# Programmers' Build Errors: A Case Study (at Google)

Hyunmin Seo
The Hong Kong University of Science and Technology
Hong Kong, China
hmseo@cse.ust.hk

Caitlin Sadowski
Google
Mountain View, CA, USA
supertri@google.com

Sebastian Elbaum
University of Nebraska
Lincoln, NE, USA
elbaum@cse.unl.edu

Edward Aftandilian
Google
Mountain View, CA, USA
eaftan@google.com

Robert Bowdidge
Google
Mountain View, CA, USA
bowdidge@google.com

## ABSTRACT

Building is an integral part of the software development process. However, little is known about the compiler errors that occur in this process. In this paper, we present an empirical study of 26.6 million builds produced during a period of nine months by thousands of developers. We describe the workflow through which those builds are generated, and we analyze failure frequency, compiler error types, and resolution efforts to fix those compiler errors. The results provide insights on how a large organization build process works, and pinpoints errors for which further developer support would be most effective.

## Categories and Subject Descriptors

D.2.8 [**Software**]: Software Engineering—*Metrics*; D.3.4 [**Programming Languages**]: Processors—*compilers*; H.1.2 [**Information Systems**]: User/Machine Systems—*Human factors*

## General Terms

Human Factors, Experimentation, Measurement

## Keywords

Software builds, build errors, empirical analysis

## 1. INTRODUCTION

Building is a central phase in the software development process in which developers use compilers, linkers, build files and scripts to assemble their code into executable units. This assembly process can be quite complex as it may involve multiple subsystems and programming languages and rely on evolving libraries and platforms. For example, one build we studied included 2858 build targets and 6 languages.



*ICSE* '14, May 31 - June 7, 2014, Hyderabad, India
ACM 978-1-4503-2756-5/14/06
http://dx.doi.org/10.1145/2568225.2568255.

The speed of a build directly affects programmer productivity. Programming is often described as an "edit-compile-debug" cycle in which a programmer makes a change, compiles, and tests the resulting binary, then repeats the cycle until the program behaves as expected. Slow compiles may cause the programmer to be distracted by other tasks or lose context, and reduce the number of changes that can be attempted during a day. Failed compiles, whether because of errors in the source code or incorrect references to other software components, force the programmer to step away from the larger task to track down an unexpected or unfamiliar error. Any delay widens the gap between the programmer deciding on the next change to perform and viewing the effect of that change. Keeping the build process fast and understanding when and how it fails is a key part of improving programmer productivity [12].

Yet we lack enough quantifiable evidence of how often the build process fails, why it fails, and how much effort is necessary to fix errors. As we discuss in Section 5, the related work in this area has focused on students building at a small scale (e.g., [4, 5, 10]) or on characterizing the complexity of the building process (e.g., [16, 13, 14]). In this paper, we present the findings of a case study of 26.6 million builds involving 18,000 developers over a nine month period. Specifically, we examine the builds done in C++ and Java, Google's most common languages, and the errors produced by the javac compiler (for the Java language) and the LLVM Clang compiler (for C++) by those builds. We report on the frequency of builds, the failure ratio, the errors reported by the Java and C++ compilers, and the time to fix these errors.

Our results reveal that C++ developers build 10.1 times a day and Java developers build 6.98 times a day when averaged over months. 37.4% and 29.7%, respectively, of these builds fail. The most common errors are associated with dependencies between components; developers spend significant effort resolving these build dependency issues.

The contributions of this paper are:

- Characterization of the developer's build patterns within the development workflows of a large organization (Section 2). This characterization serves to provide a context to our study findings, but also to give some insights on the complexity of this process in large software development organizations.

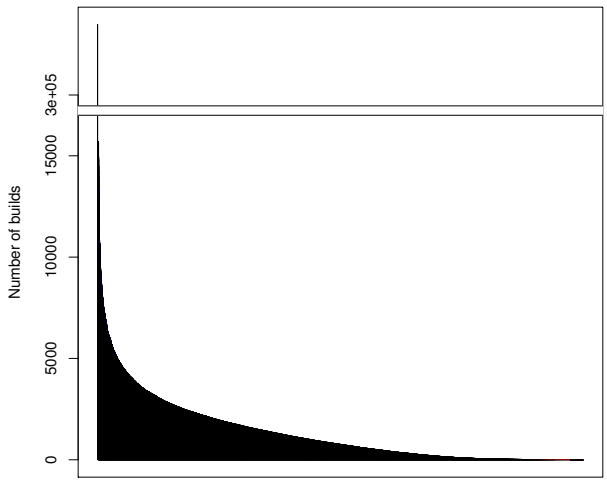



**Figure 1: Number of builds per developer. Each point on the x-axis represents a developer, in decreasing order by the number of builds.**

- Quantification, categorization, and analysis of build errors, their frequency, and their resolution based on 26.6 million builds (Section 3 and 4). The findings pinpoint particular areas where additional IDE features, build support, and analysis tools would provide the greatest benefit for developers.

# 2. GOOGLE'S BUILD ENVIRONMENT

Most Google developers employ an "edit-compile-test" cycle when working. This cycle occurs mostly within IDEs such as Eclipse [18] or IntelliJ IDEA [11] for Java (nine out of ten developers), while C++ developers typically use a text editor such as Emacs [19] or Vim [1] (eight out of ten developers). Source control, building, and testing tools are standardized through much of the company. To keep up with the large and quickly-evolving code base and the high levels of reuse, developers use a cloud-based build system. The build system allows programmers to build from the most recently committed sources ("head"), amortizing complications caused by the divergence of other components [8].

Google's cloud-based build process utilizes a proprietary build language [8]. BUILD files specify the build configuration, declaring the list of targets that need to build, the sources and dependent targets required to compile, and the compiler to use when building those targets. The primary action is a "build action" where a programmer requests one or more targets (a binary or a set of tests) to be built. The build system does all compiles and other actions necessary for each build action. Throughout this paper, a build represents a single request from a programmer which executes one or more compiles; a build only succeeds if all compiles succeed.

Most IDEs have an embedded compiler that can perform incremental compilation and unit testing, but may not exactly match the behavior of the compiler used by the build system (javac for Java, LLVM Clang for C++). Developers may build with the IDE's own compiler during development, but must ensure the code successfully builds with the centralized build system before committing.

