UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE –
EUROPEAN MASTER IN SOFTWARE ENGINEERING

Language Agnostic Proactive Early Detection of Conflicts: A Prototype

Master Thesis

Bojan Marjanovic

Madrid, October 2018

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingeniería del Software – European Master in Software Engineering
Thesis Title: Language Proactive Early Detection of Conflicts: A Prototype
Thesis no: EMSE-2018-8
October 2018

Author: Bojan Marjanovic
Bachelor with Honours in Electrical and Computer Engineering
University of Novi Sad

Supervisor:

Xavier Ferré Grau
Doctor en Informática
Universidad Politécnica de Madrid

Lenguajes y Sistemas Informáticos e
Ingeniería de Software
Escuela Técnica Superior de Ingenieros
Informáticos
Universidad Politécnica de Madrid

Co-supervisor:

Andrea A. Janes
Dottorato di ricerca in Ingegneria del
Software
Università di Klagenfurt

Faculty of Computer Science
Free University of Bozen-Bolzano

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

Abstract

Nowadays, nearly all software systems are developed by programmers working in a collaborative software development. Against this background, recent software development trends produce a version of system that unites the artifacts of simultaneous change. Usually, each developer works on their private versions of the system to secure firmness during programming. Still, this prevents them from understanding what is happening in different workspaces where their coworkers act. This can considerably impact their own work. Accordingly, conflicts occur owing to parallel work, and they become even more complicated as changes grow without being noted. Therefore, the later conflicts are identified, it is more complicated to address them, since more code has to be changed. Moreover, a conflict detected late is in most cases harder to fix because the changes that created it are no longer "recent" from a developers' perception.

Today's approaches to solving this issue frequently force developers to amend a coworkers' work in order to resolve a conflict – without even reaching the coworker. This way of resolving conflicts, periodically leads to inconsistent code, social conflicts, and utmost, anger and frustration inside the team. Moreover, conflict detection can be challenging for developers to do it on their own, since some code parts may have a complex structure, such as polymorphism, encapsulation, etc.

The main issue of merging conflicts during collaborative development is that an important merge may lead to crucial software faults. After the industry and research got involved in the problem it was acclaimed that an early detected conflict is easier to resolve than the conflict encountered at merge or in production. Being informed about conflict as early as possible allows developers to prepare themselves, making a better judgment about how to solve it. Nonetheless, every known approach demands interaction with developer to detect conflicts, and it may lead to overbearing them with notifications – thus, making it even more problematic.

Therefore, the aim of this study is to propose a solution on behalf of developers, i.e. a tool that helps in early conflict detection. With that in mind, this research turned to studying similar systems of early conflict detection in order to get a better understanding of how they work and what are their downsides. Furthermore, it combined collaborative development, version control systems and exploring future development states of software, also known as speculative analysis, in order to get an accurate account of arising conflicts and provide detailed information about them. As a result, the concept of awareness has been proposed to help developers in detecting conflicts before they occur. In terms of this study, 'awareness' is understood as "a comprehension of all activities of others that gives a background for your action".

The approach this study takes is based on two types of awareness: awareness of change and awareness of presence. The idea is to provide accurate information about possible con-

flicts between members in the collaborative software development team. In other words, the main goal of dependency-based awareness tools is to decrease the number of notifications, so a developer can be focused on more applicable ones. For example, rather than creating notifications for every change in each file, there are some tools that inform developers only about simultaneous changes of files that are related to the files changed by the developer.

The tool developed in this study follows all existing workspaces and changes inside them by continually checking and testing them inside the IDE. It enables earlier conflicts solving while developers still have 'fresh minds', and allows for an easier detection of every subsequent conflict. An empirical evaluation validated that this approach actually helps in early detection of conflicts and avoids overwhelming developers with notifications as compared to existing solutions.

Overall, in spite of program errors, it is mainly inexpensive and painless to spot and resolve conflicts earlier, before they reach the production and the relevant changes vanish from the developers' minds'. At present, this knowledge is not easily accessible to developers. The approach presented in this study prepares and serves information about the possible conflicts in a continual and precise way. The aim of this information is to empower developers for better decisions about when and how to share their changes, while at the same time reducing their overload.

Acknowledgements

First and foremost, I would like to thank Dr Janes, Dr Robbes and Dr Ferre Grau for their support in this thesis process. Without their guidance and constant help this thesis would not have been done.

I would like to thank my friends and colleagues from Schneider Electric DMS NS LLC for their marvellous collaboration. Without their participation and support the evaluation would not be possible. You were always eager to help me.

In addition, thank you, my family, for your love and unquestioning support.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Language-agnostic programing [1] or even scripting (also known as language-independent, language-natural, or cross-language) represents a software development paradigm where a specific programming language has been chosen owing to the fact that it is suitable for a specific task (considering all possible factors, from performance, developers' skills, even including ecosystem, etc.) and not strictly because of the stack used by developers.

For example, a language agnostic development team that works mainly in C#, might change to Ruby or Python for some specific tasks, in case where Ruby or Python are more relevant than C#.

The word "agnostic" [2] comes from the ancient Greek, which means "do not know". And for that reason, something that is "language agnostic" does not need "to know" about programming languages, or to know them, but it is synonymous with *language independent*. Overall, concepts that would be language agnostic include algorithms, or Agile approach, or even a runtime library that supports bindings between multiple programming languages.

"Cross-language" [3][4] has been also used in programming and scripting. Today, it is common to see this approach, since it describes a program where two or more programming languages have to be combined into a program code besides the main programming language chosen to write in. It could mean just adding a script as a reference, to be run when necessary. Typically, the code will be executed with specific Language-Independent Virtual Machines such as JVM, or Object Models such as COM to cooperate with each other, or by choosing languages that work better together.

Furthermore, as a software developer, one should not be limited to a particular set of programming languages. Ideally, one should be able to learn new paradigms and languages, (as fast as one can) that are relevant to one's needs and that are going to be used regularly. Particularly, software engineering is not a straightforward code writing, it is more about architecture and design of solutions, i.e. it implies a correct architecture and well-designed solutions.

The sooner a conflict is detected it is easier to solve it – is the core principle of collaborative workspace development [5]. Generally, workspace awareness looks to give users enough information on relevant parallel changes that are under the way and occur in different workspaces. Therefore, it allows early detection of possible conflicts. The fundamental approach is to notify developers of possible conflicts emerging from synchronous changes to the same artifact that desecrate existing parallel work.

Since we always think about productivity, nearly all software systems nowadays are developed by programmers working in collaborative software development [6]. Against this

background, recent software development trends produce a version of system that unites the objects of simultaneous changes. Usually, each developer works on their private clones of the system to secure firmness during programming. Still, this prevents them from understanding what is happening in different workspaces where their coworkers act – something which can considerably impact their own work. Accordingly, conflicts occur owing to parallel work, and they become even more complicated as changes advance without being united and as far as developers postpone the revision. Therefore, the later conflicts are identified, it is more complicated to address them, since more code has to be changed. Moreover, a conflict detected late is mostly harder to fix because the changes that created it are no longer "recent" from a developers' perception.

In their study on parallel changes in large-scale software development [7], Perry and Siy asserted that the number of software conflicts rapidly increases during parallel work, which is considered to be a significant issue and one that is (partially) supported by software tools. Although this study comes from the beginning of this century, its conclusions are still valid, since even today parallel work enlarges with the increasing circulation of software teams, and they still use tools and processes similar to those presented by Perry and Siy.

Furthermore, in one of their recent works on proactive detection of collaboration conflicts [8], Brun and Holmes studied nine of the most active open source repositories in GitHub (`https://github.com/`). The authors conclude that despite most modern version control systems, such as Git (`https://git-scm.com/`), conflicts during merging phase still appear to be "persistent, frequent, and show not only as concurrence textual edits, but also as succeeding build and failures".

After the majority of the industry became aware of conflict problems, several experts have suggested few leading practices to manage merge conflicts [9], such as Continuous Integration[10][11], which suggests periodic merges and check-ins to avoid undetected conflicts for a long time. Unfortunately, the merging process by itself is clumsy and distorts the typical programming flow, resulting in some developers not merging as regularly as advised. It is not unknown that some teams have avoided working in parallel due to complicated merges, and that some developers often hurry up with their tasks in order to bypass the merging responsibility.

However, it is not only developers' fault. Awareness of co-workers activities leads to interruption of isolation work by notifying developers, where their co-workers are currently making changes in the code. It is really the key information, since it can be used by the developers to detect conflicts earlier. Anyhow, because today's programming languages have complex semantics [12], having polymorphism, encapsulation, and late binding, it is complicated for developers to detect conflicts on their own while they are developing. Additionally, sometimes awareness can lead to overwhelming developers with enormous amount of information, hence conflict detection may be even harder.

This paper contains a solution that decreases the number of active conflicts during collaborative development. The following solution frequently commits, pushes and merges uncommitted changes in order to create a background system that checks the code, by analyzing, compiling, and testing with a goal to detect possible conflicts with high accuracy on developers' favors, while they are programming. Possible detected conflicts will be presented to the influenced developers through the Integrated Development Environment. In addition, our solution presents a prototype as a plugin for IDE.

1.1 Motivation

By the time industry and research established requirement [13] for early detection of conflicts. The following snapshot illustrates how to understand and resolve merge conflicts.

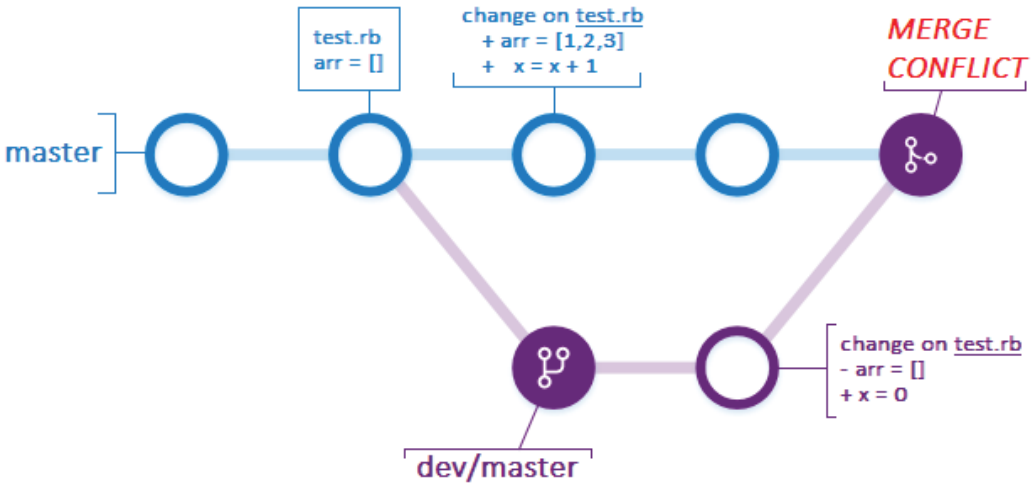

Figure 1.1: Conflicts in Git

The previous image shows an example of how conflicts occur in Git. Suppose that we have two developers, the first developer working on the main branch and the second one checking out the same working copy from their version control system. The second developer creates a branch named dev/master and simultaneously they make changes. The first developer adds “x=x+1” and changes “arr=[1,2,3]” to test.rb file, and pushes to the master. Meanwhile, the second developer removes “arr=[]” and adds “x=0” to the same file, then pushes to dev/master. Note that all pushes are clean since the developers are working on separate branches, although a merge conflict will occur. Eventually, when a developer request a merge, a conflict will arise since they have been working on the same file.

