Spinnaker and the Distributed Monorepo

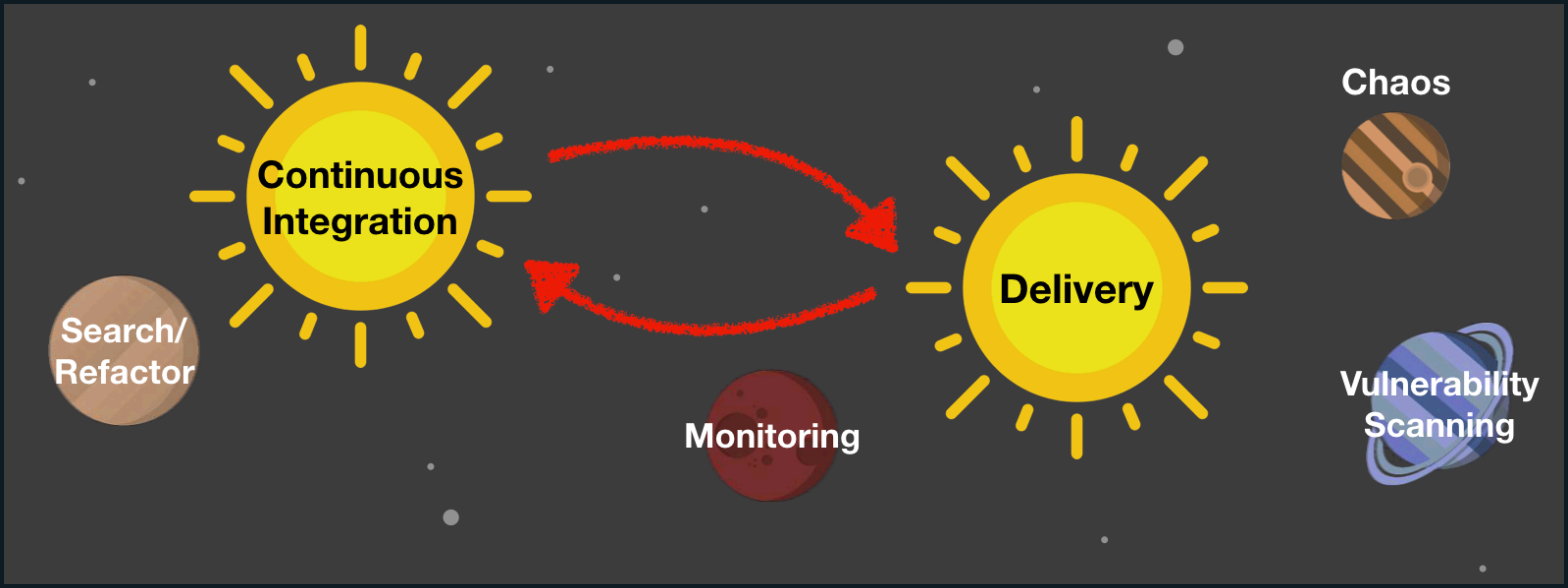
Jon Schneider

`@jon_k_schneider`

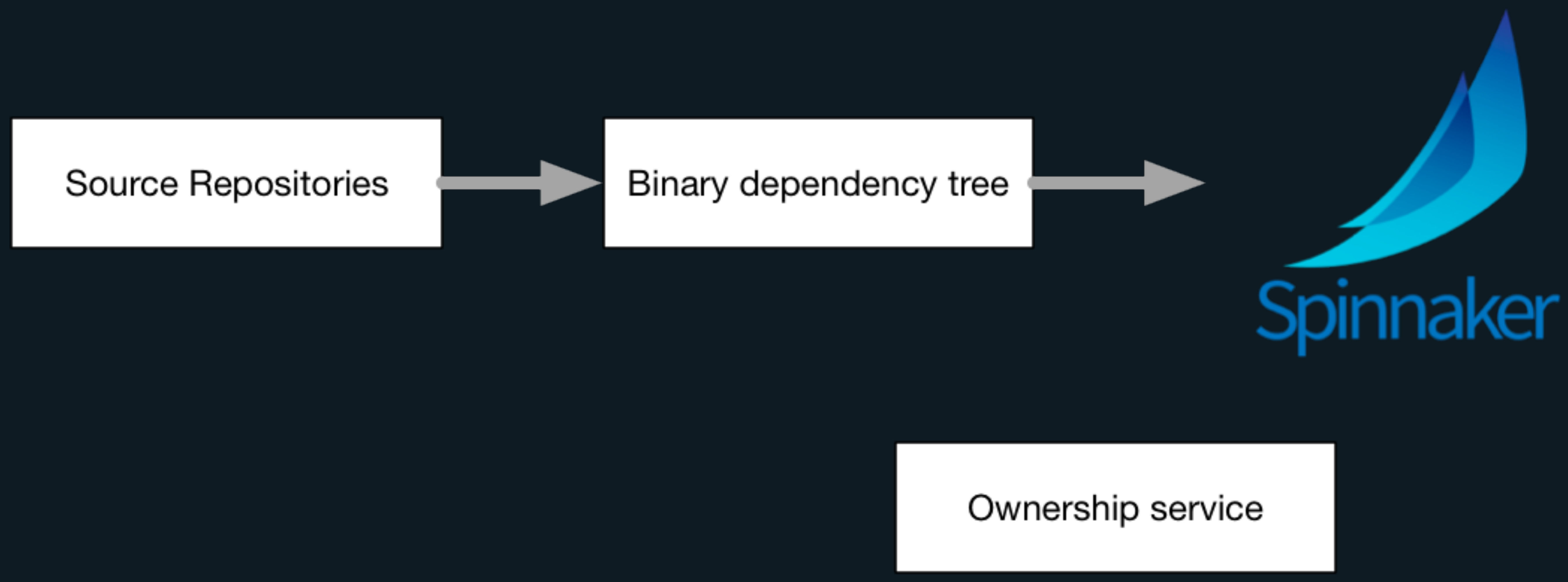
github.com/jkschneider/springone-distributed-monorepo

Part 1: The Spinnaker resource model

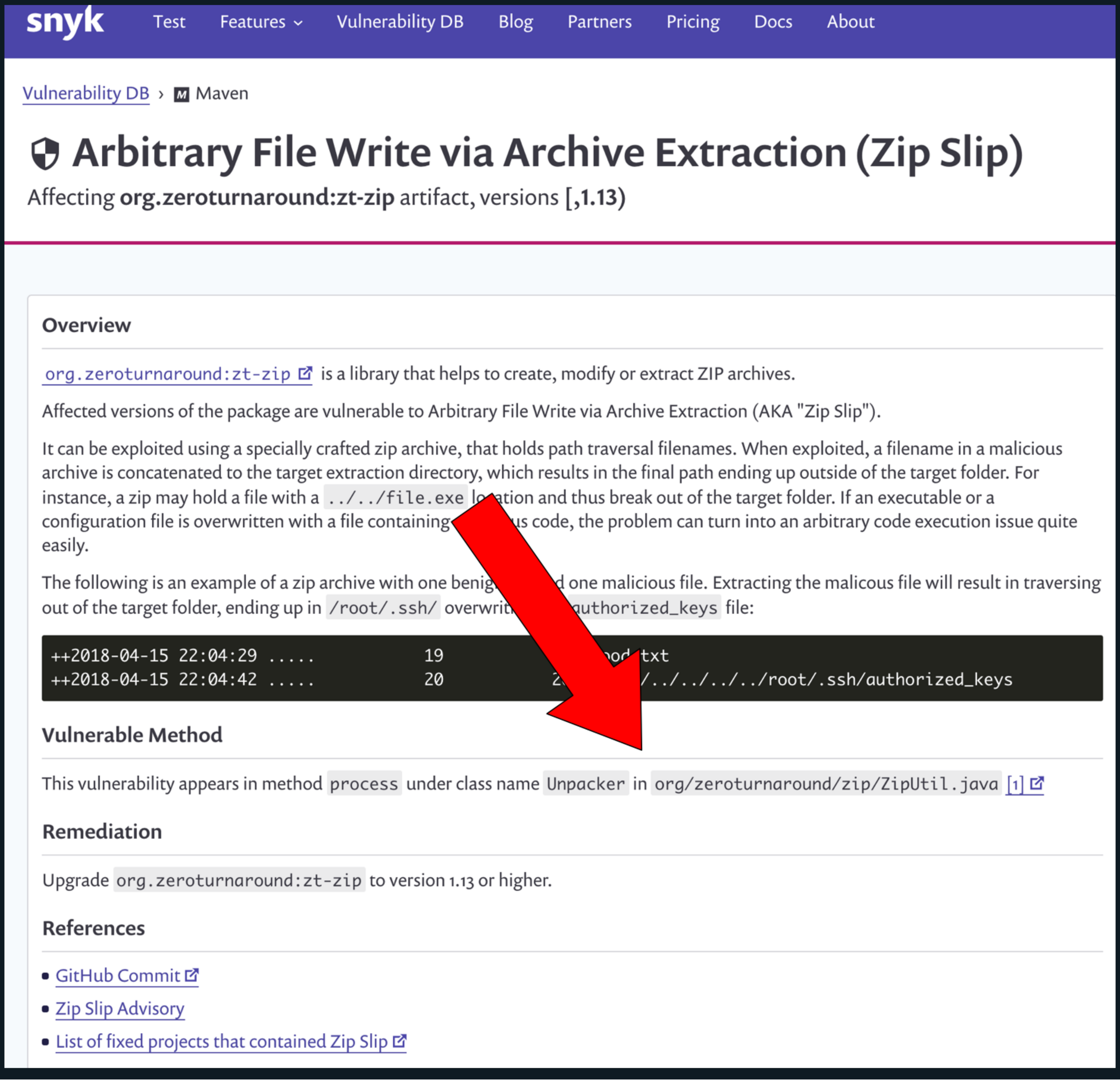
CI and CD have distinct orbits



Stitching together source, binaries, and deployed assets.



