2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation

Building Useful Program Analysis Tools Using an Extensible Java Compiler

Edward Aftandilian, Raluca Sauciuc
Google, Inc.
Mountain View, CA, USA
{*eaftan, ralucas*}*@google.com*

Siddharth Priya, Sundaresan Krishnan
Google, Inc.
Hyderabad, India
{*siddharth, sunkrish*}*@google.com*

***Abstract*—Large software companies need customized tools to manage their source code. These tools are often built in an ad-hoc fashion, using brittle technologies such as regular expressions and home-grown parsers. Changes in the language cause the tools to break. More importantly, these ad-hoc tools often do not support uncommon-but-valid code code patterns.**

We report our experiences building source-code analysis tools at Google on top of a third-party, open-source, extensible compiler. We describe three tools in use on our Java codebase. The first, Strict Java Dependencies, enforces our dependency policy in order to reduce JAR file sizes and testing load. The second, error-prone, adds new error checks to the compilation process and automates repair of those errors at a whole-codebase scale. The third, Thindex, reduces the indexing burden for a Java IDE so that it can support Google-sized projects.

***Keywords*-compilers; static analysis; bug finding; indexing; dependency checking.**

I. Introduction

Google's source code repository is monolithic, promoting reusability and allowing the same versioning, build and test infrastructure across all languages. A developer is free to use any library within the repository, thus minimizing the need to branch out or re-implement the same functionality since different projects can take advantage of common code. This high degree of coupling across millions of lines of code is mitigated by a distributed, incremental build system [1] and a similar continuous integration testing system [2]. Since development always happens on "head," it should come as no surprise that 20+ changes happen every minute, and 50% of the code changes every month [3].

Development at this speed and scale raises a new set of challenges, affecting not only Google but most large software companies who need customized tools to manage their source code. For example, a company may want to enforce style rules, check for library-specific error patterns, enforce dependency constraints between internal components, make whole-codebase changes, and so on. These are all instances where an IDE could have helped (if only the codebase were small enough!), but they are now handled by ad-hoc program analysis and source-to-source translation tools. Generally these tools use simple techniques such as regular expressions and homegrown parsers to accomplish a specific task, but they fail for several reasons. First, ad-hoc program analysis tools are often brittle and break on uncommon-but-valid code patterns. Second, simple ad-hoc tools don't provide sufficient information to perform many non-trivial analyses, including refactorings. Type and symbol information is especially useful, but amounts to writing a type-checker. Finally, more sophisticated program analysis tools are expensive to create and maintain, especially as the target language evolves.

In this paper, we present our experience building special-purpose tools on top of the piece of software in our toolchain that is in the best position to understand source code: the compiler. The compiler by definition must accept all valid code in a given language, and many valuable analyses can be performed using data structures that the compiler must compute anyway, such as the abstract syntax tree. And since the compiler must be run to generate target code, by piggybacking on existing compiler computations, we can minimize the performance and workflow impact of any additional analyses that we incorporate. In addition, the maintenance burden on the organization is reduced, since fewer ad-hoc tools must be maintained.

The other critical aspect when considering custom code analysis tools is their integration into the developer workflow. When the rate of changes is high, continuous integration is necessary to ensure that the product stays releasable. Detecting breakages and fixing them long after the offending code has been submitted will only slow down the release process and increase its cost. On the other hand, it is also expensive if the developer has to run and wait on tools outside the usual edit-compile-submit cycle.

By building on our production compiler, we are able to integrate all these special-purpose tools into the build system without incurring a performance penalty. We describe three tools in production use at Google that leverage the OpenJDK javac compiler to implement useful analyses. The first, Strict Java Dependencies, instruments the compiler to find unneeded dependencies and reduce build, test, and runtime costs. The second, error-prone, adds a framework to the compiler both to add additional errors that are triggered at compile time and to automate large-scale repair of such errors that already exist in our codebase. The third, Thindex, uses the compiler to reduce the source code indexing burden

978-0-7695-4783-1/12 $26.00 © 2012 IEEE
DOI 10.1109/SCAM.2012.28

on a Java IDE in use at Google.

Our contributions are as follows:

- We present our experience with reusing the javac compiler APIs for non-trivial program analyses and advocate for "opening up" production compilers into extensible frameworks for the benefit of large-scale development.
- We provide experimental evidence on how these compiler-based analyses fit into our development workflow and help us manage our huge Java codebase.

II. Extending OpenJDK's Java Compiler

JSR 199 [4], or the "Java Compiler API," first available in JDK 6, has been the first step in opening up the compilation pipeline for custom tools, by providing the ability to invoke the compiler via an API. Essentially all the ingredients were provided for compiling code on the fly and loading the generated classes. The writers of non-trivial static analyses, though, will find the public API offering an incomplete handle on the functionality of the compiler. For instance, there are hooks providing the abstract syntax trees after parsing or type attribution, but not after dataflow analysis, when the compiler has already computed def-use chains and control flow. While work is under way to address some of the problems (such as JSR 203 [5]), we will describe our extensions and the abstractions we believe should be exposed via a public API.

