

*CS5351 Software Engineering*  
*2024/2025 Semester B*

# Software Testing

---

Prof. Zhi-An Huang

Email: `huang.za@cityu-dg.edu.hk`

# Outline

---

- ◆ What is a Testing Process?
  - General Process Step
  - Continuous Integration
- ◆ Fuzzing
- ◆ Scientific Debugging
  - Delta Debugging

# What is Program Testing?

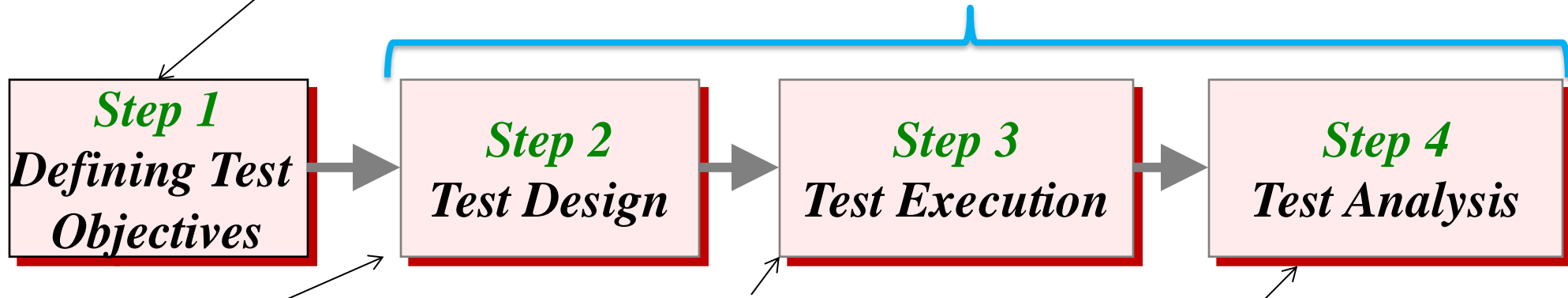
---

- ◆ Conduct **validation** on a program to find bugs or demonstrate the program working under certain scenarios (acceptance test).

# Procedure for Program Testing

The specific “Goal”

Test Automation Possible



What: Define the test requirements.  
e.g., test every exception handler in the user requirements

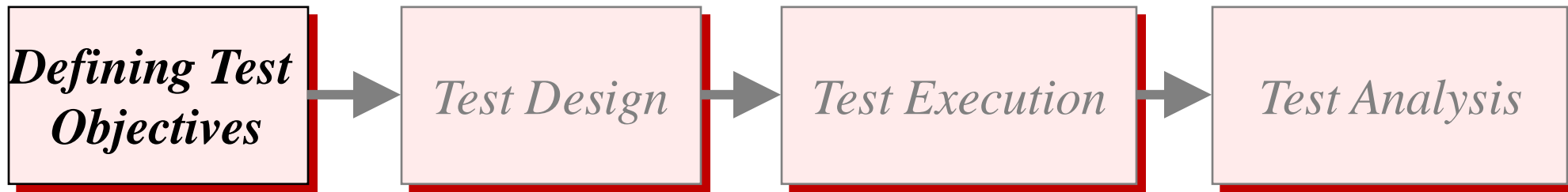
How.  
We use JUnit,  
UI capture and  
replay tool

Measure *how well*  
the “How” achieves  
the “What”

Developers should not just play around a while and judge whether the system passes the test.

# Procedure for Program Testing (1)

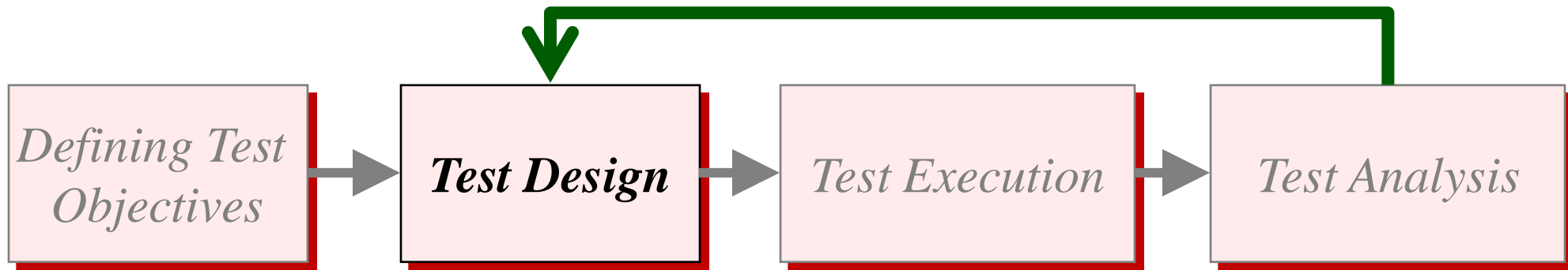
---



- ◆ Define the overall **goal** for designing and executing a test and the type of tests to be conducted.
- ◆ E.g.:
  - To get, clarify, and concretize user requirements by examples
  - To demo the production code meeting user requirements
  - To guard against previously found bugs (regression testing).
  - User acceptance test, system acceptance test, security test
  - To test the integration of two subsystems

# Procedure for Program Testing (2)

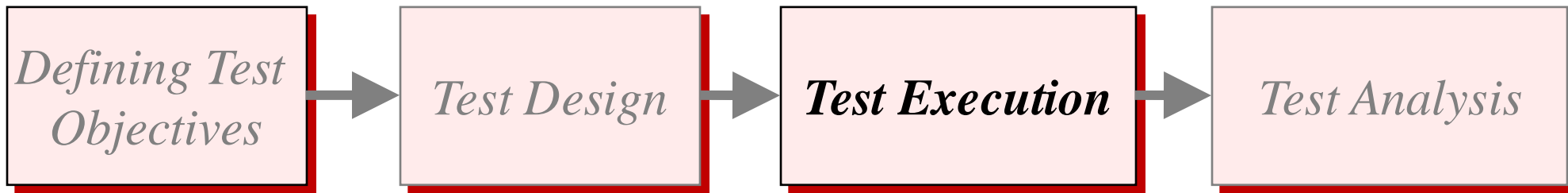
---



- ◆ **Goal:** To get, clarify, and concretize user requirements by examples
- ◆ **Implement** test code (or define test profiles for fuzzing)
- ◆ e.g., Write a test class (e.g., in *JUnit*) to exemplify I/O for user requirements.

# Procedure for Program Testing (3)

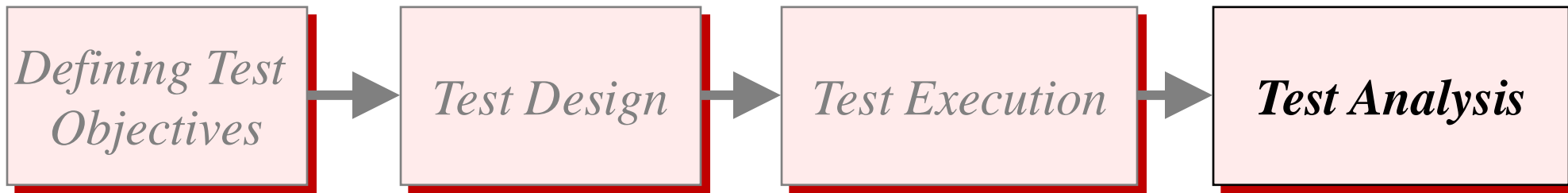
---



- ◆ Execute the test cases written in the Test Design step.
- ◆ E.g.,
  - Run the JUnit test scripts
  - Run in Continuous Integration

# Procedure for Software Testing (4)

---



- ◆ **Identify failures and anomalies** from the test results
- ◆ If we want to automate Step 2 or Step 3, we shall automate this step as well.
- ◆ **Examples:**
  - Using Assertion statements in the JUnit test script.  
`assertEqual (Account.credit (Money) - Account.balance (), 0)`
  - Crash an Android app by running Android Monkey.
  - We may manually observe and judge the test results, but non-productively.



