Extreme Team Collaboration

Synchronous collaboration in Eclipse

Jeldert Pol
Master Thesis
March 6, 2009

Master Software Engineering
University of Amsterdam

Supervisor: Prof. Dr. P. Klint
Institute: Centrum Wiskunde & Informatica
Availability: public domain

UNIVERSITY OF AMSTERDAM

Contents

Abstract

Most software systems are too large to complete single-handed. Therefore multiple programmers are needed. The work is divided amongst them. Some of the programming activities that are performed change code, or require that the code is not changed (when running tests for example). The result of their work, the changed codebase, must somehow be synchronized between the programmers. This can be accomplished by using asynchronous or synchronous collaboration.

Asynchronous collaboration means changes are being synchronized after a programmer is finished making changes. This is usually done with a version control system like Subversion. Synchronous collaboration means changes are being synchronized between programmers in real-time. This can be done with a collaborative real-time editor (CRTE). Using a combination of both allows programmers to choose which advantages and drawbacks are most appropriate for each situation. This comes down to using a synchronous collaboration when working on a related task, and using asynchronous collaboration when working on an unrelated task.

Current implementation for using synchronous collaboration in Eclipse do not support all programming activities. Shared editors only allow programmers to simultaneously edit one file, while distributed pair programming allows only one programmer to make changes.

Extreme Team Collaboration, or XTC, is less restrictive. It allows multiple programmers to simultaneously edit multiple files. All changes to the codebase are intercepted, including automated refactorings, creating new code, moving files, removing folders etc. Because of this, XTC supports most programming activities.

This is accomplished by a tight integrating with the Eclipse IDE. All changes made are intercepted by XTC, and sent to the server component of XTC. This server forwards these changes to every other participant, where the changes are applied. The result is that the codebase is synchronized in real-time, while not being restricted to the limitations of CRTE's or (distributed) pair programming.

Preface

This thesis is the result of fullfilling my Master Software Engineering at the University of Amsterdam. A year of hard working was needed to accomplish this. I could not have done this solely by myself. Therefore I would like to the persons who made this possible.

I would like to thank all the professors, teachers and assistants involved in the Master Software Engineering a the University of Amsterdam, all persons of the SEN1 group at the CWI, especially Paul Klint and Jurgen Vinju, my fellow students Jeroen van den Bos, Hidde Baggelaar and Qais Ali, my friends Annemieke, Hannes and Rutger, my parents, brother and sister, and everyone who showed interest in my progress during this last year.

Jeldert Pol
March 2009

Chapter 1

Introduction

People who write software want to do this fast, and do it right. A common way to accomplish this is to split and divide the work, because it would be too much work to do it all alone. Dividing work across programmers will result in a shorter time to completion. Also, expertise can be utilized better.

When work is divided, programmers create a copy of the codebase. While performing their programming activities, changes are made to this copy. These changes need to be synchronized with the other programmers.

This synchronization is done with tools. For programming, version control systems are often used. This is a form of asynchronous collaboration, where changes are synchronized after they have occurred, for example after finishing a task. When this is done a lot of changes may have occurred. Other programmers could have made changes as well. This not only causes synchronization problems, it may also break the code, since it may not work with the changes made.

Another problem is that working together is hard when using asynchronous collaboration. For example, it is almost impossible for two programmers to work together on the same task from different computers. They simply don't see each other's changes.

Synchronous collaboration is another way of synchronizing. Here, changes are synchronized at the moment the change occurs: it synchronizes in real-time. Only, current implementations are not optimized for programming: they don't support all programming activities.

1.1 Extreme Team Collaboration

This thesis will present Extreme Team Collaboration, or XTC for short. This is an implementation that allows synchronous collaboration. A major difference between XTC and existing solutions is that XTC actually tries to support most programming activities. This will form a better solution than existing ones, and also reduces the problem of file-sharing.

1.2 Scope

Creating a piece of software consists of many different phases (the software development process). This thesis will only focus on those phases which actually make changes to the codebase.

Version control systems are often used to store and distribute the codebase. Subversion is one of those systems. This thesis will only focus on Subversion to accomplish asynchronous collaboration.

Programming activities are mostly performed from within an IDE. This thesis will focus on the Eclipse IDE.

1.3 Research question

The main research question of this thesis is:

> Can programming activities be performed using synchronous collaboration, and what are the advantages and disadvantages of doing so?

1.4 Overview

In order to be able to answer this question, some knowledge needs to be gathered. First of all it needs to be clear what programming activities there are. This will be discussed in chapter 2. Next, the different ways of collaborating and their characteristics are being discussed in chapter 3. The ease of how to accomplish this collaboration is discussed in chapter 4.

Chapter 5 will look at existing solutions for synchronous collaboration in Eclipse. The characteristics of Extreme Team Collaboration will be discussed in chapter 6. Chapter 7 will discuss the actual implementation of XTC as a plug-in for Eclipse. A validation of XTC and its implementation can be found in chapter 8.

This thesis will be concluded in chapter 9.

Chapter 2

Programming activities

This chapter will explain the different programming activities. This will happen in two steps: from phases to tasks (section 2.1) and from tasks to activities (section 2.2).

2.1 Phases and tasks

The act of creating software consists of different phases. The most common ones are: planning, requirements, specification, architecture, design, implementation, testing, deployment, and maintenance. Depending on the software development process (or model) used, the steps may be named differently, be applied iteratively or in different order, or not exist at all.

Not all of these activities involve the creation or alteration of code, only implementation, testing and maintenance do.

These phases can be divided into tasks. A task is performed by a programmer in a specific phase. During the implementation phase, a programmer tries to understand existing code, implement new functionality, tries to improve the maintainability of the code, and may optimize code. During the testing phase, a programmer tries to verify the code, and fix the code. The maintenance phase consists of all of these tasks. This relation between phases and tasks can be seen in table 2.1.

Phase	**Task**
Implementation	Understanding existing code Implementing new functionality Improving maintainability Optimizing code
Testing	Verifying code Fixing code
Maintenance	All of the above

Table 2.1: Programming phases and tasks.

Task	Activity
Understanding existing code	Reading code Changing code
Implementing new functionality	Changing code
Improving maintainability	Documenting code Refactoring code
Optimizing code	Refactoring code
Verifying code	Creating tests Running tests
Fixing code	Debugging Fixing bugs

Table 2.2: Programming tasks and activities.

2.2 Tasks and activities

Each task can be divided into programming activities. An activity is a smaller step that is performed in order to complete the task. Table 2.2 shows the tasks and activities. How these activities are performed, and what influence they have on the codebase will be described in the next subsections.

2.2.1 Understanding existing code

Before a programmer can perform any of the other tasks involving the codebase, he must first understand any of the existing code. This means he first of all needs to know the rationale of the program (the reason of its existence). This can be done by reading project documentation and examining the functionality of the program. After this, he can zoom in further and look at the architecture, design, and finally the implementation: the code itself. Pieces of code can be understood by reading the documentation associated with the code, reading the code itself, and running the code.

Some programmers change existing code in order to better understand what influence it has on the whole program. They run the program to see how these changes affect the program. Of course, this needs to be done in isolation: the changes should not find their way into production code.

Also, changes made by other programmers should not alter the copy of the codebase that is being studied. If this happends, the behavior of the studied code could change, making it harder to understand. It may also break the code, since the changes of the other programmer may not be compatible with the self-made changes.

2.2.2 Implementing new functionality

When implementing new functionality, a program is able to perform certain actions that were not possible before. In order to do so, new code facilitating this functionality needs to be created. This includes the creation of new packages, classes, methods and statements. It also includes the alteration of existing code, and refactoring of existing code. In other words, not only the code is changed, also the layout of the codebase is changed.

2.2.3 Improving maintainability

Maintainability is an important aspect of software. Over time, requirements can change, the environment can change, and bugs can be found. For software to stay useful, it needs to adapt to this: it needs to be maintained.

Documenting code Documenting code helps programmers understand the code. This is accomplished by writting down properties of the code, like the intention of the code, any pre-conditions, post-conditions, etc. This makes it possible to match the intention of the code and the actual code. Sometimes the thoughts of the programmer are made explicit, to document why something was done in a particular way.

Since this kind of documentation is about the code, it is usually written as comments in the source code. This means that documenting code also changes the files containing the code.

Refactoring code Refactoring code means that code will be shaped differently, but still acts in the same way. This can be done to make the code more aesthetic to the eye, or to improve performance. But the most common reason to refactor code is to improve understandability (and thus increase maintainability). Duplicate code can be removed, variables can be renamed to better reflect their content, classes can be moved to better fit in the hierarchy, multiple statements can be extracted into their own method, etc.

So refactoring may make changes the whole codebase, including the creation and deletion of files and packages.

2.2.4 Optimizing code

Sometimes, code is not optimal with respect to used resources. When optimizing code the same result is accomplished while using less resources. This can be expressed in amount of memory used, time to complete a calculation, etc. Improvements can be made by implementing better algorithms, using different libraries, etc.

Optimizing may look similar to refactoring, but has a different purpose. The understandability of the code is sometimes sacrificed in order to gain the most out of an optimization, thus reducing maintainability.

Optimizing code changes code, and may alter the layout of the codebase (files and folder may be added or removed).

2.2.5 Verifying code

Verifying code is the act of making sure the behavior of running code matches the expected behaviour. This verification is done by creating and running tests. Tests run code, and expect a certain outcome, which is compared to the actual outcome. If they are the same, the test passes, otherwise it fails.

Creating tests Tests consist of code themself, and are usually written in the same programming language as the code being tested. These tests can be placed in the same file as the code, but it is much more common to place them in a file of their own. These new files can be placed in the same location (package) as

the code being tested, or in a special test package. Because tests are so tightly connected to the code, they are often part of the codebase.

Because of this, creating or changing tests includes modifying files, and making changes to the layout of the codebase.

Running tests After making changes to the code it is common to run the whole set of tests available. This makes sure that the changes made still result in the expected behavior, and do not have any side effects on other pieces of code.

Running a test does not alter the code in any way, it only uses the code. More importantly, the code being tested should not change while running the test. This is needed to guarantee that the results of the test apply to the actual code.

2.2.6 Fixing code

Bugs are common in software. Bugs can have different causes, but they all result in unexpected, or unwanted behavior.

Debugging Tests can reveal a bug in the code. However, tests do not pinpoint the cause of the bug. Debugging is the act of understanding why the bug occurred, and finding the code that is responsible. This is done by formulating a hypothesis explaining why the code fails. After this the code can be changed, to see if the hypothesis was correct, and the failure no longer occurs. This is done by observing the code again (through tests or runtime behavior). When it fails again, the process starts all over again, refining the hypothesis. This will result in understanding why the code fails, and where it fails [Zel05].

