# Systematic Testing of Refactoring Engines on Real Software Projects

Milos Gligoric[1], Farnaz Behrang[2], Yilong Li[1], Jeffrey Overbey[2], Munawar Hafiz[2], and Darko Marinov[1]

[1] University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
`{gliga, yli147, marinov}@illinois.edu`
[2] Auburn University, Auburn, AL 36849, USA
`fzb0012@tigermail.auburn.edu, jeffreyoverbey@acm.org, munawar@auburn.edu`

**Abstract.** Testing refactoring engines is a challenging problem that has gained recent attention in research. Several techniques were proposed to automate generation of programs used as test inputs and to help developers in inspecting test failures. However, these techniques can require substantial effort for writing test generators or finding unique bugs, and do not provide an estimate of how reliable refactoring engines are for refactoring tasks on real software projects.

This paper evaluates an end-to-end approach for testing refactoring engines and estimating their reliability by (1) systematically applying refactorings at a large number of places in well-known, open-source projects and collecting failures during refactoring or while trying to compile the refactored projects, (2) clustering failures into a small, manageable number of failure groups, and (3) inspecting failures to identify non-duplicate bugs. By using this approach on the Eclipse refactoring engines for Java and C, we already found and reported 77 new bugs for Java and 43 for C. Despite the seemingly large numbers of bugs, we found these refactoring engines to be relatively reliable, with only 1.4% of refactoring tasks failing for Java and 7.5% for C.

**Keywords:** Refactoring engines, Systematic testing, Test clustering

## 1 Introduction

Refactorings [11] are behavior-preserving code transformations that developers traditionally apply to improve the design of existing code. Modern IDEs—such as Eclipse, NetBeans, or Visual Studio—contain refactoring engines that automate applications of refactorings. Previous studies [6, 8, 28, 45] show that most commonly applied refactorings include renaming program elements, extracting methods, and inlining methods. The list of refactorings is growing as researchers and practitioners recognize new patterns that are worth automating [5, 9, 10, 36, 48].

Testing refactoring engines is an important yet challenging problem. It is important because bugs[3] in refactoring engines can affect programmer productivity, introduce errors in the code being refactored, and reduce the confidence of the programmer who may decide to perform manual refactorings that can be even more error-prone. It is challenging because refactoring engines require complex test inputs, i.e., programs/projects to be refactored; such test inputs are hard to generate using naive random generation or symbolic execution [14].

Automated testing of refactoring engines has gained attention in research [7, 12, 17, 35, 38, 39, 41]. Most proposed techniques require manually writing *test generators* that use sophisticated random or bounded-exhaustive generation to produce the required complex test inputs. Such techniques have had some impact on the research and practice of building refactoring engines, e.g., by finding real bugs in widely used IDEs [7, 12, 38, 39] or by affecting design of refactoring engines [15, 31, 34, 35]. However, such techniques also have several deficiencies. First, they require substantial manual effort for writing test generators. Second, the generated test inputs may not represent real refactoring scenarios; the generators often produce "corner cases" that IDE developers or users do not care about. Third, they do not provide any estimate of how reliable refactoring engines are for tasks on real software projects.

Instead of using artificially generated programs to evaluate refactoring engines, several research projects [4, 5, 37, 41, 44] use real programs. Spinellis [41] mentions testing the Rename refactoring of his CScout refactoring engine on all identifiers in the Linux kernel. Independently, Thies and Steimann [44] tested two refactorings in Eclipse in a similar manner. However, these projects did not consider the overall process from applying refactorings to inspecting failures to reporting new, unique bugs, and they did not quantify the reliability of widely used refactoring engines such as those in Eclipse. While previous studies [7,35,38,39] show (and our current study confirms) that systematic testing of refactoring engines can expose a large number of failures, it is important to map these failures to bug reports. Jagannath et al. [17] proposed a technique that clusters failures to help in inspection, but they evaluated the technique only on artificially generated programs. (Section 6 discusses related work in more detail.)

This paper makes two contributions.

**End-to-End Approach:** We propose testing refactoring engines and evaluating their reliability by combining techniques that systematically apply refactorings on a large number of places in real software projects [4, 5, 41, 44] and that effectively cluster the failures to a small number of (likely unique) bugs [17]. Our approach consists of the following steps: (1) given a set of projects, systematically apply refactorings in many places and collect failures where the refactoring

[3] The term "bug" used in this paper is more formally called a "fault", i.e., an error in the code of a refactoring engine, in contrast to a "failure", i.e., an error observed from an execution of the refactoring engine.

engine throws an exception or produces refactored code that does not compile[4], (2) split these failures into clusters such that all failures in the same/different cluster/clusters are likely due to the same/different underlying bug/bugs, (3) inspect randomly selected failures from the clusters, minimize them, identify non-duplicate bugs, and report them. We fully automated step 1, semi-automated step 2, and, for now, manually perform step 3.

While previous work explored some individual steps separately, our combined approach leads to a more effective, end-to-end methodology for evaluating refactoring engines. In contrast to techniques that automate test generation [7, 12, 17, 38, 39], our approach does not require manually writing test generators, finds bugs that occur in real refactoring tasks, and allows us to characterize reliability of refactoring engines for real refactoring tasks. We expect that bugs commonly found in real applications are more likely to be fixed than bugs discovered from artificially generated corner cases.

**Evaluation:** We use our approach to extensively evaluate the Eclipse refactoring engines for two programming languages—Java and C. Our study is the first to test *all* refactorings—23 for Java and 5 for C—currently implemented in Eclipse for these two languages. So far we have found 77 bugs in 21 refactorings for Java (not finding any bug in two refactorings) and 43 bugs in 5 refactorings for C, which is more bugs than any previous study that we are aware of [7, 12, 38, 39]. We reported these bugs to the Eclipse developers, who acknowledged our reports—"Thanks for opening all the useful bug reports. Much appreciated!" (`http://dev.eclipse.org/mhonarc/lists/jdt-ui-dev/msg01278.html`)—and have already fixed 8 of these bugs. Our clustering technique effectively reduces almost 15000 failures in Java and C to 356 clusters to be inspected. Moreover, we find refactoring engines to be relatively reliable, with the average rate of failing refactoring tasks being 1.4% for Java and 7.5% for C.

To the best of our knowledge, our study is the first to (1) evaluate this end-to-end approach of applying refactorings on real software projects and mapping the failures to unique bugs, (2) cluster failures of refactoring engines on real projects, (3) highlight the challenge of finding duplicate failures and bug reports, (4) show that this approach can be easily adopted for multiple programming languages unlike test generators that need to be written from scratch for each language, and (5) report failure rates for refactoring engines as a way to estimate their reliability. Our promising results provide motivation for the community to automate various steps from our approach, including minimization of programs that lead to failures [27, 33, 50] and searching for duplicate bug reports that involve programs as test inputs.

The key automated steps of our approach have been successfully evaluated by the ECOOP Artifact Evaluation Committee (`http://ecoop13-aec.cs.brown.edu/`) and found to meet expectations. Our main results with the links to the reported bugs are available online: `http://mir.cs.illinois.edu/rtr`.

[4] One can use other test oracles [7, 39] in addition to refactored code not compiling. Note that we check compilation only when the refactoring engine raises no warning that the refactoring should not proceed because some precondition is violated.

## 2 Example

As an example, we illustrate using our approach to test the Change Method Signature refactoring for Java. This refactoring takes as input one method and a set of changes to make to various parts of the method signature: visibility (e.g., `private`), return type, method name, and parameter list (add, remove, or reorder parameters). Changing the signature of one given method can lead to changing several other methods (e.g., those that override or are overridden by the given method) and can require changing the call sites to the method(s) being changed.

Our approach has three steps. In the first step, our automated tool compiles our corpus of real programs and finds *all* the program elements where a given refactoring can be applied (Section 3.1). For Change Method Signature, this corresponds to finding all the methods. For each such element, the tool then repeatedly applies the applicable refactoring tasks that pass the preconditions, records if there is a failure, and undoes the applied refactoring so that the next refactoring task can be applied. For Change Method Signature, our tool performs four refactoring tasks for each method: (1) changes visibility, (2) adds a parameter in first position, (3) removes the first parameter, and (4) reverses the order of parameters. (Section 3 has a detailed list.) Note that some tasks may not apply to some methods, e.g., a parameter cannot be removed if the target method has no parameters.