# Automation or Not?

- ◆ Traditional scenario:

- while (i) do

- {develop in Month-i; deploy in Month-(i+1);}

- ◆ Nowadays scenarios (continuous delivery):

- while (i) do

- {develop in day-i;

- locally test in Day-i; commit in Day-i;

- build in Day-i; deploy in Day-i; i++;}

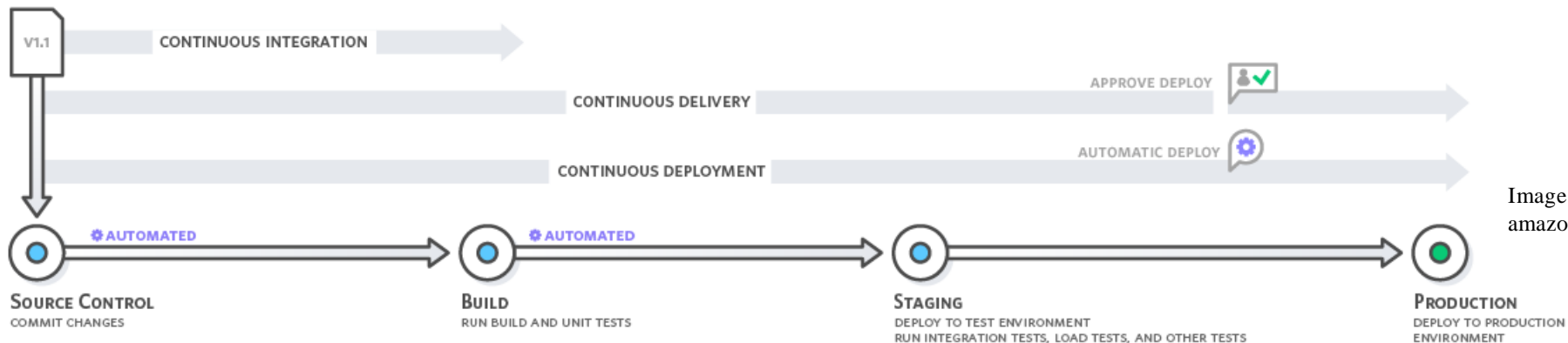


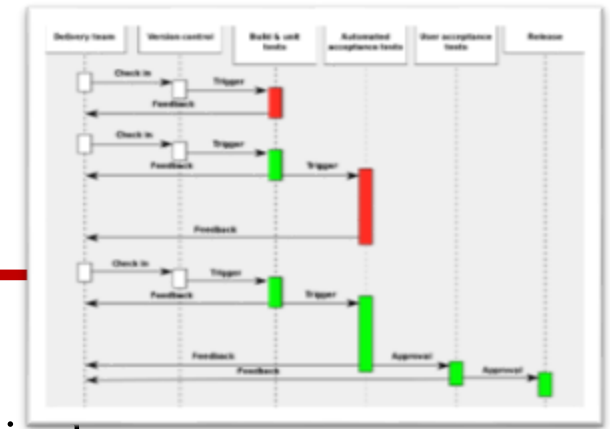
Image source:  
amazon.com

# Continuous Integration (CI)

---

- ◆ Use a single shared repository (CI server) to host and integrate the main branch of code (baseline) frequently
- ◆ Developers' practices
  - Before pulling into the CI server, conduct the unit test locally
  - Make commits every day (to synchronize the common baseline)
  - Enhance the test scripts to validate every bug-fixing commit
- ◆ Every commit in the CI server triggers an automated build in an integration environment
  - We need a build script and unit test scripts
  - The build should include self-testing (run tests by itself)
- ◆ Test in a clone of the production environment (known as staging) after validating the committed code at the unit test level
- ◆ Continuous Delivery (CD) extends CI with auto-deployment

# In a Larger Context

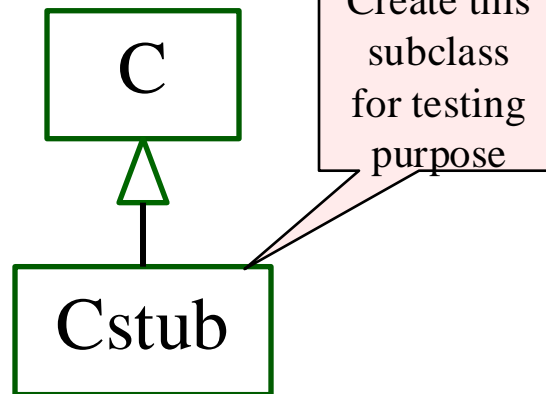


- ◆ Continuous delivery/deployment (CD) [18]
  - Is a process to deploy software into production iteratively.
- ◆ Key characteristics:
  1. Software is updated in *small increments* that are *independently deployable*
  2. Fully auto-deploy them immediately after development and testing are completed
  3. The decision to deploy is largely left up to the developers
  4. Developers responsible for development, deployment, and testing can be notified immediately when system failure in production occurs.
- ◆ Key practice (read Section 2 of Ref [18])
  - Implement automated unit- and sub-system tests
  - Must use automated testing tools to enable **early and frequent** testing
  - Incorporate code review
  - Configure auto-deployment tool
  - Deploy the updates by stage.

# Making Software Testable

- ◆ Sometimes, our code module needs to call other code modules, but we want to test our code module without any interaction with others.
  - Example: In the source code, we may have the following code structure. Suppose that we are testing the integration of A and B, but *without* the code of C is not ready.

```
...  
int a(ClassB b) {  
    ...  
    b.b2( );  
}  
...  
Class ClassB {  
    private ClassC c;  
    public ClassB(ClassC c1)  
    { c = c1; }  
    public void b2( ) {  
        c.c1();    c.c2();  
        c.c3();    c.c4();  
    };  
}
```

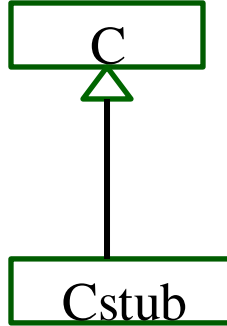


In a test script, we set up an object b1 to use an object of class Cstub instead of the original class C.

```
stubForC = new Cstub();  
b1 = new B(stubForC);  
...  
a(b1);
```

To use this style of testing, your class B needs a careful design.

# Making Software Testable



```
Class A { ...
void a(ClassB b) {
    ...
    b.b2( );
}}
...
```

Test script can set the state of b. Hence, C can be remained untested before testing A and B. So, top-down testing is feasible.

```
Class ClassB {
    private ClassC c;
    public void b2( ){
        c.c1();    c.c2();
        c.c3();    c.c4();
    }
    public void ClassB(ClassC seed){
        c = seed;
    }
}
```

**Can be tested in a standalone manner**

```
Class A { ...
void a() {
    ClassC seed = new ClassC();
    ClassB b = new ClassB(seed);
    b.b2( );
}}
...
Class ClassB {
    private ClassC c;
    public void b2( ){
        c.c1();    c.c2();
        c.c3();    c.c4();
    }
    public ClassB(ClassC seed){
        c = seed;
    }
}
```

**less testable**

Class A controls the state of B. Class C must be tested before testing A and B. So, only bottom-up testing is feasible.

For some classes, we do top-down testing; for the other classes, we do bottom-up testing. We need a *sandwich* approach.

# Some Characteristics of Effective Testing in Uncertain User Needs [3]

---

- ◆ In Agile development, **requirements are often not defined in detail, and without strict processes to elicit them**, developers need an understanding of the business and experience in the domain model.
  - **Requirement testing** (via system testing): demonstrate the (vague and incomplete) requirement rather than finding bugs in the design and implementation.
  - **Test coverage**: knowing all possible usage scenarios is infeasible. Paying *more attention to test features with larger user bases in the next release* of the program. Judge by developers on whether the feature has been *sufficiently tested for the users to make effective use of the feature*.
  - **Test automation**: **DO NOT** replace manual testing. However, it is a good way to show users that the produced code is testable to *educate users* to raise checkable requirements and support feature implementation approval in a testable way.