Part 2: Generate a binary callgraph to search for method level vulnerabilities



Vulnerability DB › Maven

Arbitrary File Write via Archive Extraction (Zip Slip)

Affecting **org.zeroturnaround:zt-zip** artifact, versions **[,1.13)**

Overview

`org.zeroturnaround:zt-zip` is a library that helps to create, modify or extract ZIP archives.

Affected versions of the package are vulnerable to Arbitrary File Write via Archive Extraction (AKA "Zip Slip").

It can be exploited using a specially crafted zip archive, that holds path traversal filenames. When exploited, a filename in a malicious archive is concatenated to the target extraction directory, which results in the final path ending up outside of the target folder. For instance, a zip may hold a file with a `../../file.exe` lo[illegible]ation and thus break out of the target folder. If an executable or a configuration file is overwritten with a file containing [illegible]us code, the problem can turn into an arbitrary code execution issue quite easily.

The following is an example of a zip archive with one benig[illegible]d one malicious file. Extracting the malicous file will result in traversing out of the target folder, ending up in `/root/.ssh/` overwrit[illegible]`authorized_keys` file:

```
++2018-04-15 22:04:29 .....           19             [illegible]ood.txt
++2018-04-15 22:04:42 .....           20             2[illegible]/../../../../root/.ssh/authorized_keys
```

Vulnerable Method

This vulnerability appears in method `process` under class name `Unpacker` in `org/zeroturnaround/zip/ZipUtil.java` [1]

Remediation

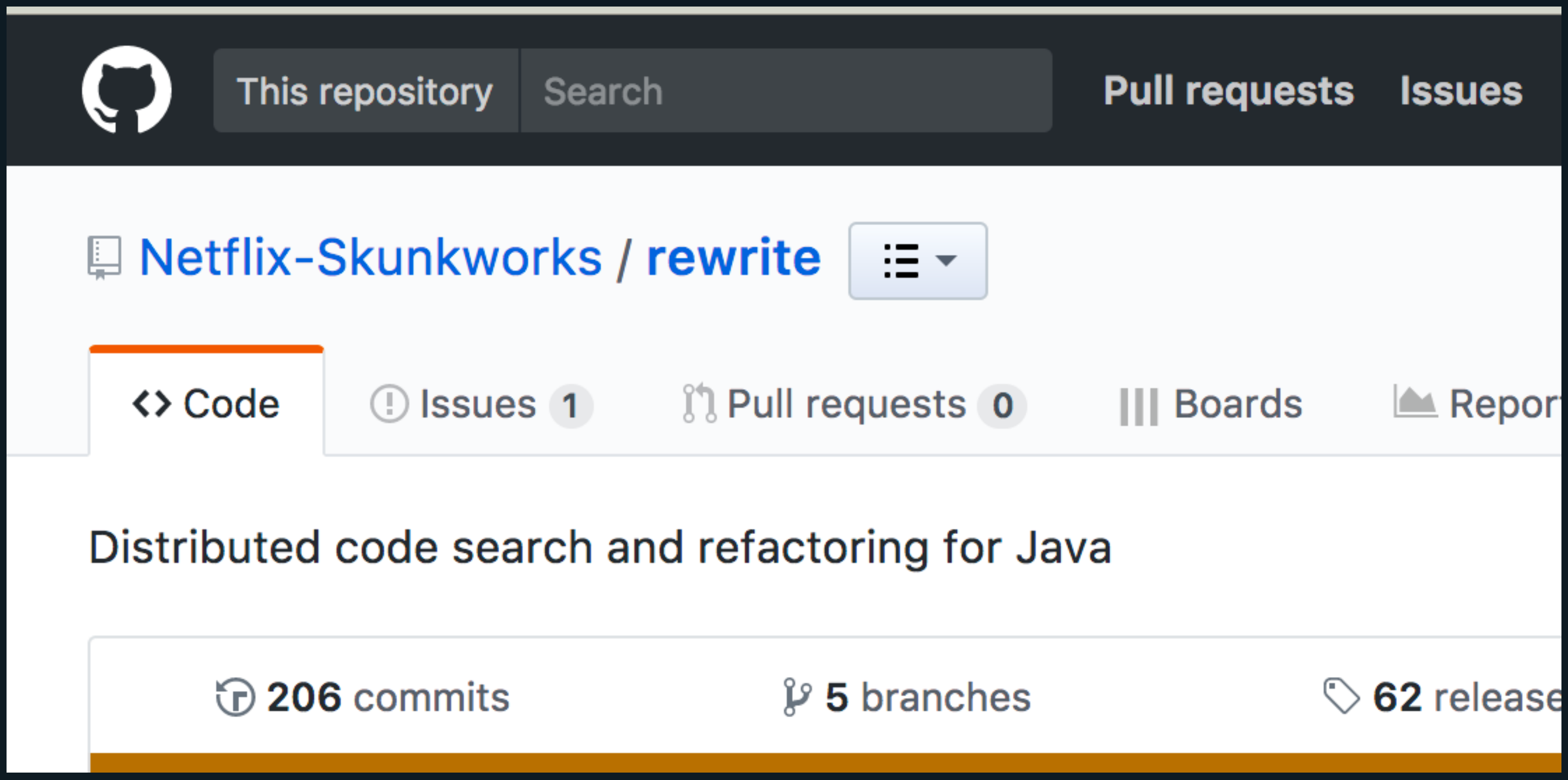
Upgrade `org.zeroturnaround:zt-zip` to version 1.13 or higher.

References

- GitHub Commit
- Zip Slip Advisory
- List of fixed projects that contained Zip Slip

Part 1: Rewriting code

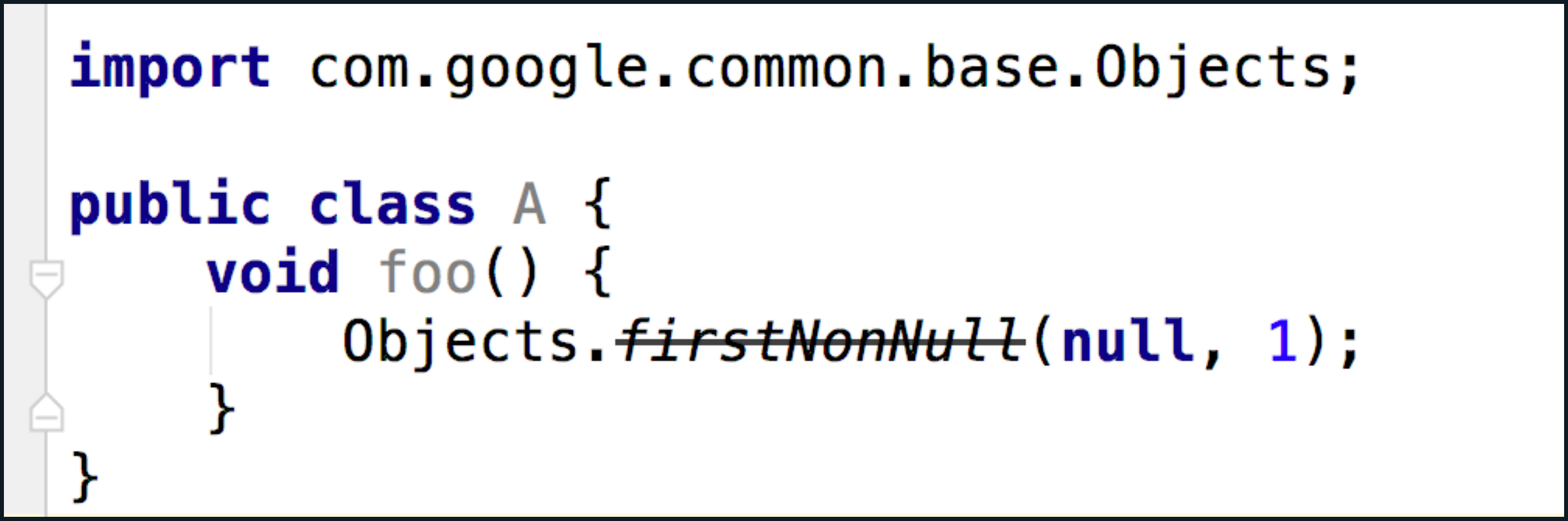
Rewrite is a programmatic refactoring tool.



Distributed code search and refactoring for Java

206 commits
5 branches
62 release

Suppose we have a simple class A.

