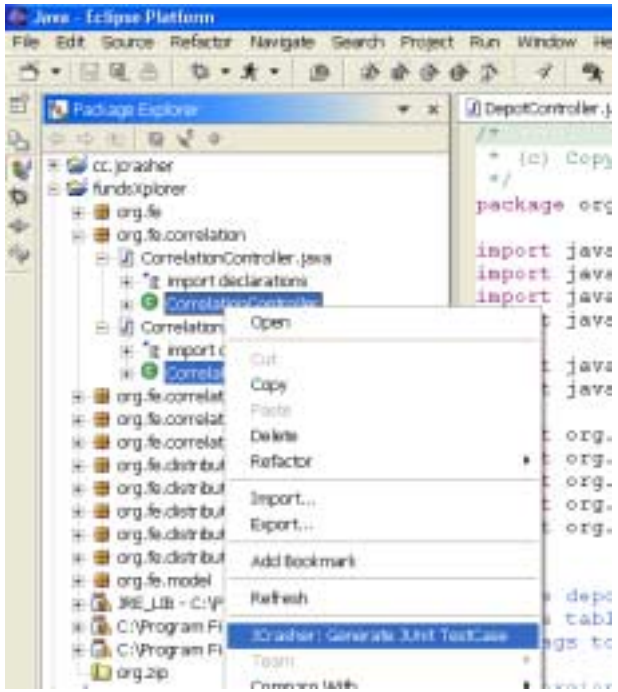
SPARC Brown Bag, Mo. 6. October 2003

JCrasher: An Automatic Robustness Tester for Java

Christoph Csallner
csallner@cc.gatech.edu

Yannis Smaragdakis
yannis@cc.gatech.edu



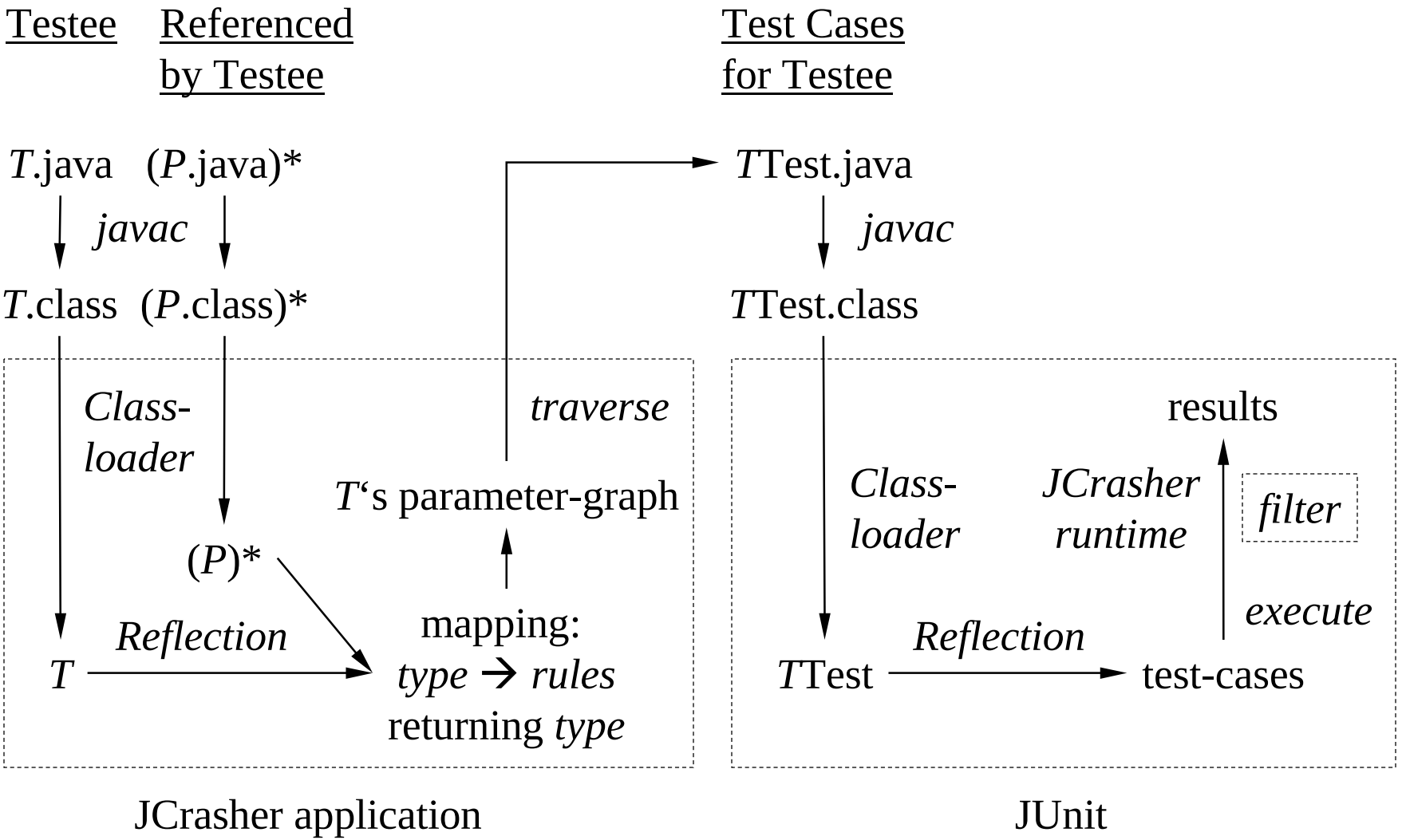
What Do We Mean by Robustness?

