Automated Testing of Refactoring Engines

Brett Daniel Danny Dig Kely Garcia Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{bdaniel3, dig, kgarcia2, marinov}@cs.uiuc.edu

ABSTRACT

Refactorings are behavior-preserving program transformations that improve the design of a program. Refactoring engines are tools that automate the application of refactorings: first the user chooses a refactoring to apply, then the engine checks if the change is safe, and if so, transforms the program. Refactoring engines are a key component of modern IDEs, and programmers rely on them to perform refactorings. A fault in the refactoring engine can have severe consequences as it can erroneously change large bodies of source code and lead to strenuous debugging sessions.

We present a technique for automated testing of refactoring engines. Test inputs for refactoring engines are programs. The core of our technique is a framework for iterative generation of structurally complex test inputs. We instantiate the framework to generate abstract syntax trees that represent Java programs. We also create several kinds of oracles to automatically check that the refactoring engine transformed the generated program correctly. We have applied our technique to testing Eclipse and NetBeans, two popular open-source IDEs for Java, and we have exposed new faults in both: 9 faults in Eclipse and 10 faults in NetBeans.

1. INTRODUCTION

Refactoring [8] is a disciplined technique of applying behavior-preserving transformations to a program with the intent of improving its design. Examples of refactorings include renaming a program element to better convey its meaning, replacing field references with calls to accessor methods, splitting large classes, moving methods to different classes, or extracting duplicated code into a new method. Each refactoring has a name, a set of preconditions, and a set of specific transformations to perform [22]. *Refactoring engines* are tools that automate applications of refactoring. The programmer need only select which refactoring to apply, and the engine automatically checks the preconditions, which often requires nontrivial program analysis. If the preconditions are satisfied, the engine applies the transformations, possibly changing many locations in the program.

Refactoring is becoming increasingly popular as evidenced by the inclusion of refactoring engines in modern IDEs such as Eclipse [5] or NetBeans [21] for Java. Refactoring is also a key practice of agile software development methodologies, such as eXtreme Programming [2], whose success prompts even more developers to use refactoring engines on a regular basis. Indeed, the common wisdom views refactoring engines as one of the safest tools used to refactor a program [9].

It is important that refactoring engines be reliable—a fault in a refactoring engine can silently introduce faults in the refactored program and lead to nightmarish debugging sessions. If the refactored program does not compile, the refactoring can be easily undone. However, if the refactoring engine erroneously changes the semantics of the refactored program, the situation is similar to encountering a fault in a compiler: the unlucky programmers usually assume that their program is at fault and spend hours or days debugging before determining that the tool is at fault.

Since refactoring engines must be reliable and are very complex, refactoring engine developers have invested heavily in testing. For example, Eclipse version 3.2 has over 2,600 refactoring unit tests (publicly available from the Eclipse CVS repository). Conventionally, testing a refactoring engine involves creating input programs by hand along with their expected outputs, each of which is either a refactored program or an expected precondition failure. The testers then execute these tests automatically with a tool such as JUnit [10]. Writing such tests manually is tedious and results in incomplete test suites, potentially leaving many hidden faults in refactoring engines.

We present an approach that automates testing of refactoring engines. The core of our approach is *ASTGen*, a framework for automated generation of input programs. ASTGen allows developers to write *imperative generators* whose executions produce input programs. More precisely, ASTGen offers a library of generic, reusable, and composable generators that produce abstract syntax trees (ASTs). With ASTGen, a developer can generate more tests and focus more on the creative aspects of testing than when with manual generation. Instead of manually writing input programs, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. For example, to test the RenameField refactoring, the generated program should contain a class that declares a field. Our generators systematically produce a large number of programs that satisfy such constraints.

ASTGen follows the *bounded-exhaustive* approach [3, 13, 26] for exhaustively testing all inputs within the given bound. This approach covers all "corner cases" within the given bound. In contrast, manual testing requires knowledge of the corner cases and manually-written tests covering each. Bounded-exhaustive testing has never been used for test inputs as complex as Java programs, and the approach of *imperative generators* introduced in this paper differs from previous approaches using declarative generators. We discuss related work in Section 7.

An important problem in automated generation of test inputs is automated checking of outputs, also known as the *oracle problem.* Our approach uses a variety of oracles. The simplest oracles check that the refactoring engine does not crash (that is, does not throw an uncaught exception) and that the refactored program compiles. More advanced oracles take into account the semantics of refactoring and check specific properties such as invertibilty: renaming an entity from A to B and then back from B to A should produce the same starting input program. We also check structural properties: moving an entity should indeed create the entity in the new location. Finally, we use *differential testing* [20, 26] in which one implementation serves as the oracle for another implementation. Specifically, we run the same input programs on Eclipse and NetBeans and compare their refactored outputs or precondition violations. Section 5.1 presents our oracles in detail.

This paper makes the following contributions:

Framework for imperative generators: We present a novel framework for generation of structurally complex test inputs. Our framework uses imperative generators that specify *how* the inputs should be generated. Previous work [3, 12, 13, 17, 18, 26] has used declarative generators that describe *what* the inputs look like and thus require potentially expensive search to generate the actual inputs.

Instantiation for generating ASTs: We instantiate the general framework to generate abstract syntax trees (ASTs) representing Java programs. Our instantiation provides basic generators that echo the structure of simple ASTs as well as more complex generators that generate entire programs used as test inputs for refactoring engines.