In order to debug, the code should not be changed by others, as this has influence on the behavior of the code. As the behavior changes, a new hypothesis needs to be formulated again. The code does need to be changes by the programmer himself, to see if the hypothesis is right.

Fixing bugs Fixing a bug is a combination of finding the bug and fixing it. A bug can be found by creating and running tests, and debugging. The source code should not be changed by others during this.

The bug can now be fixed. This includes changing code and sometimes creating new functionality, which may cause the layout of the codebase to change.

2.3 Conclusion

All mentioned tasks and activities depend on code. Running tests and some parts of debugging require that the code does not change. For every other activity, the code actually needs to change in order to perform the activity. These changes can include changing the content of a file and changing the layout of the codebase. When reading code and debugging these changes may only be done by the programmer directly involved.

Changes to the code eventually need to be synchronized again between programmers. The next chapter will look at different ways of collaboration, which can accomplish this synchronization.

Chapter 3

Ways of collaboration

Almost all programming activities are divided in some way or another. Most software systems are simply too large to be completed single-handed. Because of this people are added to the project, and the work is divided amongst them. This results in the product being finished sooner.

Of course, as Brooks points out, this does not apply to all situations: "Men and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them*" [BJ75]. This is not the case for building software: "A major problem in software development is the breakdown in communication and coordination among developers" [CdSH+04]. Communication is required, and therefore adding more people to the project will not always result in a faster time to completion.

However, it is save to say that most projects consist of more that one programmer, and by doing so a benefit in speed is gained. An additional benefit of dividing work is that expertise can be utilized better. For example: graphical experts can focus on the tasks involving graphics, while data experts can focus on the storage of information.

The result of this divided work, the changed codebase, must somehow be distributed amongst all participating programmers. What this means is that the codebase needs to be synchronized between the programmers. There are two ways of doing so: using asynchronous collaboration and synchronous collaboration. What these are will be explained in section 3.1. The influence these forms of collaboration have on different aspects of programming will be discussed in the subsequent sections of this chapter.

3.1 Asynchronous and synchronous

The difference between asynchronous and synchronous collaboration comes down to one aspect: the moment of synchronization [DB92]. In the first approach synchronization is performed at defined intervals. This can be at fixed times (end of the day) or at certain situations (after finishing a task). Synchronous collaboration on the other hand synchronizes in real-time, right at the moment a change is made.

Both ways of collaboration are almost always performed by tools: synchronizing by hand takes too much time and is error prone.

3.1.1 Version control system

An often used way to use asynchronous collaboration is by using a version control system, like Subversion [CSFP04]. With Subversion, the complete codebase is stored at a central location, the repository. Programmers create a copy of the codebase and make modifications to it. Now the modified codebase needs to be synchronized again with the repository. Other programmers can now receive the latest version from the repository. They synchronized. This process is explained in more detail in appendix A.

3.1.2 Collaborative real-time editors

For synchronous collaboration, collaborative real-time editors (CRTE's) are most used. Normally an editor allows a person to edit a resource. A collaborative real-time editor (CRTE) allows multiple persons to edit the same resource simultaneously. This concept has been around for a long time.

As described by Dourish and Bellotti [DB92], most of the early collaborative editors are based around the idea to increase awareness of other persons activities. This is done by showing what others are doing, and are mainly based on the act of writting (books, manuals, etc.). These implementations also assume that in order to do so they need "explicit, or restrictive mechanisms for ensuring an easy collaboration, such as annotations, role assignments, access rights and so forth" [DB92].

This kind or CRTE's still exists today. A recent example is Google Docs[1]. Here multiple participants can edit the same textual document. All participants instantly see each other's changes, and can also edit them again.

Solely using a CRTE is not very practical: the ability to save different versions is lost. Using a CRTE and a version control system is much more practical. Now programmes can use synchronous collaboration via the CRTE, and still have the ability to save different versions via the version control system. This version control system can now also be used to synchronize when changes where made by programmers not using a CRTE.

3.1.3 Local collaboration

Working together on the same computer is also a form of synchronous collaboration. Pair programming (see section 5.2) is a well known form of local collaboration. It is still needed to synchronize with other programmers, since they are working on different computers. To accomplish this, both asynchronous (Subversion) as synchronous (CRTE) collaboration could be used. Therefore local collaboration will not be discussed as a seperate way of collaborating.

3.2 Influence on aspects

Both ways of collaboration have a different influence on aspects of programming.

[1] `http://docs.google.com`

3.2.1 Autonomy

One aspect of collaboration is autonomy: the power to decide when changes will be made available to other programmers. Autonomy does have a big impact on collaboration. When not making changes available they cannot be seen nor used by others.

Subversion When using Subversion, programmers have autonomy. A programmer decides when his task is complete, and only then synchronizes his changes with the repository. This gives him the freedom to finish a task before sharing his changes.

CRTE There is no autonomy when using a CRTE. That is, every change is always synchronized in real-time. There is no room to finish something before sharing it.

3.2.2 Isolation

Autonomy also creates isolation: programmers are unaware of the activities and code changes made by other programmers. They can only see and use these changes at the moment they synchronize.

Subversion Programmers have some influence on isolation. They can decide when to synchronize their local copy with the repository. However, they need to do this actively. It is not possible to not have isolation.

CRTE When using a CRTE, there is no isolation. Changes made by others are instantly received. Programmers also have no influence on this isolation. The only option is to stop using the CRTE.

3.2.3 Compilability

Compilability means the code conforms to the language specifications. This does not mean the code behaves as intended, but rather that a computer can compile and run the code.

When programming it is normal that the code is in an uncompilable state from time to time. For example, when writing a statement it is not compilable until the whole statement is finished. This means that making changes to the codebase influences the compilability of the code.

Subversion Because programmers have autonomy and isolation, changes to the codebase always originate from the programmer himself. The programmer knows the code can become uncompilable when he changes something. He also knows that after finishing a task the code should be compilable again.

CRTE Because there is no autonomy and isolation, changes also originate from other programmers. When working on a task the code can become uncompilable. When the task is finished, it can still be uncompilable, since other programmers may continue making changes.

3.2.4 Traceability

Traceability means having the ability to link a change with the programmer who made the change. It also means linking a change with a specific task, so it is clear why the change was made.

Subversion Since programmers are working in isolation it is clear who made the changes when committing. This makes it traceable: it is easy to look back, and pinpoint when and who changed a specific piece of code.

Programmers are usually working on a single task at the time. When committing, only the changes related to this task are committed[2]. When committing, the programmer gets the ability to enter a message. This message is used to provide additional information about the change, for example a link to a bug report. This way the change can be linked to a task.

CRTE Because every change is instantly synchronized with everyone, it may become unclear where each change originated from. Does this piece of code come from programmer A, B, or was it already present? Some CRTE's show who changed which code, but after saving, or when starting a new session, this information is usually lost.

The changes made still need to be committed to the version control system. It is hard for the committing programmer to make a sensible changelog for his commit. This is because he also commits changes made by other programmers, which are now present in his local copy of the codebase.

3.2.5 Dependency conflicts

Programmers can make changes in different files, which are dependent on each other. The code in one file may rely on the API or data structure represented in the other file. A programmer can create code with an API or data structure in mind. At the same time, another programmer may well be changing these.

Subversion With Subversion, both programmers are unaware of the fact that a problem may arise. This is because the changes are not send to each other. Only after manually synchronizing with the repository the problem becomes clear to the programmers. To solve this problem, time and resources need to be invested in fixing one or both of the changes to be compatible again.

CRTE With CRTE's however, dependency conflicts can be detected at the moment they are created. When an API is changed, the programmer calling this API can instantly see this change. Also, the programmer changing the API may do a search for references which are now broken. This will even find the newly created reference.

3.2.6 Double work

Double work is the act of multiple persons trying to accomplish the same thing, without being aware of it.

[2]Plus any changes to make the code compatible with changes already committed to the codebase.

One way this can happen is when programmers start working on the same task individually. Because work is usually divided in more tasks than there are persons, not every task can be performed at the same time. This means some tasks have to wait and are being picked up later. It is not always clear who is assigned to which remaining task, or persons start cherry picking the available tasks. In these situations it can occur that multiple persons start working on the same task, without being aware of this unwanted situation. This may not happen very often, because dividing tasks is usually well defined, but double work can also occur when working on different tasks.

What happens if two programmers realize they depend on some functionality, which is not yet present? In order to be able to finish their original task, they may both just start implementing or changing this needed functionality, without being aware of it.

Subversion This double work can be detected when synchronizing, because they may cause a merge conflict (see appendix A). Now the situation can be corrected. However, double work may not always be detected this way, since naming or layout differences between programmers may not cause a conflict.

CRTE Double work may be detected earlier when using a CRTE, since the code can be seen and used by every participant at the moment it is created. Programmers intending to create the same functionality can see this. However, when naming or layout differences between programmers exist, this functionality may not be seen by the programmer, and will still result in double work.

3.3 Conclusion

The main difference between asynchronous and synchronous collaboration is the time of synchronization. Subversion is a form of asynchronous collaboration, a CRTE is a form of synchronous collaboration.

Both ways of collaboration have their influence on different aspects of programming. This is summarized in table 3.1. As can be seen, using Subversion (asynchronous collaboration) gives programmers a lot of control. When using a CRTE (synchronous collaboration), this level of control is almost completely lost.

If this is so, why use synchronous collaboration? The next chapter will answer this question.

	Subversion	CRTE's
Having autonomy	✓	−
Working in isolation	✓	−
Influence on compilability	✓	−
Ease of traceability	✓	±
Early detection of dependency conflicts	−	✓
Early detection of double work	−	±

Table 3.1: Influence of aspects.

Chapter 4

Ease of collaboration

Why use synchronous collaboration, when there is less control? Simply because it makes it easier to collaborate.

With synchronous collaboration programmers can work together. They instantly see changes, and can also use them. One programmer may write new code, and the other can instantly write documentation about it, optimize the code, assist the other programmer, monitor his progress etc.

4.1 Divide work

Synchronous collaboration enables programmers to divide work. Suppose a stub class needs to be implemented. With synchronous collaboration two programmers can do this. They instantly see which methods are already done, and can start with another one. With asynchronous collaboration, programmers could not do this, simply because they can not see what has and hasn't been done.

