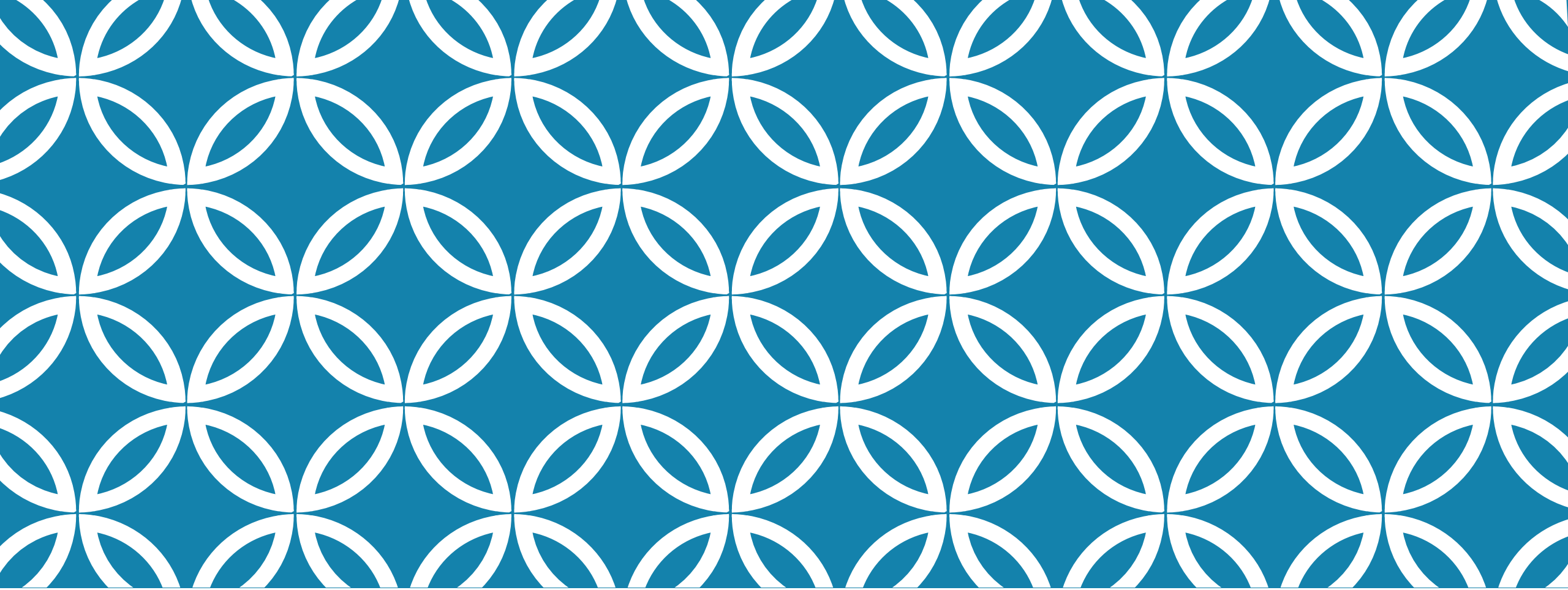


AUTOMATED TEST INPUT GENERATION

CS-HU 374 Lecture 7

AUTOMATED TEST CASE GENERATION

Manually designing test cases

- Difficult
- Tedious
- Time consuming

Automated test case generation (ATG)

- Focuses on generating inputs
- Can add small and effective assertions that captures limited program behaviors
- Expected result either coded as assertions in the code
- Or are implicit -- no run-time errors should occur

BASIC ATG APPROACH

ATG tools are called fuzzers – they vary in several techniques

How does it produce an input?