Evaluation: We have used our framework to test several refactorings in Eclipse and NetBeans, two popular open-source IDEs for Java. We have implemented automatic execution of refactoring engines on the input programs that our generators produce. We have also implemented several oracles to verify that refactorings complete as expected. Our experiments have discovered 9 new faults in Eclipse and 10 new faults in NetBeans. We have reported these faults in the bug-tracking systems of both IDEs.

Our results, including the bugs reported, and our ASTGen code are available for public download from `http://mir.cs.uiuc.edu/astgen`.

2. TESTING ENCAPSULATE FIELD

We use the EncapsulateField refactoring as our running example throughout this paper. This refactoring replaces all references to a field with accesses through setter and getter methods. The EncapsulateField refactoring takes as input the name of the field to encapsulate and the names of the new getter and setter methods. More precisely, it:

```
// before refactoring
class A {
  public int f;
  void m(int i) {
    f = i * f;
  }
}
```

```
// after refactoring
class A {
  private int f;
  void m(int i) {
    setF(i * getF());
  }

  public void setF(int f) {
    this.f = f;
  }
  public int getF() {
    return this.f;
  }
}
```

Figure 1: Example EncapsulateField refactoring

- creates a public getter method that returns the field's value
- creates a public setter method that updates the field's value to a given parameter's value
- replaces all field reads with calls to the getter method
- replaces all field writes with calls to the setter method
- changes the field's access modifier to private.

The EncapsulateField refactoring checks several preconditions: that the code does not already contain accessor methods and that the accessors are applicable to the expressions in which the field appears, among other more complex checks.

Figure 1 shows a sample program before and after encapsulating the field `f` into the `getF` and `setF` methods.

Our framework allows the tester to write generators that can exhaustively generate programs containing field references. In particular, the tester can write a generator that produces many inputs, each of which is a class that contains a field and a method that references the field in all kinds of interesting expressions, as defined by the tester. In other words, the tester has intuition for which programs to generate, but it is quite tedious to manually write input programs that cover all kinds of field references. Section 4.2.1 describes how to write this generator, effectively codifying the intuition into automatic generation. This generator is not only useful for testing EncapsulateField but also reusable for testing other refactorings that operate on fields such as RenameField and PushDownField.

This generator produces, among others, the program in Figure 2 that reveals a fault in NetBeans. In this case, the parentheses around the field reference cause the refactoring engine to leave the field reference unencapsulated. Our Differential Oracle (Section 5.1) quickly catches this fault since NetBeans and Eclipse produce different refactored code. Note that the "refactored" code from NetBeans compiles, so a simple oracle would not catch the fault. This omission could cause problems if the developer wishes to add logic to the accessor methods. The unencapsulated field reference would not trigger this additional logic.

Another useful generator is one that generates two classes, say `A` and `B`, such that `A` and `B` exhibit all possible relationships involving class inheritance, class name reference, or

```
// before refactoring              // after refactoring
class A {                          class A {
  int f;                             private int f;
  void m(){                          void m(){
    (new A().f) = 0;                   (new A().f) = 0;
  }                                  }
}
                                     ...getF...
                                     ...setF...
                                   }
```

Figure 2: EncapsulateField bug in NetBeans

```
class A {
  class B extends A {
    void m(A a) {}
  }
}
```

Figure 3: Two classes in an inheritance, inner class, and method parameter type relationship

containment (i.e., inner or local class). This generates many pairs of classes, including the pair in Figure 3 that illustrates all three relationship types: inheritance, inner class, and name reference via method parameter. One can reuse this generator to test several types of refactorings that depend on class name or location, including RenameClass, MemberToTop, and PushDownField.

The true power of our framework appears when building more complex generators from simpler ones. The previous two generators can be composed into a third generator that generates even more expressive programs in which one class declares a field and the other references it in some way. This generator outputs the program in Figure 4 that reveals a bug in Eclipse. In this case, the refactoring engine mistakenly identifies the `super.f` expression as a field read. The DoesNotCompile oracle quickly determines this is a bug.

We will return to these three generators in Section 4. They are particularly important because in concert they found the majority of the faults we report in Section 6.

3. FRAMEWORK

Before we describe how we generate abstract syntax trees for testing refactoring engines, it is necessary to briefly describe why we chose our iterative approach to test data generation and explain some of the generation tools that we built.

3.1 Why Iterative & Imperative Generation?

ASTGen is an imperative, iterative, bounded-exhaustive test data generation toolkit. It is *imperative* in that it falls on the tester to define how input data is built; *iterative* in that it generates inputs lazily, one at a time; and *bounded-exhaustive* in that it will systematically explore the entire combinatorial space of a given set of generators.

This approach has the following benefits:

Easy to understand: Testers intuitively grasp the idea of looping over a set of generated inputs. It is a natural extension to the hand-written tests that testers are used to writing.

Easy to compose: Testers can combine generators to create complex data or to tailor data generation to a particular testing domain. Our framework is abstract and generic, allowing testers to combine generators in an arbitrary fashion.

Scales well with data size: Testers can build very large and complex data structures with a small number of generators.

Scales well with amount of data: Since data is generated lazily, there is no overhead related "pre-generation" or to storing a large number of inputs.

