Un-Break My Build: Assisting Developers with Build Repair Hints

Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall
University of Zurich, Department of Informatics, Switzerland
{vassallo,proksch,gall}@ifi.uzh.ch, timothy.zemp@uzh.ch

Abstract

Continuous integration is an agile software development practice. Instead of integrating features right before a release, they are constantly being integrated in an automated build process. This shortens the release cycle, improves software quality, and reduces time to market. However, the whole process will come to a halt when a commit breaks the build, which can happen for several reasons, e.g., compilation errors or test failures, and fixing the build suddenly becomes a top priority. Developers not only have to find the cause of the build break and fix it, but they have to be quick in all of it to avoid a delay for others. Unfortunately, these steps require deep knowledge and are often time consuming. To support developers in fixing a build break, we propose Bart, a tool that summarizes the reasons of the build failure and suggests possible solutions found on the Internet. We will show in a case study with eight participants that developers find Bart useful to understand build breaks and that using Bart substantially reduces the time to fix a build break, on average by 41%.

Keywords

Software Engineering, Agile Software Development, Software Development Tools, Build Break, Summarization, Error Recovery

ACM Reference Format:
Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-Break My Build: Assisting Developers with Build Repair Hints. In *ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension, May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages.

1 Introduction

Continuous integration (CI) is an agile software development practice that advocates frequently integrating code changes introduced by different developers into a shared repository branch [11]. An automated system builds every commit, runs all tests, and verifies the quality of the software, e.g., through automated static analysis tools [6]. This helps to detect issues earlier and locate them more easily [12]. CI is widely adopted in industry and open source environments [36] and has already proven its positive effects on release frequency, software reliability, and overall team productivity [10].

Despite its undisputed advantages, the introduction of CI in established development contexts is anything but trivial. Hilton et al. [9] found that *build breaks* are a major barrier that hinders CI adoption and various reasons exist for a build to break [35], e.g., compilation, testing, code quality, or dependency resolution. Developers need to learn how to efficiently identify the reasons for a build break and, unfortunately, the required skill set is still different to traditional debugging. Established techniques that are widely used in the development environment [22], like setting breakpoints to investigate a program right before a crash, are not applicable, which makes it difficult and time consuming to remove a build break [9]. As a result, developers spend a significant amount of their working time comprehending and solving build breaks. It takes on average one hour to fix build breaks [12].

Those results motivate the need for new ways to support developers in *understanding* build breaks and in *deriving a fix*. Existing works have already proposed automatic build-fixing techniques, e.g., [15]. However, such approaches are typically limited to a specific type of build break (i.e., fixing unresolved dependencies). In this paper, we propose a *developer-oriented* assistance system that supports build break fixes by summarizing available information and linking to external information. We do not focus on a specific build problem, but empower the developer by providing relevant information in a wide range of build failures. To the best of our knowledge, we are the first to propose such an *information-centric* developer support during build breaks. We will investigate to which extent generated summaries can help developers with comprehending build logs. We will also empirically analyze the effect of a build summarization tool on the time needed to understand and fix a build break. More specifically, we will answer two research questions:

RQ$_1$ Are summarized build logs more understandable?
RQ$_2$ Does a semi-automated support system influence the time that is required to fix a broken build?

We have implemented the *Build Abstraction and Recovery Tool* (BART) to study these questions. Bart is a Jenkins plugin that summarizes failed build logs and that links related StackOverflow discussions to help solve the build failure. To answer both research questions, we deployed Bart in an empirical study with eight developers. Our results show that developers consider the generated summaries helpful for fixing build breaks; as a further result, the resolution time for fixing the build can be significantly reduced.

In summary, this paper makes the following contributions:

- Presentation of a novel idea to support build fixing through build log summarization and linking to StackOverflow resources.
- Proof-of-concept implementation for Bart.
- Investigation of the effect of understandability of build failures on the fixing and validating of builds.

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

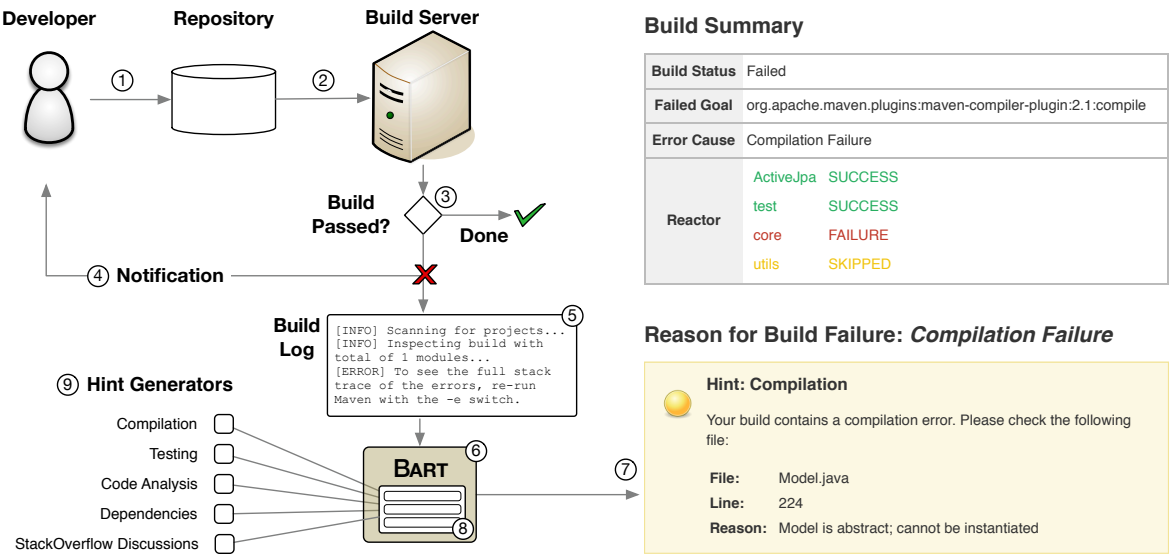

Figure 1: Overview over the Build Summarization Approach

2 Overview

To understand our vision of *developer-oriented* assistance, it is important to reflect on the typical CI workflow that is illustrated in Figure 1. Developers working in such a workflow synchronize their *working copy* frequently with the central repository that is shared by all team members (1). They pull changes from others and push their own contributions. The repository is being monitored by the build server. Every time a new commit is pushed to the repository, the build server will update its working copy and build the project (2). This typically includes multiple stages of a Maven build, for example, compiling the sources, running the tests, generating documentation, or validating the software quality through static analysis. If all these stages have passed (3), the build is considered to be successful, which typically results in the release of the software. If the build fails, on the other hand, developers are being notified by the build server about the error. This typically happens through sending an email or by visiting the web frontend of the build server (4). The developers have to consult the build log (5) to understand the problem and provide a fix, a difficult and time-consuming task that typically consists of three steps.