Besides this one, there are a lot of different examples. Sometimes the VCS will override the code or perform an auto-fix if possible. Unfortunately, if a version control system performs auto-fix without alerting the developer, some important artifact may be changed and it could lead to a bug will entering production, and every further step would be based on top of the broken code. All of the merge conflicts are unexpected, since developers are not aware of what is happening on different workspaces. Some of examples can show that conflicts can be really complex for detecting and costly for resolving.

Generally, the main issue when a conflict occurs is that a developer has to overlook other branches, since nobody can suspect his co-workers’ changes. In addition, when a merge is done, it is necessary to run all tests a sometimes a test suite fails because of a conflict in merge phase.

If somehow a bug leaks to the production, the whole product may misbehave and the developers have to find the bug and try to resolve it. Also, they have to remember everything what they have done in previous tasks, and discover the bugs’ impact on other parts of the code. The main pitfall is that all this undoubtedly takes developers’ time since the changes

are not any more recent in developers' minds. Generally it will lead to removing a duplicated code, renaming or refactoring an artifact. Because it takes time, a conflict by itself is costly, and the cost rise exponential with a time delay of finding.

Finally, each member of the team typically feels accountable for his or her work. Today's VC approaches [14], frequently force developers to amend a coworkers' work with a goal to resolve a conflict, without even reaching the coworker. This way of resolving conflicts, periodically leads to inconsistent code, social conflicts and, most of all, anger and frustration inside the team. Our idea is oriented towards reducing such conflicts and increasing the developers' productivity which would lead to the product perfection.

1.2 Objective

It would be useful to detect a conflict before it occurs during programming and avoid all possible modifications. Awareness [9][15], explained as "a comprehension of all activities of others that gives a background for your action" [15], has been proposed with the aim of helping developers to detect conflicts before they occur [16][17][18][19]. Generally, these approaches are based on awareness of change and presence.

Awareness of presence notifies a developer where in the code other developers are working. Since it may be useful to detect colleagues for collaborative work, it is not useful with conflict detection. Documenting which files are under the change may overbear developers with announcements that are unrelated to developers' work. There are some tools that use dependency-based awareness and inform when two files link by a path of dependencies changed by the developer and his colleague concurrently. The main goal of dependency-based awareness tools [9][20] is to decrease the number of notifications, and afterwards a developer can be focused on more applicable ones, which may lead to a conflict with another developer. E.g. rather than notifying for every simultaneous change in each file, there are some tools that inform on only simultaneous changes of the file dependencies that are changed by the developer.

In addition, it does not matter how many notifications a developer receives, they still have to explore each notification in order to decide whether they may bear any conflict. Consequently it will take time from developing. Regrettably, conflict detection on their own can be challenging for developers, since some code parts may have a complex structure [12], such as polymorphism, encapsulation, etc.

1.3 Approach

In spite of program errors, it is mainly inexpensive and painless to spot and resolve conflicts earlier, before they reach the production and the relevant changes vanish from the developers' minds'. At present, this knowledge is not easily reachable to developers.

Our approach prepares and serves information about the possible conflicts in a continual and precise way. The aim of this information is to empower developers for better decisions about when and how to share their changes, reducing the overload for developers at the same time. To achieve the mentioned: we study similar systems to get a better understanding on how they work and what are their downsides, we combine collaborative development, version control systems and speculative analysis explore future (possible) status of the software, all together to get accurate discovery of arising conflicts and provide detailed information about them. The future of version control systems may improve the way

in which developers spot and resolve conflicts. We design and construct a tool that analyzes and presents hints to developers, with the objective to identify, resolve and intercept conflicts.

1.4 Structure of the thesis

The thesis is divided into eight sections; it starts with an introduction, pointing out the motivation, the objective and giving an overview of the applied approach. Next section is background information, providing a brief explanation on the background, the scenario and the terminology that has been used. Chapter 3 points out the problem, explaining the scope of our work. The next step is listing the background of related work, summarized in chapter 4. Further on, we show what we have done to solve our problem. Furthermore, the thesis goes over to the evaluation of the developed tool, analyzing it using experiments and questionnaire, and reporting the results of the evaluation. In the discussion chapter, we discuss the aim of our work. Finally the last section summarizes the major conducted activities and their relevance and illustrates faced technical problems and some possible future work. t

Chapter 2

Background Information

Projects in Software Engineering are becoming more complex [21][22], thus requiring more software engineers to put their efforts in the production of a large software system. Essential to this effort is to evolve a shared understanding of coordination of a common set of artifacts, where each artifact symbolizes its own model, during the complete development process. This focus on collaborative software development within a bigger project is to develop a truly collaborative culture, which tends to understand the team composition, identify the barriers to change, create user-focused environment and regular project reviews [23].

Being humans, we have integrated limitations that influence our capability to produce nearly any piece of code. During the work with high levels of abstraction, such as software designing, writing requirements, and writing a code, or even creating test cases, we are error-prone and slow [21]. As an outcome, in order to finish large projects within expected time, we have to work together, and involve other people who try to catch our faults. As soon as we start working together, we encounter other problems. First one is our language, which we use to communicate, that is astonishingly expressive, but habitually double-edged. Even though our memory is brilliant, it is not accurate and error-free enough to recall a project's crucial detail. We are powerless in tracing everyone's work in a large group, and there is a risk of duplicating or throwing someone's work. Knowing that huge systems can be frequently perceived from different approaches, for this reason engineers have to agree on only one design and architecture.

Collaboration is an important aspect of software engineering. In time software engineering has gradually developed specific routines with the aim of reducing our limitations. Nowadays, collaborative software development without some version control systems (VCSs) is impossible to imagine or comprehend. Version control systems have certain importance, since such systems effectively control simultaneous revisions on a single artifact executed by various developers. On the other hand, the pessimistic approach only lets one developer at a time change an artifact. As every member of a team works on a single copy of the project file (source code, build revisions, etc.), they repetitively create revisions to theirs local files, and synchronize with their teammates. The synchronization is a double-edged process; it allows quick development progress, but also results in conflicting files during simultaneous work between developers. Allways when we interact with someone, there is a risk of facing a conflict. Every conflict is costly. Firstly the project will be delayed because it takes a while till the conflict is recognized and fixed. Furthermore, a developer can postpone the merge of teammates' work because of the threat that a conflict can be tough to deal with. It is odd, but the threat of possible conflict may cause developer to go even further, increasing the

likelihood of encountering real conflicts [8][24].

2.1 Background - Scenario

Think about a simple scenario where two developers are adding features to a project by working on their local code. As a part of a feature, developer makes some changes, on his own copy. When they are done, each of them runs the tests on their local copies (test suite passed successfully for both of them), thenceforth they push their changes to the master repository. After the first developer has pushed his code, a second developer tries the same. Thenceforth the second developer is informed that a conflict has occurred. Then, they have to recall their earlier changes and expectations, and their fixes might force them to modify other code that had been written in the meantime.

The way to minimize this type of troubles is to use a recognizing tool that alerts, which teammate is working on the same part of code, letting a developer be more aware of a possible conflict that may occur in these parts. E.g. when one developer makes a change, a recognizing tool can notify another developer that someone else is also modifying the code he is working on. On the other hand, when a developer changes a file, the tool may indicate a false positive warning, if the change had not influenced another developer. Additionally, a developer might have been trying some things and making some changes, without having intention to share half-way changes with other developers. Therefore, the tool has the ability to show early notice, but also the possibility to show some false warnings.

2.2 Version Control Terminology

Version Control systems allow you to track all your files at the same time. If there is an issue, you can easily roll back to a previous revision and work on it, or detect the problem and try to fix it. Nowadays, most of version control systems stick to the similar concepts. The following table 2.1 is Git based.

Table 2.1: Git terminology

Name	Description
branch	A "branch" is an active line of development. The most recent commit on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch. A single Git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the "current" or "checked out" branch), and HEAD points to that branch.
changeset	BitKeeper/cvsps speak for "commit". Since Git does not store changes, but states, it really does not make sense to use the term "changesets" with Git.
checkout	The action of updating all or part of the working tree with a tree object or blob from the object database, and updating the index and HEAD if the whole working tree has been pointed at a new branch.
clean	A working tree is clean, if it corresponds to the revision referenced by the current head.

Continued on next page

Table 2.1 – continued from previous page

Name	Description
commit	A single point in the Git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by Git in the same places other revision control systems use the words "revision" or "version"
fast-forward	A fast-forward is a special type of merge where you have a revision and you are "merging" another branch's changes that happen to be a descendant of what you have. In such a case, you do not make a new merge commit but instead just update to his revision. This will happen frequently on a remote-tracking branch of a remote repository.
fetch	Fetching a branch means to get the branch's head ref from a remote repository, to find out which objects are missing from the local object database, and to get them, too.
gitfile	A plain file .git at the root of a working tree that points at the directory that is the real repository.
head	A named reference to the commit at the tip of a branch. Heads are stored in a file in $GIT_DIR/refs/heads/ directory, except when using packed refs.
HEAD	The current branch. In more detail: Your working tree is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the heads in your repository, except when using a detached HEAD, in which case it directly references an arbitrary commit.
hooks	During the normal execution of several Git commands, call-outs are made to optional scripts that allow a developer to add functionality or checking. Typically, the hooks allow for a command to be pre-verified and potentially aborted, and allow for a post-notification after the operation is done. The hook scripts are found in the $GIT_DIR/hooks/ directory, and are enabled by simply removing the .sample suffix from the filename. In earlier versions of Git you had to make them executable.
master	The default development branch. Whenever you create a Git repository, a branch named "master" is created, and becomes the active branch. In most cases, this contains the local development, though that is purely by convention and is not required.
merge	To bring the contents of another branch (possibly from an external repository) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first fetching the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a pull. Merging is performed by an automatic process that identifies changes made since the branches diverged, and then applies all those changes together. In cases where changes conflict, manual intervention may be required to complete the merge.
origin	The default upstream repository. Most projects have at least one upstream project which they track. By default origin is used for that purpose. New upstream updates will be fetched into remote-tracking branches named origin/name-of-upstream-branch, which you can see using git branch -r.

Continued on next page

Table 2.1 – continued from previous page

Name	Description
pull	Pulling a branch means to fetch it and merge it.
push	Pushing a branch means to get the branch's head ref from a remote repository, find out if it is an ancestor to the branch's local head ref, and in that case, putting all objects, which are reachable from the local head ref, and which are missing from the remote repository, into the remote object database, and updating the remote head ref. If the remote head is not an ancestor to the local head, the push fails.
rebase	To reapply a series of changes from a branch to a different base, and reset the head of that branch to the result.
remote repository	A repository which is used to track the same project but resides somewhere else. To communicate with remotes, see fetch or push.
repository (repo)	A collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.

For further references see Git Glossary documentation

2.2.1 Checkout

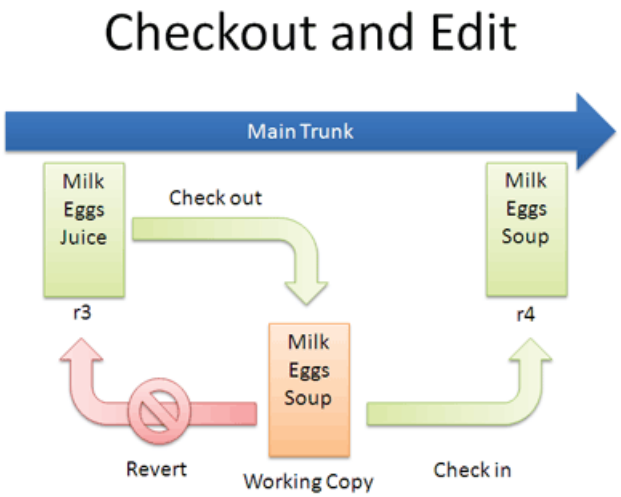

Figure 2.1: Checkout and edit - Visual explanation

In real life, you do not have to check a file regularly, you can only checkout, edit and commit. If you are not satisfied with your changes, you can always revert the changes to the previous version and start again. In case when you are checking out, you will receive the latest revision by default. In case that you want a specific revision, you can check out with an additional parameter, specifying a particular version or a branch.