The same applies to documenting code. When documentation needs to be written or improved, other programmers can instantly see and use this new information. There is no possibility of documenting code twice, since once a programmer started, the others can instantly see this and move on to another part.

Refactoring a part of the codebase can also be divided. Multiple programmers can refactor a different part. This may change something in the code another programmer is refactoring. But since these changes are also applied in his copy of the codebase, there will be no merge conflicts.

Suppose one programmer renames class "Job" to "Vacany". This class uses "Loan", which is renamed to "Salary" by another programmer. This last rename will also rename any uses of this class, as long as it is performed with an automated refactoring from an IDE. When performed asynchronous, the rename of "Loan" to "Salary" would not be present in "Vacany", causing a compilation error after synchronizing. This is illustrated in figure 4.1.

Creating tests can also be divided. Everyone involved can see which tests are already written, and which ones still need to be done.

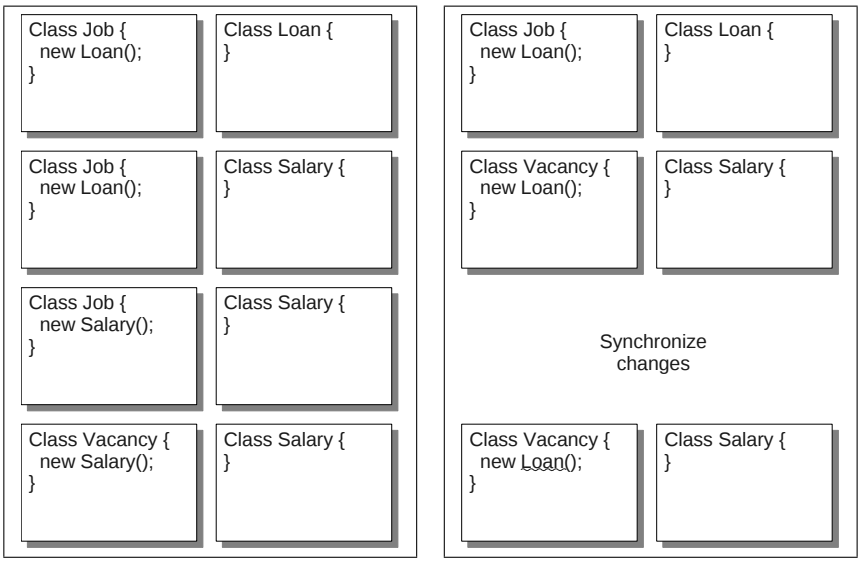

Figure 4.1: Refactoring done synchronous (left) and asynchronous (right).

4.2 Work together

Synchronous collaboration could also be used to work together.

Programmers may both be working on the same piece of code. Both may be studying a method, to see if it can be improved. A proposal made by one programmer can be put to practice immediately. The other programmer can see this, and make suggestions, call a halt to the action, or simply help him with making the change.

Another example. Suppose one programmer is struggling with a piece of code. By using synchronous collaboration he can receive help from another programmer. Because they are already synchronized, the other programmer sees the exact same code, and can thus help him with finding a bug, suggest a different approach, or simply fix a non-compiling statement.

4.3 Drawback

There is a drawback. As discussed in chapter 2, running tests and debugging demand an unchanging codebase. This is hard to accomplish when others can change the local copy of the programmer trying to run a test, or step through code with a debugger.

This makes using synchronous collaboration less suitable for these tasks. They can work, but then every programmer involved must stop making changes. This is only feasible when they are all working on related tasks, and know that one of them is going to run a test.

For instance, one programmer optimized one method, and wants to run a test to verify the behavior of the code. If another programmer is making changes to a different part of the codebase, it may become uncompilable. The codebase can even be in such a state, that it takes a long time before it is compilable

again. Just imagine a programmer who just started refactoring, just before one wants to run some tests.

What happens when a programmer is finished with a task, and wants to commit this to the version control system? He enters a message containing information about the change. But if changes made by others are also present in his local copy, he cannot enter any information about these changes.

A general rule is to only commit code to the repository that is compilable and tested. However, when changes keep coming in, it is almost impossible to make sure the code passes the tests.

4.4 Conclusion

Using synchronous collaboration does have advantages. It enables programmers to divide work and work together. It basically allows them to collaborate.

Drawbacks include difficulty for running tests, debugging, and committing.

It can be concluded that synchronous collaboration is useful, but only when the programmers involved are working on related tasks, so they know what to expect.

The next chapter will look if CRTE's actually support the programming activities discussed in chapter 2.

Chapter 5

Collaborative real-time editors for Eclipse

The previous chapters showed what programming activities there are, and that synchronous collaboration makes collaboration easier when working on a related task. This chapter will see if current CRTE's actually support the discussed programming activities.

Since most programmers use an IDE while programming, the CRTE's must work from inside the IDE. Since the scope of this thesis was set to Eclipse, the CRTE's must work in Eclipse.

5.1 Shared editors

Shared editors in Eclipse allow programmers to share a single file with another programmer. This works by opening a file, and sending an invitation to the other programmer. When he accepts, the same file is opened. Both programmers can now edit the same file: changes made by each programmer are synchronized in real-time.

5.1.1 Real-Time Shared Editing

Real-Time Shared Editing [RTS] was one of the first efforts to implement a collaborative real-time editor for Eclipse.

Even though the motivation of this project is pair programming, it does not work to the principles for pair programming. In pair programming only one programmer makes the changes to the code, here both programmers can make changes to the code. This has the potential that it can be used for non-strict pair programming, or working semi-individual. This means programmers can work on the same or closely related task, while still using synchronous collaboration.

Still, this plug-in is not ideal for programming. Simply the fact that both programmers can edit simultaneously, but are restricted to the same file, is a big restriction in the freedom of the programmers. Furthermore, multi-character operations (copy/paste) are not supported. This also implies that automated refactoring is not supported, since Eclipse treats this as a combination of multi-character operations.

This could not be tested however, since the plug-in was non-functional in several recent versions of Eclipse (3.2, 3.3 and 3.4 where tried). Since this plug-in is not actively supported anymore, no further attemps where made to verify the suspected behaviour.

5.1.2 DocShare

DocShare [DSP] is a continuation of Real-Time Shared Editing, and uses parts of its code. The only difference is that this plug-in works in more recent versions of Eclipse. It works in the same way as the Real-Time Shared Editing plug-in, and therefore has the same characteristics.

5.2 Distributed pair programming

There are quite a lot of systems aiming to support distributed pair programming. Normally, while pair programming, two programmers sit behind the same computer and work together on the same task [BA04]. Only, by doing so, they are restricted in their geographical freedom. Distributed pair programming can restore this geographical freedom. In this approach two programmers can still use pair programming while being at different locations. Research have shown that developing software with distributed pair programming seems to be comparable to pair programming [Bah02].

This is done by allowing persons to enter a session. In this session one programmer is the driver, the other one the navigator. When more programmers are allowed they are usually observers. The driver is the one that does the actual modifications to the code. The navigator and observers can not make changes. The navigator can assist the driver, keep him on track, and think of better ways to do things [BA04]. Communication between the programmers is usually done by means of an integrated chat facility.

Showing the actions of the driver can be transferred to the navigator and observers in two ways. The screen of the driver can be captured, and can be transferred to the navigator. This takes a lot of bandwidth. Another way is to capture all actions taken by the driver, and replaying them at the site of the navigator. Now only information about the action is being transferred, costing much less overhead. The plug-ins which will be described all use the later approach.

5.2.1 Sangam

As one of the first plug-ins to support distributed pair programming, Sangam [HRGW04, San] is quite feature rich. It synchronizes the editor, resources, refactoring actions and running of code. This means any actions related to these subjects are intercepted by Sangam, and reproduced at the site of the navigator. For example, when the driver opens a file, the same file is opened for the navigator. When the driver changes code, the changes are applied for the navigator. This covers most of the activities while programming.

A shortcoming of Sangam is that it only works when programming Java. This is because Sangam only intercepts, or listens to, Java related events. Currently, Sangam is no longer being developed.

5.2.2 PEP

With PEP [PEP], the driver shares a project. The whole project is then sent to the navigator. All actions of the driver are repeated on the side of the navigator.

Whether all programming tasks can be performed by PEP is unclear. The website does not provide much information, and the plug-in has not been maintained for more than one and a half year.

5.2.3 Saros

Also Saros [Sar] is similar to Sangam. But it also shows the current position of the driver in the code, and the selection the driver makes. Changes to the code are marked in the editor.

5.2.4 XecliP

A plug-in similar to Saros. XecliP [Xec] however is a bit more advanced. It has strong features for user management, like defining skills for developers. It can also record and playback a session.

Like Saros, all programming tasks are supported, but these actions are only supported for Java files.

5.2.5 XPairtise

The most advanced pair programming plug-in is XPairtise [XPa]. It allows multiple programmers to connect to a server. Sessions can be created containing multiple projects. Other programmers can join on of these sessions, and the projects will be imported or synchronized in their Eclipse. It features an integrated chat, whiteboard, overview of connected programmers, and the ability to switch roles.

5.3 Conclusion

Most of the discussed plug-ins for Eclipse focus on pair programming. This means that only one programmer can make changes to the codebase, while every other programmer is just watching him make these changes. They can not make changes themselves. They are restricted to the intention of the other programmer.

Other plug-ins allow two programmers to make changes simultaneously. However, they are restricted to a single file. This means they can only use synchronous collaboration when editing the same file at the same moment. Changes to other parts of the codebase (moving files, creating packages, etc.) are not supported. So also in this approach, the programmers are very restricted.

The next chapter introduces Extreme Team Collaboration, a different way of supporting synchronous collaboration while programming, which none of the plug-ins discussed in this chapter support.

Chapter 6

Extreme Team Collaboration

As the previous chapter showed, current implementations for using synchronous collaboration are very restrictive. This chapter will describe Extreme Team Collaboration, or XTC for short. This is a way to support synchronous collaboration, while being less restrictive.

6.1 Focus

As could be seen from chapter 5, current implementations are very restrictive. When working on a related task, pair programming is not a solution, since only one programmer may make changes. Using a CRTE is also not possible, because it restricts the programmers to the same file, and does not synchronize any other changes to the codebase.

Extreme Team Collaboration tries to be less restrictive. It allows a team of programmers to use synchronous collaboration, while all programmers can make changes, and supporting most programming activities.

6.1.1 Working semi-individually

Xtc allows programmers to work semi-individually. This means all programmers using xtc may individually make changes to the codebase. These changes will be synchronized with every other programmer. This is what the "semi" in semi-individually means: they still receive changes made by others. So unlike pair-programming, programmers are not restricted to the driver/observer model. They are also not restricted to the same file, as is the case with current CRTE's.