- Assume testee written in an object-oriented language.
- General robustness quality goal
  A public method should not throw an unexpected runtime exception when encountering an internal problem, regardless of the parameters provided.
  - Instead: should handle internal problem and throw for example an IllegalArgumentException.
  - Goal does not encode knowledge about the testee's domain.
  - The same robustness goal applies to all testees.
- General function to determine testee's robustness:
  exception type → {pass | fail}

Testing for Robustness

- What is testing?
  - Write test case method, which creates parameters, calls testee with parameters, compares result with expected result, and reports
  - xUnit frameworks do not contain or retrieve quality goals of testees.
    → xUnit frameworks cannot automate test case generation.
- Robustness testing: huge parameter space
  - Example: m(int, int) has $2^{8*4} * 2^{8*4}$ parameter combinations
  - Covering all parameter combinations is generally impossible
- You might not need all parameter combinations to cover all control paths through the method that throw an exception
  - Pick a random sample
  - Control flow analysis on byte code (JABA) could derive parameter equivalence classes

JCrasher Automates Robustness Testing

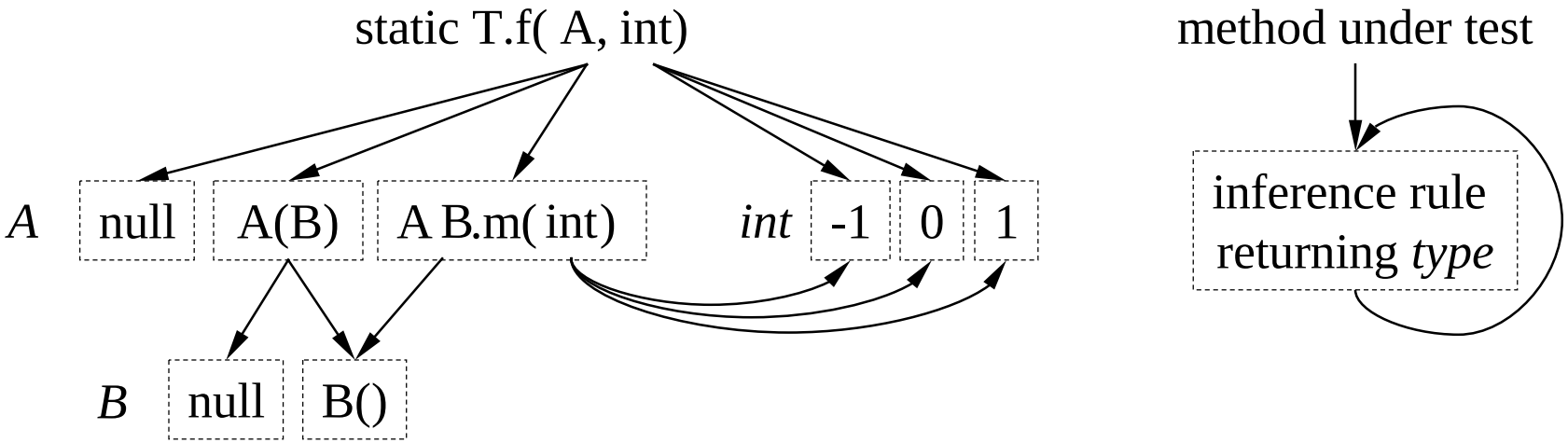


Collect Type Inference Rules

- Search class under test for inference rules
- Transitively search referenced types
- Inference rules
  - Method $T.m(P_1, P_2, ..., P_n)$ returns X:
    $X \leftarrow T, P_1, P_2, ..., P_n$
  - Sub-type Y {extends | implements} X:
    $X \leftarrow Y$
  - Constructors and preset values are implicitly known
- Add each discovered inference rule to mapping:
  $X \rightarrow$ inference rules returning X

Generate Test Cases For a Method

Parameter Graph for Method T.f(A, int)