For the experiments, we use five popular Java projects: JPF, JUnit, log4j, Lucene, and Math (Section 5). On these projects, our tool performs a total of 28526 refactoring tasks for Change Method Signature. These tasks result in 565 failures. While the absolute number of failures is relatively large, the relative rate of failures is 565/28526=2.0%, i.e., only a relatively small fraction of all refactoring tasks result in a failure. Of these 565 failures, 555 are compiler errors denoting that the resulting program does not compile any more, and 10 are exception cases in which the refactoring engine throws an exception while applying the change. For each failure, our tool records where the refactoring is applied, the type of failure, and the messages produced by the failure—compiler error messages or exception stack traces.

It is worth pointing out that no prior study on testing refactoring engines [7, 12, 17, 35, 38, 39, 41, 44] report finding any exception case. At least one paper [7] explicitly states checking for such cases but finding no failure, and other papers use automated tools that would likely crash for uncaught exceptions and thus be observed by the researchers. Hence, the large number and diversity of real refactoring tasks, arising from applying our approach systematically on several open-source projects, enables us to discover these cases missed by previous work.

Even when the absolute number of failures is relatively large, many of them are due to the same underlying bug in the refactoring engine. Inspecting all the failures is prohibitively expensive and unnecessary to identify *unique* bugs. Since one of our goals is to identify new, unique bugs in the refactoring engine, we want to inspect a relatively small number of the failures that likely have different underlying bugs. A naive approach that randomly selects some number of failures to be inspected does not work well [17], and it is not obvious a priori how many failures to randomly inspect. Thus, one can end up wasting time by inspecting several failures with the same underlying bug, or one can miss a bug by not inspecting any failure for that bug.

In the second step, our tool splits the failures into clusters based on the messages produced by the failure (Section 3.2). Ideally, clustering should satisfy two conditions: (1) all the failures in the same cluster should have the same underlying bug such that inspecting only one representative from each cluster will not miss any bug, and (2) the failures from different clusters should have different underlying bugs such that inspecting representatives from multiple clusters will not find duplicate bugs. To cluster the failures, we build on the idea of *abstract messages* [17]. This idea was previously proposed for automatically generated test inputs for refactoring engines but was not evaluated for failures on real refactoring tasks.

For Change Method Signature, our clustering splits 555 failures with compiler errors into 10 clusters (that have between 1 and 526 failures per cluster) and splits 10 failures with exceptions into 2 clusters (that have 4 and 6 failures). If one has insufficient resources to inspect all the clusters, one can prioritize the clusters based on the type of bugs one is looking for. For example, one can inspect clusters that have more failures before clusters that have fewer failures (thus looking for common bugs rather than looking for more "corner cases"), or inspect clusters that have failures arising from multiple projects before clusters that have failures arising from only one project (thus looking for bugs that are more common to be encountered by the users), or inspect clusters with exceptions before clusters with compiler errors (our anecdotal experience shows that Eclipse developers fix the exception cases faster as they may consider these bugs to be more severe).

In the third step, we manually analyze the clusters to identify and report non-duplicate bugs. For our running example, we inspect one randomly selected failure from each of these 10+2 clusters. This inspection involves two tasks: (1) minimizing the input project to understand the underlying bug and to prepare a bug report that makes debugging easier, and (2) identifying likely duplicates among the bugs in our clusters and bugs already in the Eclipse Bugzilla database. While there is research on automated minimization [27,33,50], we currently perform minimization manually. We experienced that minimizing a failure can sometimes take less time and effort than identifying duplicate bugs. Minimizing a failure took us 5–60 minutes, with an average around 10 minutes, while identifying duplicates sometimes took over 60 minutes, with an average around 15 minutes. For the examples in figures 1a and 1b, the minimization took 10 and 60 minutes, respectively. In the end, by inspecting 10+2 (compiler+exception) failures, we found 4+2 unique bugs, and of those 1+2 bugs were previously unreported in Eclipse Bugzilla.

We discuss in more detail the two bugs that lead to uncaught exceptions. Figure 1a shows the minimized code based on the Lucene project [24] that leads to a `NullPointerException` when the Change Method Signature refactoring is used to reorder the two parameters of the method `m`. In this case, the names

```
// C.java
class C {
  C(Object o) {}
  void m() {
    new C(new Object() {
      // Reorder parameters
      void m(int i, int j)
      {}
    });
  }
}
```

(a) NullPointerException

```
// .settings/org.eclipse.jdt.core.prefs
org.eclipse.jdt.core.compiler.source=1.4

// A.java
class A {
  // Remove parameter
  void add(int i) {
  }
}
```

(b) IndexOutOfBoundsException

Fig. 1: Examples of bugs found in CHANGE METHOD SIGNATURE by applying refactorings on Lucene and log4j projects, respectively

of the class, method, and parameters are not relevant for reproducing the exception. Figure 1b shows the minimized code from log4j [23] that leads to an `IndexOutOfBoundsException` when CHANGE METHOD SIGNATURE is used to remove the parameter of the method `add`; as opposed to the first bug, the method name must be `add` in order to be able to reproduce the bug. Additionally, the project must be using Java version 1.4 or lower (shown in the `settings` file in Figure 1b). Indeed, we find that reproducing some bugs requires more information about the project rather than just the source code of the program. This bug cannot be exposed by existing automated techniques for testing refactoring engines [7,12,38,39], because they focus on generating Java source code and not project configurations. In contrast, we found these bugs by applying refactorings on real projects.

While the minimized versions can look like "corner cases", the bugs actually arise on real code, and the IDE developers can use that information to prioritize fixing of the bugs. For example, `NullPointerException` related to Figure 1a arises in four refactoring tasks, whereas `IndexOutOfBoundsException` related to Figure 1b arises in six refactoring tasks.

## 3 Approach

This section describes in more detail our end-to-end approach for testing refactoring engines. Our approach consists of three main steps: (1) *collecting failures* discovers all refactoring tasks, runs these tasks, and outputs failing tasks (Section 3.1), (2) *clustering failures* splits failing tasks into clusters (Section 3.2), and (3) *inspecting failures* minimizes one failing task per cluster and finds duplicate failures to report new, unique bugs (since this step is currently manual, we do not discuss it in this section).

### 3.1 Collecting Failures

Figure 2 outlines the basic procedure for collecting failures. The procedure takes three inputs: the refactoring under test (RUT), a Java/C project containing the

```
1  collect_failures(refactoring, project, threshold):
2    elements = find_elements(refactoring, project)
3    for el in elements:
4      if is_reached(threshold): break
5      refactoring_tasks = create_refactoring_tasks(refactoring, project, el)
6      for task in refactoring_tasks:
7        configure_properties(task)
8        try:
9          if check_preconditions(task):
10           refactored_project = perform(task)
11         else: continue
12       except exc:
13         report("Failure: Refactoring threw an exception", exc, task)
14         continue
15       errors = compile(refactored_project)
16       if errors is not empty:
17         report("Failure: Refactored program failed check", errors, task)
18         continue
19       report("Success", task)
```

Fig. 2: Collecting failures for one given refactoring and project

program on which the refactoring will be applied, and a threshold that determines the maximum number of times to apply the RUT on the project. The procedure first finds the set of program *elements* in the given project on which the RUT can be applied and then computes a set of refactoring tasks for each element. For example, for Change Method Signature, the set of elements consists of all methods in the project, and the set of tasks can include changing method visibility, adding a parameter, removing a parameter, and reversing parameter order.

For each refactoring task, the procedure performs several steps in a loop. It first configures the properties for the refactoring task: in addition to the input project and program element, each refactoring can have a number of properties. For example, changing method visibility in Change Method Signature requires providing the new visibility: `private`, `protected`, `default`, or `public`. The specific property values depend on the particular refactoring task. For example, to actually change the method visibility we need to choose a new value for the visibility that differs from the old value, so different values can be provided for different refactoring tasks.