# Testing in Uncertain User Needs

---

# Pair Programming

---

- ◆ Pair programming is a practice used in agile development
- ◆ Developers X and Y write the same piece of code simultaneously.
  - $\neq$  X writes the code, and Y reviews the code after code completion
  - $\neq$  X independently completes part A, and Y independently completes part B, and then the two parts are integrated
  - $=$  Y reviews the code being written **while** X writes each line; X and Y may switch their roles during the process



# *Testing in Uncertain User Needs:*

## **In-Class Exercise on “Lab1”**

---

- ◆ Study the Java code project on the next slide. The code project contains at least one bug, and **the exact requirement is unclear to you.**
  - See the cheat sheet for Hello World in Java in the appendix if you do not know Java.
- ◆ Form a **team of two students** (i.e., for pair programming)
  1. Use your judgment to “guess” what the code project aims to do. Add more JUnit test cases to represent the requirements you guessed. Get a consensus between the two while writing the same line of code
  2. **Brainstorm low-probability** usage scenarios that the code project can use. Create JUnit test cases to represent these usage scenarios.
    - Still recall the brainstorming session method in requirement elicitation?

# *Testing in Uncertain User Needs:*

## In-Class Exercise on “Lab1”

---

```
package Lab1;

public class UR {

    int sizeOfCake;

    UR(int size) {
        this.sizeOfCake = size;
    }

    boolean dividingCake(int pieces) {
        int size = this.sizeOfCake / pieces;
        System.out.println("the size of each piece of cake: " + size);
        return true;
    }
}
```

```
package Lab1.testcase;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

public class TestUR {

    @Test
    public void test() {
        UR r = new UR(6);
        r.dividingCake(6);
        r.dividingCake(3);
    }
}
```

Fix the bugs if your test case exposed it.

# Testing in Uncertain User Needs: In-Class Exercise on “Lab1” (Solution)

**Actual User Requirement:** The goal of the method `dividingCake()` is to divide a cake into several pieces of equal size. The method is designed to return a Boolean value. Specifically, if the size of a piece of cake is less than 2, the method should return `false`, otherwise `true`. A message about the size of each piece should be displayed.

```
package Lab1;

public class UR {

    int sizeOfCake;

    UR(int size) {
        this.sizeOfCake = size;
    }

    boolean dividingCake(int pieces) { // return a Boolean value
        float size = this.sizeOfCake / (1.0*pieces);
        System.out.println("the size of each piece of cake: " + size);
        if (size < 2) return false;    // omission fault
        return true;
    }
}
```

Note that it is almost impossible to guess out the return value logic by developers themselves. We call such a bug an **omission fault** [4], which accounts for about 50% of bugs in programs.

```
package Lab1.testcase;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

public class TestUR {

    @Test
    public void test() {
        boolean s;
        UR r1 = new UR(6);
        r1.dividingCake(6);
        r1.dividingCake(3);
        r1.dividingCake(4); // bug found
        r1.dividingCake(0); // exception happened
        r1.dividingCake(-1); // negative size?

        UR r2 = new UR(-6); // still logical?
        r2.dividingCake(-2); // what?

        UR r3 = new UR(6);
        assertFalse(r3.dividingCake(6));
        assertTrue(r3.dividingCake(2));
    }
}
```

# How to Spot Omission Faults?

---

Typical strategies are:

- ◆ Conduct Code Review
- ◆ Follow the guidelines and checklist of the company
- ◆ Always ask users or the client developers of the functions whether something should be returned apart from the text outputs.
- ◆ Do not aim to validate the written code only. Also, check against user intention or requirements.
- ◆ Reference the code serving similar purposes if available
- ◆ Use automated testing tools and static analysis tools to guard against generic errors (e.g., security vulnerability)
- ◆ Ask Large Language Model (see the next slide)

# Large Language Model on Lab1

However, there are a few issues in this code:

1. Integer Division: Since both `sizeOfCake` and `pieces` are integers, performing integer division (`/`) may lead to unintended results. If you want to obtain a more precise result, you can convert one of the variables to a floating-point type, like `double` or `float`, before performing the division. For example:

java



```
double size = (double) this.sizeOfCake / pieces;
```

2. Unconditional Return Value: The method `dividingCake` always returns `true`, regardless of the size of the cake or the number of pieces. If you intended to return `true` only when the cake can be divided into pieces, you should modify the logic accordingly. For instance:

java



```
boolean dividingCake(int pieces) {  
    int size = this.sizeOfCake / pieces;  
    System.out.println("the size of each piece of cake: " + size);  
    return size >= 2; // Return true only if the size is greater than or equal to 2  
}
```

# Fuzzing

---

An automated method to generate test input, execution tests and expose security vulnerability (crashes)

# Examples

## Google Security Blog

The latest news and insights from Google on security and safety on the Internet

### Fuzzing Java in OSS-Fuzz

March 10, 2021

Posted by Jonathan Metzman, Google Open Source Security Team

[OSS-Fuzz](#), Google's open source fuzzing service, now supports [fuzzing](#) applications written in Java and other [Java Virtual Machine \(JVM\)](#) based languages (e.g. Kotlin, Scala, etc.). Open source projects written in JVM based languages can add their project to OSS-Fuzz by following [our documentation](#).

Find bugs + code coverage



## Fuzzing Java Applications

Automated security testing for Java applications helps find bugs in code and APIs.

APPLICATION SECURITY

### Google Brings AI Magic to Fuzz Testing With New Results

Google sprinkles magic of generative-AI into its open source fuzz testing infrastructure and finds immediate results.



By Ryan Naraine  
August 11, 2023



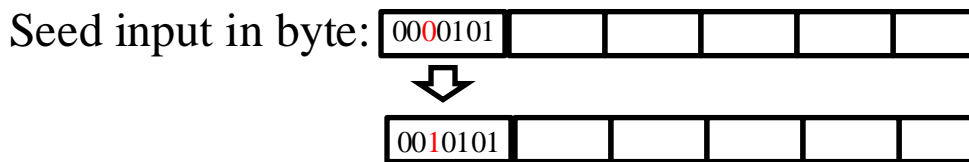
Google has sprinkled the magic of artificial intelligence into its open source fuzz testing infrastructure and the results suggest LLM (large language model) algorithms will radically alter the bug-hunting space.

Google added generative-AI technology to its OSS-FUZZ project (a free service that runs fuzzers for open source projects and privately alerts developers to the bugs detected) and discovered a massive improvement in code coverage when LLMs are used to create new fuzz targets.

# What is Fuzzing?

- ◆ If you developed Android apps before, you may use *monkey* [5], which is a fuzzing tool.
- ◆ Fuzzing is the process of finding vulnerabilities code by repeatedly testing the code with modified inputs [6].
  - is becoming standard in the commercial software development process [6].

Seed input in text: High, 30



The *mutate* function mutates a byte of the input in the *byte* position

**Algorithm:** Simple Random Fuzzing

**Input:** *Seeds*, Target program *B*

**Result:** *MaliciousInputs*

```
for Seed ∈ Seeds do
  for iterations ← 0 to limit do
    input ← Seed
    length ← len(Seed)
    mutations ← RandInt(length)
    for mut ← 0 to mutations do
      byte ← RandInt(length)
      mutate(input, byte)
    end
    result ← Execute(P, input)
    if result is crash then
      Append input to MaliciousInputs
    end
  end
end
```



## e.g., Android Monkey



- ◆ Prerequisites : install Java JDK and Android SDK
- ◆ Select the required Android version in Android SDK Manager
- ◆ Use the AVD Manager to create an Android Virtual Device
  - Choose a device template from Device Definitions, then press [Create AVD]
  - Select the CPU type (e.g., ARM) and Skin (to emulate some of the most common screen configurations), and then launch the AVD
- ◆ Use Windows cmd window
  - Go to directory `c:\users\<user>\appdata\local\Android\android-sdk\platform-tools`
  - Type 'adb devices' to find the AVD name (e.g., emulator-1234)
- ◆ Start Monkey
  - `adb shell monkey [options] <event-count>`
  - `adb shell monkey -p your.package.name -v 500`

