PADERBORN UNIVERSITY

PADERBORN UNIVERSITY

JCRASHER: AN AUTOMATIC ROBUSTNESS TESTER FOR JAVA

SEMINAR: SOFTWARE TESTING

Theo Harkenbusch

Paderborn, 27.11.19

Software Robustness

Definition (Software Robustness)

“The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions”

– IEEE Standard Glossary of Software Engineering Terminology [1]

Software Robustness

Definition (Software Robustness)

“The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions”

– IEEE Standard Glossary of Software Engineering Terminology [1]

Why We Need Software Robustness

```java
public void setAge(int age) {
   this.age = age;
}
```

Why We Need Software Robustness

```
public void setAge(int age) {
   this.age = age;
}

// ...

public void bornInYear() {
   return currentYear - age;
}
```

Why We Need Software Robustness

```
// e.g. age = -5;
public void setAge(int age) {
   this.age = age;
}

// ...

public void bornInYear() {
   return currentYear - age;
}
```

Why We Need Software Robustness

```java
// e.g. age = -5;
public void setAge(int age) {
   this.age = age;
}

// ...

public void bornInYear() {
   return currentYear - age; // 2019 - (-5) = 2024
}
```

Why We Need Software Robustness

```
// e.g. age = -5;
public void setAge(int age) {
   if (age <= 0) {
      throw new IllegalArgumentException("argument
          ’age’ must be positive");
   }
   this.age = age;
}
```

JCrasher

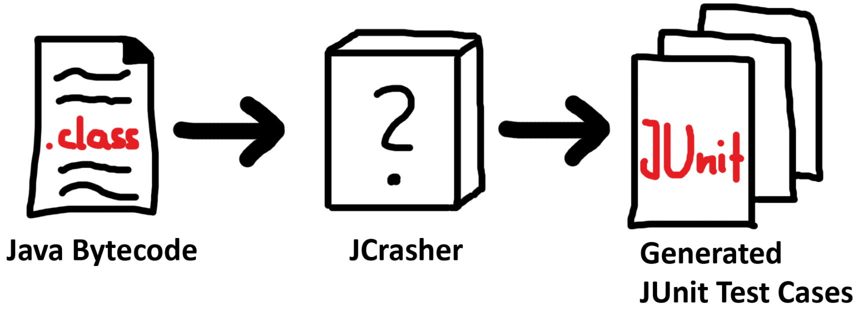


Figure: Input and Outputs of JCrasher

Test Case Generation

- Testing public methods
- Generate different parameter combinations
  - One combination per test case
- Randomly select inputs
- Inputs can be Java objects

Parameter-Graph

- In-memory data-structure
- Edges denote ways to construct values of given type
- Type $\mapsto$ pre-set value
- Type $\mapsto$ methods returning this type
- Create different parameter combinations by traversing the graph

Parameter-Graph Example

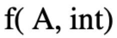


Figure: Method under test: `f(A, int)`. Taken from [2, p. 1030].

Parameter-Graph Example

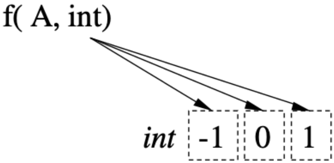


Figure: JCrasher using predefinied values -1, 0, 1 for `int`. Taken from [2, p. 1030].

Parameter-Graph Example

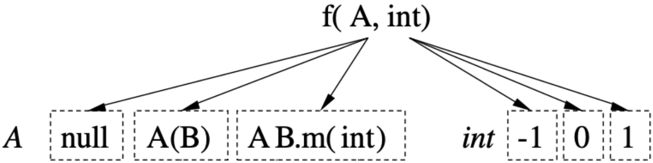


Figure: `A(B)` and `B.m(int)` are methods returning type `A`. **null** is predefinied for reference types. Taken from [2, p. 1030].

Parameter-Graph Example

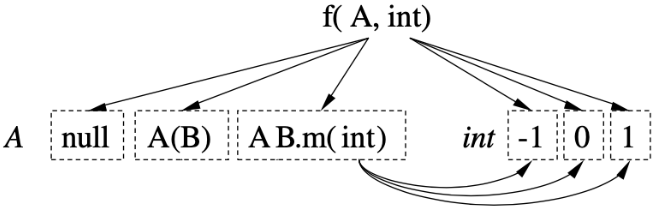


Figure: Again, type `int` uses pre-sets. Taken from [2, p. 1030].

Parameter-Graph Example

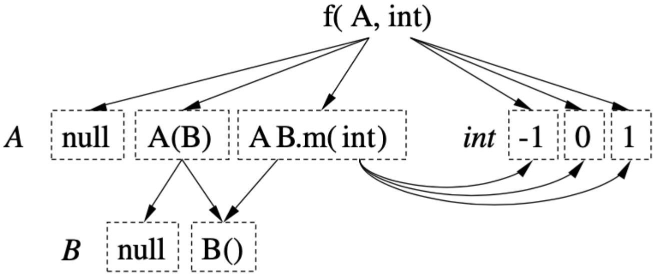


Figure: Traversable parameter-graph to create different parameter combinations. Taken from [2, p. 1030].