The procedure next checks if the refactoring task should proceed (line 9). In some cases the refactoring engine gives a warning that the refactoring could change the program behavior thus violating the definition that refactorings are behavior-preserving. For example, the refactoring engine could give a warning if we attempt to change visibility of a method to `private` when the method is called from outside its class. In those cases, our procedure does *not* proceed with the refactoring as checking the resulting program for compiler errors could produce many false positives because the problems do not arise from real bugs in the refactoring engine but from the ignored warnings. An alternative would be to proceed despite warnings but to check only whether the refactoring engine throws an exception and not whether the refactored program compiles.

```
 1 # the procedure maintains a set called "abstractions"
 2 # each abstraction maps a concrete message (exception or compiler error) to an abstract message
 3 # applying abstractions to a concrete message returns either an abstract message or "cannot abstract"
 4
 5 cluster_failures(refactorings, projects, threshold):
 6   # collect all failures
 7   failures = {} # empty set
 8   for rf in refactorings:
 9     for pj in projects:
10       failures += collect_failures(rf, pj, threshold)
11
12   # cluster all failures
13   all_abstract_messages = {} # empty set
14   for failure in failures:
15     failure.abstract_messages = {} # empty set
16     for c_msg in failure.concrete_messages:
17       if apply(abstractions, c_msg) = "cannot abstract":
18         if can_automatically_abstract_messages(): # JDT tool
19           abstractions += automatically_abstract_message(c_msg, failure.task)
20         else: # current CDT tool
21           abstractions += ask_user_for_abstraction(c_msg)
22       a_msg = apply(abstractions, c_msg)
23       failure.abstract_messages += a_msg
24       all_abstract_messages += a_msg
25
26   clusters = {} # empty set of sets of failures
27   for rf in refactorings:
28     for type in { exception, compiler }:
29       for a_msg in all_abstract_messages:
30         cluster = { f ∈ failures | f.refactoring = rf ∧ f.type = type ∧
31                                         a_msg ∈ f.abstract_messages }
32         clusters += cluster
```

Fig. 3: Clustering failures for several given refactorings and projects

The procedure then performs the program transformation. Note that both this action and the previous action (lines 9 and 10) execute the actual RUT code from the refactoring engine. If these actions result in an uncaught exception, the procedure records a failure with an exception message.

If the refactoring task produced a refactored project, the procedure checks whether the new project compiles (line 15). One could optionally check other oracles [7,39], e.g., whether the refactored project still passes all its tests [41]. If there are any compiler errors, the procedure records a failure with all the error messages. Note that when one refactored project does not compile, there can be multiple compiler error messages, whereas when the refactoring engine throws an exception, there is only one message with a stack trace.

## 3.2 Clustering Failures

Figure 3 shows the procedure that runs a set of refactorings on a set of projects (up to the maximum number of refactoring tasks per refactoring and project pair), collects the failures, and then clusters the failures (lines 12 to 24). The goal of clustering is to reduce the number of failures that should be inspected to detect *unique* bugs.

The clustering first computes *abstract messages* from the concrete messages that were recorded with the failures. Each failure corresponds to a refactoring task that either threw an exception during the refactoring or produced compiler error(s) on the refactored project. The concrete messages are the actual strings, e.g., "`The type new MultivariateFunction(){} must implement the inherited abstract method MultivariateFunction.value(double[])`" and "`The type FieldValueHitQueue<T>.OneComparatorFieldValueHitQueue<T> must implement the inherited abstract method PriorityQueue<T>.lessThan(Object, T, T)`". The goal of abstracting these messages is to form clusters of failures that are likely due to the same underlying bug.

The procedure maintains a set of abstraction functions, each of which maps a concrete message (exception or compiler error) to an abstract message. For exceptions, our tool currently maps a failure to the top stack frame from the stack trace. For compiler errors, the abstractions are *regular expressions*. Our tool for Eclipse JDT automatically creates a regular expression from an object representing a compiler error during Eclipse execution; to obtain the error object, our tool reruns the refactoring task (line 19) and replaces all the arguments (e.g., `new MultivariateFunction(){}`) of the error message with "`.*`". For example, it can create a regular expression "`The type .* must implement the inherited abstract method .*`". Our tool for Eclipse CDT currently requires the user to manually provide a set of such regular expressions; we do not have full automation because some messages are more project specific because they are output from `make` not just compiler errors. These regular expressions typically ignore the project-specific details such as identifiers, file names, or line/column numbers. For each error, the tool checks whether the error matches one of the regular expressions; if not, the user is asked to provide a new expression. If yes, the regular expression itself is used as the abstract message. For example, the two messages from the previous paragraph are both abstracted to the same abstract message from this paragraph. Note that many regular expressions can be reused across refactorings. Across all failing refactoring tasks in our experiments, we had 112 automatically generated regular expressions for Java and 50 manually written regular expressions for C; it takes under a minute to manually write one regular expression, and we did not find it to be a big burden.

After the messages are abstracted, the clustering splits the failures into groups that have the same refactoring name (ignoring options for the refactoring task), the same type of failure (either exception or compiler error), and contain the same abstract message. As a result, one failure can belong to multiple clusters (known as "overlapping clustering" or "multi-view clustering" [16]), i.e., if a failure has multiple compiler errors, it is put in all the clusters that correspond to these errors. The expectation is that failures in the same cluster are likely due to the same bug, and failures in different clusters are likely due to different bugs. (Note that one failure by itself may be due to several bugs.) The clustering splits the failures based on the refactoring name because the chance is lower that failures for different refactorings are caused by the same bug, although the chance is not zero as some refactorings share code. Likewise, the clustering keeps separate the clusters for exceptions and compiler errors because those clusters are unlikely to be caused by the same bug.

## 4 Implementation

This section describes how we implemented our approach for two refactoring engines in Eclipse—JDT [18] for Java and CDT [3] for C. While we implemented the approach only for Eclipse because of our familiarity with the infrastructure, the approach also applies to other IDEs.

### 4.1 Testing JDT

We implemented our tool as an Eclipse plug-in that supports testing all 23 refactorings available in the Eclipse refactoring menu (the first column of Figure 4)[5]. Our plug-in fully automates all the steps from figures 2 and 3 for the Eclipse JDT refactoring engine.

Our plug-in selects the set of relevant program elements for each refactoring based on the refactoring specification [32] (e.g., it selects methods for Change Method Signature). The second column of Figure 4 shows the precise set of elements that our tool selects by default. It selects only a subset of elements for some refactorings to match what was used in previous studies [7,12,38,39], e.g., for Rename these studies selected all fields, local variables, and methods but not types or packages. Our implementation offers a number of options that can select a superset or subset of the default set of elements, but our evaluation uses the default set.

Many refactorings have a number of properties that can be configured, e.g., for Convert Local Variable to Field the properties include: mark the field as final, mark the field as static, name for the field, location of initialization, and modifiers. By default our plug-in uses only one configuration of property values. Our experiments (Section 5) show that using one configuration suffices to find many new bugs in the current Eclipse refactoring engines; in the future, we plan to explore testing multiple configurations. The third column of Figure 4 lists the precise pairs `(property, value)` for all properties that our plug-in explicitly sets. For the properties that are not listed, our plug-in uses the default values that Eclipse provides. We select the values such that refactorings are likely to proceed and not raise warnings about violated preconditions, e.g., we rename a program elements to a name that is new to the project rather than some existing, conflicting name.

The main loop of our plug-in executes refactoring tasks and checks the results. These operations would be very slow if implemented naively by first creating a new Eclipse Java project for each refactoring task, then populating this project with the source code under test, refactoring the code, and compiling the entire refactored project to check for compiler errors. Our plug-in provides two important optimizations. First, it does not create a new Eclipse Java project for

[5] The order of the refactorings matches the order in the Eclipse refactoring menu.

| Refactoring | Elements | (Property, Value) |
|---|---|---|
| Rename | fields$^{\mathcal{F}}$<br>local variables$^{\mathcal{L}}$<br>methods$^{\mathcal{M}}$ | $^{\mathcal{F},\mathcal{L},\mathcal{M}}$new name, non-conflicting<br>$^{\mathcal{L},\mathcal{M}}$update references, `true` |
| Move | instance methods$^{\mathcal{IM}}$<br>static methods$^{\mathcal{SM}}$ | $^{\mathcal{IM},\mathcal{SM}}$delegate updating, `true`<br>$^{\mathcal{IM},\mathcal{SM}}$deprecate delegates, `true`<br>$^{\mathcal{IM}}$inline delegator, `true`<br>$^{\mathcal{IM}}$use getter/setters, `true`<br>$^{\mathcal{IM}}$target, non-primitive parameter types<br>$^{\mathcal{SM}}$target, previous type in lexicographical order |
| Change Method Signature | methods$^{\mathcal{M}}$<br>parameters$^{\mathcal{P}}$ | $^{\mathcal{M}}$visibility, `private`<br>$^{\mathcal{P}}$default value for added, `null`<br>$^{\mathcal{P}}$add/remove position, 0<br>$^{\mathcal{P}}$new order, reverse |
| Extract Method | expressions | new name, non-conflicting<br>visibility, `public`<br>replace duplicates, `true` |
| Extract Local | non-void expressions | declare final, `true`<br>replace all occurrences, `true`<br>new name, non-conflicting |
| Extract Constant | literals<br>exp.s with literals<br>method invocations | replace all occurrences, `true`<br>visibility, `public`<br>qualify references, `true` |
| Inline | constants$^{\mathcal{C}}$<br>local variables<br>methods$^{\mathcal{M}}$ | $^{\mathcal{C}}$remove declaration, `true`<br>$^{\mathcal{M}}$delete source, `true` |
| Convert Local To Field | local variables | - |
| Convert Anonymous | anonymous classes | new name, non-conflicting<br>declare static, `true` |
| Move Type To New File | non-local types | - |
| Extract Superclass | top level classes | create method stubs, `true`<br>instanceof, `true`<br>delete methods, `true`<br>elements, all public methods |
| Extract Interface | types | annotations, `true`<br>visibility, `public`<br>replace, `true` |
| Use Supertype | types | destination, all supertypes |
| Push Down | types | element to push, a member |
| Pull Up | types | use keyword this, `true`<br>override annotation, `true`<br>destination, a supertype<br>element to pull, a member |
| Extract Class | fields | create getter/setter, `true` |
| Introduce Param. Object | methods | top level, `false` |
| Introduce Indirection | methods | update references, `true` |
| Introduce Factory | methods | protect constructor, `true` |
| Introduce Parameter | expressions | - |
| Encapsulate Field | fields | visibility, `public`<br>encapsulate declaring class, `true` |
| Generalize Declared Type | types | destination, a supertype |
| Infer Type Arguments | compilation units | - |

