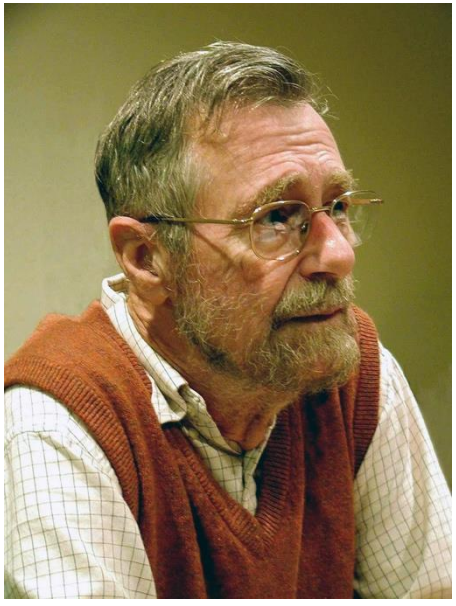
CMSC 433
Programming Language Technologies and Paradigms

Testing

Edsger W. Dijkstra



Program testing can be used to show the presence of bugs, but never to show their absence!

Verification vs Testing

- Verification and Static Analysis are great
  - Lots of interesting ideas and tools

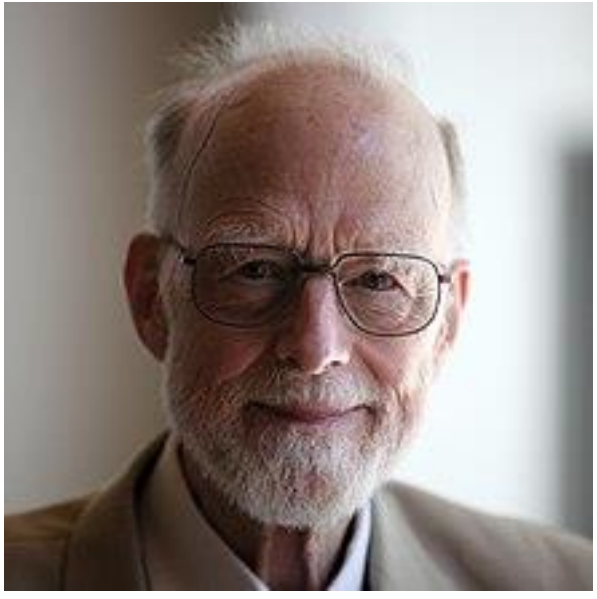
- But can developers use it?
  - Formal verification of computer programs are hard.
  - Commercial static analysis tools have a huge code mass to deal with developer confusion, false positives, warning management, etc.

“Software testers always go to heaven; they’ve already had their fair share of hell.”

(*Anonymous*)

Tony Hoare

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.



Simple Hashmap

```
let empty v = fun _-> 0;;
let update m k v = fun s->if k=s then v else m s

let m = empty 0;;
let m = update m "foo" 100;;
let m = update m "bar" 200;;
let m = update m "baz" 300;;
m "foo";; (* 100 *)
m "bar";; (* 200 *)
let m = update m "foo" 101;;
m "foo";; (* 101 *)
```

Testing is important

- Estimated **50%** of programmers time spent on finding and fixing bugs.

- Testing is not the only, but the primary method that industry uses to evaluate software under development.

