Automated Testing of Refactoring Engines

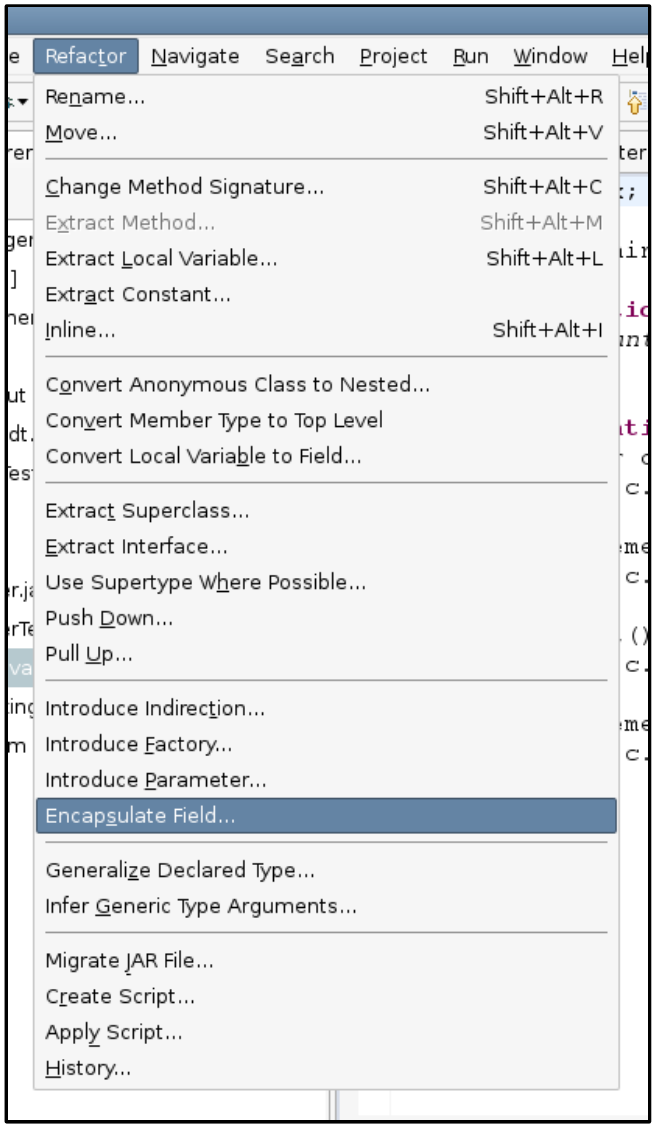
Brett Daniel
Danny Dig
Kely Garcia
Darko Marinov

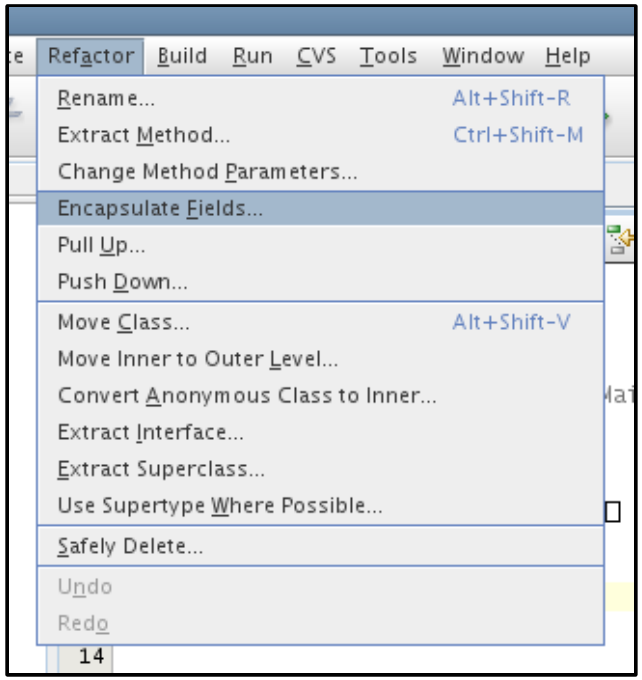
ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

ESEC/FSE 2007

Refactoring engines are tools that automate the application of refactorings

Eclipse and NetBeans





Why Test Refactoring Engines?