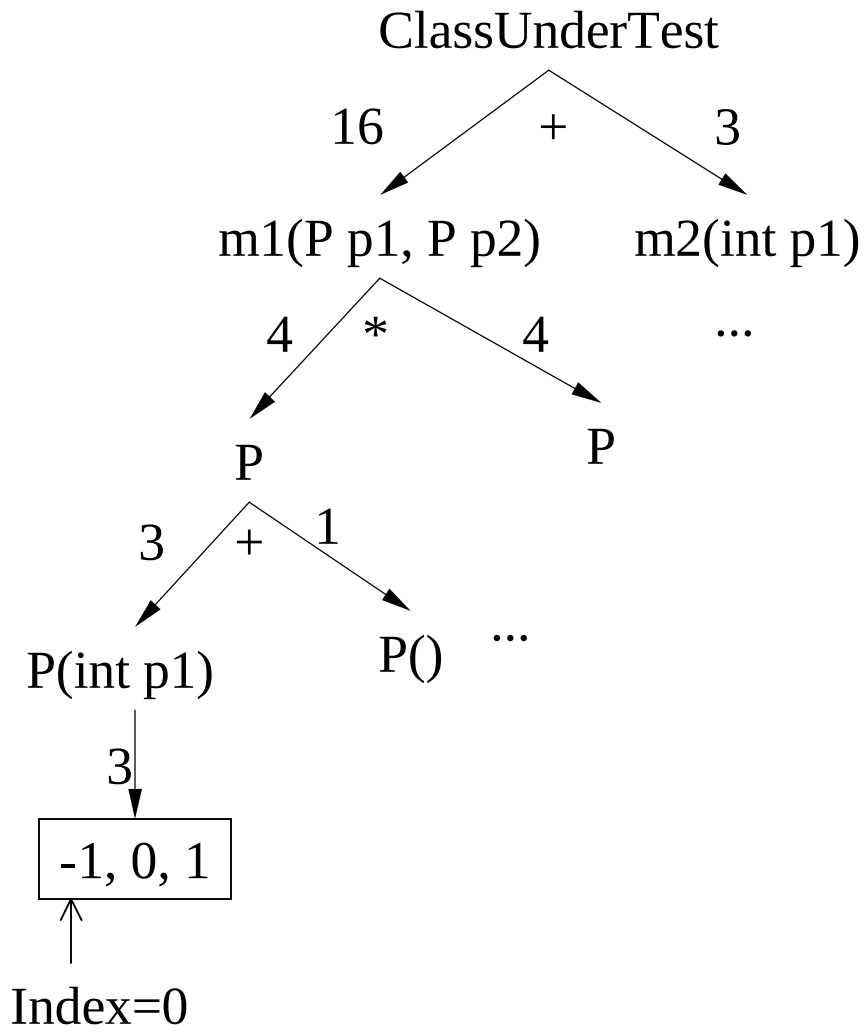
Test Cases:

f(null, -1), f(null, 0), f(null, 1),
f(A(null), -1), ...,

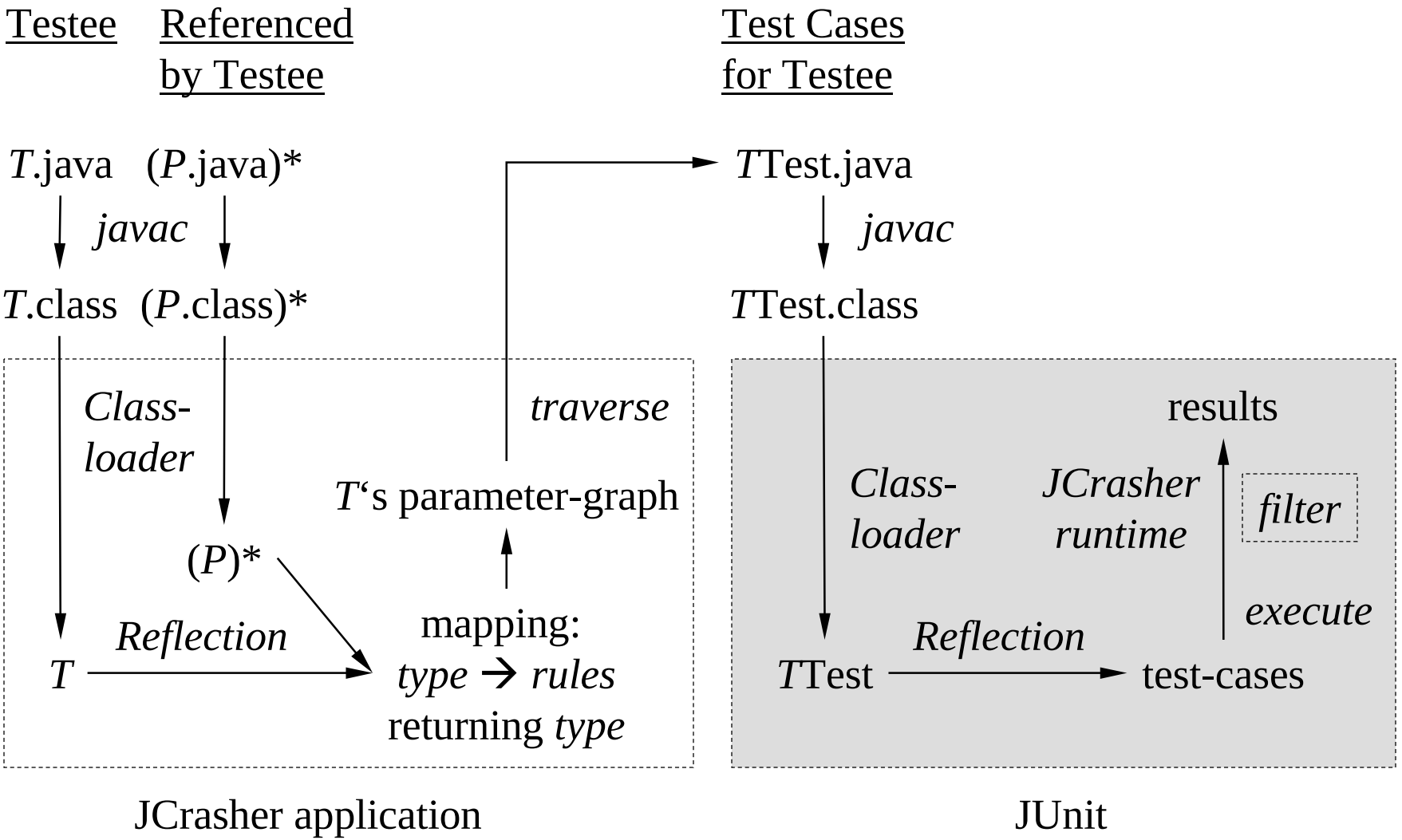
Generate a Random Sample of Test Cases

- Tree never completely in memory—implicitly represented by mapping: type → rule

1. Build tree of depth *n* = 2
2. Determine number of possible test cases = 19
   1. Inference rule: product of the parameter sub spaces
   2. Parameter: sum of the inference rule sub spaces
3. Pick a random sample, for example 1, 3, 16
4. Derive test cases:
   1. 1<16 → m1
   2. 1=0*4+1*1 → m1(-1, 0)



Test Case Execution and Exception Filtering



Test Case Execution and Exception Filtering

- JCrasher generated test cases look like:

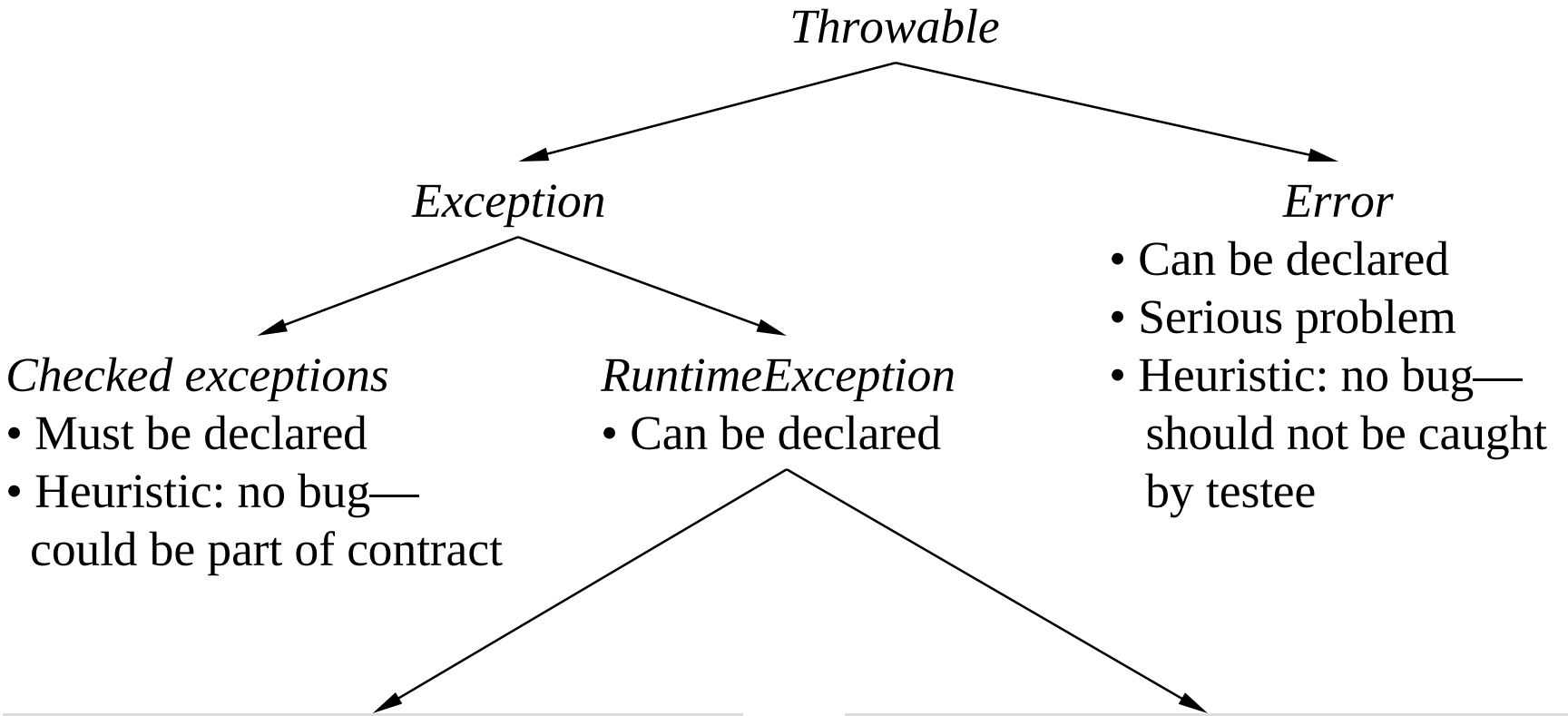
```
public void test1() throws Throwable {
   try { /* test case */ }
   catch (Exception e) {
       dispatchException(e);   /* JCrasher runtime */
   }
}
```