Figure 1 provides a first characterization of developers' build frequency sorted by the number of builds. This dataset contains over 26.6 millions build attempts performed by over 18,000 developers during a period of nine months, and logged by the centralized build system. Section 3.2 will provide a more detailed account of the dataset and Section 4 will provide a breakdown of the data by factors such as the programming language and the failure rate. However, we already observe quite a variance in a long tail distribution, with a mean of 1446 and a median of 759 builds per developer over nine months.

Not all developers doing builds give us insights into behavior during the "edit-compile-debug" cycle and ways to improve compiler error messages. Developers with extremely high and low numbers of builds often are not doing "normal" development, and so their experiences do not characterize the behavior of the compiler errors faced by programmers during maintenance or enhancement.

For example, developers with the highest numbers of builds (on the left side of the graph) include *janitors*, *infrastructure developers*, *builders*, and *robots*. *Janitors* perform large-scale changes across the whole Google codebase as they migrate users to a newer API or fix all instances of a certain bug pattern using automated tools. Janitors trigger a large number of builds when testing their changes. *Infrastructure developers* work on critical tools such as compilers. Deploying new versions of infrastructure tools often requires testing that the new tools correctly handle the entire code base, so infrastructure developers often do huge numbers of successful builds. *Builders* include developers in charge of releasing, testing, or isolating failures. These tasks require a continous cycle of building and testing, often with a large number of failures. *Robots* are automated tools performing similar tasks to builders. Janitors, infrastructure developers, and robots make up approximately 0.5% of the developers in our build logs.

On the right side of the distribution in Figure 1, we find the *inactive developers* who do very few builds. Inactive developers may work in a programming language or on a project that is not part of this study, may only need to build tools from source as part of their job, or may not be actively coding (on leave, project manager, release engineer).

Between these extremes, the bulk of the developers belong to a group that we call the *standard developer*. The standard developer works primarily on a small number of projects, adding features or performing maintenance. This group constitutes the primary focus of the rest of this paper.

# 3. STUDY

In this section we define the research questions, describe the data collection process and metrics, and explain the threats to validity and their potential impact on our findings.

## 3.1 Research Questions

We explore three research questions:

- **RQ1. How often do builds fail?** We report on the failure-to-success build ratio across developers and languages to characterize the distribution of build failures.

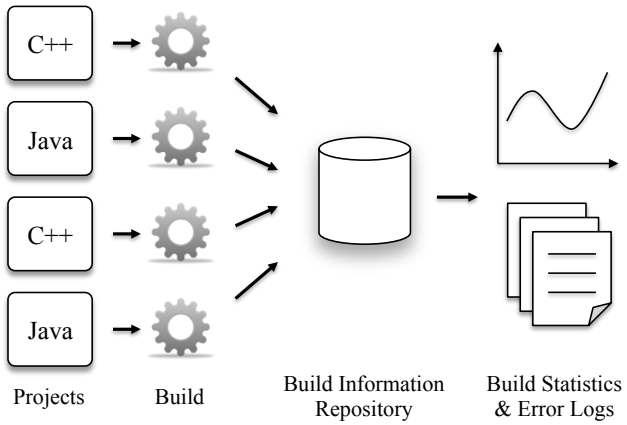



**Figure 2: Overview of data collection**

These findings help us understand the potential impact of changes in the development workflow.

- **RQ2. Why do builds fail?** We report on the frequency of the error kinds and taxonomize them to better understand what kinds of challenges are most commonly faced by the developers. These findings can help prioritize areas where development environments and program analysis tools may provide the most benefit.
- **RQ3. How long does it take to fix broken builds?** We report on the time spent by developers fixing each kind of compiler error, and describe particular instances of fixing efforts. These findings highlight which kinds of errors require the most effort to correct, and hint at the tool support and training that might help programmers fix troublesome classes of errors more easily.

## 3.2 Data Collection

The results of all builds in Google's cloud-based build system are saved in persistent logs describing the result of each build (succeeded or failed) and the errors produced by all compiles during the build. These logs are the main source of data for this study. We analyzed build logs to extract information about build sessions, success/failure ratios, and error messages. Figure 2 shows an overview of the logging pipeline. For this study, we examined log data from the build system for a period of nine months from November 2012 through July 2013. This data represented the compilation work necessary to build C++ and Java binaries that would run on Google's own servers. The data did not include projects built by other build systems within Google (generally client software for Mac, Windows, or mobile, or outside software such as javac, gcc, or the Linux kernel).

Because we wanted to distinguish compiles by *standard developers* from non-development builds in our build frequency and failure rate data, we defined rules to distinguish between users performing automatic and user-requested compiles. We recognized *janitors*, *robots* and *infrastructure developers* either by the specific build commands they used, or by detecting users performing more than 50 builds per day over the entire nine months. We also classified *non-active developers* as any user performing less than one build per day. Across standard developers, each month Java programmers performed an average of 140 builds (median: 101 builds) and C++ programmers performed an average of 202 builds (median: 147 builds).

In addition to partitioning users based on the logs, we also routinely performed sample checking throughout the data collection process to validate our categorizations and thresholds. We talked with specific developers to validate how we classified them, and checked with the engineers maintaining internal tools to validate our assumptions.

As mentioned previously, the data collected includes 26.6 million build invocation commands performed by 18,000 unique developers. 9.6M of the build invocations included at least one Java target, and 17.7M builds included at least one C++ target. 7% of the Java builds and 3.9% of the C++ builds included both Java and C++ targets.

We analyzed a total of 57.8M error messages captured in the logs. The errors logged are those generated by the compilers. Errors generated by other tools, such as the C++ linker, are not currently logged, so some problems, e.g. omitting a dependency required at link time, could not be detected. However, other dependency problems could be inferred. For example, if the developer has forgotten to list a dependency in her BUILD file, sometimes the compilation will fail with a "`undefined symbol`" error.

We built a parser to map each error message to its *kind*. Compiler configuration files define the error kinds and their format string. For example, Clang, the LLVM compiler frontend, defines an error kind `member_decl_does_not_match` and its format string "`out-of-line definition of %0 does not match any declaration in %1`". The actual error messages are generated by replacing the placeholders in the format string with identifiers. We used the parser and mapped each logged error message to its kind.