Initiatives similar to ours gave rise to Roslyn (Microsoft's complete retooling of the .Net compilers), Dehydra [6] and Pork [7] (Mozilla's GCC plugin for static analysis and refactoring engine) and the Clang frontend [8]. We show how little is needed to bring javac into a powerful toolchain as well. Note that we will provide only a brief overview and point the interested reader to the Javadoc.

The package `com.sun.tools.javac.main` provides the main entry point, the class `JavaCompiler`, as the means to construct a new compiler and run different compilation phases on a set of source files. All the phases of the compiler are exposed as public methods: parsing, annotation processing, type attribution, dataflow checks, lowering and code generation, etc. They operate on the abstract syntax trees via the `TreeScanner` visitor.

Each `JavaCompiler` instance uses a `Context` object as a repository for all the compiler's components. Each component is registered with a *key*, and the compiler looks up its components (such as the symbol table, the type attributor, the class reader, etc.) via keys. Extending any one of them amounts to subclassing and registering the new component with the parent's key. For instance, one can extend the `Log` class, used for emitting warnings and errors, and provide more visually appealing messages. Most of the compiler's functionality can be altered this way.

Another important abstraction of the JavaCompiler is the *file manager*, which also supports custom extensions via the `ForwardingJavaFileManager` wrapper. In a nutshell, the compiler sees all files (JARs, source and class files) as `JavaFileObject`s. The file manager handles the creation of these `JavaFileObject`s and is free to manipulate their contents, for example by constructing the sources on the fly or keeping the generated classes in memory. JSR 203 brings the ability to easily inject user-provided filesystems. Within our tools, we query the file manager to map Java types to their originating JAR files on the classpath.

The missing abstractions include a mechanism to register callbacks and perform custom processing after each phase, and custom command-line flags. This forces us to subclass the JavaCompiler and override the methods of interest, in order to insert hooks anywhere in the compilation pipeline. The `Context` allows us to pass extra information to the compiler after parsing the custom command-line flags, while the `Log` class makes it easy to emit custom warnings and errors as our analyses dictate. Our analyses are expressed as `TreeScanner`s, typically running after the dataflow pass, when all types have been resolved and the symbol table is complete. With this architecture, it is easy to construct plugins and "register" them with the `Context`, extending the functionality of the compiler.

III. Strict Java Dependencies

A. The Build System

Let us consider the simple example in Figure 1. A BUILD file contains the project specification, which in our case comprises three *targets*, two Java libraries and a test suite. For each target, we declare the source files and the direct dependencies, or *deps*, needed for compilation. Figure 2 shows the corresponding dependency graph, with gray nodes representing either source or generated files, and white nodes representing the build targets. During the build, an *action* is constructed for each target based on its type. In our example, for each `java_library` there will be a corresponding invocation of the javac compiler, with the classpath set to the *transitive closure* of all the dependent targets' outputs, which are their corresponding JAR files, and the output being another JAR file.

The dependency graph is topologically sorted, and independent actions will be scheduled in parallel and distributed across different build servers. This also implies that whenever a target is changed, everything in the *transitive closure* of the reverse graph will need to be rebuilt. Moreover, the continuous integration system runs all affected tests for each submitted change, so all test targets that depend directly or transitively on the modified target will have to be re-run. In our example, the unit tests in `testA` will be run when either `libA, libB, libC` or any of their dependencies change. Given the rate of changes and the size of our repository, one can understand the demands on the build and test infrastructure.

```
java_library(name='libA',
    srcs=['A.java'],
    deps=[':libB'])
java_library(name='libB',
    srcs=['B.java'],
    deps=['path/to/C:libC'])
java_test(name='testA',
    srcs=['UnitTestsForA.java'],
    deps=[':libA'])
```

Figure 1. An example BUILD file. The BUILD file defines two Java library targets, which correspond to output JAR files, and one Java test target, which defines a set of tests to run.

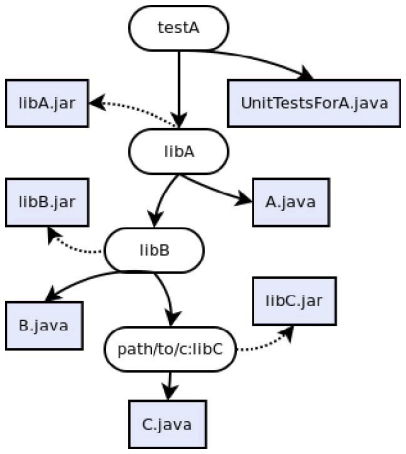


Figure 2. The dependency graph for the sample BUILD file in Figure 1.

Manually specifying dependencies simplifies the build process since there is no need for a dependency extraction phase, as the dependency graph is fully specified in BUILD files. For Java code, automatically extracting the dependencies requires the code to be fully parsed, which is the most expensive part of compilation. The disadvantage of declared dependencies is that over time they tend to drift away from the real dependencies, especially when the rate of change is high and developers keep adding dependencies until the code builds.