Catches corner cases: Bounded-exhaustive testing will cover all inputs within a given bound, including those that random testing [16] may miss or that testers are unaware of.

```
// before refactoring              // after refactoring
class A {                          class A {
  int f;                             private int f;
}                                    ...setF...
                                     ...getF...
class B extends A {                }
  void m() {
    super.f = 0;                   class B extends A {
  }                                  void m() {
}                                      getF() = 0;
                                     }
                                   }
```

Figure 4: EncapsulateField bug in Eclipse due to super access

Since we wrote ASTGen in Java [11], testers do not have to learn any new languages or syntax, and they have the full power of a general-purpose language at their disposal.

Intuitively, we can view the generators as synchronous electronic circuits that generate program elements instead of electronic signals. The same way that a circuit designer combines primitive circuits to create more complex ones, a tester combines more primitive generators to generate more complex data. This analogy goes much further. The basic building blocks of circuits are sequential elements that store the state and combinatorial elements that combine the signals. Generators also consist of sequential components that store the generation state and combinatorial components that build larger data structures from smaller data structures. Testers combine the combinatorial parts to pass information from one to the other and to compute the final value.

The separation of sequential and combinatorial parts has several desirable properties. First, this separation is a familiar concept from circuit design and is used almost exclusively since asynchronous circuits are harder to design and test. Second, it makes connecting generators simpler since it decouples generator iteration from data composition. This means that generators can change their state in any order, and only after all generators have computed their values does the parent generator combine the values together. Finally, this decoupling also makes developing components easier: it simplifies the class interface and allows one to test data composition separately from generator iteration.

Because a generator cannot access any data other than that which is already stored in its internal state, one can consider data composition in terms of higher-order functions. This means, for example, that the generator that builds the `super.f` expression in Figure 4 cannot be "passed" the name of the field it is referencing. Instead, this generator effectively produces a program with "holes", and other generators then supply "plugs" that fill these "holes", creating complete program elements. We will return to higher-order generation in Section 4.2.

3.2 Iterative Generation Tools

We define an iterative generator as one that outputs a new generated value every time `next` is called. The basic interface looks like the following:

```
interface IGenerator<T> implements Iterable<T> {
    boolean hasNext();
    boolean isReset();
    T next();
    T current();
    Iterator<T> iterator();
}
```

Since generators implement the `java.lang.Iterable` interface by declaring the `Iterator<T> iterator()` method, it is very easy to loop over all generated values [7].

```
IGenerator<T> valueGen = ...;
for (T value : valueGen) {
    doSomething(value);
}
```

We have built several supporting tools that implement and act upon the `IGenerator` interface. The simplest is called `Literal`, and it produces a single value.

```
Literal<String> stringLiteral = new Literal<String>("foo");
Literal<Integer> intLiteral = new Literal<Integer>(1);
```

The next generator is called `Chain`. It is a generator that when given values or other generators, produces all values in order.

```
Chain<String> chain1 = new Chain<String>("a", "b", "c");

Chain<String> chain2 = new Chain<String>();
chain2.add("d");
chain2.add(chain1);
chain2.add(new Literal<String>("e"));

for (String s : chain2) {
   System.out.print(s + " ");
}
// Outputs: d a b c e
```

These simple generators are useful, but their true power appears when they are linked together. An abstract generator called `CompositeGenerator` encapsulates this task. It represents a generator with any number of child generators. When iterated, it iterates its children.

```
abstract class CompositeGenerator<T>
      implements IGenerator<T> {

   abstract List<IGenerator> getChildren();
   abstract T generateCurrent();

   boolean hasNext() {
      return childrenCanIterate();
   }
   T next() {
      invokeNextOnChildren();
      return generateCurrent();
   }
   void reset() {
      resetChildren();
   }
   ...
}
```

Say we have a data structure representing a simple pair of values.

```
class Pair<L, R> {
   L left;
   R right;

   Pair(L left, R right) {...}
}
```

Using `CompositeGenerator`, we can build a simple `PairGenerator` that will generate the Cartesian product of two generators. This generator illustrates the separation between data composition, which occurs in in `generateCurrent()` and generator iteration which is inherited from `CompositeGenerator.next()`.

```
class PairGenerator<L, R>
      extends CompositeGenerator<Pair<L, R>> {

   IGenerator<L> leftGen;
   IGenerator<R> rightGen;

   ... constructors and accessors ...

   List<IGenerator> getChildren() {
      return Arrays.asList(
         leftGen,
         rightGen);
   }

   Pair<L, R> generateCurrent() {
      return new Pair<L, R>(
         leftGen.current(),
         rightGen.current());
   }
}
```

Now that we have defined `PairGenerator`, we can use it by passing the appropriate generators to its constructor or accessors.

```
Chain<String> chain1 = new Chain<String>("a", "b");
Chain<Integer> chain2 = new Chain<Integer>(1, 2, 3);
PairGenerator<String, Integer> pairGen =
   new PairGenerator<String, Integer>(chain1, chain2);

for (Pair<String, Integer> pair : pairGen) {
   System.out.print(pair);
}
// Outputs: [a,1] [a,2] [a,3] [b,1] [b,2] [b,3]
```

As we shall demonstrate, these simple generators are well-suited to creating abstract syntax trees as well as more complex structures.