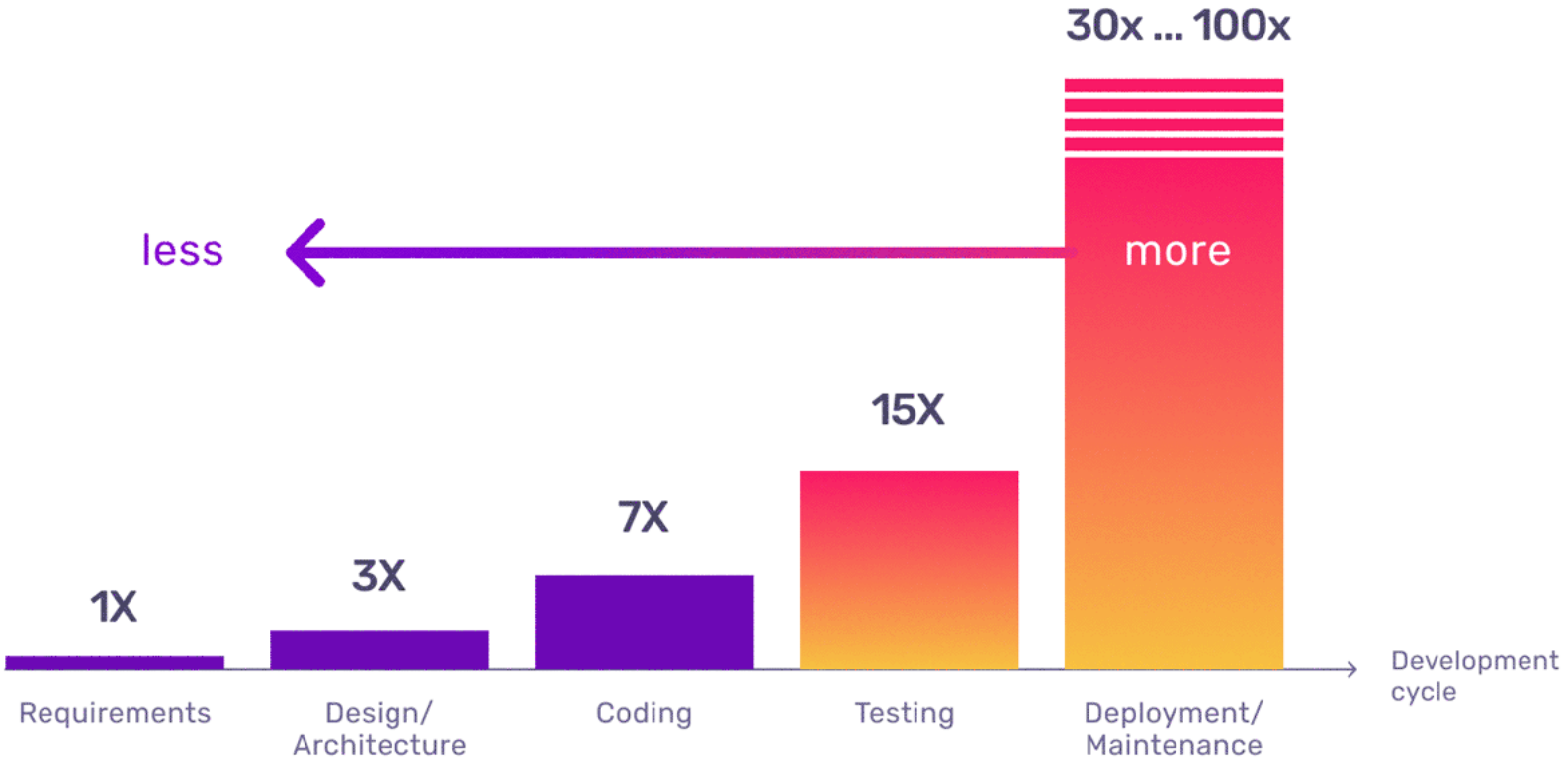
Testing is important

- Ideas and techniques of testing have become essential knowledge for all software developers.

- Expect to use the concepts presented here many times in your career.

- A few basic software testing concepts can be used to design tests for a large variety of software applications.