As a consequence, due to the transitive nature of dependencies, if a low-level core library is compiled with even a few dozen JARs on the classpath, client libraries and user-facing binaries end up with hundreds, if not thousands, of JARs as dependencies. Even with a highly parallel and distributed build, the critical path is impacted since the compiler has to wait for I/O while indexing through a huge classpath. These numbers are typical for any monolithic and finely grained project repository (for similar data on the C++ side, see [9], [10]).

Open-source projects using Maven [11] or similar Java build systems end up with fewer JARs on the classpath mainly because the boundaries between libraries are stable and code reuse is restricted by the narrow APIs that library developers choose to expose for a particular version. Dependency versioning is a problem, i.e. when trying to upgrade just a subset of dependencies to a newer version, conflicts can arise between these and older versions of the same dependencies being pulled in through other dependent libraries. Especially problematic are newer versions that offer slightly different functionality and APIs. Google avoids these versioning issues by building everything from HEAD, and the continuous integration system catches any incompatibilities early on through unit tests.

B. Overspecified and Underspecified Dependencies

Returning to the dependency graph of Figure 2, let us consider the case when `B.java` doesn't actually need `libC` to compile. The build system cannot detect this case, and the compiler will not complain if the classpath contains unneeded entries. However, the impact of these unneeded or *overspecified* dependencies is considerable.

First, the classpaths for all targets depending on `libB`, such as `libA`, will be larger, which translates into more I/O and longer compile times. Second, a change to `libC` will be detected by the continuous build system, triggering unnecessary recompilations and test runs for all dependent targets, such as `testA`. Third, the sizes of all dependent binaries will increase by having to include the unused classes from `libC`.

The related concept of an *underspecified* dependency can be illustrated if we assume `A.java` needs `libC` to compile. Apparently this is not a problem, since `libC.jar` will always be on the transitive classpath of `libA` because of `libB`. However, this dependency prevents the removal of the unneeded dependency from `libB` to `libC`, as doing so will break the build for `libA`.

Previous attempts at dealing with incorrectly specified dependencies included tools that parsed the Java code with regular expressions, looking at the import statements, or tools that relied on first building the target and then extracting dependencies from the generated bytecode. These tools were designed to be manually invoked after the build, at the developer's convenience. Since no enforcement mechanism was in place, the bad dependencies proliferated.

C. Compiler Hooks

We integrate the dependency checking as a plugin for our extensible compiler. The build tool has to pass additional information to the compiler, essentially specifying for each JAR on the classpath whether it was provided by a direct or an indirect (transitive) dependency. We use two new command line flags, `--direct-dependency` and `--indirect-dependency`, each linking a JAR on the classpath to its originating target. The compiler consumes these custom flags and builds two dependency maps to be passed to the dependency checking plugin.

The plugin registers two callbacks, one for the type attribution phase and one for the end of compilation. After the type attribution phase, we use a `TreeScanner` to walk the ASTs, and for each type literal encountered, we check whether the type was resolved from the classpath JARs by consulting the two maps. If the type was resolved from an indirect dependency, we emit a warning through the provided `Log` object, pointing to the dependency that should have been declared in the *deps* section. This effectively handles the *underspecified* dependencies. The other callback is invoked at the end of the compilation, when the symbol table is scanned and for all the types encountered we mark their corresponding JARs. Then, we emit a warning for every JAR that hasn't been marked, as it is a potential *overspecified* compile-time dependency.

D. Evaluation

The Strict Java Dependencies plugin incurs minimal overhead, as it only requires an extra pass over the ASTs and storing two additional HashMaps. Because most Java targets in our build system are small, we used the OpenJDK source itself as the benchmark and found a 1% CPU time increase.

Next, we ran a build of the whole repository and collected the warnings emitted by the compiler for each Java target. We built 200,000 Java targets (libraries, binaries or tests) declaring a total of 2,700,000 direct dependencies, and got warnings for 800,000. In total, 29% of the declared direct dependencies were marked as unneeded by the compiler. These included runtime dependencies, which we then heuristically filtered out, lowering the ratio to 10%. For each of these dependencies, we checked whether removing them would impact the declaring target, because in some cases they would be required by other dependencies in the transitive closure of the declaring target. 5% of the dependencies turned out to be truly overspecified dependencies, no longer referenced by any of the source files in their respective targets or dependent libraries. Quantifying the exact overhead the unneeded dependencies add to our build and test infrastructure is very hard due to the incremental nature of these systems and the caching layers. However, our experiments so far indicate a reduction of up to 5% in binary sizes, a metric that correlates with the first.

Currently we are in the process of pruning the overspecified dependencies from the codebase, and carefully adding all the underspecified (missing direct) dependencies. The latter step is necessary to prevent upstream projects from suddenly failing to build. Each target with correct dependencies is marked with a special attribute in the BUILD file. The Strict Java Dependencies plugin emits errors instead of warnings for these targets, effectively preventing incorrect dependencies from being checked in. When the codebase is clean, we will be able to turn these dependency checks into errors globally.

IV. ERROR-PRONE

A. Motivation

Any sufficiently large software system contains bugs. Many bugs are very simple—off-by-one errors, violations of API contracts, even spelling errors—and can be detected with a bug pattern detector like FindBugs [12]. Beyond static analysis, software engineering best practices such as code review and unit testing aim to prevent these bugs from being checked into production code. Yet these bugs persist even when all these techniques are used.