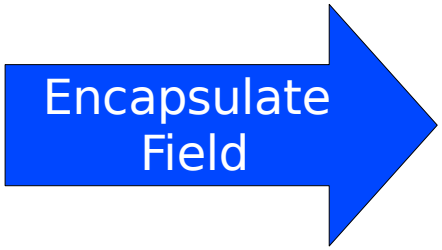
- Widely used
- Complex
  - Complex inputs: programs
  - Require nontrivial program analyses and transformation
- Can silently corrupt large bodies of code

Refactoring engines contain bugs

Example: Encapsulate Field

Replaces all field reads and writes with accesses through getter and setter methods

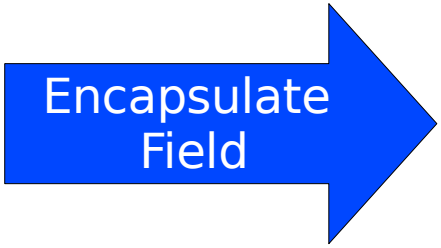
```
class A {
    int f;

    void m(int i) {
        f = i * f;
    }
}
```




```
class A {
  5 private int f;

    void m(int i) {
        setF(i * getF());
    }   4        3

  2 void setF(int f) {
        this.f = f;
    }

  1 int getF() {
        return f;
    }
}
```

Eclipse Bug

```
class A {
  int f;
}



class B extends A {
  void m() {
    super.f = 0;
  }
}
```

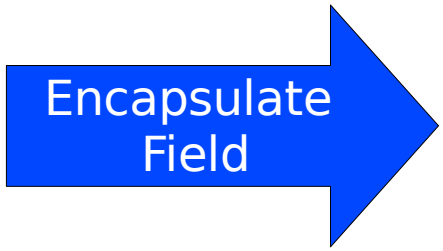



```
class A {
  private int f;

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

class B extends A {
  void m() {
    super.getF() = 0;
  }
}
```

```
super.setF(0);
```

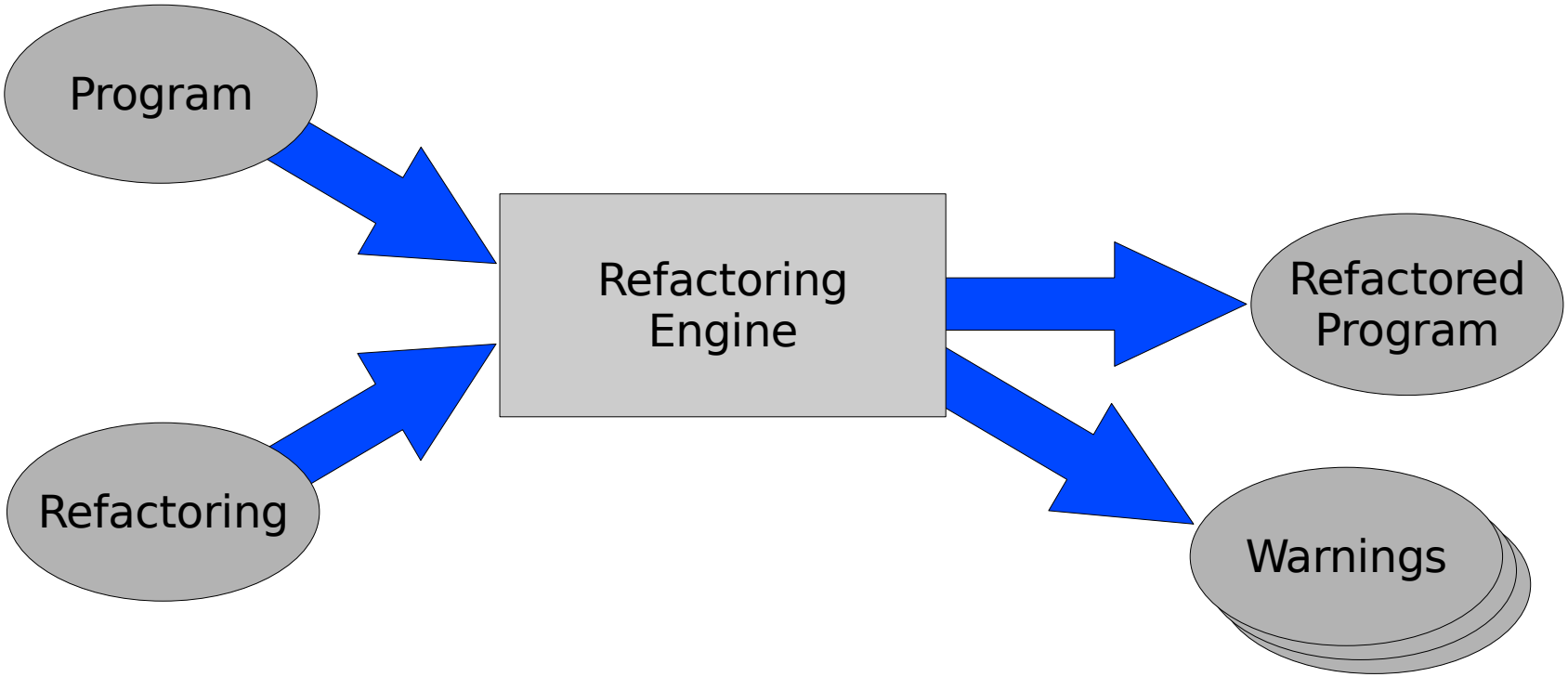
NetBeans Bug

```
class A {
  int f;




  void m() {
    (new A().f) = 0;
  }
}
```




```
class A {
  private int f;

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

  void m() {
    (new A().f) = 0;
  }
}

new A().setF(0);
```