Cost of Defects

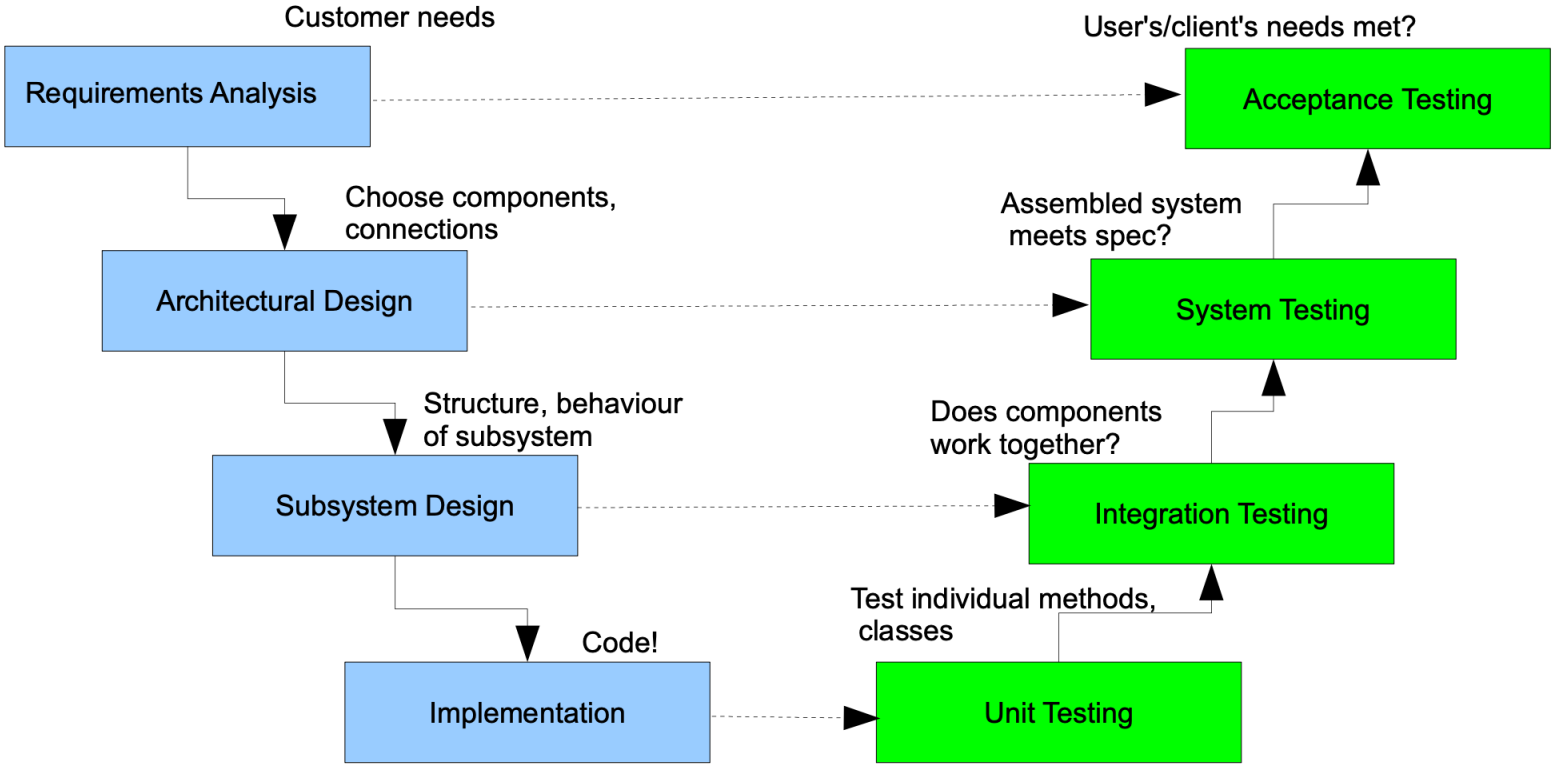


Testing Scale

- Unit testing: testing individual classes/functions
- Integration Testing: testing packages/ subsystems
- System tests: testing the entire system

Unit Test Example: https://github.com/cedar-policy/cedar/blob/main/cedar-policy-core/src/evaluator.rs

V Model



There are many variants

Testing Process

- Test first: Test driven development (TDD)
  - Write tests before the code
  - Write the code to pass the test
- Test after
  - Check whether existing code passes the tests
- Iteration
  - Retesting
  - Refactoring

Testing: Purpose

- Functional testing
- Performance Testing
- Security testing
- Usability testing
- Availability testing

Property-based Testing

- a framework that repeatedly generates random inputs, and uses them to confirm that properties hold