4. INSTANTIATION FOR ASTS

We next illustrate how ASTGen can be used to test refactoring engines by generating abstract syntax trees of increasing complexity. First we show how a simple syntax element can be implemented as a tree of generators. Then, we will discuss the implementation of more complex abstract syntax tree generators such as those described in our running example from Section 2.

4.1 A Simple AST Generator

Figure 5 shows an AST node representing a much-simplified field declaration. It has three other AST nodes as children. It is common for simple generators to mirror the structure of their corresponding data structure. In this case, `FieldDeclarationGenerator`, shown in Figure 6 contains a child generator for each AST node in `FieldDeclaration`.

```
class FieldDeclaration {
   Modifier modifier;
   Type type;
   Identifier identifier;

   ... constructors and accessors ...
}
```

Figure 5: Field declaration data structure

```
class FieldDeclarationGenerator
      extends CompositeGenerator<FieldDeclaration> {
   IGenerator<Modifier> modifierGen;
   IGenerator<Type> typeGen;
   IGenerator<Identifier> identiferGen;

   ... constructors and accessors ...

   List<IGenerator> getChildren() {
      return Arrays.asList(
         modifierGen,
         typeGen,
         identifierGen);
   }

   FieldDeclaration generateCurrent() {
      FieldDeclaration generated = new FieldDeclaration();
      generated.setModifier(modifierGen.current());
      generated.setType(typeGen.current());
      generated.setIdentifier(identifierGen.current());
      return generated;
   }
}
```

Figure 6: Field declaration generator

To use the generator, one can initialize it by setting each child generator to a `Chain` of values as illustrated in Figure 7. For simplicity, we show the Java syntax elements (i.e. `public`, `int`) as if they were defined as variables, rather than showing the code used to build their Java AST nodes.

`FieldDeclarationGenerator` illustrates several concepts integral to our generation implementation. First, it shows how data composition is separate from generator iteration. The child generators are iterated by the `ChildIterator` returned from `getChildIterator` while the data composition takes place in `generateCurrent`.

Second, it illustrates the flexibility of generators. `FieldDeclarationGenerator`'s child generators are all `IGenerators`, allowing one to substitute any generator that generates the correct type. Unlike grammars which remain static once written, this genericity allows generators to be reused in many contexts. One need simply initialize the appropriate

```
IGenerator<Modifier> modifierGen =
   new Chain<Modifier>(public, private);
IGenerator<Type> typeGen =
   new Chain<Type>(int, boolean);
IGenerator<Identifier> identifierGen =
   new Chain<Identifier>(someField, anotherField);
FieldDeclarationGenerator fieldDeclGen =
   new FieldDeclarationGenerator(
      modifierGen,
      typeGen,
      identifierGen);
```

Figure 7: Pseudocode for initializing FieldDeclarationGenerator

children.

We have implemented basic generators for 29 common Java syntax elements. These generators encapsulate abstract syntax tree generation and thus increase reusability in that one does not need to define `Chains` of values everywhere. For example, in Figure 7, we could have used `ModifierGenerator` rather than a chain of values. Also, many of the AST generators perform much more complex generation such as that described in the following sections.

4.2 Complex AST Generators

We have shown that our iterative approach can easily generate simple syntax elements represented as AST nodes. Next we shall discuss the three complex generators from our running example.

4.2.1 *Field Reference Generator*

The `FieldReferenceGenerator` generates a class containing a field and a method that references the field in many possible ways. It produces several thousand programs, one of which is similar to Figure 2.

It is composed of the following five child generators:

- An `IGenerator<FieldDeclaration>` (such as a `FieldDeclarationGenerator` from Figure 6), provided by the caller. In the example, this generator generated the `int f;` declaration.
- A `FieldReferenceExpressionGenerator` that uses the field name from the field declaration to build an expression that references the field. This expression can be the name of the field, `this.f`, `A.this.f`, or, as in Figure 6, `new A().f`.
- A `ParenthesizingExpressionGenerator` returns an expression that either echoes the expression explicitly or parenthesizes it. In the example, the generator has parenthesized the referencing expression yielding `(new A().f)`.
- A `NestedExpressionGenerator` that nests the parenthesized expression in one of the many possible expressions applicable to the field's type. In the example, the generated expressions is the assignment expression: `(new A().f) = 0`. However, since the field has type `int`, other applicable expressions include the binary arithmetic operators, unary operators, and many others.
- A `ExpressionInStatementGenerator` that nests the full expression in one of many types of statements. In the example, the statement simply contains the expression itself, but it can also generate other control flow or looping statements.

It is interesting to note that the `NestedExpressionGenerator` can accept another `NestedExpressionGenerator`, allowing one to create expressions nested within each other to an arbitrary depth. Indeed, the initial design for the `FieldReferenceGenerator` included a `NestedExpressionGenerator` in place of the `ParenthesizingExpressionGenerator`, but we found that the resulting combinatorial explosion increased generation time substantially and did not yield any new bugs not already found by simply parenthesizing the expression. This sidebar is interesting because it illustrates how a tester

```
// Get child generators' values (order does not matter)
FieldDeclaration fieldDecl =
   fieldDeclGen.current();
FieldReferenceExpressionMethObj fieldRefExprMO =
   fieldRefExprGen.current();
ParenthesizingExpressionMethObj parenExprMO =
   parenExprGen.current();
NestedExpressionMethObj nestedExprMO =
   nestedExprGen.current();
ExpressionInStatementMethObj exprInStmtMO =
   exprInStmtGen.current();

// Fill "holes" in ASTs (order matters)
Expression fieldRefExpr = fieldRefExprMO.fill(fieldDecl);
Expression parenExpr = parenExprMO.fill(fieldRefExpr);
Expression nestedExpr = nestedExprMO.fill(parenExpr);
Statement exprInStmt = exprInStmtMO.fill(nestedExpr);

// Build AST to return
MethodDeclaration methodDecl = makeMethod("m");
methodDecl.addStatement(exprInStmt);

TypeDeclaration typeDecl = makeClass("A");
typeDecl.addField(fieldDecl);
typeDecl.addMethod(methodDecl);
```

Figure 8: FieldReferenceGenerator generation

can tailor generators such that they produce data applicable to a particular test target.