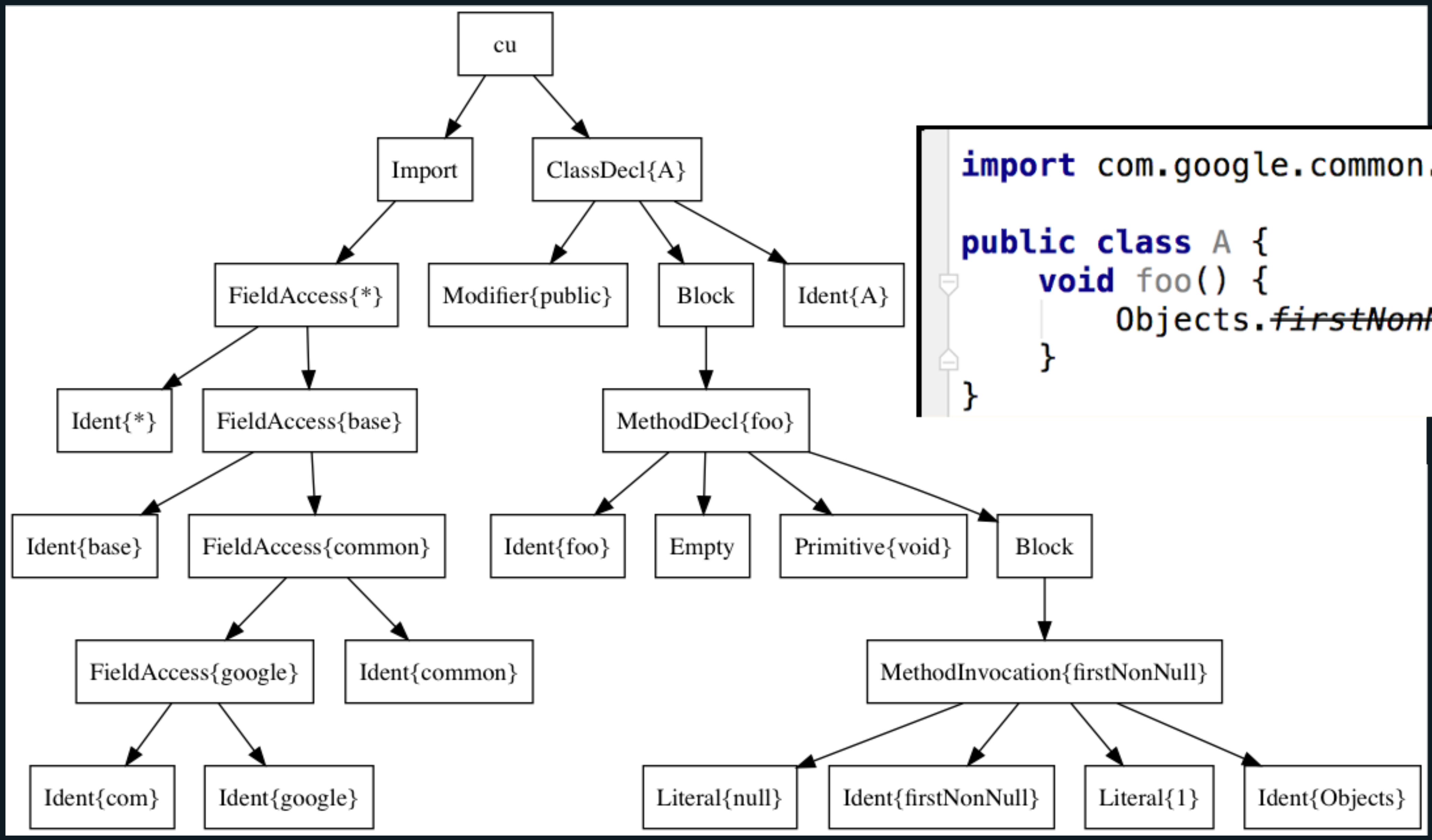


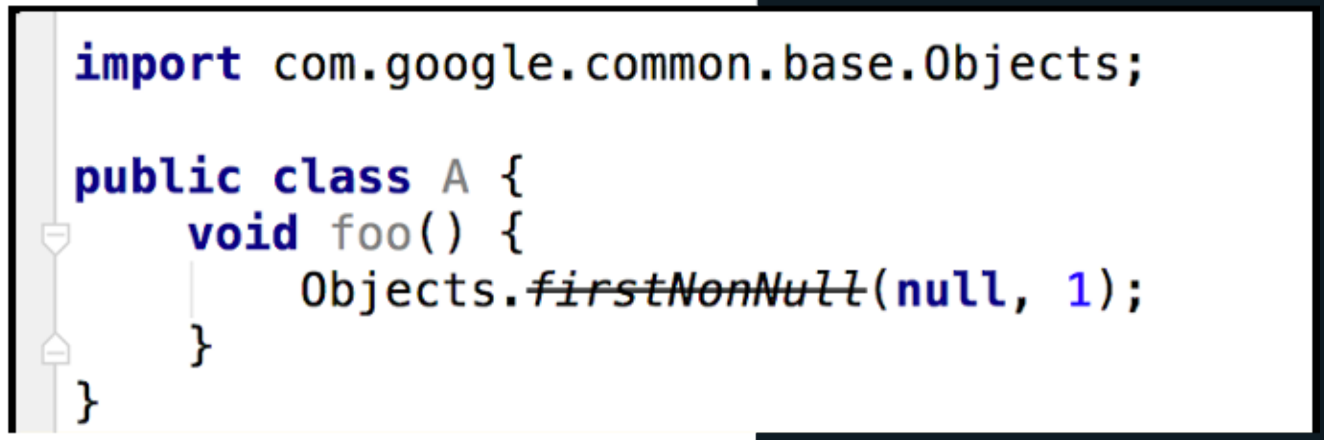
```
import com.google.common.base.Objects;

public class A {
    void foo() {
        Objects.~~firstNonNull~~(null, 1);
    }
}
```

Raw source code + classpath = Rewrite AST.

```java
String javaSource = /* Read A.java */;
List<Path> classpath = /* A list including Guava */;

Tr.CompilationUnit cu = new OracleJdkParser(classpath)
        .parse(javaSource);

assert(cu.firstClass().getSimpleName().equals("A"));
```

The Rewrite AST covers the whole Java language.



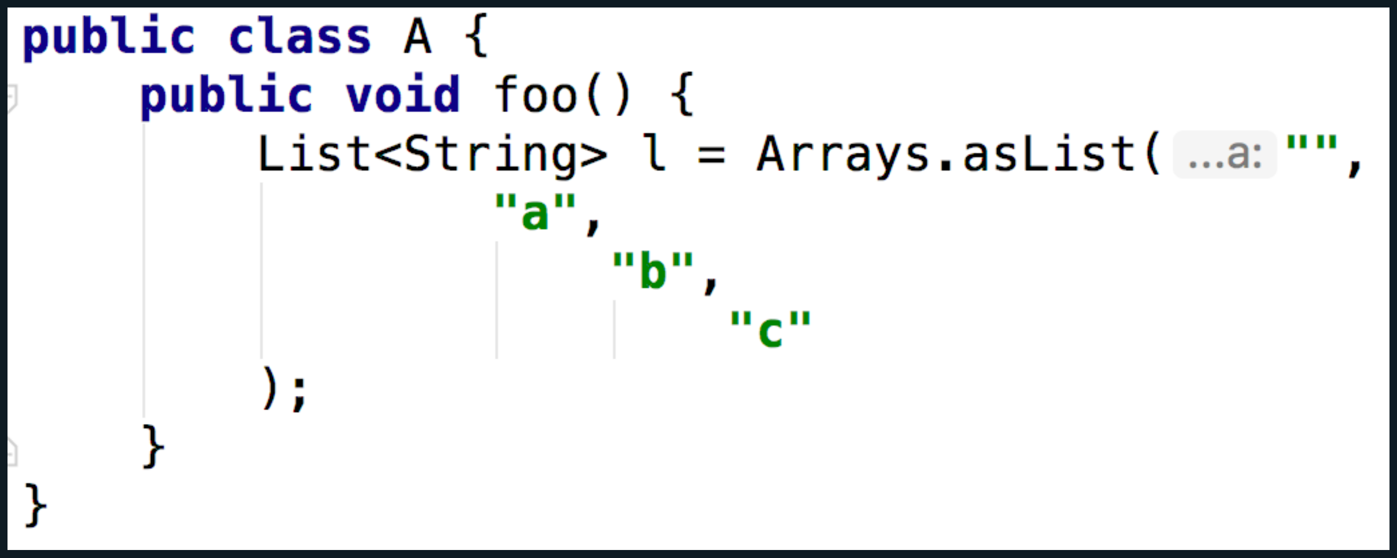


```
import com.google.common.base.Objects;

public class A {
    void foo() {
        Objects.~~firstNonNull~~(null, 1);
    }
}
```

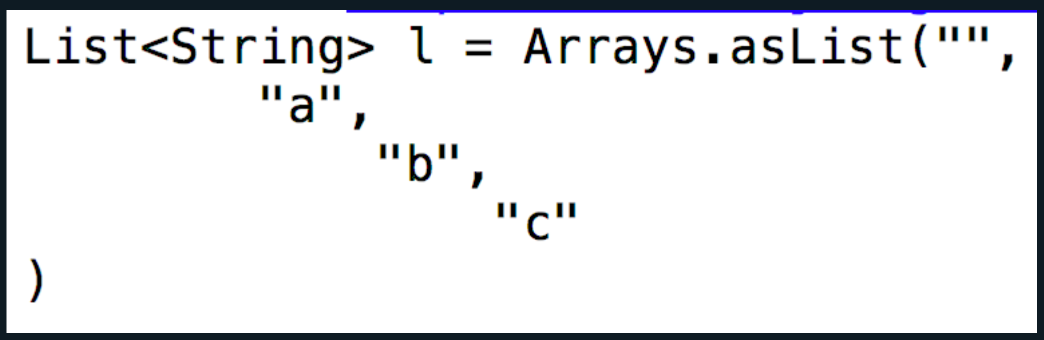
Rewrite's AST is special.

1. Serializable
2. Acyclic
3. Type-attributed

Rewrite's AST preserves formatting.