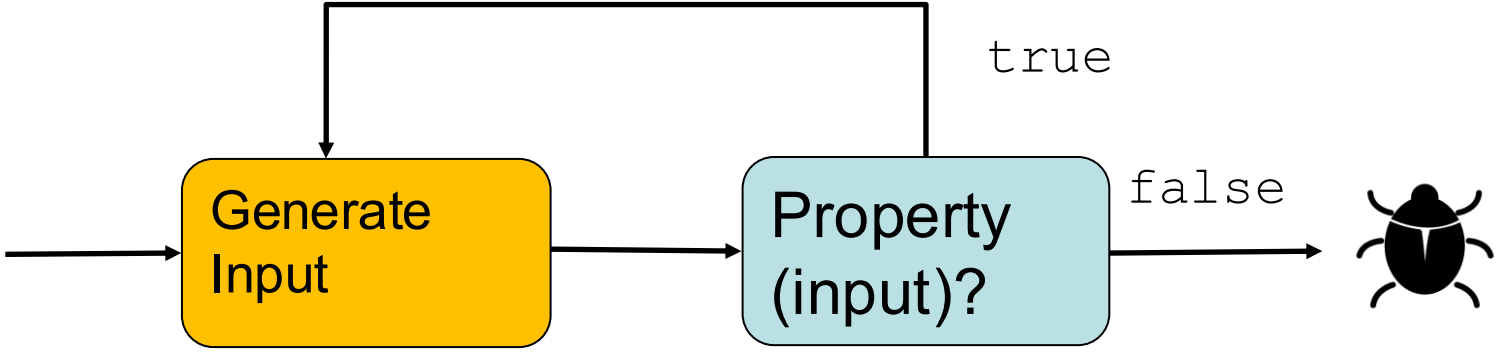
```
testList(l1:List<T>) {

   l2 = reverse(reverse(l1))
   assertEquals(l1, l2);
}
```

Repeatedly generate input randomly

Confirm the property holds for the given input

QuickCheck: Property-Based Testing

- QCheck tests are described by
  - A generator: generates random input
  - A property: `bool`-valued function



Shrinking

- the process of automatically simplifying a failing test case to produce the smallest or simplest possible input that still triggers the failure.
- Example:

```
[123, -999999999999, 5, 0, 5]
→ [123, -1, 5, 0, 5]
→ [123, -1]
→ [-1]
```

How Shrinking Works

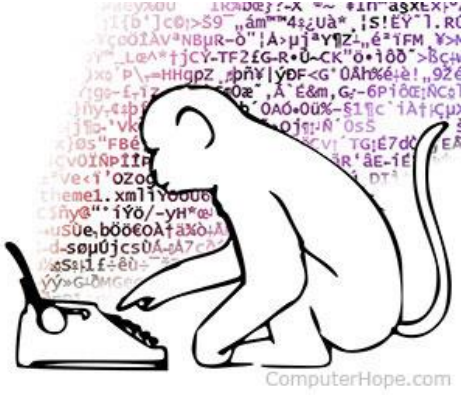
- Shrinking strategies depend on the data type:
  - **Numbers:** Try smaller magnitudes or 0.
  - **Lists:** Try removing or shortening elements.
  - **Strings:** Try shorter substrings or simpler characters.
  - **Custom objects:** Shrink their fields recursively.
    - Trees:
      - Shrink the value at each node
      - Remove subtrees (reduce branching)
      - Replace a node with one of its subtrees
      - **Shrink recursively within subtrees**

Fuzz Testing

- Fuzz testing is a quality assurance technique used to discover coding errors and security loopholes in software, operating systems or networks.

- It involves inputting massive amounts of random data, called fuzz, to the test subject in an attempt to make it crash.

- If a vulnerability is found, a software tool called a fuzzer can be used to identify potential causes.



Test Generators

- generator is a component or algorithm that creates input data for the system under test.
- Instead of just using *completely random bytes*, a generator *produces structured, meaningful inputs* — often closer to what the program actually expects.
  - Defines the structure of valid inputs (like grammar, JSON, XML, binary protocol, etc.)
  - Produces variations that test edge cases (small vs. huge values, missing fields, weird nesting)
  - Maintains validity, so the target program doesn't reject everything outright

Mutation Testing

- Mutation testing involves modifying a program in small ways.

```
if (a && b)
   { c = 1; }
else
{ c = 0; }
```

The condition mutation operator would replace && with || and produce the following mutant:

```
if (a || b)
   { c = 1; }
else
   { c = 0; }
```

Mutation Operators

- Many mutation operators have been explored by researchers. Here are some examples of mutation operators for imperative languages:
  - Statement deletion
  - Statement duplication or insertion, e.g. goto fail;
  - Replacement of boolean subexpressions with *true* and *false*
  - Replacement of some arithmetic operations with others, e.g. + with *, - with /
  - Replacement of some boolean relations with others, e.g. > with >=, == and <=
  - Remove method body
  - …

Code coverage

- **Function coverage** – Has each function been called?
- **Statement coverage** – Has each statement been executed?
- **Branch coverage** – Has each branch of each control structure (such as in *if* and *case* statements) been executed?
- **Condition coverage** (or predicate coverage) – Has each Boolean sub-expression evaluated both to true and false?
- Many more