At first glance, it would appear that the five child generators must be iterated in order since "later" generators depend on the data generated by "earlier" ones. However this is not the case since iteration and generation are decoupled. How then, for example, does the `NestedExpressionGenerator` use the parenthesized expression generated by `ParenthesizingExpressionGenerator`? The answer is that all the generators except the initial `IGenerator<FieldDeclaration>` generate higher-order "Method Objects" that represent ASTs with "holes".

Thus, the parent `FieldReferenceGenerator` generates data by first retrieving the method objects from each of its child generators. Then, it fills the "holes" by passing the appropriate AST nodes to the method objects. Finally, it builds the AST that it will return using the ASTs built by the method objects. The pseudocode for this procedure is listed in Figure 8.

4.2.2 *Class Relationship Generator*

The `ClassRelationshipGenerator` generates combinations of inheritance, class name reference, or location-based (i.e. inner or local class) relationships between two generated classes. When provided literal class generators, this generator produces several hundred relationship pairs, one of which is similar to Figure 3.

`ClassRelationshipGenerator`'s child generators are the following:

- Two `IGenerator<TypeDeclaration>`s provided by the caller that generate the classes to be related.
- A `InheritanceTypeGenerator` determines whether one class inherits from the other.
- A `ClassNameReferenceGenerator` similar to the `FieldReferenceGenerator` from Section 4.2.1. It generates many expressions, statements, and declarations that can contain a reference to the name of a class. In Figure 3, this generator produced the method declaration with a parameter type `A`.
- A `LocationTypeGenerator` determines where one class is located in relation to the other. In Figure 3, this generator specified that `B` is an inner class of `A`. Other possible locations are local in which a class is declared inside a method and separate in which both class are top-level elements.
- Three `DirectionGenerators` determine the direction in which the three relationship types "point". In Figure 3, all relationships are in the B-to-A direction: B inherits from A, B is an inner class of A, and B references A through a method parameter.

The `InheritanceTypeGenerator`, `ClassNameReferenceGenerator`, and `LocationTypeGenerator` all generate higher-order method objects. These method objects consume the two generated `TypeDeclarations` and a generated `Direction`, and return the same two `TypeDeclarations`, related in a particular way. After applying all three generated method objects, the pair of classes is returned to the caller.

Due to their exhaustive nature, generators often produce programs that do not compile. For example, depending on the combination of direction and relationship type, certain generated class relationships may be invalid. The following code illustrates such a relationship in which `B` is related to `A` by location, but `A` is related to `B` by inheritance.

```
class A extends B {
   class B {}
}
```

We can overcome this problem in three ways. First, we can filter invalid data by testing the current values of all child generators. In this example, we can "skip" the generated value if the `LocationGenerator`'s current value is "inner" in the B-to-A direction and the `InheritanceTypeGenerator`'s current value is "extends" in the A-to-B direction. Second, the caller can limit the generation to only those programs applicable to a particular task. We shall see in the next section that the `DualClassFieldReferenceGenerator` generates classes in all location and inheritance relationships in all directions, but omits the class name reference relationship because it is irrelevant to testing field references. Finally, we can delegate to the compiler and skip any generated programs that do not compile.

4.2.3 *Dual-Class Field Reference Generator*

The `DualClassFieldReferenceGenerator` generates all possible class relationships in which one class declares a field and the other references it. When supplied the simplest possible class and field generators, this generator generates over 14,000 programs, one of which is similar to Figure 4.

This generator combines aspects of the other two complex generators that we have discussed. First, it uses the `ClassRelationshipGenerator` to generate the inheritance and location relationships between the supplied classes. Then, it uses the `SingleClassFieldReference` to generate references to the field. Construction of the AST proceeds analogously to the previous two cases.

5. TESTING REFACTORINGS USING AST GENERATORS

We next show how we use our AST generators to test refactoring engines. First, we describe the oracles that we use to verify that a refactoring completes correctly on our generated data. Then, we show how we run generated programs through the refactoring engines.

5.1 Oracles

An important problem in automated generation of test inputs is automated checking of outputs, also known as the *oracle problem*. An ideal oracle for a refactoring engine would tell whether an automatically generated input program and its refactored version have the same semantics. Proving that two programs have the same semantics is generally undecidable [24]. However, given the fact that refactorings are program transformations with well defined structural properties, we can still check several useful properties of a refactored program. The following lists the oracles that we have implemented.

DoesNotCrash Oracle: Our simplest oracle checks that the refactoring engine does not throw an uncaught exception. Such an oracle is often used in "smoke testing" as a sanity check.