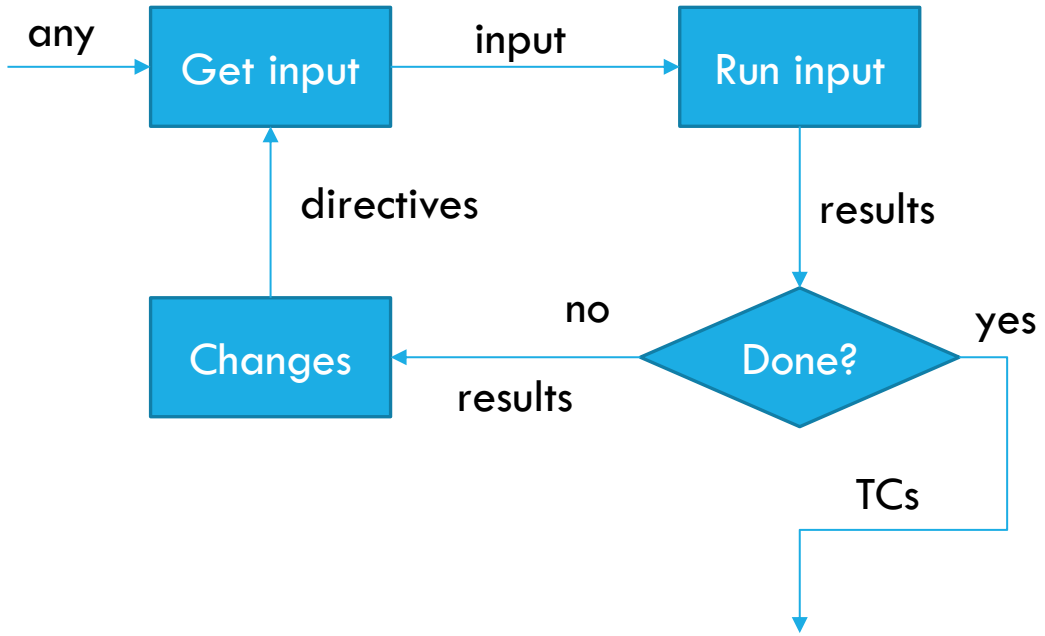
- Generation based
- Mutation based

What is the type feedback from the previous input runs?

- Blackbox based
- Whitebox based
- Grey-box based

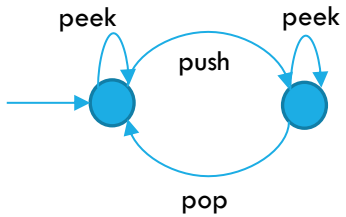
How does it change the input generation strategy?

- Naive way
- Smart way



GENERATION VS MUTATION BASED

- Generation based
  - Takes a description of valid inputs and generates new inputs based on it
  - Use Java's Context-free grammar to derive valid programs to test compiler implementation
  - Use some formula to generate dependable inputs: generate randomly a and b and then compute $c = \sqrt{a^2 + b^2}$; generating valid/invalid triangles generate randomly positive a and b and then compute c such that $a + b \leq c$
  - Traverse a graph model, e.g., stack operations



- Mutation based
  - Starts with existing set of valid inputs, i.e., seeds and then change them in a systematic way
  - Switch the input values, e.g., (1,-2) becomes (-2,1), or change sings e.g., (1,-2) becomes (-1,2)
  - Extends sequence of method calls on an object (s.push(a), s.pop()) becomes (s.push(a), s.pop(), s.peek())

BLACKBOX VS WHITEBOX

- Leverage feedback of already executed tests to increase test case diversity
- Blackbox
  - Observes results of the previous test inputs executions for further extension → This lecture
  - How many triangle classifiers of each type has been returned? If there is no EQUILATERAL then execute directives on how to increase chances on generating those.
- Whitebox
  - Keeps track of structural elements covered and not covered by previous test cases → Next lecture
  - How many statements have been covered? Which ones are missing? How to generate a test input to execute them?