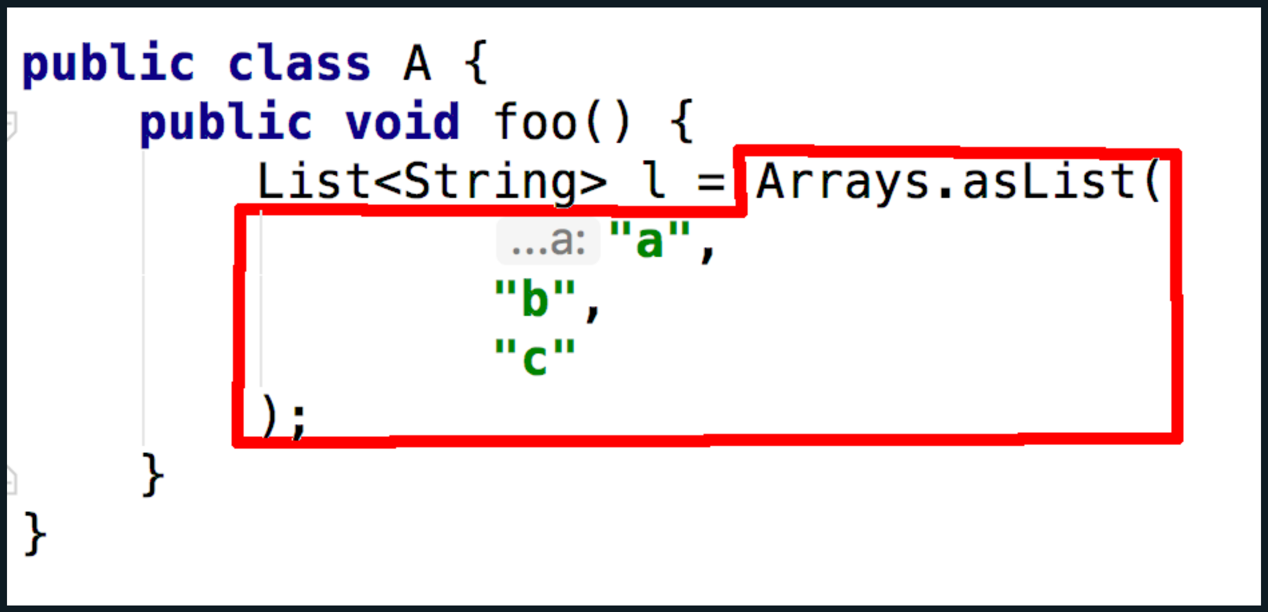
```
public class A {
    public void foo() {
        List<String> l = Arrays.asList( ...a: "",
                "a",
                    "b",
                        "c"
        );
    }
}
```



```
List<String> l = Arrays.asList("",
        "a",
            "b",
                "c"
)
```

```
Tr.CompilationUnit cu = new OracleJdkParser().parse(aSource);
        assertThat(cu.print()).isEqualTo(aSource);

cu.firstClass().methods().get(0) // first method
  .getBody().getStatements() // method contents
  .forEach(t -> System.out.println(t.printTrimmed()));
```

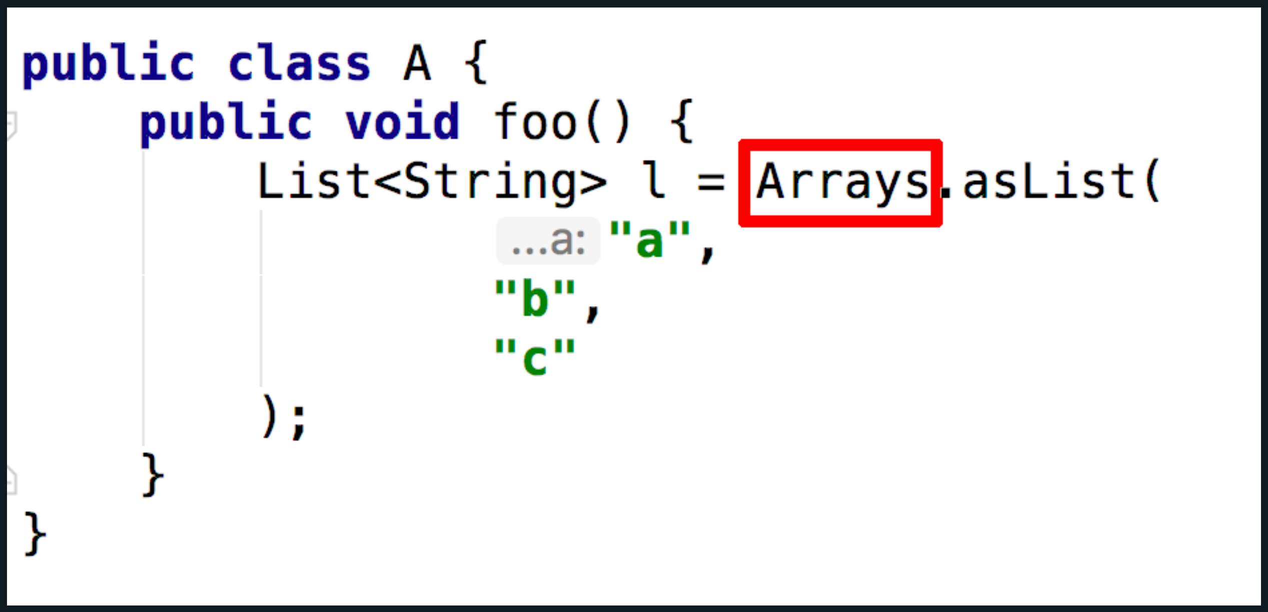
We can find method calls and fields from the AST.



```java
public class A {
    public void foo() {
        List<String> l = Arrays.asList(
                ...a: "a",
                "b",
                "c"
        );
    }
}
```

```java
Tr.CompilationUnit cu = new OracleJdkParser().parse(aSource);

assertThat(cu.findMethodCalls("java.util.Arrays asList(..)")).hasSize(1);
assertThat(cu.firstClass().findFields("java.util.Arrays")).isEmpty();
```

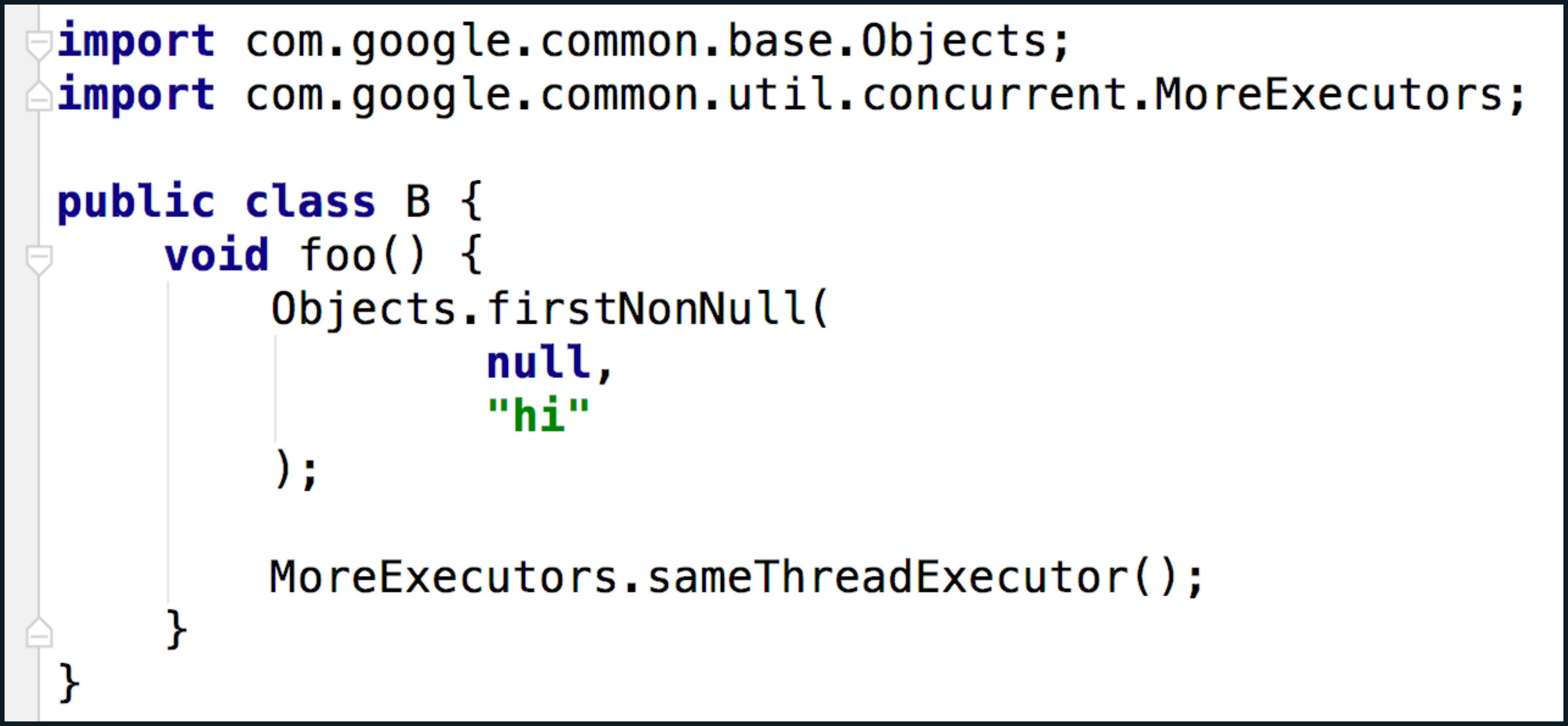
We can find types from the AST.



```
public class A {
    public void foo() {
        List<String> l = Arrays.asList(
                ...a: "a",
                "b",
                "c"
        );
    }
}
```

```
assertThat(cu.hasType("java.util.Arrays")).isTrue();
assertThat(cu.hasType(Arrays.class)).isTrue();

assertThat(cu.findType(Arrays.class))
  .hasSize(1).hasOnlyElementsOfType(Tr.Ident.class);
```

Suppose we have a class referring to a deprecated Guava method.