XTC is a form of synchronous collaboration, because changes are synchronized in real-time.

Teams of programmers do not all work on the same task. Some of them work individually on a task, some of them work on related tasks. XTC allows programmers to work semi-individual. This means when working on a related task, they can use synchronous collaboration, while not being restricted to a driver/observer model, nor being restricted to a single file, which is the case

with current CRTE's. Also, because they are using synchronous collaboration, it reduces the problems caused by asynchronous collaboration.

6.1.2 Supporting programming activities

Current CRTE's are not optimized for programming activities. This means they do not support all programming activities. Extreme Team Collaboration tries to support most programming activities.

How XTC can accomplish this will be explained in the next section.

6.2 Requirements

This section describes the requirements that are needed to actually realize the points of focus. These requirements are divided into four categories: synchronization, working semi-individual, optimized for programming, and communication and awareness.

6.2.1 Synchronization

Since XTC is a form of synchronous collaboration, it needs to synchronize. The following requirements are related to this category.

Multiple participants

Synchronization is only possible when there are multiple participants. Therefore multiple programmers must be able to use XTC. Changes being made must be propagated to every other participant.

Integrity

Integrity is one of the key goals of synchronization in general. Files should be the same across all participants. Changes made should be propagated to all participants. When changes are made at the same time, the resulting environment should still be identical for all participants.

Type of files

Typically, two types of files are encountered: text files and binary files. Code can be dependent on the content of textual and binary files. Changes to both kind of files should be propagated. Not doing so would violate the integrity of the codebase.

File operations

Adding, removing, and renaming of files and folders are all actions performed during programming. They are part of the codebase. Not propagating these changes would violate the integrity of the codebase.

6.2.2 Working semi-individually

The following requirements are related to working semi-individually.

Be optional

XTC should be optional, meaning that programmers can use XTC, but are not required to do so in order to make changes to the source code. Not all programmers will work on related tasks. Also, not all activities can be done while using synchronous collaboration, because they require isolation.

Open and closed files

Not all files in a codebase are relevant for each task. Therefore, not every participant will have the same files opened. Changes should be propagated whether a participant has the file open or not. Being required to have a file open because another participants makes changes to that file reduces the possibilities of working semi-individual.

6.2.3 Optimized for programming

The following requirements are related to optimizing XTC for programming.

IDE integration

The programming activities are performed from within an IDE. Therefore XTC should be integrated into an IDE.

Usable

Changes should be usable right away. This means a change needs not only be propagated and be visible to other participants, it should also be recognized by the programming environment. For example, the auto-complete function should recognize a method which was just made by another programmer. This is needed so there will be no distinction between the origin of the change.

Multi-character operations

Programmers not only type code, they also copy/paste and move code around. Instead of only supporting single-character changes, multiple-characters should also be supported. This means that insertion and deletion of multiple-characters will be propagated to every participant.

Automatic changes

Changes do not only originate directly from programmers. Refactorings or formatting options can be performed by the IDE. Not doing so will violate the integrity of the codebase.

Undo

Mistakes are easily made, undoing them is a function provided by the IDE. This functionality should still be supported. However, the undo functionality should only apply to self-made changes. This is because undo will revert the last taken action. However, this last action could have originated from another programmer. Undoing his change is not intended.

6.2.4 Communication and Awareness

Because participants have no influence of the action of others, there should be a way to communicate with them.

Communication

When working semi-individual, the participants are not always located at the same location. Direct communication is not always possible. In these situations it should be possible to communicate with all participants.

Awareness

When multiple participants make changes, it can become unclear where the change originated from. It can also become unclear who is working on what task and code. In order to raise the awareness, it should become more clear on which code one is working.

6.3 Conclusion

The main difference between Extreme Team Collaboration and existing synchronous collaboration implementations is that XTC is less restrictive. All programmers using XTC may individually make changes to the codebase. At the same time, these changes are synchronized in real-time.

The requirements described in this chapter are a more formal way to describe how XTC can realize this. These requirements also make sure that all changes to the codebase are supported. By doing so XTC will support most programming activities.

The next chapter discusses the actual implementation of Extreme Team Collaboration as a plug-in for Eclipse.

Chapter 7

Implementation

The implementation of XTC consists of a plug-in for Eclipse and a sever component. These are connected by a ToolBus script. Section 7.1 will show how this implementation fullfills most of the requirements described in the previous chapter. The architecture of this implementation will be presented in section 7.2. Last, section 7.3 discusses some details of the implementation.

7.1 Overview

The two points of focus of Extreme Team Collaboration are allowing programmers to work semi-individual and to be optimized for programming. This section explains how the implementation accomplished this.

7.1.1 Sessions

XTC works with sessions. A session is associated with a project in Eclipse, which contains the codebase of a program. When Eclipse is started, XTC does nothing. A programmer has to make the decision to start or join a session. Once in a session, all changes being made to the codebase are synchronized with the other participants in the same session. It is also possible to leave a session. This makes XTC completely optional.

This implements the requirements "multiple participants", "be optional", and "IDE integration".

7.1.2 Intercepting changes

Once in a session, every change to the codebase is intercepted by XTC and send to the server.

All sorts of changes are intercepted. Eclipse provides two ways of listening to these changes. The first is a listener that notifies listeners of changes made in an editor. This are changes a programmer makes by typing in code, copy/pasting code, etc. Changes made by Eclipse itself are also intercepted by this listener. This makes it possible to listen to code generated with auto-complete functions, refactorings, generation of getters and setters, etc. This implements the requirements "multi-character operations", and "automatic changes".

The second listener notifies listeners of changes made to resources. This are folders and files inside a project. This makes it possible to listen to changes on the codebase that do not originate from an editor. This can be things like adding, removing, renaming and moving files and folders. These changes are all intercepted by XTC and send to the server. This implements the requirements "type of files", and "file operations".

Saving a file is also seen as a change to a resource. This saving can be performed by a programmer from inside an editor. It can also be caused by automated refactorings, which might change the content of other files. Eclipse does not know what part of the content of a file changed, only that it changed. XTC reads the complete content of the file and sends it to the server. This implements the requirement "open and closed files".

7.1.3 Applying changes

Changes being send to the server are forwarded to every other participant. These can now apply the change to their copy of the codebase. There are three possible situations.

A textual change originated from an editor. The information inside this change includes the changed text, its offset in the file, and which file the change applies to. XTC will look if the same file is opened inside an editor. If this is the case the text is inserted at the specified offset. The change is now applied.

When a textual change is received, but an editor for that file is not open, the change will be ignored. This does not violate the integrity, as will be explained in section 7.3.6.

A change can also originate from a resource, and not from an editor. In this case the change can be applied on the same resource of the other participant. For example, when a file is moved, the file is also moved for the other participant. When the content changed, the new content is applied to the file, etc. If the resource is also opened inside an editor, it will refresh itself, showing the new content.

This implements the requirement "usable".

7.1.4 Server

All changes are send to the server. The server stores all these changes. This is done so other programmers can join a session after it is started. As long as the same revision of the project is used, one can join the session. All changes already made during the session are send to this new participant, so the codebase of each participant is in exactly the same state.

7.2 Architecture

XTC consists of several parts, as can be seen in figure 7.1. These parts all have their own distinctive function, which will be explained in the next subsections.

7.2.1 Eclipse

Eclipse is an IDE used by programmers to perform their programming activities. Eclipse provides editors to edit source code, automated refactoring, integration

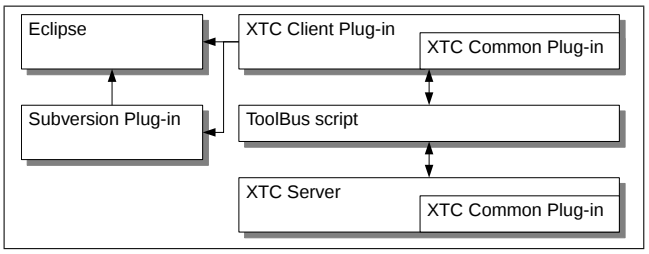

Figure 7.1: XTC Overview

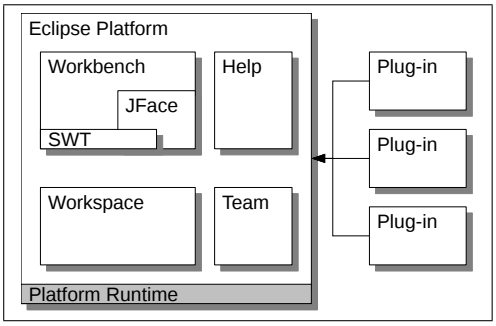

Figure 7.2: Eclipse Platform architecture

with a version control system, and of course the ability to compile, run, test and debug the source code.

This functionality can be extended by using plug-ins. Eclipse provides interfaces and gives access to resources that plug-ins can use. Eclipse is divided into several parts. These are displayed in figure 7.2, and will be explained in the next paragraphs.

Workspace The workspace is the location where projects, source code, and other files are stored. The workspace provides access to these resources to plug-ins.

Workbench The workbench is part of the graphical interface of Eclipse. A workbench consists of different perspectives, views and editors. To be more specific, a perspective defines what kind of information is shown on the screen. For example, the Java perspective shows the workspace view, a console view, and an area where editors are shown. A view shows only a specific piece of information. For example, the console view only shows the output of a running program, the workspace view only shows the projects and files in the workspace. Editors show the content of a file, and can edit these files. A screenshot of Eclipse is shown in figure 7.3.

SWT and JFace SWT is a graphics library, used to create buttons, labels, etc. JFace is a library providing often used graphical interfaces, like input dialogs.