## 3.3 Metrics

We computed the following metrics from the log files:

- **Number of builds.** We count the number of build commands performed by each developer.
- **Build failure ratio.** We count the number of failed builds and divide it by the total number of build attempts by the developer.
- **Build errors.** We count the number of errors for each error kind. We investigated two strategies for counting the number of errors: *count-all* and *count-distinct*. The count-all strategy considers all error messages, even if one build has many of the same kind of errors. The count-distinct strategy counts the number of different kinds of errors in a build. For example, when a build fails with three `doesnt.exist` errors, ten `cant.resolve` errors and two `expected` errors, count-distinct would add 3 errors to the histogram (one for each kind) and count-all would add 15 (one for each individual error). Counting each error once is straightforward, but we noticed that unfair weight was given to certain error kinds when one fault generates multiple errors. For example, missing an import statement may generate a large number of name resolution errors, and counting each error once would result in over-representation of the name resolution error kind. We decided instead to count each error kind only once on each build failure.

- **Resolution time.** For each error kind, we measure the elapsed time a developer spends to fix the error. Figure 3 shows an example of how we measure resolution time. In this example, a developer has two failing builds before the build succeeds. We measure the time from the completion of the first failing build to the beginning of the next successful build. In addition, we count the number of failing builds before succeeding. In this example the build attempt count is two. In computing this metric we only considered build failures containing one error, and that were corrected within twelve hours of their occurrence to avoid compounding resolution time with the developers' schedule.

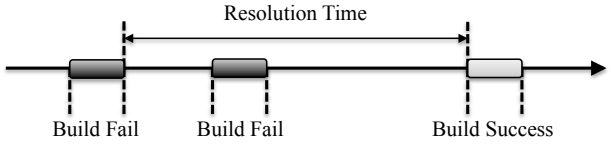



**Figure 3: Resolution time**

## 3.4 Threats To Validity

This study is performed within a single company with particular processes, constraints, resources, and tools. This constitutes a threat to the generalizability of the results. Still, the magnitude of the study in terms of the number of builds, developers, and systems involved provides a valuable baseline for the community, and the details provided characterizing the environment of the study should help to contextualize our findings.

We were able to capture data only from the centralized build system. We do not include errors encountered when a developer performs compiles with other compilers, such as using the incremental build feature built into IntelliJ or Eclipse. For C++, we also do not have data on linker errors encountered; we track only the compiler errors received. So, even through the distribution of errors and resolution times we found are based on 56M error messages, they are still limited by the richness of the data source.

There are also many variations of the software development workflows we describe that may have made it into our dataset. We are trying to understand builds done as part of development, the relative proportion of developers doing tests, preparing releases, or experimenting with a new language could bias the data. To address this threat, we filtered out extreme users, such as code janitors or compiler developers, and sample-checked the dataset by manually inspecting the data having unusual value.

The collected data includes projects belonging to various domains, with different complexity, schedules, development contexts, and personnel. In this paper, we perform an analysis that does not attempt to deeply discriminate across those factors even though we recognize they may have affected our findings. Teasing those factors apart is part of our future research agenda.

Other threats to validity include the focus on only Java and C++, potential errors in our data gathering process, the choice to count each error kind once per build failure, our decisions in mapping errors to a taxonomy, and our choice of cutoffs to remove noise from our data. To mitigate these threats, we performed multiple checks on the data and on the implications of the decisions we made in the design of the study, and have documented them in the paper.

# 4. RESULTS

In this section we address each research question.

## 4.1 RQ1: How often do builds fail?

Figure 4 shows the distribution of build failure ratios across developers. This ratio is approximately normally distributed, with the median percentage of build failures higher for C++ (38.4%) than Java (28.5%). The differences across languages can be attributable, at least in part, to IDE usage as most JAVA developers benefit from the IDEs' built-in checks.

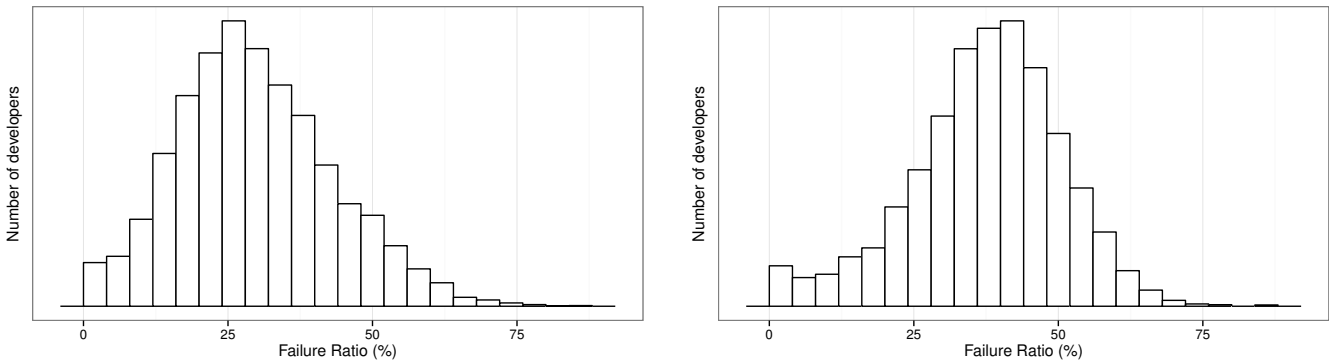
We further investigated developers with very high (more than 85%[1]) or very low (less than 5%) failure rates in order to understand what leads to such extreme outcomes. Developers with a low build failure rate had a median of 45 and 41 builds (Java and C++) and developers with a high failure rate had a median of 22 and 27 builds (Java and C++) per month. The first thing we notice is that both of these numbers are well below the rest of the developers who have a median of 101 and 147 builds per month for Java and C++. Developers with either very low or very high failure rate seem to be rare builders.

Developers with very low failure rates were not regular developers in the specific language or projects. Some developed in other languages, some worked on projects outside the build system, one worked on release tools, and others were in non-coding roles. They had rare or occasional builds without participating in active development, in some cases consisting of small conservative changes with no discernible effects, and this resulted in very low build failure rates.

Developers with very high failure rates were also typically not regular developers in the specific language. For example, we found a case where a prolific Java developer had done a small number of C++ janitorial changes that broke many unexpected targets which resulted in a very high failure ratio in C++. We did find, however, a small number of cases where prolific developers had high failure rates. These instances occurred when developers were working on infrastructure tools such as compilers that trigger the re-building of many other systems, and cause cascading build errors when faulty.