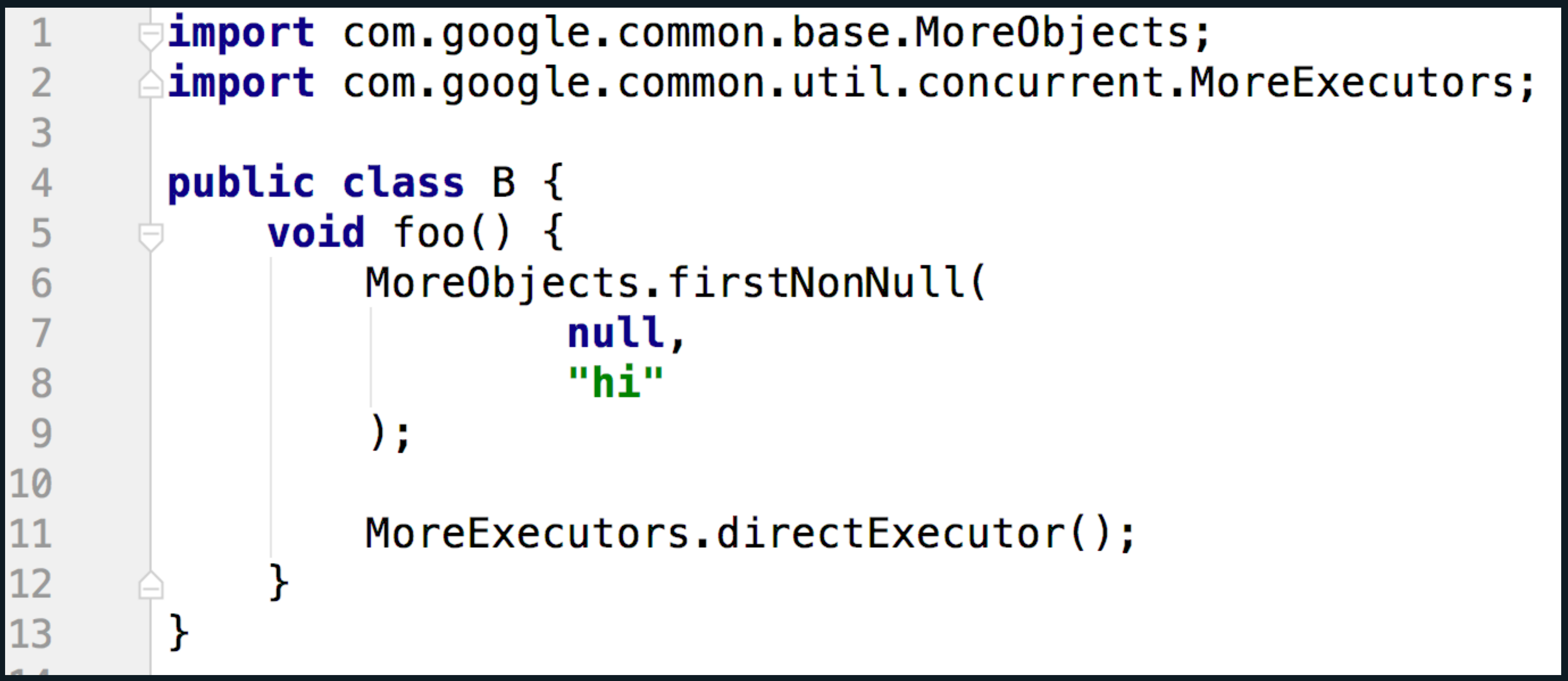


```java
import com.google.common.base.Objects;
import com.google.common.util.concurrent.MoreExecutors;

public class B {
    void foo() {
        Objects.firstNonNull(
                null,
                "hi"
        );

        MoreExecutors.sameThreadExecutor();
    }
}
```

We can refactor both deprecated references.

```java
Tr.CompilationUnit cu = new OracleJdkParser().parse(bSource);
Refactor refactor = cu.refactor();

refactor.changeMethodTargetToStatic(
  cu.findMethodCalls("com.google..Objects firstNonNull(..)"),
  "com.google.common.base.MoreObjects"
);

refactor.changeMethodName(
  cu.findMethodCalls("com.google..MoreExecutors sameThreadExecutor()"),
  "directExecutor"
);
```

The fixed code emitted from Refactor can be used to overwrite the original source.

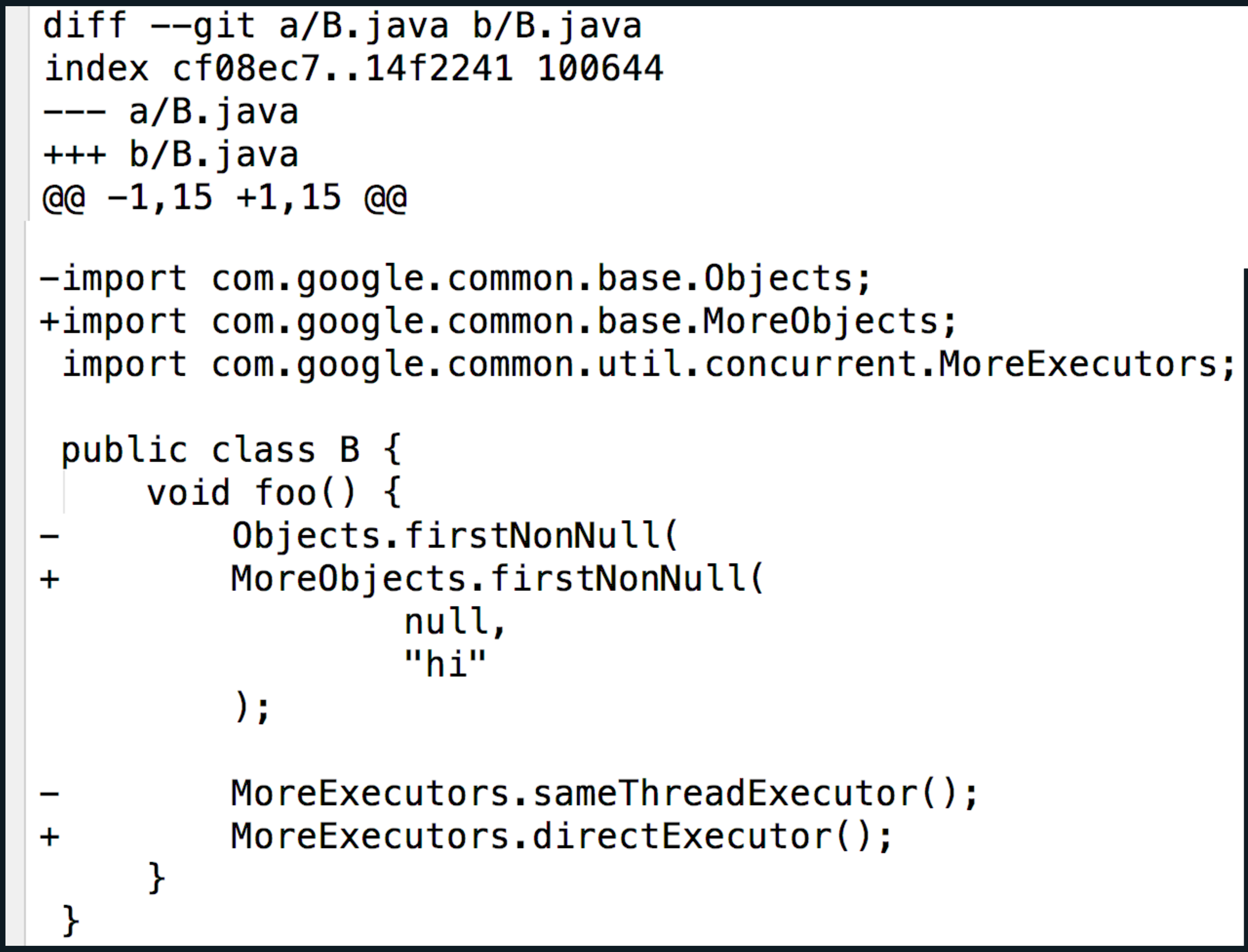


```java
import com.google.common.base.MoreObjects;
import com.google.common.util.concurrent.MoreExecutors;

public class B {
    void foo() {
        MoreObjects.firstNonNull(
                null,
                "hi"
        );

        MoreExecutors.directExecutor();
    }
}
```

```java
// emits a string containing the fixed code, style preserved
refactor.fix().print();
```

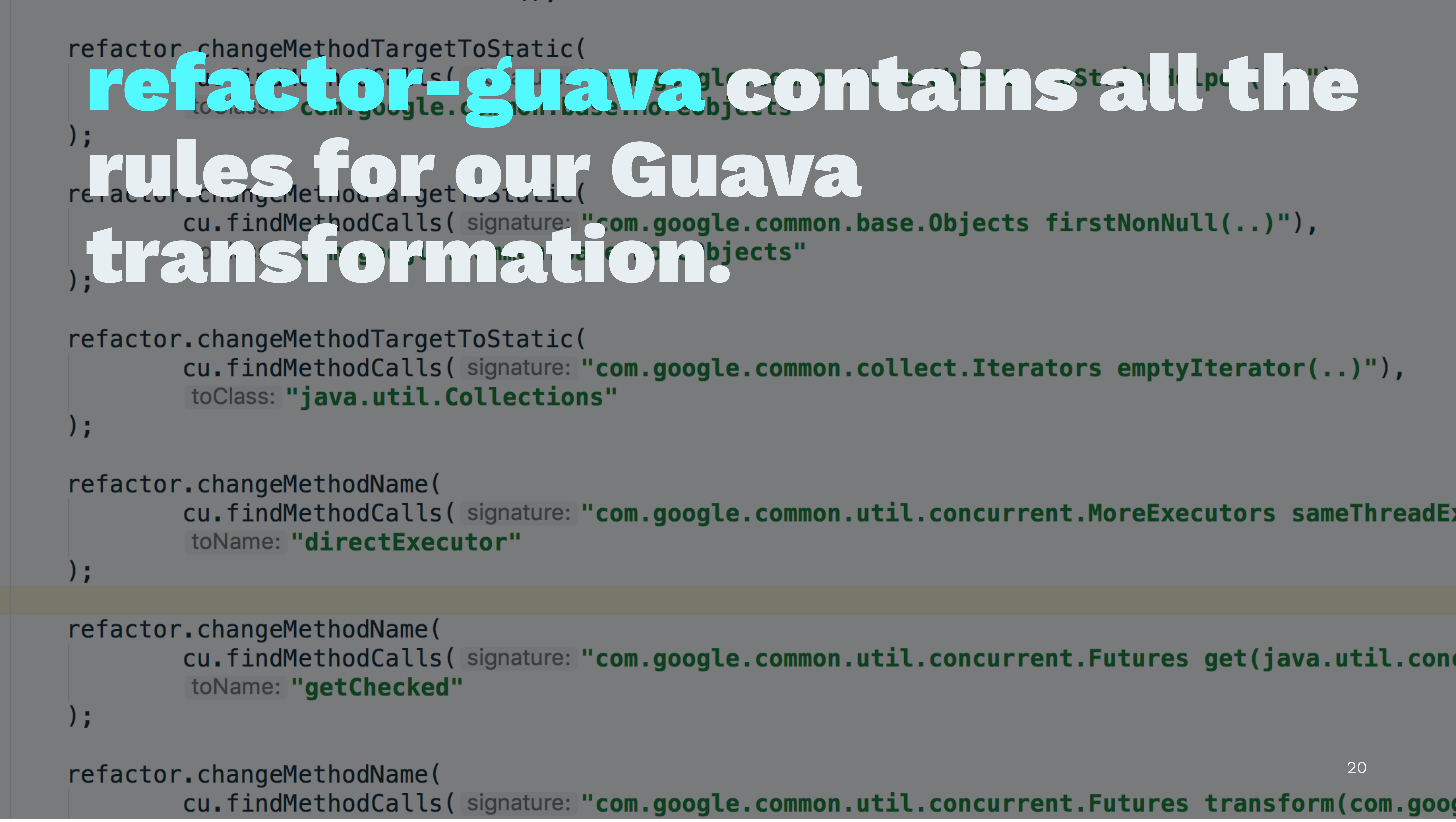
Or we can emit a diff that can be used with `git apply`

```
// emits a String containing the diff
refactor.diff();
```



```
diff --git a/B.java b/B.java
index cf08ec7..14f2241 100644
--- a/B.java
+++ b/B.java
@@ -1,15 +1,15 @@

-import com.google.common.base.Objects;
+import com.google.common.base.MoreObjects;
 import com.google.common.util.concurrent.MoreExecutors;

 public class B {
     void foo() {
-        Objects.firstNonNull(
+        MoreObjects.firstNonNull(
                 null,
                 "hi"
         );

-        MoreExecutors.sameThreadExecutor();
+        MoreExecutors.directExecutor();
     }
 }
```