2.2.2 Diffs

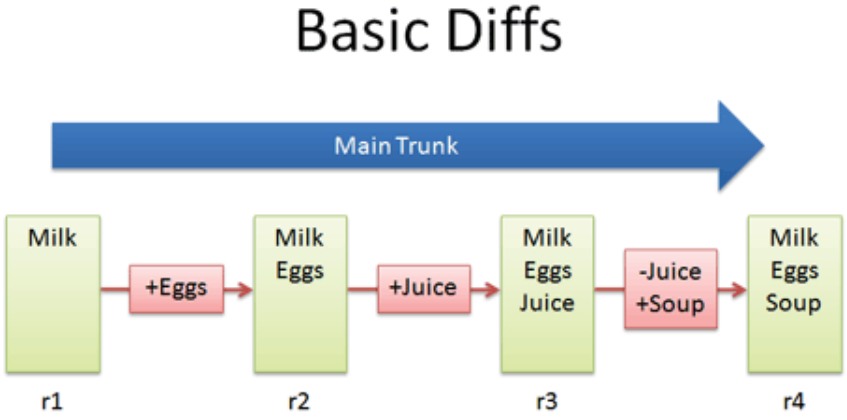

Figure 2.2: Diffs - Visual explanation

The trunk contains a history of all changes during a period of time. Diffs represent the changes that have been made during the development process. The idea is similar to stickers, you can “peel” them off, and “stick” them to a file. Nowadays, version control systems work with diffs. It is more preferable than working with full copies of the file. It helps to save

disk space, but it is also helpful for noticing changes between revisions or even applying them between branches.

2.2.3 Branching

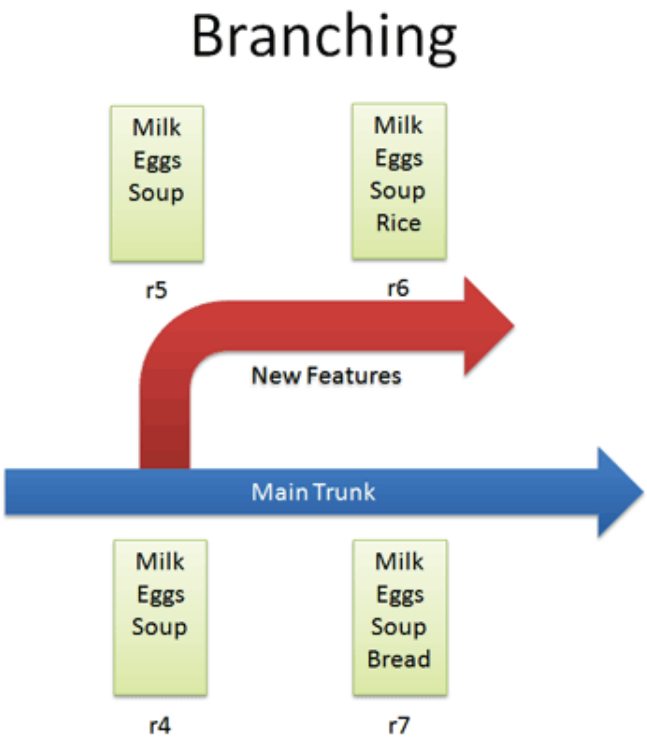

Figure 2.3: Branching - Visual explanation

Collaborative development cannot be finished without branches. They let us copy the code to a different folder and we can play around it separately. E.g. we want to create a branch for experimental things, odd stuff like eggs with rice or milk with soup. When we have a branch, we can play with our code and cut the edges. Because we are in a different branch, changes and tests will be isolated, and we are sure it is not going to influence anyone. Version control system controls every branch history.

2.2.4 Merging

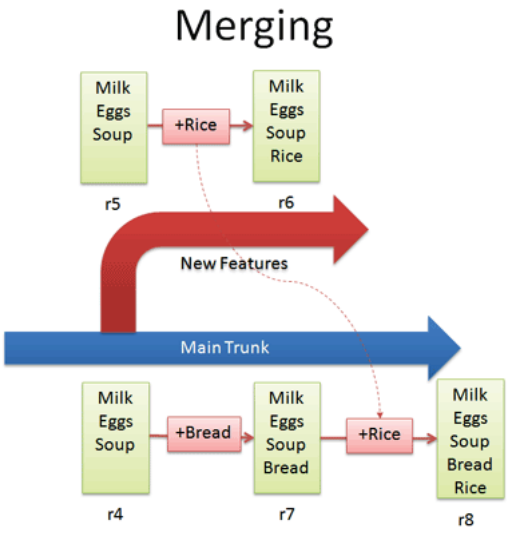

Figure 2.4: Checkout and edit - Visual explanation

To add the "Rice" feature from our New Features branch to the Main one. What are we going to do? We want to apply the changes that appeared in the New Features branch. That suggests we diff r5 and r6, and put that in the Main branch. Wonder what would happen if we diffed r6 and r7? The "Bread" feature from the Main branch would be lost. The point is- imagine that we peel off the change from New Features branch (+Rice) and stick it to the Main one. Furthermore, Main branch can have other changes, but that is fine, since we only want to add the Rice feature.

2.2.5 Conflicts

Conflicts represent our main topic. Frequently, version control systems can automatically merge changes between files. Typically, conflicts occur when we have changes that do not unify. Here we have two developers, Joe and Sue. Joe creates a branch, removes eggs (-eggs) and adds cheese (+cheese). And on her branch, Sue replaces eggs (-eggs) with a hot dog (+hot dog). Currently, Joe's branch looks like this Milk; Cheese; Juice, and Sue's branch contains Milk; Hot Dog, Juice. Thereupon, if Joe wants to commit a change first, the change will be valid and Sue cannot make her change (because there are no eggs in the Main branch, so it can be removed). In this case, when changes across and similar to this, the version control system can display a conflict and forbid following commits. Conflicts are do not occur often, but when they appear, sometimes you have to strain to resolve it. There are a few approaches for simplifying the issue:

- Re-apply your changes by synchronizing to the latest revision and re-applying your changes to conflicted file.

- Override their changes with yours by checking-out the latest version, overriding with your version and committing effected files.

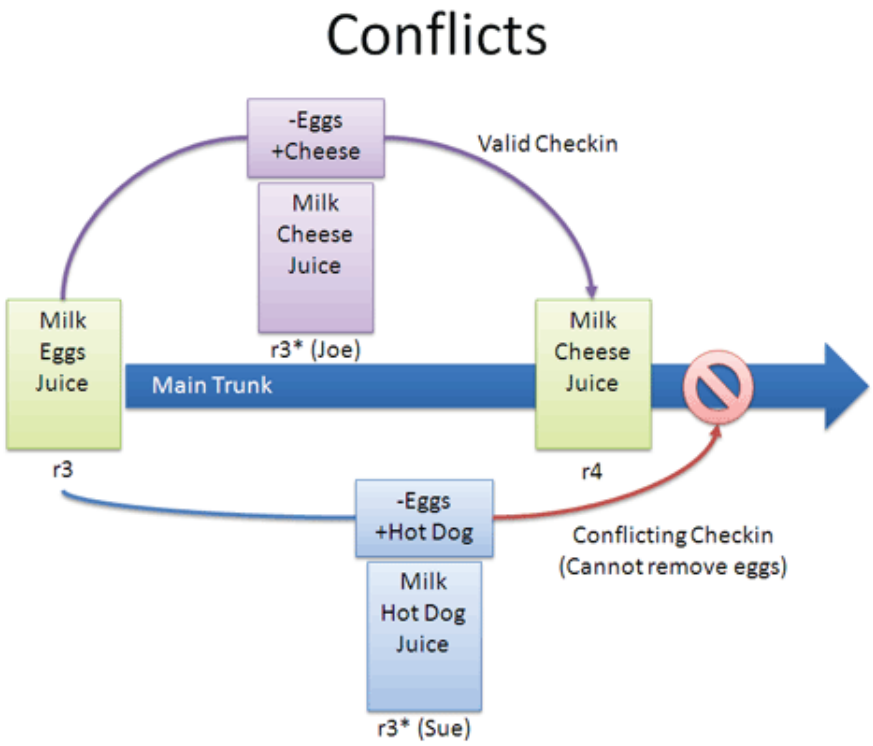

Figure 2.5: Checkout and edit - Visual explanation

2.2.6 Repository relationships

In this section we will describe different relations that can happen between two repositories. Generally there are three core relationships [8]:

- Same
  - ◇ Representing two repositories with the same change-sets. A repository is the SAME as the master repository only if they both consist of same change-sets, generally a fresh copy is in the SAME relationship.
- Ahead
  - ◇ Situation when the repository contains the same and additional change-sets of the other repository. Ahead is the number of commits on our branch that do not appear on the base branch.
- Behind
  - ◇ Opposite of Ahead, a situation when the repository contains a subset of other repository's change-set. It is the number of commits on the base branch that do not appear on our branch.

Ahead and behind are practically a type of "age" metric. The number representing ahead informs you approximately how much impact our branch will have on the base branch in the case of merging. On the other side, the behind number informs you approximately how

much effort has been done on the base branch since our branch was created. The behind number could be very helpful, since you can determine whether a branch is going to be merged without any conflict. In case when a plenty of work has been done on the base branch, it is probable that the two branches have been modified on the same lines. When the behind number is high, perhaps you should think about merging the base branch into your branch for synchronization. After you merge them, the behind number will be zero.

In addition to three core relationship, we can add four more relationships that show outcomes of sequences of change-sets [8].

- Textual X (pronounced as "textual conflict")
  - ◇ Different changes lead to a textual conflict, which cannot be automatically merged from version control system, and it needs human intervention.
- Build X (build conflict)
  - ◇ Version control system can automatically merge repositories, but it leads to build failure.
- Test X (tests failure)
  - ◇ The merge is successful, followed by successful build, but the test suite fails.
- Test Y (tests successful)
  - ◇ The repositories were merged, build is successful and test suite passes.

To summarize, a potential merge can lead to textual conflict and opposite. If there is not textual conflict, the next phase is to check for a build. The result could be successful or it may fail to build. If the build is successful, the test suite check comes afterwards. It leads to failing or passing test suite.

2.2.7 Git hooks

When comparing many other version control systems with Git, we can say that Git has a specific way of triggering different scripts in cases when specific actions occur [25]. That way is well known as Git Hooks and it splits into two groups of hooks: server-side and client-side. Server-side hooks are more oriented to running on network actions such as obtaining pushed commits and on other side client-side hooks aim at actions such as merging and committing. Moreover, we can combine these hooks for our own reasons, so it is a really powerful tool.

By default, the hooks are located in a subdirectory of the Git directory, named hooks. Generally, that is git/hooks path. During the initialization step of a new repository with git init, Git automatically creates the hooks directory and populates with a bundle of specific scripts, which can be used as an example. They are normally useful, but they require additional knowledge on specific programming languages. All of provided examples are written as shell scripts, with a bit of Perl inside, however every executable script will work fine, so you can write them in Python or Ruby or any other language you are familiar with. The scripts can be bundled, but they require specific preference. Some of client side hooks are: precommit, prepare-commit-msg, commit-msg, post-commit, pre-rebase,

post-rewrite, postcheckout, post-merge, pre-push, and pre-auto-gc, then we have hooks for email workflow: applypatch-msg, pre-applypatch, and post-appypatch, and lastly server-side hooks: pre-receive, update, and post-receive.