To address these bugs, we developed error-prone. error-prone builds on top of javac to perform FindBugs-style static checks on Java source code. error-prone presents its results as compiler errors, no different from the other errors that javac presents to users. The interface is familiar, results are presented early, while the developer is still working on the code, and most importantly, it prevents these errors from entering the codebase—code with these errors will not even compile. error-prone encodes a small set of error checks (currently six, with an goal of 15-20). The errors detected are serious and almost always errors; this avoids warning fatigue. Finally, error-prone can also be used to detect and repair defects at the scale of Google's whole codebase. error-prone checks include a "suggested fix," which encodes a potential repair for the problem detected. We run the checks over our whole codebase in parallel and apply fixes automatically, freeing programmers from manually fixing existing instances of a specific bug. At that point the error check can be turned on in the compiler, and the compiler then enforces that the bug can never recur, eradicating it from the codebase.

error-prone is extensible with new checks and configurable to include different sets of checks. The checking portion is open-source and freely available [13].

B. Using error-prone

In this section we describe the usage of error prone: how it is implemented, how errors are presented to the user, how new checks are written, and how our checks may be used for automatic repair of errors.

1) Implementation: We implemented error-prone as an additional compiler pass in javac, as described in Section II. For error-prone, we perform our error checking pass after the *flow* phase of the compiler, during which a dataflow analysis is computed to look for errors. At this point, all type attribution and symbol information is available for our analyses, and all errors detected by the standard javac have been reported.

Our current checks are very simple. They are intraprocedural and do not require dataflow analysis. Thus, we can implement them as a scanner over AST nodes.

We provide a set of matchers that can be used to write error-prone checks. The matchers are declarative, composable predicates on AST nodes and can be thought of as an

embedded domain-specific language. In Figure 3, we show an example of how these matchers can be composed to implement an error check. The current set of matchers is small; we expect the library to grow as we implement more checks and learn what code patterns users want to match on.

2) User Interface: error-prone analysis results are presented as compiler errors, so any detected errors will cause the compilation to fail. Thus, we have to select only checks with a low false-positive rate. On the other hand, this approach guarantees that instances of these bugs will be fixed, since otherwise the compilation will fail. In addition, the interface is familiar to developers, and our existing development tools do not have to be modified to present the results. Finally, since results are presented early in the development process, the bugs detected are less costly to fix.

In some corner cases, there may be a legitimate reason for an outlawed code pattern to appear. For example, there may be test code that checks that an exceptional condition is triggered on an error, or there may be compiler test cases that ensure the compiler does not crash on an erroneous code pattern. For these cases, we provide an escape valve: the SuppressWarnings annotation [14]. The user may annotate her code with the SuppressWarnings annotation, passing the name of the check to suppress as an argument, and error-prone will not run that error check on that node and its children.

3) Writing Checks: Checks are written as boolean predicates on javac AST nodes. The predicates may be written procedurally or by composing the declarative matchers mentioned in Section IV-B1. To show how an error-prone check is written, we describe our implementation of a check for a common error in uses of the open-source Google Core Libraries [15]. The library provides a utility method, `Preconditions.checkNotNull`, that ensures that a reference is not null. This method takes two arguments. The first is an Object that represents the reference that should not be null. The second is an Object that represents the message to display on error; this is usually a constant string. A common error is to swap the order of the arguments, passing the constant string as the reference to be checked for non-nullity. Since a constant can never be null, the check trivially passes.

The predicate code for this example is shown in Figure 3. The `matches` method takes a tree node of type `MethodInvocationTree` (javac's representation of method calls) and determines whether:

- It is a call to a static method (line 3).
- The fully qualified name of the method is `com.google.common.base.Preconditions.checkNotNull` (line 3).
- The first argument is a string literal (line 4).

Note that the method is written in a declarative style and does not require much understanding of the java AST. error-prone wraps that complexity in its matcher library.

C. Automatic Repair

error-prone checks provide a "suggested fix" for the detected problem. In the case where the problem is detected at compile time, the suggested fix is presented in the error message, as shown in Figure 4.

The suggested fix code for the Preconditions.checkNotNull checker is shown in Figure 5. Our library provides utility methods for deleting whole AST nodes, replacing or swapping nodes, reflowing overly-long lines, and adding or removing import statements when necessary.

We use the same facility to automatically repair code when making large-scale changes. We run error-prone over the whole Google Java codebase in parallel using FlumeJava [16], but instead of emitting errors, error-prone emits textual diffs containing the suggested fix. We can then apply the diffs to our codebase, scrubbing it of all instances of this error before turning on the check in the compiler. Thus we reduce the pain of fixing existing instances, and we avoid breaking existing builds when we turn on the error check.

Suggested fixes are not required to be semantics-preserving. In fact, in most cases the fix *should* change the behavior of the program. For example, in the Preconditions.checkNotNull example above, the semantics-preserving fix would be to simply remove the call to the checkNotNull method. However, the intent of the programmer was to check that the non-constant argument was non-null, so our suggested fix swaps the order of the arguments. We rely on our continuous testing infrastructure to reduce the likelihood of breakages, and we request human code reviews for riskier changes, in which the programmer's intent was unclear.