DoesNotCompile Oracle: Our next oracle checks if the refactored program compiles. All input programs are previously filtered and only the programs that compile are passed to the refactoring engine. This oracle compiles the refactored output programs and alerts the tester if there are compilation errors.

WarningStatus Oracle: Refactoring engines warn the user when a refactoring might change the semantics of the program. This oracle checks that the refactoring engine produces a warning status after the precondition checks. This oracle is used in conjunction with generators that create programs which intentionally do not meet the preconditions of a specific refactoring. For example, we wrote a generator called MethodWithParameterReference that creates many methods containing a reference to a parameter. One would expect that such programs would always fail the RemoveParameter refactoring's precondition checks.

Inverse Oracle: Refactorings are invertible program transformations. Given any refactoring τ and all programs P that meet the preconditions of τ, there exists another refactoring τ^{-1}, the inverse refactoring, such that:
$\tau^{-1}(\tau(P)) = P$.

Since refactorings make well-defined structural changes, we can compare the original and doubly-refactored programs by determining whether their ASTs are isomorphic. We implemented an AST comparator that normalizes the trees by sorting the methods and fields by name. With sorted methods and fields, the problem becomes one of comparing method bodies and field values. By ignoring whitespace, we are able to compare ASTs well enough to serve as an oracle.

Custom-Structural Oracle: We have implemented several refactoring-specific oracles. Since these oracles are aware of the structural changes made by their corresponding refactorings, they can check that the refactored program exhibits the expected changes. For example, we can verify that RenameField leaves no occurrences of the old name anywhere in the AST.

Differential Oracle: The last oracle we implemented is used in differential testing [20, 26]. This oracle takes an input program and a refactoring, and it feeds this pair to two refactoring engines, Eclipse and NetBeans. It then takes the output programs returned by the two engines and checks whether their ASTs are isomorphic using the AST comparator described above. If the two ASTs differ, a human inspects the two output programs to check whether the difference is caused by a bug in one of the refactoring engines. This oracle finds additional bugs that are not found by the previous oracles. Among others, it found the bug shown in figure 2.

```
String fieldName = "f";
FieldDeclarationGenerator fieldDeclGen =
   new FieldDeclarationGenerator(fieldName);
IGenerator<Program> testGen = new ...(fieldDeclGen);
for (Program p : testGen) {
   if (!p.compiles) {continue;}

   Refactoring r = new EncapsulateFieldRefactoring();
   r.setTargetField(fieldName);
   Program pPrime = r.performRefactoring(p);

   checkOracles(pPrime);
}
```

Figure 9: Pseudocode for testing EncapsulateField

Behavioral Oracles: In the future, we plan to implement a new oracle that will automatically generate a test suite for a given input program by using automated regression testing techniques [6, 27]. It will then compare whether the behavior of the input program (exercised under the test suite) is the same as the behavior of the refactored program.

5.2 Running Refactorings

We have described how to compose generators that produce interesting test data for testing refactorings. Here we return to our EncapsulateField example to explain how we test refactorings.

First, we create a FieldDeclarationGenerator initialized with the name of the field we expect to encapsulate. Then, we pass this generator to one of the three generators described in section 4.2. This generator produces the programs to refactor.

For each generated program, we first test if it compiles. If so, then we instantiate a refactoring provided by the IDE, initialize it with the field name, and invoke the refactoring engine. This yields a refactored program that we pass to each of the oracles. Figure 9 lists the pseudocode for this process.

We implemented a generalization of this process that we use to test several refactorings described in the next section. In Eclipse it takes the form of a custom plug-in that uses the platform's built-in test harness. In Netbeans, we extend the existing unit test suite.

6. EVALUATION

Here we evaluate ASTGen's usefulness as a reusable and composable test data generation framework. Our goal is to find and report bugs in refactoring engines using ASTGen. We tested several refactorings and found 9 bugs in Eclipse and 10 in Netbeans.

All results and code can be downloaded from [1].

	Generation				Oracles							
Refactoring	Primary Generator	Total Generated	Time [min:sec]	Compilable Inputs	WarningStatus		DoesNotCompile		Custom/Inverse	Differential	**Bugs Reported**	
					Ecl	NB	Ecl	NB			Ecl	NB
Rename(Class)	ClassRelationships	108	1:02	88	0	0	0	0	0	0	0	0
Rename(Method)	MethodReference	9540	89:12	9540	3816	0	0	0	0	5724	0	0
Rename(Field)	FieldReference	3960	28:20	1512	0	0	0	304	0	40	0	1
Rename(Field)	DualClassFieldRef.	14850	76:55	3969	0	0	0	0	0	0	0	0
EncapsulateField	ClassArrayField	72	0:45	72	0	0	48	0	0	48	1	0
	FieldReference	3960	15:19	1512	0	0	320	432	14	121	4	3
	DualClassFieldRef.	14850	41:45	3969	0	0	187	256	100	511	1	2
PushDownField	DualClassFieldRef.	4635	10:56	1064	760	380	152	228	0	380	2	2
CS(ChangeReturnType)	MethodReference	3816	37:36	3816	1992	-	0	-	0	-	0	-
CS(RemoveParameter)	MethodReference	5724	54:29	5724	1908	0	0	0	0	0	0	0
CS(RemoveParameter)	MethodParamRef.	1680	7:11	772	772	772	0	0	0	0	0	0
MoveToTop	ClassRelationships	70	0:36	51	0	0	0	2	0	2	0	1
	DualClassFieldRef.	6600	29:04	2824	0	0	353	507	0	2824	1	1
										Total Bugs:	**9**	**10**

Figure 10: Refactorings tested, generation results, oracles, and bugs reported. CS = ChangeSignature, Ecl = Eclipse, NB = NetBeans.