For additional information visit Git Hooks documentation

2.2.8 Conflict types

There are a lot of different types of conflicts that can appear during the collaborative development [26][27]. Firstly, we have to differentiate direct conflicts from indirect conflicts. Direct conflicts include conflicting, side by side changes to the same artifact. On the other side, indirect conflicts include conflicting, side by side changes beyond artifacts. This division straightforwardly splits conflicts influenced by the scope of the changes, which is, in case a conflict is referred to a single artifact or includes several, related artifacts. Secondly, the degree of the problem in fixing a conflict is highly complex. It is difficult to scale it, since it includes multiple factors, such as the syntax and semantics of a programming language, experience of a developer, different tools, different test cases, etc. Generally, we can find that incompatible syntactical changes cause a conflict more often. Infrequently conflicts are caused by having two or more incompatible changes that present tangled timing issue in a collaborative system. Thinking about first situation, if a build failed, it will undoubtedly indicate a developer to the right spot quite fast. In the second situation, a developer should wish for that the present test cases will catch and reveal the issue, otherwise, it might be impossible to detect the problem until the product appears in the production.

We are interested in these conflicts whose appearance might be identified by using automated checks. Presently, conflicts are usually undetected until all changes are completed, just after changes are checked, merged, built, and tested. In order to detect these conflicts before they are revealed, important data from each of the developers are taken at once for the inspection. This data will certainly be incomplete, since it will indicate real time changes in progress that have not been done yet. Furthermore, it is unimaginable to share whole project between developers in the case of the large collaborative group, due to scale issues. Thus, relevant data have to be specified between a workgroup in order to predict a conflict. The main idea of our design is to inform developers of possible conflicts in the development stage, which will include all artifacts. Our approach will not distinguish difficulties in resolving a conflict, but it will note any suspicious modification of the same artifact. Additionally, developers are allowed to take a closer look at the result of possible merged changes, whether or not an actual conflict occurred.

Textual conflicts In the collaborative development it is unlikely to have all possible duos of developers with stable relations (Same, Ahead, or Behind) between each other. Thus, to get a better understanding on how often conflicts occur we obtained relevant data from a few relevant public repositories [8]. Looking at the 2.6 we can see how often developers' changes were merged and which relationships they run into.

system	merges	TEXTUAL✘		TEXTUAL✓			
				BUILD✘		BUILD✓	
						TEST✘	TEST✓
Git	1,362	227	17%	2	0.1%	53 4%	1,080 79%
Perl5	185	14	8%	7	4%	51 28%	113 61%
Voldemort	147	25	17%	15	10%	5 3%	102 69%
Gallery3	506	47	9%			459 91%	
Insoshi	93	23	25%			70 75%	
jQuery	18	1	6%			17 94%	
MaNGOS	194	81	42%			113 58%	
Rails	362	51	14%			311 86%	
Samba	748	100	13%			648 87%	
total	3,635	572	16%			3,065 84%	

Figure 2.6: Conflicts caused by merging

Looking at the total of the merges, 16% or every sixth merge had a textual conflict diagnosed by Git's mechanism. This number represents the Textual X relationship. Remaining 83% of the total merges were without textual conflicts, resulting from the fact that the developers were in the Textual Y relationship, either Build X or Test X. The amount of the Textual X relationship is important, because uncategorized Textual X relationship between two repositories can cause conflicts. In addition, it could be a case when a developer is uncertain whose changes led to a conflict, and might avoid fixing them, letting conflicts to remain and spread out.

system	merges	TEXTUAL✘		TEXTUAL✓	
Git	179,249	15,965	9%	163,284	91%
Perl5	7,352	1,290	18%	6,052	82%
Voldemort	4,512	1,534	34%	2,978	66%
Gallery3	6,924	1,262	18%	5,662	82%
Insoshi	1,742	736	42%	1,006	58%
jQuery	74	13	18%	61	82%
MaNGOS	4,967	1,092	22%	3,875	78%
Rails	10,418	2,971	29%	7,447	71%
Samba	77,683	30,635	39%	47,048	61%
total	292,921	55,498	19%	237,423	81%

Figure 2.7: Conflicts caused by potential merge

Figure 2.7 also shows every commit where developers could potentially merge their changes before. Averagely 19% of the potential merges would have led to a textual conflict. In other words, if the developers were using an additional tool for conflicts recognition, for 19% of the commits developers would be notified about possible Textual X relationship. For another 81% of the clean merges, developers would probably be informed that merge can be safely executed.

Higher-order conflicts In other words, 16% of total merges needed human intervention to fix a textual conflict. The human effort has been underrated because of it, furthermore textual merges cannot be safe forever. Besides an automatical merge can be followed by build or a test failure. Figure X sums up the following, during the development of Git, Perl5 and Voldemort, 76% of all merges were completed without difficulty, upcoming 16% of the merges raised a textual conflict, where 1% of total merges caused a build failure, and the rest of 6% of total merges produced a test failure. To sum up, the 266 textual conflicts were reported by VCS, but they constitute 67% of all conflicts. The rest of 33% were pure merges, exactly 399 of them headed to a build or a test conflict. There are a handful of popular recognition tools that can spot a higher-order conflict, mostly they will inform a developer of all changes that appear in a repository or any simultaneous changes which occur in the same file.

Chapter 3

Problem Statement

Collaborative software development invites programmers with their tasks to synchronize their work and merge discrete addition into separated workspaces (e.g. branches). Commonly, synchronization is based on a chain of "update, modify and commit" commands, upcoming merge conflicts emerge consequent to commit where discrete changes have deviated and have to be solved clearly. In addition, researchers and industry have been proposing improvements [9] saying that providing awareness information on right time regarding "who changes what" may not only help with handling conflicts, but also enhance productivity and team performance in collaborative work.

Normally, the influence of awareness of information on software development, especially world-wide distributed is enormous.

First of all, the absence of awareness happens more regularly than merge conflicts, and this information can reveal how each action may affect other developers and their relationship states can aid developers in making superior decisions [28]. Secondly, the lack of awareness of information or insufficient awareness of information may more negatively influence developers' performance than merge conflicts [28].

Developers execute operations in a tree hierarchy, which leads only to pushing and pulling from a main repository. This coordinates how developers mostly collaborate with version control system. Since the relationship between developers relies on others' activities, we are going to examine only the information relevant to pair of developers who share a common main repository. By executing activities, developers may affect how long a conflict is going to last, or maybe halt it from arising. We classify relevant information into several categories, which concerns possible actions or the relationship. After all, by knowing possible actions and relationship, the awareness of information can help developers determine what they are going to do, or which action they are going to perform. Bearing in mind that conflict relationship may hearten the developer to deal with it earlier, whilst the changes are still fresh in the developers' minds, furthermore it may minimize the conflicts' existence moreover the exertion that needs to be abolished. When developers know about specific relationship in advance, they can be convinced that it is risk-free to integrate others' changes, following with approximately close development states. Sometimes, it could empower the developer to halt some likely conflicts entirely. At least, previously mentioned relationships are able to provoke developers to collaborate, that lead to conflict reduction in the developers minds and development approach.

Anyhow, the relevant information notifies the developers of what is applicable to a conflict, and it decreases the required time for solving it. In addition, notifications can tell de-

velopers the exact time to do an action for extinguishing the overload of determining out whether an action can be executed now and perhaps you can do upcoming undo action. This whole concept can let the developers have a peek into the time ahead, then restricting undo and redo of the code. Finally, it will allow a developer to know if someone else is resolving a conflict easier, thus benefiting the effort required for resolving.

Generally, any early recognition of conflicts between several developers leads to reduction of time for fixing the conflict, or at least is mostly unlikely to increase the time.

In our case, a tool cares instead of a developer and superbly informs the developer about the possible conflict that would emerge as the outcome of the merge. Using this information, the developer can use different approaches, may do the merges in another sequence, or manually merge in order to avoid the issue and make sure that other developers were not harmfully affected.

Knowing that version control systems have frequent and significant conflicts, and that an overall perspective of the VCS is able to observe and spot conflicts followed by frequent and harsh reduction, we should identify the ways to improve the current situation in delivering effectively information and guidance to developers. It could be a tool, which carries the key information without swamping or amusing the developer. The information has to be compact, to summarize all relevant data, letting a developer immediately recognize cases that may require a hand. In addition, the information should not be limited, but crucial, until a developer displays a particular enthusiasm in it. To make it easier for catching the eye, specific icons and colors should be used in the corresponding frame, rather than using a textual interpretation. Furthermore, scalability needs to be addressed, developers should be able to select an interested repository. This means, the provided information about relationships should be shared within developers, without the assumption that whole team uses the tool. However, each developer will be independent, and their choice is whether to use the tool or not. Moreover, their choice should not affect other developers, but it is necessary to use the tool in order to have valid examinations. If more developers are using the tool, the likelihood of identifying a conflict increases. Still, a well designed and implemented tool will lead to main goal that performs analysis of version control actions. Last, but not least, any proposal for improvement will be positive, and future work should be based on different evaluations, user studies and questionnaires.

After we summarize all related things, including our idea, we are focused on developing a conflict awareness detection tool, with a goal to provide extra stimulation for resolving merge conflict issues. Generally, awareness tools look to increase developers' awareness of all the changes presented by other developers, following the goal of conflict reduction and boosting team management and strategy. In addition, awareness is mainly oriented towards direct conflicts and effortless indirect conflicts such as small changes in methods [18]. Tools for detecting conflicts and predicting them mainly check for early conflicts and report them as early as possible. Normally these tools are running by regularly checking for changes in workspaces of different developers, then trying to merge, and if a hypothetical merge fails, it indicates that a possible conflict might happen. In addition, these tools should provide, beside conflict detection, feature for compiling and testing the product.

Chapter 4

Review of the state of the art

This section points out the background of related work, such as evaluating the cost of conflicts, early conflict detection, awareness in a collaborative environment, continuous development, and mining software repository. Version control systems are considered to be the first configuration management solution for awareness. A notable work exists in order to increase developers' productivity that is based on awareness and assistance, and they are helpful to developers in making better decisions mainly regarding collaborative development.

4.1 The Cost of conflicts

In order to achieve a productive software development, the key requirement is well organized coordination. As the number of parallel changes rises, the numbers of possible conflicts increases, but the developers are still able to adequately organize the threats to their workspaces if they are aware of the effects of their commits on other workspaces. Developers forgo parallel development to dodge resolving conflicts when they occur after commit, or hurry their work since they do not want to be the developer with responsibility of resolving conflicts. There are a lot of experiments and studies that appraise the advantage of collaborative awareness for developers and configuration management. Expanding these outcomes, we analyzed several real projects followed by their developers with a goal to estimate the potential benefit. After the analysis, we received that potential awareness could be helpful for a development team, for their coordination and for individual development. After checking the classification of collaborative tools for software development, we could consider that our tool may provide ongoing awareness in real-time development stage and the instruction in the possible outturns of potential future conflicts.