Parameter-Graph Example

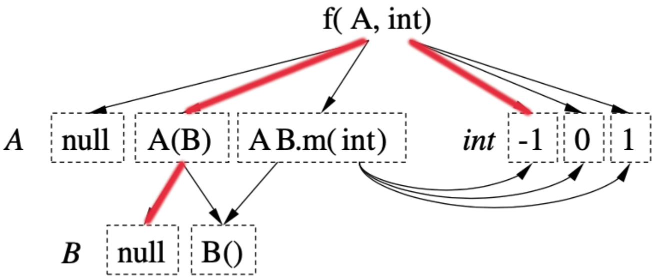
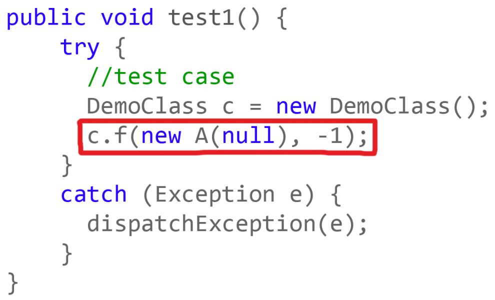


Figure: For example, `f(new A(null), -1)` would be a syntactical valid method call. Taken from [2, p. 1030].

Test Case Execution



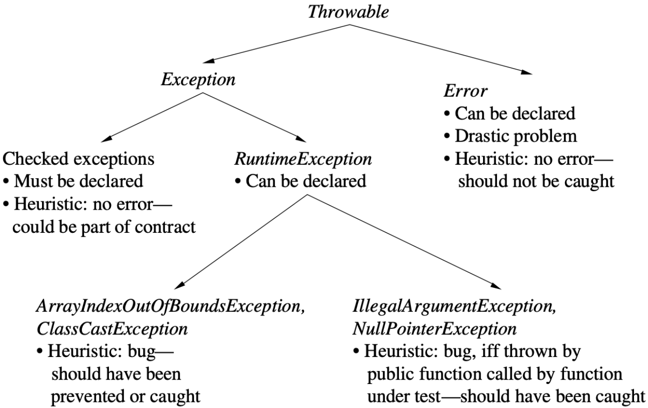
```
public void test1() {
    try {
      //test case
      DemoClass c = new DemoClass();
      c.f(new A(null), -1);
    }
    catch (Exception e) {
      dispatchException(e);
    }
}
```

Figure: Based upon [2, p. 1032].

Heuristic Approach

- JCrasher catches all exceptions
- Tell bugs and violated preconditions apart
- Exception indicates either
  - Violation of code's preconditions (no bug)
  - Method failed to handle exception in subroutine (bug)
- Actions to take
  - Bug → report exception to JUnit
  - Expected exception → ignore

Exception Filtering Hierarchy



Throwable

Exception

Error
- Can be declared
- Drastic problem
- Heuristic: no error—should not be caught

Checked exceptions
- Must be declared
- Heuristic: no error—could be part of contract

RuntimeException
- Can be declared

ArrayIndexOutOfBoundsException, *ClassCastException*
- Heuristic: bug—should have been prevented or caught

IllegalArgumentException, *NullPointerException*
- Heuristic: bug, iff thrown by public function called by function under test—should have been caught

Figure: Java sub-class hierarchy of `java.lang.Throwable`. Taken from [2, p. 1033].

Example: Bug

```
// testing with pos = 5;
public void method(int pos) {
   int[] myArray = {2, 4, 8};
   // ...
   myArray[pos];  // ArrayIndexOutOfBoundsException
}
```

Example: No Bug

```
// testing with i = 5;
public void method(int i) throws CustomException {
   if (i < 10)
      throw new CustomException();
   // ...
}
```

Possible Side-Effects of Test Cases

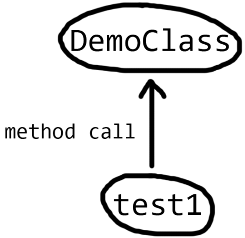


Figure: Test case executes method on an object `DemoClass`.

Possible Side-Effects of Test Cases

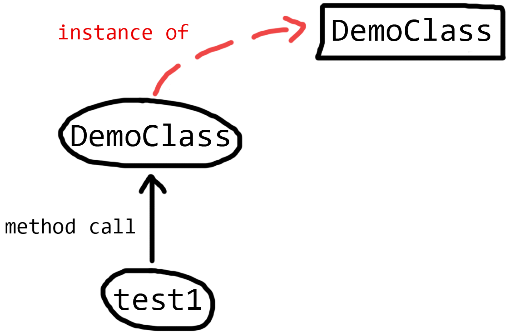


Figure: Object `DemoClass` is instance of class `DemoClass`.