D. Evaluation

In this section, we discuss the results of applying error-prone checks to Google's Java codebase. We present results showing the number of bugs found by each check, and we discuss why these bugs persisted in our codebase despite good code health practices such as code reviews and unit testing.

1) Checks and Error Counts: Currently, we have implemented six error checks. Three are novel, two replicate FindBugs checks, and one replicates a javac lint warning. A full list and source code can be found on our website [13]. In Table I, we present the total number of instances of each error that we found and repaired in Google's codebase.

Considering that error-prone checks look for "obviously wrong" patterns, it is surprising that a non-trivial number of them existed in our codebase. Previous work has shown that even the best programmers make mistakes [12], so it is not surprising that several instances of even an obviously incorrect bug might exist in an extremely large codebase. However, it is interesting that these bugs persisted despite testing and code reviews. We examined the bugs found and

```
1  public boolean matches(MethodInvocationTree methodInvocationTree, VisitorState state) {
2      return allOf(
3          methodSelect(staticMethod("com.google.common.base.Preconditions", "checkNotNull")),
4          argument(0, kindIs(STRING_LITERAL, ExpressionTree.class)))
5          .matches(methodInvocationTree, state);
6    }
```

Figure 3. The matcher code for the error-prone Preconditions.checkNotNull checker. This method matches calls to the static method `Preconditions.checkNotNull` in which the first argument is a string literal.

```
PositiveCase1.java:9: [PreconditionsCheckNotNull] Literal passed as first argument to
Preconditions.checkNotNull()
  (see http://code.google.com/p/error-prone/wiki/PreconditionsCheckNotNull)
Did you mean 'Preconditions.checkNotNull(thing, "thing is null");'?
    Preconditions.checkNotNull("thing is null", thing);
```

Figure 4. An error-prone error message for the Preconditions.checkNotNull check. The error message suggests a possible fix.

```
1  public Refactor refactor(MethodInvocationTree methodInvocationTree, VisitorState state) {
2      List<? extends ExpressionTree> arguments = methodInvocationTree.getArguments();
3      ExpressionTree stringLiteralValue = arguments.get(0);
4      SuggestedFix fix = new SuggestedFix();
5      if (arguments.size() == 2) {
6        fix.swap(arguments.get(0), arguments.get(1));
7      } else {
8        fix.delete(state.getPath().getParentPath().getLeaf());
9      }
10     return new Refactor(stringLiteralValue, refactorMessage, fix);
11   }
```

Figure 5. The suggested fix code for the error-prone Preconditions.checkNotNull checker. If the method call has only one argument, it suggests removing the statement that contains the method call, as currently it is a no-op. If the method call has two arguments, it suggests swapping their order.

Check name	Errors found
Empty if statement	9
Exception created but not thrown	22
Objects.equal self comparison	26
Preconditions.checkNotNull wrong argument order	103
Preconditions.checkNotNull autoboxing	82
Self assignment	60

Table I
THE NUMBER OF ERRORS FOUND AND REPAIRED BY EACH ERROR-PRONE CHECK WHEN RUN OVER GOOGLE'S JAVA CODEBASE.

repaired by error-prone to understand better why these bugs were not fixed earlier.

- Many instances of these bugs are difficult to pick out visually. For example, in the following code sample, the error is that the parameter name is misspelled. Yet, at a glance, the code appears correct. Many of the self assignment errors we found fit this pattern.

```
public void setFoo(int foa) {
  this.foo = foo;
}
```

- In many cases, the error would only manifest in an exceptional condition, and the exceptional condition is never tested in the unit tests. For example, the `Preconditions.checkNotNull` method normally ensures that a parameter passed into a method is not null. The error found by error-prone would cause this check not to occur; however, this would only change the behavior of the program if the parameter really were null, which violates the contract of the method. If there are no tests that violate the contract, then the bug will not be detected.
- Some of these errors occur in unused code. In these cases, it is important either to repair the bug, in case someone later begins using the code, or to delete the unused code.
- Some of these bugs cause subtly incorrect results. For example, the Objects.equal self comparison bug pattern would cause an `equals` method to return true in a few cases when it should return false. These bugs are especially dangerous and hard to find by testing.
- A few instances of the "error" were intentional, used to test an exceptional condition or as an intentional no-op. We learned that we needed to provide an "escape valve" (the SuppressWarnings annotation) to support these legitimate uses.
- Some instances are in third-party code, which may not be as thoroughly tested and reviewed as our own code.