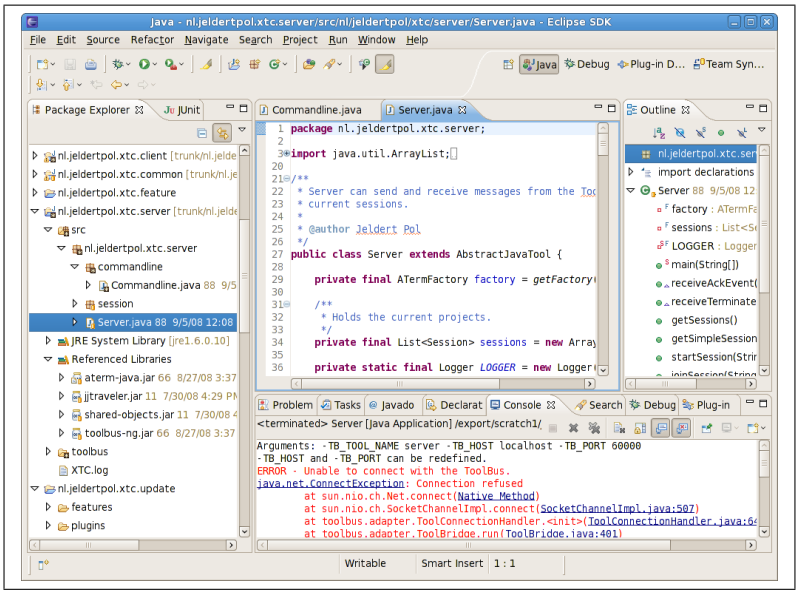

Figure 7.3: Eclipse with Java perspective

Team This part provides support for plug-ins focussing on collaboration, especially for using version control systems.

Plug-in Plug-ins extend the functionality of Eclipse. All previous parts are accessible by plug-ins. For example, a plug-in can create a JFace dialog displaying all projects contained in the workspace.

7.2.2 XTC Client Plug-in

This part is the actual plug-in for Eclipse, and adds the functionality to allow synchronous collaboration from withing Eclipse. This works by listening to any changes the programmer makes, sending it to the server, and applying changes received from the server. Each component of the client plug-in (see figure 7.4) has its own task.

Activator The activator is the only part of the plug-in that Eclipse can access. It is used to start and stop the plug-in, so they can initialize and shutdown properly.

Session and Server A programmer can ask XTC to start a new session, or to join an existing one (see figure 7.5). Once in a session, several listeners are activated. All information coming from this listeners, are passed to the server class, who converts this information, and sends it to the ToolBus. This is done to abstract away ToolBus specific data types and connection information.

Change Listeners Once in a session, XTC starts listening to any changes made to the codebase. There are two kind of listeners. The first one, the

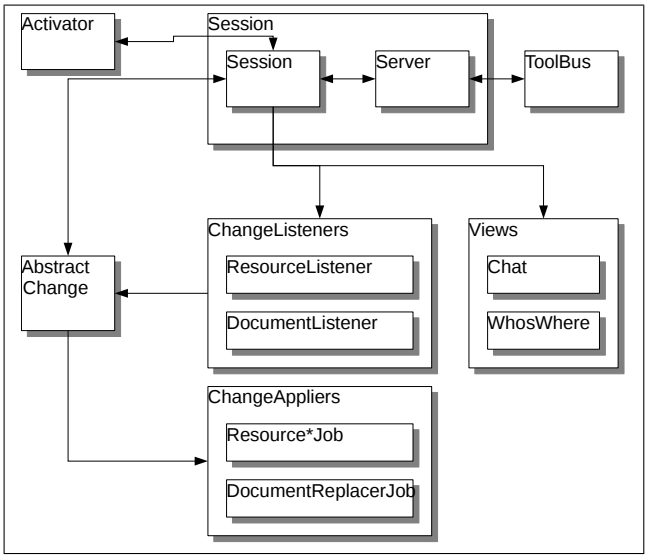

Figure 7.4: XTC Client

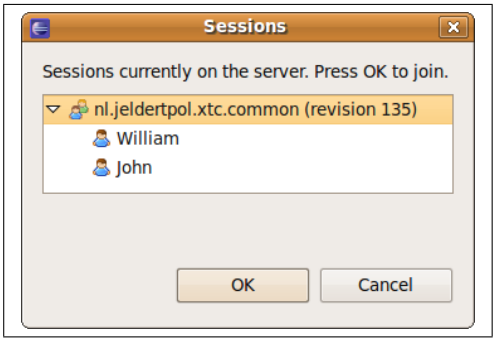

Figure 7.5: Current sessions

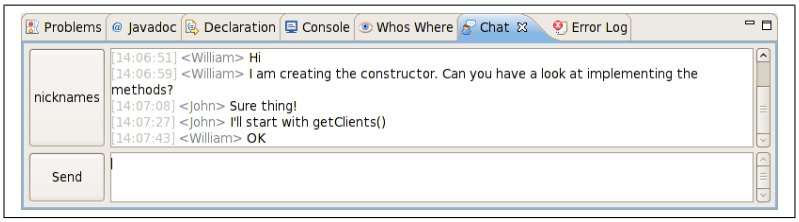

Figure 7.6: Chat view

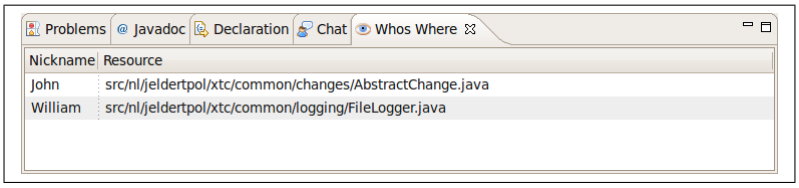

Figure 7.7: WhosWhere view

ResourceListener, listens to any changes to resources made in the workspace. Only relevant information needed to replicate the change is extracted. For example, when moving a file, only the original filename, and the new filename are put in an AbstractChange object. With only this information the change can be applied at the other clients in the session.

The other listener, the DocumentListener, listens to textual changes being made to documents. These are files opened inside an editor. Once a textual change is made, the listener creates an AbstractChange object. Again, only the information needed to reproduce the change is send to the server.

Change Appliers When receiving a change, it needs to be applied. The change is passed to a change applier. This takes the information from the AbstractChange object and reproduces it. Again, there are two kind of appliers. First, there are Resource*Jobs for every kind of change possible to a resource (added, removed, moved, changed content). Second, the DocumentReplacerJob looks if the same document is opened in any of open editors. If so, the change is applied to that editor. If not, the change is ignored (see section 7.3.6).

When applying a change, the codebase is changed. This is detected by the listeners. XTC prevents sending this applied change back to the server again. When applying a change, the change applier first deactivates the listener, then applies the change, and then activates the listener again.

Views Last, there are two views. The chat view allows programmers to chat with each other. Messages are received by every connected client (see figure 7.6).

The WhosWhere view (see figure 7.7) shows each programmer and the file he is currenlty editing. Double clicking on the programmers name initiates a new chat message, with the name of the programmer already filled in. Double clicking the file he is editing will open that same file inside a new editor. When the file is already opened inside an editor, it will be activated. This implements the requirements “communication”, and “awareness”.

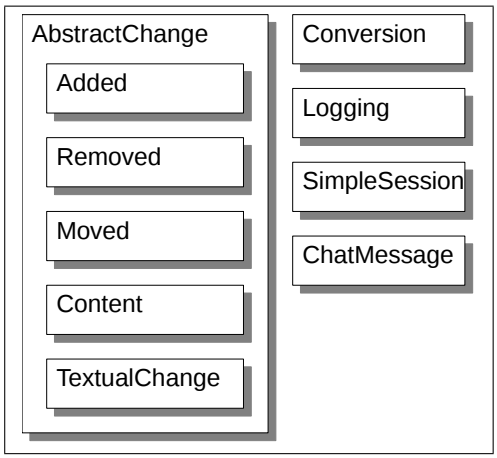

Figure 7.8: XTC Common

7.2.3 XTC Common Plug-in

The common plug-in houses functionality that is needed by both the client and the server. These can be seen in figure 7.8.

Changes The most important functionality is converting the changes made by the programmer to a format that can be send to the server and other clients. Every change is converted into their corresponding AbstractChange object.

Conversion The ToolBus has only limited support for what kind of data can be send and received. It can not send Java objects, so these need to be converted to an array of bytes, which it can handle. The Conversion class does this conversion, and can also convert the array of bytes back to Java objects.

A second task is to convert the content of a file to an array of bytes. In Java, the file content can be accessed with a FileStream. Only, such a stream cannot be converted directly into a format the ToolBus understands. Therefore, each byte in the file is read and put in an array. This is different from the object conversion, where the complete object is converted. Also, it can write this array of bytes back into a file.

Logging Logging is used to capture interesting events during the usage of XTC, and write them to a log file. Since logging is used on both the server and the client, the class handling logging is placed in the common plug-in.

SimpleSession A SimpleSession is a simple representation of a session. The server stores all changes made during a session. A SimpleSession does not have this information. It is used to represent which sessions are active and who is connected to this sessions.

Chat The messages being send via chat are wrapped in a ChatMessage object. This contains the name of the client, and the message.

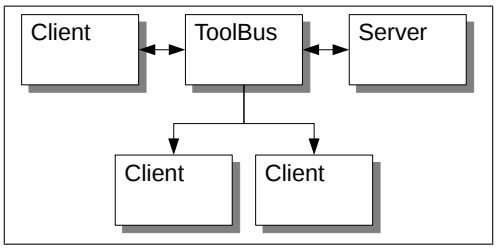

Figure 7.9: ToolBus script

```
// Send a change
let
    Project : str,
    Change : blob,
    Nickname : str,
    Success : bool
in
    rec-request(Client, sendChange(Project?, Change?, Nickname?))
    . snd-eval(Server, sendChange(Project, Change, Nickname))
    . rec-value(Server, sendChange(Success?))
    . snd-response(Client, sendChange(Success))
    . snd-note(change(Project, Change, Nickname))
endlet
```

Figure 7.10: Excerpt of ToolBus script

7.2.4 ToolBus script

This central part handles all communication between the clients and the server. What this means is that the clients and server connect to the ToolBus instance running this script. This script only contains the logic needed for communication between the clients and server. This has the advantage that both clients and server do not need to have any communication logic. They only need to know the ToolBus, send messages to it, and handle messages received from it. The ToolBus handles the forwarding of these messages to the server and other clients. This is illustrated in figure 7.9.

An excerpt of the script is shown in figure 7.10. The first 4 indented lines define the variables used. The last 5 indented lines mean:

1. Wait for a client to send a change. The client sends the name of the project, the change, and its nickname.
2. Forward the change to the server.
3. Receive an answer from the server, if the change is received successfully.
4. Send a response to the client that the change was received.
5. Send a note to all other clients. They will now receive the change as well. This part makes sure that the change is not send back to the client the

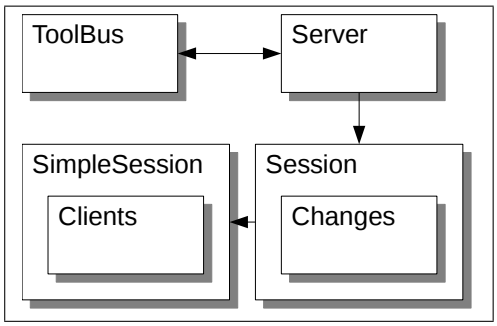

Figure 7.11: XTC Server

change originated from.

