UNIVERSITY OF ZURICH

BACHELOR THESIS

Improving Build Failure Resolution Through In-IDE Assistance

Alex Scheitlin

of Bülach, Switzerland (15-709-959)

supervised by

Prof. Dr. Harald C. Gall
Dr.-Ing. Sebastian Proksch
Carmine Vassallo

at

Software Evolution and Architecture Lab
Department of Informatics

October 30, 2018

UNIVERSITY OF ZURICH

BACHELOR THESIS

Improving Build Failure Resolution Through In-IDE Assistance

Alex Scheitlin

Bachelor Thesis

Author: Alex Scheitlin

Project Period: 01.05.2018 - 01.11.2018

Software Evolution and Architecture Lab
Department of Informatics
University of Zurich

Acknowledgements

First and foremost, my thanks go to Dr.-Ing. Sebastian Proksch and Carmine Vassallo from the Software Architecture and Evolution Lab at the University of Zurich for their support and guidance throughout my thesis. Thank you for the regular meetings, insightful discussions, and for letting me benefit from your experience and expertise. Especially, I appreciate your valuable inputs and the liberty you gave me to shape this thesis into a piece of my own work.

Further, I thank Prof. Dr. Harald C. Gall, dean of the Faculty of Business, Economics, and Informatics, director of the Software Evolution and Architecture Lab, and professor of software engineering at the University of Zurich, for giving me the opportunity to write this thesis.

Special thanks go to all the participants of the controlled experiment for helping me evaluate CAESAR and giving me feedback on my work.

Also, I thank my colleagues Moritz Eck and Nik Zaugg for discussing approaches and ideas.

Last but not least, I thank my parents for their support and encouragement during my bachelor studies.

Abstract

Continuous integration is an essential part of modern software engineering and helps programmers to automate their build process (*e.g.*, compiling source code, packaging binary code, or running tests). While developers benefit from earlier caught bugs, more stable code bases, and shorter software release cycles, they still face several problems when working with continuous integration. For example, if a software build fails, additional time is needed to resolve detected problems by build servers (*i.e.*, build failures). Developers first need to locate the root of failure and then fix the error. For that purpose, they leave their Integrated Development Environment (IDE) and scan through long log files on the build server.

To simplify and reduce the effort required for this process, the vision is to bridge the gap between the build-related information available on the build server and the local development environment where the source code needs to be fixed. Therefore, CAESAR (Ci Assistant for (Build Failure) Resolution and Summarization) was developed, which is an IDE plugin that leverages detailed build-related information to support developers in debugging build failures directly within their IDE. The tool summarizes build logs, classifies errors, shows the files in which the errors occurred, gives hints or error descriptions, and enables to directly debug the error. This is all possible without leaving the IDE.

To evaluate the usefulness of CAESAR, a controlled experiment was conducted, showing that on average, developers could resolve build failures 48.4% faster when working with CAESAR. Especially, the evaluation of the experiment showed that developers working with CAESAR were able to reduce the time needed to identify the error in the build log and locate it within the source code. Lastly, participants stated that, thanks to CAESAR's assistance, context switches between their own IDE and the build server were no longer required.

Zusammenfassung

Continuous Integration bildet ein wesentlicher Bestandteil der modernen Softwareentwicklung. Sie hilft Programmierern das Erstellen von Software zu automatisieren (z.B. Kompilierung von Quellcode, Abpacken von Binärcode oder Durchführen von Tests). Während Entwickler von früher identifizierten Fehlern, stabilerem Quellcode und kürzeren Softwarerelease-Zyklen profitieren, werden sie bei der Anwendung von Continuous Integration Methoden immer noch mit einigen Problemen konfrontiert. Wenn beispielsweise ein Software-Build fehlschlägt, wird zusätzliche Zeit benötigt, um die von Build-Servern identifizierten Probleme (Build-Fehler) zu lösen. Entwickler müssen zuerst die Ursache des Build-Fehlers finden und anschliessend den Quellcode verbessern. Dazu verlassen sie ihre Entwicklungsumgebung und durchsuchen Logdateien auf dem Build-Server.

Um diesen Prozess für Entwickler zu vereinfachen und den dafür benötigten Zeitaufwand zu reduzieren, ist es das Ziel, die Lücke zwischen den auf dem Build-Server verfügbaren Informationen und der lokalen Entwicklungsumgebung, in welcher der Quellcode repariert werden muss, zu schliessen. Dazu wurde CAESAR (Ci Assistant for (Build Failure) Resolution and Summarization) entwickelt, eine Erweiterung für Entwicklungsumgebungen, welche zentrale Informationen über den Software-Build nutzt um Entwickler lokal bei der Fehlerbereinigung zu unterstützen. Das Programm fasst Build-Protokolle zusammen, klassifiziert Fehler, zeigt die Dateien an, in welchen die Fehler aufgetreten sind, gibt Hinweise oder Fehlerbeschreibungen und ermöglicht es, den Fehler direkt Schritt für Schritt zu reproduzieren. All dies ist möglich ohne die Entwicklungsumgebung zu verlassen.

Um den Nutzen von CAESAR zu evaluieren, wurde ein kontrolliertes Experiment durchgeführt, welches zeigen konnte, dass Entwickler fehlgeschlagene Builds im Schnitt 48.4% schneller reparieren konnten, wenn sie mit CAESAR arbeiteten. Die Auswertung zeigte ebenfalls, dass es Entwicklern durch die Unterstützung von CAESAR gelungen war, die notwendige Zeit, um die Fehler zuerst zu finden und zu verstehen und anschliessend im Quellcode wiederzufinden, zu reduzieren. Zusätzlich gaben die Teilnehmer der Studie an, dass dank CAESAR kein Kontextwechsel zwischen der eigenen Entwicklungsumgebung und dem Build-Server mehr nötig war.

Contents

1 Introduction **1**

2 Related Work **3**
2.1 Build Failures and Continuous Integration 3
2.2 Existing Tools for Build Failure Resolution 4

3 CAESAR **7**
3.1 Barriers in the Continuous Integration Workflow 7
3.2 Build Log Abstraction Through a Meta-Model 9
3.3 Augmented Continuous Integration Workflow with CAESAR 10

4 Controlled Experiment **15**
4.1 Procedure 15
4.2 Results 17
4.2.1 RQ1: Build Failure Resolution Time 18
4.2.2 RQ2: In-IDE Assistance 19
4.2.3 RQ3: Instant Test Debugging 22

5 Discussion **23**
5.1 Implications and Future Work 24
5.2 Threats to Validity 25

6 Summary **27**

A Attached CD **29**

List of Figures

3.1 Continuous Integration Workflow 7
3.2 Meta-Model 10
3.3 Augmented Continuous Integration Workflow with CAESAR 11
3.4 Overview of All Builds 12
3.5 Summarization of a Build 13
3.6 Abstraction of a Build Log 14

4.1 Highest Qualification and Current Position of Participants 16
4.2 Experience of Participants with Study-Relevant Tools 17
4.3 Practical Relevance of Error Types and Representativeness of Introduced Errors . . 18
4.4 Recorded Build-Fix-Times and Average Time Reduction per Error Type 19
4.5 More Time-Consuming Task without and with CAESAR 20
4.6 Assistance Through CAESAR in General, Through Build Log Summarization, and with Error Descriptions and Hints 21

List of Tables

4.1 Problem Types and Introduced Errors 16

A.1 CD Content Concerning CAESAR (at: *appendix\A-caesar*) 29
A.2 CD Content for the Controlled Experiment (at: *appendix\B-controlled-experiment*) . 29

List of Listings

3.1 Detected Dependency Issue 12
3.2 Detected Compilation Error 13
3.3 Detected Test Failure 13

Chapter 1

Introduction

In contemporary software engineering, *agile software development*[1] is a widely known and applied practice [10]. Its state of the art methods provide effective approaches for developing high-quality software and enable fast adoption to evolving requirements and solutions during the software development process [6]. As a central discipline of agile software development, *Continuous Integration (CI)* assists developers in merging their source code continuously (*e.g.*, several times a day) and ensures that changes integrate properly [10,17]. This results in (i) more stable code bases, (ii) earlier caught bugs, and (iii) shorter software release cycles [8].

Despite these benefits, developers face several barriers when using CI [7]. In case of a failure on the build server, developers need to investigate and analyze the provided build log on the build server as well as the underlying source code. Often, finding and understanding the point of failure is very complicated and time-consuming, as build logs can be very large (*e.g.*, hundreds of thousands of lines) [7]. In addition, the necessary information for locating and understanding the error is often distributed over the whole log.

Recent research proposed tools to foster the resolution of build failures. BART [18], for example, summarizes build logs of broken builds and suggests hints, based on information found on the internet, to fix them. Another tool, BUILDMEDIC [11], automatically repairs failed builds related to dependency issues. Both approaches aim to be either integrated into the build server or used locally.

However, within the CI workflow, the local environment and the remote build server are two separated entities. While the source code is written in an *Integrated Development Environment (IDE)* on the developer's local machine, the software is built on a build server that, for example, has different software installed, runs on another operating system, or has restricted access to remote networks and resources. As build failures arise on the build server but need to be fixed within the local environment, developers need to switch context and try to reproduce the error locally. Thus, it is difficult to locate, understand, and debug build failures. Moreover, developers often do not have full control over the build server, which further complicates the retrieval of relevant information for fixing build failures.

The vision of this thesis is to bridge the gap between those two isolated nodes in order to support developers in locating and fixing build failures. Developers should be able to access all build-related information right next to the source code and it should no longer be necessary to leave the IDE to fix a build failure. This would reduce the complexity of detecting and understanding build failures and let developers focus on the actual problem: fixing the source code. Moreover, the reduced time needed to fix the issues would increase developer productivity and make software release cycles even faster.

[1] Manifesto for Agile Software Development: http://agilemanifesto.org/

This thesis makes a first step towards this vision by proposing a novel approach to foster build failure resolution; through CAESAR (Ci Assistant for (Build Failure) Resolution and Summarization). The IDE plugin is based on a meta-model abstracting plain text build logs and allows developers to directly resolve build failures on their local machines without accessing build logs on the build server. CAESAR supports developers by (i) summarizing build logs, (ii) classifying errors, (iii) showing the files where the errors occurred, (iv) giving fixing hints or error descriptions, and (v) enabling to directly debug the error. Additionally, before starting to fix the detected errors, CAESAR provides the possibility to immediately save the current changes to the source code the developer is working on and instantly loads the broken code base into the IDE.

To analyze and evaluate the usefulness and benefit of CAESAR to developers, a controlled experiment was conducted, where participants were asked to solve sets of build failures with and without the assistance of CAESAR. The following research questions address the fields of interest:

RQ1: *Can in-IDE assistance reduce the time required to fix a build failure?*

RQ2: *Why can broken builds be resolved faster with in-IDE assistance?*

RQ3: *Does instantly debugging a failed test help fixing a build failure?*

By proposing a new approach to foster build failure resolution through assisting developers during the fixing process within the IDE, this thesis makes the following contributions:

- Build abstraction, representation, and summarization by parsing build logs.
- Error detection and classification by analyzing build logs.
- Bridging the gap between the remote build server and local development environment through in-IDE assistance and integration.