Log Inspection The developer investigates the build log to get further information about the build failure. While it is often simple to spot the part in which the build failed, it is very often difficult to read the log and to understand the failure reason.

Hypothesis Once the developer has an intuition about the root cause for the break, the problem should be replicated, if possible on the developer machine and ideally by providing a test. This makes it possible to use a debugger to inspect the failure.

Fix If the root cause of the build failure is identified, fixing it is usually the easy part. The developer implements the fix, pushes it to the repository, and waits for the result of the new build. All the steps are re-executed if the build fails again.

Executing these three steps is difficult and deriving a hypothesis about the root cause of the failure requires experience. If the developer gets stuck, a common strategy is to ask more experienced team members [13] or to search on the Internet for solutions [33].

In this paper, we present Bart (6), the *Build Abstraction and Recovery Tool* that supports developers by enriching the build log through summarization and linking of external resources. We have designed Bart as a support tool for Maven builds and we have created a proof of concept implementation for the build server Jenkins. Our solution is complementary to the existing workflow that we have discussed before. Bart does not replace the inspection of the build log, instead, the build log is embellished with further information to facilitate a faster and better decision making of the developer. For example, our screenshot (7) shows the *Build Summary* (a general summary of the build result) as well as a list of *hints*, in this case details about a compilation error. These hints are included in the Jenkins build overview page.

Bart facilitates the generation of these hints with two reusable parts. First, it parses the build log, extracts all relevant sections (e.g., keywords, commands, built modules), and stores this preprocessed information in a meta-model (8). Second, Bart is extensible through additional *Hint Generators* (9). We have built five different hint generators that summarize the information found in the build log, as well as hint generators that use information from the build log to search for solutions in the Internet. For example, our proof-of-concept implementation can link to related discussions on StackOverflow.

We will introduce these individual parts in the remainder of this section. Section 2.1 introduces our build-log meta model and describes our parsing. Section 2.2 contains the extension point mechanism for hint generators and a description of the four different hint generators that summarize build log information. Section 2.3 discusses the hint generator that links build failures to external information, such as discussions on StackOverflow.

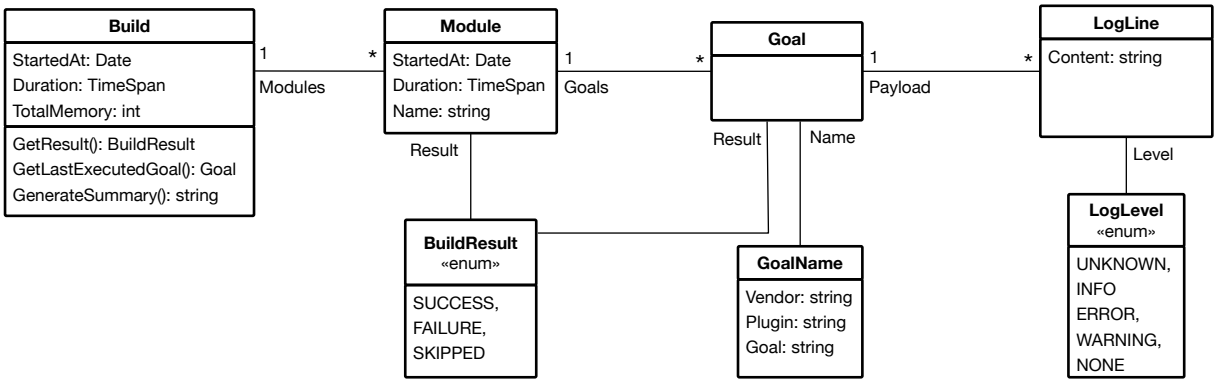

Figure 2: Meta-Model that Is Available to Hint Generators In Bart

2.1 Detecting Failure Information in the Log

Build tools typically log all their actions in a detailed log that allows developers to reconstruct their actions after the fact. Such a build log is typically stored as plain text. All details about the build are contained, but such logs are very large, e.g., even the build log of the relatively small Maven build tool itself (~ 130K LoC) results in a build log of more than 1,500 lines. To make the creation of new hint generators in Bart straight-forward, we preprocess these logs. We provide an abstraction over a Maven build log that makes it easy to find exactly the information that the hint generators need. While we are going to focus on the Maven build system [17], the most used build tool among Java developers [16], the underlying idea is general and can also applied to other build systems. In this section, we will first briefly describe Maven's building concept, the structure of its build log, and our parsing. We will then introduce our meta model that we use to store the relevant build information.

Maven follows the concept of convention over configuration. It provides a standard build configuration that defines several *phases* that are run one by one in the default build lifecycle (e.g., `compile`, `test`, `verify`). The set of phases is fixed and most of them are empty by default. A concrete build job can now add specific *goals* to the different phases, if needed. So, for example, a project could add the invocation of a *static analysis tool* to the `verify` phase. In practice, build files contain the configuration for many of such *goals* that range from dependency resolution in the very beginning of builds to packaging or deployment that typically take place at the end.

Maven builds can be hierarchically organized. In addition to the goals that are configured in the build file for the current *module*, parent configurations can be references, from which all configuration options are inherited. In addition, it is possible to refer to *submodules* that are then build together with the current *module*.

At each build and starting from the module for which the build was triggered, Maven creates the dependency tree between all (sub-) modules and schedules the individual builds in an order that does not violate their dependencies. In Maven terminology, this *build plan* that contains the names of all *modules* is called the *reactor*. During the build, one *section* is dedicated to each *module*. This section contains entries for each executed *goal*, which might also prints additional output to the log. At the end of the build, Maven generates a *reactor summary*, which again contains the individual build results. In addition, the reactor summary will also list the consumed memory, and -in case of a build failure- further information about the module and goal, in which the build broke.

A developer that has to read such a build file, has to navigate through a big log to find the relevant information. This is also a hard task for an automated processor, because the individual parts need to be parsed or otherwise processed with string utilities.