Testing a Refactoring Engine



State of the Practice

- Manually written tests
  - Input: Program files and code to invoke refactoring
  - Output: Hand-refactored program file or warnings
- Automatically executed tests
  - Eclipse 3.2 has over 2,600 manually-written JUnit tests
  - NetBeans 6.0M3 has 252 XTest tests

Automated Testing

- Goal: Automate input generation and output checking
- Assumptions
  - Tester has intuition for input programs that might expose bugs
    - e.g. Encapsulating inherited fields
  - It is labor-intensive to manually write many input programs
    - e.g. Thousands of ways to reference an inherited field

Challenges

- How to “codify” the tester's intuition to create many interesting programs
- How to automatically check that refactoring completes correctly

Solution

- Developed ASTGen
  - Framework for generating abstract syntax trees
  - Provides library of generators that produce simple AST fragments
  - Tester writes complex generators composed of smaller generators
- Developed variety of oracles

ASTGen Design Goals

- Imperative
  - Tester can control how to build complex data
- Iterative
  - Generates inputs lazily, saving memory
- Bounded-Exhaustive
  - Catches “corner cases”
- Composable
  - Tester can create complex generators by reusing simpler parts

Testing Process

- Tester builds a generator using ASTGen
- Tester instantiates the generator for the test at hand.
- Tester runs generator in a loop. For each generated value:
  - Run refactoring
  - Check oracles

Testing Encapsulate Field

```
String fieldName = "f";
IGenerator<Program> testGen = new ...(fieldName);

for (Program in : testGen) {
  Refactoring r = new EncapsulateFieldRefactoring();
  r.setTargetField(fieldName);

  Program out = r.performRefactoring(in);
  checkOracles(out);
}
```

Example Generator

- Double-class field reference generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

...

Example Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

...

Example Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

...

Example Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

...

Example Generator

"Produces pairs of classes related by containment and inheritance. One class declares a field, and the other references the field in some way."