We explored other conjectures that, based on our experience, seemed likely but were not supported by the data. For example, we hypothesized that developers who build more frequently may have a higher failure ratio, but we found no strong correlation between the build counts and the build failure ratio. We also examined the correlation between the number of builds and build time. Although we hypothesized that developers with faster build times build more frequently, we found no strong correlation. We also hypothesized that experienced developers, across all developers that built software, would have a lower failure rate. We defined experienced developers as those who had invoked a build at least 1000 times in the past 9 months, and novice developers as those who had only performed a build within the past three months and had fewer than 200 builds. However, we did not see any correlation in this data. In part, this may be due to the difficulties in precisely characterizing experience or expertise in a setting where developers frequently move

[1]No standard developer had more than 90% fail ratio.




(a) Java (b) C++

**Figure 4: Build failures**

across projects and new developers with varied experience continually join the company, and a relationship between expertise and task complexity that we did not capture in this work.

## 4.2 RQ2: Why do builds fail?

To investigate why builds fail, we analyzed the compiler errors reported in the build log files. Each error message in the dataset was mapped to an error message kind by using the compiler configuration files used by javac and Clang. For example, we mapped the Clang error message "`use of undeclared identifier %0`" to `undeclared_var_use`. We retained the variations of error messages used by the compilers. For example, `expected` is an error message of the form "`{0} expected`" ({0} for the first parameter) whereas `expected3` is an error message of the form "`{0}, {1}, or {2} expected`" (with 3 parameters). We decided to keep each error variation separate so as to identify any differences between them. Tables 2 and 3 provide the error message kinds we considered/found, a brief description of each one, and their category.

After this mapping was complete, we identified the 25 most frequent error message kinds for Java (Figure 5a) and C++ (Figure 5b). We used the count-distinct methodology described in Section 3.3 to avoid over-weighting error kinds in which one fault generates multiple errors.

For Java, the top five errors constitute approximately 80% of the errors reported, and the most common error kind, `cant.resolve`, appears in 43.25% of the errors seen. This error kind occurs when the compiler encounters a symbol it does not recognize, such as when there is a missing dependency or when a developer mistypes a name. See Table 4 for a sample of why these errors occur. Similarly, the `doesnt.exist` message occurs when a developer tries to import something that is not in the classpath. The `strict` errors are Google-specific and identify situations in which transitive dependencies should be added to the list of direct dependencies for a particular target [3]. `cant.apply.symbol` and `cant.apply.symbol.1` are both the result of a method called with an incorrect argument list. The majority of error kinds for Java are dependency-related (all `strict` and `doesnt.exist` errors and a significant fraction of `cant.resolve` errors); we believe the extensive use of IDEs for Java development contributes to this trend because many simpler errors are captured within the IDE.

For C++, the top error kind, `undeclared_var_use`, also means that the compiler could not recognize a symbol. Clang provides more specific error messages than Java depending on the context of the errors. For example, `undeclared_var_use` and `undeclared_var_use_suggest` are basically the same error except that Clang provides a suggestion after trying to find a similar name from the symbol table for the latter one. `no_member` is reported when the compiler knows that the name is located within a class but could not find it. Similarly, `unknown_typename_suggest` and `ovl_no_viable_function_in_call` are reported when the compiler finds an unknown typename or function calls. The majority of error kinds for C++ are, as with Java, dependency-related. On the other hand, we find that simple syntax errors such as `expected_rparen` occur more frequently than in Java.

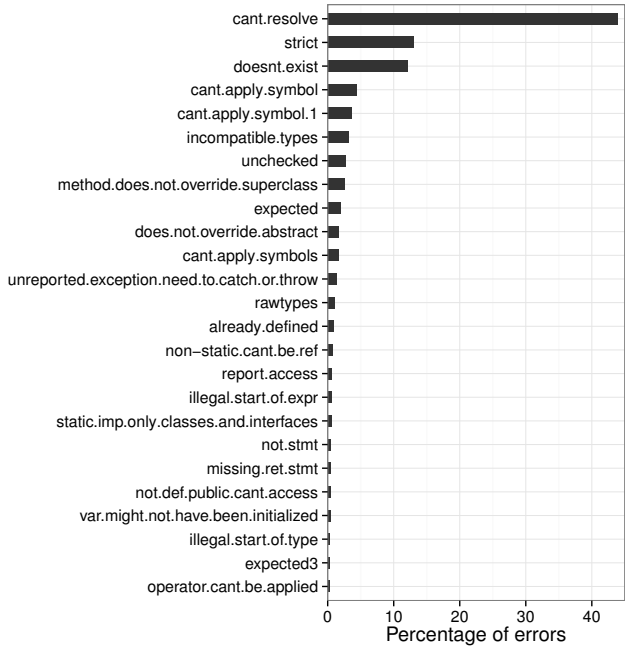
To better understand the errors occurring across languages, we taxonomized them. Table 1 summarizes the number of error kinds we used in forming this taxonomy, relative to the total number of error message kinds for each language, and the percentage of total error messages in our data set covered by the proposed taxonomy. For example, among 1284 different error kinds in C++, we used 102 in the taxonomy and they covered more than 88% of all the errors in C++.

**Table 1: Number of error kinds covered in taxonomy**

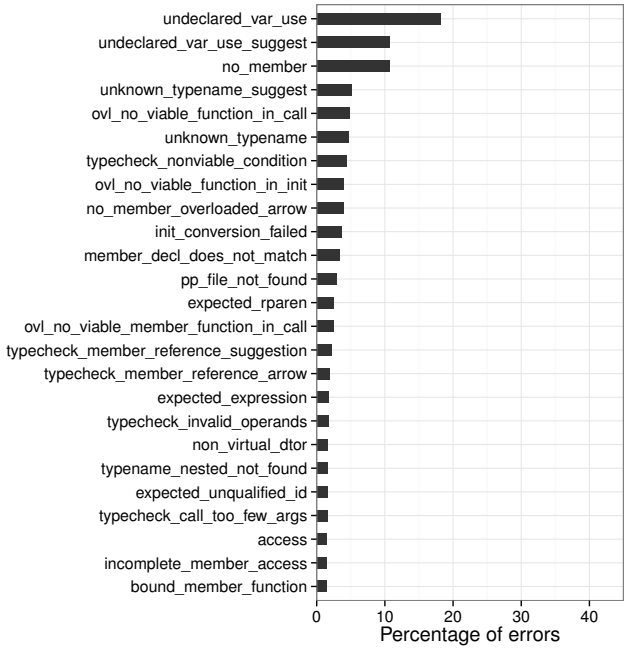
| | Java | C++ |
|---|---|---|
| # error kinds used in taxonomy | 25/230 | 102/1284 |
| % dataset messages covered | 91.96% | 88.17% |

We used open coding to initially categorize the message kinds. Open coding refers to a part of analysis that deals with the labelling and categorising the data in qualitative research [15, 7]. Once we had the initial categories, we enlisted experts from the Java and C++ compiler teams to decide which category was the best fit for each message. We put each error message kind into only one category. In total, we involved seven developers in this process. As a side benefit, in the process of coming up with this taxonomy, members of one of the compiler teams realized that some error messages were confusingly worded and updated the wording for two message kinds. Table 2 and 3 shows which category applies to all of the errors in Figure 5a and Figure 5b.