Fig. 4: Default set of elements and (property, value) pairs for JDT

| Refactoring | Elements | (Property, Value) |
|---|---|---|
| Rename | global variables<br>local variables<br>function parameters<br>functions<br>structure members<br>macros | new name, non-conflicting |
| Extract Function | expressions<br>single statements | new name, non-conflicting |
| Extract Local Variable | expressions | new name, non-conflicting |
| Extract Constant | literals | new name, non-conflicting |
| Toggle Function | functions | create header, `true` |

Fig. 5: Default set of elements and (property, value) pairs for CDT

each task; instead, it creates one Eclipse Java project for the first task and then, after applying the refactoring and checking the results, it undoes the refactoring to restore the original project state. Undoing the refactoring is over an order of magnitude faster than creating a new Eclipse project. Currently, we rely on the Eclipse implementation of undo refactoring. However, as this implementation may be incorrect by itself, we could optionally copy the project to check the undo and to ensure that the project under refactoring is consistent before each refactoring task. Second, the plug-in does not compile the entire refactored project but only focuses on the file that contains the program element being refactored. Although this optimization significantly improves the performance, it may lead to false negatives as compiler errors may be in other affected files or the files that depend on the affected files. Our plug-in could be easily configured to compile the entire project after each refactoring task.

### 4.2 Testing CDT

Our implementation for Eclipse CDT, which targets the C and C++ programming languages, is similar to the implementation for JDT. For CDT we also implemented an Eclipse plug-in that supports all five C-specific refactorings available in CDT. We tested Eclipse 4.2.1 and CDT 8.1.1 (the Juno SR1 release of both Eclipse and CDT) [21].

Similar to our plug-in for testing the JDT refactorings, the set of relevant program elements for the CDT refactorings was derived from the refactoring specification. The default sets of elements and the default configurations for the refactorings are shown in Figure 5.

## 5 Evaluation

Our main goal was to evaluate how our proposed end-to-end approach helps in testing refactoring engines and estimating their reliability. This section describes

| | Subject | Description | Version | LOC |
|---|---|---|---|---|
| Java | JPF [19] | Model checking tool | hg:960 | 95962 |
| | JUnit [20] | Unit testing framework | git:r4.8.1-408-ge8b91fa | 18199 |
| | log4j [23] | Logging framework | svn:1406847 | 30058 |
| | Lucene [24] | Text search engine library | 3.5.0 | 129820 |
| | Math [26] | Library of mathematics components | 3.3.0 | 120424 |
| C | GMP [13] | Arbitrary precision arithmetic library | 4.3.2 | 81900 |
| | libpng [22] | Official PNG reference library | 1.2.6 | 33908 |
| | zlib [51] | Lossless data-compression library | 1.2.5 | 19855 |

Fig. 6: Subject programs used in the experiments

the projects that we used for testing Eclipse JDT and CDT refactoring engines, the failures that were collected, the clusters that were created, and the bugs that we reported to Eclipse Bugzilla.

## 5.1 Projects Under Refactoring

Figure 6 shows the Java and C projects that we use in our evaluation. We tabulate the project name and the reference from which the project was obtained, a brief description of each project, version/revision number, and the number of (non-comment, non-blank) lines of code. We selected these projects because we were familiar with them, and they provide a diverse set of projects of various sizes (#LOC, #classes, #methods) and using different programming language features and design styles. For example, JUnit is a representative of a highly modular object-oriented design, whereas Math has a large number of local variables and constants.

## 5.2 Failure and Clustering Statistics

Figures 7 and 9 show the execution statistics from applying refactorings (in configurations from figures 4 and 5) on the selected set of projects for Java and C, respectively. For each refactoring and project, we tabulate the number of times that the refactoring is applied, the number of failures, and the total execution time (which includes finding the places where to apply the refactoring, applying the refactoring, and checking the refactored project).

**JDT Results** We ran all JDT experiments on a 64-core Intel Xeon CPU L7555 @ 1.87GHz with 64GB of main memory, running Oracle Java version 1.7.0_04. In all runs we set the maximum number of refactoring tasks per file to 100 to limit the execution time. The runs still took over 200hrs of machine time overall.

The bottom of Figure 7 shows the ratio of the total number of failures and the total number of refactoring tasks applied on each project. The maximum ratio of 2.0% indicates that the Eclipse JDT refactoring engine is quite reliable, but there is still space for improvements.

| | JPF | | | JUnit | | | log4j | | | Lucene | | | Math | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Refactoring | #Refact. Tasks | #Failures | Exec. Time [min] | #Refact. Tasks | #Failures | Exec. Time [min] | #Refact. Tasks | #Failures | Exec. Time [min] | #Refact. Tasks | #Failures | Exec. Time [min] | #Refact. Tasks | #Failures | Exec. Time [min] |
| Rename | 20024 | 10 | 760 | 4025 | 8 | 69 | 5320 | 2 | 92 | 25897 | 24 | 626 | 26447 | 48 | 489 |
| Move | 1557 | 0 | 62 | 246 | 1 | 6 | 438 | 0 | 9 | 750 | 4 | 42 | 507 | 0 | 23 |
| Change Method Signature | 5021 | 289 | 429 | 5286 | 30 | 125 | 4261 | 65 | 129 | 6146 | 124 | 516 | 7812 | 57 | 440 |
| Extract Method | 26405 | 182 | 156 | 6183 | 28 | 27 | 8770 | 72 | 32 | 39468 | 152 | 206 | 39177 | 234 | 265 |
| Extract Local | 34834 | 0 | 245 | 4460 | 0 | 22 | 11020 | 0 | 46 | 50278 | 0 | 353 | 50796 | 0 | 393 |
| Extract Constant | 16752 | 0 | 183 | 2355 | 0 | 13 | 5444 | 0 | 26 | 26965 | 0 | 245 | 29997 | 0 | 257 |
| Inline | 18446 | 180 | 329 | 3745 | 60 | 41 | 4853 | 34 | 48 | 23499 | 569 | 323 | 21978 | 662 | 174 |
| Convert Local To Field | 7605 | 1 | 58 | 1041 | 0 | 4 | 2307 | 4 | 7 | 13802 | 8 | 58 | 2934 | 0 | 17 |
| Convert Anonymous | 146 | 1 | 4 | 143 | 0 | 1 | 7 | 0 | 0 | 293 | 38 | 3 | 314 | 5 | 5 |
| Move Type To New File | 198 | 19 | 10 | 440 | 5 | 9 | 82 | 3 | 1 | 550 | 22 | 24 | 165 | 2 | 5 |
| Extract Superclass | 248 | 0 | 74 | 419 | 0 | 11 | 87 | 0 | 2 | 232 | 0 | 24 | 590 | 0 | 43 |
| Extract Interface | 1251 | 237 | 69 | 730 | 31 | 15 | 332 | 43 | 7 | 1365 | 725 | 51 | 1122 | 121 | 29 |
| Use Supertype | 940 | 44 | 48 | 257 | 4 | 3 | 278 | 0 | 3 | 1195 | 24 | 31 | 814 | 13 | 19 |
| Push Down | 2666 | 79 | 353 | 387 | 11 | 9 | 529 | 7 | 8 | 2932 | 72 | 328 | 2001 | 14 | 51 |
| Pull Up | 3146 | 34 | 145 | 692 | 14 | 9 | 1774 | 2 | 11 | 5550 | 25 | 216 | 1657 | 87 | 40 |
| Extract Class | 61 | 2 | 1 | 363 | 45 | 2 | 92 | 15 | 1 | 221 | 22 | 4 | 2033 | 490 | 48 |
| Introduce Param. Object | 4590 | 114 | 1258 | 2421 | 105 | 73 | 2036 | 30 | 83 | 5185 | 233 | 515 | 7793 | 519 | 743 |
| Introduce Indirection | 4418 | 30 | 170 | 1813 | 4 | 37 | 1248 | 2 | 28 | 4835 | 41 | 200 | 4768 | 44 | 112 |
| Introduce Factory | 776 | 53 | 9 | 197 | 7 | 2 | 297 | 9 | 2 | 1069 | 62 | 13 | 1140 | 100 | 17 |
| Introduce Parameter | 895 | 359 | 12 | 2711 | 298 | 64 | 1711 | 110 | 42 | 2365 | 556 | 64 | 1721 | 194 | 44 |
| Encapsulate Field | 2749 | 37 | 50 | 563 | 3 | 6 | 977 | 71 | 11 | 4309 | 109 | 71 | 2885 | 10 | 41 |
| Generalize Declared Type | 2734 | 314 | 280 | 1285 | 157 | 15 | 2506 | 544 | 24 | 1240 | 144 | 37 | 2638 | 65 | 35 |
| Infer Type Arguments | 886 | 8 | 8 | 330 | 1 | 2 | 250 | 0 | 0 | 817 | 10 | 5 | 970 | 72 | 10 |
| $\sum$ | 156348 | 1993 | 4713 | 40092 | 812 | 565 | 54619 | 1013 | 612 | 218963 | 2964 | 3955 | 210259 | 2737 | 3300 |
| #Failures/#Refact. Tasks | 1.3% | | | 2.0% | | | 1.9% | | | 1.4% | | | 1.3% | | |