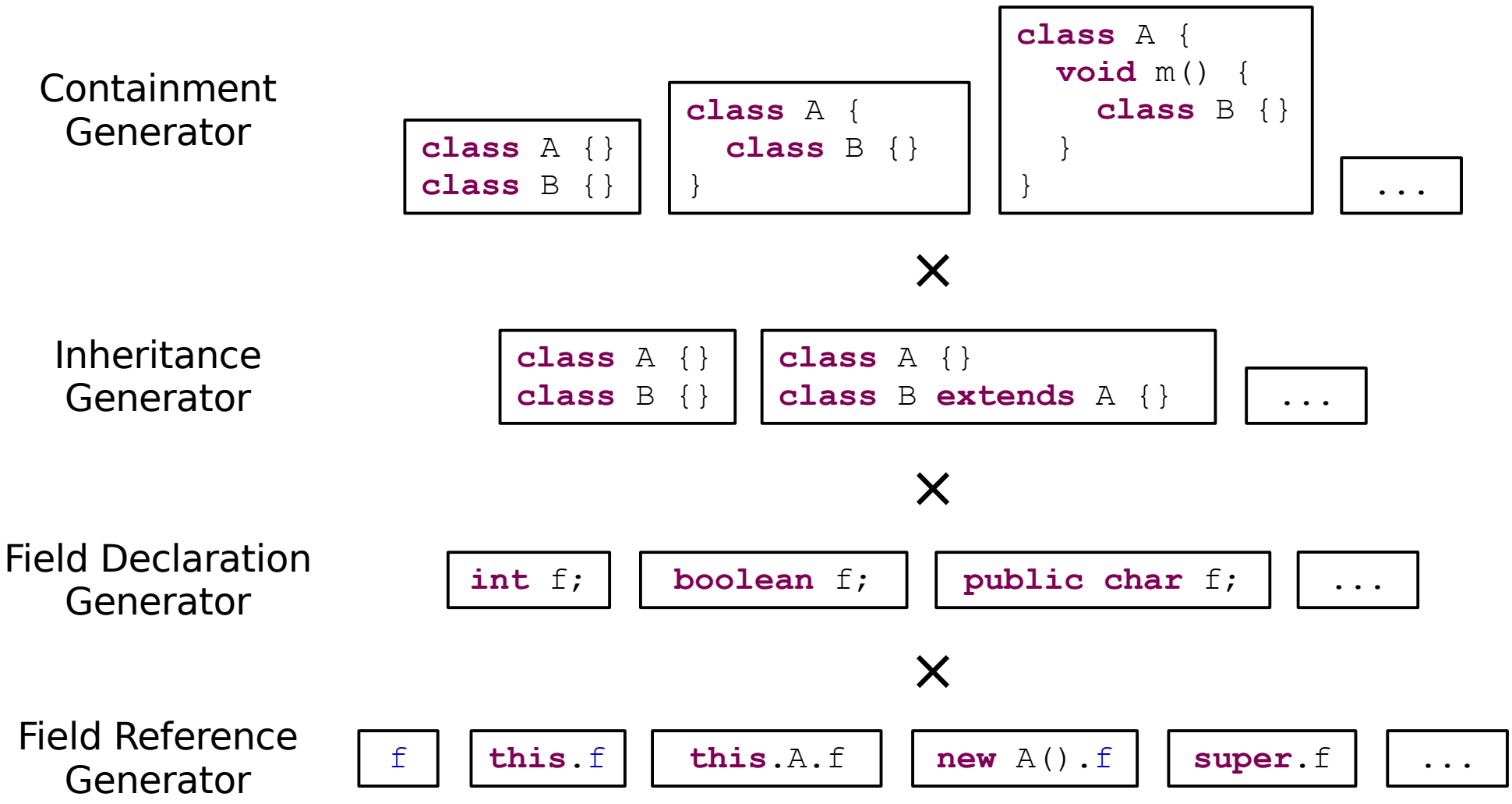
```
class A {
  int f;
}

class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
class A {
  boolean f;
}

class B {
  void m() {
    new A().f = true;
  }
}
```

```
class A {
  int f;

  class B {
    void m() {
      int i = f;
    }
  }
}
```

...

