Topics for this Lecture

Automatic Test Generation

- Random Testing
- Evolutionary Search
- Symbolic Execution

Random Testing (RT)

- Create program inputs randomly
- Cheap & easy to implement
- Easy to understand
- Works pretty well in many cases. Actually, used in industry
- If a systematic approach is not better than random testing, it is not useful

Random Testing in the Real World

- Random testing (usually called "fuzzing") is highly effective
  - Mozilla and Google use random testing to detect critical security flaws in JavaScript engines and browsers
    - Search for Google's "ClusterFuzz"
- Extremely effective for finding bugs in C compilers, including over 400 in GCC and LLVM
  - Search for the Csmith compiler testing project
- Used to find bugs in Apache commons, Java core libraries, other widely used code
- At NASA, random testing is used to test file systems for missions with costs well over $100M

How to Write a Simple Random Tester

1. **Identify** the class under test (**CUT**)
2. **Identify** all the dependencies (**parameters**)
3. **Identify** All methods, constructors, fields
4. Write code to **randomly choose** a **constructor** of the CUT to generate an instance of the CUT.
5. Write code to **generate random inputs** for the constructors
6. Write code to **randomly choose** a **method** of the CUT
7. Write code to **generate random inputs** for the chosen method
8. Invoke the method (execute the method)
9. Check if stopping criterion is not satisfied go to step 4.

- *Note: steps 5 and 7:*
  - If the input is a primitive data type,
    - generate a random primitive value
  - If the input is a reference, choose randomly among:
    - The null value
    - The constructor with no arguments (if it exists)

a Simple Random Tester for Stack

```
public class StackRandomTest {
    public static int RandInt(Random random){
       int n = random.nextInt();
        return (int) n;
        }
    public static String RandMethod(Random random){
        String[] methodArray =
       new String[] {"push","pop","top"};//

       int n = random.nextInt(3);

        return methodArray[n] ;
        }
```

a Simple Random Tester for Stack

```
@Test
public void test() {
 //Instead of a fixed number (100), we can choose time for your test budget,
such as 5, 10, or 60 minutes !
for (int k = 0; k < 100; k++){
     Stack st = new Stack();
      long randomseed = System.currentTimeMillis();
      Random random = new Random(randomseed);

      for (int i = 0; i < 100; i++) {
           String methodName = StackRandomTest.RandMethod(random);
           int n = StackRandomTest.RandInt(random);
           try {
             if (methodName.equals("push")){
                 st.push(n);
                 assertTrue(n==st.top());
              }
              else if (methodName.equals("pop")){
                    st.pop();
              }
              else if (methodName.equals("top")){
                      int l=st.top();
                      assertTrue(l==st.top());
               }
             } catch (EmptyStackException e) {
                 // that's fine
     }} }
```

A Simple Random Tester for Container

```
public class ContainerRandomTest {
     private static final int MAX_VALUE=10;
     private static final int NUM_TESTS=10;

System.out.println("Start testing...");
for (int k = 0; k < 100; k++) {
     Container c = new Container();
     HashSet<Integer> ref=new HashSet<Integer>();
     long randomseed = System.currentTimeMillis();
     Random random = new Random(randomseed);

     for (int i = 0; i < NUM_TESTS; i++) {
          String methodName = ContainerRandomTest.RandMethod(random);

          int n = ContainerRandomTest.RandInt(random);

            if (methodName.equals("put")){
               int r1=c.put(n);
               int r2= ref.add(n)?1:0;
               assertEquals(r2, r1);
           }
           else if (methodName.equals("get")){
                    int r1=c.get(n);
                    int r2= ref.contains(n)?1:0;
                    assertEquals(r2, r1);
               }
               else if (methodName.equals("remove")){
                    int r1=c.remove(n);
                    int r2= ref.remove(n)?1:0;
                    assertEquals(r2, r1);
                    }
                    else if (methodName.equals("size")){
                              int r1=c.size();
                              int r2= ref.size();
                              assertEquals(r2, r1);
                         }
     }//for i
} //for k
System.out.println("done!");
```

```
public static String RandMethod(Random random){
    String[] methodArray = new String[]
{"put","get","remove","size"};//

      int n = random.nextInt(4);// get a random number
between 0 (inclusive) and  4 (exclusive)

        return methodArray[n] ; // return the method name
 }
```

```
public static int RandInt(Random random){
      int n = random.nextInt(MAX_VALUE);// get a random number
between 0 (inclusive) and  MAX_VALUE=10 (exclusive)
        return (int) n;
}
```