4.2 Early conflict detection

At these moments developers are growingly embracing version source code management systems (SCM) with vast support for branching, collaborative development, parallel development, and merging, using well known tools such as Git, Subversion and Mercurial. Usage of these tools allows openness for developers, but on the other hand also causes a lot of issues when numerous branches have to be merged together. One of the famous quotes, reported by Microsoft developers says that most of the time dedicated to merge operations

is used in conflicts solving and code validation. Summing up, merging multiple branches is a painful, expensive, and error-prone process that requires specific techniques to be handled efficiently. The whole idea behind the techniques for early detection of conflicts and unforeseen synergy between changes on different branches is oriented towards reduction of the effort that is needed to handle the software development and simultaneous growth. The idea has been investigated of foreseeing conflict detection by merging the code in the developers' working copy with the code pulled from other branches. Then, the analysis is being performed on the developers' machines with the goal of detecting potential conflicts (textual, build, and test), before the actual merge is handled by the SCM system. This approach consists of running various client-side analyses of every operation. WeCode [29] is one of these tools, it uses a merged system to detect complex conflicts using compilation and testing, that is a feature of typical awareness tool. It does a single background merge of all developments working on the same branch, followed by relevant notifications. In addition, the tool called Crystal does separate background merges of a pair of repositories, involving a master repository, of the representative developer and of co-workers. It provides detailed information and guidance on foreseen conflicts. On the other side there is Palantir [26], a workspace awareness tool that destroys isolation by notifying developers of parallel under-way changes done by other developers. By providing awareness in developer workspaces, developers are able to perceive potential conflicts before they actually occur.

4.3 Collaborative awareness

Software development projects are mainly essentially collaborative, requiring more developers to coordinate their work and efforts with the goal of creating a complex system. An essential part of the whole process is a shared understanding between developers, regarding the state of the project's tasks, artifacts, and all activities developers are working on within the project. This is known as awareness information, which affects developers, either directly or not. There are a lot of similar researches that are oriented towards collaborative awareness – how to increase awareness of the activities among team members. This awareness may be a diversion, unless a conflict is forthcoming, thus these tools for collaborative awareness have embraced growingly revolutionary methods in order to avoid false positive warnings. There is a tool for collaborative software development called Syde [30], which reduces false positive warnings, using a fine-grained analysis of the abstract syntax tree modifications, also known as ASTs. Having two possibly conflicting changes of the same file will be labeled for a developer only when they additionally influence changes of the same chunk of the code. E.g. having two developers working on the same project, and in case if they change, delete or insert the same method, a "yellow" flag will be raised above the changes. In addition, if one of the developers committed the changes, a "red" flag will be raised instead indicating that a merge conflict may occur. Last, but not least, Syde [30] is triggered every time a change is saved, and examining files afterwards. Another reprehensive tool is Palantir [26],which displays which developers are changing which file within what scope. Our motivations embody Palantirs [26]' motivation, which is: "providing workspace awareness to users will enable them to detect potential conflicts earlier, as they occur". Theoretically developers can afterwards in a proactive manner synchronize their work with a goal to eschew those conflicts. A virtual dashboard for fostering awareness in software teams called FASTDash [19] is a similar tool, which is an interactive visual representation of which files each developer is modifying. The tool amplifies existing software development tools with

a particular point on aiding developers in perceiving what is going on with another team members' work. In addition to the above mentioned, tools such as CollabVS [17] and Safe-commit [31] are tools with the most thorough analysis. CollabVS [17] covers all collaboration activities and involves a single collaborative session host and one or more guests. The host is the person that starts the collaborative session and anyone that joins is a guest. It is an expansion tool for visual studio that detects a potential conflict when a developer is editing an artifact that is dependent and edited by another developer but not committed by him. On the other hand, an algorithm named Safe-commit [31] is an analysis-based technique for determining changes that are committed without endangering the stability of the repository, including cases when there are failing tests in a developer's local directory. It does the massive program dependence analysis, recognizing the changes that are surely not to be a starting point of test failure. Looking over the algorithm, it enables pushing some of changes earlier, based on the theory that increased repetition of publishing changes can lead to decrease in the amount of identical work and the probability of merge conflicts. Since every algorithm or approach suffers some bottlenecks, our idea is to find the best of each of them and put them together. Firstly, our approach looks for every possible conflict, rather than being focused on a single one. The approach is based on developers' minds, so it follows steps that a developer would literally do hereafter. As usually, it will execute a version control command, check for any conflicts, then run the scripts for compiling and building, and finally run the test suite. Thus, our goal is to present only conflicts that would actually occur while executing foregoing workflow. Also, we should consider that a tool is strongly related to a version control system, so any further improvement of the version control system may be beneficial for the tool. Secondly, the tool will report any possible conflict that may occur in real-time, so our developers will be aware what is going on in other workspaces, even though the changes have not been committed by a developer. This helps in time cost reduction, since developers are informed what is happening between them, and they can interact in order to prevent losing time in debugging and conflict resolving. In addition, opposite to the previous tools, our aim is to help developers in executing risk free merges with protection of conflicts. Lastly, the tool supports various levels of conflicts, such as textual, build, or even test.

4.4 Continuous development

The term continuous development is contained by continuous integration, continuous delivery and continuous testing. It adds additional automation to the software development process. As soon as all automated delivery tests are finished successfully, whole code is deployed into production. Since all the changes are distributed to users rapidly without human involvement, the risk of continuous development appears. However, our approach may be identified as a part of continuous development, since it follows the same structure. The environment for development is an IDE that provides regular execution of the program, as it is developed. Current programming environments keep the project in a compiled state whole time during the development phase, focusing the development in two directions. At first, all compilation errors are presented to the developer, letting the developer fix a bug in the code while the code is still fresh in his mind. Secondly, the developer does not have to think when to compile, thus the developer is not distracted by compilation, building or testing tasks, since intervention is not necessary. Continuous testing adopts the exact concept to testing, it runs tests suite in the background, without interrupting the developer while

programming, and also everything takes place on the developer's computer. It is calculated to reduce effort and time maintain code clean and intercept errors for a longer time period. Since it is executed after every change, the developer is aware instantly whether the change has ruined the tests. Many of today's researchers have put a lot of effort assisting developers with the goal to decrease their uncertainty in committing changes using version control systems. Remember FASTDash [19], with the aim to help developers avoid conflicts, our goal is to calculate and analyze the possibility of merging conflicts rather that to predict the presence of them.

4.5 Mining software repositories

Nowadays the term mining software repositories appears more and more often between software engineers at conferences, in research papers, etc. It is a software engineering field where researches and software practitioners use specific techniques to analyze the data in software repositories to extract applicable and useful information made by developers during the development phase. MSR requires relevant tools that extract data and structured code facts from different projects. One of them is a tool named SeCold [32], a platform that supports data extraction and on-the-fly inter-dataset integration of major version control, issue tracking, and quality evaluation systems. Also, well known is MetricMiner [33], a web application that facilitates the work of researchers involved in mining software repositories. It helps researchers in different phases of the MSR process, such as cloning the repository, extracting specific data, creating some datasets, or even running different tests. Since we are based on MSR model, our effort coincides with MSR's one. So, we are separating different directories, since we do not want to mess with the developer's main code, which leads to incorporating changes among the team members. Additionally, the purpose of the mining is to determine if the tool is beneficial. Beside the mentioned, we mostly aim to improve the software development process in a collaborative environment, by informing developers or other team members of uncommon activities so it is possible to assign more relevant resources to specific parts in a collaborative development process.

Chapter 5

Problem Solution

This part will answer how we solved the problem from the chapter 3.

5.1 A prototype - Design and implementation of the plugin

The plugin is a prototype for proactive early detection of conflicts in collaborative work and for improvement of the scalability and reliability of the project. Currently it works only with the Git version control system and Microsoft Visual Studio. The plugin is based on the speculative analysis [34]. The goal is to predict possible code conflicts, test conflicts and run conflicts, and to prevent them as much as possible. It is integrated into the environment and can present information when it is needed. It provides the screen for configuration and summarizes all forks of the project depending on the situation. It currently supports Microsoft Visual Studio and Eclipse IDE. Presently, the plugin is a concept, but the idea is to evolve into an open-source, standalone tool and to be available on Github's webpage.

Down below is a snapshot 5.1 of the plugin:

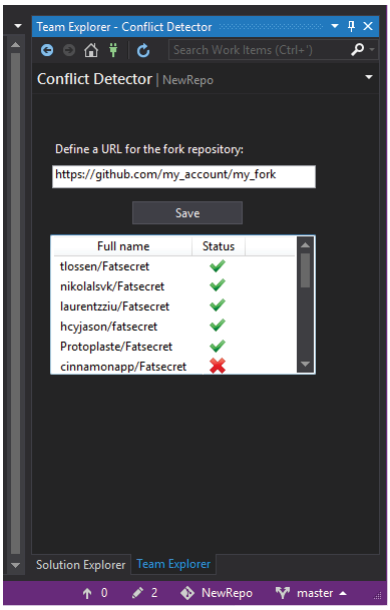

Figure 5.1: Snapshot of the plugin

It shows the URL of developer's fork and his relationship with other repositories (forks) and the other collaborators.

The local state tells a developer whether he must commit changes in order to resolve a conflict, or it is inevitable to fix it by himself. The plugin monitors multiple development forks (repositories) and it informs each developer when it is safe to push their changes, whether they have fallen behind or they should pull changes from the main repository, and whether changes other developers have made will cause a possible conflict. It inspects all existing forks of the main repository, each fork represents a working copy for a single developer. A fork contains all possible modifications in real time. The point is that commits are main units to work with, getting notifications created. If a conflict occurs, the plugin will inform a developer depending on configuration file, so the developer may fix the conflict as soon as possible. If commits were made without conflicts, the tool allows developers trust to continue working on their changes without being afraid of a possible side effect. The tool configuration displays the location of the developers' fork, all forks from the main repository, status, related to each of a fork, a command to execute compiling the project, a command to execute running tests and a command to execute the project.

5.2 Centralized vs. Distributed Version Control Systems

We are informed about many version control systems, which are split into two categories: "centralized" and "distributed".

The majority of developers are familiar with centralized version control systems such as Subversion (SVN), Perforce, and CVS, whereas others run directly into the distributed version control world, like Git, Mercurial and Bazaar. Each of them has their own advantages and disadvantages.

Centralized VCS are built on the idea to have a single "central" copy of a project located somewhere on a server, and developers commit their changes to the central copy. Committing is a process where a developer stores a change in the central system. Other developers are able to view this change. If there is a change, developers can "pull down" and the version control tool will automatically update the content of all changed files. The majority of present VCS work with "change-sets", which are a set of changes handled as a single unit. Centralized version control resolves the issues where developers were keeping many copies of their projects on hard drives, but now the version control tool can interface with central copy and restore any revision of the code. If you are working with a centralized VCS, a workflow for fixing a bug or implementing a new feature will normally follow the following pattern:

- Pull down every change from the central copy that was made by other developers
- Make some changes, test the code and make sure it works correctly
- Commit the changes to the central copy, hence it will be visible to other developers

During the last several years a new type of apps have become noticeable, called "distributed" version control system (short DVCS). The main difference with these systems is that they do not depend on a central server to keep all the revisions of a project's code. Instead, each developer "clones" a copy of a repository on their local drive, and they have the full history of the project. The copy contains all revisions and metadata of the source code. This approach may sound useless, but in reality, it is an upper hand. Most of the projects

are based mainly on the plain code (text files), and using modern system with compression, these files will be unnoticed in a hard drive space. In a workflow, we have actions called "pulling," which is a way of getting new changes from the repository, and "pushing", the action of uploading your changes to the repository. In either way, you will move change-sets (changes to files handled as a single unit), not only single-file difference. There is a common misbelief about DVCS that there cannot be a central project repository. It is fallacious, because you can think in the way of that a central repository is demanded by the used tool, but it is non-compulsory.

