# Scalable, Example-Based Refactorings with Refaster

Louis Wasserman
Google, Inc.
lowasser@google.com

## Abstract

We discuss Refaster, a tool that uses normal, compilable before-and-after examples of Java code to specify a Java refactoring. Refaster has been used successfully by the Java Core Libraries Team at Google to perform a wide variety of refactorings across Google's massive Java codebase. Our main contribution is that a large class of useful refactorings can be expressed in pure Java, without a specialized DSL, while keeping the tool easily accessible to average Java developers.

***Categories and Subject Descriptors*** D2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques

***Keywords*** global, large-scale, library, Java, Refaster, syntax tree, error-prone, OpenJDK, Guava, refactoring

## 1. Problem

The Java Core Libraries Team (JCLT) at Google maintains a variety of basic Java libraries, including collections, caching, concurrency, and math utilities. Many of these libraries are open-sourced as the popular Guava project. [1]

A large proportion of these library APIs were written as better-designed replacements for Google-internal utilities, but the JCLT is also responsible for deleting the old APIs and migrating their callers, which can involve changing anything from a small handful of files scattered across the codebase, to thousands of files spread across hundreds of projects. These migrations are an everyday part of the JCLT's work. In 2012, the JCLT migrated the callers of, and deleted, 156 APIs.

Notably, most library refactorings are straightforward to describe and do not require deep analysis of the code, especially since coding style and use patterns are relatively consistent across the Google codebase. I use the term *refactoring* (or *global refactoring*) to refer to large-scale transformations intended to preserve the correctness of the codebase, whether or not the transformation is semantics-preserving in the general case.

I present a refactoring tool, which I named *Refaster*, designed to satisfy three key goals:

1. *Scalability*: the ability to efficiently perform a refactoring over the entire Google codebase.
2. *Expressiveness*: the ability to express a large class of library refactorings.
3. *Usability*: should be easily usable, understandable, and predictable to most Java developers.

### 1.1 Related Work

Most preexisting refactoring tools the JCLT was aware of fell into one of three categories, none of which quite satisfied the team's needs.

- The "canned" refactoring tools built into most IDEs, such as renaming, inlining, or extracting methods, are focused on the most common operations needed for relatively local edits. These refactoring tools are highly usable, but could not express some of the more complex transformations we required. Some of the more complex refactorings needed by the JCLT, such as "change calls to method $A$ whose argument is the result of method call $B$ into method call $C$," seemed unreasonable to add to an IDE with constrained menu space. Additionally, none of these tools could be scaled to the Google codebase.
- Refactoring tools like Jackpot [5] used specialized DSLs to express code analyses and transformations. These tools matched our needs much better than the simple canned refactorings, but had some problems: none of them could express Java generics constraints, they required users to learn a specialized and unfamiliar DSL, and none of them were built to scale.
- A project at Google, named "error-prone," [2] built an open-source Java API (built on top of the OpenJDK compiler) to perform arbitrary analysis on the fully typed syntax tree of source code, and to output simple text replacements. On top of this API, a tool called JavacFlume was built to map these transformations over the entire Google codebase. JavacFlume performs a Map/Reduce (powered

Copyright © ACM, 2013. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in WRT '13, Workshop on Refactoring Tools, http://dx.doi.org/10.1145/2541348.2541355.

*WRT '13*, Oct 27, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2604-9/13/10. . . $15.00.
http://dx.doi.org/10.1145/2541348.2541355

by the FlumeJava pipeline tool [3]) over a daily snapshot of everything needed to build all Java code at Google: all source code, and the compiled dependencies of each target. This Map/Reduce recompiles every Java file at Google with a modified compiler that runs the custom AST analysis and outputs simple text replacements for each file. Most analyses take between 5 and 20 minutes, which we considered adequately scalable.

error-prone's API is primarily designed to identify classes of programmer errors, a task at which it excels, but the API can be difficult to use for simple refactorings. Many users found programmatically manipulating the AST to be disorienting, and to take a long time even for relatively simple refactorings. Recipients of refactoring changes found it difficult to understand the logic used by the refactoring code.

## 2. Refaster

Based on these experiences, I developed Refaster. Refaster is a command-line tool that takes as input a normal, compilable Java class containing a before-and-after example of Java code, and applies the corresponding transformation across the Java codebase. Refaster's primary goal is to be able to express most library refactorings while being intuitive and easy to use.