# Example (from the Web)

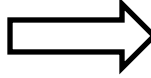
◆ `adb -s emulator-5554 shell monkey -v 500`

Event probability

```
C:\Users\Sun\AppData\Local\Android\android-sdk\platform-tools>adb -s emulator-5554 shell monkey -v 500
Monkey: seed=1439713591933 count=500
IncludeCategory: android.intent.category.LAUNCHER
IncludeCategory: android.intent.category.MONKEY
// Event percentages:
// 0: 15.0%
// 1: 10.0%
// 2: 2.0%
// 3: 15.0%
// 4: -0.0%
// 5: 25.0%
// 6: 15.0%
// 7: 2.0%
// 8: 2.0%
// 9: 1.0%
// 10: 13.0%
Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10200000;component=com.android.email/.activity.Welcome;end
// Allowing start of Intent < act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.android.email/.activity.Welcome > in package com.android.email
:Sending Trackball <ACTION_MOVE>: 0:<1.0,4.0>
:Sending Touch <ACTION_DOWN>: 0:<13.0,15.0>
:Sending Touch <ACTION_UP>: 0:<33.609848,0.0>
:Sending Trackball <ACTION_MOVE>: 0:<-2.0,3.0>
:Sending Touch <ACTION_DOWN>: 0:<40.0,114.0>
:Sending Touch <ACTION_UP>: 0:<0.0,231.99048>
:Sending Touch <ACTION_DOWN>: 0:<18.0,45.0>
:Sending Touch <ACTION_UP>: 0:<29.697094,41.766052>
:Sending Touch <ACTION_DOWN>: 0:<76.0,296.0>
:Sending Touch <ACTION_UP>: 0:<88.968254,302.35077>
// Allowing start of Intent < act=android.intent.action.EDIT dat=content://i.email.android.com/settings cmp=com.android.email/.activity.setup.AccountSettings > in package com.android.email
```

```
:Sending Touch <ACTION_UP>: 0:<73.75486,117.503>
:Sending Trackball <ACTION_MOVE>: 0:<-5.0,-1.0>
:Sending Touch <ACTION_DOWN>: 0:<154.0,295.0>
:Sending Touch <ACTION_UP>: 0:<145.98016,228.80>
:Sending Trackball <ACTION_MOVE>: 0:<-5.0,3.0>
Events injected: 500
:Sending rotation degree=0, persist=false
:Dropped: keys=0 pointers=0 trackballs=0 flips=0
## Network stats: elapsed time=20664ms (0ms mobile)
// Monkey finished
```

# Monkey

Monkey's command line argument.  Developers can specify the distribution of kind of events to define a **test profile**.

```
Events
-s <seed>

--throttle
<milliseconds>

--pct-touch
<percent>

--pct-motion
<percent>

--pct-trackball
<percent>

--pct-nav
<percent>

--pct-majornav
<percent>
```

- ◆ The Monkey tool emulates user interaction with an app. It generates and injects gestures commands (see the table on the right) or system events based on a predefined event frequency distribution into the app's event input stream.
- ◆ The tool watches for exceptional conditions, e.g., the app crashing or throwing an exception.
- ◆ The tool can vary the “throttle”, i.e., the time delays between events; by default, there is no delay between events (throttle = 0).

Table 1: Monkey's Default Events and Percentages

Event ID	Description	Frequency (%)
TOUCH	single touch (screen press & release)	15
MOTION	“drag” (press, move, release)	10
TRACKBALL	sequence of small moves, followed by an optional single click	15
NAV	keyboard up/down/left/right	25
MAJORNAV	menu button, keyboard “center” button	15
SYSOPS	“system” keys, e.g., Home, Back, Call, End Call, Volume Up, Volume Down, Mute	2
APPSWITCH	switch to a different app	2
FLIP	flip open keyboard	1
ANYTHING	any event	13
PINCHZOOM	pinch or zoom gestures	2

# Monkey

## [Crash (Yes), Code Coverage (No)]

Find crash via stress test? **Yes**

<i>App</i>	<i>Installs (millions)</i>	<i>#Events</i>	<i>Time to crash (seconds)</i>
Shazam	100–500	1,270	36
Spotify	100–500	9,011	83
Facebook	1,000–5,000	9,047	197
Whatsapp	1,000–5,000	14,768	175
Splitwise	1–5	819	12
UC Browser	100–500	5,575	51
NY Times	10–50	24,358	329
Instagram	1,000–5,000	279	7
Snapchat	500–1,000	4,676	24
Walmart	10–50	6,510	56
MX Player	100–500	1,742	41
Evernote	100–500	8,733	91
Skype	500–1,000	17,435	95
Waze	100–500	7,261	38
Google	1,000–5,000	1,220	34
<i>Mean</i>		<i>7,513</i>	<i>85</i>

(set throttle = 0)

Manual testing has better code coverage than Monkey

<i>Coverage type</i>	<i>Mean</i>	
	<i>Monkey</i>	<i>Manual</i>
Class	52.4	<b>54.7</b>
Method	42.2	<b>45.0</b>
Block	38.6	<b>42.5</b>
Line	38.1	<b>41.6</b>

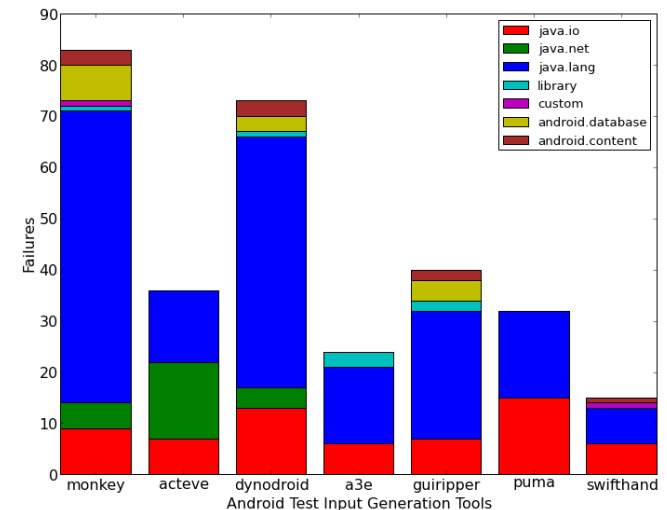
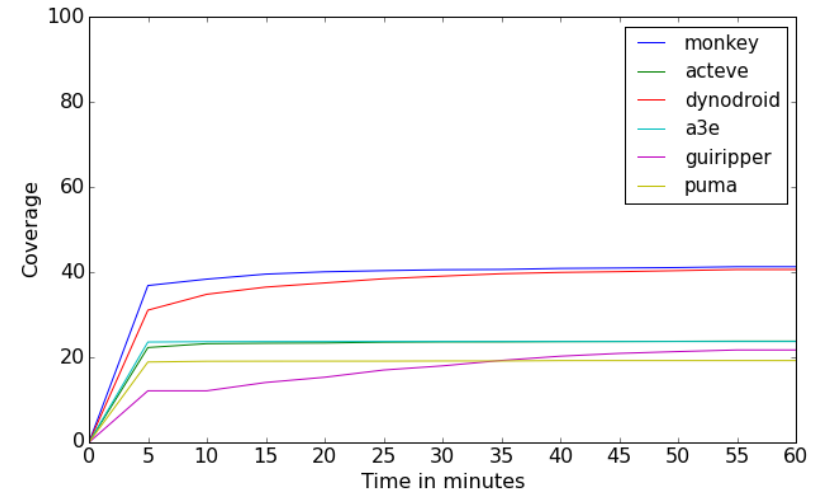
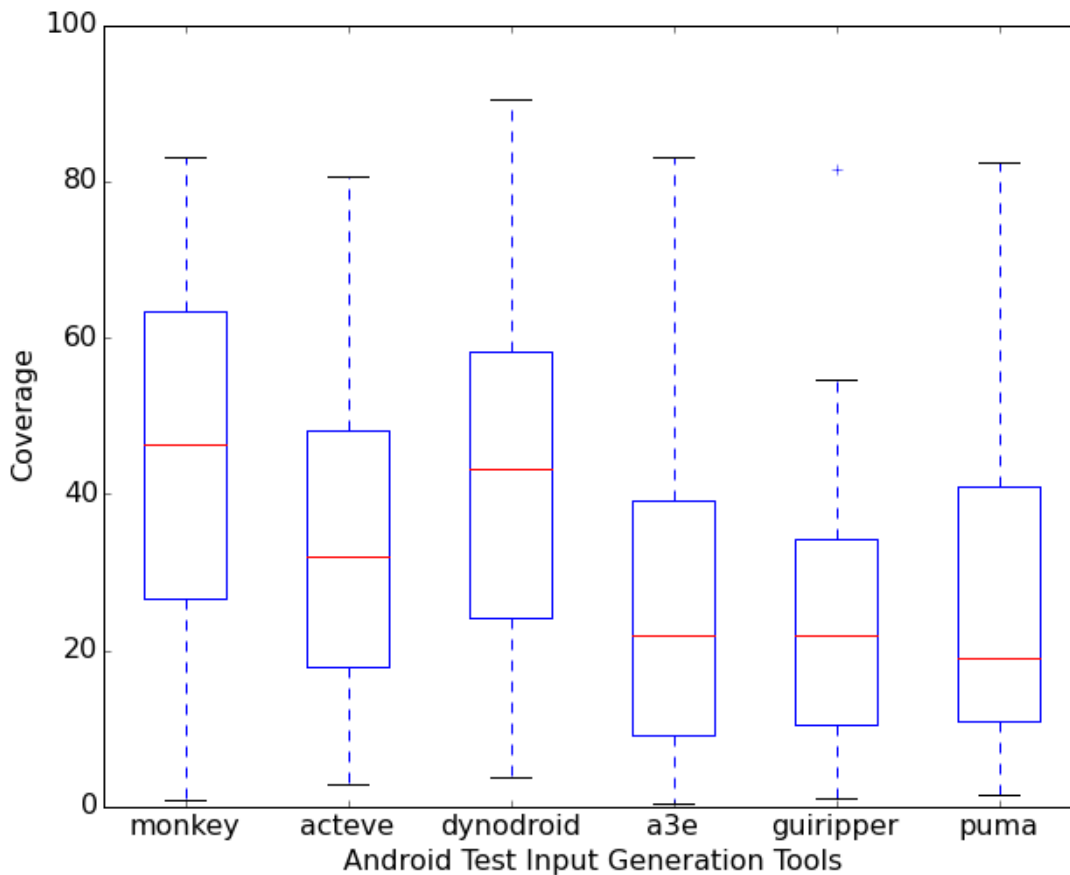
Setting different time delays is insignificant to improve code coverage