This thesis is structured as follows. In Chapter 2, previous study findings concerning build failures and continuous integration are summarized and similar, existing tools are presented. Chapter 3 first identifies barriers developers are faced with in the continuous integration workflow and introduces a meta-model abstracting all build-related information to make them easily accessible. Then, the augmented workflow with CAESAR is illustrated. In Chapter 4, the set up of the controlled experiment as well as the results and answers to the research questions are given. The discussion of the findings of this work, implications for continuous integration and research, thoughts about future work, and threats to the results validity are discussed in Chapter 5. Finally, Chapter 6 concludes this thesis by summarizing the work and highlighting the key findings and contributions.

Chapter 2

Related Work

This chapter first summarizes study findings related to *build failures* and *continuous integration* (Section 2.1). For one thing, the presented findings deliver collected insights into failing builds and the necessity to resolve them, and, for another, they provide essential knowledge used to develop CAESAR. Then, *existing tools* from research and industry assisting in fixing build failures are introduced and compared with CAESAR's approach (Section 2.2).

2.1 Build Failures and Continuous Integration

Software Build Process Analysis. At Google, Seo *et al.* [16] conducted an empirical study to learn more about compiler errors in software build processes (*i.e.*, why and how frequent builds fail and how much effort a fix demands). Therefore, 26.6 million builds, produced by thousands of developers, were processed. The authors refer to the *edit-compile-debug* programming cycle, where developers first *edit* the source code to apply the necessary changes, then *compile* it to verify that the changes are integrated properly, and finally *debug* occurred errors to fix them by *editing* the source code again. This process is repeated until the program acts as anticipated. Thus, the faster one cycle is, the faster the whole development process will be. This illustrates the importance of fast and successful compilations and hence, software builds. The study identified that around 10% of the error types caused 90% of the build failures and that dependency related errors were the most common ones. Furthermore, the median percentage of build failures was 38.4% for C++ and 28.5% for Java, which means, that on average, every third build failed. To fix most of the errors, two build iterations were needed (5 minutes for C++ and 12 minutes for Java).

Rabbani *et al.* [13] replicated this study in the Visual Studio context of the Mining Software Repositories Challenge[1]. They analyzed 13,300 builds to examine how often builds fail and how long it takes to fix them. This replication showed, that build failures occurred in 9.2% to 13.2% of the builds (67%-76% less frequent) and that they were fixed in 2.7 minutes (46%-78% faster).

These experiments show that the build failure rate can be very high and thus, failing builds are still a big issue. This motivates to foster their resolution and focus on how developers can be supported.

Build Failures in Open-Source Software. Rausch *et al.* [14] analyzed 54,248 log files from 14 open-source Java projects by linking repository commits to data of the corresponding CI builds to learn more about why builds fail. They identified 14 common error categories, whereby test failures were by far the most common reason why a build failed (up to 80%). Furthermore, fre-

[1] Mining Software Repositories Challenge: https://2018.msrconf.org/track/msr-2018-Mining-Challenge

quent errors were attributed to compilation or quality. The build failure ratio went from 14% to 69% with a mean of 38%, which is similar to the study results of Seo *et al.*

Beller *et al.* [1] studied 2,640,825 Java and Ruby builds of open-source software to gain insight into testing practices with CI. They found that most build failures were caused by failing tests. In addition, they identified that CI is not an adequate substitute for local testing and that it creates a latency standing in contrast with local in-IDE testing.

Hilton [8] *et al.* studied the usage of CI in open-source projects. They surveyed 442 developers and analyzed 34,544 open-source projects with a total of 1,529,291 builds. Identified benefits were, amongst other things, (i) earlier caught bugs, (ii) more stable code bases, (iii) easier integration, and (iv) faster releases (more than twice as often as when not using CI). In contrast to these benefits Hilton *et al.* also identified several open issues of CI, for example, the lack of assistance during the debugging process of build failures.

These three studies show that failing tests are a common reason why builds fail and that developers are still missing support when debugging broken builds in the context of CI. Based on that, one of CAESAR's main focus is to help developers in understanding and fixing test failures. In addition to the previous studies, again, a quite high ration of build failures was identified.

Build Failure Categorization. Vassallo *et al.* [19] made a first attempt to compare CI processes and build failure occurrences from 349 Java open-source software projects with 418 projects from a financial organization (*i.e.*, ING Nederland). As a result of analyzing 34,182 build failures, they derived a taxonomy with 20 categories of build failures and their frequency of occurrence. On average, 26% of the builds failed and most of the build failures were attributed to failing tests. For open-source projects, they found that compilation errors and dependency issues were the second most reason why builds failed.

The resulting taxonomy from this study is used by CAESAR to classify build failures. Based on that and the previously presented study findings, CAESAR mainly focuses to support three of the most common build failures; dependency issues, compilation errors, and test failures.

Build Failure Fixing Effort. Krezazi *et al.* [9] analyzed 3,214 builds of a large software company in order to study the impact of build failures on the development process. They find that 17.9% of the builds fail and require approximately 900 to 2000 man-hours to fix them. On average, it takes 3 hours to become aware of a failed build and 57 minutes to fix it.

These numbers, even more, motivate to provide fast and easy access to build results in order to reduce the amount of time required to fix a build failure.

2.2 Existing Tools for Build Failure Resolution

Build Failure Summarization and Build Repair Hints. Vassallo *et al.* [18] propose BART *(Build Abstraction and Recovery Tool)* to support developers in fixing broken builds. BART is a Jenkins[1] plugin that summarizes build logs of failed builds and suggests possible solutions based on information found on the internet[2]. In a case study with eight participants, they identified that using BART helps to understand build failures. Additionally, the time needed to fix a broken build could on average be decreased by 41%.

The work of this thesis is similar to the work done by Vassallo *et al.* but differs in the approach of where in the CI workflow developers are assisted. Both CAESAR and BART aim to support developers in locating and fixing broken builds faster by first summarizing the build log of a

[1] Jenkins: https://jenkins.io
[2] StackOverflow: https://stackoverflow.com

failed build and then listing and describing possible causes and proposing solutions to fix them. However, while BART is a plugin for the build server, CAESAR is a plugin for the IDE. With CAESAR, all build-related information is directly available in the IDE and thus right next to the code that needs to be fixed. Furthermore, BART uses internet-based information to generate hints, whereas CAESAR only relies on log-based and locally available information (*i.e.*, from the local Maven repository). In addition to pointing out where the error happened (*i.e.*, file name and line of code), CAESAR enables to directly jump to this line of code and for some errors also provides instant debugging with pausing just before the error happened.

Automatically Repairing Dependency-Related Build Failures. Macho *et al.* [11] propose BUILD MEDIC to automatically repair failed Maven builds related to dependency issues. They investigated how developers repair dependency-related build failures and to what extent those failures can be repaired automatically. Based on 37 failed Maven builds, originating from 23 open-source Java projects, BUILDMEDIC supports three repair strategies (*i.e.*, version update, dependency deletion, and repository addition). They found that developers fix 46% of the build failures related to dependency issues by just changing the version. Moreover, 81% of the fixes only contain a single change. In the evaluation of the tool, BUILDMEDIC could automatically repair 45 out of additional 84 failed builds from the 23 projects (54%). In 36% of these successfully repaired Maven builds, BUILDMEDIC was able to propose at least one correct repair solution.

While, to a certain extent, BUILDMEDIC is able to repair dependency-related build failures automatically, CAESAR only locates the error and, if possible, proposes solutions to fix it or where to look for further information. Moreover, CAESAR supports the detection and fixing of dependency errors related to Maven group IDs, artifact IDs, and version numbers, while BUILDMEDIC handles version- and repository-related errors.

TeamCity Integration for IntelliJ Plattform-Based IDEs. Amongst others, JetBrains[1], a software development company that creates tools for software developers, offers the popular IDE *IntelliJ IDEA*[2] (primarily used for Java development) and the build server *TeamCity*[3]. For all IntelliJ platform-based IDEs they also provide a *TeamCity Integration*[4] to interact with TeamCity from within the IDE. Developers may, for example, trigger new builds, debug code running on TeamCity, or view test results and build logs.

While the TeamCity integration provides access to information from the build server and facilitates in-IDE interaction with it, CAESAR specifically summarizes build logs of broken builds, shows important information, and provides functionality to directly locate, fix, and debug detected errors.

[1] JetBrains: https://www.jetbrains.com
[2] IntelliJ IDEA: https://www.jetbrains.com/idea
[3] TeamCity: https://www.jetbrains.com/teamcity
[4] TeamCity Integration Plugin: https://plugins.jetbrains.com/plugin/1820-teamcity-integration

Chapter 3

CAESAR

This chapter first describes a typical interaction of a developer with continuous integration and identifies points of improvements (Section 3.1). Then, it is shown how a meta-model abstracts build logs to make build results easily accessible for both CAESAR and developers (Section 3.2). The last section introduces the enhanced continuous integration workflow with CAESAR and presents how developers can use the tool to fix build failures (Section 3.3).

3.1 Barriers in the Continuous Integration Workflow

The development process adhering to the conventional continuous integration workflow is described in Figure 3.1. Typically, a developer makes changes to the source code, commits them to the version control system, and *pushes them to a shared remote repository (1)* where other developers push their changes to too. In order to verify that the introduced changes integrate properly, the build server then *automatically pulls the latest version of the source code (2)* and *initiates a new software build (3)*. If the *build is successful (4)*, the developer may continue working on the source code and the just built software can be released. However, if the *build fails (5)* the *developer will be notified (6)* and needs to suspend his work to investigate the root of the failure. This is normally achieved by *scanning through the provided build log (7)*. Detecting the point of failure within the build log and locating the error within the source code often takes a lot of time. In addition, this poses a challenging task, especially with long build logs and complex errors. Besides that, the developer probably also needs to save the current work and roll back to the version of the source code that failed on the build server. Depending on the type and complexity of the error, the developer is forced to switch back and forth between his IDE and the build server.