### 2.1 Example

One recent refactoring migrated users to a new API for Base64 encoding [4]. The original code might have looked like, for example,

```
System.out.println(Base64.encodeWebSafe(
  Files.toByteArray(file),
  false /* no padding */));
```

with the new API looking like

```
System.out.println(
  BaseEncoding.base64Url().omitPadding()
    .encode(Files.toByteArray(file)));
```

A Refaster refactoring for this migration looks like:

```
class BaseEncodingMigration {
  @BeforeTemplate public String before(
      byte[] bytes) {
    return Base64.encodeWebSafe(
      bytes, false /* no padding */);
  }
  @AfterTemplate public String after(
      byte[] bytes) {
    return BaseEncoding.base64Url()
      .omitPadding()
      .encode(bytes);
  }
}
```

This refactoring migrates users who pass in a literal `false` to `Base64.encodeWebSafe` to use the `BaseEncoding` API instead. Note that `bytes` is a placeholder for any expression of type `byte[]`, not just a variable.

Notably, Refaster includes no dataflow analysis; it requires a *syntactic* match. For example, this code would not be refactored:

```
doPadding = false;
System.out.println(Base64.encodeWebSafe(
  Files.toByteArray(file), doPadding));
```

### 2.2 Transformation Structure

A Refaster transformation consists of a class with one or more methods annotated `@BeforeTemplate`, and one method annotated `@AfterTemplate`. Each template method may have arguments, called *expression variables*, corresponding to expressions of the appropriate type appearing in user code, such as `bytes` in the Base64 example. The body of the template method is a single `return` on an expression; this is necessary to make sure the template compiles as normal Java code, with the return type corresponding to the type of the expression. Refaster currently only supports refactoring expressions, not multi-statement constructs. The transformation should be read as replacing any expression which matches one of the `@BeforeTemplate` expressions with the corresponding expression in the `@AfterTemplate`. Refaster evaluates a match of a `@BeforeTemplate` method $T$ against an expression in user code $E$ as follows:

1. Verify that the syntax tree of $E$ matches the syntax tree of $T$, binding expression variables to the corresponding expressions from $E$.
2. Verify that any method overloads called in $E$ match the method overloads called in $T$.
3. Invoke the compiler's type checker on an attempt to invoke $T$ with the actual expressions from $E$ substituted for the expression variables. (This notably allows Refaster to automatically benefit from generics, autoboxing, and the other features of the Java type system, while using typechecking rules already familiar to Java developers.)

If each of these steps is successful, Refaster generates a text substitution replacing $E$ with the expression from the `@AfterTemplate`, with the actual expressions from $E$ substituted for the appropriate expression variables.

Refaster was built on top of the JavacFlume infrastructure, and runs across the Google codebase as fast as any other JavacFlume refactoring, between 5 and 20 minutes. This cost is small enough to be dominated by testing and code reviewing time, and is not a major bottleneck. Notably, the time to assemble and prepare the resources for the Map/Reduce dominates the time to perform the actual analysis, so it was not critical to heavily optimize Refaster's analysis pass. Notably, due to Google's approach of a single repository with all development done at head, changes that

modify the entire codebase are not significantly different from local changes, and do not necessarily need to be split up.

### 2.3 Results

Refaster has proven itself adequately scalable and usable enough for the JCLT's refactoring needs. Refaster has been used for over 30 refactorings committed to the Google codebase, including both local and global refactorings, and refactorings performed inside and outside the JCLT. Users so far have reported that expressing a transformation with a before-and-after example of normal Java is natural and straightforward. Additionally, once a user understands the basic structure of a Refaster template, the meaning of any other Refaster template is almost instantly apparent, compared to code that has to decompose an AST. The Base64 migration example above is in many ways typical of the JCLT's API migration needs: the API to be migrated is simple and stateless, and the migration has essentially no preconditions beyond what can be expressed with a syntactic match and with the type system. Support for complex refactorings was deliberately set aside in favor of keeping the tool as simple to use as possible. Despite these simplifications, Refaster can express several useful constraints:

- refactoring invocations where an argument is a particular literal, as in the Base64 example
- refactoring a particular chain of method invocations:

```
// this refactoring was actually performed
class UseHashBytesShortcutRefactoring {
  @BeforeTemplate public HashCode before(
      HashFunction hashFn, byte[] bytes) {
    return hashFn.newHasher()
      .putBytes(bytes)
      .hash();
  }
  @AfterTemplate public HashCode after(
      HashFunction hashFn, byte[] bytes) {
    return hashFn.hashBytes(bytes);
  }
}
```

- constraining arguments to be the result of a particular method call:

```
// this refactoring was actually performed
class HashStringRefactoring {
  @BeforeTemplate public HashCode before(
      HashFunction hashFn, String string,
      Charset charset) {
    return hashFn.hashBytes(
      string.getBytes(charset));
  }
  @AfterTemplate public HashCode after(
      HashFunction hashFn, String string,
      Charset charset) {
    return hashFn.hashString(
      string, charset);
  }
}
```