Possible Side-Effects of Test Cases

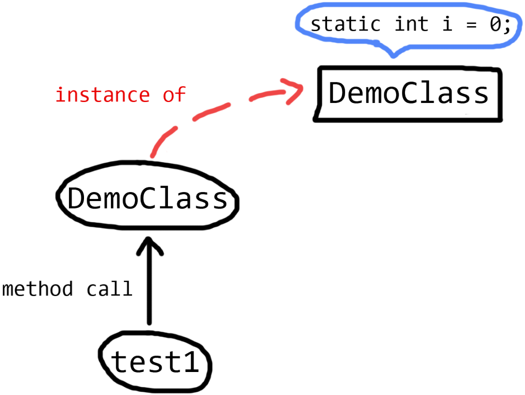


Figure: Class containing static variables.

Possible Side-Effects of Test Cases

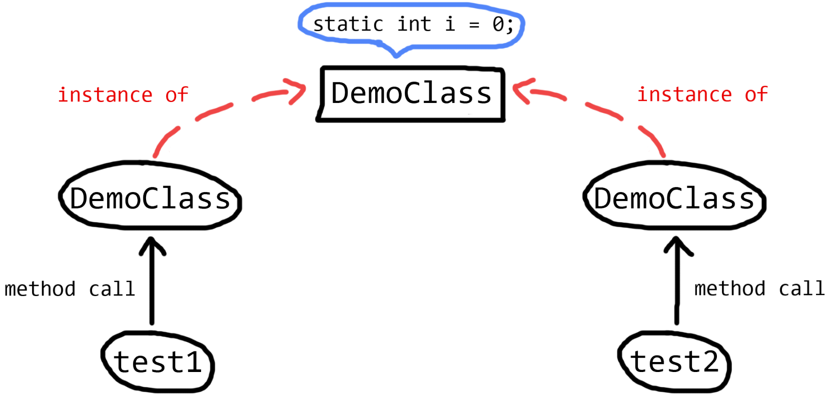


Figure: All test cases use the same class object at runtime as there is only a single JVM instance.

Possible Solutions For Side-Effects

1. Using multiple JVMs
2. Each test case operates on a new copy of a class object
3. Use same class object over again but reset its state after execution

Resetting Static State

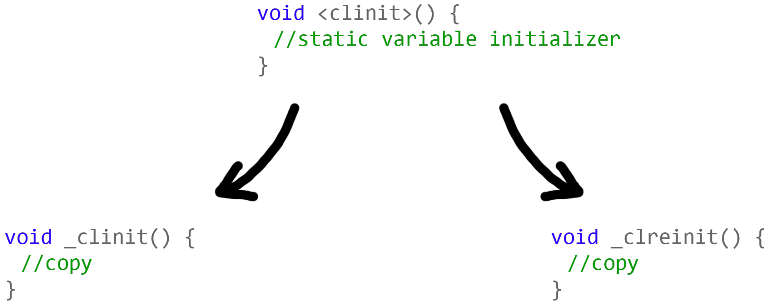
- Imitation of JVM’s class initialization algorithm
  - Re-initialize already used classes
- Modification to JUnit
  - Replace class loader with custom one
  - Modify bytecode of a class before loading
- Re-initialization at end of each test case execution

Implementing Re-initialization

```
void <clinit>() {
 //static variable initializer
}
```

Figure: Variable initializer of static fields are compiled into `<clinit>()`.

Implementing Re-initialization



```
void <clinit>() {
 //static variable initializer
}
```

```
void _clinit() {
 //copy
}
```

```
void _clreinit() {
 //copy
}
```

Figure: Copy `<clinit>()` to callable methods `_clinit()` and `_clreinit()`.

Implementing Re-initialization

```
void <clinit>() {
 //static variable initializer
}
```

```
void _clinit() {
 //copy
}
```

```
void _clreinit() {
 //modified copy
 //do not reset constants
}
```

Figure: Modification of `_clreinit()` to not reset final static fields.

Implementing Re-initialization

```
void <clinit>() {
 _clinit();
 jCrasher.register(this.class);
}
```

```
void _clinit() {
 //copy
}
```

```
void _clreinit() {
 //modified copy
 //do not reset constants
}
```

Figure: Modification of `<clinit>()` to make this class resettable by JCrasher.

Evaluation of Re-initialization

- Fast and (nearly) correct
- Differs from original Java initialization
  - Order of class re-initialization depends on order of their original initialization
  - Eager re-initialization instead of lazy
- Cyclic class dependencies possible

Summary and Outlook

- Automated robustness testing for Java programs
- Designed with practical usage in mind
- Parameter-Graph and random testing
- Heuristic
  - Differentiate bugs from input violations
- Resetting static class state
  - Avoid side-effects between test cases
- Development of follow-up tools

References

[1] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84.

[2] C. Csallner and Y. Smaragdakis. "JCrasher: an automatic robustness tester for Java". In: *Software – Practice & Experience* 34 (11) (2004), pp. 1025–1050.

PADERBORN UNIVERSITY

PADERBORN UNIVERSITY

JCRASHER: AN AUTOMATIC ROBUSTNESS TESTER FOR JAVA

SEMINAR: SOFTWARE TESTING

Theo Harkenbusch

Paderborn, 27.11.19