To simplify the access to the contained information, we implemented a parser that, taking a build log as input, fills the meta-model that we have created, as depicted in Figure 2. The model follows the structure that we have introduced before. The root entity of a build log is a `Build`, which has basic properties like the required memory for the build. A build refers to several `Module` definitions that are part of the build. In addition to timing information, each module has a unique name and a result. It also contains information about the different `Goals` that were executed while building it. Each `Goal` combines the `GoalName` (i.e., a reference to the tool that was run), the `BuildResult` or the invocation, and a `Payload`, which contains all output that was generated for this `Goal`. Each line is annotated with a `LogLevel` and contains some content as a `string`.

The original build log contains a *reactor summary* at the very end that can be requested by calling `Build.getSummary()`. We do not explicitly store the contained information in the meta-model, because it can be fully inferred from the stored data. In general, this meta-model does not lose any information of the original log. It splits information into individual sections and provides easy access, but it could be transformed back into the original log file.

Please note that the parser, which we built for this paper, does not support the complete meta-model yet. We have focused our implementation on the parts that were required for the experiments in the remaining sections of the paper. This does not represent a conceptual limitation of the meta model though and can be solved by spending more implementation effort on the parser.

2.2 Summarizing Build Log Information

Understanding the extensive build logs generated through a Maven build is tedious. The *reactor summary* that is automatically generated at the end of a build contains basic information about a build failure and represents a first step in the right direction. However,

the information is presented as plain text without any highlighting and it is hard to read by developers. In addition, only the lines of the failing goal that are marked as *error* are included in the summary, surrounding information, which could further explain the error, is omitted. We are convinced that this current build summarization is not sufficient. Thus, we propose an improved summarization and a highlighting of the important pieces to ease life of developers, and we make them more efficient in understanding the build log.

Inspecting the failing section of the build log should provide all required information to understand the cause of the failure, so it represents the starting point of any investigation of a build failure. However, the actual information that is included in the corresponding part of the build log can be very extensive (e.g., long lists of executed tests) and it also heavily depends on the failing goal. Fortunately, it has already been shown that build failures can be assigned to different categories, based on the goal or build step that failed [35]. We propose to provide a better guidance in the fixing process by tailoring the summary to the failure category.

Algorithm 1 shows the conceptual framework that we use to present build summaries. The actual implementation is highly integrated in the build server Jenkins and presents the output in the polished form that is shown in Figure 1. We first parse the build log into our meta-model. We do not add hints to successful builds, so we skip non-failed builds in the algorithm. The build status and the individual build goals are directly available in the model. For a failing build, we determine the failure category based on the last executed goal. Using this category, it is possible to filter for applicable hint generators to save execution time. Every hint generator is asked to provide hints that might point the developer towards a build fix. A `Hint` is just a key-value dictionary that can contain arbitrary contents and hint generators can return an empty list, a single hint, or also multiple hints. The presentation to the user is achieved by iterating over all hints and by putting all their key/value pairs into a table. We do not imply any restriction on the type of key that can be generated, because the actual hint generator should select the most meaningful information for the developer.

In this paper, we focus on a proof-of-concept implementation that supports the most frequent build break categories [26, 29], i.e., *Compilation*, *Dependencies*, *Testing*, and *Code Analysis*. In addition, we want to provide an additional *Build Summary* that provides an overview over the whole build. In the following, after describing such overview, we will briefly introduce these different hint generators, which information we want to show to the developer in each case, and how we can get access to the required build log data.

Build Summary The Build Summary provides a high-level overview over the result of the build. It mimics Maven's reactor summary, but reduces the amount of information to a minimum. Rich formatting options are applied to highlight the different information. You will find an example of the build summary in Figure 1. Each summary is composed by the following sections.

Reactor Summary: The list of modules can be requested from the `Build` object, their individual results can be directly used.
Build Result: Can directly be requested from a `Build` object.
Failed Goal: The last executed goal of a failing `Build`.

Algorithm 1: Conceptual Framework for Generating Hints

```
1  buildLog ← ...;
2  registeredHintGenerators ← ...;
   // parse model
3  model ← parse(buildLog);
4  if model.Result == FAIL then
5      cat ← determineFailureCategory(model);
6      gens ← findApplicableHintGenerators(cat);
       // generate hints
7      hints ← list();
8      forall gen in gens do
9          curHints ← gen.getHints(model);
10         hints.addAll(curHints);
11     end
       // show them to the user
12     i = 0;
13     forall hint in hints do
14         print("Hint_" + i);
15         i ← i + 1;
16         forall key in hint.keys do
17             print(key + " : " + hint[key]);
18         end
19     end
20 end
```

Error Cause: The error cause consists of the error message that is printed by the failing goal. These can be extracted by selecting all lines of the goal that have the log level "error".

Compilation Failures The hint generator should provide detailed information about the location of the compilation error. All this information can be found in the `Payload` of the failing goal.

Type: Name of the type (e.g., class) that could not be compiled.
Line: Line number, in which the error has occurred.
Reason: Textual description of the compilation error, e.g, instantiation of an abstract class, when provided by the build log.

Dependency Failures Declared dependencies can lead to various build failures. Our summarizer helps understanding the dependency error by showing the following information.

Dependency: The name of the dependency that causes the failure. The Maven coordinates of the dependency are mentioned in the error message and we use a regular expression to parse them.
Reason: Textual description of the dependency error. Typical reasons are invalid versions numbers or missing internet access.

Testing Testing failures are particularly tricky to fix, because they can occur after introducing a change in a completely different part of the system. For this reason, it is important that a hint does not only contain the location of the test, which is required to replicate the failure locally, it should also contain the reason that explain the failure. As a result, the hint generator reports the following:

Location: The location, in which the testing failure occurs. This contains both the *test class* and the failing *test case*.

Stack Exchange Analysis

Related Post 1:

This is normal behaviour. Abstract classes are not supposed to be instantiated. You should test the classes which inherit from the abstract class, not the abstract class itself.

Full Discussion: https://stackoverflow.com/questions/5028082/rails-3-activerecord-abstract-objects

Figure 3: Hint that Links Related StackOverflow Discussion

Reason: A textual description of the test failure. This is taken from the *failed* assertion statement, so the quality of these descriptions depends on the concrete test case. In case of an *error*, also the stack trace of the failure will be included.

Code Analysis Many builds use static analysis tools to validate properties of the system. For example, projects apply Checkstyle [5] to ensure a consistent programming style in the code base. Each such tool produces a different output and individual hint generators are needed to cover them. We selected Checkstyle as a representative for such static analysis tools and include the following information that help to understand related build failures:

Location: The path to the *file*, in which the style violation was detected. The location also includes the *line number*.