Another observation is that even though other tools in

use at Google detect some of these bugs (e.g., FindBugs detects self assignment and dead exceptions), they nonetheless persist in our codebase. Our experience with FindBugs has taught us several lessons. First, static analysis results must be presented early to have a good chance of being fixed. Second, static analysis tools must be integrated into our development process; most developers will not go out of their way to run a separate tool. These observations drove our choice to build error-prone on top of javac. Since javac is our production compiler, we can generate static analysis results at every compile and present them to the user before code is checked into the depot. In addition, because javac is already used to compile our Java code, the extra checks occur naturally as part of the development workflow. Finally, we have also observed that developers are unlikely to go through a list of bugs and fix them on their own. This motivated our decision to provide an automatic repair facility and repair the bugs ourselves before turning on a new compiler error. **The error-prone model of fixing all existing bug instances ourselves lets us quickly get to a point where checks can catch developers introducing these bugs into new and potentially critical code.**

2) Performance evaluation: Because error-prone runs as part of the build process, it must not add significant overhead to the compiler. We tested performance by compiling a large Java project using both an unmodified javac (version 1.6.0) and a javac that includes error-prone with all six checks turned on. We repeated the compilation five times and averaged the elapsed times and memory consumption values. We built the project on a single, unused machine and ensured that no output files were cached.

We found that error-prone incurs a 9.0% time overhead and a 0.95% memory overhead, which are tolerable for our build environment. We have not yet invested effort in optimizing error-prone, and it should be possible to improve its performance.

E. Related Work

error-prone builds on a long tradition of static analysis tools. It is most closely related to the additional bug checks provided by IDEs such as Eclipse [17] and IntelliJ [18]. These tools check for simple bug patterns and sometimes provide a suggested fix. error-prone differs primarily in its choice of compiler to build upon (IntelliJ inspections are built on their own internal representation, *PSI trees*, and Eclipse warnings are built on the ecj compiler). In addition, by design error-prone supports only errors and not warnings.

error-prone is also closely related to bug pattern detector tools like FindBugs [12] and PMD [19]. error-prone differs from these tools in that it is integrated into our production compiler and thus seamlessly integrates into our build system. In addition, error-prone detects only code patterns that are almost certainly errors; we intentionally do not support warnings.

More sophisticated static analysis tools such as Coverity [20], Klocwork [21], and Parasoft [22] can find more complex bugs than error-prone. However, they are slower and do not provide results quickly enough to be presented as part of the build process. error-prone analyses, because they are so simple, can be run at every compile.

Mozilla's Pork [7] is another example of a large-scale static analysis and refactoring platform. It is built on the elsa C++ parser, which "can parse most C++ in the wild" [23]. The elsa website describes their efforts to mimic the bugs of production compilers like gcc and MSVC; this is evidence that building on top of the production compiler itself is a better choice.

Mozilla's Dehydra [6] is a static analysis tool built on top of GCC. It allows the user to query a C++ codebase using consise JavaScripts, and because it builds on GCC, it is easy to integrate with projects that use GCC as their production compiler.

error-prone has been heavily influenced by the work of the Clang [8] team at Google. The Clang team has been adding new errors to the LLVM C++ compiler, and they have been using a parallel framework (Clang MapReduce [24]) to detect and repair errors in C++ code at a large scale.

error-prone development was also informed by previous experience with FindBugs at Google [25], [26]. We learned that developers were more likely to fix bugs if presented with analysis results early and as part of their normal workflow. We also learned the value of building on a piece of infrastructure, the compiler, that is already deeply integrated into our build system, rather than having to integrate a separate tool.

V. THINDEX

A. Large Codebases and IDEs

A monolithic code repository presents extra challenges for Integrated Development Environments (IDEs). An IDE offers the programmer many facilities for working with large amounts of code, including:

- Auto-completion of symbols such as variables and function and class names.
- "Go to definition," the ability to navigate through cross-references and jump to the source line defining the symbol of interest.
- Instant feedback on syntax and semantic issues, as code is continuously parsed and checked in the background.

To accomplish all the above, and more, the IDE first reads and processes all the code added to a project in order to build a database of symbols and abstract syntax trees of the source code. This is called "*indexing*." The IDE also needs to either periodically poll or set up a callback to identify changes to any of the files and re-run indexing when any of the files changes. Large amounts of code added to a project can result in several minutes spent in indexing, with a large volume of I/O incurring a noticeable slowdown of the environment.

To solve this problem, we demarcate a portion of the repository as the *writable client*, representing the set of packages the developer wishes to work on. While the writable client is sufficient for editing, it does not contain all information needed to compile and run the project. The code not included in the writable client but required for compilation and running comes via *dependencies*, in the form of JARs added to the IDE's classpath. These dependencies cannot be edited but contribute towards the global symbol database. For a given writable client, the dependencies are usually found by constructing the *transitive closure* of all the "deps" sections in the corresponding BUILD files (see Section III-A), starting from the packages in the writable client.

B. Compile-time Dependencies vs. Transitive Closure

The transitive closure previously described is usually an order of magnitude larger than what is required by the IDE to provide context-sensitive information or even to compile the code in the writable client. This stems from the fact that manually provided "deps" attributes tend to over-estimate the actual dependencies, augmented by the fact that the transitive closure brings in dependencies without distinguishing whether they are needed for the compilation of the targets in the writable client or their dependencies. We again rely on program analysis to refine the dependency set based on the actual source code, and distinguish between *accessible* and *compile-only* dependencies as follows. A type T (i.e. a Java class or interface) has:

- An **accessible dependency** on type S if the type S is present in the signature of a publicly visible method or field belonging to type T, or if type T extends, implements, or is parameterized by type S;
- A **compile-only dependency** on type S if the type S is present in the implementation of a method or a static initializer, or if it only appears in the signatures of private methods or fields.