We classified error kinds into five categories. *Dependency* includes error kinds related to dependencies between source




(a) Java




(b) C++

**Figure 5: Reasons for build errors**

code components. For example, name resolution errors or missing file errors belong to this category. However, other errors, such as when produced by mistyping a variable name, also fall into this category as they have the same underlying compiler error kind. *Type mismatch* includes type errors. For example, errors occurring when a variable is assigned to an incompatible type or when a function is called with wrong argument types belong to this category. *Syntax* represents simple syntax errors such as omitting a parenthesis or includes errors that occur when the compiler expects an expression but is given something else. *Semantic* includes errors related to class access rule violations or not implementing abstract methods. We classified the remaining errors into *Other*. Undefined virtual destructors, redefined names, or uncaught exceptions belong to this category.

The percentage of error messages in each category is shown in Figure 6. Dependency-related errors are the most common error type for both C++ (52.68%) and Java (64.71%).[2] We also noticed that there are more syntax errors in our data set for C++; this is again consistent with the greater IDE usage for Java.

## 4.3 RQ3: How long does it take to fix builds?

To measure how long it takes to fix build errors, we collected resolution time data as described in Section 3.3. Resolution time measures the time interval between the completion of a first failed build and the start of the next successful build. Resolution time may not measure the time

[2]Note that we may classify mistyping a variable name as a dependency error when the compiler produces the same error message for both cases. However, even if we assume 3/4 of `cant.resolve` errors are the result of typos (as in the sample from Table 4) and remove them from this category, dependency errors are still the most common error category for Java.

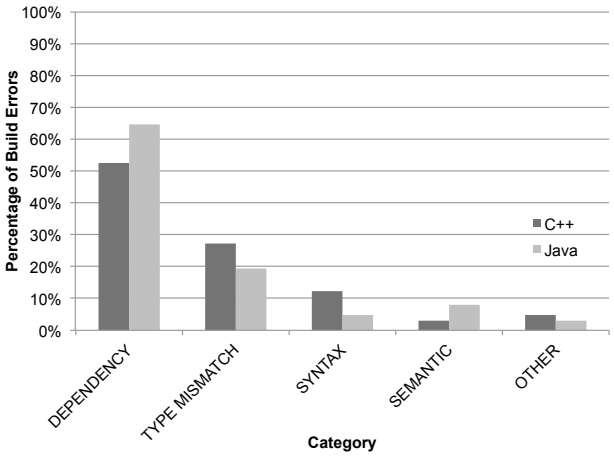



**Figure 6: Build error category (joins C++ and Java data)**

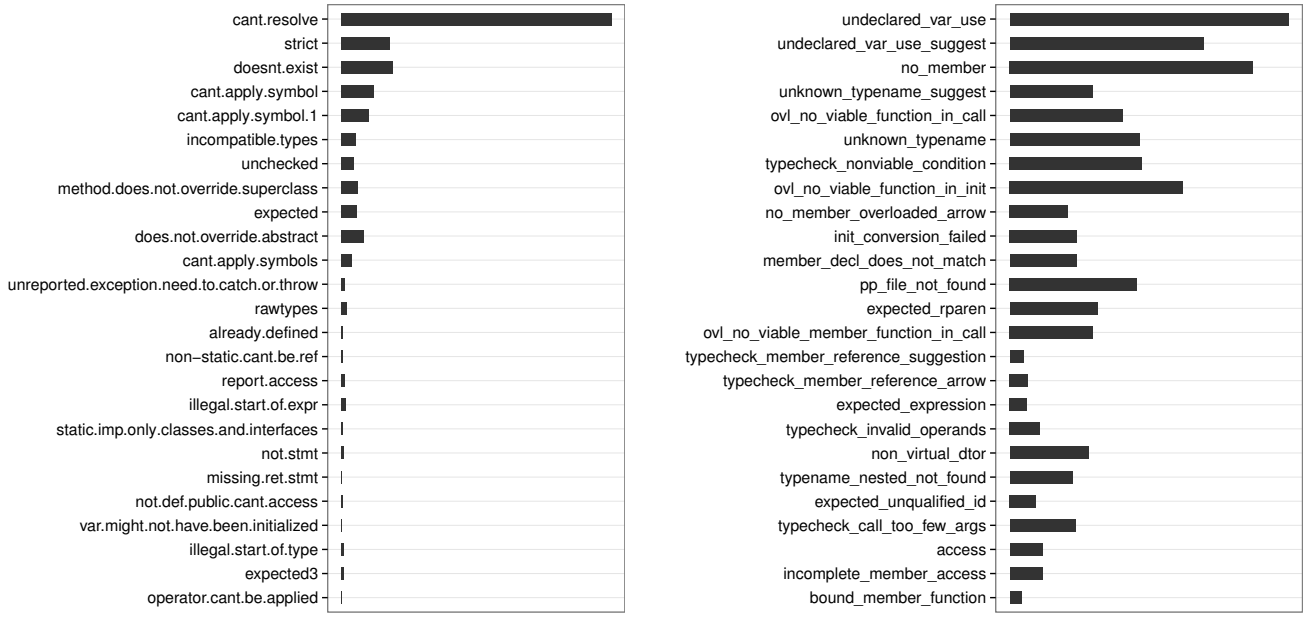
that the developer actually spent addressing the problem and waiting for compiles to complete; it is possible that the developer switched to a different task or was away from her desk during some error fixing sessions we logged. To remove extreme cases (e.g., the developer went home for the day), we omitted any builds in which the resolution time was greater than twelve hours. When multiple error kinds are resolved at once, it is not easy to discriminate resolution time per error kind. Dividing the total resolution time by the number of error kinds is one possible approach but this introduces further imprecision to the resolution time. Instead, we constrained our analysis to failing builds with one kind of error message. This filter retains 60% of the Java builds and 40% of the C++ builds, for a total of more than 10 million builds.