- JCrasher runtime catches all exceptions and uses heuristics to decide whether the exception is a
  - Bug of the testee → pass exception on to JUnit
  - Expected exception → suppress exception

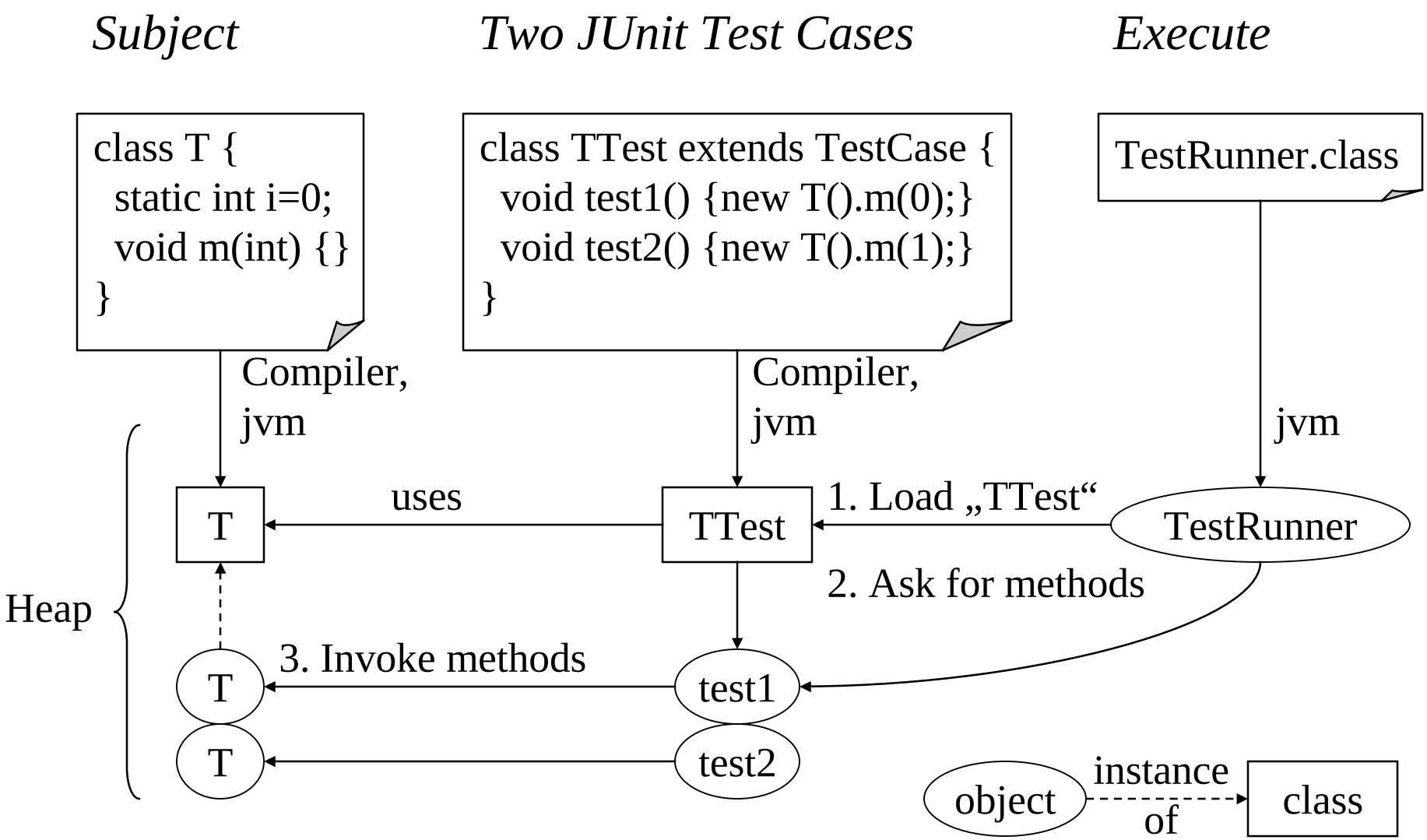
Exception Filter Heuristics

- An exception indicates one of the following
  - As a part of the method's contract, the method under test signals a violated precondition—no bug.