Based on these definitions, we can express the compile-time dependencies for the source files in the writable client as the transitive closure over accessible dependencies, starting with the compile-only dependencies of the writable client. To illustrate with an example, consider the following code snippet, and assume class A is present in the writable client. If class A uses an object of type B, a compile-only dependency by our definition, then the IDE must be aware of B to enable usage of B. If B has a public method $getC()$, then all types in its signature are accessible dependencies by our definition. For instance the return type, C, may or may not be required based on whether A calls $B.getC()$. Therefore we must conservatively provide C: while editing A the developer will expect $getC()$ to be readily usable since she is already working with an object of type B. On the other hand, if $getC()$'s implementation creates an object of another type D (another compile-only dependency), then D is not required for compilation of A because it is not accessible from A.

```
public class A {
  public static void main(String[] args) {
    B b = new B();
    ...
  }
}

public class B {
  public C getC(E e,...) {
    D d = new D();
    ...
  }
}
```

We use our extensible compiler in conjunction with the distributed build system to compute these finer-grained dependencies. For each target built, an extra output artifact contains the (local) symbol cross-reference information. The cross-reference extraction phase is again implemented via a `TreeScanner` that classifies all types seen in the ASTs based on their location (class/method/field signature or code section) and their visibility (public/private/protected). Due to caching and distribution, what could have been a very expensive computation locally—the parsing and indexing of a transitive closure of classes—is broken into small tasks to be run on demand. These output artifacts are shared between users, such that low-level libraries and targets which do not change very frequently are scanned and their cross-references cached and reused across users.

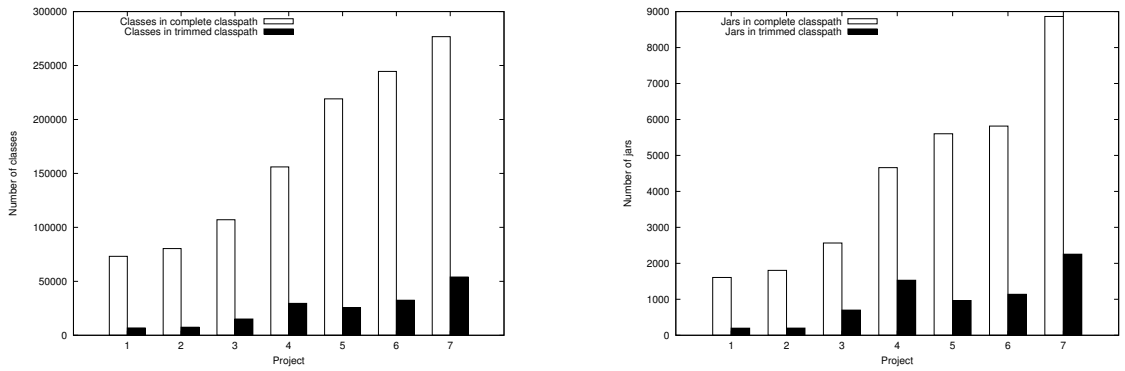
C. The Thindex Algorithm

The algorithm works by forming the union of compile-only dependencies of all files in the writable client. Then we recursively find the accessible dependencies of all the currently identified dependencies. The upper bound for this list of dependencies is the transitive closure. However, in practice the set of dependencies built using the algorithm is much smaller than the transitive closure, as shown in Section V-D.

Algorithm 1 Calculate the set of white listed files

$W \leftarrow \emptyset$
$Q \leftarrow \mathcal{T}$
while $Q \neq \emptyset$ **do**
  $e \leftarrow DEQUEUE(Q)$
  for all $d \in XREFS(e)$ **do**
    if $d \notin W$ **then**
      $ENQUEUE(Q, d)$
      $W \leftarrow \{d\} \cup W$
    end if
  end for
end while

The cross-reference information extracted by our compiler creates a directed graph over all Java types. The edges mark



(a) Classes (b) Jars

Figure 6. Comparison of the number of classes and JARs indexed with and without Thindex. The number of classes decreases by a mean of 85%, and the number of JARs decreases by a mean of 77%.

either accessible or compile-only dependencies, and let us denote, for a node T, the set of all accessible dependencies with $\mathcal{A}(T)$ and the set of all compile-only dependencies with $\mathcal{C}(T)$. The Thindex algorithm traverses the graph using breadth-first search using a queue Q, and collects nodes in the set W of white-listed source files, which are then added in full to the IDE's index.

$$\mathcal{T} \leftarrow \{\text{Types in Writable Client}\} \tag{1}$$

$$XREFS(t) = \begin{cases} \mathcal{C}(t), & \text{if } t \in \mathcal{T} \\ \mathcal{A}(t), & otherwise \end{cases} \tag{2}$$

D. Evaluation

We implemented the above algorithm in IntelliJ IDEA 10, using the symbol cross-reference data computed by our compiler plugin. We ran the algorithm on writable clients of varying sizes. The projects were created using both the standard and Thindex approaches and the time taken to index in each case was measured. Figures 6b and 6a show the reduction in dependency size we get by applying the Thindex algorithm. The number of classes indexed decreases by a mean of 85%, and the number of JARs indexed decreases by a mean of 77%. Figure 7 shows the reduction in indexing time, which decreases by a mean of 38%. The improvement is not proportional to the decrease in dependency size since there is a constant time overhead for indexing the JDK and committing the index to disk.