Reason: Name of the style rule that caused the failure. These names are typically very expressive, e.g., "method name too long", so the proposed hints are potentially very meaningful.

Future Extensions Future hint generators might require other information from the build log in their hints. They can either reuse our meta model or provide their own extraction strategy to find the interesting information in the build log. It is possible, for example, to use custom regular expressions to parse specific information from the `Payload`. As a fallback, it is always possible to recover a full build log from our meta-model, which ensures support for all hint generators that work on the build log. Extensions that require *external files* in addition to the build log, like test coverage reports, represent a special case. These files are not contained in the build log. Hint generators that require access can still parse the respective path from the build log and open these files separately.

2.3 Hints from External Sources

Summarizing local information improves the ability to understand the contents of the build log. However, developers may encounter situations, in which the error message is easy to understand, but requires a complex fix. For example, if the *source level* is not configured in Maven, it will use Java version 5 by default. If the developer now writes Java code in a newer version, e.g., version 8, and uses one of the newly introduced constructs, e.g., lambda expressions, the Maven compiler plugin will fail with a syntax error, even though no problem will be reported in the development environment. While the summary will point out a syntax error very clearly, in this case, an inexperienced developer will struggle to solve this on their own and will most likely ask a more experience colleague for help or simply search for a solution on the internet. For this reason, we also need to provide an infrastructure in Bart that allows the creation of hint generators, which can go beyond a *local summarization*. These *external* hint generators should be able to add additional hints and link to external resource in their suggestions for possible solutions.

Algorithm 2: StackOverflow Recommendation Algorithm

```
buildLog ← ...;
hints ← getSummaryOfLocalHintGenerators(buildLog)
// query generation
query = createStackOverflowQuery(clean(hints));
posts = searchOnStackOverflow(query);
// ranking
cleanedLog ← clean(buildLog);
words ← split(cleanedLog);
keywords ← removeStopWords(words);
forall post in posts do
    post.score ← countKeywords(post, keywords);
end
proposals = posts.orderBy(p : p.score, DESC).take(4);
```

It is very likely that, in case of a build failure, a similar build break has already been discussed online. Previous work has already shown that question and answer sites, like StackOverflow, can provide a great source of information to support developers [24]. The site contains almost 60K discussions that are related to Maven development [31], which makes us very confident that it can also be a good source for tips on how to fix a broken build. An example of a hint that refers to a StackOverflow discussion is shown in Figure 3. The example hint explains a specific compilation failure and also links the full discussion to provide additional context.

To obtain relevant discussions from StackOverflow, we use a twofold approach. First, we query StackOverflow for discussions that are related to the build log. Second, we rank the returned posts and present the most relevant discussions to the developer. The exact algorithm is shown in Algorithm 2. The hint engine starts with requesting the build log and the hints that have already been generated in the *local summarization* step. Given that the *local* hint generators have already identified the key parts of the build log, we make use of this information to create a query that is as specific as possible. The hints are being *cleaned* by removing local information (e.g., paths or file names) and common overhead that is added in every Maven build (e.g., formatting characters or *goal* names). The resulting query mainly contains the error message that describes the failing build and it is used to search on StackOverflow.

In a second step, the algorithm ranks the returned posts to identify the ones that are most related to the actual build log. To achieve this, the build log is first *cleaned* in the same way as the query and then *tokenized* to create a set of keywords that describe the build. Common english stop words (e.g., "the", "or", "and") are removed to improve this set of keywords. For each post, we calculate a post score by counting how many different keywords are used in the body of the discussion. After ordering the posts by their score, the top four proposals are selected and shown to the developer.

3 Empirical Study

We conduct an empirical study to investigate Bart's capability to improve the *understandability* of build failures and the *performance* of developers when fixing broken builds. Our study consists of two

Table 1: Analyzed Projects

Project Name	Version	Size (Loc)	#GitHub Stars	#Builds
ActiveJPA	0.3.5	39,335	143	123 (master)
Sentry-java	5.0.0	113,332	312	509 (master)
Fongo	2.1.1	31,088	374	404 (master)

parts, a *controlled experiment* and a *questionnaire*, that cover the different aspects that we want to investigate.

Understandability: In the first part, we assess whether or not the summaries generated by Bart make it easier to understand the cause of a build break and to formulate a solution strategy.

Performance: In the second part, we measure Bart's effect on the required time to fix certain types of build breaks.

In the following, we will introduce our methodology that we have applied to investigate both aspects.

3.1 Context

The *context* of our study includes (i) as *objects*, build breaks that we have generated from selected Java projects, and (ii) as *subjects*, developers that participated in our controlled experiment.

The three software systems that we considered in our study are illustrated in Table 1. ActiveJpa [2] is a Java library that implements the active record pattern on top of Java Persistence APIs (JPA). Sentry-java [28] is an error tracking system that helps developers to monitor and fix crashes in real time. Fongo [7] is an in-memory Java implementation of MongoDB. The considered systems are hosted on GitHub and built on the TravisCI [32] platform. We followed the methodology of Bavota et al. [3] to select systems that developers can easily get familiar with and that are, at the same time, representative for real software systems. While our selected systems have less than 500K lines of code, they are very popular (more than 100 stars on GitHub) and frequently built (more than 100 builds on the master branch). For our study, we have injected bugs into these systems to generate broken builds. The introduced bugs belong to the four most recurrent categories of build failures [35], i.e., *compilation*, *dependencies*, *testing*, and *code analysis*. We created different mutations of the extracted systems for every category of broken builds and ended up with five broken ActiveJPA versions, two broken Sentry-java versions, and one broken Fongo version.

More details about these broken versions are depicted in Table 2. We always created two mutated versions to avoid learning effects in both tasks of the study. To generate the *testing* build breaks we changed an assertion in the test class `FongoAggregateProjectTest` from `assertNotNull` to `assertNull`. We have also altered the `count` method of the class `org.activejpa.entity.Model` by adding 100 to the returned value, which causes the related test to fail. To provoke *dependency* build breaks, we have inserted an obvious non-existing dependency. For the second case, we included a typo in an existing dependency. We have introduced two *code analysis* build breaks in Sentry-java, by adding a new method with a very long name to the class `SentryAppender` and by deleting the Javadoc comment of the method `doClose` in the class `AsyncConnection`. Both are picked up by CheckStyle, which will raise the errors *Very long function name* and *Javadoc has empty description section*. Finally, to create *compilation* errors we added a return statement in the void method `close()` of the class `org.activejpa.JPA` and inserted an illegal combination of `static` and `abstract` in the signature of the method `deleteAll` of class `org.activejpa.entity.Model`.