**Table 2: Java error messages taxonomy**

| Category | Name | Description |
|---|---|---|
| Dependency | cant.resolve | cannot find symbol |
| | doesnt.exist | package does not exist |
| | strict | Google-specific dependency check |
| Type mismatch | cant.apply.symbol | method called with incorrect argument list |
| | cant.apply.symbol.1 | method called with incorrect argument list |
| | incompatible.types | incompatible types |
| | unchecked | unchecked conversion from rawtype to parameterized type |
| | cant.apply.symbols | method called with incorrect argument list |
| | rawtypes | use of raw type rather than parameterized type |
| | operator.cant.be.applied | wrong types for operator |
| Syntax | expected | certain keyword or symbol was expected and not found |
| | illegal.start.of.expr | illegal start of expression |
| | not.stmt | statement expected but not found |
| | missing.ret.stmt | missing return |
| | illegal.start.of.type | illegal start of type |
| | expected3 | certain keyword or symbol was expected and not found |
| Semantic | method.does.not.override.superclass | method has @Override annotation but does not override a superclass or interface method |
| | does.not.override.abstract | concrete class must override abstract method in superclass or interface |
| | non-static.cant.be.ref | non-static cannot be referenced from static context |
| | report.access | access disallowed by visibility modifier |
| | static.imp.only.classes.and.interfaces | cannot static import this symbol |
| | not.def.public.cant.access | symbol is not public and cannot be accessed |
| Other | unreported.exception.need.to.catch.or .throw | checked exception must be caught or thrown |
| | already.defined | symbol already defined |
| | var.might.not.have.been.initialized | variable might not have been initialized |

**Table 3: C++ error messages taxonomy**

| Category | Name | Description |
|---|---|---|
| Dependency | undeclared_var_use | use of an undeclared identifier |
| | no_member | no member (e.g. class variable) in the given class |
| | undeclared_var_use_suggest | use of an undeclared identifier, with a suggested fix |
| | unknown_typename_suggest | unknown typename, with a suggested fix |
| | unknown_typename | unknown typename |
| | no_member_overloaded_arrow | no member (e.g. class variable) in the given class, with a suggested fix |
| | member_decl_does_not_match | out of line definition does not match any declaration |
| | pp_file_not_found | could not find header file |
| | typename_nested_not_found | no type with given name within the specified class or namespace |
| | incomplete_member_access | member access into incomplete type |
| Type mismatch | ovl_no_viable_function_in_call | calling an undeclared version of an overloaded function |
| | typecheck_nonviable_condition | cannot convert |
| | ovl_no_viable_function_in_init | no matching constructor for the initialization expression |
| | init_conversion_failed | cannot initialize with the passed-in value |
| | ovl_no_viable_member_function_in_call | no matching member function for the call |
| | typecheck_member_reference_suggestion | using '->' instead of '.' or vice-versa, with a suggested fix |
| | typecheck_member_reference_arrow | using '->' instead of '.' or vice-versa |
| | typecheck_invalid_operands | invalid operands to binary expression |
| | typecheck_call_too_few_args | too few arguments to a function call |
| Syntax | expected_expression | expected an expression |
| | expected_rparen | expected right parenthesis |
| | expected_unqualified_id | expected an identifier |
| | bound_member_function | reference to a non-static member function |
| Semantic | access | accessing a protected or private member |
| Other | non_virtual_dtor | virtual functions but no virtual destructor |




(a) Java (b) C++

**Figure 7: Cost per error. Cost is computed based on the frequency times the median resolution time.**

Figure 8a and Figure 8b show resolution time of the top 25 error kinds for C++ and Java. There is a box per error type, where the box is bounded by the 25 and 75 percentiles, the line within a box is the median. Overall, the median resolution time of build errors were 5 and 12 minutes for C++ and Java, respectively, and can vary by an order of magnitude across error kinds.[3]

In Figure 8a, we observe that some errors, like `missing.ret.stmt`, are much faster to fix than others, like `does.not.override.abstract`.[4] The longer resolution times for certain errors make sense; `does.not.override.abstract` will likely require the programmer to implement additional methods and think about design issues. More interesting is `rawtypes`, introduced in Java 7, which is seen in 2% of the errors but has a median resolution time of close to 5 minutes. This insight combined with conversations with the developers about the value of this message in the identification or fixing of bugs led to the disabling of this error. The sample size for `static.imp.only.classes.and.interfaces` was very small (less than 10).

Figure 8b shows the same data for C++. Overall, C++ resolution time is less than Java. However, some compiler errors show a higher median resolution time as they are more difficult to puzzle out (`non_virtual_dtor`, `ovl_no_viable_function_in_init`, `typecheck_nonviable_condition`, or `typename_nested_not_found`), or else may take time to track down the missing dependency (`pp_file_not_found`). `expected_rparen` and `typecheck_call_too_few_args` show very large distributions and `incomplete_member_access` shows a very tight distribution. We inspected the data and found that sample sizes for these error kinds are much smaller than the other error kinds (less than 110, as compared with thousands or tens of thousands); this skews the distributions.

We also investigated the cost of fixing a particular error message. In Figures 7a and 7b, we multiply the median resolution time to fix a particular error kind by the number of errors of that kind. The errors are ordered in decreasing frequency. For Java, this closely matches the frequency order of error kinds, but some C++ errors (such as `ovl_no_viable_function_in_init` and `pp_file_not_found`) have a significantly higher cost.