  - The method under test has run into an unforeseen problem and is terminated because it has not handled an exception thrown by code invoked by the method—bug.
- JCrasher uses heuristics to distinguish between bugs and violated preconditions.

Exception Filter Heuristics



Test Case Execution: Problem of Side-Effect between Test Cases



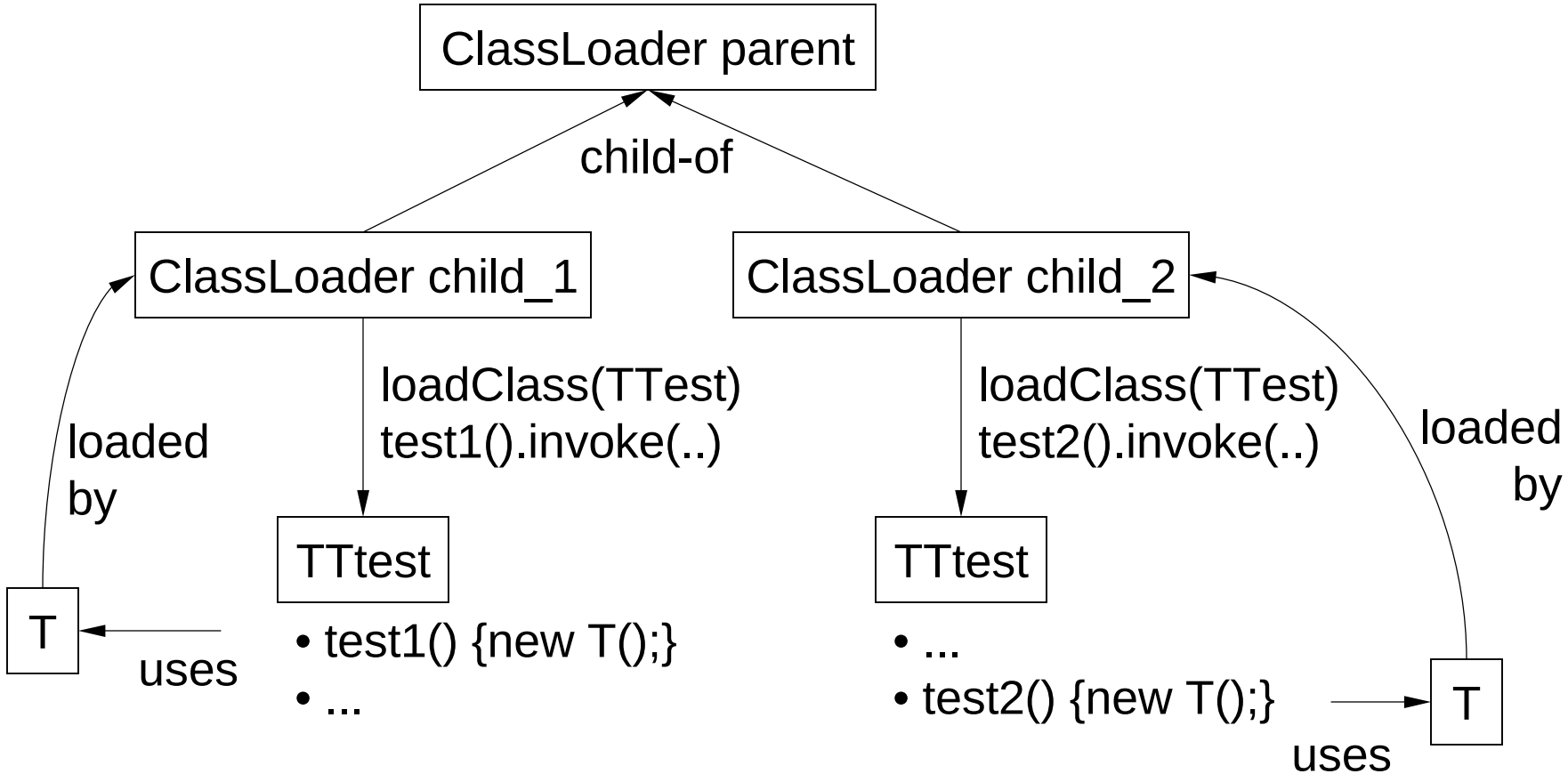
Problem of Side-Effects Between Test Cases