: Line Coverage when Varying Throttle

<i>Throttle (msec)</i>	<i>Mean Coverage (%)</i>
0	40.8
100	<b>45.0</b>
200	42.5
600	42.6

# Other Android Fuzzing Tools [11]

## Lower code coverage and fewer crashes than monkey

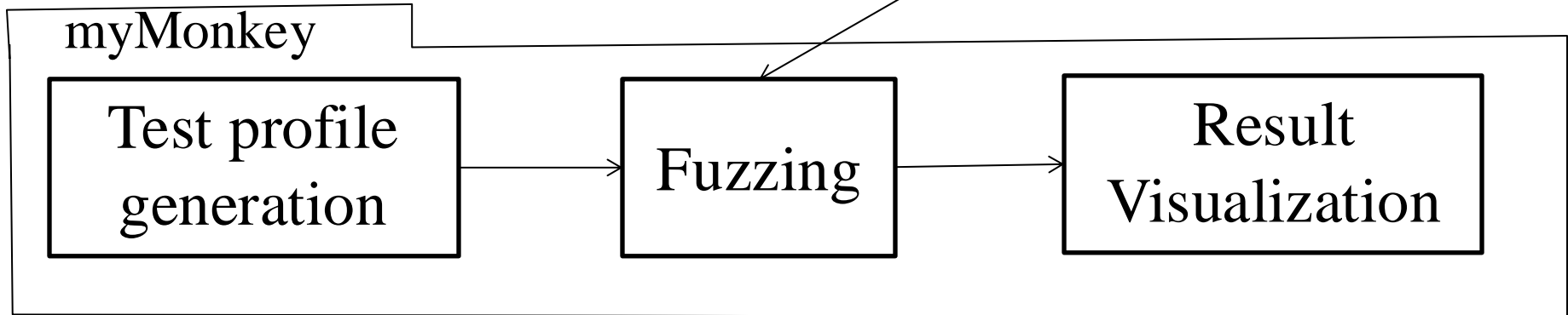


Instead of command-line monkey, one can write code to interact with an API version of monkey, and thus produce one's own (simple) fuzz testing tool.

<https://developer.android.com/studio/test/monkeyrunner/>

## Variants of Monkey

---



- ◆ You can build your own fuzzer
- ◆ The best-known fuzzing tool is **American Fuzzy Lop** [8], a grey-box mutation-based feedback-based fuzzer for gcc/clang c++ code projects. It has been ported to support testing the code written in other programming languages and beyond Linux.
  - E.g., for Windows binaries: <https://github.com/ivanfratric/winafl>
  - E.g., for Android app: <https://github.com/ele7enxxh/android-afl>

# American Fuzzy Lop (AFL)

---

- ◆ **Feedback-based:** run, get info from previous runs in guiding the selection of the next input
- ◆ **Mutation-based:** change inputs to become new inputs
- ◆ **Grey-box:** need both input seeds and code coverage

The basic strategy of AFL:

1. Start with a small set of input sample files as a queue of inputs
  - The test cases for user stories (in Scrum) can be these seed files
2. Mutate “randomly” an input file from the queue
3. If the mutated file reaches new or simpler path(s) (in log2 sense), add it to the queue



# American Fuzzy Lop Algorithm [13]

The fuzzing routine takes as input a program and a set of user-provided *seed inputs*. The seed inputs are used to initialize a *queue* (Line 2) of inputs. AFL goes through this queue (Line 4), selects an input to mutate (Line 5), mutates the input (Lines 10, 13), runs the program on and, simultaneously, collects the coverage information for the mutated inputs (Line 24), and finally adds these mutated inputs to the queue if they achieve new coverage (Line 26). An entire pass through the queue is called a *cycle*. Cycles are repeated (Line 3) until the fuzz testing procedure is stopped by the user.

AFL's mutation strategies assume the input to the program under test is a sequence of bytes, and can be treated as such during mutation. AFL mutates inputs in two main stages: the *deterministic* (Algorithm 1, Lines 8-11) stages and the *havoc* (Lines 12-14) stage. All the deterministic mutation stages operate by traversing the input under mutation and applying a mutation at each position in this input. These mutations (Line 9) include bit flipping, byte flipping, arithmetic increment and decrement of integer values, replacing of bytes with "interesting" integer values (0, MAX\_INT), etc. The number of mutated inputs produced in each of these stages is governed by the length of the input being mutated (Line 8). On the other hand, the havoc stage works by applying a sequence of random mutations—setting random bytes to random values, deleting or cloning subsequences of the input—to the input being mutated to produce a new input. Several mutations are applied to the original input (Line 21) before running it through the program (Line 14). The number of total havoc-mutated inputs to be produced is determined by a performance score, *score* (Line 7).