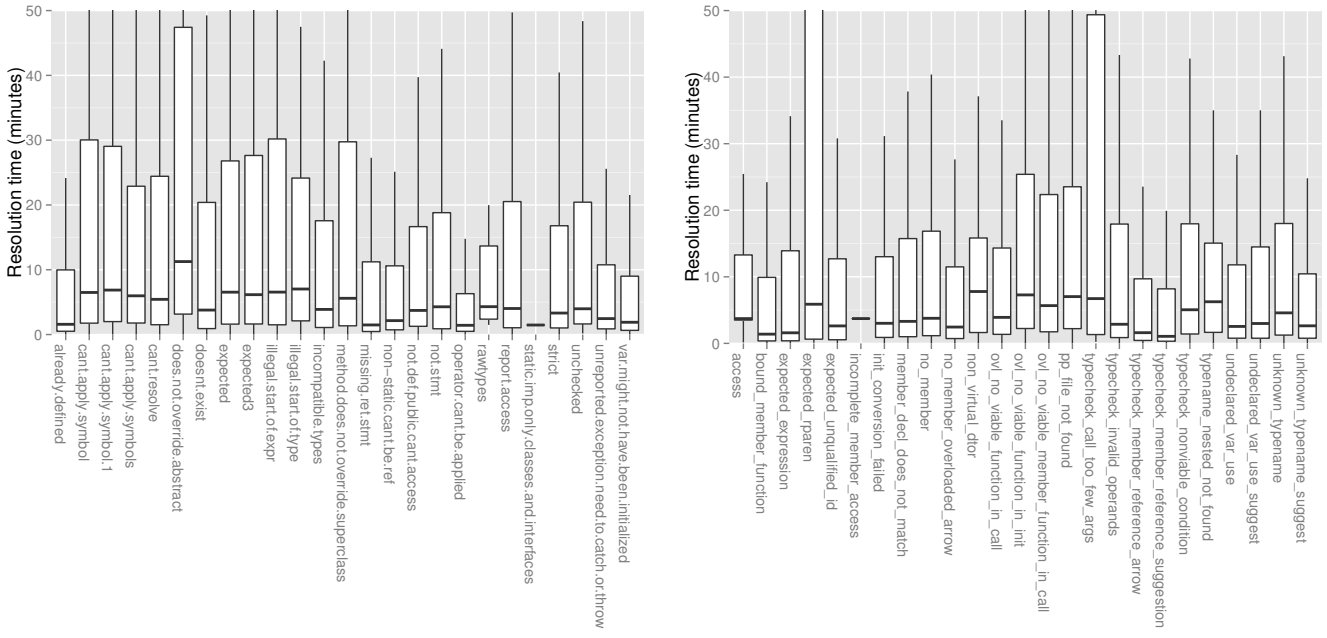
The median number of build attempts until the errors are resolved was 1. 75% of build errors were resolved within at most two builds for all of the 25 most common error kinds for both Java and C++. For Java targets, 75% of build errors were resolved by the next build in 18 of the 25 most common error kinds. For the remaining seven error kinds (`cant.apply.symbol`, `cant.apply.symbol1`, `cant.apply.symbols`, `cant.resolve`, `strict`, `unchecked` and `does.not.override.abstract`), 75% of build errors were resolved within two builds.

For C++ targets, 75% of build errors were resolved by the next build in 20 error types. In the remaining 5 error types (`ovl_no_viable_function_in_call`, `ovl_no_viable_function_in_init`, `pp_file_not_found`, `ovl_no_viable_member_function_in_call`, and `typecheck_nonviable_condition`), 75% of build errors were resolved within two builds.

To better understand the effort involved in fixing dependency errors, we looked at the dataset in more detail with two case studies. First, we randomly selected 25 Java build failures having a `cant.resolve` error kind that were resolved by the following build. We acquired the source code and

[3]As is not surprising, the difference between some error kinds is significant, with a p-value $< .001$ returned by the Kruskal-Wallis rank sum test.

[4]Wilcoxon rank sum test reports a p-value $< .001$ and the Cliff's delta effect size is -.043 (medium).




(a) Java (b) C++

**Figure 8: Resolution time of different error types**

BUILD file at both the failed build and succeeded build and compared them to see how the errors were fixed. We categorized the errors and fixes and present them in Table 4. Note that the total number of errors exceeds 25 because some errors fit into two categories. Approximately 1/4 of the errors were related to a missing dependency statement, but the most significant fraction of errors were the result of a misspelled identifier.

These simple fixes hint that many problems might be automatically fixed. In fact, the resolution time for `strict` errors was cut in half when the build tool started providing suggested fixes for `strict` errors.

Second, we investigated an extreme case in which a developer had five consecutive failed builds before finally resolving a `cant.resolve` error kind. The developer actually had multiple `cant.resolve` errors at first, and performed five builds over 10 minutes, removing one `cant.resolve` error each time, each by adding dependencies. In this case, the developer might have appreciated more guidance on how to fix all the errors at once, perhaps by an error message that more clearly describes which dependencies may be missing.

## 4.4 Findings and Implications

Our analysis of over 26M builds shows that building is an activity central to software development, that exhibits variation in terms of workflow and effectiveness depending on the context such as the programming language and the IDE used, but that is consistently performed on average multiple times a day by developers. An analysis of the errors found during the build process revealed that, independent of programming language, approximately 10 percent of the error types account for 90% of the build failures. Furthermore, dependency errors are the most common. In terms of the cost of fixing build errors, we found that on average it took one build iteration, and most errors are fixed in two build iterations.

Quantifying and understanding build patterns and practices has implications for practitioners, tool builders, and researchers. For practitioners, it provides a means to identify areas where additional expertise, tool usage, or development activity (e.g., reducing dependencies) may be most beneficial. In the context of our organization it served to highlight the values of available infrastructure to address some of the common errors.

For tool builders, the analysis on the frequency of build errors and resolution time serves to identify areas where developers can get the most benefit by assistive tools. In fact, our study clearly shows that better tools to resolve dependency errors have the greatest potential payoff, and the infrastructure team in our organization is already allocating their efforts based on this outcome. Similarly, the quantification by error message and type served for the compiler team to identify error messages that, based on their failure rates relative to other messages, need to be revisited as they may be confusing developers.

For researchers, this work contributes in establishing the importance of building in the overall development process, and in quantifying the cost of specific resolution times. Further, this work highlights the need for techniques that can pro-actively analyze a combination of source code and build files to reduce the introduction of build errors, and support the fix of such errors when they occur.