6.1 Refactorings Tested

We tested the following refactorings:

Rename: Rename a class, method, or field and change all of its occurrences to reflect the new name.

EncapsulateField: Replace every occurrence of a field with an accessor method. Its custom-structural oracle checks that the unencapsulated field appears a maximum of five times in the refactored program.

PushDownField: Move a field from a superclass to all subclasses. Its custom-structural oracle checks that the field no longer exists in the superclass.

ChangeSignature: Change a method signature by changing its return type or parameters. Its custom-structural oracle checks that the method was changed appropriately.

MemberToTop: Move an inner class out of its containing class. Its custom-structural oracle checks that the container class no longer contains an inner class.

We chose these refactorings because they are seemingly simple and demonstrate a variety of refactoring targets. EncapsulateField and PushDownField each target field declarations; ChangeSignature targets method declarations; MemberToTop targets inner classes; and Rename targets almost everything.

The first column of Figure 10 lists the specific refactorings performed. The last two columns lists the number of bugs reported for each. The table shows that even "trivial" refactorings such as Rename are susceptible to bugs. NetBeans does not support the ChangeReturnType refactoring, so we have inserted dashes in the appropriate columns.

6.2 Generation Evaluation

Section 4.1 describes how one can build a simple generator that produces AST nodes. This process is very straightforward. We asked two colleagues who had no experience with ASTGen to write a simple AST generator similar to FieldDeclarationGenerator. It took them each only about an hour, including the time needed for us to briefly describe the important classes in the framework.

In addition to a large library of simple AST generators, we built the following complex generators for testing refactorings. To conserve space, we have omitted the "Generator" suffix used in previous sections.

ClassRelationship: Generates two classes related in many ways. See Section 4.2.2.

FieldReference: Generates a class containing a field and a method that references the field in many ways. See Section 4.2.1.

DualClassFieldReference: Generates two classes related in many ways. One class declares a field and the other references it. See Section 4.2.3.

MethodReference: Generates a class with two methods. One method calls the other in one of many ways and may overload the called method by adding or removing parameters.

MethodParamReference: Generates a method declaration with a parameter referenced in many ways.

ClassArrayField: Generates a class that declares a field with many different array types. While simpler than the other generators listed above, it still revealed a bug in Eclipse.

The second column of Figure 10 lists the generators used to test each refactoring. By tailoring inputs, we were able to reuse generators for several refactorings. For example, DualClassFieldReference was useful for both field- and class-targeted refactorings and found bugs in both.

The third column lists the total number of programs generated. The number is very sensitive to the way in which the tester initializes the generator. For example, a fully-exhaustive DualClassFieldReferenceGenerator used for EncapsulateField produces 14,850 programs. In contrast, a version limited to producing inheritance relationships for PushDownField yields just 4,635.

Execution time, shown in the fourth column, is calculated as the time needed to generate and run all tests for a particular refactoring. Performing the refactoring takes up the vast majority of the execution time. It takes DualClassFieldReference just 13 seconds to generate 14,850 programs when refactoring execution is omitted. This fact is further corroborated by the close correspondence between compilable inputs—column 5—and execution time. Our suite tests compilable inputs at a consistent rate of about 100 per minute.

We ran our tests on a dual-processor 1.8 Ghz Dell D820 laptop with 1 gigabyte of RAM. It took just under seven hours to run our entire suite in Eclipse. Netbeans took about four times longer. While this time is not conducive to on-the-fly development testing, but it is certainly feasible for an overnight or weekend build process.

MethodReference and MethodParamReference are particularly interesting for several reasons. First, they create their method and parameter references by reusing the `ParenthesizingExpression`, `NestedExpression`, and `ExpressionInStatement` generators described in Section 4.2.1. This demonstrates how one can adapt existing generators for different tests. Second, it took one of us about two workdays to write both generators as well as infrastructure needed to run the four refactorings that they test. Together, they generate many more tests (20,760 of which 19,852 compile) than even the most talented tester could produce by hand in the same amount of time. Finally, by filtering in the manner described in Section 4.2.2, these generators produced only compilable inputs for three refactorings. In the other cases we relied on the compiler so that we did not accidentally filter valid tests.

6.3 Oracle Evaluation

Columns six through 11 of Figure 10 show the oracles exercized by each refactoring. The DoesNotCompile oracle yielded the most bugs in Eclipse while the Differential oracle found the most in NetBeans. The Custom-Structural oracle found one bug for EncapsulateField. We have omitted a column for the DoesNotCrash oracle since neither refactoring engine crashed.

In all cases, the WarningStatus oracle found only expected precondition failures, not bugs. Note, for example, that every compilable input generated by MethodParamReference for the RemoveParameter refactoring failed the WarningStatus oracle, just as one would expect. Had any passed, it would have indicated a bug since one cannot remove a parameter if it is referenced in the method body. This demonstrates the usefulness of generating inputs that are known to fail a particular oracle.