7.2.5 XTC Server

The server receives all changes from the clients via the ToolBus. It converts the array of bytes back to AbstractChange objects, and stores them with their associated session. This is shown in figure 7.11.

When a new client connects to an existing session, the server send all received changes back to this new client. This way, new clients will be in the same state as the other clients.

Also, when a client detects that an error occurred when applying a received change it can convert to its last working state, and request all changes made from that moment.

7.3 Details

This section will explain some details and implementation difficulties of XTC.

7.3.1 Version control

XTC only works with projects that are under version control. When starting a session, XTC asks the plug-in responsible for version control for the revision of the project. Now a session is created. Other programmers can now join this session. However, they need to have the same revision of the project. This is done to make sure all programmers start with the exact same codebase (integrity), without needing to send the complete codebase to every participant, or having to compare the hash of each file.

There can be files in a project that are not under version control. This can be files that are automatically created when compiling the code. Changes to these files will be ignored, since there is no guarantee that the other participants will have these files as well. Also, compiled code may be different between participants when they use a different environment (Windows/Linux, 32-bit/64-bit, etc.).

Modifications to the codebase which are not yet synchronized with the version control system are considered as new changes. When starting a session all modified files are send to the server. When other programmers join this session

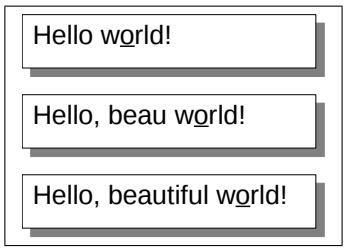

Figure 7.12: Edit cursor of programmer B (underlined)

they instantly receive these modifications. This makes sure the state of the codebase is the same for every participant.

7.3.2 Missing changes

Occasionally XTC misses a change made by a programmer. When this happens the change is applied locally, but is not send to the server. This results in the codebase being in different states between programmers.

The cause of this is the way XTC applies changes. As discussed in the previous section, XTC uses listeners to intercept changes. When applying changes, these listeners need to be removed. If this is not done, applying the change is detected by a listener and send to the server again. This causes an infinite loop. By removing the listeners, the change can be applied, without detecting it and sending it to the server again.

In the short timeframe these listeners are removed a second change can be received from the server. This is placed in the queue of changes, and applied as soon as the first one is processed. By doing so, no changes of other programmers can be missed.

The problem of missing changes happens when a change is being applied, and the programmer himself makes a changes. This change can not be detected by XTC, since the listeners are removed.

XTC provides build in error recovery. This will not prevent this situation, but tries to recover from it, so the codebase of the programmers are synchronized again.

7.3.3 Error recovery (editor)

Textual changes are applied in the editor at the offset specified. If this offset in the editor contains the same characters as the originating editor is not checked. This leads to a race condition, meaning who comes first, is served first. One might think this will violate the integrity of files almost instantly when simultaneously editing the file. This is not the case.

The AbstractEditor of Eclipse turns out to handle with this situations quite well. When text is inserted in the editor, the edit cursor is forwarded. For example, an editor contains “Hello world!”. Programmer A wants to change this to “Hello, beautiful world!”, while the edit cursor of programmer B is located at offset 7 (the letter “o” of “world”). Programmer A types in “, beautiful”. Instead of leaving the edit cursor of B at offset 7, it is moved forward, to offset 18, which is again the letter “o” of “world”. This is illustrated in figure 7.12.

When programmer B is typing at the same time as programmer A, the text is inserted at the location he intended. This reduces the need to check for integrity.

While testing XTC, the editors did get out of sync occasionally. However, when one of the programmers saves the file, the complete content is send to the other programmer. There the editor is refreshed, and the editors are in sync again.

7.3.4 Error recovery (resources)

Changes to resources can also produce errors. For example, a file is deleted, but xtc missed this. The file is now only deleted at the programmer who deleted it, but every other programmer still has the file. One of those programmers can change the file and save it. XTC will then send the content to the programmer who doesn't have the file. An error occurs at the programmer who doesn't have the file.

In case a situation like this occurs, XTC tries to recover from it. This is done in a simple way. First, XTC leaves the current session. It then makes a local copy of the codebase, as backup. The original codebase is now reverted to the state it was in when the session was joined. This is done by asking Subversion to revert the project in Eclipse. This also reverts the deleted file, so it is present again. Now the session is joined again. Every change so far is applied. Since the change to delete the file was never send to the server, it will not be deleted. The change with the modified content was send to the server, and can now be applied, because the file is now present. This should lead to a synchronized codebase again. If for any reason XTC fails to apply a change again, it gives up. It is then up to the participant to solve the problem.

7.3.5 Unwanted changes

Sometimes changes are made to the codebase which are not wanted by everyone. Imagine a file (class) is deleted, because it is not used by any of the other code. A programmer may have created this file, but decided to completely implement the class before using it. Instead of preventing this deletion (performed by another programmer), XTC allows this, and applies the change anyway. This was decided because XTC is not in the position to decide whether the action was intended, or done by accident.

As a side note, in situations like this, it seems like a good moment to communicate with each other. If one programmer wants to keep the file, but the other doesn't know this, it is time to make this clear.

Still, no work is lost, since the deletion of the file is placed in the undo-buffer of Eclipse, so it can be recovered again. When the file is already under version control it is even less of an issue, since it can be restored even more easily.

7.3.6 Ignoring textual changes

As said earlier, all changes are applied when they are received by a client, except for textual changes when the file is not open. They are ignored because Eclipse does not provide the ability to insert text directly into files. It can only be done

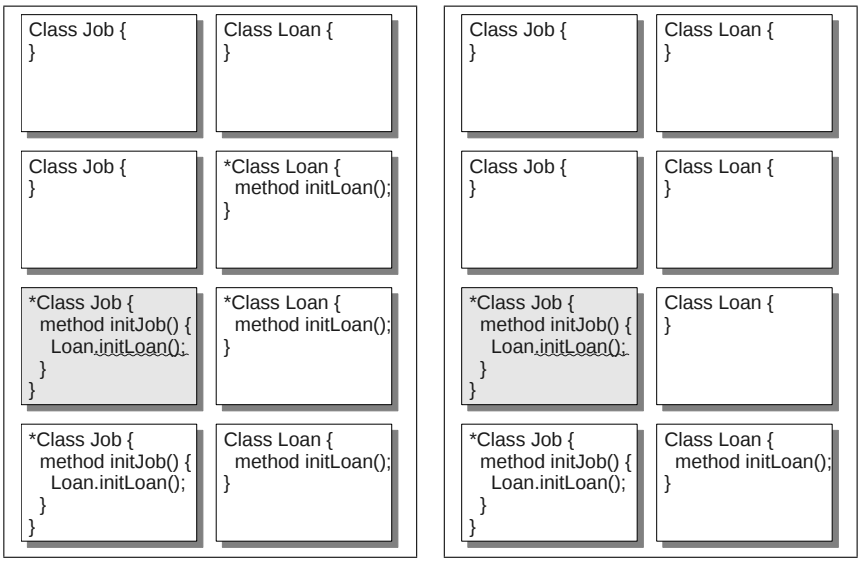

Figure 7.13: Codebase for programmer A (left) and B (right).

via an editor. However, ignoring these changes does not violate the integrity of the codebase.

Suppose two programmers are in the same session. There are two files, class Job, and class Loan (first row in figure 7.13). Programmer A has both files open, B only has Job open. Now programmer A is trying to initialize class Loan inside class Job. He first creates the initLoan method in class Loan. This change is not applied for programmer B, since he doesn't have this file open (second row).

Now programmer A tries to call this new method inside Job. He can't, because Eclipse doesn't recognize this new method. Eclipse can only recognize methods in saved files, or in opened (and unsaved) files if the new method is in that same file. Since it's neither, Eclipse produces an error. This error is also seen by programmer B, since he did have class Job open (third row). Programmer A needs to save class Loan first, for Eclipse to recognize the new method. This change is applied for programmer B (it is a resource change, which is always applied). Now both A and B do not get the error anymore, because Eclipse recognizes the new method in Loan for both programmers (fourth row). Both programmers now have the same codebase again.

Opening Loan What happens if programmer B decides to open class Loan, after A made changes to it, but before it is saved? In this situation the state of the file is different between the programmers. XTC can solve this. When a file is opened XTC asks the server if there are any textual changes for this file since it was last saved. The server can know this, since it receives all changes. The textual changes are sent to programmer B, which can now be applied, since he has an editor open for class Loan. Both files are now in the same state again.

Final note Textual changes of code do not affect the running behavior of the code until these changes are saved. Ignoring unsaved textual changes will thus

not violate the integrity of running the code. Also, features like auto-complete and parsing only recognize new pieces of code from an active editor, or when the changes are saved. Programmers must thus frequently save files before these changes can be used.

7.3.7 Pause

To further improve the ability to work semi-individual, XTC has a pause/resume feature. As chapter 3 showed, synchronous collaboration has a negative impact on testing, debugging and fixing bugs. In order to reduce this impact a pause feature has been integrated in XTC.

When in pause, all incoming changes are ignored, and placed in a buffer instead. This gives a programmer some isolation, allowing him to run tests, start debugging etc., with an unchanging codebase.

When resuming, all changes placed inside the buffer are applied. New incoming changes are then applied as usual. Xtc resumes when being asked, and when the programmer himself makes a change. This means all buffered changes are first applied, then the change of the programmer is applied and send to the server. This is done to make sure the codebase is synchronized again, before self-made changes are applied and send to the server.

7.3.8 Awareness

Some of the CRTE's from chapter 5 highlight or mark changes being made by other programmers. Doing so increases awareness significantly. XTC does not do this. The reason for this is that the DocumentListener of XTC listens to changes to the AbstractEditor. This is the base editor for textual documents. Applying textual changes also work by calling methods of this AbstractEditor. The advantage of doing so is that XTC works with virtually all editors, including editors for Java, configuration files, XML, etc. A drawback is that this AbstractEditor does not have support for highlighting or marking content of the editor.

7.3.9 Undo

The only requirement not met so far is "undo". Instead of only being able to undo self-made changes, every change can be made undone. Changes are applied by calling methods provided by Eclipse. This causes the change to be placed in the undo-buffer of Eclipse. This means that changes made by other programmers can also be undone.