# 5. RELATED WORK

This study is novel for characterizing how programmers in industry interact with their compiler and build tools. Previous studies either examined student programmers or studied the behavior of industrial programmers using other

**Table 4: Errors and user fixes for random sample of 25 `cant.resolve` compiler errors**

| Count | Error | Fix |
|---|---|---|
| 10 | Misspelled identifier | Fix spelling |
| 5 | Wrong number of args to constructor call | Add or remove arguments |
| 4 | Missing import | Add import |
| 2 | Missing dependency | Add dependency to BUILD file |
| 2 | Incorrect type parameter in arg to method | Fix type parameter |
| 1 | Called a non-existent method | Removed method call |
| 1 | Accessed a non-existent field | Added field |
| 1 | Removed a class but didn't remove all uses | Removed remaining uses of class |

metrics. Jadud [9][10], for example, used an instrumented IDE to record how often undergraduate programmers edited and recompiled their code. Jadud's analysis differed from ours by reasoning about individual programming sessions rather than over months. Jadud found programmers had short intervals between compiles, usually compiling within seconds after receiving a syntax error, but with minutes between compiles after a successful compile. Our data on time to resolve errors suggests longer delays between compiles, especially when fixing dependency-related issues. Students encountered different Java errors in Jadud's study, slightly resembling our C++ developers with syntax problems and unknown variables (25% of errors) being the most common errors. Jadud also found that 44% of failed compiles were followed by another failed compile, compared with our observation that most classes of errors were resolved in one compile 75% of the time. The difference is most likely the result of the fact that we only consider the resolution time for compiles with one error. Denny et. al [4] also logged compile errors encountered by student programmers coding in Java. They found that 48% of compiles failed; `cannot resolve identifier` was the most common error, followed by `type mismatch`. Like us, their students were using a standard Java compiler.

Speeding compilation and removing errors is often assumed to affect programmer productivity. Dyke not only studied the frequency of compilation and compiler errors, but tested for correlation with productivity (as measured by final grade) [5]. Number of runs (both debug and normal) did correlate with grade, suggesting that the more successful compiles and more runs completed are related to better productivity.

Less is known about the behavior of programmers in industry, and most studies focus on higher-level tasks rather than specifically on compiles [6]. Singer et al. [16] observed programmer work at a large company, via logging tool use as well as direct observation. Compilers were the most frequent tool used, accounting for 40% of all actions performed by the programmers; they similarly found testing and verification runs complicated analysis, but they provided few details about the compiles performed.

Other studies have looked at the behavior of builds and releases in large systems, though none have directly studied the cost of build-related problems for programmers during a programming session. McIntosh et al. explored the complexity of the build systems for several large open-source projects, and measured cost of complex build systems to developers [13]. Neitsch et al. explored the complexity and fragility of typical build specifications by noting the problems building source distributions of various open-source projects [14]. Both papers highlight the complexity of large systems, and hint at why certain errors caused by build configurations might require multiple attempts to correct.

Other studies are more focused on showing the impact of tools meant to support build file maintenance and evolution. For example, Adams et al. [2], designed and implemented a tool to reverse engineer build files to support system evolution. The application of this tool to two large code bases reflects the complexity involved in large, industrial build processes. The recent work of Tamrawi et al. [17] brings symbolic analysis to operate on build files to derive dependency graphs. These graphs are then used to support error and smell detection in build files; studies show their potential on open code bases and in an assessment including eight graduate students. Our study highlights why such tools are important by quantifying the time programmers spend trying to fix compile errors that could be repaired automatically. Our results on errors causing the most difficulty suggest additional tools and areas of analysis to help programmers spend less time fixing compile errors.

# 6. CONCLUSIONS AND FUTURE WORK

In this work we presented what we believe is the largest study about the errors developers encounter when building their code. We quantified, categorized, and analyzed build errors, their frequency, and their resolution based on 26.6 million builds. In spite of its magnitude, like any empirical study, this work has limitations. It reflects the setting of one software development organization, the primary source of data is global build system logs, it does not discriminate based on project or contextual attributes, and the developer and error counting and classification schemes were defined carefully but still arbitrarily. Our future work involves deeper analysis on dependency-related errors, finer measures on resolution time and studying fix patterns for automatic fix. Still, the insights derived from this study can be used as a baseline to better understand how developers build and to guide future process refinements and tool development to improve developer productivity.

# 7. ACKNOWLEDGEMENTS

Hyunmin Seo was an intern and Sebastian Elbaum was a visiting scientist at Google when this work was performed. We are thankful to the Google engineers that helped us, especially Ambrose Feinstein, Matt Beaumont-Gay, Nick Lewycky, and Jaeheon Yi. Sebastian Elbaum is being supported in part by NSF #1218265.

# 8. REFERENCES


[1] Vim. Available at `http://www.vim.org/`, 2014.

[2] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter. Design recovery and maintenance of build systems. In *International Conference in Software Maintenance*, pages 114–123, 2007.

[3] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible compiler. In *Workshop on Source Code Analysis and Manipulation (SCAM)*, 2012.

[4] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Conference on Innovation and Technology in Computer Science Education*, 2012.

[5] G. Dyke. Which aspects of novice programmers' usage of an IDE predict learning outcomes. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 505–510, New York, NY, USA, 2011. ACM.

[6] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, and J. Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *Productivity and Performance in High-End Computing Workshop*, 2006.

[7] B. G. Glaser and A. L. Strauss. *The discovery of grounded theory: Strategies for qualitative research.* Transaction Books, 2009.

[8] Google. Build in the cloud: How the build system works. Available from `http://http://google-engtools.blogspot.com/2011/08`, 2011.

[9] M. C. Jadud. A first look at novice compilation behavior using BlueJ. *Computer Science Education*, 15:25–40, 2005.

[10] M. C. Jadud. Methods and tools for exploring novice compilation behavior. In *ICER '06: Proceedings of the 2006 International Workshop on Computing Education*, 2006.

[11] Jetbrains. IntelliJ IDEA. Available at `http://www.jetbrains.com/idea/`, 2014.

[12] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *CHI '95: Human Factors in Computing Systems*, 1995.

[13] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 141–150, New York, NY, USA, 2011. ACM.

[14] A. Neitsch, M. W. Godfrey, and K. Wong. Build system issues in multilanguage software. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 140–149, Washington, DC, USA, 2012. IEEE Computer Society.

[15] N. R. Pandit. The creation of theory: A recent application of the grounded theory method. *The qualitative report*, 2(4):1–14, 1996.

[16] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 21–. IBM Press, 1997.

[17] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. Nguyen. Build code analysis with symbolic evaluation. In *International Conference in Software Engineering*, pages 650–660, 2012.

[18] The Eclipse Foundation. The Eclipse Software Development Kit. Available at `http://www.eclipse.org/`, 2009.

[19] The Free Software Foundation. GNU Emacs. Available at `http://www.gnu.org/software/emacs/`, 2014.