Fig. 7: Execution statistics of our JDT plug-in on a set of Java projects

| | | | Compiler Error | | | | | Exception | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Refactoring | #Refact. Tasks | #Failures | #Failures | #Clusters | Min Cluster | Max Cluster | #Bugs | #Failures | #Clusters | Min Cluster | Max Cluster | #Bugs |
| Rename | 81713 | 92 | 90 | 5 | 1 | 54 | ‡6 | 2 | 1 | 2 | 2 | 1 |
| Move | 3498 | 5 | 5 | 2 | 1 | 4 | 1 | 0 | 0 | - | - | 0 |
| Change Method Signature | 28526 | 565 | 555 | 10 | 1 | 526 | 8 | 10 | 2 | 4 | 6 | 2 |
| Extract Method | 120003 | 668 | 665 | 14 | 1 | 390 | 5 | 3 | 1 | 3 | 3 | 0 |
| Extract Local | 151388 | 0 | 0 | 0 | - | - | 0 | 0 | 0 | - | - | 0 |
| Extract Constant | 81513 | 0 | 0 | 0 | - | - | †1 | 0 | 0 | - | - | 0 |
| Inline | 72521 | 1505 | 1475 | 42 | 1 | 790 | 12 | 30 | 3 | 2 | 23 | 0 |
| Convert Local To Field | 27689 | 13 | 13 | 9 | 1 | 5 | 2 | 0 | 0 | - | - | 0 |
| Convert Anonymous | 903 | 44 | 29 | 8 | 1 | 10 | 2 | 15 | 1 | 15 | 15 | 0 |
| Move Type To New File | 1435 | 51 | 50 | 22 | 1 | 18 | 5 | 1 | 1 | 1 | 1 | 0 |
| Extract Superclass | 1576 | 0 | 0 | 0 | - | - | 0 | 0 | 0 | - | - | 0 |
| Extract Interface | 4800 | 1157 | 1143 | 16 | 1 | 725 | 4 | 14 | 1 | 14 | 14 | 0 |
| Use Supertype | 3484 | 85 | 85 | 21 | 1 | 16 | 6 | 0 | 0 | - | - | 0 |
| Push Down | 8515 | 183 | 183 | 11 | 1 | 121 | 6 | 0 | 0 | - | - | 0 |
| Pull Up | 12819 | 162 | 45 | 10 | 1 | 23 | 3 | 117 | 2 | 3 | 114 | 0 |
| Extract Class | 2770 | 574 | 574 | 16 | 1 | 275 | 3 | 0 | 0 | - | - | 0 |
| Introduce Param. Object | 22025 | 1001 | 839 | 15 | 1 | 455 | 2 | 162 | 4 | 2 | 140 | 0 |
| Introduce Indirection | 17082 | 121 | 72 | 7 | 4 | 31 | 1 | 49 | 3 | 4 | 32 | 2 |
| Introduce Factory | 3479 | 231 | 231 | 7 | 1 | 223 | 3 | 0 | 0 | - | - | 0 |
| Introduce Parameter | 9403 | 1517 | 0 | 0 | - | - | 0 | 1517 | 2 | 1 | 1516 | ‡3 |
| Encapsulate Field | 11483 | 230 | 212 | 10 | 1 | 94 | 8 | 18 | 1 | 18 | 18 | 0 |
| Generalize Declared Type | 10403 | 1224 | 1176 | 22 | 1 | 339 | 8 | 48 | 3 | 6 | 26 | 1 |
| Infer Type Arguments | 3253 | 91 | 7 | 7 | 1 | 3 | 1 | 84 | 2 | 83 | 84 | 2 |
| $\sum$ | 680281 | 9519 | 7449 | 254 | | | **87** | 2070 | 27 | | | **11** |
| #Failures/#Refact. Tasks | 1.4% | | | | | | | | | | | |

Fig. 8: Failure and cluster statistics for Java projects. (The number of bugs is likely higher; while we minimized one failure from each of 281 clusters, we have not checked duplicates for 141 minimized failures.) †The refactoring implements too strong precondition. ‡We reported two bugs that had the same stack trace but result in different compiler errors in the latest version of Eclipse.

Figure 8 shows additional statistics about failures. The column "#Refact. Tasks" shows the number of refactoring tasks performed across all the projects, and the column "#Failures" shows the total number of failures. The next two groups of columns split the results for the failures that have compiler errors or exceptions. Each group tabulates the number of failures, the number of clusters, the minimum and maximum sizes of clusters (measured by the number of failures), and the number of bugs we found based on these clusters.

| Refactoring | GMP #Refact. Tasks | GMP #Failures | GMP Exec. Time [min] | libpng #Refact. Tasks | libpng #Failures | libpng Exec. Time [min] | zlib #Refact. Tasks | zlib #Failures | zlib Exec. Time [min] |
|---|---|---|---|---|---|---|---|---|---|
| Rename | 6688 | 555 | 739 | 2395 | 43 | 118 | 1569 | 0 | 9 |
| Extract Method | 16742 | 1176 | 579 | 5092 | 548 | 172 | 1496 | 58 | 24 |
| Extract Local | 5893 | 363 | 1240 | 2660 | 554 | 65 | 3473 | 406 | 59 |
| Extract Constant | 18788 | 387 | 902 | 2208 | 142 | 62 | 2800 | 331 | 28 |
| Toggle Function | 1434 | 403 | 10 | 302 | 293 | 2 | 332 | 167 | 1 |
| $\sum$ | 49545 | 2884 | 3470 | 12657 | 1580 | 419 | 9670 | 962 | 121 |
| #Failures/#Refact. Tasks | 5.8% | | | 12.4% | | | 9.9% | | |

Fig. 9: Execution statistics of our CDT plug-in on a set of C projects

We believe that the total of 680281 refactoring tasks cover a diverse spectrum of refactoring tasks and allow us to identify bugs that can be encountered in practice. The total number of failures is 9519, which may look large but is a relatively small fraction of the total number of refactoring tasks.

These failures are split into a total of 281 clusters: 254 compiler error clusters and 27 exception clusters. Clusters vary in size from 1 to 1516 failures, with the median and mean of 5 and 40.3, respectively. Recall that the cluster size can be used to prioritize inspection and/or fixing of bugs, and the same failure can appear in multiple clusters. For example, consider the two exception clusters for Infer Type Arguments. One cluster has all 84 failures, and the other cluster has 83 failures. It means that 83 failures have two messages each, and one failure has only one message (that abstracts to the same abstract message as one of the two messages from the other failures).