We contacted participants by sending out invitations to students from the University of Zurich (UZH) and Swiss Federal Institute of Technology in Zurich (ETHZ). In total, eight students participated in our controlled experiment. The majority of our participants (5, 62.5%) report three to five years of programming experience, while two participants (25%) declare that their experience even exceeds five years. Only one participant reported less than three years of programming experience. Six participants (75%) have already obtained at least a Bachelor's degree and six participants (75%) declared that they work as professional developers, with overlaps between both groups. We asked the participants to self-estimate their programming experience in a five-point *Likert scale* [14] from *very low* to *very high*. Out of the 8 participants, 3 reported an experience level of *above average* or higher (*very high*: 1). Only 2 participants rated their experience as *below average* and no one rated their experience as *very low*. Our participants represent a small but diverse group with different backgrounds. While all participants are early career software developers, some of them have already extended programming experience, which allows us to study build failure resolution along developers with different degrees of expertise.

3.2 Experimental Procedure

The empirical study we conducted with our participants consists of two different tasks and was supervised by one of the authors. We provided summaries and solution hints generated by Bart to our participants to study the *understandability* of build breaks. In the second task, we investigate whether Bart can speed up the fix.

First Task: Understandability In the first task, our participants answered a questionnaire about the *understandability* of the build break summaries provided by Bart. We used Bart to generate summaries and solution hints for the broken builds of the mutated software components in Table 2 (*Task 1*) and asked our participants to evaluate them. We provided our participants with the following three questions and we asked them to answer on a five-point Likert scale from *very high* to *very low*:

- How much did your understanding of the build failure improve through the summary of the build log?
- To what extent do the suggested solutions help you in conceiving a strategy to solve the build failure?
- To what extent are the suggested solutions applicable to the specific build failure?

We have also requested basic demographic information in the survey to better understand the background of our participants.

Second Task: Resolution Performance In the second task, we measured the time it takes developers to fix a broken build to analyze the effect of Bart. Every participant was asked to fix two of the four manually injected bugs for Task 2. We have designed the experiment as a *between-subject* study and each participants had to fix one bug with treatment (i.e., support through Bart) and

Table 2: Mutated Components in the Analyzed Systems

Build Break Type	Task 1		Task 2	
	Project	**Mutated Component**	**Project**	**Mutated Component**
Test	Fongo	com.github.fakemongo.FongoAggregateProjectTest	ActiveJPA	org.activejpa.entity.Model
Compilation	ActiveJPA	org.activejpa.jpa.JPA	ActiveJPA	org.activejpa.entity.Model
Code Analysis	Sentry-java	net.kencochrane.raven.connection.AsyncConnection	Sentry-java	net.kencochrane.raven.log4j.SentryAppender
Dependencies	ActiveJPA	pom.xml	ActiveJPA	pom.xml

one without. We have avoided learning effects between the two different fix attempts of each participant by changing the type of build failure and by changing the software component, in which the bug was introduced. In total, we tested eight scenarios and each of the four different build failures was fixed twice, once with and once without treatment. We assigned the different scenarios such that four participants started with the treatment and the other without treatment for the first build fix. All participants managed to fix both their assigned builds without external help.

One of the authors supervised the task. Before starting it, he introduced Bart and asked the participants to import the assigned projects into their development environment. Furthermore, the supervisor gave our participants time to get familiar with the projects and with the Jenkins instance that was used to build the projects. Note that in case of a build failure, Jenkins produces a build overview that indicates the build result (i.e., Failed), the last Git commit that was pushed to the remote repository, and the name of the committer. In addition, Jenkins provides access to the generated build log. To start the fix attempt of the build failure, our participants were asked to trigger a new build of the assigned project and to repair the resulting build failure. The supervisor of the task measured the *resolution time*, i.e., the time between the build break and the next build success. The same methodology was applied for the second build fix attempt. After finishing the experiment, we discussed the usability of Bart with the participants in an unstructured interview.

4 Results

This sections presents the outcome of our study. We will discuss the results and will answer our research questions.

4.1 Understandability of Build Breaks

Our first research question was how build summarization can improve understandibility. To answer this question, we evaluate the ratings of our participants for the generated summaries of Bart. We visualize the answers in three *diverging stacked bar charts* [27] that illustrate their rating regarding the *understandability* of the summaries (Figure 4), their *relevance* (Figure 5), and their *applicability* to the build break (Figure 6). We use the *Likert* values *very high*, *above average*, *average*, *below average*, and *very low*.

Understandability Figure 4 shows how participants rate the understandability of Bart's summaries compared to the raw build logs that are provided by Jenkins. All participants agree across the board that the understandability of the build break summaries is at least *above average*, with the majority saying that it is *very high*. Only in a single case, for the dependency related build break, one participant found that Bart's summary was comparable to the default build-log presentation in Jenkins, but that it did not improve it.

The developers seem to agree that Bart's summaries helps them to better understand the build log. One of the participant describes the actual build logs as *"cryptic" (S8)*, which could be caused by a lack of experience in reading it. However, the overloading amount of information that is contained in a build log is a recurring theme in the answers of our participants, even from experienced developers. Another participant said that *"Maven logs tend to be verbose, having a quick summary [...] greatly reduces the time needed to find and correct a build failure" (S5)* and another one that *"[Bart] helps to find the programming errors quickly" (S4)* and *"a structured summary is way easier to grasp than many unstructured lines of text" (S4)*.

> *Our participants almost unanimously agree that Bart's build summaries improve the understandability of build logs.*

Relevance & Applicability Figures 5 and 6 illustrate the relevance and applicability of the proposed solution hints from StackOverflow. The solutions hints for *compilation* and *code analysis* breaks were mostly positively rated. Most our participants found their relevance and applicability *above average*, more than half of them rated them even as *very high*. However, two participants find the applicability of the solution for the *code analysis* break *below average* and one of them, according to the background information a very skilled developer, has also considered the relevance of the solution *below average*. The one participant that has considered the solution hint for the *compilation* build break as *very low* has little programming experience and uses Java only occasionally. We assume that he simply did not understand the suggested hint.