- Cause
  - All test cases refer to the same object representing a class at runtime
  - Each test case can change this object's state
  - Subsequent test cases may execute on modified state
- Makes it hard to understand result of test cases
- Two solution approaches
  - Provide a separate copy of a class object for each test case
  - Stick to same class object for each test case, but reset its state after a test case has executed
- Limitation of solutions: change to external state cannot be undone, for example files, databases

Separate Copies of a Class Object

- Naive approach: separate JVM instances
  - Script to start each test case in its own JVM instance
  - High overhead of JVM initialization
  - Contradicts the JUnit execution model
- Separate class objects in same JVM instance
  - Class at runtime defined by
    tupel (fully qualified name, class loader)
  - Given the same fully qualified class name, each class loader
    - Independently loads the class's byte code if not already done so by itself or a parent class loader
    - Keeps its own class state

Separate Class Objects in Same JVM Instance with Hierarchy of Class Loaders



Reset State of All Referenced Classes After a Test Case Has Been Executed

- The JCrasher runtime imitates the JVM's Class Initialization Algorithm
- Requirements
  - A list of the referenced classes in the order in which they have been initialized.
  - The ability to reset the values of the static fields of each of these classes to the default all-zero value (null, 0, false).
  - The ability to execute the variable initializer of each static field. A class's variable initializers are compiled into the class's `<clinit>()` method.

Implementation

- Modify JUnit to use a class loader that changes a class's byte code before loading it [BCEL]
  - Copy static initializer method `<clinit>()`to user-accessible methods `_clinit()` and `_clreinit()`
  - Modify `_clreinit()` to avoid resetting static constants.
  - `<clinit>()` is changed to first call `_clinit()` and then appends the class to the list of classes to be reset
- JCrasher runtime method after a test case has executed
  - Set static fields of listed classes to all-zero values.
  - Call `_clreinit()` of each listed classes

Discussion of Re-Initialization Approach

- Benefits
  - Faster: eliminate re-loading of classes for second, third, …, *n*-th test case.
  - Less memory needed: all test cases reference same class object
- Weakened semantics
  - Initialization order of first test case fixes re-init order.
  - Eager initialization instead of Java's lazy initialization.
  - Incorrect if static initializer depends on previous state changes.

    ```
    class A {static int a = 100;}
    class B {static int b = A.a;}
    ```

    `A` referenced before `B`: `a=100; a=???; b=a`

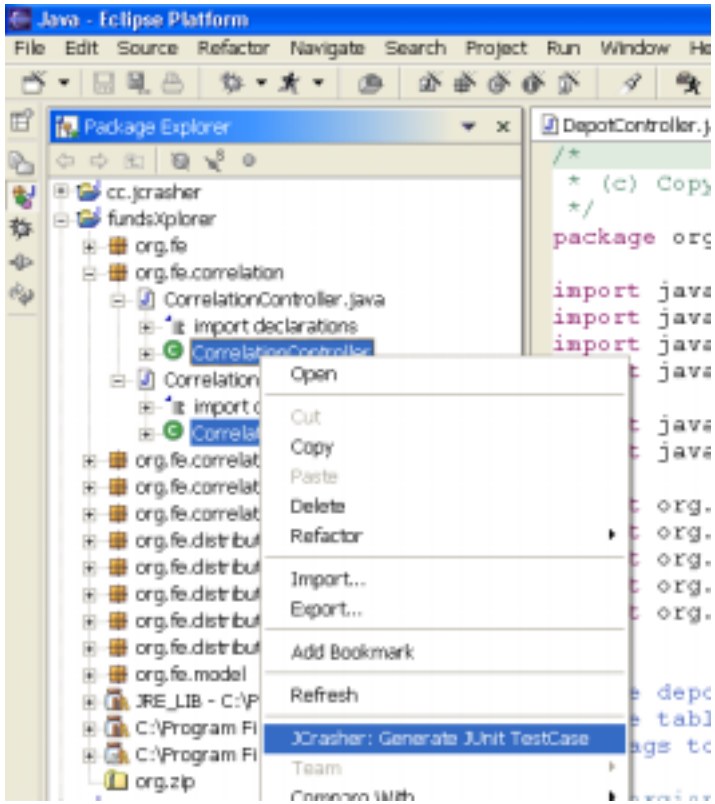
    `B` referenced before `A`: `a=100; b=a`
  - False positives: bad style, rare—tradeoff with benefits
  - False negatives: inherent to random testing anyways