**CDT Results** We ran all CDT experiments on an Intel Xeon Quad Core CPU X3440 @ 2.53GHz with 16GB of main memory.

For Rename, we run refactoring tasks on all C files in a given project. For each file, we attempt to rename at most 50 local variables, 50 function parameters, 20 global variables, 20 function names, and 20 macros. Across all three projects, the Rename refactorings run for a total of 866 minutes—overall, 10652 refactorings are attempted with 598 failures. Of these failures, 42% are compiler errors, while 58% are exception failures. We find a larger percentage of exceptions in CDT, presumably because it is less stable than JDT.

For Extract Function, we attempt to extract at most 100 statements and 100 expressions per C file. Out of 23330 attempts, 1782 fail, with 1453 compiler error failures and 329 exception failures. The total run takes 775 minutes. For Extract local Variable, Extract Constant, and Toggle Function, we attempt to extract at most 100 expressions, literals, and functions per C

| | | | Compiler Error | | | | | Exception | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Refactoring | #Refact. Tasks | #Failures | #Failures | #Clusters | Min Cluster | Max Cluster | #Bugs | #Failures | #Clusters | Min Cluster | Max Cluster | #Bugs |
| Rename | 10652 | 598 | 252 | 3 | 1 | 173 | 2 | 346 | 2 | 3 | 343 | 0 |
| Extract Method | 23330 | 1782 | 1453 | 34 | 1 | 435 | 21 | 329 | 8 | 1 | 56 | 3 |
| Extract Local | 12026 | 1323 | 754 | 11 | 3 | 412 | 7 | 569 | 4 | 6 | 263 | 3 |
| Extract Constant | 23796 | 860 | 142 | 3 | 1 | 84 | 2 | 718 | 3 | 125 | 426 | 2 |
| Toggle Function | 2068 | 863 | 9 | 2 | 1 | 8 | 1 | 854 | 5 | 23 | 409 | 2 |
| $\sum$ | 71872 | 5426 | 2610 | 53 | | | **33** | 2816 | 22 | | | **10** |
| #Failures/#Refact. Tasks | 7.5% | | | | | | | | | | | |

Fig. 10: Failure and cluster statistics for C projects

file, respectively. Across all refactorings, libpng has the highest failure rate with 12.4%, followed by zlib and GMP.

While we only check that the refactored program compiles, one can use other oracles [7, 39]. For example, for the Rename refactoring on GMP, we ran tests (‘`make test`’) in addition to compilation (‘`make`’). However, this did not produce any extra errors and was taking too much time, so we did not run tests for other cases. In the future, we plan to evaluate our approach with other oracles.

Figure 10 shows additional statistics about failures. We had 75 clusters in total: 53 compiler errors clusters and 22 exception clusters. These clusters vary in size from 1 to 435, with the median and mean of 27 and 79.5, respectively.

## 5.3 Bugs

After clustering all the failures, we inspected one, randomly selected failure from each of 281+75 clusters. We first minimized the project under refactoring such that the failure is preserved in the minimized version. We performed minimization manually, which took between a few minutes and 1hr, with the average of around 10min. In the future, we plan to evaluate the existing automated minimization techniques [27, 33, 50].

After we prepared a minimized version, we want to check whether it is a new, unique bug. We compared the minimized version with the other bugs that we found and also searched through the Eclipse Bugzilla database to ensure that the bug we found had not been reported before. This search for duplicates is also performed manually and took on average 15min per bug. (So far we have performed the search for 140 of 281 clusters for JDT and all 75 clusters for CDT.) Our goal was to report as few duplicates as possible, and we found it somewhat harder to search for duplicates than to minimize the project. One could consider searching for duplicates directly from failures, even before minimization, but our experience showed that the result is not obtained faster, e.g., searching based purely on compiler errors does not provide a good result. The existing techniques [1,40,47] for searching duplicate bug reports mostly use natural language processing and do not focus on searching for programs that are inputs to refactoring engines. We leave it as future work to explore automated search for duplicates in this context. We point out that our study is the first to raise this concern; previous related studies [7,12,17,35,38,39,41,44] did not report the effort for inspecting duplicates, presumably because (1) they found a smaller number of bugs than we found, (2) the number of bug reports for Eclipse was smaller at the time when they searched for duplicates than it was when we searched for duplicates, and/or (3) they did not search for duplicates.

So far we have reported a total of 77 bugs in JDT and 43 bugs in CDT. Each report includes the minimized example on which the bug can be reproduced. Our work is ongoing; we have 141 more Java minimized examples to check for duplicates and plan to run our tool for more projects in the future. The updated list of our reports is available online: `http://mir.cs.illinois.edu/rtr`.

**Java Results** Figure 11 shows summary information about the bugs we have reported for Eclipse JDT so far. The first column lists the refactorings. The next column lists how many of the bugs we found are likely duplicates (either of previously reported bugs in Bugzilla or among our own clusters), which we did not report. The next group of columns lists how many reports we submitted and the current status of those reports in Bugzilla (`NEW` - the bug is reported but not yet considered by the Eclipse developers, `ASSIGNED` - the bug is confirmed and assigned to a developer, `FIXED` - the bug is fixed, and `DUPLICATE` - the bug is marked as a duplicate by the Eclipse developers).

The last row of the table summarizes the results: of the 77 bugs reported, 8 were already fixed, 62 assigned as real bugs, 4 marked as duplicates, and the rest were not inspected. We have noticed that the developers were more responsive if a reported bug causes an exception rather than a compiler error.

The remaining groups of columns for Java show the results for reproducing in two other IDEs, specifically NetBeans and IntelliJ IDEA, the bugs we found in Eclipse. The goal is to find how many of these bugs appear in one IDE but not another. Presumably the developers of an IDE may want to prioritize more the bugs that are unique to their IDE. We attempted to run on NetBeans and IntelliJ each minimized example that we used as part of our bug reports for Eclipse. We find that some of the bugs from Eclipse do not apply in other IDEs (e.g., because they do not have an equivalent refactoring or have too strong preconditions for the refactoring). Of the bugs that do apply, some are shared between independent implementations of refactorings, 24 between Eclipse and NetBeans, and 26 between Eclipse and IntelliJ (although not the same as NetBeans). A number of bugs (12) is shared even for all three IDEs. Of the bugs that could potentially apply, a number of bugs from Eclipse do not appear in the other IDEs. Note that this does not imply that the other IDEs are more reliable as we did not evaluate their bugs on Eclipse. Indeed, our goal was to find how bugs we found are shared among refactoring engines rather than to compare IDEs.