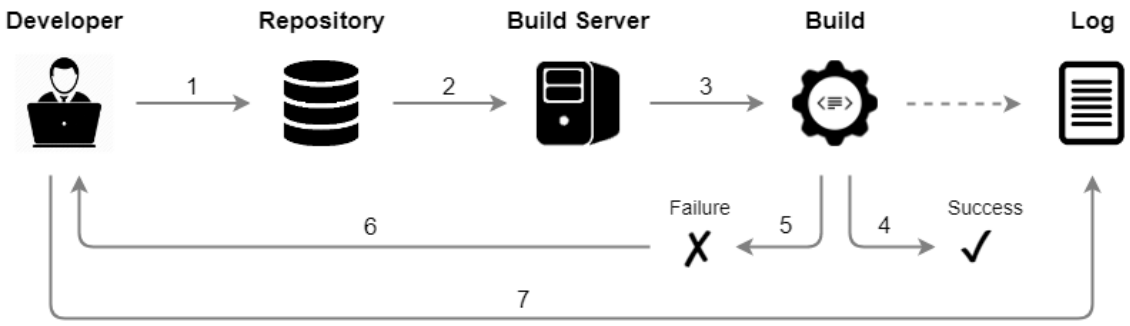

Figure 3.1: Continuous Integration Workflow

In an ideal, seamless workflow, the developer is notified immediately after a build fails on the build server. A notification could be displayed directly within the IDE, where the developer is working on the source code. Before starting to resolve the build failure, saving the ongoing work and loading the broken code base into the IDE can be automated. Furthermore, by providing the build results and build log within the IDE, no context switching between the build server and the IDE is necessary. Hence, locating the error within the source code could be faster and easier. Additionally, instead of manually searching through the build log to detect the cause of the build failure, it could be helpful to automatically extract the relevant information, list error messages, and show where the error occurred within the source code. Thus, the developer can focus on fixing the build failure rather than spending time locating it. The following paragraphs describe example use cases of solving common build failures and also indicate how the developer could be assisted.

Dependency Issues. On the build server, a new build starts within a clean environment. Therefore, all specified dependencies need to be fetched first. If a build fails due to a dependency issue, the build server was not able to download a dependency when starting to build the source code. This may have several different reasons. For example, the specified dependency is available on the local machine of the developer but not to the build server. Thus, even though a local build was successful, the build server is not able to access the dependency. Further, the specified dependency might not even exist at all. This could be due to a spelling mistake or that the configuration refers to the wrong version of the dependency. In either case, developers first need to know which dependency caused the build failure and then figure out why. After detecting which dependency caused the issue (by reading through the build log provided on the build server), developers most likely check whether the dependency is available on their machines and if the source code can be built there successfully.

Hence, besides showing developers which dependency caused the build break, they could profit from knowing which dependencies are available locally. This would assist them to quickly evaluate whether the issue is caused by a dependency that is not available on the build server or does not exist at all and thus point them in the right direction to search for the cause. Moreover, it could be helpful to provide links to the repository where the dependency is stored, so that developers could directly check whether the specified dependency should be available to the build server or not.

Compilation Errors. During the compilation of source code to binary code, the source code is parsed and analyzed. Common failures are, for example, misspelled identifiers (*e.g.*, variable, method, or class names), type mismatches, or a wrong number of specified arguments in constructor calls [3, 16]. Even though such compilation errors can be detected during a local build (*i.e.*, before committing the changes to the remote repository), they still occur on build servers (*e.g.*, because no local builds are run). After a compilation error is identified and located in the source code, it is often enough to have a short glance at the corresponding code section to figure out what let the compilation process fail.

Thus, fixing them cannot really be supported without additional information from outside the build log. Besides listing all compilation errors and showing where they happened, possible hints could be given on the most well known issues with compilation errors.

Test Failures. Like compilation errors, a lot of test failures can be detected locally. However, when changes from different developers are integrated remotely, the build only fails on the build server. The necessary information from the build log most likely contains the name of the test that failed and a stack trace of the called methods that led to the failure. Often the section with the test results contains a lot of entries and may be unstructured or distributed over different parts of

the build log. This makes it hard to locate the errors. To actually understand why the test failed, developers lack sufficient context information, for example, the values of the variables at run time. Most likely, the final step of fixing a broken test is to reproduce the error locally by debugging its execution. Thus, developers can understand what happens before the test fails. Some test failures are nevertheless hard to fix even if the test was run locally. This may be caused by differences between the local and the remote environment, where, for example, different software is installed or the access to remote networks and resources is limited.

Instead of searching through the build log to detect each test failure, switching to the IDE to locate a failed test, looking for the line that failed to set an execution breakpoint, and then start to debug the test, the whole process could be automated. The relevant information could be extracted from the build log and presented to the developer, so that he could directly debug each test failure with one click to reproduce the error locally. If however, the local and remote environment differ and it is not possible to find the cause of the error with a local test execution, the developer should be able to debug the test while running on the build server directly from within the IDE. Thus, he could work in the exact same context where the test failed before.

3.2 Build Log Abstraction Through a Meta-Model

A crucial part to access the necessary information about a broken build within the IDE is parsing the build log which is provided by the build server as plain text. Therefore a meta-model was developed. It enables CAESAR to extract the information required to show why a build failed. Additionally, developers may explore the original build log output in an abstracted form which facilitates faster access to the relevant information. This section shows how a build log is structured and describes the meta-model.

When source code is built on the build server, every action is written to a log file (*i.e.*, build log). On one hand, it contains all information regarding the steps configured for a certain build, like, for example, setting up the environment before the build starts or publishing artifacts after the build has finished. On the other hand, it contains the log of the used build tool, where developers search for the cause of a build failure.

A build tool is used to standardize the build process and helps developers to manage and execute software builds. Maven[1], for example, is a popular build tool for Java applications. Both CAESAR and the meta-model are designed to support Maven builds but the underlying concept can also be applied to other build tools. Maven builds software according to consecutively executed *build phases* and *plugin goals*. Build phases represent, for instance, the processes to (i) validate (check if all necessary information is available), (ii) compile, (iii) test, and (iv) package the source code. A single build phase consists of plugin goals that are responsible for executing specific tasks (*e.g.*, compiling). If a Maven build fails, it is because an error occurred during the execution of a specific goal. As soon as this happens, Maven stops the execution of the build. Hence, in case of a build failure, the log of the last executed Maven goal contains the relevant information to why the whole build failed.

As most build logs are very large and thus finding all necessary information to understand why the build failed can become a challenging task, there is a need for making this data better accessible. To achieve this, the build log is parsed and mapped to a meta-model (Figure 3.2). It follows the structure of the original build log and the execution of Maven. First, all the data concerning the configuration of the build server is mapped to the model. The *build server* may have multiple configured *projects* where each project may have multiple *build configurations*. These build configurations are responsible for different kinds of builds. Some configurations might, for example, just compile or test the source code and others might also package it. For each build

[1] Maven: https://maven.apache.org

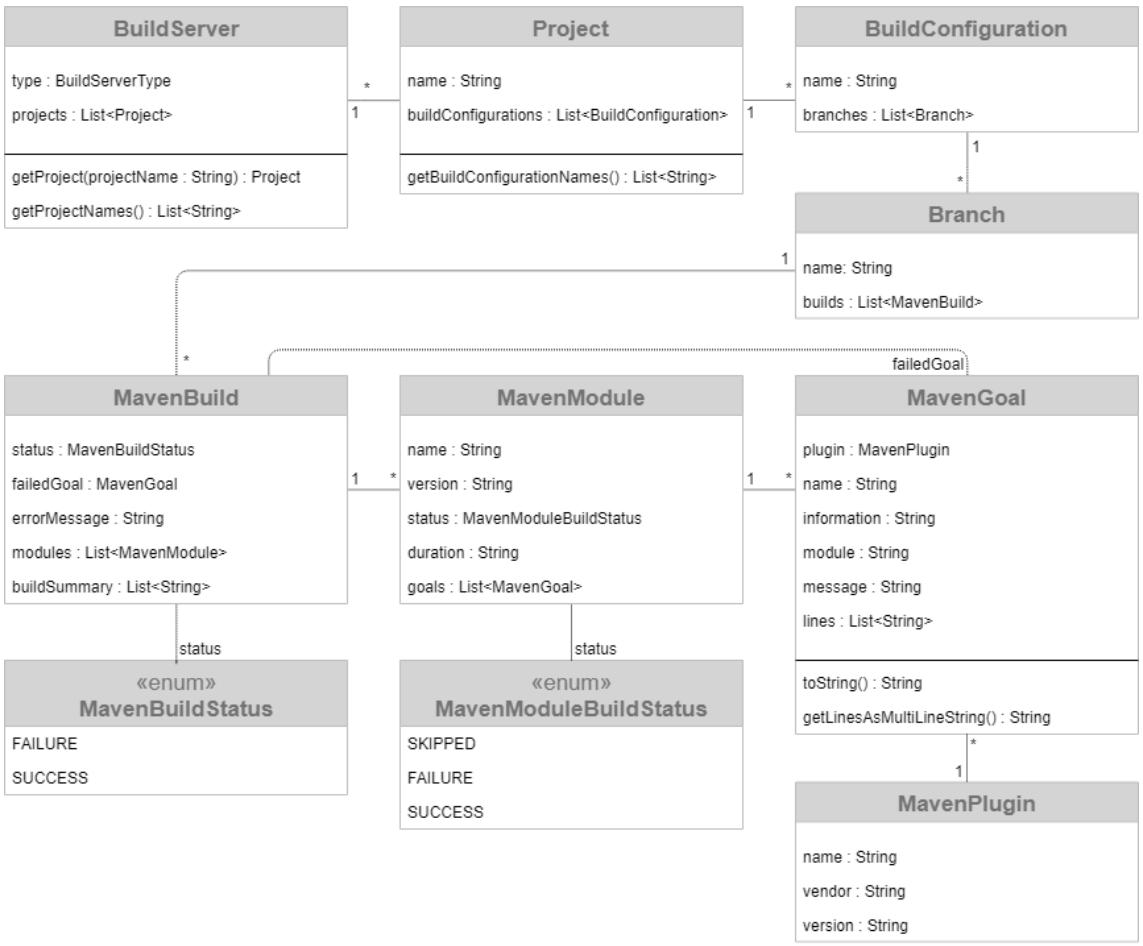

Figure 3.2: Meta-Model

configuration there are multiple *branches* of the source code that may be built, since the source code is managed with a version control system. Every branch may be built multiple times and each build is executed as a Maven build. It consist of multiple *Maven modules* representing sub-projects. Each of these modules executes *Maven goals*; depending on the build configuration.

3.3 Augmented Continuous Integration Workflow with CAESAR

Based on the presented meta-model, CAESAR summarizes the broken build and assists the developer within the IDE. There is no need to switch to the build server to search for the reason of the failure. The enhanced continuous integration workflow is presented below. Afterwards it is shown how CAESAR assists developers in resolving build failures.