In some more complex cases, Refaster has been useful in performing intermediate steps in a more involved migration process. For example, the following API

```
interface InputSupplier {
  InputStream openStream();
}
class ByteStreams {
  ...
  static HashCode hash(InputSupplier supplier,
      HashFunction function) {..}
}
```

was migrated to a new, fluent replacement API:

```
abstract class ByteSource {
  abstract InputStream openStream();
  HashCode hash(HashFunction function) {...}
}
```

To accomplish this migration, the JCLT made `ByteSource` temporarily implement `InputSupplier`, and began migrating all the sources of `InputSupplier` instances to return `ByteSource` instances. At the same time, Refaster was used with the following template:

```
class ByteStreamHashMigration {
  @BeforeTemplate HashCode hashByteSource(
      ByteSource src, HashFunction fn) {
    return ByteStreams.hash(src, fn);
  }
  @AfterTemplate HashCode fluentHash(
      ByteSource src, HashFunction fn) {
    return src.hash(fn);
  }
}
```

This Refaster refactoring was safe even though new uses of `InputSupplier` crept into the codebase while we were still trying to migrate existing users. Follow-up migrations replaced more `InputSupplier` occurrences with `ByteSource`, allowing the migration to progress incrementally.

### 2.4 Limitations

Users have encountered two main failure modes in Refaster: first, the false negatives discussed above, where Refaster fails to migrate a method invocation because of a syntactic mismatch, where dataflow analysis might have produced a positive result. False positives take a variety of forms: in some cases, side-effecting expressions can cause problems if Refaster rearranges them. Additionally, Refaster sometimes attempts to rewrite the implementation of the new API. For example, if `BaseEncoding.base64Url()` above

used `Base64.encodeWebSafe` in its implementation, Refaster might create an infinite loop by trying to reimplement `BaseEncoding.base64Url()` in terms of itself. Refaster also requires special logic to ensure that it does not refactor its own input templates, as the `@BeforeTemplate` methods, by definition, invoke the method to be replaced. None of these failure modes have been insurmountable problems; a small handful of manual migrations or additional `@BeforeTemplate` rules have sufficed to address false negatives, and unit tests have sufficed to identify false positives in all observed cases.

The team have so far encountered only a handful of JCLT refactorings where Refaster was not useful at all. For example, some refactorings have required whole-class analysis, or adding or removing a thrown exception type from user methods, which Refaster cannot do. In many ways, it is unsurprising that most JCLT refactorings require at most small variations on traditional common transformations such as renaming and inlining. Renaming and inlining make up a large percentage of smaller-scale refactorings [6]; I suspect the variations required by the JCLT are largely due to the unusually large differences between the old and new APIs.

One unfortunate limitation has been that Refaster gains all of the limitations of the Java compiler's type inference, along with the benefits: most notably, it fails to match in cases where the Java compiler would require an explicit type argument, i.e. where the compiler would require `Foo.<String>bar()`. Additionally, Refaster cannot currently recognize cases where the `@AfterTemplate` would require an explicit type argument. Even if the original user code used an explicit type argument, Refaster cannot assume that method invocations in the `@BeforeTemplate` are in one-to-one correspondence with the method invocations in the `@AfterTemplate`. This issue could conceivably be addressed by recompiling the user code with the appropriate substitution, but this would require significant changes to Refaster's implementation.

## 3. Future Work

Refaster does not include any dataflow analysis for two main reasons: the additional effort required to add dataflow analysis, and the difficulty of supporting dataflow analysis without introducing a DSL, while at the same time keeping Refaster simple and predictable even for beginner users. This would be an interesting direction to explore further, as dataflow analysis would make possible some of the refactorings we have not yet been able to apply with Refaster.

In the future, we hope to open-source Refaster. The primary obstacle is integrating Refaster with an open-source build system like Maven, as Refaster requires compilations to be augmented with a side input (the refactoring template) and to output modified source code.


## Acknowledgments

Refaster builds on the work of many people, but especially the error-prone creators, Eddie Aftandilian, Caitlin Sadowski, and Alex Eagle, and the team behind the static analysis infrastructure at Google, led by Jeffrey van Gogh. Kevin Bourrillion and Toby Smith were instrumental in their support for the Refaster project.


## References


[1] Guava project on Google Code, 2013. URL `https://code.google.com/p/guava-libraries/`.

[2] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 14–23. IEEE, 2012.

[3] 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*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. . URL `http://doi.acm.org/10.1145/1806596.1806638`.

[4] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), Oct. 2006. URL `http://www.ietf.org/rfc/rfc4648.txt`.

[5] J. Lahoda, J. Bečička, and R. B. Ruijs. Custom declarative refactoring in NetBeans: tool demonstration. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 63–64, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1500-5. . URL `http://doi.acm.org/10.1145/2328876.2328886`.

[6] E. Murphy-Hill, C. Parnin, and A. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012. ISSN 0098-5589. .