refactor-guava contains all the rules for our Guava transformation.



```
refactor.changeMethodTargetToStatic(
        cu.findMethodCalls( signature: [illegible]St[illegible]lp[illegible]"),
        toClass: "com.google.c[illegible]base.MoreObjects"
);

refactor.changeMethodTargetToStatic(
        cu.findMethodCalls( signature: "com.google.common.base.Objects firstNonNull(..)"),
        [illegible]bjects"
);

refactor.changeMethodTargetToStatic(
        cu.findMethodCalls( signature: "com.google.common.collect.Iterators emptyIterator(..)"),
        toClass: "java.util.Collections"
);

refactor.changeMethodName(
        cu.findMethodCalls( signature: "com.google.common.util.concurrent.MoreExecutors sameThreadEx
        toName: "directExecutor"
);

refactor.changeMethodName(
        cu.findMethodCalls( signature: "com.google.common.util.concurrent.Futures get(java.util.conc
        toName: "getChecked"
);

refactor.changeMethodName(
        cu.findMethodCalls( signature: "com.google.common.util.concurrent.Futures transform(com.goog
```

Just annotate a static method to define a refactor rule.

```java
@AutoRewrite(value = "reactor-mono-flatmap",
             description = "change flatMap to flatMapMany")
public static void migrateMonoFlatMap(Refactor refactor) {
  // a compilation unit for the source file we are refactoring
  Tr.CompilationUnit cu = refactor.getOriginal();

  refactor.changeMethodName(
    cu.findMethodCalls("reactor..Mono flatMap(..)"),
    "flatMapMany");
}
```

Part 2: Using BigQuery to find all Guava code in Github

Identify all Java sources from BigQuery's Github copy.

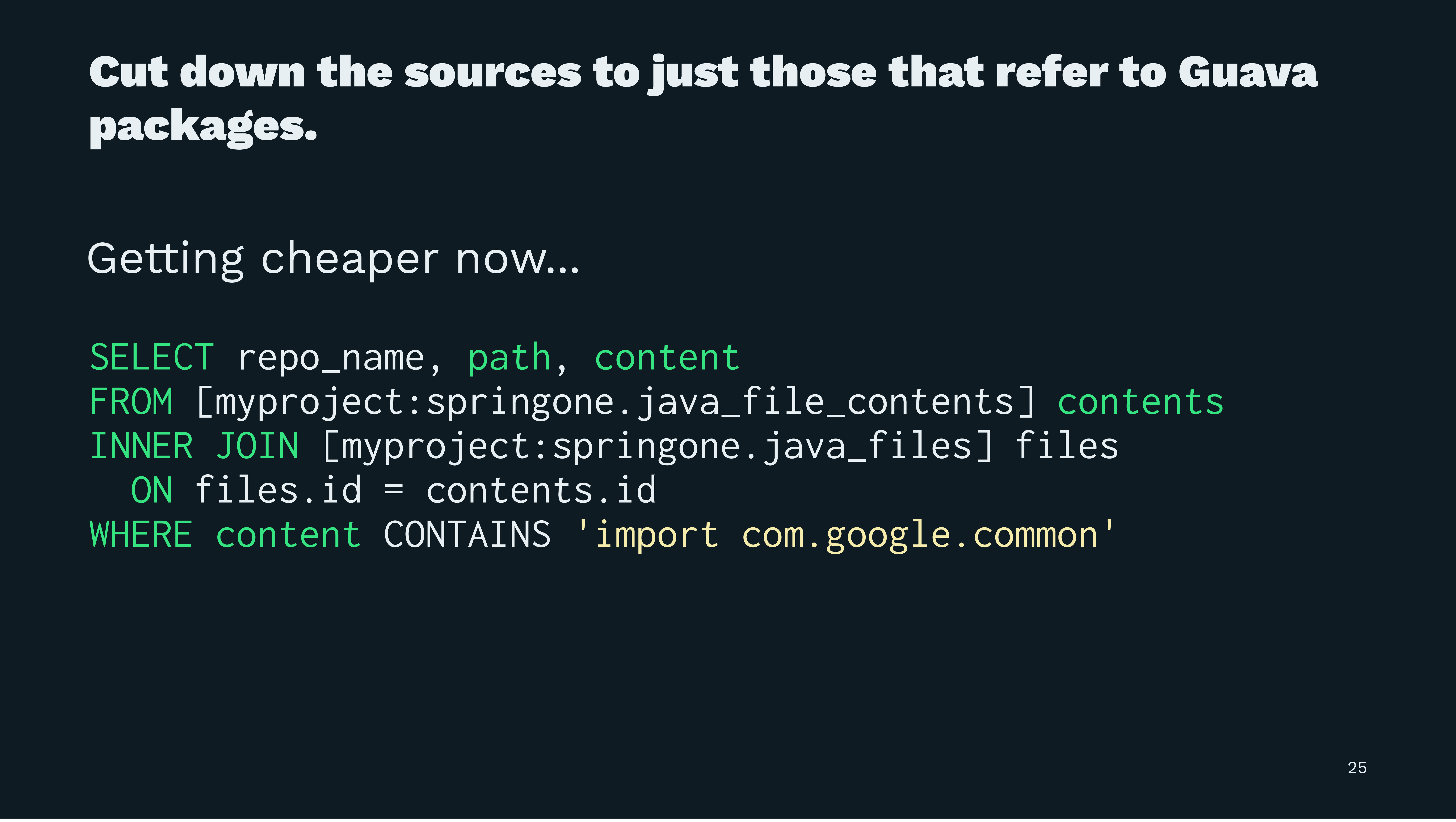
```
SELECT *
FROM [bigquery-public-data:github_repos.files]
WHERE RIGHT(path, 5) = '.java'
```

In options, save the results of this query to: `myproject:springone.java_files`. You will have to allow large results as well. This is a fairly cheap query (336 GB).

Move Java source file contents to our dataset.

```
SELECT *
FROM [bigquery-public-data:github_repos.contents]
WHERE id IN (
  SELECT id
  FROM [myproject:springone.java_files]
)
```

Note: This will eat into your $300 credits.
It cost me ~$6 (1.94 TB).



Cut down the sources to just those that refer to Guava packages.

Getting cheaper now...

```
SELECT repo_name, path, content
FROM [myproject:springone.java_file_contents] contents
INNER JOIN [myproject:springone.java_files] files
  ON files.id = contents.id
WHERE content CONTAINS 'import com.google.common'
```

Notice we are going to join just enough data from `springone.java_files` and `springone:java_file_contents` in order to be able to construct our PRs.

Save the result to `myproject:springone.java_file_contents_guava`.