The act of cloning a whole repository grants distributed version control tools few advantages over centralized systems. Executing actions are rapid since the tool uses a hard drive, not a remote server. Commits can be local without affecting other developers, and once you are done with change-sets, you can push them all at once. The internet connection is needed for pushing and pulling, but you can work offline since you are not forced to commit lots of changes as one huge change-set. Furthermore, considering that each developer has a full local copy of the repository, they can share their changes with another person at a time, if they want to get a response before sharing the changes with all of them.

The goal of the version control systems is to resolve a particular issue that developers encounter, "storing and sharing various revisions of the code".

5.3 Used approach

Having three key insights helped us create our approach. They are:

1. Conflicts take time to evolve, the time which developers spend to create code modification in their personal repositories;

2. Developers want to arrange their work activities in order to reduce the possibility of conflicts;

3. Conflicts are costly, having in mind the time we spend in production and the time used for fixing them.

It does not matter what we are changing in a code, it takes time, whether it is thinking about the code, typing, compiling, testing, fixing mistakes, or even refactoring someone else's code. Accordingly, conflicts do not arise immediately in full shape, simply they appear as the outcome of parallel coding between two or more developers. Basically, a conflict appears as a result of a small change, it may be a single line change to the same file, or a single change of a method that has been called by another developer simultaneously. However, with time passing a conflict will generally spread alongside extra changes that were made by developers. Single line change may lead to important changes. On the other hand, a single change of method may lead to crucial refactoring of the entire class.

Having in mind the time-frame during which conflicts appear, our approach will notify developers about parallel changes from other repositories, which is an advantage on our side. We have an assumption that, if we inform developers of a possible conflict, they will take some eager actions to fix those conflicts as early as they can while the conflicts are still small. There are two main advantages:

1. The amount of effort that increases by the time when a conflict is perceived will be reduced by the downgrading amount of work that will be invested in redoing the issue

2. Reducing the amount of effort in the time when the changes are still small will lead to faster development

Normally, the effort will expand during the observing process, but we should be aware that information provided from the process is compulsory, since we want to take correlative steps. In our approach, developers are considered as a core of the solution, since they are behind the motivation and the problem statement. In addition, developers are settled in a different environment, where instead of ongoing coding and overlooking a conflict, they are demanded to take eager measures to approach, and if needed, fix a rising conflict of which they have been notified. Furthermore, sometimes is necessary to use additional ways of communication, using Slack, messaging tool, to contact the other developers, or even walking to their spot for a discussion and reorganization of the corresponding task. In order to make a choice to hold on a single change till the other developers recheck theirs, it could be done using version control systems' functionally for exploring the changes in another branch or repository. For these actions, a developer should put in some extra time for observing and clarifying the data that has been given. We are aware that additional effort is significant comparing how a conflict can spread out.

The tool works by checking changes in progress existing in local repositories and continuously sharing relevant information related to those changes with other developers to whom it is applicable. Therefore, the information represents what other developers are changing and to what extent. In case a developer appears in parallel changes to the single artifact, but doing it knowingly, that amount of change that happens to the same artifact in another repository, the amount of change is key information for us. It could be a small amount, so a developer may try to resolve a conflict without getting in touch with the other developer. Otherwise, if the amount is large, a developer would like to approach the other developer with the aim of resolving the conflict. In such case, they will agree who will make further changes, and who will wait, and how it will be done.

The plugin will work two-way, when a conflict occurs, each developer will be informed about other's modifications. In case that one developer selects to ignore the notification, but for another developer it looks really important, then he will contact the first developer in any way and additionally they will together deviate from their main path by suspending the current assignment for a short period of time.

Configuration is all about this tool, and to support direct and indirect conflicts, every developer should make a proper configuration file. A single change (i.e., code refactor, modification in a class' variables and methods, change of names, etc.) in one repository will lead to comparing the changes in all other available repositories. A repository is a developers' fork of the core project. The idea is to each repository taking the changes from other forks and analyze whether they generate a conflict with any of their local changes. If that is the case, a developer will be informed about the conflict related to a specific fork (repository). In addition, every developer will be aware of all changes happening in real time, not only of changes that are committed by a developer. Generally speaking, it is a bit annoying to be notified every time when someone is working on an artifact, but perhaps it is better to keep in mind that your change can lead to a conflict. Moreover, if a developer knows which artifact is under the change and what other artifacts can influence directly or indirectly your tasks, it lets you think about and organize your tasks, either to change the order or delay them, and know that no conflicts will occur. Obviously, every developer would like to avoid the conflict, rather than engage in a task blindly and hope for no conflicts. Lastly, the relevant information should not discourage developer from not working on his tasks, but it is

given to have a better understanding what is happening in the whole team. The plugins' user interface is designed to be readable and understandable, so developers can see notifications at any time, usually when their work involves another artifact, or when they move from task to task. It allows a developer to work undisturbed on their code, still providing enough information to understand remote changes when they happen.

In our approach we think about our developer, it is important to present all information through the design, therefore well build design is crucial for the tool. The first thing supported is that a developer is constantly informed about changes happening in real-life (relevant information will be transmitted on specific timeframe, such as every 60 seconds, or when a file is (auto) committed). When a conflict is detected, it will be labeled in the tool, with specific mark, depending on the conflicts' type.

Secondly, a developer will be informed about the conflict with relevant data. The data will contain information related to files infected with the conflict. Moreover, the data shows precisely where in the code an independent conflict occurs, so it will be more useful for the developer, since he is being able to instantly acquire in which way someone's work infects another developer. If more conflicts appear, it means that more involvement exists between different workspaces, thence it is barely possible that the relevant changes can be effortlessly merged. Considering that our approach is based on terms of artifacts (i.e., checking out, trying to merge, compiling, and testing), the feedback will contain different ways for representing conflicts, including the owner of corresponding fork.

Altogether, our approach grants collaborative work while giving a possibility for detection and resolving conflicts before they occur and before all changes are made and pushed to the branch. Although in our approach we say conflict, but technically speaking conflicting changes are not actually pushed, so it is more relevant to tag them as "potential conflicts". There is a miss benefit from the tool, because it is possible that an emerging conflict spreads during some time, then all suddenly vanishes because a developer who was simply experimenting chose to roll back their changes. During that time, a developer is aware of possible conflict, but as soon as another developer rolls back their changes, the relevant icon will be changed correspondingly. However, a conflict is not spreading always, sometimes it can diminish when a developer eliminates some of the code related to that conflict. Our tool cannot pledge that a "potential conflict" will develop into a real one in the following time or that will be at the same rank it is at the present time. In spite of that, it is harmful to get in touch with another developer to understand their ideas, as a result of that deciding should be a potential conflict handled as a real conflict and should be dealt instantly or should be ignored for a while. From now on, we are not going to make a difference between conflicts and potential conflicts, the term conflicts will be used for purposes of simplicity.

5.4 Implementation

The tool is completely implemented in Visual Studio 2017 as Visual Studio extension of Team Explorer functionality. It is written in C# language followed by the .NET framework. At the present time it is available as an extension for Visual Studio and Eclipse, and it currently supports the Git version control system.

5.4.1 Tools' architecture

Proposed architecture consists of the following elements:

- Configuration file, specific per developer and stores data related to the current workspaces, commands for compiling and testing, and git hooks configuration.
- A panel where a developer sets up the configuration and the panel monitors related activities and presents them for the developer. In addition, the panel contains all relevant forks of the main repository.
- Auto-committer, a background job that commits all changes in a specific time-frame. The changes are committed to a developers' fork, which is defined in the configuration file.
- Auto-pusher, another background job that pushes all commits after a certain period of time. They are pushed to the developers' fork.
- Git-hooks integration, after a developer is done with the configuration, a few git scripts will be pushed to the related repository. We are using post-receive, update, and post commit hooks. The hooks are implemented in Ruby. Depending on the hook, a certain event will trigger related script, and then we will be notified every time someone changes something. It is one of the reasons we implement a git hooks, so we are not going to lose any relevant information during the adjusted time-frame and we are sure that we are on the right path.
- Merge compatibility analyzer, allows us to access any remote repository, including relevant forks. Since we got all relevant forks, we can access the phase where we compare a fork with our local workspace and analyze the repositories. After previous step is done, we will get a response depending on the relationship between two repositories.

5.4.2 Configuration file

Every developer should install the tool and configure it. The configuration process demands from a developer to specify a relevant fork path for their workspace. In addition, a developer can select what git-hooks wants to enable and add commands for compiling and testing the code. All data will be stored in a discrete file. Down below, we can see a snapshot 5.2 of the configuration dialog.

5.4.3 The plugins' panel

Our plugin is an extension of the TeamExplorer, and in order to access the plugin we have to select a valid repository from the Team Explorer page. From the Team Explorer page, we can see an extra option titled 'ConflictDetector'. After we select our tool, we are redirected to the new panel, from where we can set up configuration and see relevant information related to our workspace. Beside configuration data, we perceive information about all existing forks and relationship between them and our local workspace. Each fork contains title with owner username and relationship status awarded with an icon. The following snapshot 5.3 shows the panel and 5.1 possible relationship icons.

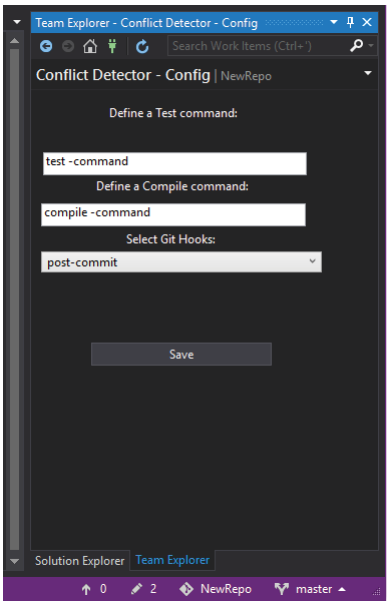

Figure 5.2: Configuration of the plugin

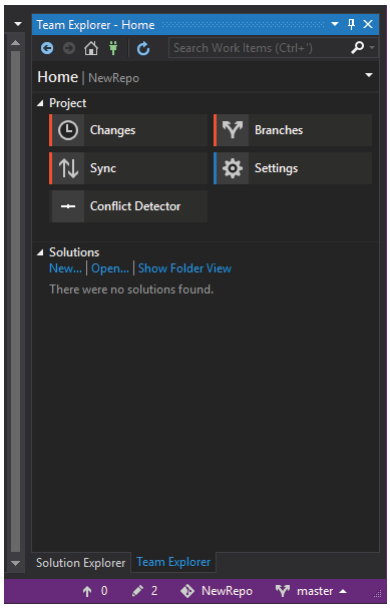

Figure 5.3: Plugin's panel

5.4.4 Auto-committer

As soon as we run our tool, an auto-commit handler will start its job. It works as a watcher on our workspace, with given interval it will wait for changes. For every CRUD operation the handler will count the number of changes and in case when the number is above zero, a commit will be performed. The watcher looks for any created, modified or renamed files. The code A.1 gives a brief explanation on how auto commit handler works.

Since we are defining the time interval for auto commits, it could be changed afterwards, but after a lot of tests, we settled for 60 seconds as default interval A.2.

The code A.2 catches changes during the interval, every change will be caught and stored in a HashSet. When the time runs out and if a number of changes is positive, the changes will be committed. Afterwards, a new check cycle starts.

5.4.5 Auto-pusher

The auto push job works similar as auto commit, but it pushes changes to a remote location, in our case the developer's fork. The fork location should be defined during the configuration step. If the fork URL is valid, the committed changes will be pushed to the remote location. The code A.3 represents an auto push handler. Again, the time interval is set to 300 seconds by default.