Composing Generators



Invalid Combinations

- Composition may be invalid

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

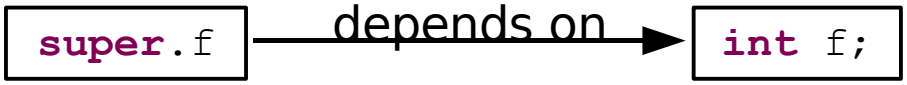
+

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

=

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

- Three solutions
  - Tester writes filter that verifies values
  - Dependent generators



  - Delegate to compiler

Oracles

- Check that the program was refactored correctly
- Challenges
  - Don't know expected output
  - Semantic equivalence is undecidable
  - Need to verify that correct structural changes were made

Oracles

- DoesCrash
  - Engine throws exception
- DoesNotCompile (DNC)
  - Refactored program does not compile
- WarningStatus (WS)
  - Engine cannot perform refactoring
  - Presence or lack of WarningStatus may indicate bug

Oracles

- Inverse (I)
  - Refactorings are invertible
  - Check that a refactoring is undone by its inverse
  - ASTComparator: Compares normalized ASTs
- Custom (C)
  - Check for desired structural changes
- Differential (Diff)
  - Perform refactoring in both Eclipse and NetBeans

Case Study

- Tested Eclipse and NetBeans
- Eight refactorings
  - Target field, method, or class
- Wrote about 50 generators
- Reported 47 new bugs
- Compared effectiveness of oracles

Generator Evaluation

Refactoring	Generator	Inputs	Time (m:ss)	Bugs	
				Ecl	NB
EncapsulateField	ClassArrayField	72	0:45	1	0
	FieldReference	1512	15:19	4	3
	DoubleClassFieldRef.	3969	41:45	1	2
	SingleClassTwoFields	48	1:16	1	0
	DoubleClassGetterSetter	417	8:45	3	3
Rename(Class)	ClassRelationships	88	1:02	0	0
Rename(Method)	MethodReference	9540	89:12	0	0
Rename(Field)	FieldReference	1512	28:20	0	1
Rename(Field)	DoubleClassFIeldRef.	3969	76:55	0	0
...	...	...	...	...	...
			Total	21	26

- Generation and compilation time less than refactoring time and oracles
- Human time: Took about two workdays to produce MethodReference
  - Reused many generators

Oracle Evaluation

Refactoring	Generator	WS		DNC		C/I	Diff	Bugs	
		Ecl	NB	Ecl	NB			Ecl	NB
EncapsulateField	ClassArrayField	0	0	48	0	0	48	1	0
	FieldReference	0	0	320	432	14	121	4	3
	DoubleClassFieldRef	0	0	187	256	100	511	1	2
	SingleClassTwoFields	0	0	0	0	48	15	1	0
	DoubleClassGetterSetter	216	0	162	162	18	216	3	3
Rename(Class)	ClassRelationships	0	0	0	0	0	0	0	0
Rename(Method)	MethodReference	0	0	0	0	0	0	0	0
Rename(Field)	FieldReference	0	0	0	304	0	40	0	1
Rename(Field)	DoubleClassFieldRef.	0	0	0	0	0	0	0	0
...	...	...		...		...	...	...	...
						Total		21	26

- DoesNotCompile found the most bugs
- WarningStatus, Inverse, and Differential can give false positives
- Many input programs exhibit same bug

Results

- 47 new bugs reported
  - 21 in Eclipse: 20 confirmed by developers
  - 26 in NetBeans: 17 confirmed, 3 fixed, 5 duplicates, 1 won't fix
  - Found others, but did not report duplicate or fixed
- Currently working with NetBeans developers to include ASTGen in testing process

Related Work

- Grammar-based test data generation
  - P. Purdom 1972
  - P. M. Maurer 1990
  - E. G. Sirer and B. N. Bershad 1999
  - B. A. Malloy and J. F. Power 2001
- Declarative, bounded-exhaustive generation
  - C. Boyapati, S. Khurshid, and D. Marinov 2002
  - S. Khurshid and D. Marinov 2004
  - R. Lämmel and W. Schulte 2006
- QuickCheck
  - K. Claessen and J. Hughes 2000

Future Work

- More refactorings
- Apply ASTGen to other program analyzers
- Removal of redundant tests, bug targeting
- Reduce or eliminate false alarms
  - Improved AST Comparator

Conclusions

- Despite their popularity, refactoring engines contain bugs
- ASTGen allows one to create many interesting ASTs
- We reported 47 new bugs

http://mir.cs.uiuc.edu/astgen/

Imperative vs. Declarative

- How to produce data
  vs.
  What data should look like
- TODO

Refactorings are behavior-preserving program transformations that improve the design of a program