Like in a typical continuous integration workflow the developer first *pushes the changes to the remote repository (1)* where *the build server automatically fetches the latest version (2), initiates a new build, and writes every action into the build log (3).* In the augmented workflow with CAESAR, the

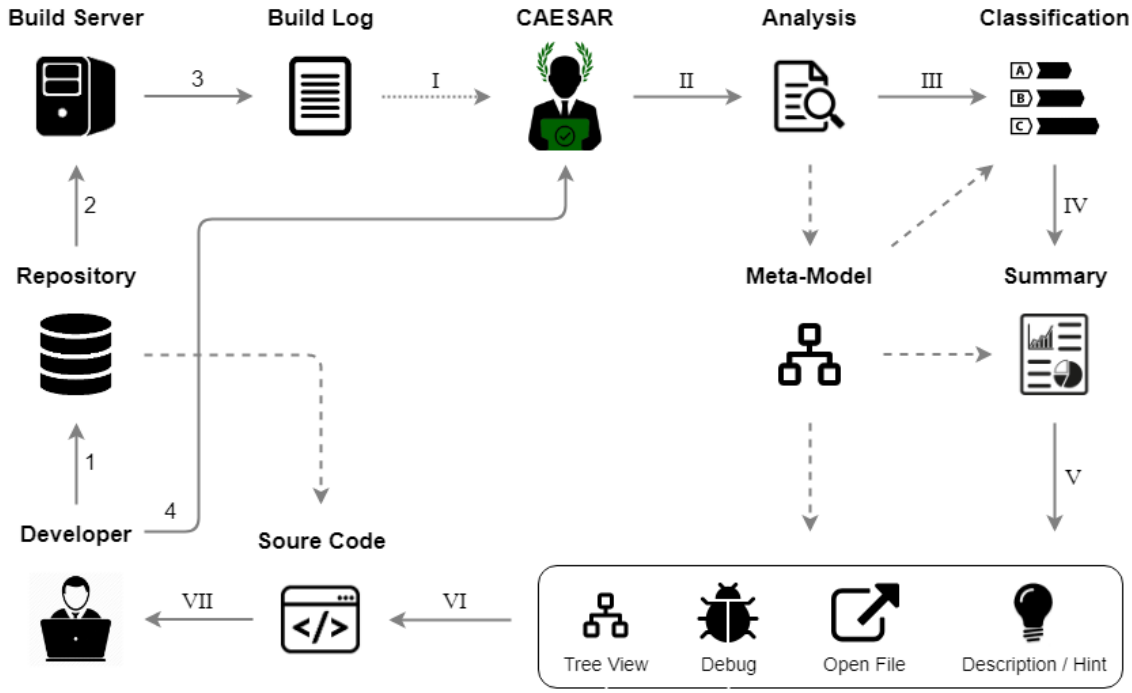

Figure 3.3: Augmented Continuous Integration Workflow with CAESAR

developer then opens the CAESAR IDE plugin and *checks whether the build was successful or not (4)*. If the build failed and the developer decides to fix it, he selects the build and CAESAR *downloads the corresponding build log (I)* and starts processing it. The build log is read line for line, *its content analyzed (II)*, and all information stored to the *meta-model*. CAESAR then operates on the meta-model *to classify the type of error (III)* and *to create a summary (IV)* of the most relevant data to be *presented to the developer (V)*. This includes a list of all detected errors with *error descriptions or hints* for each error. Additionally, the errors can be directly shown *in the source files* and some errors can also be *debugged* using the debugging features of the IDE. Based on the meta-model, a *tree-like view* of the build is presented to explore the log of the build in a more convenient manner. Optionally, the developer can initiate to *load the source code (VI)* that failed on the build server into the IDE. Finally, the developer is able to use CAESAR *to explore the errors and fix them (VII)*. The following paragraphs describe how CAESAR can be used to do so.

CAESAR aims to simplify the process of fixing build failures by making all the necessary information from the build log available within the IDE. First, CAESAR lets the developer choose a project to get a list of all executed builds on the build server (Figure 3.4). The builds are grouped by their corresponding branch in the version control system. For every build, the developer instantly sees whether the build was successful or not. Additionally, he gets a short status message of the build outcome. In case of a build failure, this might already be a first indication of the cause.

When selecting one of the builds, a summary of the build is displayed (Figure 3.5). At the top, the build status indicating a success or failure and the status message are displayed again. Then, the developer gets information about what kind of build failure was detected. Depending on the information found within the build log and at which point the build failed, CAESAR classifies each build failure in one out of 13 categories [19]. This aims to help the developer to pre-estimate what went wrong. Additionally, the failed build step (build goal) is displayed. The developer also sees the name of the configured project on the build server and the build configuration that

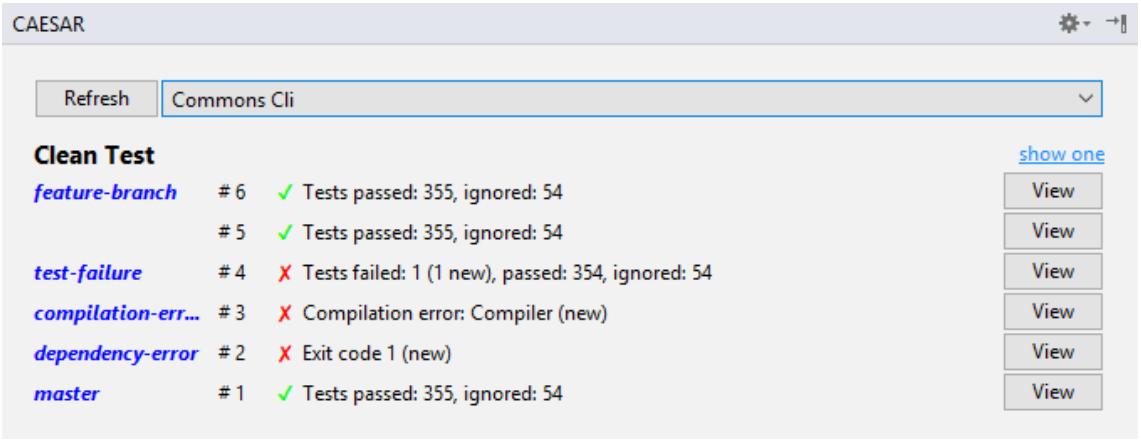

Figure 3.4: Overview of All Builds

was used to run the presented build. To give further contextual information to the build failure, the branch of the version control system is also displayed.

The core section of this view lists all errors found within the build log. Besides the file name and line where the error happened, CAESAR displays a short error description. Depending on the failure category, an additional hint is displayed too. Next to each error, there is a button to open the source file at the line where the error occurred. At the very bottom, the developer may go back to the build overview (Figure 3.4) or load the code base that failed on the build server into his IDE. If there are any changes that are not committed to the version control system yet, CAESAR automatically saves them, so that the developer can access them after he has finished fixing the errors. The following four paragraphs show how CAESAR treats the three supported build failures and how unsupported build failures are handled.

Dependency Issue. An example error description of a dependency issue is displayed in Listing 3.1. The developer gets information about which dependency could not be resolved on the build server and whether it is locally available. CAESAR distinguishes between three different cases why the dependency could not be found locally: because of the (i) group name (*e.g.*, the name of the company that developed the dependency), (ii) the artifact name (*i.e.*, dependency name), or (iii) the specified version of the artifact. If the artifact could not be found, the error message additionally lists alternative, local artifacts or versions. This might signalize that the wrong artifact or version was specified. In the end, a link to the remote repository is given, so that the developer can check whether the specified artifact can be found there. If the dependency could be found on the developer's machine, CAESAR tells the developer to check whether the dependency really is available in one of the defined repositories. Hence, the developer knows that the dependency is correctly specified but the build server is not able to download it.

```
Could not find artifact: junit:junit:5.12 (version '5.12' is locally not avail-
able) Please check why this version can not be found on maven central or in any
of the defined repositories in the pom.xml file. Does the <version> really
exist? Locally, only the following versions are available: 3.8.1, 3.8.2, 4.11,
4.12. Check all available versions of this artifact on maven central:
https://mvnrepository.com/artifact/junit/junit
```

Listing 3.1: Detected Dependency Issue

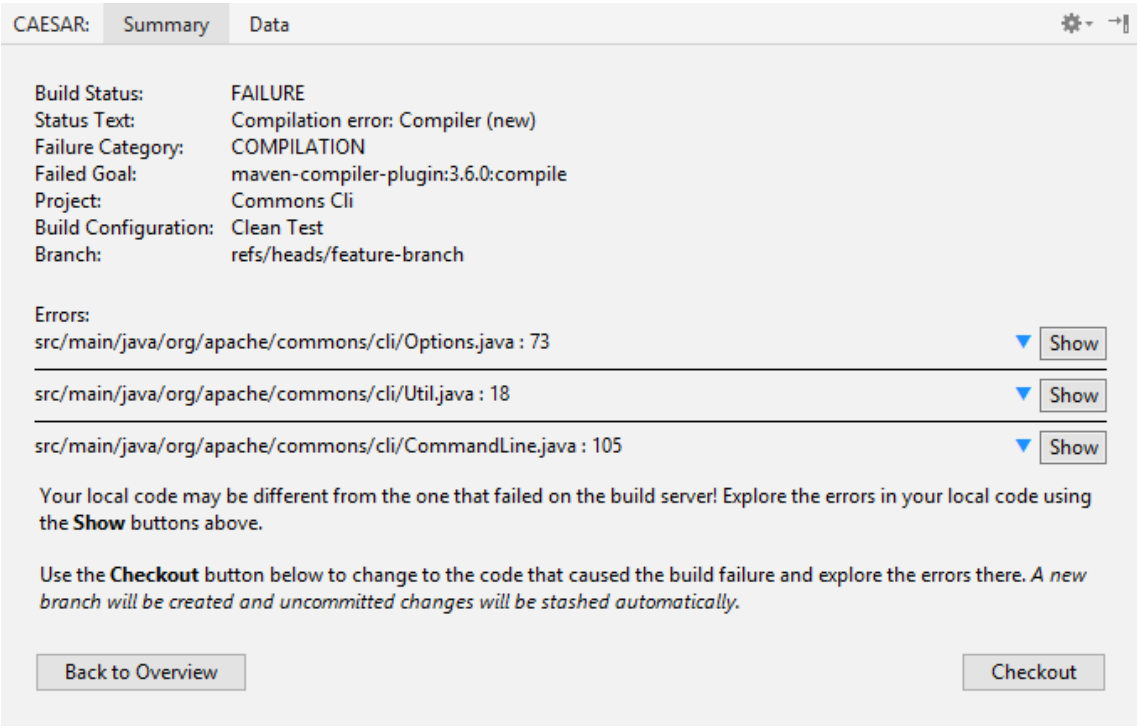

Figure 3.5: Summarization of a Build

Compilation Errors. The support for compilation errors is rather simple (Listing 3.2). Besides showing where the error occurred in the source code and letting the developer navigate there, CAESAR displays a short description of the error. Often, this is enough information to resolve the failure.

```
src/main/java/org/apache/commons/cli/Options.java: 20
package org.apache.commons.cli.entities does not exist
```

Listing 3.2: Detected Compilation Error

Test Failure. CAESAR lists all test failures found in the build log and displays a short error description. An example is given in Listing 3.3. In addition to the button to open the test file at the line where the test failed, there is also a button to directly debug the error. When the developer clicks on it, CAESAR starts executing the test that failed and pauses before the error happened. With the debug instruments provided by the IDE, the developer is now able to see the values of all variables and gets the relevant context information to fix the error.

```
src/main/java/org/apache/commons/cli/UtilTest.java: 31
org.junit.ComparisonFailure: expected:<[]foo> but was:<[-]foo>
```