5.4.6 Workflow

Once our tool is configured and started, the whole cycle works in the background. The aim is the last step and it works as a checker for possible merges. Going over all forks and comparing them with our local workspace, depending on the relationship between each two, the tool provides a corresponding output. To access all forks of single repository we use a GitHub API client library for .NET known as Octokit A.4.

The code above allows us to access all forks of master repository. All forks are stored in forks result atribute. Since we got all forks, we can iterate over them and analyze them. Analyzing process is pretty straightforward, considering that we have all forks and master branch.

The next step is to analyze repositories and recognize different relationships between them. As already mentioned, we have three base relationships: ahead, behind and same.

Chapter 6

Evaluation

Solving a problem is one thing, evaluating how good the solution is, is another thing. Here we describe the evaluation, which displays that our solution does not overburden developers with annoying notifications and boosts early conflict detection in contrast to other tools and approaches constructed on changes and awareness. In addition, our tool is conceived on mentioned approaches and tools, by extracting the best side of each of them.

In our evaluation we conducted a few experiments that analyze the usability and success of our tool through different tasks in order to minimize flaws that were found by various participants. First of them was a small experiment carried out with two participants. The second one was based in a real-life environment within a team of several developers, working on an actual project. In addition, we added a questionnaire to the mentioned team, which contained specific questions for the purpose of getting better feedback of the tool.

6.1 Experimental design

To show the first experiment, we executed several administered user experiments involving all possible ways of execution that correlate with the particular level for conflict detection. Having two participants, we created a test suite consisting of all mentioned relationships between repositories. Knowing that developers have to work in parallel in order to realize the experiment, every attempt led to a specific repository relationship. Using this strategy, we wanted to estimate the number of occurrences detected in each attempt and the notification overwhelm.

The second experiment involved a whole team of developers, computer science graduate students from the university, with an adequate level of experience with relevant tools for the experiment, i.e. Git, C#, Visual Studio, .NET, etc. The experiment was executed simulating a workflow of collaborative software development, where the team members were doing their work, each of them programming in their distinct workspaces, randomly leading to specific relationships and awareness notification. The experiment lasted for a day, and during that time the participants were aware of their involvement in the experiment and the outcome shall describe how they managed conflicts appearance inside simultaneous programming tasks. The project they were working on was a .NET based application with a total of 12 classes and 924 lines of code, which they were familiar with. Before the experiment started, to familiarize themselves with the tool, participants were introduced with two real-time sessions with the tool behavior during which possible events may occur. Afterwards, each of them was assigned a fully described task, which they have to follow in a

specific order and to note every relevant event for the experiment. Each task was conceived to complete concurrently and to result a specific behavior, so the tool can catch it and guide the developer through. All participants signed that they understood their tasks and that they would honestly perform their job. Finally, after the whole experiment was ended, the participants received a questionnaire including 5-point Likert scale and 10 questions. The questionnaire was designed with a goal to gain better insight of the tool and to realize shortcomings, and it takes no more than 15 minutes to fill out. In these cases of evaluations, specific inequality among various participants may influence the result of an experiment, and blur the upshot that tool is supposed to yield. The main differences that disturb our evaluation are developer's technical skills and the foreseen divergence when conflicts occur. It could be the order or velocity at which a developer attempts to finish his task. Because of the listed, we clearly defined our experiment with the goal to reduce the influence of these differences. Everything was done with the respect to developer's technical skills.

The main goal of our experiments was to imitate collaborative software development with possible conflicts that may occur, thus letting us perceive how specific person notes a conflict and which steps a person takes to resolve the conflict. Since the nature prevents us to follow more than one developer at a time, because each of them works in their own local workspace, we are using their notes to have everything under our control. In addition, any interaction among the team is also noted, and it is helpful to get a better picture of the whole concept.

6.2 Results

The results collected from our experiments reveal how merge conflicts and the awareness during collaborative software development impact developers' activities, specifically coordination between developers in the same team. In this section we present the results that have been extracted from a combined analysis of the two experiments and the questionnaire (described in the previous chapter).

Also, we want to add a point to the popular problem of merge conflicts [8] [24] [7] [5], since developers are trying to push their local changes into a common repository, they may run into inconsistent changes that have to be harmonized. Additionally, here we will discuss the frequency of occurred conflicts, the time needed for a conflict to be resolved, and what the origin of conflicts is, aiming at local or remote team member.

Frequency. To get a better idea of how often merge conflicts occur, we get out all relevant data from developers' notes they have recorded during the project development. The 6.1 table recaps the questionnaire feedbacks. Altogether, above 94% of the developers had to manage some merge conflicts, and only few of developers stumbled into them repeatedly.

Significance. In order to get an idea of the merge conflicts harshness, we put a question in the questionnaire to understand how much time each of the developers spends in a single conflict resolution, thinking about the works and average case. The table 6.2 combines and summarizes the developer's notes and the questionnaire feedbacks.

Generally, in our case conflicts did not take a lot of time to be resolved, around 70% of our developers spent less than 10 minutes on average to resolve a conflict. Yet, looking at the worst case side of fixing a conflict, we can say that it may take significantly more time, after all almost half of developers needed above 20 minutes to solve a conflict.

Origin. Considering that we have organized our experiment so we have local and remote members, and local pair programming, with the aim to evaluate whether a conflict occurred

Number of conflicts	Number of responses	% - percentage
0	1	5.9
1-3	6	35.3
4-6	5	29.4
7-9	3	17.6
10+	2	11.8

Table 6.1: Number of conflicts in the merging phase occurred during the development period

Time (minutes)	Worst case		Average case	
	Responses	% - percentage	Responses	% - percentage
<=1	0	0	1	5.9
1-4	2	11.8	5	29.4
5-9	3	17.6	6	35.3
10-19	4	23.5	2	11.8
20+	8	47.1	3	17.6

Table 6.2: Worst/Average case spent for resolving a single merge conflict

due to a remote member, a local one or the paired members, we added a question asking what the source of most conflicts is: a local team member, a remote team member, or pair programming side. However, the reason for using pair-programming is to check whether it leads to rare conflicts occurrence, knowing that work in pairs removes any cases of unwanted interaction between team members. Table 6.3 sums up the questionnaire answers and the developer's notes.

Most conflicts with:	Responses	% - percentage
Local team members	7	43.7
Remote team members	6	37.5
Pair-programming members	3	18.8

Table 6.3: Origin of conflicts, local vs. remote vs. pair-programming members

Looking at remote and local team members, we can say that the difference between them is not that huge and apparently not appreciable, and this can point that the odds of conflicts rely more on additional elements, than developer's site. More notably, the previous responses reveal that the works of pair-programming members as the main source of conflicts demonstrate that this practice is preferable to use, since it leads to less that 20% of merge conflicts.

Our goal was based on using individual repositories for each developer, letting them have access to the work of other team members, while the Git features allow them to perform their work undisturbed. All-inclusive, we can say that our experiment did not find significant differences between remote and local team member's work as the origin of conflicts, but the tool helps them to decrease the final number of conflicts.

If awareness is deficient, interruptions in developing workflow occur quite often in teams

of significant number of developers. In such cases, more common interruptions are related to missing or incorrect functionalities, and occasional due to lack of documentation. Furthermore, these workflow interruptions have a bad influence on motivation, productivity, and they are even time consuming. The majority of them demand time to collect missing information in order to continue working. Knowing that workflow interruptions may be a dangerous thing, we ask developers, whether they would encourage tools that forge awareness information regularly than continue using typical tools. To get an answer to the question how frequent and detailed awareness information should be, we ask developers how useful it would be to have knowledge about the work of other developers before they push their work to the repository. The following table 6.4 shows the answers.

How helpful?	Not helpful	Less helpful	Neutral	Helpful	Very helpful
Responses	0	2	4	6	5

Table 6.4: How helpful would it be to have awareness information?

Altogether, summarizing the previous table, we can say that there is some fondness, but not a very strong one. In addition, it may be hard for developers to visualize how precisely awareness information would be presented.

To improve our product, we asked developers to choose what time is the preferred one when information on other developers' work should be shown. They got several "times": having real-time access to the other developers' repository, only when a developer commits, and only when a developer pushes. The table 6.5 below summarizes the answers.

Time	Responses	% - percentage
Real time	4	23.5
Commit time	6	35.3
Push time	7	41.2

Table 6.5: When should awareness notifications pop up?

Looking at the table, we can say that the majority of developers would like to be notified at the push time, following by committing time, and the least chosen option was real time. All over, the answers indicate that real-time awareness may lead to distraction rather than being useful. The drawback of real time awareness is mostly related to larger teams, since it leads overwhelming and complex interactions.

Last, the amount of data that should be displayed is crucial, after all it is closely tied with useful or useless notification. It represents the dimension of "details", so we asked developers to choose the preferred level of details that should be included in a notification. We offered following options: full-detail of all changes to the code, all affected classes, all affected packages/modules, all affected subroutines (functions, methods, procedures, etc.), and without any information related to the change.

Table 6.6 recap the question.

Levels	Responses	% - percentage
Full detail level	6	35.3
Subroutines level	5	29.4
Classes level	4	23.5
Packages level	1	5.9
No info	1	5.9

Table 6.6: What should be the level of details of awareness information?

After observing the previous table, we can say that developers generally tend to choose more detailed notifications. That means, that a single notification should contain all available information within the tool, with an additional option for user to adjust what can be displayed, since one thing does not fit for everyone and different developers may have individual preferences concerning the level of details that a tool should display.

Generally speaking, merge conflicts will occur always, but with the tool they are not very periodic, the majority of them have finite weight, and they take a small amount of time for resolution. Additionally, the location of team member does not appear to have a crucial consequence on the probability that their work will lead to conflict.

Chapter 7

Discussion

In this section we will discuss what the implications of the entire work are and we are reflecting back on the key lessons learnt from the evaluation and their outcomes for designing future similar tools. We are making a step back and rethinking what we did and how it contributed to the problem we stated.

Today's user interface designs for fostering awareness tools [19], such as ours, are based on the assumption that developers should be regularly updated on side-by-side changes in progress, so they are informed of changes and the time they arise. Notwithstanding, the review of our experiment shows that participants usually need the awareness hint provided by the tool only at a precise point during programming. It signifies that continuous flow of information or repeated notifications may not be needed. For example, it may be achievable to prepare information and show it only when a developer shifts to work on another artifact. As an additional example, it will be advantageous to provide a simple view that displays recap of histories. This mentioned example will improve awareness system in workspace, it might be cheaper and leads to reduction of information overload.

However, neither solution can be selected as universal one, since every user is one of a kind, and each of them has a particular set of preferences in how they would like to be enlightened. During our experiment, we found that developers would like more detailed notification about specific events. Our participants were mainly oriented to their programming tasks, focusing completely on the environment. Because of this engross, some developers may occasionally overlook notifications that were delivered by the tool. To provide to this group of users, we have to implement more straightforward notification process, like notifications that instantly emphasize the user's awareness of the relevant event. Therefore, instantly giving notice is a clash between the user's attention and distraction, and it is a critical part during the design of such notifications, in addition it may introduce various notification techniques regarding to different type of events.

It turns out that, regardless of the availability of similar tools such as ours, developers after all continue using causal strategies for conflict reduction. Some of these strategies are soft locks and partial commits [26], approaches without any support. Our examination displays that, by using such tools as ours, developers are guided to improve these strategies. The tips from these tools [26] [19] represent an important part of development process, since developers are able to accept some of the strategies and mix them so they suit them best. This mix can be pursued even further, differing per conflict, per developer and per pair developers

Looking over to the interaction between developers, they can establish communication

on some changes or a certain task rapidly. Owing to the fact that all team members share the same information on a change, each team member is aware and they can start a conversation. We strongly believe that awareness tools could improve a lot in the future, so they can expand their support systems and provide enhanced conversations and other sides. E.g. such systems may supply thorough background and distinct information on changes and allow various users to access them and help in coordination process.