---

## Algorithm 1 AFL algorithm.

---

```
1: procedure FUZZTEST(Prog, Seeds)
2:   Queue  $\leftarrow$  Seeds
3:   while true do                                      $\triangleright$  begin a queue cycle
4:     for input in Queue do
5:       if  $\neg$ ISWORTHFUZZING(input) then
6:         continue
7:       score  $\leftarrow$  PERFORMANCESCORE(Prog, input)
8:       for  $0 \leq i < |input|$  do
9:         for mutation in deterministicMutationTypes do
10:          newinput  $\leftarrow$  MUTATE(input, mutation, i)
11:          RUNANDSAVE(Prog, newinput, Queue)
12:        for  $0 \leq i < score$  do
13:          newinput  $\leftarrow$  MUTATEHAVOC(input)
14:          RUNANDSAVE(Prog, newinput, Queue)
15: procedure MUTATEHAVOC(Prog, input)
16:   numMutations  $\leftarrow$  RANDOMBETWEEN(1,256)
17:   newinput  $\leftarrow$  input
18:   for  $0 \leq i < numMutations$  do
19:     mutation  $\leftarrow$  RANDOMMUTATIONTYPE
20:     position  $\leftarrow$  RANDOMBETWEEN(0, |newinput|)
21:     newinput  $\leftarrow$  MUTATE(newinput, mutation, position)
22:   return newinput
23: procedure RUNANDSAVE(Prog, input, Queue)
24:   runResults  $\leftarrow$  RUN(Prog, input)
25:   if NEWCOVERAGE(runResults) then
26:     ADDTOQUEUE(input, Queue)
```

---



# in the AFL Algorithm

---

- ◆ Line 4, always select the first element in Queue.
- ◆ isWorthFuzzer(x): the input x discovered new program branch at line 25 when x was added to *Queue* (so, always worth fuzzing)
- ◆ PerformanceScore(P, x): return 1 for AFL
- ◆ Mutation: a binary string mutation operator

## Deterministic mutation strategies:

- Bit flips
  - single, two, or four bits in a row
- Byte flips
  - single, two, or four bytes in a row
- Simple arithmetics
  - single, two, or four bytes
  - additions/subtractions in both endians performed
- Known integers
  - overwrite values with interesting integers (-1, 256, 1024, etc.)

## Random mutation strategies:

- Stacked tweaks
  - performs randomly multiple deterministic mutations
  - clone/remove part of file
- Test case splicing
  - splices two distinct input files at random locations and joins them

- ◆ newCoverage(x's result): e.g., running P(x) and find that x executes a new branch not covered by any test cases before (or executes a branch an order of magnitude (in binary sense) smaller than previous test cases).

# Ways to Modify AFL

---

- ◆ Why not replacing the set of mutation operators in AFL by a new mutation tool?
  - Yes. Try radamsa. <https://gitlab.com/akihe/radamsa>
  - Yes. Write your own set of mutation operators
- ◆ PerformanceScore(x) should be fixed when the same input is selected?
  - **Any good idea** to make the score returned for the same input x be adaptive?
- ◆ Can we change AFL to a pure block-box fuzzer or GUI fuzzer?
  - Yes. How to modify Algorithm of AFL? **Any good idea?**
    - Probably, newCoverage(x's result) is determined from program output, or compare x against those inputs in *Queue*.
    - Can we use UIAutomator API in Android to check Widget coverage and integrate with Monkey to generate better inputs?
- ◆ Any better way to find more unexplored paths? see [13]
- ◆ A faster version of AFL? See [14]. Google ossfuzzer is built on top of afl.

# SideBar

---

1. Read Section 4.4 in Ref [3]
  - What is the consequence of designing an inadequate set of test scenarios?
  - Read some low-rated reviews/posts of a high-rated app in an app store (App Store, Google Store etc). Can you propose ways to find out what are missing?
2. Try American Fuzzy Lop together with Address Sanitizer (ascan) through the following steps in Tutorial 2 and 3 of Ref [8], and then pick a serious program (e.g., see the program list on <http://lcamtuf.coredump.cx/afl/>) and start fuzzing it.
  - The combination of American Fuzzy Lop and Address Sanitizer is a gold standard for program fuzzing.
3. There are many open-source Sanitizers to detect different kinds of bugs.
  - AddressSanitizer (ascan) for memory corruption bugs (e.g., buffer overflow)
  - Thread Sanitizer (tscan) for thread concurrency bugs (e.g., data races)
  - MemorySanitizer (mscan) for of uninitialized reads bugs
  - UndefinedBehaviorSanitizer (ubscan), Kernel Address Sanitizer (kscan)
  - **Some of them are available for testing Android apps:**  
<https://source.android.com/devices/tech/debug/sanitizers>

# OK, Crashes Found. So What?

---

- ◆ Fuzzing finds crashes or ANR (App Not Responding), so-called code vulnerabilities
- ◆ In most cases, developers may start debugging based on the failing test cases (e.g., with stack traces)
  - How do we know we have fixed the problem?
- ◆ If an input is long (e.g., crash a computer game by fuzzing), the debugging will be tedious and slow.
  - More desirable to find a smaller input to trigger the same bug before debugging.
  - How?

# Scientific Debugging

---

# Distinguishing **Failures** from **Bugs**

- ◆ A *failure* is an *incorrect output* of a program.
  - E.g., in JUnit, we may use the API call *AssertEqual()* to determine whether or not the test case fails.
- ◆ A *bug (or fault)* is *an incorrect implementation (coding)* that causes the program to produce failures.

Suppose: Correct:  $\text{int } f_c(x) = \{ \text{return } x*x + 3 * x + 2; \}$

Buggy:  $\text{int } f_b(x) = \{ \text{return } x*x + 2 * x + 3; \}$

$f_b(0) = 3 \neq 2 = f_c(0)$ . So, the test case  $x = 0$  produces an incorrect output 3 from  $f_b$ .  
Comparing the code of  $f_b$  and  $f_c$ , we see that the sentence is written wrongly.  
The sentence  $x*x + 2 * x + 3$  contains a bug (fault).

# How to Debug?

---

We use three steps in a row for this purpose:



# A model of program execution

---

- ◆ A program  $P$  is a set of *statements* and a set of *variables*.
  - Each variable keeps a value.
  - Each statement may read a value from one or more variables and/or write a value to a variable.
  - Outputting a value from a program means to write a value to a (system-specific) variable.
- ◆ A *program state* is a set of key-value tuples, each containing a variable  $v$  as the key and the value of the variable  $v$ .
- ◆ An *execution trace* is a sequence of executed instructions to represent the execution of a test case.
  - Note that the same program statement may occur in the same trace multiple times (and so we have 1<sup>st</sup> occurrence, 2<sup>nd</sup> occurrence, etc.)
  - Thus, by walking through the statements in an execution trace  $\sigma$  from start to end and one by one, the program state of the execution trace  $\sigma$  is updated while we walk through the trace.



# *A model of program execution*

## Failure and Bug

---

### ◆ Failure

- A *failure* of a program  $P$  is an *incorrect program output* of an execution trace  $\sigma$  of the program  $P$ .
- We mark the statements that produce the failure on the trace  $\sigma$

### ◆ Run

- A *failing run* refers to an execution trace that is marked with failure(s).
- A *passed run* refers to an execution trace that is not marked with any failure.

### ◆ Bug

- A *bug* (also known as a *fault*) is a *mistake* appearing in a program  $P$ .
- In an execution trace  $\sigma$ , a bug is marked as an incorrect single statement or an incorrect non-empty subsequence of statement in it. Thus, the positions of a bug can be marked in the execution trace  $\sigma$ . Note that some execution traces do not contain any bug.

## *A model of program execution*

# Propagation from Bug to Failure

---

- ◆ To observe a failure from a run, a bug should *propagate* its *error* along the trace from a buggy statement (say at position  $\sigma[b]$ ) to the statement outputting a failure (say at position  $\sigma[f]$ ).
- ◆ The buggy statement  $\sigma[b]$  transforms a correct program state before executing this buggy statement into an *infected* program state  $E_b$ .
- ◆ If there is any difference between the expected program state and an infected program state, the infected program state incurs an *error*.
- ◆ From the position of this infected program state  $E_b$ , the subsequent instructions (at positions  $\sigma[b+1]$ ,  $\sigma[b+2]$ , ...,  $\sigma[f]$ ) in the trace  $\sigma$  will further transform  $E_b$  into a sequence of infected program states  $E_1, E_2, \dots, E_f$ . When the statement at the position  $\sigma[f]$  is executed, the corresponding error in the infected program state  $E_f$  will be observed as a failure of the execution trace  $\sigma$ .