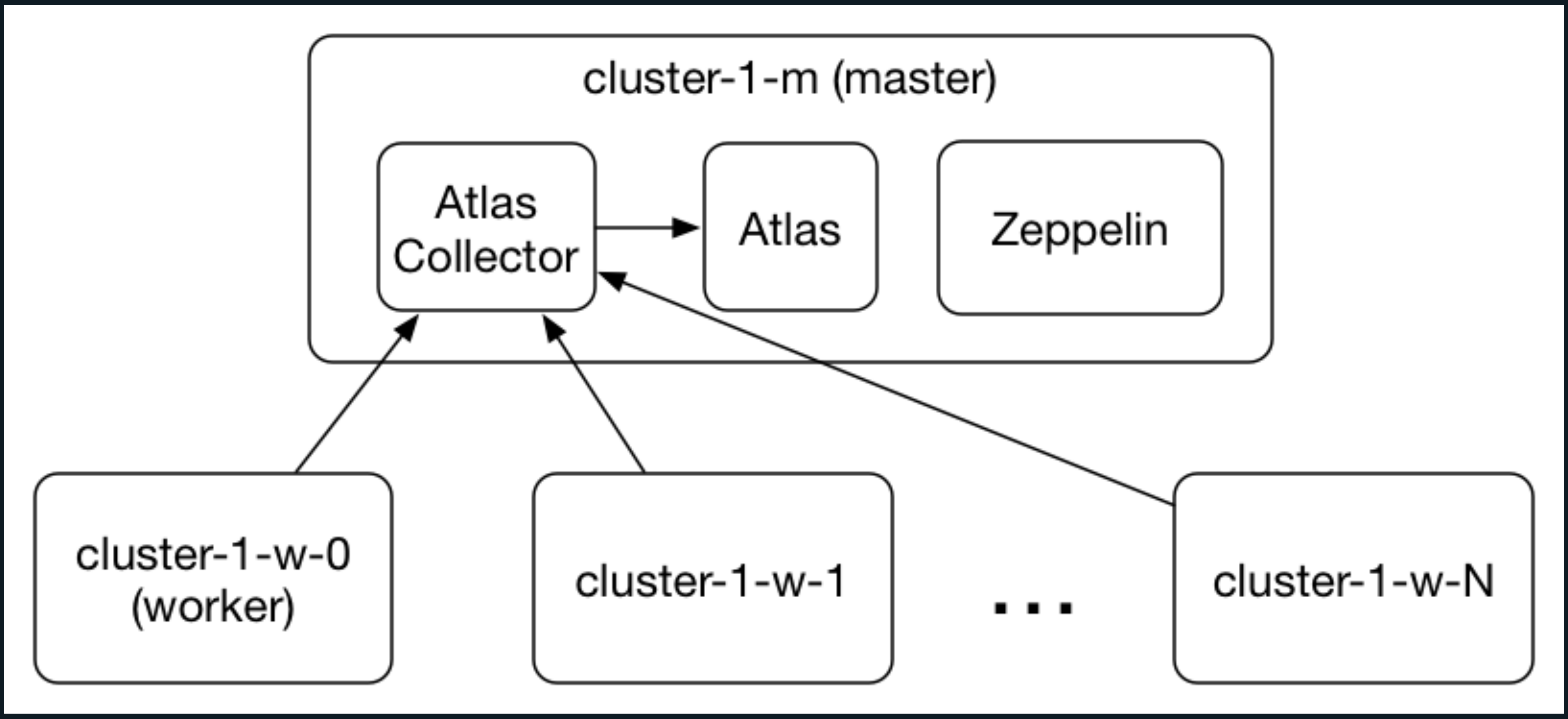
Through Step 3, we have cut down the size of the initial BigQuery public dataset from 1.94 TB to around 25 GB. Much more manageable!

We now have the dataset to run our refactoring rule on.

1. 2.6 million Java source files.
2. 47,565 Github repositories.

Part 3: Employing our refactoring rule at scale on Google Cloud Dataproc.

Create a Spark/Zeppelin cluster on Google Cloud Dataproc.



Monitoring our Spark workers with Atlas and spring-metrics

```
@SpringBootApplication
@EnableAtlasMetrics // (1)
public class AtlasCollector {
    public static void main(String[] args) {
        SpringApplication.run(AtlasCollector.class, args);
    }
}
```

…

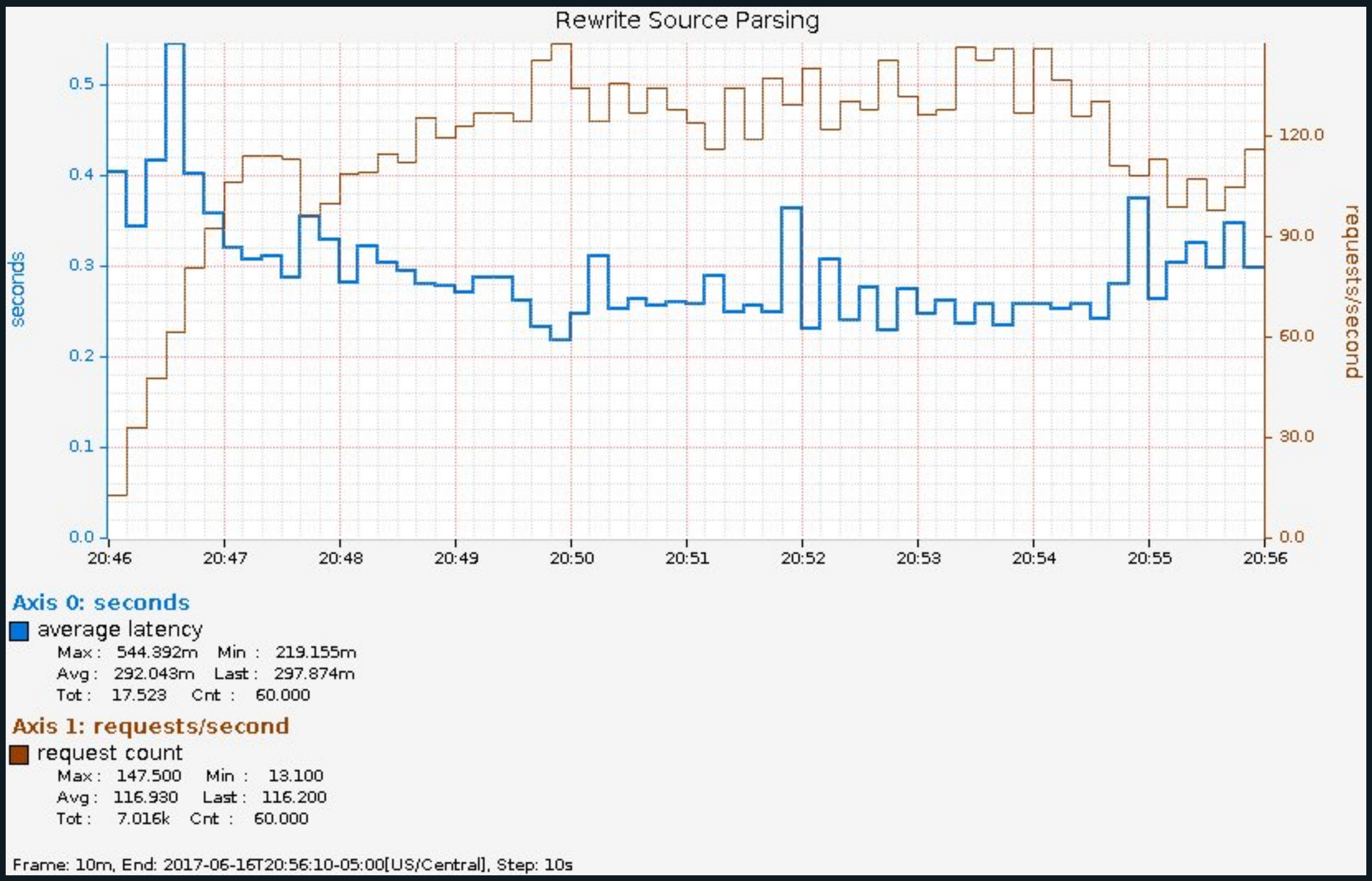
Monitoring our Spark workers with Atlas and spring-metrics

```
@RestController
@Timed // (2)
class TimerController {
  @Autowired MeterRegistry registry; // (3)
  Map<String, Timer> timers =
      new ConcurrentHashMap<>();

  @PostMapping("/api/timer/{name}/{timeNanos}")
  public void time(@PathVariable String name,
    @PathVariable Long timeNanos) {
      timers.computeIfAbsent(name, registry::timer)
          .record(timeNanos, TimeUnit.NANOSECONDS); // (4)
  }
}
```

We'll write the job in a Zeppelin notebook.

1. Select sources from BigQuery
2. Map over all the rows, parsing and running the refactor rule.
3. Export our results back to BigQuery.

Measuring our initial pass.



```
Axis 0: seconds
■ average latency
     Max :   544.392m    Min :   219.155m
     Avg :   292.043m    Last :  297.874m
     Tot :    17.523     Cnt :    60.000
Axis 1: requests/second
■ request count
     Max :   147.500    Min :    13.100
     Avg :   116.930    Last :  116.200
     Tot :     7.016k   Cnt :    60.000

Frame: 10m, End: 2017-06-16T20:56:10-05:00[US/Central], Step: 10s
```