Listing 3.3: Detected Test Failure

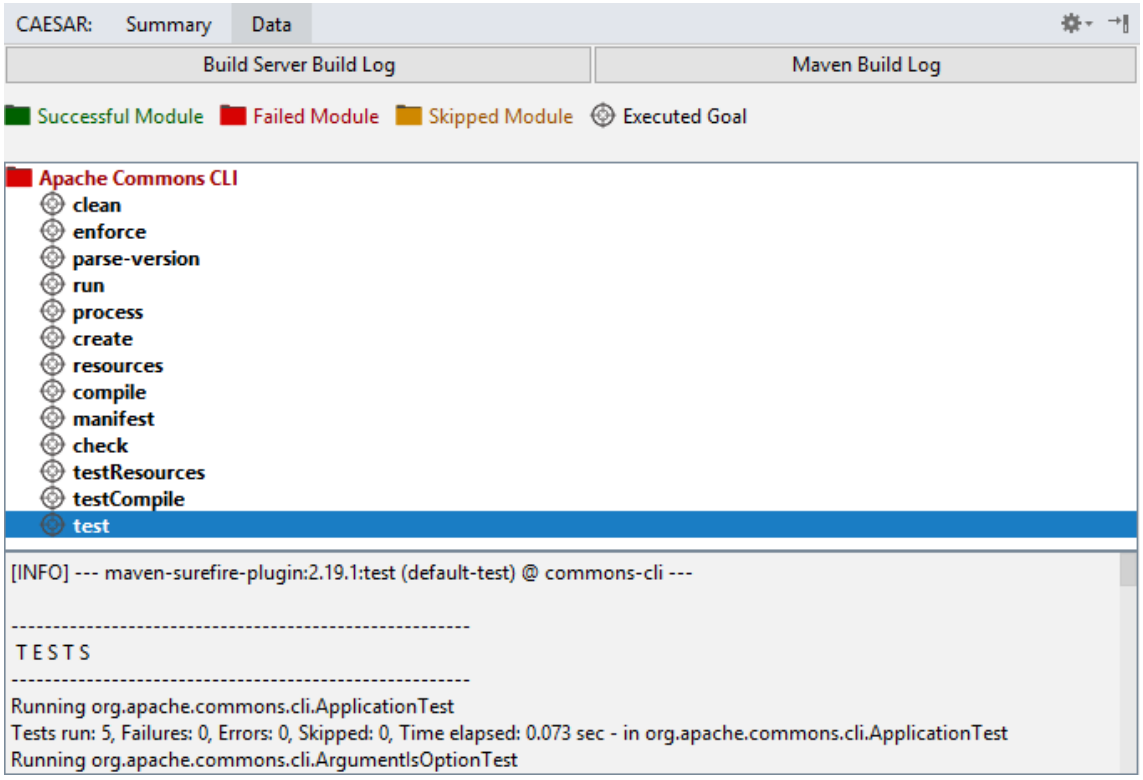

Figure 3.6: Abstraction of a Build Log

Fall Back for Other Error Types. Regardless of whether CAESAR provides tailored assistance for a certain error type, he also provides an abstracted tree view of the build. This especially becomes useful to developers if no errors could be found. In this case CAESAR does not support full assistance for the failed Maven goal (*i.e.*, build step). However, based on the meta-model, the build log is disassembled into its base components. On the first level, all Maven modules (*i.e.*, sub-projects) are presented. Different colors indicate whether the module (i) was built successfully, (ii) could not be built, or (iii) was skipped because the build failed in a module before. For all processed modules, (i) and (ii), CAESAR lists the executed Maven goals and the corresponding section of the build log. Therefore, if a build failed, the last listed goal of the failed Maven module contains the section of the build log that provides important information of why the build failed. In addition to that, the developer also may read through the original and complete build log of the build server or the extracted Maven build log.

Chapter 4

Controlled Experiment

To evaluate CAESAR's assistance to developers, a controlled experiment was conducted. This chapter first explains the course of the controlled experiment and shows some demographic data about the study participants (Section 4.1). Then, the results of the study and answers to the research questions are presented (Section 4.2).

4.1 Procedure

The study context includes (i) as *objects*, build failures that are introduced into existing source code, and (ii) as *subjects*, developers that participate in the controlled experiment. The study is designed as a *between-subject* study where each participant solves one task with treatment (*i.e.*, assistance of CAESAR) and one without. The controlled experiment is segmented into four parts: *Introduction*, *Training*, *Problem Solving*, and *Feedback*. The following paragraphs describe what each part consists of. All necessary files and information (*e.g.*, repositories or steps to prepare the experiments) are listed in Appendix A.

Introduction. The introduction aims to ensure that every participant has the same knowledge of the tools and underlying concepts used in the controlled experiment. To further guarantee that every participant gets the same introduction, a video was recorded. It contains information about the used (i) concepts (*i.e.*, version control systems, build tools, build servers, and continuous integration) and (ii) tools (*i.e.*, IntelliJ IDEA[1], TeamCity[2], Maven[3], and CAESAR).

Training. The goal of the training part of the controlled experiment is to let the participants get to know how to work with the development environment and especially with CAESAR. The provided project contains every type of build failure that CAESAR supports (*i.e.*, dependency error, compilation error, and test failure). Each participant is asked to use both the build server (*i.e.*, TeamCity) and CAESAR to fix a build failure. In addition, participants are asked to work with the terminal integrated in the IDE to interact with the version control system, use the debugger, and locally run Maven lifecycle phases.

Problem Solving. The main part of the controlled experiment is about solving two build failures. The used project is the Apache Commons CLI[4], which was previously used in studies

[1] IntelliJ IDEA: https://www.jetbrains.com/idea
[2] TeamCity: https://www.jetbrains.com/teamcity
[3] Maven: https://maven.apache.org
[4] Commons CLI: https://commons.apache.org/proper/commons-cli

#	Type	Introduced Error
Problem 1	dependency error	non-existing version number of a dependency
Problem 2	compilation error	class import from a non-existing sub-package
Problem 3	test failure	incorrect method implementation

Table 4.1: Problem Types and Introduced Errors

[4, 5, 12, 15] and with 22 classes and approximately 12'000 lines of code considered large enough but not too small. For this project, three different build failures (*i.e.*, problems) were provoked by introducing three errors (Table 4.1).

Each participant gets to solve two randomly assigned problems. Half of the participants first solves a build failure with CAESAR and then without and the other half first without and then with CAESAR. This ensures, that over all runs of the experiment, participants do not profit from learning effects (*i.e.*, knowing the project from the previous task). In addition, every problem is solved both twice with and without CAESAR and both twice as first and as second task. Thus, the results do not depend on a particular order of the problems.

Before each participant starts solving the problems, for every task a new build is run. To determine the time spent to solve a build failure, the participants start and finish each task with an empty commit. After fixing the build failure, each participant is then asked to locally run the Maven lifecycle phase *test* and, if the Maven build was successful, commit the changes. The empty commit to indicate to be finished with solving the build failure should be made after a new build is successfully run on TeamCity.

Feedback. To get feedback and comments to the solved tasks and CAESAR, every participant completed a questionnaire split into three parts: (i) questions about the task without using CAESAR, (ii) questions about the task with CAESAR's assistance, and (iii) general questions. Besides the actual times needed to solve the build failure the responses to these questions are used to answer the research questions.

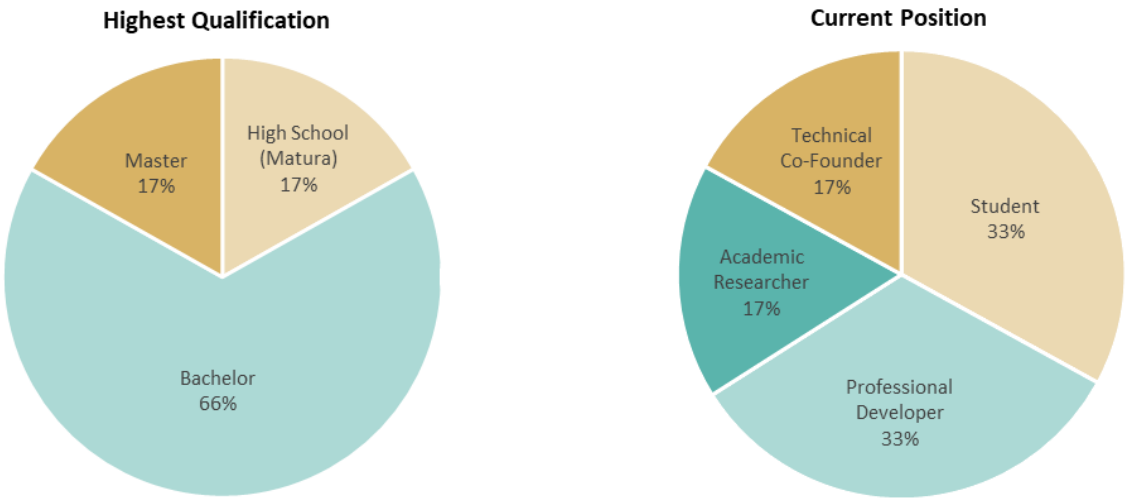

Figure 4.1: Highest Qualification and Current Position of Participants

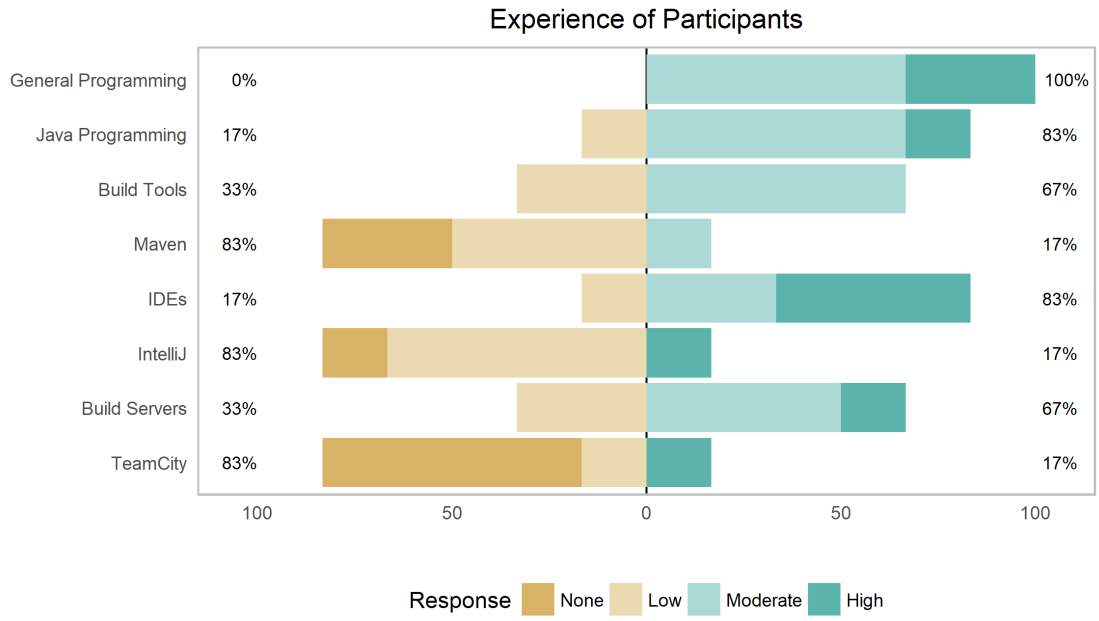

Figure 4.2: Experience of Participants with Study-Relevant Tools

All six participants are computer science students at the University of Zurich. To collect information about their experience with the tools and concepts used in the controlled experiment, every participant filled out a questionnaire before the controlled experiment. Figure 4.1 shows the *highest qualification of the participants related to computer science*. Two thirds completed their bachelor studies, one his master studies, and one participant is still in his bachelor studies. The figure also shows the *current position of the participants where they get in contact with build servers*. All participants holding a bachelors degree get in contact with build servers either at work or as students, the participant holding a masters degree as an academic researcher, and the one pursuing his bachelor studies as a technical co-founder.