# Illustration: Relating Execution Information to Program States

---

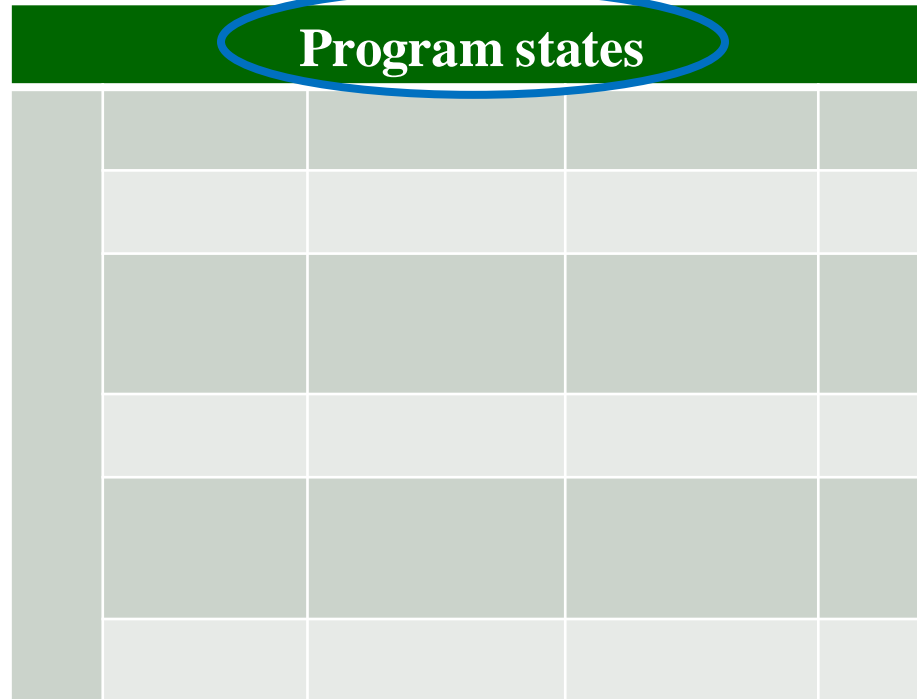
*A pointer assigned to NULL*

*Another assignment*

*A problematic memory access*

```
1. int foo (int a, int b)
2. {
3.     POINT pA = null; // bug
4.     ...
5.     for () {
6.         POINT pB = pA;
7.         ...
8.         if () {
9.             pB[3] = 0;
10.            ...
11.        }
12.    }
13. }
```

# Program State



The diagram shows a table representing program state. The top row is a dark green header with the text "Program states" in white. This header row is circled in blue, and a blue arrow points from the circle to the main title "Program State". Below the header is a grid of 10 rows and 5 columns. The first column is a solid light gray. The remaining four columns are divided into alternating light gray and white cells, creating a checkerboard pattern.

Program states				

```
1. int foo (int a, int b)
2. {
3.     POINT pA = null; // bug
4.     ...
5.     for () {
6.         POINT pB = pA;
7.         ...
8.         if () {
9.             pB[3] = 0;
10.            ...
11.        }
12.    }
13. }
```

# Program State

Program states				
↓	Start	pA : N\A	pB : N\A	...

N\A = undefined

```
1. int foo (int a, int b)
2. {
3.     POINT pA = null; // bug
4.     ...
5.     for () {
6.         POINT pB = pA;
7.         ...
8.         if () {
9.             pB[3] = 0;
10.            ...
11.        }
12.    }
13. }
```

# Program States

*Infected: It means the value of a variable is not correct.*

*infect*

Program states				
↓	Start	pA : N\A	pB : N\A	...
	Line 3	<b>pA : null</b>	pB : N\A	...

```
1. int foo (int a, int b)
2. {
3.     POINT pA = null; // bug
4.     ...
5.     for () {
6.         POINT pB = pA;
7.         ...
8.         if () {
9.             pB[3] = 0;
10.            ...
11.        }
12.    }
13. }
```

# Program States

Program states				
↓	Start	pA : N\A	pB : N\A	...
	Line 3	pA : null	pB : N\A	...
↓				
	Line 6	pA : null	pB : null	...
↓				

*infect*

Note that the infected program state has prorogated along the execution.

```
1. int foo (int a, int b)
2. {
3.     POINT pA = null; // bug
4.     ...
5.     for () {
6.         POINT pB = pA;
7.         ...
8.         if () {
9.             pB[3] = 0;
10.            ...
11.        }
12.    }
13. }
```

# Program States

Program states				
↓	Start	pA : N\A	pB : N\A	...
	Line 3	pA : null	pB : N\A	...
↓				
	Line 6	pA : null	pB : null	...
↓				
	Line 9	pA : null	pB : null	...

*crash*

Null pointer reference!

A program crash is the result.

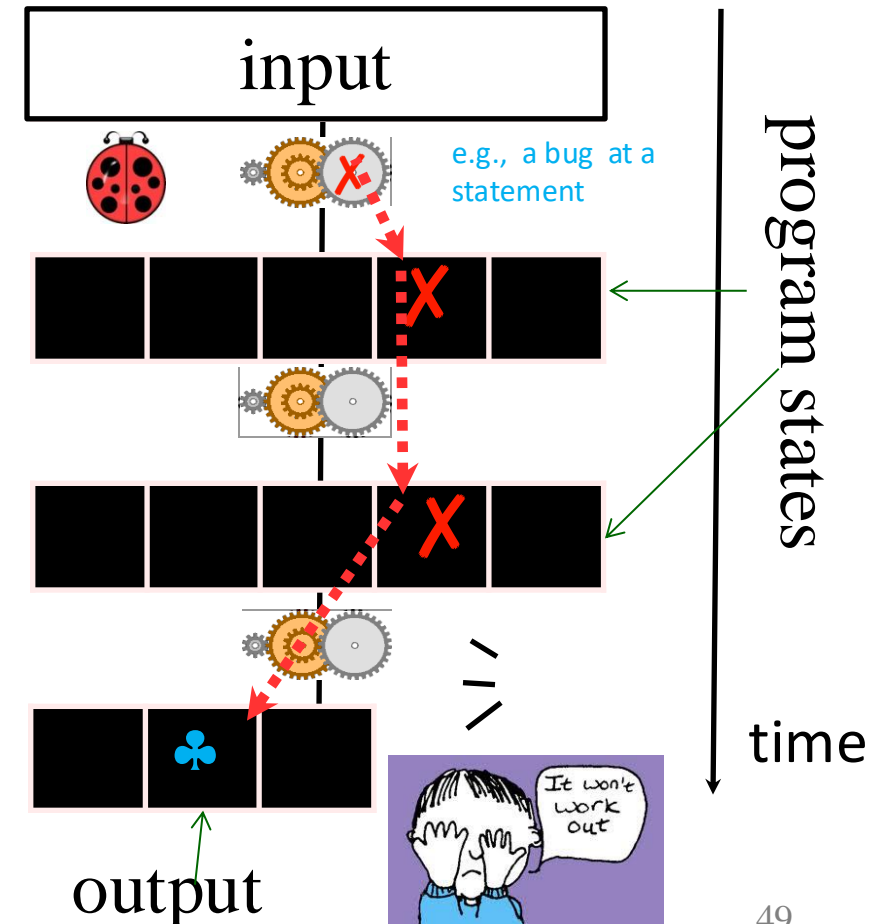
```
1. int foo (int a, int b)
2. {
3.     POINT pA = null; // bug
4.     ...
5.     for () {
6.         POINT pB = pA;
7.         ...
8.         if () {
9.             pB[3] = 0;
10.            ...
11.        }
12.    }
13. }
```



# From Bug to Failure

*Our Debugging Objective:* We want to break the chain so that a bug will not cause the program to produce a wrong output.

1. A programmer creates a *bug*
2. When executed, the fault creates an *infection* [感染] (an error in the program state) (e.g., **X**)
3. The infection *propagates* to make the following statements to compute wrongly.
4. The infection causes a *failure* ♣ at the program output.