| | Java | | | | | | | | | C | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Eclipse | | | | | | NetBeans | IntelliJ | | | VAX | XRef |
| | | | Bugzilla | | | | | | | | | |
| Refactoring | #Likely Duplicate | #Reported | #`NEW` | #`ASSIGNED` | #`FIXED` | #`DUPLICATE` | Reproducible? yes/no/na | Reproducible? yes/no/na | Ecl.∩NetB.∩In.J | #`NEW` Eclipse | Reproducible? yes/no/na | Reproducible? yes/no/na |
| Rename | 2 | 5 | 2 | 3 | 0 | 0 | 0/5/0 | 1/4/0 | 0 | 2 | 1/1/0 | 0/0/2 |
| Move | 0 | 1 | 0 | 1 | 0 | 0 | 1/0/0 | 0/1/0 | 0 | - | - | - |
| Change Meth. Sig. | 7 | 3 | 1 | 1 | 1 | 0 | 0/2/1 | 0/2/1 | 0 | - | - | - |
| Extract Method | 0 | 5 | 0 | 4 | 1 | 0 | 4/1/0 | 2/3/0 | 2 | 24 | 17/7/0 | 5/2/17 |
| Extract Local | 0 | 0 | 0 | 0 | 0 | 0 | 0/0/0 | 0/0/0 | 0 | 10 | 0/0/10 | 0/0/10 |
| Extract Constant | 0 | 1 | 0 | 1 | 0 | 0 | 0/0/1 | 0/1/0 | 0 | 4 | 0/0/4 | 0/0/4 |
| Toggle Function | - | - | - | - | - | - | - | - | - | 3 | 0/0/3 | 0/0/3 |
| Inline | 5 | 7 | 0 | 7 | 0 | 0 | 4/3/0 | 4/3/0 | 3 | - | - | - |
| Loc. To Field | 0 | 2 | 0 | 2 | 0 | 0 | 1/0/1 | 0/2/0 | 0 | - | - | - |
| Anon. To Nested | 0 | 2 | 0 | 2 | 0 | 0 | 1/1/0 | 0/2/0 | 0 | - | - | - |
| Move Type | 3 | 2 | 0 | 2 | 0 | 0 | 0/2/0 | 1/1/0 | 0 | - | - | - |
| Extract Superclass | 0 | 0 | 0 | 0 | 0 | 0 | 0/0/0 | 0/0/0 | 0 | - | - | - |
| Extract Interface | 0 | 4 | 0 | 4 | 0 | 0 | 0/1/3 | 3/1/0 | 0 | - | - | - |
| Use Supertype | 1 | 5 | 0 | 4 | 0 | 1 | 2/2/1 | 2/3/0 | 1 | - | - | - |
| Push Down | 2 | 4 | 0 | 3 | 0 | 1 | 3/1/0 | 1/3/0 | 1 | - | - | - |
| Pull Up | 0 | 3 | 0 | 2 | 1 | 0 | 2/1/0 | 2/1/0 | 2 | - | - | - |
| Extract Class | 0 | 3 | 0 | 3 | 0 | 0 | 0/0/3 | 0/0/3 | 0 | - | - | - |
| Intro. Param. Obj. | 0 | 2 | 0 | 2 | 0 | 0 | 0/2/0 | 1/1/0 | 0 | - | - | - |
| Intro. Indirection | 0 | 3 | 0 | 3 | 0 | 0 | 0/0/3 | 0/0/3 | 0 | - | - | - |
| Intro. Factory | 0 | 3 | 0 | 3 | 0 | 0 | 3/0/0 | 1/2/0 | 1 | - | - | - |
| Intro. Param. | 0 | 3 | 0 | 1 | 0 | 2 | 1/2/0 | 1/2/0 | 0 | - | - | - |
| Encapsulate Field | 1 | 7 | 0 | 5 | 2 | 0 | 2/5/0 | 5/2/0 | 2 | - | - | - |
| Gen. Decl. Type | 0 | 9 | 0 | 8 | 1 | 0 | 0/0/9 | 2/6/1 | 0 | - | - | - |
| Infer Gen. Type | 0 | 3 | 0 | 1 | 2 | 0 | 0/0/3 | 0/3/0 | 0 | - | - | - |
| $\sum$ | 21 | **77** | 3 | 62 | 8 | 4 | 24/28/25 | 26/43/8 | 12 | **43** | 18/8/17 | 5/2/36 |

Fig. 11: Number of bugs for each refactoring from the Eclipse refactoring menu

**Anecdotal Experience** We found out that even duplicate reports can help developers, confirming some published results [1]. After inspecting a failure from one of the exception clusters, we discovered that the bug had been reported previously. However, the original bug reporter explicitly stated being unable to create a small example that causes the exception. After we added our minimized example to the bug report, it was fixed within a day (by adding one line and updating one line), more than 4 years after the bug had been originally reported.

We also found out that some refactorings are quite reliable. As can be seen from Figure 11, we did not report any new bug for the Extract Local Variable and Extract Superclass refactorings. Suspecting that our configurations (Section 4.1) may be incorrect for these refactorings, we modified the configurations and rerun the refactorings but still did not find any failure. In the future, we would like to explore many more additional configurations for these refactorings and to investigate the implementation of these refactorings to identify the reasons for their reliability.

**C Results** Figure 11 (rightmost columns) shows summary information about the bugs we have reported for Eclipse CDT so far. The "Eclipse" column lists the number of bugs, and they are all still marked `NEW` in Bugzilla. We tried to manually reproduce the 26 Rename and Extract Function CDT bugs in two other refactoring engines: Visual Assist X (VAX) [46] and XRefactory [49]. The other three refactorings are not supported in these refactoring engines. VAX is a plugin that provides refactorings for Visual Studio; XRefactory is a plugin for Emacs, xEmacs, and jEdit. We used VAX running on Visual Studio 2008 and XRefactory version 2.0.14 running on Emacs version 24.1.

Of the two Rename bugs in CDT, one was about renaming functions in system/external libraries (e.g., `printf`), and the other one was about renaming a macro in a file that has been declared or used in another file. VAX successfully handled the first case, but failed in the second case. The bugs were not applicable to XRefactory, which did not make any changes on the given inputs. Interestingly, XRefactory does not allow renaming of any function except `main`. This is obviously a bug, but not exactly the one that we identified in CDT. Hence, we marked this case as not applicable.

Of the 24 bugs in Extract Function, 17 could be reproduced in VAX. Of these bugs, 8 produced the same outputs as CDT after extraction. The remaining 9 produced outputs that were different from CDT outputs, but they were incorrect too. 6 failures were related to incorrect return type of the functions. For example, when a user attempts to extract an assignment expression with VAX, the extracted function has a boolean return type, even if the assignment's type is not boolean. CDT also introduces incorrect return type for an Extract Function refactoring: it incorrectly determines the return type of a pointer variable to be a non-pointer variable of that type.

We could not apply 17 out of 24 Extract Function bugs in XRefactory. Most of them (16 out of 17) were about applying Extract Function on expressions, whereas XRefactory only allows extraction on statements. Another case was about extracting multiple configurations of the C preprocessor; in this case, XRefactory did nothing. Among the remaining 7 cases that were applicable, 5 were buggy. These cases failed while attempting to extract a statement related to a macro definition, a statement containing a variadic function, or a goto statement to a function.

We also looked at the quality of our clustering approach. In GMP, we found one duplicate compiler error bug across 6 different clusters and another duplicate compiler error bug across other 3 different clusters. We also found 4 different compiler error bugs within one cluster. Predictably, there are more bugs in the extract refactorings, because they are more complex. The search for duplicates is easier in CDT as it has fewer overall bugs reported for it. It is also less actively developed, so we did not see any change in Bugzilla for our CDT bug reports.

**Application to OpenRefactory/C** We are actively integrating the approach described in this paper to test OpenRefactory/C [30], a new refactoring engine for C that we are developing. The approach is established as an essential part of developer testing: we are accumulating a set of well-known, open-source C projects, and refactorings are not eligible to be deemed “production quality” until they have successfully passed on those projects.

To use our approach for continuous developer testing, we have found it helpful to keep the number of tests relatively small (e.g., 50–100), at least during the early stages of testing. This typically produces just a few clusters, which the developer can investigate and fix immediately. Test results are persisted outside of Eclipse runs (currently in a database), which allows the developer to re-run failed tests after fixing a bug, or to continue running tests where a previous test run stopped. At the time of this writing, only one refactoring—Rename—in OpenRefactory/C is mature enough to be continously tested using this approach. Applying it to GMP, libpng, and zlib has already identified seven bugs: four bugs in the Rename refactoring itself (one unexpected exception) and three bugs in the supporting infrastructure (including one unexpected exception).

**Use Frequency vs. Failure Rate** Several researchers have studied the frequency of refactoring use in Java IDEs [6, 8, 28, 45] and ranked Rename, Extract Local Variable, Inline, Extract Method, and Move as the top five most commonly performed automated refactorings in practice. It would be reasonable to expect the most commonly used refactorings to have fewer bugs and thus a smaller failure rate. However, our study does not find a very strong correlation between the frequency of use and failure rate of a refactoring. While the top five used refactorings are also among the most reliable refactorings, the ordering based on the failure rate does not perfectly match the ordering based on the use frequency. Also, we find that Extract Superclass, which is almost unused in practice (its share reported as less than 0.2%), is one of the most reliable refactorings; on the other hand, Extract Interface is the least reliable refactoring that has some number of real uses (reported as around 0.5%). We believe that the frequency of refactoring use should be a factor that aids in ranking the importance of bug reports.

## 6 Related Work

Testing refactoring engines requires input programs, which are rather complex test inputs. Programs can be represented as data structures such as abstract syntax trees (ASTs). Automated systematic generation of complex data structures has been proposed a while ago [2, 25], but only more recently Daniel et al. [7] proposed a systematic technique, called ASTGen, for testing refactoring engines. ASTGen requires the user to write imperative generators that can build parts of Java programs and offers a new approach to combine these generators. Given these generators, ASTGen systematically generates a large number of (small) Java programs described with the generators. Although ASTGen exposed several bugs in Eclipse and NetBeans, it comes at the cost of writing the imperative generators, requires skillful users, and may not be adequate for describing some properties of complex inputs. We proposed UDITA [12], a non-deteministic language that enables the users to combine imperative generators with declarative filters to describe a set of complex test inputs in a concise way. Like ASTGen, UDITA requires rather sophisticated users. More recently Soares et al. [38, 39] follow an approach similar to UDITA and combine it with random testing to search for semantic changes introduced by refactorings. In contrast to these approaches, the approach presented in this paper introduces an end-to-end approach for testing refactoring engines on existing projects: applying refactorings on a number of existing projects, clustering failures, minimizing failing inputs, and detecting duplicate bugs. Our approach avoids the effort of writing test generators and increases the confidence that the bugs found are more important. However, our approach requires minimization of programs, which for now we perform manually; in the future, we plan to evaluate automated minimization approaches [27, 33, 50].