> *Most study participants find the solution hints for build breaks caused by* compilation *and* code analysis *errors relevant and applicable.*

In case of the *dependency* build break, the participants do not agree on a rating for the relevance and applicability of the solution hints. The ratings are centered around *average*, some of the participants find the suggestions relevant and applicable (one participant considered it even *very high*), while others rate it *below average*. Two participants even think that the applicability of the solution hints is *very low*. One of them is no frequent Java user, but the other one has a very strong background in Java programming, so a lack of expertise alone does not explain the different ratings.

> *Suggested solutions for* dependency *errors are often not considered as valuable hints by our participants.*

Our respondents were also not convinced about the relevance and applicability of solution hints for *testing* build breaks. Most of our participants consider them *below average* or even *very low* when

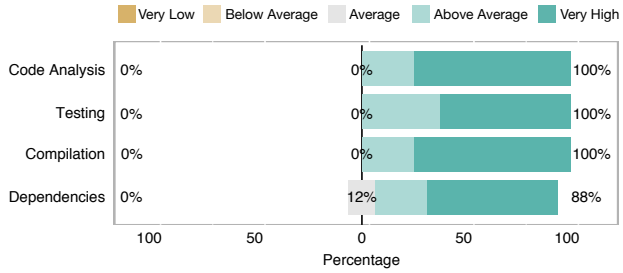

Figure 4: Understandability of Summaries

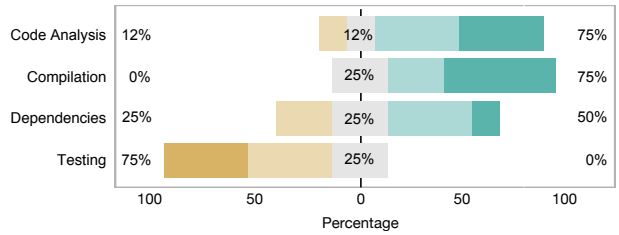

Figure 5: Relevance of Proposed Solution

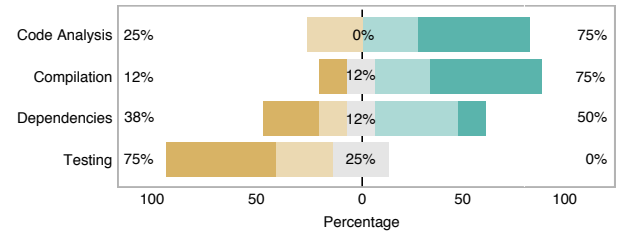

Figure 6: Applicability of Proposed Solution

compared to the original build log. One possible explanation is that the available information in the build log of a failing *testing* build, i.e., the name of the failed test, is project specific. This makes it impossible to find related solutions for such local errors in external resources without taking other information into account.

> *Testing-related build failures are project specific. The build log alone is not sufficient to identify related external resources.*

4.2 Resolution Time of Build Failures

Our second research question was whether Bart can reduce the time that is required to fix a build. To answer this questions, we have asked our participants fix the failing builds of the second task and measured their required time. Table 3 illustrates the results of this experiment, it shows the average time of both approaches to repair the different build failure types with and without Bart. While the previous research question has revealed that the ratings for relevance and applicability of Bart's solution hints differ between the build break types, the results of the second task shows that using Bart leads to a substantial reduction from 27% to up to 62% in the required time to fix a build across all scenarios. We will discuss the different break types individually to explain what seems to be a contradiction at a first glance.

Code Analysis and Compilation The study participants found that Bart's summaries improve understandability and that the solution hints are both relevant and applicable. These positive ratings can also be confirmed in the practical task. The time to fix a build break could be reduced by 32.6% for build breaks related to *code analysis* and by 27.3% for build breaks related to *compilation* errors.

The error messages of both *compilation* and *code analysis* are self-explanatory, but a certain degree of expertise is needed to understand them. Fortunately, the exact same error messages and warnings appear in other projects as well, so it is easy to find information online that provides context to understand the error message. Our solutions hints are able to enhance the description of a warning or even replace it and the developers get an explanation of an error or of a violated rule without having to look it up on external resources. This aspect is particularly useful when the developer is not used to a specific code analysis tool.

Our participants found it very helpful that Bart integrates all required information to understand the meaning of a rule violation *"Less searching for the relevant part in the error message, hence faster bug resolution" (S2)*. Moreover, the links to StackOverflow are highly appreciated when the meaning of a warning is non-obvious. *"In the less obvious error causes, the stack overflow extracts prove to be very useful" (S5)*. In these cases, the StackOverflow discussion about the proper solution can provide additional context information to understand the problem. The StackOverflow solution hints speed up the development process, because *"You can often get the information from bart without having to search the internet" (S6)*.

> *In addition to the summary, solution hints can provide the required context that helps with understanding the cause of a build break.*

Dependency and Testing The relevance and applicability of the suggested solution hints were not considered useful for *dependency* breaks and *testing* failures. These low ratings can easily be explained though. A search for the corresponding error message would either return many unrelated resources (e.g., cases in which other developers had trouble with some other dependency) or none (because the error message of a test failure is project specific). However, we could still see a substantially reduced fixing time for both categories. The improvement for *dependency* related build breaks (62.4%) has even been the most significant reduction among all considered cases.

When considering error messages in both categories, it becomes apparent that both are typically very self-explanatory. The errors immediately point to the missing dependency or name the failing test. The required action to fix such issues is straightforward: search for the missing library in the *Maven Repository* and add it to the build file or fix the failing test, respectively. The participants that fixed such kind of breaks have reported that the reorganization of the information contained in the build logs significantly reduced the amount of time needed to understand the cause of a build break. One of our participants stated that *"[Bart is] mostly a timesaver, not really a skill enhancer. Carefully reading the log usually allows the extraction of the same information" (S5)*. Another participant found that *"directly serving the relevant solution, the debugging process is drastically sped up" (S1)*.

Another important aspect that affects the time to fix a build is the debugging environment. Previous work has shown [9] that CI server like Jenkins do not provide sufficient support to debug a build

Table 3: Average Resolution Time per Build Failure Type

Build Break Type	excl. Bart (s)	incl. Bart (s)	Reduction (%)
Testing	531	310	41.6%
Compilation	196	142	27.3%
Dependencies	303	114	62.4%
Code Analysis	158	107	32.6%
Overall Average			41.0%

break. According to our participants, however, Bart summaries *"add more capabilities to the environment" (S6)* compared to the raw build logs and *"might make debugging unnecessary when the bug becomes evident" (S2)*. They report that *"if some tests fail, the Bart output can be helpful in finding out why" (S5)*.