Recipe for Refining a Random Tester

1. Gather code coverage
   - Is everything interesting being covered?
   - Is important code not covered?
2. Adjust the code to generate inputs
   - Try to “stay random” but shift the probability space
   - Augment random with fixed inputs of interest
3. Break the code and see if your tests detect the problem
   - If not, why not?
4. Improve your oracle code (i.e., assertions) until all problems that should be caught *are* caught
5. Repeat until coverage and “fake bugs” both show the testing is rock solid

Random Testing (RT)

- Disadvantages:
  - It does not benefit from source code information.
  - It is difficult to find “deep” errors.
- When To Use RT
  - When we have very complex problem: system testing rather than Unit Testing
  - In the first we can use RT and monitor coverage. Then, if we need to, we apply other (more advanced) techniques, (for example) if failure rate becomes too low to use RT

Random Testing (RT) Pitfalls

Random testing (RT) suffers from deficiencies that can negatively impact its effectiveness

1. Useful test

```
Set t = new HashSet();
s.add(“hi”);
assertTrue(s.equals(s));
```

2. Redundant test

```
Set t = new HashSet();
s.add(“hi”);
s.isEmpty();
assertTrue(s.equals(s));
```

3. Useful test

```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

4. Illegal test

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1); //pre: argument >=0
assertTrue(d.equals(d));
```

5. Illegal test

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Random Testing (RT) Pitfalls

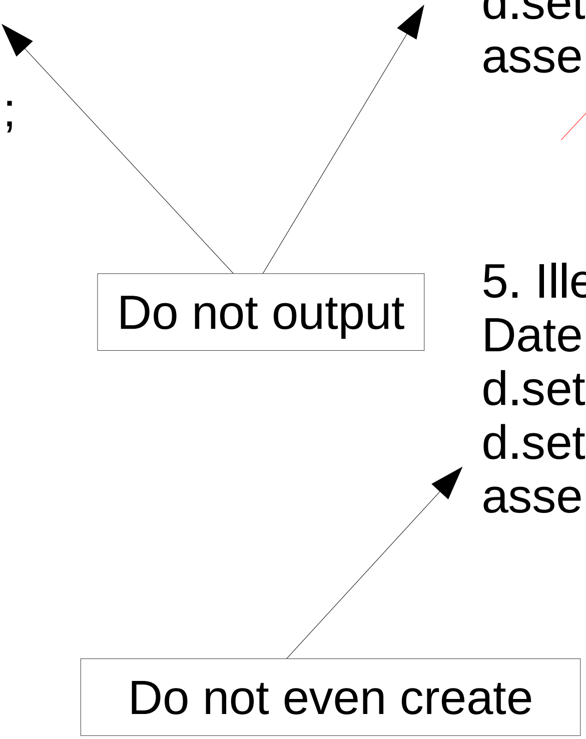
1. Useful test
Set t = new HashSet();
s.add(“hi”);
assertTrue(s.equals(s));