This is not solved in XTC at the moment, even though it does have a big impact on the usability of XTC. If two programmers are editing the same file, and one programmer tries to undo his last change, he could undo a change made by the other programmer, if that change was performed after the change of the first programmer.

7.3.10 Statistics

Table 7.1 displays some statistics of the implementation. This contains only information about the Java files written, not for any libraries used. Also, lines

	Client	Common	Server	ToolBus script
Classes	51	13	3	N/A
Methods	171	42	20	N/A
McCabe cyclomatic complexity (avg)	1.848	1.174	1.864	N/A
Lines of code	2642	331	324	194

Table 7.1: Implementation statistics.

of code refers to a line containing code, ignoring empty lines and documentation.

Note that the ToolBus script does not contain any classes or methods. Rather, it defines 2 processes, containing 13 possible communication possibilities.

7.4 Requirements

This section will describe how XTC implements the requirements from chapter 6.

Multiple participants The server allows multiple session. Each session allows multiple participants to be present.

Integrity When starting a session the revision of the codebase is requested. When another programmer joins this session, he must have the same revision. Any changes made so far in the session are applied. New changes are intercepted by the client, and forwarded to the server. Furthermore, there is a basic form of error recovery.

Type of files The listeners of XTC intercept changes to all files, textual and binary.

File operations ResourceListener intercepts all file operations, like renaming and removing files, adding folders etc.

Be optional Sessions can be started, joined, and left at any time. This allows programmers to decide whether to use XTC or not.

Open and closed files DocumentListener intercepts all textual changes to opened files. Other kind of changes to opened and closed files are intercepted by ResourceListener.

IDE integration XTC is a plug-in for Eclipse, making it fully integrated.

Usable All changes are instantly applied. Eclipse sees these changes.

Multi-character operations DocumentListener intercepts multi-character operations, like copy/paste, refactoring and formatting of code.

Automatic changes Textual changes are intercepted by DocumentListener, ResourceListener intercepts changes to files when they are saved. It does not matter what the origin of the change is.

Undo Undoing only self-made changes is not implemented at this moment.

Communication XTC provides an integrated chat.

Awareness WhosWhere displays who is editing which file. More advanced forms to increase awareness, like highlighting changes in the editor, are not implemented at this moment.

7.5 Conclusion

XTC fullfills most of the requirements from chapter 6. The only requirement not met is undoing only self-made changes. Also, there is only limited support for raising awareness.

There are not a lot of screenshots of XTC in action in this chapter. The reason for this is that XTC works silently in the background. Changes are intercepted and applied without user interference.

Chapter 8

Validation

Because of resource constrains (time and people), it was hard to validate the advantages and disadvantages of XTC in a realistic situation. Therefore, no hard claims can be made about any advantages and disadvantages. However, XTC was used by a few programmers at the CWI. The information gathered from this is used as a way to validate the implementation of XTC, not to validate any advantages and disadvantages. This can be found in section 8.1.

Section 8.2 will discuss some realistic situations when XTC will add value to the programmers.

8.1 Validating the implementation

XTC needs to synchronize all changes made by programmers. The implementation of XTC tries to accomplish this. This was validated by programmers at CWI.

8.1.1 What was tested

Two programmers worked on the same project while using XTC. Before they started, the codebase of both programmers was identical. Standard diff-tools under UNIX where used to verify this. Now both programmers joined the same session, and started making changes to the codebase. At the end on the session both programmers disconnected from eachother. The codebase at both computers was still identical to eachother, and contained all changes made by each programmer. This proves that XTC synchronized the codebase while programmers where making changes to it.

The following list describes which actions where performed by one programmer, and what happened in Eclipse at the other programmer:

- Creating a new file.
  - The newly created file was send to the other programmer.
- Renaming a file.
  - The same file was renamed at the other programmer.

- Deleting a file.
  - The deleted file was deleted at the other programmer.
- Creating a new folder.
  - The newly created folder was send to the other programmer.
- Renaming a folder (with content).
  - The same folder was renamed at the other programmer.
- Deleting a folder (with content).
  - The deleted folder (with content) was deleted at the other programmer.
- Importing a single file.
  - The imported file was send to the other programmer.
- Importing multiple files.
  - The imported files were send to the other programmer.
- Importing a folder (with content).
  - The imported folder (with content) was send to the other programmer.
- Modifying (replacing) binary files.
  - The new binary file was send to the other programmer.
- Typing in text file while other programmer has same file open.
  - The inserted text was send and placed at the correct position in the editor of the other programmer.
- Two programmers typing in the same text file.
  - The inserted text of both programmers is correctly placed at the right position in the editors of both programmers.
- Typing in text file while other programmer has same file closed.
  - The inserted text was send to the other programmer, but was ignored.
- Typing in text file while other programmer has same file closed, but opened it after typing.
  - The inserted text was send to the other programmer, but was ignored. After opening the file the new text was inserted in the editor.

All these actions where performed vice versa, multiple times, in diffrent order, and even simultaneous, meaning both programmers made changes to the codebase at the same time.

A similar, but smaller test also showed that XTC behaved correctly while three programmers where making simultaneous changes to the codebase.

8.1.2 Issue's with implementation

During this validation, some issue's where found with the implementation. First of all it became clear that most programs are divided into multiple projects. Even XTC itself is divided in 4 projects (client, common, server and ToolBus). When programming, it is not uncommon to change files in more that one project. Since the current implementation of XTC only works with one project, only changes to that particular project are synchronized. This greatly reduces the usability of XTC.

Another result is that a project may contain a lot of files which are not under version control. The compiler compiles the source code, and puts the compiled files in a "build", "bin", or similar named folder. The content of these folders are not under version control. A better way of handling these kind of folders is needed.

A last known issue with the current implementation is related to changing a textual file, but not saving it. Every textual change made in an editor is send to the server, and stored there. When another programmer opens the same file inside an editor, all these intermediate changes are send to him, so the content of the editor is the same for both programmers. When the programmers close the editor without saving the file, the intermediate changes are still present on the server. Every time the file is opened these changes are applied to the editor again. Not until a programmer actually saves the file (opening the file, undoing the received intermediate changes, saving the file) this unwanted behavior is resolved. This is because when saving a file the complete content of that file is send to the server, overruling the intermediate changes. However, the server is unaware of the editor being closed, and the programmer trying to discard the changes made. It is clear this behavior should be fixed in a future version of XTC.

8.2 When to use XTC

This section will describe situations where XTC is a usefull asset for the programmers. For each programming task, from chapter 2, a situation is created.

8.2.1 Helping another programmer

From time to time a programmer may get stuck. He may simply be calling the wrong methods, receive errors, or the run-time behavior of their code is not as expected. Another programmer can use XTC to connect to the programmer needing help. XTC makes sure both programmers have an identical codebase after connecting to eachother. The programmers can now help eachother, since they are both looking at the exact same code. This means one can easily spot errors in the code. This can now be fixed.

Since there are no constraints to the location of the programmers, they can help eachother while sitting in the same room, another room, or on the other side of the world.

8.2.2 Refactoring a large project

During the lifecycle of a program, it is sometimes needed to refactor large parts of it. Classic approaches include branching the code, performing the refactoring, and synchronizing the code with the trunk again. With XTC, this can still be done. A major differece however is that this kind of large refactorings can now be performed with multiple programmers at the same time.

Lets say the API of two classes need some serious changes. One programmer can change the first class, while the other changes the second class. However, since calls to these APIs also need to be changes, a lot of changes may be created in the codebase. More important, code using this API could also need large changes. In total, a refactoring like this changes a lot of code in the codebase. This makes it hard for both programmers to stay synchronized, even though they are working in a different branch.

With XTC, both programmers can again change one API each. Every change this inflicts, and even additional changes to other code, are instantly synchronized. At every moment of this process, the codebase for both programmers is identical. Therefore, there cannot be any synchronization problems.

8.2.3 Improving documentation

Sometimes large pieces of code lack documentation. Not only the internal documentation, but also the external documentation may be missing. One or more programmers may be given the task to improve the documentation.

A logical part to start with, is by documenting the API of a class. It can be clear to both programmers in which class to start, so they won't document the same code twice. The public methods however also call private methods, or methods in other classes. These may all be lacking documentation, so they need to be documented as well. Suddenly, it's not so clear anymore which methods are documented by which programmer. A solution for this is to use a lot of communication, or synchronizing often, to see it the other programmer already commented a piece of code.

By using XTC, each programmer always has a clear view on what has been documented already, and what still has to be done. Excessive communication and manual synchronizing is not needed anymore. It even impossible to improve eachothers documentation, before it is committed to the repository.

8.2.4 Distributed pair programming

With XTC it is even possible to perform distributed pair programming. One of the programmers can be the driver, while the other observer. Changing roles is simple. The driver just needs to stop typing, and let the observer take control.

The current implementation of XTC however is not ideal for distributed pair programming. Following the driver through the code needs to be done manally, which is far from ideal. Also, an observer normally does not have direct access to a keyboard, because the driver is using it. With XTC, both programmers have a keyboard, and currently there is no way to disallow an observer from using it.

8.3 Future uses

With some adaptions, XTC could also be made suitable for other uses.

8.3.1 Gathering information

Since each change being made by a programmer is instantly send to the server, a lot of information can be gathered. Imagine knowing how much a programmer types during a day, and comparing this to what he actually commits. With this information, the efficiency of a programmer can be measured.

This information retrieval can fairly easily be realized. The server only needs to be expanded for extracting this information and storing it for future processing.

8.3.2 Playback actions

The server of XTC already has the ability to send multiple changes to a client. Imagine this can be done with a delay between each change. This allows XTC to playback actions of a programmer.

The only adaption XTC needs for this, is the ability to send changes with an interval, instead of sending all performed changes on connect.

8.4 Conclusion

The implementation of XTC was verified. This revealed some issue's which, when solved, improve the implementation and usefullness of XTC. Overall the implementation works as expected.

Advantages or disadvantages of using XTC could not be verified in this thesis, because of resource contraints. In order to verify these advantages and disadvantages, XTC needs to be used by real programmers, for a longer period of time.

The situations described show that XTC can be very usefull, but only when programmers are working closely together on the same, or related task.

With adaptions, XTC could also be used for other uses, like gathering information about programmers.

Chapter 9

Conclusion

Using synchronous collaboration has multiple advantages over using asynchronous collaboration. Current plug-ins for Eclipse support synchronous collaboration, but are very restrictive. Not all programming activities are supported. With some plug-ins only one file can be edited at a time, while other plug-ins only allow one programmer to make changes to the codebase.