> Dependency *breaks and* testing *failures seem to be easy to understand. Providing a good summary that highlights the locality of the issue seems to be the most crucial factor on fixing time.*

Overall, it seems that the different build break categories require different support strategies. Some categories benefit from links to external resources that provide additional context about an error (e.g., *compilation* errors), others benefit more from an improved summarization (e.g., *testing* failures). Bart combines both in one tool and substantially reduces the time to fix a build break across all scenarios in our study, on average by 41%.

5 Implications and Future Work

Our findings have important implications for both researchers and vendor of automated static analysis tools. Existing CI servers provide a build overview, but refer developers to the build logs for detailed information, e.g., to understand the reason for a build failure. Our results suggest that build logs are difficult to understand though and that summaries of the build failure should be directly integrated into the build overview to support developers in the comprehension process.

We show that providing solution hints that link to external resources can be useful to developers and that they can provide additional context, which can be helpful to derive a solution strategy, especially when the root cause of a build failures is unclear or when the solution is non-trivial. So far, our infrastructure only considers information from the build log to identify related resources. Future hint generators should consider other resources produced during the build, like generated reports or information about deployed libraries, to create a more holistic picture of the failure. A better context awareness of the summarization tool might help to overcome existing limitations, e.g., for testing related build breaks.

This work introduces a technique to support developers when fixing a build break by providing them with summarization and solution hints. However, some build breaks cannot be reproduced locally and need to be solved on the server. Future work should investigate new ways of bridging this gap by considering differences between the remote environment on the CI Server and the local IDE environment when searching for solution hints. Also novel debuggers that are tailored to the CI workflow might help to improve the effectiveness when fixing build breaks.

Assistance tools like Bart do not only have a positive effect on the developer that fixes the build, they also reduce the downtime of the team that is caused by the build break. Supporting the individual developer has the potential to increase the team productivity. Future work should find novel summarization techniques for build log summarization to reduce the required time even further.

6 Threats to Validity

The work presented in this paper was carefully planned and executed, but several threats to validity exist for our results. In the following, we will discuss them and our mitigation strategies.

Threats to *internal validity* concerns the confounding factors that might have affected our results. The broken versions that we artificially created could be not representative of errors occurring in the reality. We tried to mitigate it by injecting realistic bugs that, according to previous work [29, 35], are the most common causes of build breaks. Another aspect that might affect the reliability of our results is the complexity of systems considered during the analysis. We tried to reduce this threat by considering build breaks in our study, which belong to projects that are not too big, but at the same time representative of real systems. It is also possible that our participants didn't fully understand the questions in our questionnaire. We have reserved time before starting, to allow participants to ask questions about the experimental procedure. Another threat is the manual time measurement that could introduce a bias. However, the substantial differences that we have measured far exceed the imprecision of the manual measurement. Other build summarizers might exist and requiring our participants to read a plain-text build log could introduce a bias in our experiment. However, we are not aware of any frequently used summarization tools and we think that using the information that is available in a standard Jenkins installation represents a valid baseline for our comparison.

Threat to *external validity* concern the generalizability of our findings. We considered only four types of build breaks in our study. However, those represent the most relevant and recurrent categories of build breaks that have been observed [4, 18, 29]. Furthermore, the participants to our study could be unrepresentative of all kind of developers. We mitigated such threat trying to reach people with different programming skills, to make general consideration about beginner and expert developers. Our tool, Bart, is the first implementation of an approach for build logs summarization. The current design of our study only looks at errors introduced by users. Future work should expand the scope and investigate build errors that are caused by the environment of the build server (e.g., different locale settings). The results presented in this work might not generalize beyond the considered build failure types.

7 Related Work

This paper is related to three lines of research: works on build failures, source-code summarization, and mining Q&A sites. In the following, we will discuss the most related previous works from these areas and relate them to the work presented in this paper.

Build Failures Prior studies have investigated the nature of build breaks. Miller [18] found that the most recurrent causes of build failures in Microsoft projects are poor code quality, testing failures and compilation errors. Other researchers [4, 26] studied the

frequency of different build failure types in open source projects, finding that builds generally fail because of failed test cases. In our study, we focused on the most common build break types according to those studies. While several works focused on one particular type of build failure, e.g., code quality [39] or compilation [29], Vassallo et al. [35] proposed a broader taxonomy of build failures. They have analyzed 418 Java-based projects from a financial organization and 349 Java-based OSS projects and have identified differences and commonalities of failures between industrial and open source projects. Because we summarized Maven build logs of Java projects, we decided to reuse this taxonomy to categorize our build failures.

Kerzazi et al. [12] have analyzed 3,214 builds in a large software company over a period of 6 months to investigate the impact of build failures on the development process. They observed a high percentage of build failures (17.9%), which aggregate to a cost of more than 2,000 man-hours when each failure needs one hour of work to be fixed. Thus, build failure slow down the release pipeline and decrease the team productivity. This was one of the motivation for our study: providing developers with a tool able to support them while fixing build failures making the recovery process faster.

Existing plugins try to achieve the same goal. For example, Log Parser [1] is a Jenkins plugin that allows developers to add custom parse rules in the form of regular expressions. Matching parts of the build log are then highlighted for the developer. Bart pursues a different goal. It automatically selects the relevant information with no effort required from developers and organizes this information in summaries and by linking external information.

Other researchers tackled the problem of speeding up the build failure repair not considering the information contained in the build log. Macho et al. [15] proposed an approach to automatically repair Maven builds that break due to dependency related issues. They propose three repair strategies for an automated build repair, i.e., *version update*, *delete dependency*, and *add repository*. Their tool, BuildMedic, was able to repair 54% of dependency-related build breaks. The focus of Bart is *developer-oriented* and complementary to automated approaches. We assume that very often developer interaction is required to fix a build. Therefore, we try to empower the developer by improving build log understandability though summarization and linking to external resources.