A profound after-match of using such tools is of the social kind, where users feel closer to their colleagues and not lonely at all. Therefore, using these tools will intensify a feeling of shared existence within a team, unlike in cases where developers used SCM systems. Moreover, developers are looking to be more aware about others' activities, and they want to make additional effort to make their collaboration more productive (e.g. informing coworkers about a possible issue, tending to resolving issues instead of overwriting others' changes, etc.). Further, readiness to put additional work signifies that developers are seeing the advantage of the effort they make, so they are more likely to reach a few more steps (e.g. adding comments to a commit, linking a commit with a corresponding task, etc.) in order to make awareness technology more productive.

In the end, we bear in mind that the amount of information is crucial for awareness tools. As said before, we found that developers rely on awareness hints in order to synchronize their activities (e.g. changing a task, delaying a task, using a placeholder, etc.). Some of the participants noted that they start conversations only when there is insufficient information for them provided by the tool. We know that more communication means better coordination. However, users are able to obtain necessary information through other sources and then take the corresponding action. As a matter of fact, the existence of extra communication may also be a proof of fruitless coordination strategies being used. This should be taken into account in the future work.

Chapter 8

Conclusion and Further Studies

This chapter shows a brief summary of the *thesis*. All conclusions are directly related to the statement from the chapter 3.

8.1 Conclusion

The main issue of merging conflicts during collaborative development is that an important Merge may lead to crucial software faults [9] [28]. After the industry and research took their hands on the problem they acclaimed that an earlier detected conflict is way easier to resolve than encountered conflict at merge or in production [9] [13]. Exploring future development states of software, also known as speculative analysis [34], over some version control systems prepares accurate information about possible conflicts between members in the collaborative software development team. Such conflicts are known as textual, build, and test, and they are highly possible to arise, apart from the situation when a developer changes a committed file.

Thereupon awareness has been suggested to guide developers in earlier conflict detection [9] [28] [24]. Being informed about conflict as early as possible allows developers to prepare themselves, making a better judgment on how to solve it. Nonetheless, every known approach demands interaction with developer to detect conflicts, and it may lead to overbearing them with notifications, and making it even more problematic.

Our goal was to propose a solution on behalf of developers, a tool that helps in earlier conflict detection. It follows all existing workspaces and changes inside them, continually checking them and testing inside the IDE. This allows earlier conflict solving while developers still have fresh minds, and furthermore every fix is way easier. An empirical evaluation validated that our approach actually helps in early detection of conflicts and avoids overwhelming developers with notification compared to existing solutions. Moreover, our tool gives solid information about possible conflicts while staying mostly humble.

Many different adaptations, tests, and experiments have been left for the future due to lack of resources and time (i.e. the experiments with real data, a group of developers and environment are usually very time consuming, requiring days and weeks to finish them). Future work concerns deeper analysis of particular approaches, new proposals to try different methods, or simply curiosity.

On one side, collaborative software development is fundamental, but there are always some drawbacks, such as problematic expression. The first empirical and functional step of collaborative software development is to create relevant and accurate information available

for developers, which allows the developers to locate and resolve conflicts before they occur, followed by cost reduction and reduction of effort [30].

8.2 Future work

There are some ideas that I would like to try during the design and the implementation of the tool. However, there is always a possible direction for the work, and our future work is concerned with a number of various directions.

Firstly, we know that our tool works in detecting possible conflicts before they actually occur, so our challenge is to continue improving tools' feature and usability, to extend the range of all possible conflicts.

Secondly, we want to do a study with professional development in an industry, since our current evaluations are limited in not considering the possible outcome that takes place when the tool is used for a prolonged period of time on a real system that the developers are working with. There are a lot of factors that can only be comprehended when the tool is used in a live project over a long time period. In addition, we would like to discover the impact of continuous merging in their software process, and to investigate if new collaborative patterns appear.

Finally, it will be preferable to estimate the final effect of continuous merging on the quality of software. Moreover, to improve the tool behavior on behalf of remote teams, they are geographically distributed. In such cases, the tool needs to involve additional strategies, to make it easier for the teams.

Appendix A

Code listings

```
public AutoCommitHandler(int intervalSeconds, string folder)
    {
      _folder = folder;

      Console.WriteLine("Watching {0} for changes", _folder);

      _watcher = new FileSystemWatcher(_folder) {IncludeSubdirectories = true};
      _watcher.Changed += watcher_Changed;
      _watcher.Created += watcher_Created;
      _watcher.Renamed += watcher_Renamed;

      _watcher.EnableRaisingEvents = true;

      _timer = new Timer(intervalSeconds*1000);
      _timer.Elapsed += _timer_Elapsed;
      _timer.Start();
    }

        public AutoCommitHandler(int v, IGitRepositoryInfo gitRepositoryInfo)
        {
            this.v = v;
            this.gitRepositoryInfo = gitRepositoryInfo;
        }
```

Listing A.1: Auto-commiter

```
1  private void _timer_Elapsed(object sender, ElapsedEventArgs\
       e)
2      {
3        if (_changes.Count == 0)
4          return;
5
6        try
7        {
8          _timer.Stop();
9
10         var changes = new HashSet<string>();
11
12         string result;
13         while (_changes.TryTake(out result))
14           changes.Add(result);
15
16         if (changes.Count > 0)
17         {
18           foreach (var file in changes)
19           {
20             //no file...
21             if (!File.Exists(file))
22               continue;
23
24             Console.WriteLine("Committing␣changes␣to␣{0}", \
                   file);
25             RunGit("add␣\"" + file + "\"");
26           }
27
28           var commitMessage = BuildCommitMessage();
29
30           RunGit("commit␣--file=-", commitMessage);
31         }
32       }
33       finally
34       {
35         _timer.Start();
36       }
37     }
```

Listing A.2: Timer for commits

```
public AutoPushHandler(int intervalSeconds, string folder, string remoteURI)
        {
            _folder = folder;

            Console.WriteLine("Setting a remote for", _folder);

            RunGit("git remote set-url origin", remoteURI);

            _timer = new Timer(intervalSeconds * 1000);
            _timer.Elapsed += _timer_Elapsed;
            _timer.Start();
        }
```

Listing A.3: Auto-pusher

```
var client = new GitHubClient(new ProductHeaderValue("my-app"));
Task<IReadOnlyList<Repository>> forks = client.Repository.Forks.GetAll(repo_owner, repo_title);
```

Listing A.4: Octokit

Bibliography

[1] Oscar Karnalim. *Language-Agnostic Source Code Retrieval Using Keyword & Identifier Lexical Pattern.* International Journal of Software Engineering and Computer Systems (IJSECS), 2018.

[2] Ronald W. Hepburn. *The Encyclopedia of Philosophy.* MacMillan Reference USA (Gale), 2005.

[3] David Chisnall. *The Challenge of Cross-language Interoperability.* ACM New York, NY, USA, 2013.

[4] Todd Malone. *Interoperability in Programming Languages.* Scholarly Horizons: University of Minnesota, Morris UndergraduateJournal: Vol. 1: Iss. 2, Article 3., 2014.

[5] Lisa Burk & Jean Richardson. *Conflict Management in Software Development Environments.* Pacific Northwest Software Quality Conference, 2000.

[6] Jim Whitehead. *Collaborative Software Engineering.* Springer-Verlag Berlin Heidelberg, 2010.

[7] D. E. Perry & H. P. Siy. *Parallel changes in large-scale software development: An observational case study.* ACM Trans. Softw. Eng. Methodol. vol. 10, 2001.

[8] Y. Brun & R. Holmes. *Proactive Detection of Collaboration Conflicts.* ACM ESEC/FSE '11, 2011.

[9] Mario Luıs Guimaraes & Antonio Rito Silva. *Improving Early Detection of Software Merge Conflicts.* ICSE '12 Proceedings of the 34th International Conference on Software Engineering, 2003.

[10] M. Fowler. Continuous integration, May 2006. `http://martinfowler.com/articles/continuousIntegration.html`.

[11] M. Fowler & P. Duvall and S. M. Matyas. *Continuous Integration:Improving Software Quality and Reducing Risk.* Addison-Wesley Professional, 2007.

[12] Alexandr Savinov. *Concept-oriented programming: from classes to concepts and from inheritance to inclusion.* Cornell University Library, 2015.

[13] Maysoon Aldekhail & Azzedine Chikh and Djamal Zian. *Software Requirements Conflict Identification: Review and Recommendations.* (IJACSA) International Journal of Advanced Computer Science and Applications, 2016.

[14] Andres Loh & Wouter Swierstra and Daan Leijen. *A Principled Approach to Version Control*. Microsoft Research, 2005.

[15] M. Fowler & P. Duvall and S. M. Matyas. *Awareness and Coordination in Shared Workspaces*. ACM, CSCW '92, 2007.

[16] T. Schummer & J. Haake. *Supporting Distributed Software Development by Modes of Collaboration*. Kluwer Academic Publishers, 2001.

[17] P. Dewan & R. Hegde. *Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development*. Springer, 2007.

[18] A. Sarma & G. Bortis. *Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces*. ACM, ASE '07, 2007.

[19] J.T. Biehl & M. Czerwinski. *FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams*. ACM, CHI '07, 2007.

[20] Prasun Dewan. *Dimensions of tools for detecting software conflicts*. ACM, RSSE '08, 2008.

[21] Jim Whitehead. *Collaboration in Software Engineering: A Roadmap*. FOSE'07 IEEE, 2007.

[22] Jim Whitehead. *Collaborative Development Environments*. Springer-Verlag Berlin Heidelberg, 2010.

[23] Grady Booch and Alan W. Brown. *Collaborative Software Engineering: Concepts and Techniques*. Rational Software Corporation, 2002.

[24] Y. Brun & R. Holmes & Michael D. Ernst and David Notkin. *Early Detection of Collaboration Conflicts and Risks*. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2013.

[25] Jon Loeliger. *Version Control with Git*. O'Reilly Media, 2009.

[26] A. Sarma & D. F. Redmiles and Andre van der Hoek. *Palantir: Early Detection of Development Conflic ts Arising from Parallel Code Changes*. CSE Journal Articles, 2012.

[27] Khaleda Yasmin & Aleya. *Workplace conflicts: Classifications, causes and management strategies*. International Journal of Academic Research and Development, 2017.

[28] H.Christian Estler & Martin Nordio & Carlo A. Furia and Bertrand Meyer. *Awareness and Merge Conflicts in Distributed Software Development*. IEEE, 2014.

[29] M.L. Guimaraes & A. Rito-Silva. *Towards Real-Time Integration*. ICSE Workshop, 2010.

[30] Lile Hattori and Michele Lanza. *Syde: A tool for collaborative software development*. ICSE, 2010.

[31] Jan Wloka & Barbara Ryder & Frank Tip and Xiaoxia Ren. *Safecommit analysis to facilitate team software development*. ICSE, 2009.

[32] Iman Keivanloo & Christopher Forbes & Aseel Hmood & Mostafa Erfani & Christopher Neal & George Peristerakis and Juergen Rilling. *A Linked Data Platform for Mining Software Repositories*. Conference on Mining Software Repositories, 2012.

[33] Francisco Zigmund Sokol & Mauricio Finavaro Aniche and Marco Aurelio Gerosa. *MetricMiner: Supporting researchers in mining software repositories.* IEEE, 2013.

[34] Y. Brun & R. Holmes & Michael D. Ernst and David Notkin. *Speculative Analysis: Exploring Future Development States of Software.* FoSER, 2010.