Performance

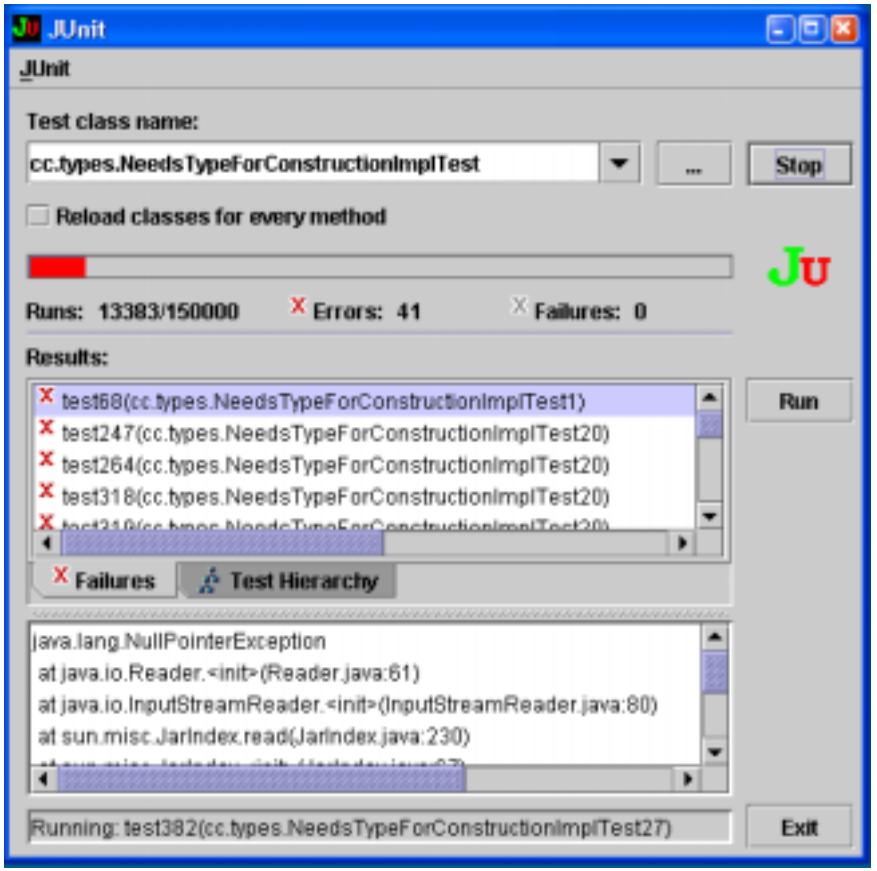
- Summary, testee executing only a few instructions
  - Restarting JVM approach = 100 percent
  - Multiple class loader approach = 45 percent
  - Resetting class state approach = 2 percent
- Details
  - Start JVM and execute a trivial JUnit test = 270 ms.
    - Load JUnit classes, test class, testee, and run JUnit code.
    - Starting a JVM and execute a trivial method = 170 ms.
    - Average time to execute a test with JUnit = 5 ms.
  - Multiple class loader approach, reload a single class = 120 ms.
    - Multiple class loader approach saves the 270 ms of JVM and JUnit startup.
    - Going to disk and reload a single class file reduces benefit to about 150 ms.
  - Re-initialization approach
    - JCrasher machinery to reinitialize a class with 10 static fields = 0.06 ms.
  - Test environment: 1.2 GHz Intel mobile Pentium 3 processor with 512 MB RAM running Windows XP and a 12 ms avg., 100 MB/s hard disk.

JCrasher Can Be Integrated Into Eclipse

JCrasher as eclipse plug-in



JUnit swingUI



Conclusions And Future Work

- General definition of robustness testing
- JCrasher automates robustness testing
  - Cheap way to supplement structured testing
- Undesired side-effect between JUnit test cases
- Two approaches to reset class state
  - Slow and correct: multiple JVM instances or class loaders
  - Fast and almost correct: imitate JVM's class initialization
- Integration with popular tools Eclipse and Junit
- Replace/ compare test case selection with control flow analysis (=JABA) based test case selection.

References

[BCEL]
http://jakarta.apache.org/bcel/

[eclipse]
http://www.eclipse.org

[JCrasher]
http://www.cc.gatech.edu/~csallnch/jcrasher

[JTest]
http://www.parasoft.com/jsp/products/home.jsp?product=Jtest

[JUB]
http://jub.sourceforge.net/

[JUnit]
http://www.junit.org