Rie08] O. Rieger. Weiterentwicklung einer Eclipse-Erweiterung f ”ur verteilte Paar-Programmierung im Hinblick auf Kollaboration und Kommunikation, 2008.