Source-Code Summarization During their regular work, developers have to cope with a large amount of external data [8], e.g., bug reports or source code, which is produced during software development. They need support while trying to comprehend such data and summarization techniques can facilitate this process. Several techniques have been proposed to summarize source code [21]. Haiduc et al. [8] proposed automatic source code summarization leveraging the lexical and structural information in the code. Moreno et al. [19] conceived a technique to automatically generate human readable summaries for Java classes relying on class and method stereotypes in conjunction with ad-hoc heuristics. Other approaches generate summaries from source code artifacts, such as code fragments [38] or code usage examples [20]. Moreover, Panichella et al. [23] studied the impact of test case summaries on the number of fixed bugs, proposing an approach that automatically generates summaries of the portion of code exercised by each individual test. Other researchers focused on the summarization of build reports [25] or user reviews [30]. Our approach complements these approaches and presents a novel summarization approach for another important software development artifact, i.e., the build log.

Mining Q&A Sites Question and answer websites like StackOverflow have been analyzed by several researchers to provide developers with helpful data during software development. Ponzanelli et al. [24] enhance the IDE with a Prompter that automatically captures the code context in the IDE to propose related StackOverflow discussions. Bart is very similar to this work, it is integrated into the build server and acquires contextual information about failing builds to assist developers with deriving a fix. Other researchers, investigated the impact of searching for answers on StackOverflow on development workflow. Vasilescu et al. [34] analyzed the interplay between StackOverflow activities and code changes on GitHub. While a switch to StackOverflow interrupts the coding, they were able to show a correlation between visits of StackOverflow and code changes. Developers seem to frequently switch between their IDE and StackOverflow when they get stuck, which supports our assumptions of Section 2. Finally, Wong et al. [37] generated summaries for Java classes by mining source code descriptions on StackOverflow. We also extract information from StackOverflow, but follow a different goal, i.e., providing hints for build fixes.

8 Summary

This paper presented Bart, a system that supports developers in understanding build failures and effectively fixing them. Bart works on the build log, summarizes build failures, and provides solution hints using data from StackOverflow. We conducted an empirical study with eight developers to assess the effect of Bart on repairing build breaks. Our results show that developers find Bart useful to understand build breaks and that using Bart substantially reduces the time to fix a build break, on average by 41%.

Acknowledgments

The work presented in this paper is the result of the Bachelor's thesis of one of the authors [40]. We would like to thank all the study participants. C. Vassallo and H. Gall acknowledge the support of the Swiss National Science Foundation for their project SURF-MobileAppsData (SNF Project No. 200021-166275). T. Zemp acknowledges the student sponsoring support by CHOOSE, the Swiss Group for Software Engineering.

References

[1] Log parser plugin. https://wiki.jenkins.io/display/JENKINS/Log+Parser+Plugin. Accessed: 2018-02-08.
[2] Active JPA: A Simple Active Record Pattern Library in Java that Makes Programming DAL Easier. https://github.com/ActiveJpa/activejpa/. Accessed: 2018-02-08.
[3] G. Bavota, C. Gravino, R. Oliveto, A. De Lucia, G. Tortora, M. Genero, and J. A. Cruz-Lemus. Identifying the weaknesses of uml class diagrams during data model comprehension. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 168–182, Berlin, Heidelberg, 2011. Springer-Verlag.
[4] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *International Conference on Mining Software Repositories*, 2017.
[5] Checkstyle. http://checkstyle.sourceforge.net. Accessed: 2018-02-08.
[6] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.
[7] Fongo: Faked Out In-Memory Mongo for Java. https://github.com/fakemongo/fongo/. Accessed: 2018-02-08.

[8] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *ICSE (2)*, 2010.

[9] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2017*, page To Appear, 2017.

[10] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. 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)*, pages 426–437, 2016.

[11] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.

[12] N. Kerzazi, F. Khomh, and B. Adams. Why do automated builds break? an empirical study. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–50. IEEE, 2014.

[13] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[14] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[15] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, page To appear, 2018.

[16] S. Maple. Java tools and technologies landscape report 2016. *ZeroTurnaround post*, 2016.

[17] Maven. http://maven.apache.org/. Accessed: 2018-02-08.

[18] A. Miller. A hundred days of continuous integration. In *Proceedings of the Agile 2008*, AGILE '08, pages 289–293, 2008.

[19] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *ICPC*, pages 23–32. IEEE Computer Society, 2013.

[20] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus. How can I use this method? In *ICSE (1)*, pages 880–890. IEEE Computer Society, 2015.

[21] L. Moreno and A. Marcus. Automatic software summarization: the state of the art. In *ICSE (Companion Volume)*, pages 511–512. IEEE Computer Society, 2017.

[22] G. J. Myers. *The art of software testing (2. ed.).* Wiley, 2004.

[23] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: an empirical investigation. In *ICSE*, pages 547–558. ACM, 2016.

[24] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Mining StackOverflow To Turn The IDE Into A Self-Confident Programming Prompter. In *MSR*, 2014.

[25] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *ICSE (1)*, pages 505–514. ACM, 2010.

[26] T. Rausch, W. Hummer, P. Leitner, and S. 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'17, page nn, New York, NY, USA, 2017. ACM.

[27] N. B. Robbins and R. M. Heiberger. Plotting likert and other rating scales. In *Proceedings of the 2011 Joint Statistical Meeting*, pages 1058–1066, 2011.

[28] Sentry Java: A Sentry SDK for Java and other JVM languages. https://github.com/getsentry/sentry-java/. Accessed: 2018-02-08.

[29] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge. Programmers' build errors: a case study (at Google). In *Proc. Int'l Conf on Software Engineering (ICSE)*, pages 724–734, 2014.

[30] A. D. Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *SIGSOFT FSE*, pages 499–510. ACM, 2016.

[31] StackOverflow: "Maven". https://stackoverflow.com/questions/tagged/maven. Accessed: 2018-02-08.

[32] Travis-CI. https://travis-ci.org. Accessed: 2018-02-08.

[33] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807, New York, NY, USA, 2011. ACM.

[34] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *SocialCom*, pages 188–195. IEEE Computer Society, 2013.

[35] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella. A tale of ci build failures: an open source and a financial organization perspective.

[36] T. E. J. Vos, P. Tonella, W. Prasetya, P. M. Kruse, A. Bagnato, M. Harman, and O. Shehory. FITTEST: A new continuous and automated testing process for future internet applications. In *CSMR-WCRE*, pages 407–410. IEEE Computer Society, 2014.

[37] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *ASE*, pages 562–567. IEEE, 2013.

[38] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *ESEC/SIGSOFT FSE*, pages 655–658. ACM, 2013.

[39] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 334–344. IEEE Press, 2017.

[40] T. Zemp. BART Build Failure Summarization. *Bachelor Thesis, University of Zurich, Switzerland,* 2017.