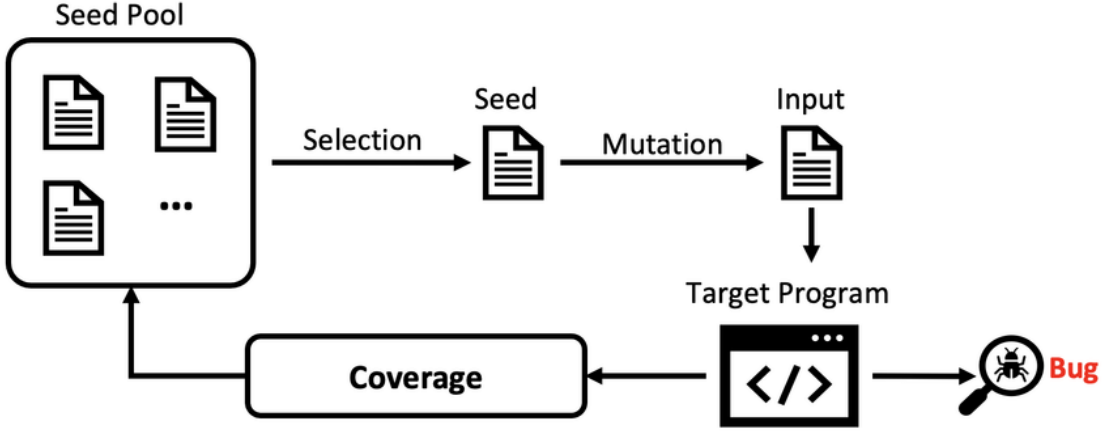
Coverage Based Randomized Testing

- An approach to software testing that combines **random test generation** (like fuzzing) with **code coverage feedback** to intelligently explore more of a program's behavior.

- Instead of just generating random inputs blindly, coverage-based randomized testing *observes what parts of the program each test executes* — and then uses that feedback to guide future test generation
  - Maximize code coverage by generating inputs that explore *new* execution paths.

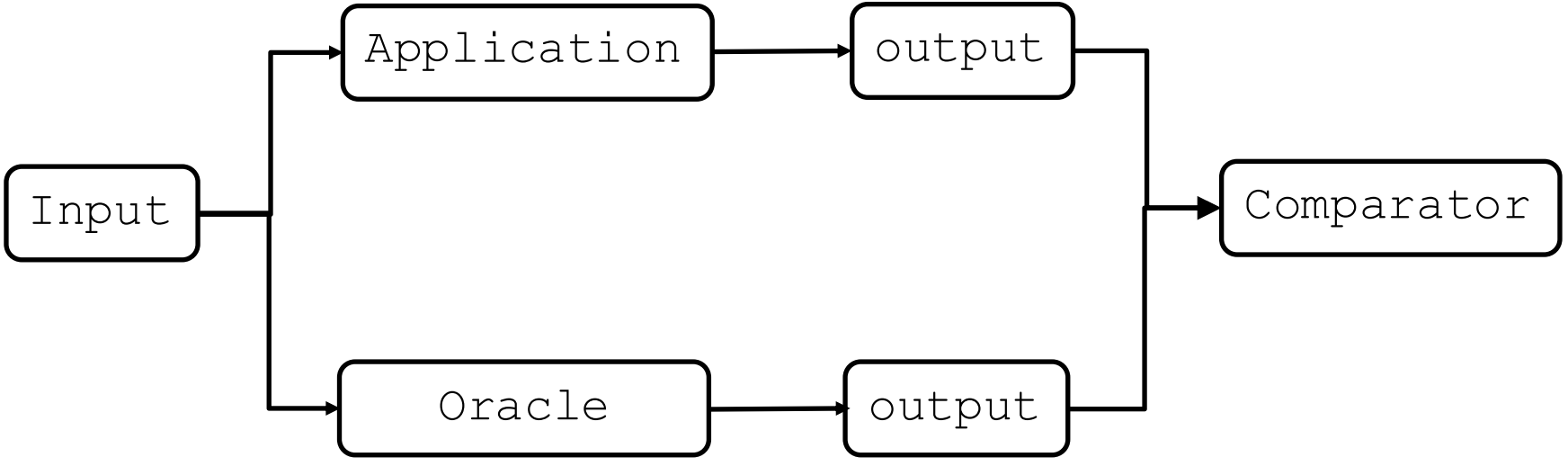
Coverage Based Randomized Testing

- Generate random input (e.g., random file, data, request)
- Run the program under instrumentation
- Measure coverage (which functions, lines, or branches were executed)
- Keep interesting inputs — those that cover new code paths
- Mutate those “interesting” inputs to explore further variations
- Repeat!