To assess the participants level of experience with the tools and concepts used in the study, a four-point Likert scale (*i.e.*, *None*, *Low*, *Moderate*, and *High*) with the additional option for *No Answer* was used (Figure 4.2). While most of the participants self estimated their *programming experience* as moderate or high, they stated that their experience with the used tools (*i.e.*, *IntelliJ* and *TeamCity*) and concepts (*i.e.*, *Maven*) is low or that they do not have experience using or applying them at all. For the general experience with *build tools*, *IDEs*, or *build servers* however, the average participant shows moderate to high experience.

4.2 Results

Subsequently, the results of the controlled experiment are presented and all research questions are answered. Before that, it is ensured that the chosen error types are relevant in practice and that the introduced errors are representative.

Even though the decision to specifically support three of the most common error types (*i.e.*, dependency error, compilation error, and test failure) was based on previous research, the introduced errors (Table 4.1) in the study are artificially set up. Thus, it first needs to be made sure that

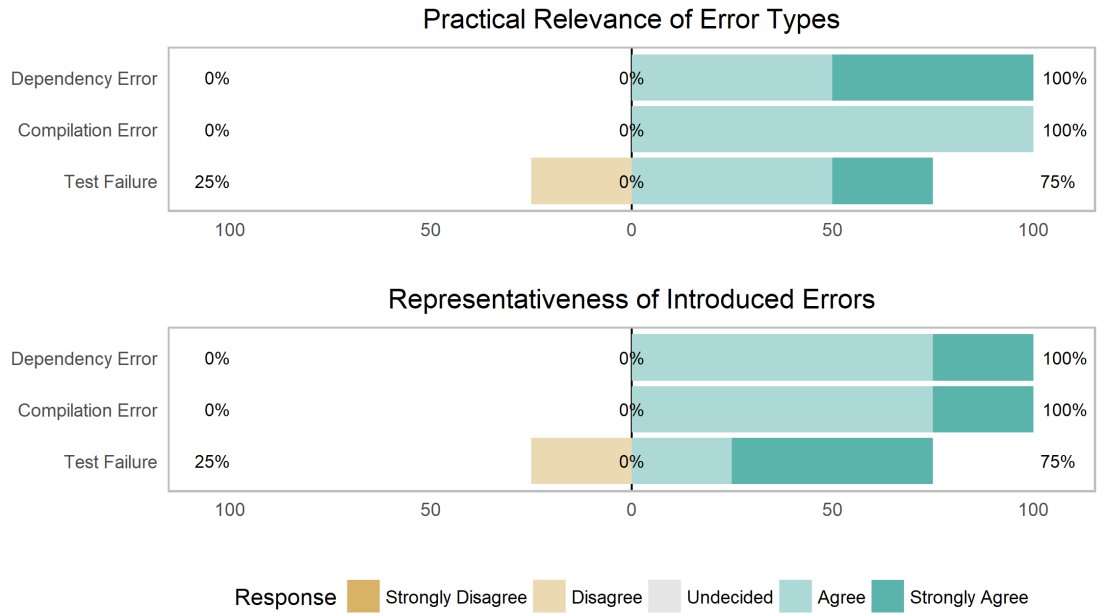

Figure 4.3: Practical Relevance of Error Types and Representativeness of Introduced Errors

they are also realistic. Therefore, each participant was asked to rate (i) the *practical relevance of the three error types* in general, and (ii) the *representativeness of the introduced errors* after he finished the experiment (Figure 4.3). To indicate their agreement to whether the error types are relevant and the introduced errors are representative, the participants rated the statements, depending on the errors they solved, with a five-point Likert scale (*i.e.*, *Strongly Disagree*, *Disagree*, *Undecided*, *Agree* and *Strongly Agree*) with the additional option for *No Answer*.

All three error types as well as the introduced errors are considered as relevant and representative with a total rate of acceptance of 92% (*Strongly Agree* and *Agree*). Only one participant did not rank the error types and introduced errors as relevant and representative.

4.2.1 RQ1: Build Failure Resolution Time

The first research question deals with *whether the time needed to fix a build failure can be reduced with in-IDE assistance*. This specifically refers to CAESAR's assistance to developers and if their productivity can be increased. The point of interest is, whether providing build-related information within the IDE helps developers in resolving broken builds faster and thus let them focus on developing the source code further rather than spend time fixing it.

To answer this questions, for every introduced error the required time to fix the build failure was measured. This provides empirical evidence whether CAESAR can reduce the time to resolve a broken build or not. Therefore, the commits, before and after the errors were solved, are taken to determine how long each participant needed to fix the assigned build failure. Figure 4.4 shows all these fix-times grouped by the three error categories; (i) dependency error, (ii) compilation error, and (iii) test failure. Within each category, the times for fixing the failure without CAESAR (to the left) and with CAESAR (to the right) are shown. Additionally, for every error category, the average time reduction between the tasks without CAESAR (upper line) and with CAESAR (lower line) is indicated.

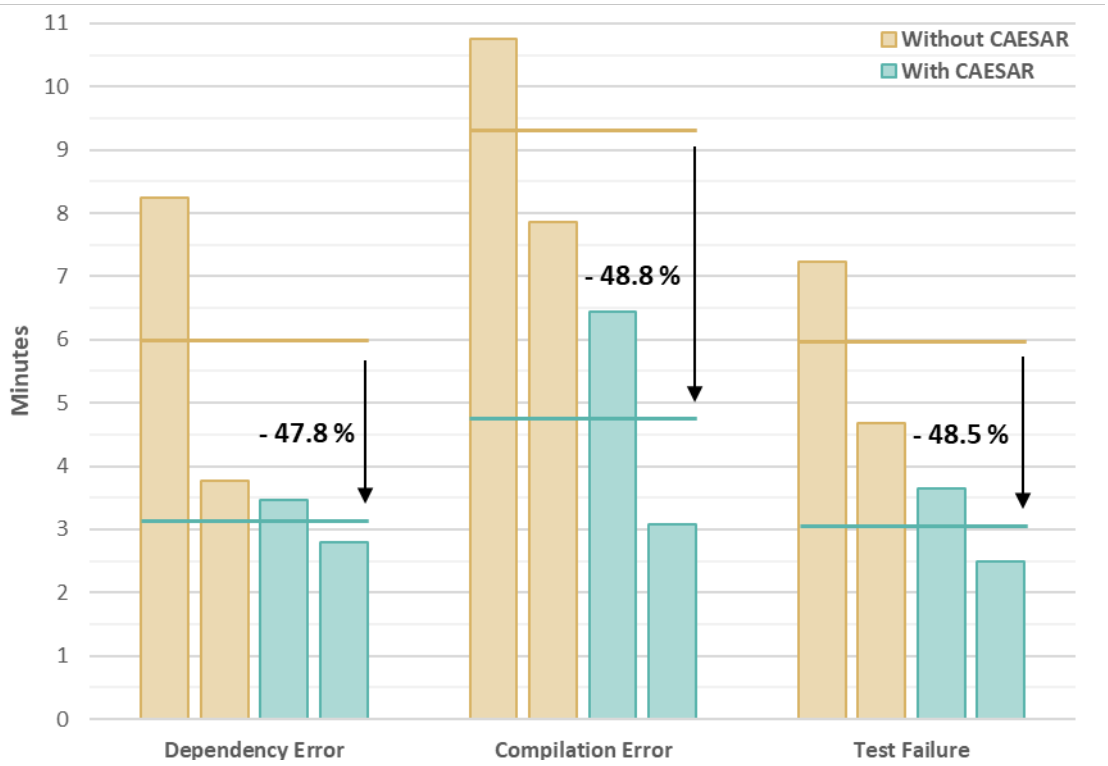

Figure 4.4: Recorded Build-Fix-Times and Average Time Reduction per Error Type

The analysis shows that, on average and over all types of build failures, the participants could reduce the time needed to fix the broken build from approximately 7 to 3.5 minutes (48.4%) when working with CAESAR. Dependency errors are solved 47.8% faster, compilation errors 48.8%, and test failures 48.5%. It can be observed, that one participant who was asked to fix a dependency error without CAESAR spent twice as long to resolve it than the second participant with the same task. On the other hand, two different participants that were asked to fix the same broken build with CAESAR roughly required the same amount of time to resolve it. This reflects the total estimated standard deviation of 1.2 minutes which is the highest for dependency errors (with 2.3 minutes). For compilation errors and test failures the savings in time are better distributed. Even though also for compilation errors, the required times to solve it with CAESAR deviate by a factor of two.