The above measurements do not include the time taken to identify and mark the dependencies. Since the identification is done as a part of the compilation step, this does not add any measurable overhead to the overall time.

E. Other Enhancements

One caveat with using the Thindex algorithm is that it only operates on symbols referenced by code in the writable client. For all compile-time and accessible dependencies, we keep their complete ASTs. But if the developer wants to use a new library from the repository and use code completion, she has to first add it to the dependencies and rebuild the project in order to refresh the index. This makes using new symbols difficult and negates some of the advantages, but can be solved through an extension called an "autocompletion provider." Essentially, we add a plugin that extends the module generating the list of auto-completions and provides a custom list of symbols, whenever auto-completion is triggered.

The use of an external auto-completion provider extends itself naturally to eliminate the restriction of using only the transitive closure to enabling the use of the entire codebase. We precompute the list of classes in the entire codebase and keep it cached, and we use the auto-completion provider to allow the developer to use any class from the entire code base. When the developer wants to reference a symbol outside the compile-time dependencies, the auto-completion provider will assist with the name. As soon as the type name appears as a literal in the source, we load the dependency in the background and make its index available after a short delay. We also add the dependency into the "deps" section of the appropriate BUILD file. This delay is not generally significant, and we effectively distribute the total cost of indexing over incremental, on-demand steps. The entire system working together frees the developer from having to make decisions about which dependencies must be defined and allows them to use any type defined in any part of the codebase while still ensuring that the IDE remains responsive.

VI. Conclusion

In this paper, we have described our experience building custom code analysis tools by extending a production compiler, applying them to a huge Java codebase, and integrating them into the development workflow at Google.

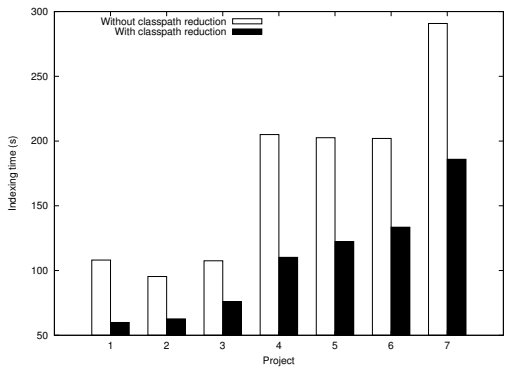


Figure 7. Comparison of indexing time with and without Thindex. Indexing time decreases by a mean of 38%.

We encourage compiler engineers to prioritize extensibility when designing future compilers, so that their work can be built upon by others.

ACKNOWLEDGMENTS

The authors would like to thank Alex Eagle for creating the error-prone project and providing guidance, Peter Epstein for proposing the idea for Thindex, and Chandler Carruth for his mentorship. We would also like to thank David Morgenthaler, Robert Bowdidge, and Jeremy Manson for their helpful comments on early drafts.

REFERENCES

[1] C. Kemper, "Build in the cloud: How the build system works. Google Engineering Tools." http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html, August 2011.

[2] P. Gupta, M. Ivey, and J. Penix, "Testing at the speed and scale of Google," http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html, June 2011.

[3] A. Kumar, "Development at the speed and scale of Google," http://www.infoq.com/presentations/Development-at-Google, December 2010.

[4] "Java compiler API," http://jcp.org/en/jsr/detail?id=199, December 2006.

[5] "More new I/O APIs for the Java platform (NIO.2)," http://jcp.org/en/jsr/detail?id=203, July 2011.

[6] https://developer.mozilla.org/en/Dehydra.

[7] https://wiki.mozilla.org/Pork.

[8] http://clang.llvm.org/.

[9] A. Telea and L. Voinea, "A tool for optimizing the build performance of large software code bases," in *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, ser. CSMR '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 323–325.

[10] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos, "Reducing build time through precompilations for evolving large software," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 59–68.

[11] http://maven.apache.org.

[12] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[13] http://code.google.com/p/error-prone/.

[14] http://docs.oracle.com/javase/6/docs/api/java/lang/SuppressWarnings.html.

[15] http://code.google.com/p/guava-libraries/.

[16] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 363–375.

[17] http://www.eclipse.org/projects/project.php?id=eclipse.

[18] http://www.jetbrains.com/idea/.

[19] http://pmd.sourceforge.net/.

[20] http://www.coverity.com/.

[21] http://www.klocwork.com/.

[22] http://www.parasoft.com/.

[23] S. McPeak, "Elkhound: A GLR parser generator and Elsa: An Elkhound-based C++ parser," http://scottmcpeak.com/elkhound/, 2006.

[24] C. Carruth, "Clang MapReduce: Automatic C++ refactoring at Google scale," http://llvm.org/devmtg/2011-11/#talk2/, November 2011.

[25] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 1–8.

[26] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241–252.