NAÏVE VS SMART DIRECTIVES

Directives on how to create a next test input:

- Naïve
  - Are not aware – randomly changes inputs.
  - Might find rare inputs but generates many invalid inputs that do not reach program's logic.
- Smart
  - Aware of input structure – grammar, model or protocol based, or other systematic processes
  - Instead of negating values now switch them around
  - Instead of traversing each edge once, now traverse each edge twice
  - For context-free grammars: use the production rule $A \rightarrow aBb$ first in the string derivation
- The level of directives is on a continuum between Naïve and Smart approaches.

BLACKBOX FUZZING

Randomly generates inputs

- Simple and effective approach
- Term fuzzing originated in1988 by Barton Miller at the University of Wisconsin
  - Randomly generated files and command line parameters for Unix utilities
- Pros
  - Generates thousands of inputs fast and runs them in parallel
  - Scaled to any program size and applied to different programming languages
- Cons
  - Purely random generation can produce redundant inputs, i.e., in the same equivalence class
- Develop a fuzzer that learns to diversify inputs by observing an input-output relation
  - Negative inputs always return value -1, or throw an exception.
  - Uses Machine Learning to detect patterns and Artificial Intelligence to search for new inputs
- Many tools – easy to put one together
  - https://wiki.mozilla.org/Security/Fuzzing/Peach - Peach fuzzer, a commercial fuzzer
  - https://github.com/guilhermeferreira/spikepp -- Spike is a fuzzer creation kit that provides APIs for creating your own fuzzer

IN-CLASS WORK: WRITE A SIMPLE FUZZER

Recall TriangleClassifier.java

```
static TtriangleType classify(int a, int b, int c)
```

Finish TriangleClassifierFuzzer.java in w4_code package

- Takes an integer parameter N and generates a JUnit file with N test cases as
- Randomly generates three integers N times: valA, valB, valC
- Creates a test case that invokes Triangle.classify(valA, valB, valC)

1. TODO: write to a JUnit file in w4_test package (reuse your code from P1 – week1 to write to a file) TriangleClassifierTest_N.java
2. Generate 4 files: N = 1, N = 10, N = 100, N = 1000

IN-CLASS WORK: EVALUATE A SIMPLE FUZZER

Recall TriangleClassifier.java

static TtriangleType classify(int a, int b, int c)

For each JUnit file (N=1, 10, 100, 1000) report line and branch coverage numbers on correct program

Would any of them find a bug in the faulty implementation? Run JUnits on the TriangleClassifierFaulty (in the "code" package)

Test size	classify(int a, int b, int c)		bug?
N	line%	branch%	yes/no
1	80%	50%	no
10			
100			
1000			
100_improved			

Make your fuzzer "smarter" - be as creative as you wish

- Run it with N =100 and report the same data: yes/no, line%, branch%

PARTICIPATION 4

P4 – is due by the end of Tuesday

1. Show and run 4 original JUnit files N = 1, 10, 100 and 1000
2. The table filled (Google docs or sheets) with the bug detection and the coverage data
3. At least one improvement on your fuzzer and its JUnit with N=100 with the same data, i.e., bug found, statement and branch coverage, added to the table. (Show and run it)
4. Answer the following questions in your document:
   1. Do the coverage metrics change with N?
   2. How effective is your fuzzer comparing to your equivalence partition-based test suite T_B for classify (week 2)

ADVANCED BLACKBOX FUZZER

Generating random integer inputs is easy

Generating random structured inputs is more complicated

- PDF files
- Email addresses, SQL queries
- Heap structures: linkedlist, stacks, Person object.

Advanced fuzzers can do it

- Randoop (https://randoop.github.io/randoop/) – feedback directed test case generator
- Able to handle complex data types

NEXT CLASS: EVOSUITE

EvoSuite – a Whitebox fuzzer

Instruments program to track covered elements

Uses sophisticated genetic search algorithms (from Artificial Intelligence) to generated new inputs

Coverage information is used to evaluate the "fitness" of an input

http://www.evosuite.org