RQ1: Can in-IDE assistance reduce the time required to fix a build failure?
Yes, by providing build results within the IDE, CAESAR *reduces the time needed to fix a build failure; on average, by 48.4%.*

4.2.2 RQ2: In-IDE Assistance

The second research question is interested in *why broken builds can be resolved faster with in-IDE assistance*. While there might be several possible answers to this questions, for example, not needing to switch back and forth between the IDE and the build server, having a summarized build log instead of scanning through it, getting error descriptions and hints, or being instantly able to

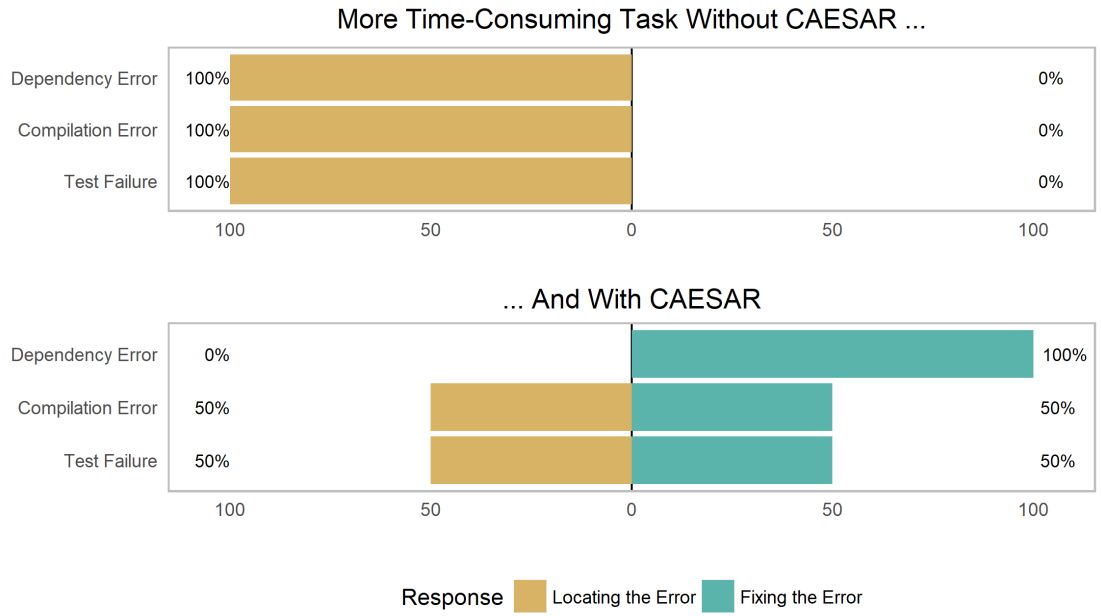

Figure 4.5: More Time-Consuming Task without and with CAESAR

debug failed tests, this question focuses on two steps of the build resolution process: (i) locating the error and (ii) fixing it. This is based on the hypothesis that developers become faster in locating and understanding the error thanks to the provided summarization of the build log directly within the IDE.

To answer the research question, the participants (i) were asked whether *locating or fixing the error* was more time-consuming and (ii) how they rate the *perceived assistance of* CAESAR during the resolution process of the build failure. In the context of this research question, the term *fixing* relates to the fixing phase and the term *resolving* labels the whole process; locating and fixing the build failure.

Locating vs. Fixing the Error. Assuming, that CAESAR helps in faster locating the error, the participants were asked whether they spent more time for *locating the error or fixing it*. The question was answered for both the *task without and with* CAESAR. Figure 4.5 shows the answers to these questions. Participants simply had to state whether *locating* or *fixing* the error was the more time-consuming task. First the responses for the problems without CAESAR are shown and then with.

When solving a build failure without the assistance of CAESAR, all participants spent more time for locating the error than fixing it. However, when working with CAESAR, on average, 67% of the participants spent more time for fixing than locating the error. While all participants that needed to solve a dependency error spent more time for fixing the error, only 50% of the participants that solved a compilation error or test failure considered fixing the error the less time consuming task.

Perceived Assistance of CAESAR. In order to assess the perceived assistance of CAESAR by the participants, three questions were asked (Figure 4.6). First, the participants stated whether CAESAR *generally helped* them while resolving the build failure. Then, they specified if the *summa-*

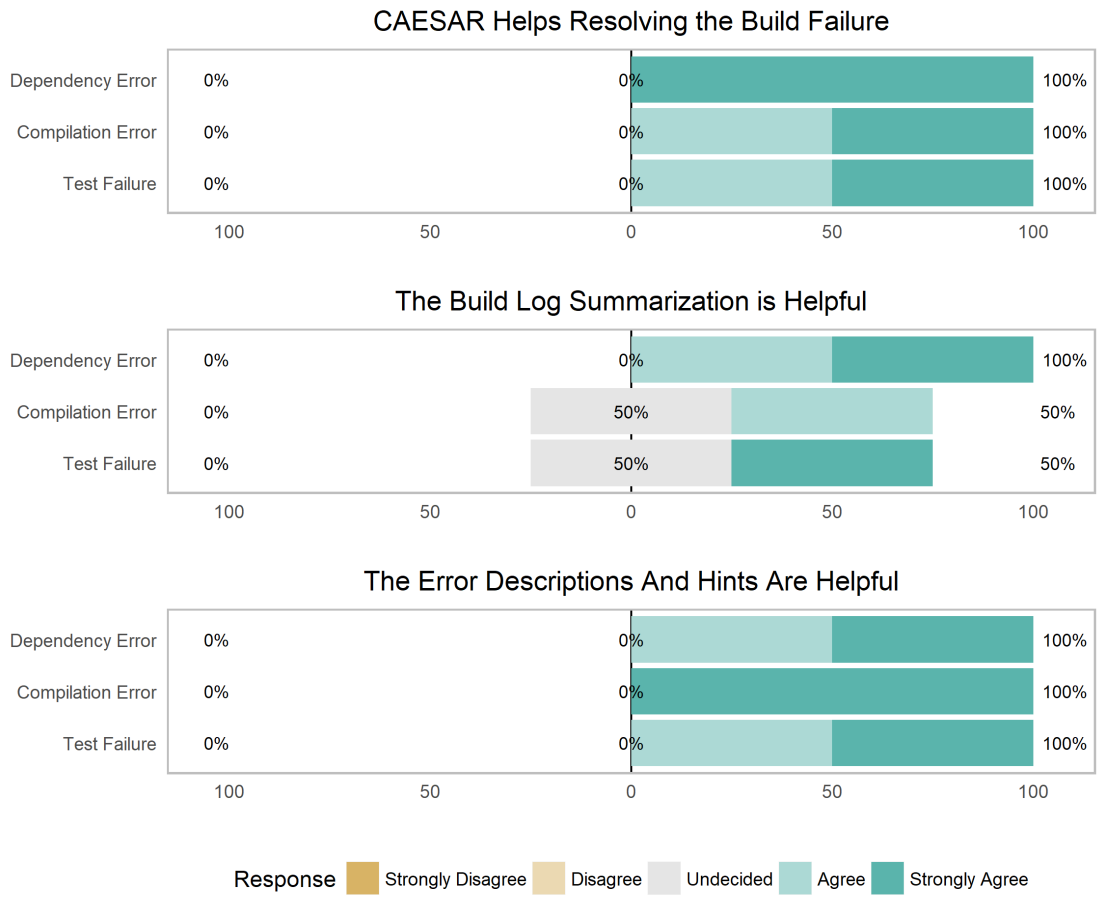

Figure 4.6: Assistance Through CAESAR in General, Through Build Log Summarization, and with Error Descriptions and Hints

rization of the build log was helpful during the location of the error. Finally, they were asked whether the *error descriptions and hints helped* them understanding and fixing the failure. For all three questions, the same five-point Likert scale was used again. The answers to the questions are shown per error type.

Regarding the general assistance of CAESAR, the participants unanimously agreed that CAESAR helped them resolving the build failure. The provided build log summarization was considered as helpful. The ratings however show, that the summarization for dependency errors was appreciated more than for compilation errors and test failures. For the error descriptions and hints, the participants again consistently agreed that they were helpful.

RQ2: Why can broken builds be resolved faster with in-IDE assistance?

When resolving a build failure with CAESAR*, developers generally spent less time locating the error than when working without* CAESAR*. On average, then gained time from locating the error could be used to fix it. Further, the build log summarization was considered helpful when locating the error. In addition to that, the error description and hints were then helpful to understand and actually fix the concrete error.*

4.2.3 RQ3: Instant Test Debugging

The last research question was, *whether instantly debugging a test failure helps solving a build failure*. The debugging feature is one part of CAESAR and is specifically designed to support developers in locally reproducing the results of failed tests on the build server. The goal is to give some context to the test results and make it easier to understand why tests fail. The question aims to analyze whether, on top of the build log summarization and error descriptions and hints, this feature is helpful in fixing build failures.

To answer this question, the participants who solved a test failure were asked to state whether this feature helped them or not. Unfortunately, both participants that solved a broken test with the assistance of CAESAR and could have used the instant debugging feature were able to solve the test failure without debugging the test at all.

RQ3: Does instantly debugging a test failure help solving a build failure?

Could not be answered during the controlled experiment. In the training session however, this feature was tested and considered as supporting. More experiments are needed to answer this questions.

Chapter 5

Discussion

In this chapter, the results of the controlled experiment and the answers to the research questions are discussed. Then, implications for continuous integration and research are described and some thoughts about the future work on CAESAR are shared (Section 5.1). Finally, threats to the study's validity and their mitigation are shown (Section 5.2).

RQ1: Build Failure Resolution Time. With CAESAR serving as in-IDE assistance to developers for resolving build failures, participants of the controlled experiment could reduce the required time to fix a build failure by 48.4%. Although similar results have been obtained in a previous study with a build server plugin, where the time could be reduced by 41% [18], the reported estimated standard deviation of the average build-fix-times is non-negligible. This can be explained by the low number of study participants and demands for a controlled experiment with a larger group of developers.

Although, the average time required to resolve a build failure was nearly identical for all examined error types, there are differences in the time between the individual recordings. For example, one dependency error fixed without CAESAR required twice as much time as other performances on the same error. The participant stated that he *"needed to search through the build log to find the error"* and that this *"was time consuming"*. This can be traced back to different approaches the participants had on how to scan a build log which might be based on their experience. Moreover, this shows that locating the error within the build log is a major issue when working without assistance; not only for this type of error but also for others; as reported by other participants when resolving different failures.

On the other hand, the measured times for compilation errors and test failures are better distributed. This could be explained by the fact that just showing the location of the issue causing the build failure might be the most important step in fixing the error. In addition to that, the support for dependency errors is the most advanced one (with providing hints based on locally available dependencies), which was also considered to be more helpful than the assistance for compilation errors and test failures.

RQ2: In-IDE Assistance. When working with CAESAR, participants could reduce the time to resolve a build failure due to two reasons: they were (i) able to save time when locating the error and (ii) assisted in understanding and fixing the error. When working without CAESAR "locating the error" was more time-consuming than "fixing the error". Thanks to CAESAR's build log summarization the time spent "locating the error" could be decrease which was appreciated by most participants. One participant stated that *"in only a few clicks [he] was able to instantly switch to the line where the error occurred"* and another reported that he liked that *"[CAESAR] showed the*

exact position in the source code". Hence, the summarization and direct linking to the source code contributed to reduce the time "locating the error".

In addition, CAESAR also helped to understand the error. The summarization provides convenient access to the build results and the most important information necessary to resolve the build failure faster. Additionally, it lets developers focus on "fixing the error" as they do not need to switch context between the IDE and the build server as frequent. For example, as stated by one of the participants: *"when switching between IDE and TeamCity to inspect the errors, developers might lose track of what they are doing, as they have to change the environment"*. Another participant mentioned that CAESAR helped *"to locate the root cause of the test failure as [he] already knew (from the summarized maven build log) what the potential reasons for the test failure might be"*. The obtained results support the finding of Vassallo *et al.* [18] stating that the build summarization improves the understandability of build logs.