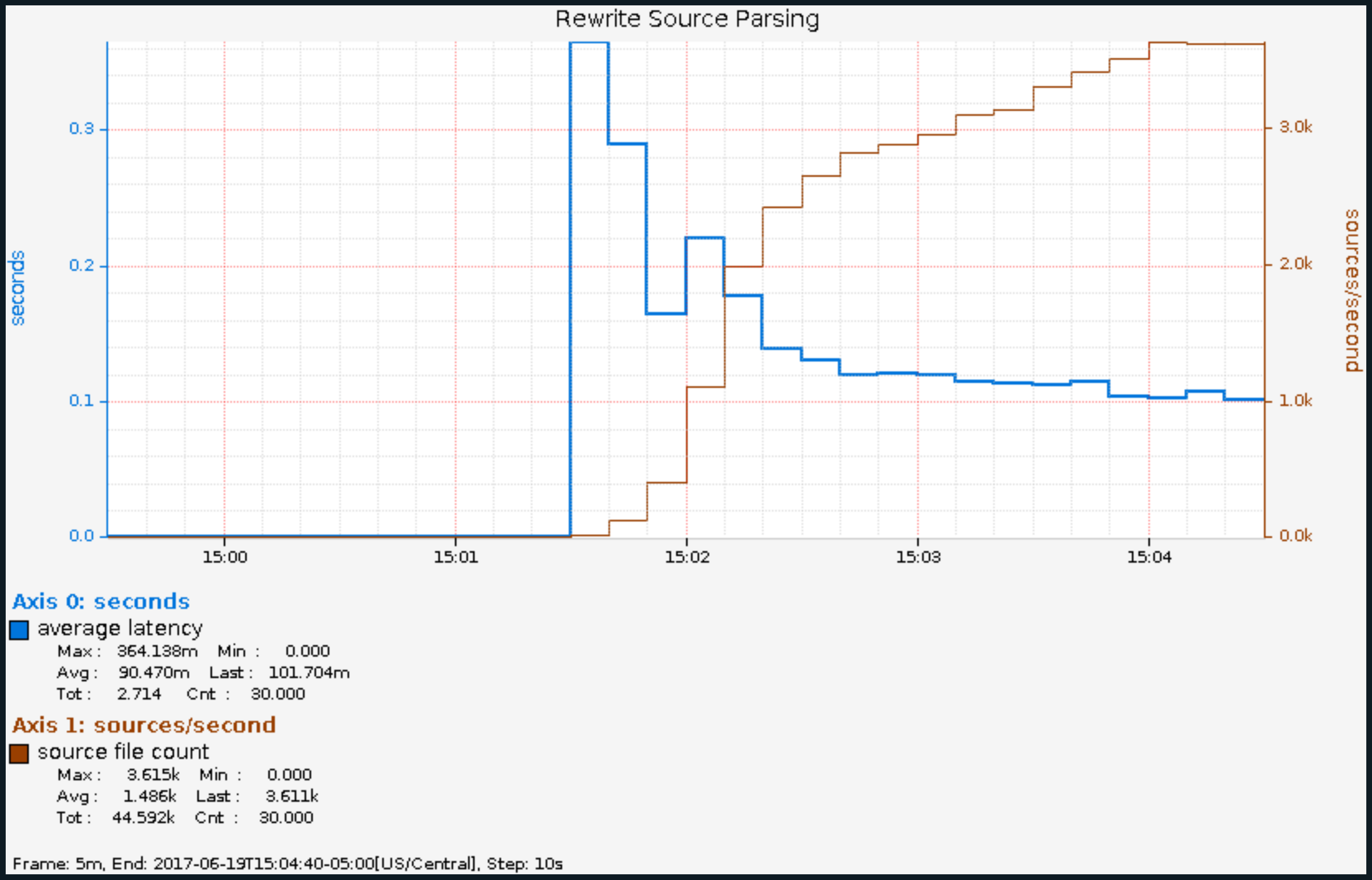
Measuring how big our cluster needs to be.

1. Rewrite averages 0.12s per Java source file
2. Rate of 6.25 sources per core / second
3. With 128 preemptible VMs, we've got:
   512 cores * 6.25 sources / core / second

3,200 sources / second = ~13 minutes total

We hope...

After scaling up the cluster with a bunch of cheap VMs.



Axis 0: seconds

🟦 average latency

```
Max :  364.138m   Min :     0.000
Avg :   90.470m   Last :  101.704m
Tot :    2.714    Cnt :    30.000
```

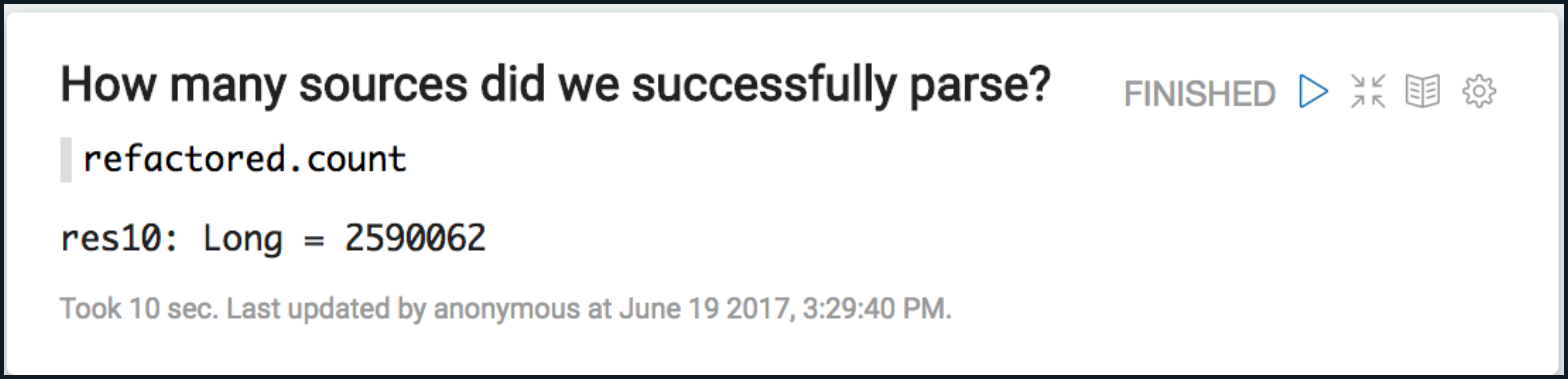
Axis 1: sources/second

🟫 source file count

```
Max :    3.615k   Min :     0.000
Avg :    1.486k   Last :    3.611k
Tot :   44.592k   Cnt :    30.000
```

Frame: 5m, End: 2017-06-19T15:04:40-05:00[US/Central], Step: 10s

Some source files are too badly formed to parse.


How many sources did we successfully parse?

```
refactored.count
```

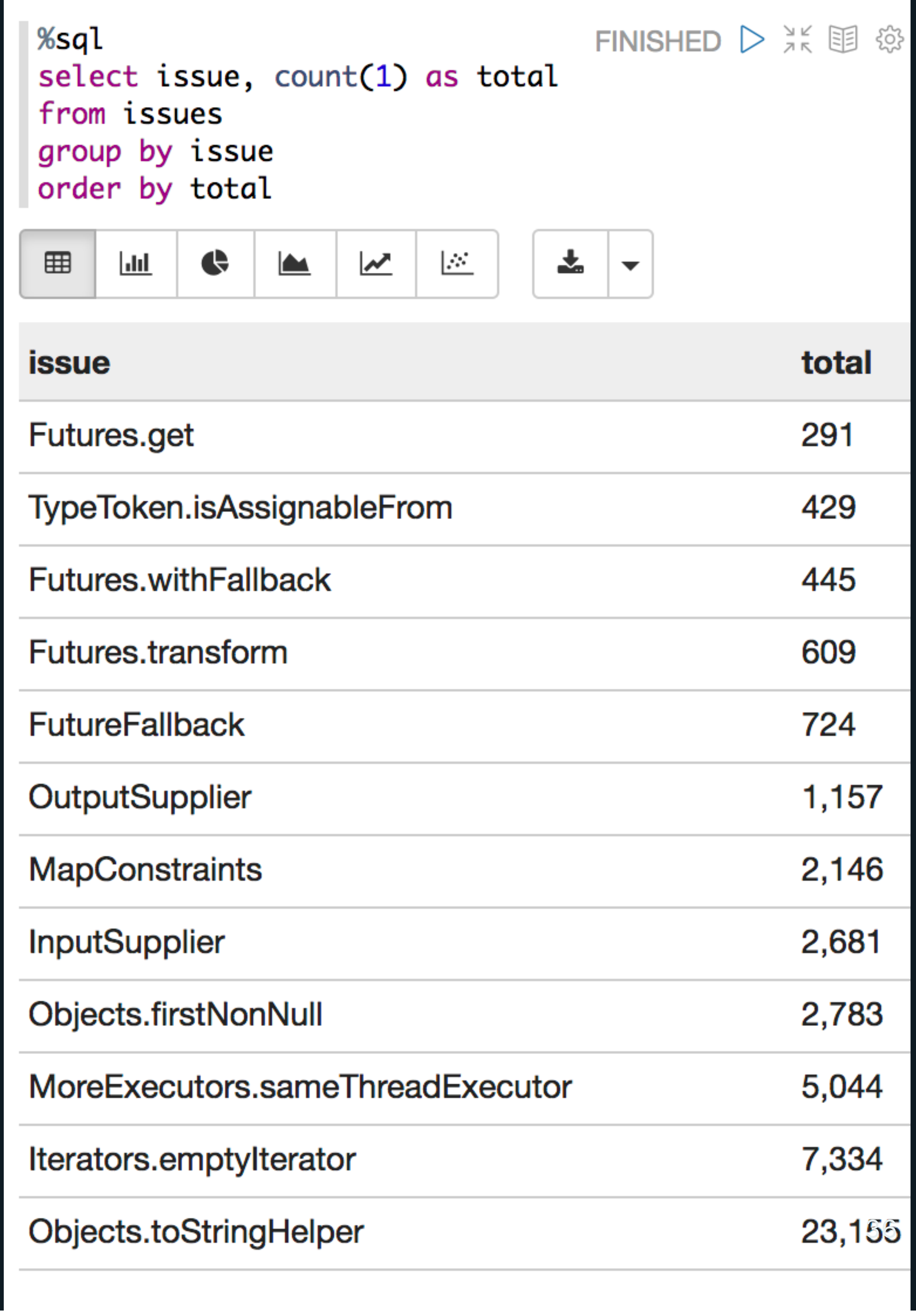
```
res10: Long = 2590062
```

Took 10 sec. Last updated by anonymous at June 19 2017, 3:29:40 PM.

2,590,062/2,687,984 Java sources = 96.4%.

We found a healthy number of issues.

- 4,860 of 47,565 projects with problems
- 10.2% of projects with Guava references use deprecated API
- 42,794 source files with problems
- 70,641 lines of code affected



```
%sql
select issue, count(1) as total
from issues
group by issue
order by total
```

issue	total
Futures.get	291
TypeToken.isAssignableFrom	429
Futures.withFallback	445
Futures.transform	609
FutureFallback	724
OutputSupplier	1,157
MapConstraints	2,146
InputSupplier	2,681
Objects.firstNonNull	2,783
MoreExecutors.sameThreadExecutor	5,044
Iterators.emptyIterator	7,334
Objects.toStringHelper	23,155

Epilogue: Issuing PRs for all the patches

Generate a single patch file per repo.

```
SELECT repo, GROUP_CONCAT_UNQUOTED(diff, '\n\n') as patch
FROM [cf-sandbox-jschneider:springone.diffs]
GROUP BY repo
```

Thanks for attending!