Differential Testing

is a software testing technique that detects bugs by comparing the outputs of multiple implementations of the same functionality.



Differential Testing Example: Csmith

- Generates random, valid C programs.
- Compiles each program with multiple C compilers and compares results.
  - `gcc test.c -o a.out`
  - `clang test.c -o b.out`
  - `tcc test.c -o c.out`
- Result: Found hundreds of bugs in GCC and Clang

Property Based Testing Demo

- Setting Up Junit-QuickCheck

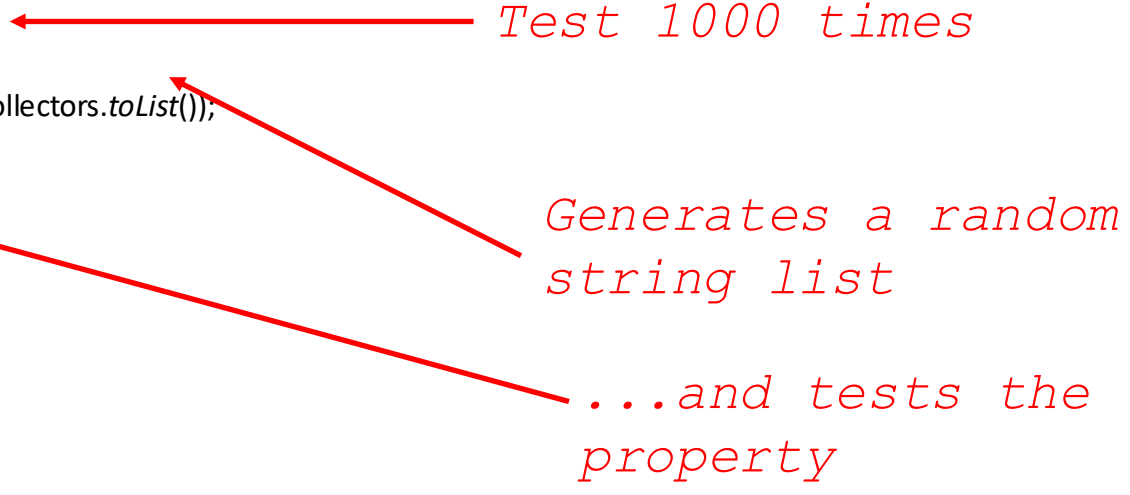
- Maven

```
<dependency>
<groupId>com.pholser</groupId>
<artifactId>junit-quickcheck-core</artifactId>
<version>0.7</version>
</dependency>
```

- Eclipse:
  - Add the jar files

Let's Test Our Property

```
@RunWith(JUnitQuickcheck.class)
public class PBT {
  @Property (trials = 1000)
  public void testList(List<String> l1) {
    List<String> l2 = l1.stream().collect(Collectors.toList());
    Collections.reverse(l2);
    Collections.reverse(l2);
    assertEquals(l1, l2);
  }
}
```



Buggy Reverse

```
Reverse(List<?> l){ return l} //returns the same list
```

The property did not catch the bug!

```
reverse((reverse (l))) == l
```

A simple unit test would catch the bug

assertEquals (`reverse ([1,2,3]), [3,2,1])`

Another Property

```
testRev (List<Integer>l1, Integer x, List<Integer l2){
   assertEquals(
      rev (l1 ++ [x] ++ l2) , rev l2 ++ [x] ++ rev l1
   )
}
```

```
rev [1,2]++[3]@[4;5] = rev [4,5] ++ rev [3] ++ rev [1;2]
```

Junit-QuickCheck

- **junit-quickcheck: Property-based testinga, JUnit-style**

  `github:` `https://github.com/pholser/junit-quickcheck`
- Documentation:
  - https://pholser.github.io/junit-quickcheck/site/1.0/

- Generator: random generators
- Shrink: Producing “smaller” values
- Seed: source of randomness

Demo

https://github.com/anwarmamat/cmsc330/tree/master/java/junit_quickcheck