RQ3: Instant Test Debugging. Unfortunately, the controlled experiment could not provide insight in whether CAESAR's feature to instantly debug failed tests contributes to solving build failures more easily. The participants only used the feature during the training session but not for solving the build failures. This might have different reasons.

First of all, the test setup with the introduced errors was not appropriate. The errors were artificially inserted into existing source code and designed for developers with rather low levels of programming experience. However, the demographics data shows that they are not low skilled. In addition, the participants also stated that *"[the] errors were not particularly difficult to fix"* and *"solving the error was quite trivial"*. Due to the simplicity of the errors they did not use the debugging feature and could fix the error *"by just having a quick glance at the code"*.

Secondly, the debugging feature might not have been used due to the fact that IDE-debuggers are not used as often as expected. This is the result of a study conducted by Beller *et al.* [2] where the debugging behavior of developers was investigated. Other reasons might be that the feature was not properly introduced or implemented. Although it was part of the introduction video and the participants worked with it in the training, they did not consider using it to fix the build failures. To further evaluate this feature, a second experiment needs to be conducted specifically targeting to resolve broken builds related to test failures.

5.1 Implications and Future Work

Implications for Continuous Integration and Research. The results of the controlled experiment show that in-IDE assistance provides large benefits to developers for resolving build failures. They can locate errors faster and understand the root of failure better. This eventually leads to increased developer productivity and faster release cycles. Due to this fact, developers should be provided with more advanced tools assisting them in fixing broken builds. Tool builders should provide easy access to all build related information, integrate them with the IDE, and aim to bridge the gap between the information available on the build server and in the IDE.

Future research could investigate the effect of the combination of tools that summarize build logs, automatically propose fixes, and collect solutions from the internet. Moreover, the impact of such tools on different levels of experience of the developers should be analyzed in order to better design the tools and provide specific assistance. Also, the difference between varying error complexities must be investigated to grasp the benefits of such a tool in a modern high-skill software development environment.

Significance. To learn more about CAESAR's benefit to developers in fixing build failures, CAESAR needs to be evaluated in a larger scope. First, developers should solve injected build failures in multiple different projects. This ensures that the results do not depend on a special type of project or the style of development. Secondly, a more diverse group of developers should be chosen to conduct the controlled experiment. Additionally to the participants of the study for this thesis, there should also be some developers from different industries with varying levels of experience in both programming and working with CI. Even though the artificially introduced errors were considered as relevant, there should be a more diverse pool of errors to better show the assistance through CAESAR when fixing the three error types. Finally, duplicating the problem sets and letting them be solved by multiple participants makes the results more comparable and insightful. For the same reason, every developer should also solve at least one build failure of each error category.

Application. The current implementation of CAESAR is limited to be used for Java projects, developed with IntelliJ, managed by Maven, and built with TeamCity. The supported error types are: dependency errors, compilation errors, and test failures. In order to support and summarize various types of build failures, parsers for additional Maven plugins need to be added. The implementation could also be extended to support and integrate other build tools than Maven like, for example, Gradle[1]. Besides this, the concept of CAESAR and the underlying meta-model can also be transferred to other build servers, version control systems, and programming languages.

Features. On top of the existing functionality of CAESAR, there is a large range of features that could be added to assist developers even better. First of all, the interaction with the build server could be improved by directly triggering new builds from within the IDE and automatically getting notified as soon as a new build fails. For further support in fixing build failures, CAESAR could be combined with other approaches by collecting hints based on internet searches and proposing automatically generated suggestions to fix certain failures. After a fix is made, it could be helpful to let CAESAR automatically run the same build stage that failed on the build server locally to verify that the fix works as expected. To further simplify the interaction with the underlying version control system, CAESAR could provide the possibility to choose where to merge the branch containing the fix and automatically reload the state of the work from before starting to fix the broken build.

5.2 Threats to Validity

Replicability. *Can these results be replicated?* This threat is addressed by providing all the material used for the controlled experiment. On the attached CD (Appendix A), the used version of CAESAR is available as well as all training and problem repositories. Moreover, instructions on how to set up all projects with the build server and the IDE are given and the introduction video is provided. The only unknown variable are the participants. Due to the fact that this study only had six participants, the results of a replication of the controlled experiment might slightly differ depending on the levels of experience and skills of the participants.

Construct Validity. *Is the controlled experiment designed correctly?* The interest of this thesis and controlled experiment is to improve the process of fixing build failures. Thus, the asked questions reflect these interests to some extent. However, by always giving the participants the option not to answer a question or rate a statement and providing the possibility to add comments and

[1] Gradle: https://gradle.org

annotations to the experiment or asked questions, biasing the answers with the preconceived ideas about CAESAR's use is avoided. Using artificial failures for evaluating the tool might not be the most ideal strategy. Even though participants considered them as realistic, the effect on the results might not be the same as if existing errors from within the chosen project had been selected.

Internal Validity. *Is the accuracy of the results skewed?* The selection of the project for the controlled experiment was based on projects used in previous studies. Thus, it has been proven to be adequate to use for the study's purpose. Although the participation was voluntary and no rewards were given, the participants were self-selected due to the scope of this thesis.

External Validity. *Are the results generalizable?* By conducting the controlled experiment with only six participants, the results might not reflect the usefulness of CAESAR to the developer community as a whole. To mitigate this, participants, having different backgrounds of working with CI and knowledge about the used tools, were selected. Even though some of them work in industrial fields, they are all students at the University of Zurich. Furthermore, only three types of build failures were considered in this controlled experiment. Even if previous research shows, that these are the most common reasons for build failures, the results may not be generalized for other types (especially more complex ones) and build failures in general.

Chapter 6

Summary

Continuous integration is a well known, agile software development practice that supports developers in building software. It helps finding bugs early and thus contributes to more stable code bases and faster release cycles. Despite its wide adoption, developers still face barriers when working with CI. If a build fails on the build server, developers need to scan through large build logs to locate the root of error. Additionally, the context switching between the local development environment and the build server makes locating and fixing an error a time-consuming and complex task for developers.

Recent research proposes tools to support developers in resolving build failures. One of the tools summarizes build logs of broken builds and generates hints based on information found on the internet. In a case study, the tool helped developers to better understand build failures and reduced the time to fix a broken build by 41%. Another tool focuses on automatically repairing failed Maven builds related to dependency issues.

This thesis proposes a different approach for improving build failure resolution; through in-IDE assistance. The developed tool, CAESAR (Ci Assistant for Efficient (Build Failure) Summarization And Resolution), supports developers by bridging the gap between the available information about the build failure on the build server and the local development environment with the source code. CAESAR summarizes build logs, locates and classifies errors, generates error descriptions and hints about their potential cause, and provides this information within the IDE. Subsequently, CAESAR offers to save the current work, load the broken code base into the IDE, directly show the errors within the source code, and instantly debug test failures. In addition, compilation and dependency errors are also supported.

In a controlled experiment with six participants, developers were asked to fix build failures with and without the assistance of CAESAR. On average, by working with CAESAR, the time to fix a build failure could be reduced by 48.4% (which is similar to the 41% time reduction with the build server plugin described before). Along the way, the build log summarization, error descriptions, and hints were considered as helpful in learning more about the cause of the broken builds. Especially appreciated was the direct linking of listed errors within the build summary to the actual source code.

The obtained results show significant improvement in fixing build failures. However, due to the small number of experiment participants, limited diversity of development backgrounds, and only partial support of build failures, these results may not be generalized for all type of build failures and the whole developer population. Furthermore, there still exists the difference between the local development environment and the build server environment which is not included and conquered by CAESAR. Nevertheless, this thesis contributes with a novel approach to support build failure resolution from within the IDE. Using build abstraction, representation, and summarization as well as error detection and classification, CAESAR provides information and assistance right where its needed; within the IDE.

Appendix A

Attached CD

The Attached CD contains the source code of CAESAR as well as study relevant data like the steps to follow for conducting the controlled experiment and the obtained results. The tables below give a short overview of the CD's content. Read the *readme.txt* files located at the specified paths to get further information about the files and directories.

Content	Path
Core Implementation	\caesar\
Plugin Implementation	\caesar-intellij-plugin\
Helper Projects	\helpers\

Table A.1: CD Content Concerning CAESAR (at: *appendix\A-caesar*)

Content	Path
Experiment Preparation Steps	\preparation.md
IntelliJ Plugin to Run the Experiment with	\caesar-intellij-plugin.jar
Demographic Data	\1-demographic\
Introduction Slides	\2-introduction\slides.pdf
Introduction Video	\2-introduction\video.mp4
Preparation Steps for Training Repositories	\3-training\preparation.md
Training Repositories	\3-training\
Preparation Steps for Problem Set Repositories	\4-problem\preparation.md
Problem Set Repositories	\4-problem\
Feedback Data	\5-feedback\

Table A.2: CD Content for the Controlled Experiment (at: *appendix\B-controlled-experiment*)

Bibliography

[1] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 356–367, 2017.

[2] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 572–583, 2018.

[3] Paul Denny, Andrew Luxton-Reilly, and Ewan D. Tempero. All syntax errors are not equal. In *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '12, Haifa, Israel, July 3-5, 2012*, pages 75–80, 2012.

[4] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 291–301, 2013.

[5] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? A controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4):23:1–23:49, 2015.

[6] Alfredo Goldman, Fabio Kon, Paulo J. S. Silva, and Joseph W. Yoder. Being extreme in the classroom: Experiences teaching XP. *J. Braz. Comp. Soc.*, 10(2):5–21, 2004.

[7] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 197–207, 2017.

[8] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 426–437, 2016.

[9] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 41–50, 2014.

[10] Narendra Kurapati, Venkata Sarath Chandra Manyam, and Kai Petersen. Agile software development practice adoption survey. In *Agile Processes in Software Engineering and Extreme Programming - 13th International Conference, XP 2012, Malmö, Sweden, May 21-25, 2012. Proceedings*, pages 16–30, 2012.

[11] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 106–117, 2018.

[12] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: an empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 547–558, 2016.

[13] Noam Rabbani, Michael S. Harvey, Sadnan Saquif, Keheliya Gallaba, and Shane McIntosh. Revisiting "programmers' build errors" in the visual studio context: a replication study using IDE interaction traces. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 98–101, 2018.

[14] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 345–355, 2017.

[15] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Automated unit test generation during software development: a controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 338–349, 2015.

[16] Hyunmin Seo, Caitlin Sadowski, Sebastian G. Elbaum, Edward Aftandilian, and Robert W. Bowdidge. Programmers' build errors: a case study (at google). In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 724–734, 2014.

[17] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[18] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Un-break my build: assisting developers with build repair hints. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 41–51, 2018.

[19] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of CI build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 183–193. IEEE Computer Society, 2017.