6.4 Bugs Found

The final two columns of Figure 10 show the previously unreported, unique bugs that ASTGen found: 9 in Eclipse and 10 in Netbeans. Since our approach is exhaustive, the oracles report many failures for each unique bug. This means, for example, that we only reported one unique bug for the 40 variations caught by the Differential oracle for RenameField.

A summary of each bug and links to the Eclipse and Netbeans bug reports can be found at [1].

7. RELATED WORK

There is a large body of work in the area of test-input generation. The most related to ours are grammar-based and bounded-exhaustive testing approaches.

Grammar-based testing [14,15,19,23,25] requires the user to describe test inputs with a grammar, and the tools then generate a set of strings that belong to the grammar. In 1972, Purdom [23] pioneered the algortihms for selecting a minimal set strings that achieve certain coverage criteria for grammar, e.g., strings that cover all terminals, all non-terminals, or all productions. More recently, Maurer [19], Sirer and Bershad [25], and Malloy and Power [15] developed tools for grammar-based generation that were used to find faults in several applications. We can view grammar-based approaches effectively as using first-order functional programs to specify the generation. The tools interpret these programs to generate *random* strings that belong to the input grammar. Other work by Claessen and Hughes [4] describes a random, but not grammar-based testing framework for Haskell Programs.

In constrast, the approach of Lammel and Schulte [14] and our approach *systematically* generate input data. Even more importantly, our approach allows the developers to use the full expressive power of familiar programming language such as Java to write imperative generators that produce test inputs. With our approach, the programmers can freely compose more advanced generators by reusing more basic generators. Achieving such reusability with grammars is fairly hard; for example, it is unclear how one could combine in a grammar-based approach the first two generators from Section 2 to obtain the third.

Bounded-exhaustive testing [3,12,13,17,18,26] is an approach for testing the programs exhaustively on all inputs within the given bound. We have previously developed two approaches, TestEra [18] and Korat [3], that can be in principle used for bounded-exhaustive generation of complex test inputs such as Java programs. These two approaches are *declarative*: they require the user to specify the constraints that describe *what* the test inputs look like (as well as the bound on the size of test inputs), and the tools then automatically search the (bounded) input space to generate all inputs that satisfy the constraints. TestEra requires the user to specify the constraints in a declarative language, while Korat requires the users to specify the constraints in an imperative language, but in both previous approaches, the user just specifies the constraints. In constrast, the approach presented in this paper is *imperative*: the programmer specifies *how* the test generation should proceed. The imperative approach makes the generation faster since no search is necessary. Also, the imperative approach gives the programmer more control over the generation, for example over the order of generation. Finally, the two previous declarative approaches have not been applied to generate inputs as complex as Java programs, whereas we have applied our new imperative approach to generate Java programs to test refactoring engines in Eclipse and NetBeans.

8. CONCLUSIONS

Refactoring engines have become popular because they allow programmers to quickly and (for the most part) safely change large programs. These tools also influence the culture of software development; programmers who use refactoring engines are more inclined to change large programs. Despite the high quality and widespread use of existing refactoring engines, they still contain bugs. Our goal is to help the developers of refactoring engines to reduce the number of the bugs.

We have presented a practical approach that automates testing of refactoring engines. The heart of our approach is ASTGen, a framework for generating structurally complex input programs. This framework allows testers to quickly sketch generators that mirror the ASTs of Java programs. The generators are easily reusable and composable to generate programs with arbitrary complexity. Our approach found 19 previously unreported bugs in Eclipse and NetBeans, two of the most popular refactoring engines for Java.

Based on these promising results, we believe that ASTGen can help in testing a variety of applications that operate on abstract syntax trees, e.g., compilers, code editors, and program optimizers. Also, the ideas behind ASTGen directly translate to languages other than Java; although we have presented generation of Java programs, ASTGen could just as easily generate ASTs trees for other languages.

The reported bugs and the code of ASTGen are available from its homepage at `http://mir.cs.uiuc.edu/astgen`.

9. REFERENCES

[1] ASTGen home page. http://mir.cs.uiuc.edu/astgen/.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[4] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

[5] The Eclipse Foundation. http://www.eclipse.org.

[6] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *FSE'06: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–264. ACM, November 2006.

[7] The for-each loop. http://java.sun.com/j2se/1.5.0/docs/guide/language/foreach.htm.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[9] D. Gallardo. Refactoring for everyone, Sept. 2003.

[10] E. Gamma and K. Beck. JUnit, 1997. `http://www.junit.org`.

[11] J. Gosling. *The Java Language Specification*. Sun Microsystems, 2005.

[12] S. Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Dec. 2003.

[13] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 2004.

[14] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, pages 19–38, 2006.

[15] B. A. Malloy and J. F. Power. An interpretation of purdom's algorithm for automatic generation of test cases. *First Annual International Conference on Computer and Information Science*, 2001.

[16] J. J. Marciniak. *Encyclopedia of Software Engineering*, chapter Random Testing, pages 1095–1104. Wiley-Interscience, 2001.

[17] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[18] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.

[19] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4), July 1990.

[20] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[21] The Netbeans IDE. http://www.netbeans.org.

[22] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[23] P. Purdom. A sentence generator for testing parsers. *Behavior and Information Technology*, 12(3):366–375, 1972.

[24] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[25] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proc. 2nd conference on Domain-specific languages*, 1999.

[26] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[27] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, pages 380–403, July 2006.