Applying refactorings on real programs has been explored by several researchers in different contexts. Spinellis [41] tested the Rename refactoring of his CScout refactoring engine on the Linux kernel source by systematically applying CScout to replace all identifiers with mechanically derived names and testing the correctness of the refactored code by checking that it compiles correctly. Thies and Steimann [44] tested two Eclipse refactorings, Move Class and Pull Up Method, by systematically applying them on existing open-source projects. Schäfer et al. [37] applied a few refactorings on more than million lines of open-source projects to investigate scalability of their refactoring implementation. Cinneide et al. [4] automatically applied refactorings on a number of projects as part of evaluating and comparing software metrics. Coker and Hafiz [5] tested three program transformations that fix C integer problems (signedness, overflow, underflow, and widthness problems) by applying these transformations on real C programs. In constrast, our paper evaluates an end-to-end approach for testing refactoring engines and evaluating their reliability on all refactorings in both Eclipse JDT and Eclipse CDT, and points out some key challenges in the process from applying refactorings to finding bugs, e.g., minimizing failing inputs and finding duplicate bugs.

Jagannath et al. [17] proposed clustering based on abstract messages of failures obtained by refactoring small corner-case programs. Our paper evaluates clustering on failures in real programs (there is no a priori reason to believe that a technique that works for small artificial corner cases also works well for real failures), applies it to the C language, and automates the abstraction of messages for JDT. Clustering has been also used for determining which program runs result in failures; most recently, Sun et al. [43] proposed a cost-sensitive strategy for inspecting clusters of program runs. In our approach, the outcome of each test is known after the execution of the test, and our end-to-end approach focuses on finding new, unique bugs.

Several projects have studied the usage of refactorings [6, 8, 28, 29, 45] and agreed that refactoring engines are underused. Most recently, Vakilian et al. [45], through a field study followed up by semi-structured interviews, investigated the reasons for low usage of refactorings and reported that the low usage is mostly due to unpredictability of the refactorings rather than their reliability. Our findings empirically confirm that the number of failures is not too high, but there is still a need for improvement and developing new infrastructure for building more reliable refactoring engines [15, 31, 34, 42, 45].

## 7 Conclusions

We presented a simple yet extremely effective approach to detect unique, real bugs in refactoring engines and to estimate their reliability. As opposed to previous techniques that generate input programs, our approach uses existing projects as inputs. As opposed to previous techniques that used existing projects as inputs for testing/evaluating refactorings, our approach identifies unique bugs using clustering, minimization, and finding duplicates. We applied our approach on testing Eclipse refactoring engines for Java and C, and we found and reported 77 new bugs for Java and 43 for C. We expect that bugs commonly found from real applications are more likely to be fixed than bugs discovered from artificially generated corner cases; in fact, the Eclipse developers already fixed 8 of the bugs we reported and confirmed 62 more as real bugs.

The main message of this paper is not that refactoring engines are buggy but that the proposed end-to-end approach works well to find these bugs. However, the approach also has challenges to be addressed in the future, e.g., automated minimization of programs and finding of duplicate bugs. While the paper focused on testing refactoring engines, we believe that the same approach can be used to test other aspects of IDEs that require programs/projects as test inputs, and to estimate their reliability on real projects.

## Acknowledgments

We would like to thank Anirudh Balagopal for helping with the bug reports; Zack Coker, Sarfraz Khurshid, Rupak Majumdar, Aleksandar Milicevic, Andy Podgurski, and Friedrich Steimann for discussions about this work; and Caius Brindescu, Mihai Codoban, Yu Lin, Stas Negara, Cosmin Radoi, and Mohsen Vakilian for providing comments on the text. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0746856, CNS-0958199, CCF-1213091, and CCF-1217271.

## References

1. N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful ... really? In *ICSM*, pages 337–345, 2008.
2. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.
3. CDT home page. `http://www.eclipse.org/cdt`.
4. M. Ó. Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam. Experimental assessment of software metrics using automated refactoring. In *ESEM*, pages 49–58, 2012.
5. Z. Coker and M. Hafiz. Program transformations to fix C integers. In *ICSE*, pages 792–801, 2013.
6. S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar. Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. In *ISESE*, pages 288–296, 2006.
7. B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *FSE*, pages 185–194, 2007.
8. D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107, 2006.
9. D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.
10. P. Ebraert and T. D'Hondt. Dynamic refactorings: Improving the program structure at run-time. In *RAM-SE*, pages 101–110, 2006.
11. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
12. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *ICSE*, pages 225–234, 2010.
13. GMP home page. `http://gmplib.org`.
14. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.
15. M. Hafiz and J. Overbey. OpenRefactory/C: An infrastructure for developing program transformations for C programs. In *SPLASH*, pages 27–28, 2012.
16. J. Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2005.
17. V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov. Reducing the costs of bounded-exhaustive testing. In *FASE*, pages 171–185, 2009.
18. JDT home page. `http://www.eclipse.org/jdt`.
19. JPF home page. `http://babelfish.arc.nasa.gov/trac/jpf`.
20. JUnit home page. `http://junit.org`.
21. Eclipse Juno home page. `http://www.eclipse.org/juno`.
22. libpng home page. `http://www.libpng.org/pub/png/libpng.html`.
23. Apache log4j home page. `http://logging.apache.org/log4j/2.x`.
24. Lucene home page. `http://lucene.apache.org`.
25. D. Marinov and S. Khurshid. TestEra: A novel framework for testing Java programs. In *ASE*, pages 22–31, 2001.
26. Commons Math home page. `http://commons.apache.org/math`.
27. G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *ICSE*, pages 142–151, 2006.
28. E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE*, pages 287–297, 2009.

29. S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *ECOOP*, 2013. (to appear).
30. OpenRefactory/C - A refactoring infrastructure for C. http://openrefactory.org.
31. J. L. Overbey. *A Toolkit For Constructing Refactoring Engines*. PhD thesis, University of Illinois at Urbana Champaign, 2011.
32. Refactoring actions home page. http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.user/concepts/concept-refactoring.htm.
33. J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.
34. M. Schäfer. *Specification, Implementation and Verification of Refactorings*. PhD thesis, Oxford University Computing Laboratory, 2010.
35. M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *OOPSLA*, pages 277–294, 2008.
36. M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Refactoring Java programs for flexible locking. In *ICSE*, pages 71–80, 2011.
37. M. Schäfer, A. Thies, F. Steimann, and F. Tip. A comprehensive approach to naming and accessibility in refactoring Java programs. *IEEE Trans. Soft. Eng.*, 38(6):1233–1257, 2012.
38. G. Soares. Making program refactoring safer. In *ICSE*, pages 521–522, 2010.
39. G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni. Analyzing refactorings on software repositories. In *SBES*, pages 164–173, 2011.
40. Y. Song, X. Wang, T. Xie, L. Zhang, and H. Mei. JDF: Detecting duplicate bug reports in Jazz. In *ICSE*, pages 315–316, 2010.
41. D. Spinellis. CScout: A refactoring browser for C. *Sci. of Comp. Prog.*, 75(4):216–231, 2010.
42. F. Steimann, C. Kollee, and J. Pilgrim. A refactoring constraint language and its application to Eiffel. In *ECOOP*, pages 255–280. 2011.
43. B. Sun, G. Shu, A. Podgurski, and S. Ray. CARIAL: Cost-aware software reliability improvement with active learning. In *ICST*, pages 360–369, 2012.
44. A. Thies and F. Steimann. Systematic testing of refactoring tools. In *AST (poster)*, 2010. http://www.fernuni-hagen.de/ps/prjs/rtt/rtt_poster.pdf.
45. M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, pages 233–243, 2012.
46. Visual Assist X home page. http://www.wholetomato.com.
47. X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470, 2008.
48. J. Wloka, R. Hirschfeld, and J. Hänsel. Tool-supported refactoring of aspect-oriented programs. In *AOSD*, pages 132–143, 2008.
49. XRefactory home page. http://www.xref.sk/xrefactory/main.html.
50. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Soft. Eng.*, 28(2):183–200, 2002.
51. zlib home page. http://www.zlib.net.