2. Redundant test
~~Set t = new HashSet();~~
~~s.add(“hi”);~~
~~s.isEmpty();~~
~~assertTrue(s.equals(s));~~

3. Useful test
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));

4. Illegal test
~~Date d = new Date(2006, 2, 14);~~
~~d.setMonth(-1);~~
~~assertTrue(d.equals(d));~~

5. Illegal test
~~Date d = new Date(2006, 2, 14);~~
~~d.setMonth(-1);~~
~~d.setDay(5);~~
~~assertTrue(d.equals(d));~~


Do not output

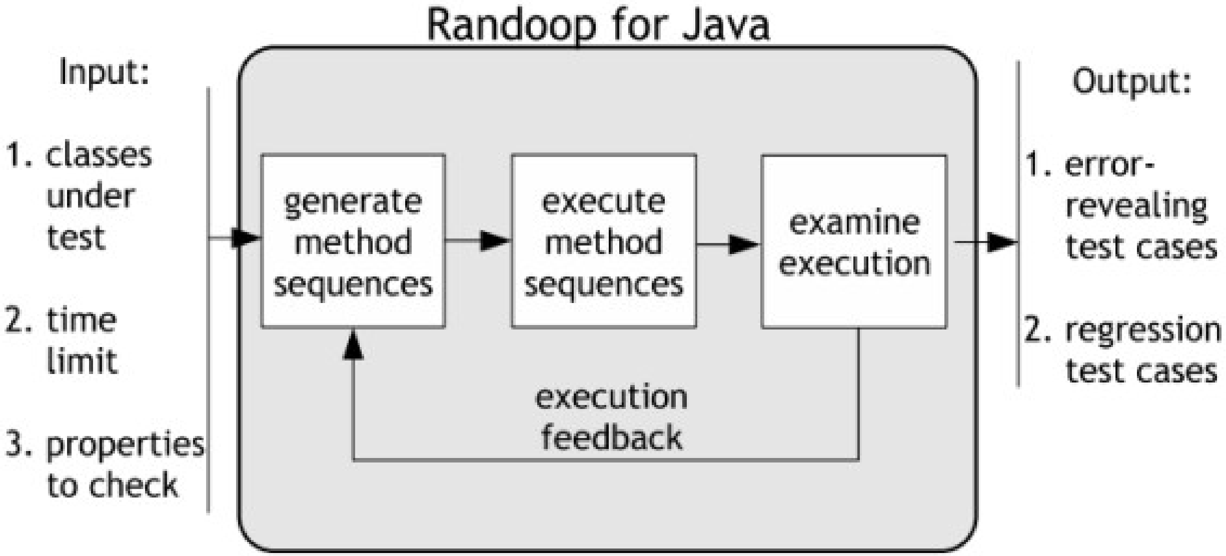
Do not even create

Feedback-Directed Random Unit Testing

- The basic idea is to build inputs incrementally
- New test inputs extend previous ones (a test input is a method call)
- Execute it and use the execution results
  - to eliminate duplicate (redundant) or uninteresting cases (illegal method sequences)
  - to build new sequences that create new object states

Randoop tool

- Randoop is a fully automated testing tool that implements the **feedback-directed random test generation** for .NET and Java.
  - **PDF**:“Randoop: feedback-directed random testing for Java”
  - **Implementation**: Randoop test generation tool



- RANDOOP found many previously-unknown errors not found by either model checking or undirected random generation.

How it works

- **Creating a sequence:**
  - Each sequence “constructs” several objects, available after the last method call is executed:
    - < result, receiver, param1, ..., paramn > of last method call
  - Example: sequence constructs two objects:

    < LinkedList, HashSet >

    ```
    LinkedList l = new LinkedList();
    Set h = new HashSet();
    l.addFirst(h);
    ```

- **Classifying a sequence:**
  - Sequences that lead to contract violations are output to the user as contract violating tests.
  - Sequences that exhibit normal behavior (no exceptions and no contract violations) are output as regression tests.
  - Sequences that exhibit illegal behavior (for example, a sequence that throws an IllegalArgumentException) are discarded.
  - Only normally-behaving sequences are used to generate new sequences

Swarms Approach

- The **SWARM** testing approach is an inexpensive approach and proposed to improve the diversity of test cases generated during random testing.
  - PDF: “Swarm Testing“
- In the random testing approach, all of the features of the Class Under Test (CUT), i.e., public methods and constructions, are available during the construction of each test case.
- The **SWARM** approach, in contrast, is based on a construct configuration that randomly chooses which features to include in each test case.
- The idea behind **SWARM** is that omitting some features increases the effectiveness of testing due to interactions between features.
  - For example, if we are testing a **Stack ADT** that provides two operations, ***push*** and ***pop***.
  - A test generator based on swarm testing first chooses a non empty subset of the **Stack** API. Then, generates a test case using that subset. Thus, one-third of the test cases contain both ***pushes*** and ***pops***, one-third just ***pushes***, and one-third just ***pops***.
- Experimental results show that **SWARM** testing increases coverage and can improve the fault detection dramatically

Some automatic test input generation tools for Java

- **JCrasher** attempts to detect bugs by causing the program under test to "crash" --to throw an undeclared runtime exception. Click here for instructions on installation and usage.
- **Palus** uses both dynamic and static analysis, and is building on top of Randoop. Click here for instructions on installation and usage.
- **Korat** is kind of functional (black-box) testing. Click here for instructions on installation and usage.

References:
http://homes.cs.washington.edu/~mernst/pubs/feedback-testgen-icse2007-slides.pdf

https://www.st.cs.uni-saarland.de/edu/testingdebugging10/slides/18_OOTesting.pdf

http://www.cs.cmu.edu/~./agroce/issta12.pdf#http://www.cs.cmu.edu/~./agroce/issta12.pdf