Extreme Team Collaboration (XTC) on the other hand is less restrictive. It allows multiple programmers to edit different files simultaneously. All programming activities that change code are supported by XTC. This is accomplished by a tight integration with the Eclipse IDE.

Xtc is less usefull for programming activities that require isolation (like debugging). This is because it cannot be guaranteed that no changes will be made to the codebase while using XTC.

The implementation of XTC works as expected, and is suitable for other uses as well after making the required adaptions. The advantages and disadvantages of using XTC could not be verified in this thesis. More research in the use of XTC is required to accomplish this.

The current implementation and future information about XTC can be found at [XTC].

Bibliography

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[Bah02] Prashant Baheti. Assessing distributed pair programming. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 50–51, New York, NY, USA, 2002. ACM.

[BJ75] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1975.

[CdSH+04] Li-Te Cheng, Cleidson R.B. de Souza, Susanne Hupfer, John Patterson, and Steven Ross. Building collaboration into ides. *Queue*, 1(9):40–50, 2004.

[CSFP04] B. Collins-Sussman, B.W. Fitzpatrick, and C.M. Pilato. *Version Control with Subversion*. O'Reilly Media, Inc., 2004.

[DB92] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, New York, NY, USA, 1992. ACM.

[DSP] Docshare plugin. `http://wiki.eclipse.org/DocShare_Plugin`. Accessed August 18th, 2008.

[HRGW04] Chih-Wei Ho, Somik Raha, Edward Gehringer, and Laurie Williams. Sangam: a distributed pair programming plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 73–77, New York, NY, USA, 2004. ACM.

[PEP] Pep. `http://pep-pp.sourceforge.net/`. Accessed August 18th, 2008.

[RTS] Rt shared editing. `http://wiki.eclipse.org/RT_Shared_Editing`. Accessed August 18th, 2008.

[San] Sangam. `http://sangam.sourceforge.net/`. Accessed August 18th, 2008.

[Sar] Saros. `http://dpp.sourceforge.net/`. Accessed August 18th, 2008.

[Xec] Xeclip. `http://xeclip.sourceforge.net/`. Accessed August 18th, 2008.

[XPa] Xpairtise. `http://xpairtise.sourceforge.net/`. Accessed August 18th, 2008.

[XTC] XTC website. `http://www.jeldertpol.nl/software/xtc/`. Accessed February 24th, 2009.

[Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

Appendix A

The problem of file-sharing

Collins-Sussman et al. describe the problem of file-sharing [CSFP04], which applies to asynchronous collaboration. Suppose two programmers have a copy of the same file. They both make modifications to this file. Programmer A is the first one to send his changes back into the repository. Moments later, programmer B sends his changes to the repository. What happens is that B does not receive the changes made by A. Even worse, the changes of A seem to be lost, since the file in the repository is being overwritten by the copy of B. The changes are not really lost. After all, they are in version control. But effectively they are lost, since B never saw them and so is unaware of them. This is known as the problem of file-sharing (figure A.1).

A.1 The Lock-Modify-Unlock solution

A solution to this problem is to lock a file before editing can start. Now programmer A requests a lock on the file, and makes his changes. Programmer B also tries to get a lock, but fails, since A already has a lock. Now A can make his changes, commit it to the repository, and release the lock. Now B can request a lock, receive the latest version, and start editing the file. This is illustrated in figure A.2.

Drawbacks This approach solves the problem of file-sharing, but it also has some drawbacks. Perhaps the most important one is that locking prevents collaboration: programmers are just taking turns.

Also, locking can create a false sense of security. Two files may be dependent on each other. Now A and B each lock one of these files and start editing. They think they are safe, since no other persons can edit the file. What happens instead is that a dependency conflict may be created (see section 3.2.5), without being aware of it. The version control system could not prevent this. Also by feeling safe, programmers may stop communicating about these changes.

Programmer A could also forget to release the lock. His colleague B is waiting on the release of this file, so can't do anything in the meantime. Eventually, he has to go to A, or to a repository administrator, to release the lock. This causes a lot of delay.

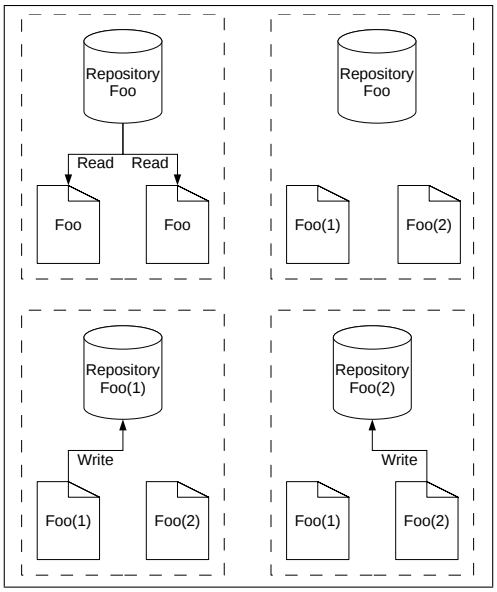

Figure A.1: The problem of file-sharing

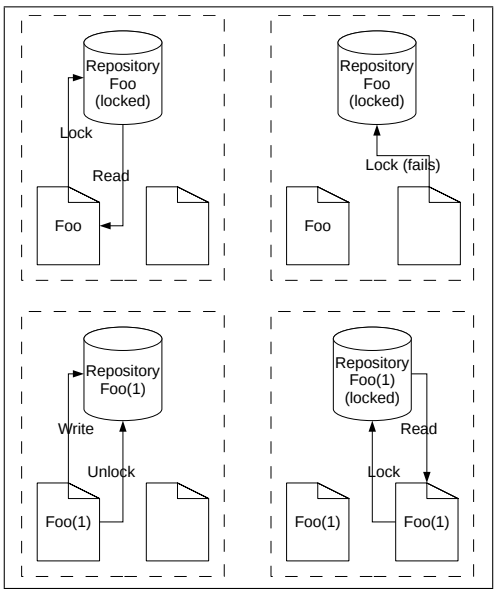

Figure A.2: Lock-Modify-Unlock

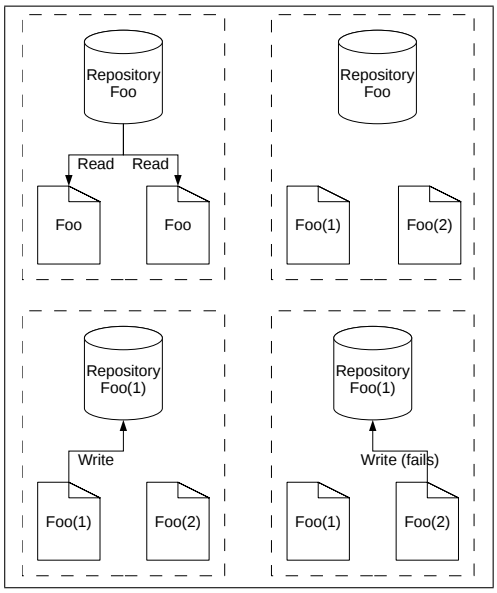

Figure A.3: Copy-Modify-Merge (step 1)

And finally, changes being made may not conflict at all. Perhaps A changes something at the beginning of the file, while B wants to change something at the end of the file. A good merge strategy may have allowed this, without having to lock the file. So again, this approach causes a delay.

A.2 The Copy-Modify-Merge solution

Another solution to this problem of file-sharing is the copy-modify-merge solution. Instead of locking, it allows a file to be edited by multiple persons at the same time. Now both programmer A and B can make changes to their local copy of a file. Programmer A can commit his changes to the repository without a problem. When B is done, and tries to commit his file, the repository notifies him that his file is out-of-date. This means the file in the repository is newer that the local copy B started with.

To solve this, B requests an update of the file. Instead of overwriting his working copy, B is given the change to merge the two files. When the changes do not overlap this merge can be done automatically. Now the changes of A will be present in the working copy of B. Now B is ready to commit his file. Programmer A can now update his file again to receive the changes made by B. Now both programmers have a file each other's changes (see figures A.3 and A.4).

Drawbacks Even though this solves the problem of file-sharing, and the drawbacks associated with locking, it still has some issues, especially when changes do conflict.

When programmer A and B change the same part of the file there might be a conflict. This means that the version control system does not know how to

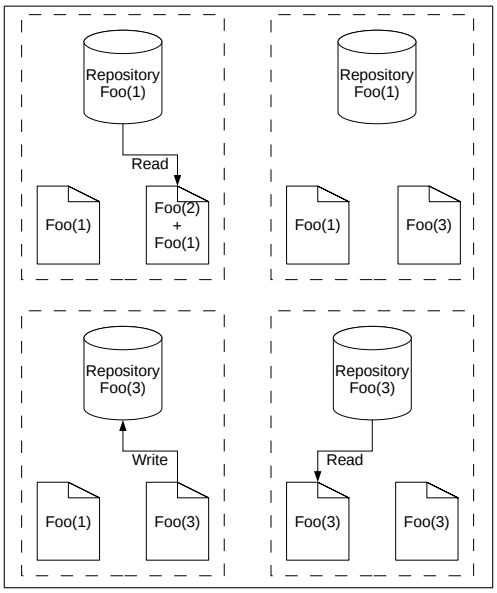

Figure A.4: Copy-Modify-Merge (step 2)

merge the files. Humans are needed to make this kind of intelligent decisions. Therefore it presents both versions to the person who initiated the merge. This person has to manually select how the merge should be performed. These merge conflicts do happen occasionally. Code is not only edited manually, but also automatically, for instance with automated refactorings. Changing the name of a method also changes all calls to this renamed method in other parts of the code. These changes can also cause merge conflicts.

Some version control systems, like Subversion, still support locking files. This is done because some types of files cannot be merged easily. For instance, it is preferred to edit binary files (like audio or graphic files) in a turn-based fashion. This is the reason why locking is still supported. This is not a drawback, since it improves the usability of the system, but by doing so the drawbacks of locking can also apply to this solution.

A.3 CRTE and Combination

The problem of file-sharing only occurs when using asynchronous collaboration. When using synchronous collaboration all changes are synchronized in real-time. This means the files of every participant are always identical. Therefore, there is no need to merge them. Because CRTE's are a form of synchronous collaboration, the problem of file-sharing does not apply to them.

When using a combination of both a version control system and a CRTE, the problem only occurs when synchronizing again with the repository. This means when using the CRTE, all changes are synchronized in real-time. There is no need to merge files, since they are already identical. However, they are different from the files in the repository. Synchronizing with the repository is a form of asynchronous collaboration, and the problem of file-sharing may occur again.