# We need Three Steps to Solve the Debugging Problem: “How to Debug?”



**Find** the infection propagation chain (or its fragments) mentioned in the previous slides

**Break** the infection propagation chain so that no failure can be observed.

**Verify** that the identified infection propagation chain *have been broken*, and *all other correct runs still output correctly*.

# The basic idea is to show “*Causes and Effects*”

---

- ◆ The *cause* of any event (“effect”) is a preceding event without which the effect would not have occurred.
- ◆ To **prove** a cause-and-effect relationship, we must show **both legs as follows**:
  1. **the effect DOES occur when the cause DOES occur** (e.g., on slide 43, the run crashes when the variable *pA* at statement 3 keeps a null value.)
  2. **the effect DOES not occur when the cause DOES not occur**. (e.g., if the run does not crash when the variable *pA* at statement 3 does not keep a null value, which require us to revise the program to confirm.)

# The Procedure of Scientific Method [19]

---

1. **Observe** some aspect of *the thing*.
2. Formulate a *hypothesis* consistent with the observation in step 1.
3. Use the hypothesis in step (2) to make *new predictions*.
4. Tests the predictions by **new experiments**
5. **Repeat** 2 to 4 to refine the hypothesis in step 2.

Notice the order of steps 2 and 3 ahead of step 4. Do not swap them.

**Example:** Combining 1 & 2, the hypothesis is wrong. It has no effect on the outcome of the prediction.

---

1. the effect **DOES** occur when the cause **DOES** occur

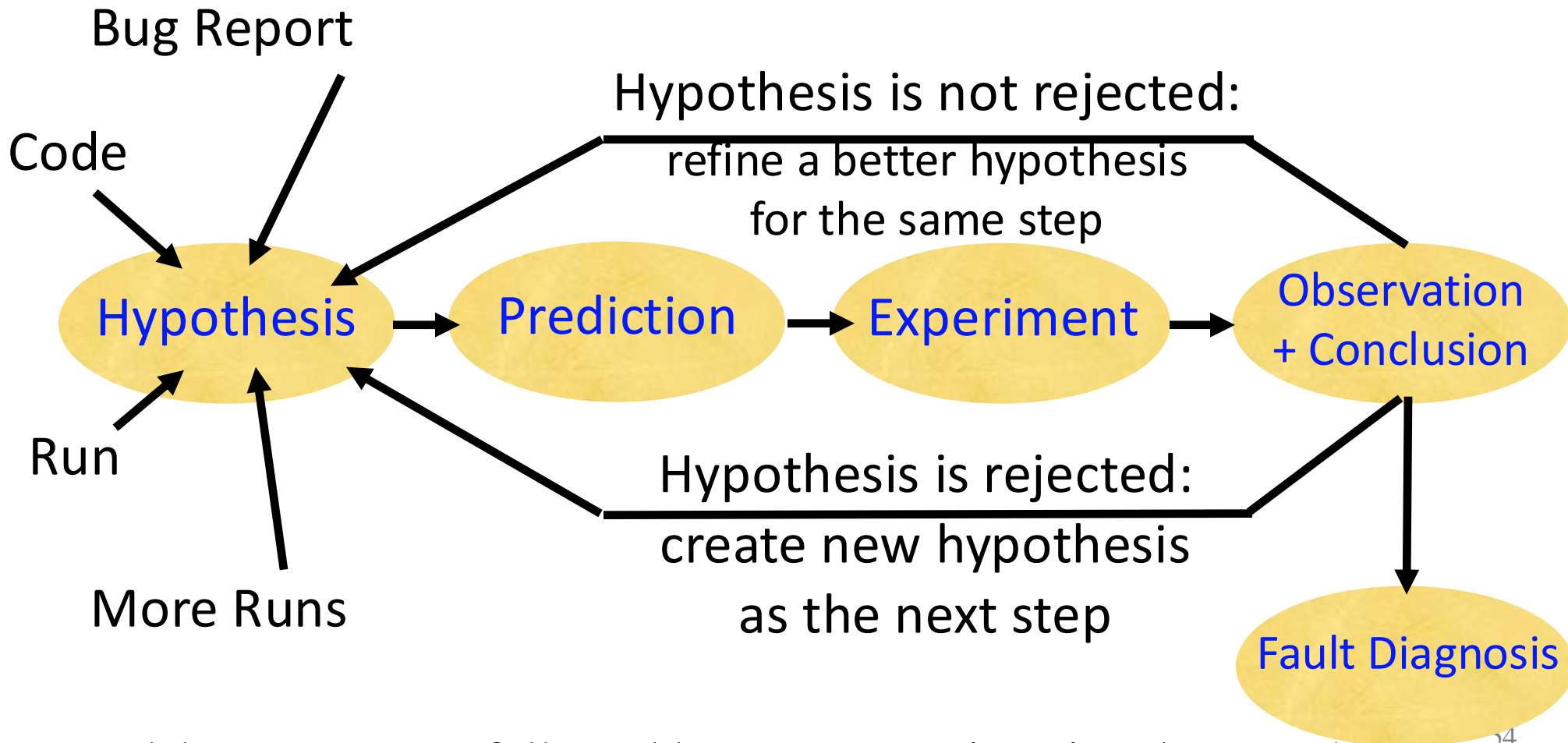
- ◆ *Observation*: John owns and drives a supercar back to office everyday.
- ◆ *Hypothesis*: Every superrich owns a supercar
- ◆ *Prediction*: His home is super large.
- ◆ *Experiment*: Place a home visit.
- ◆ *Observation from this Experiment*: Wow! He lives in a castle

2. the effect **Should NOT** occur when the cause **DOES NOT** occur

- ◆ *Observation*: Johnson does not own any supercar.
- ◆ *Hypothesis*: Every superrich owns a supercar
- ◆ *Prediction*: He is not a superrich. His home is **not** super large.
- ◆ *Experiment*: Place a home visit.
- ◆ *Observation from this Experiment*: Wow! He also lives in a castle!

# Scientific Method for Debugging

## Information



Not to debug a program followed by reverse-engineering the cause!

# Debugging a Buggy “Sort” Program

---

\$ sort 9 8 7

Output: 7 8 9

\$ sort 11 14

Output: 0 11

**Purpose of the given program sort:**

Rearrange the list of integers from the smallest to the largest

So, let's use the scientific method to debug this program.

# Initial Hypothesis

---

Hypothesis	"sort 11 14" computes correctly.
Prediction	Output is "11 14"
Experiment	Run "sort 11 14"
Observation	Output is "0 11"
Conclusion	Hypothesis is <b>rejected</b> .

There is a bug in the program!



We wonder: Does  $a[0] = 0$  hold at this statement (Line 37)? To know it, we testify whether its negation form is true.

```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
```

**Line 6** `qsort(a, argc);`

```
printf("Output: ");
```

**Line 37** `for (i = 0; i < argc - 1; i++)`  
`printf("%d ", a[i]);`

```
printf("\n");
```

```
free(a);
```

```
return 0;
```

```
}
```

From the Initial Hypothesis, we can find out that the value "0" at the output is due to the value at "a[0]".

Note: the run = "sort 11 14"

# Hypothesis 1: $a[]$

---

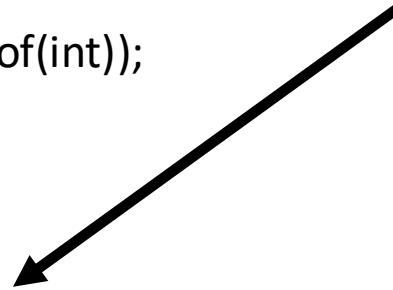
Hypothesis	The run does not cause $a[0] = 0$
Prediction	At Line 37, $a[0]$ should not be 0.
Experiment	Observe the value of $a[0]$ at Line 37.
Observation	$a[0] = 0$ , contradict to the prediction
Conclusion	Hypothesis is <b>rejected</b> .

We now know that  $a[0]$  is a part of the *infected* state

```
int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
```

We wonder: Is the program state of `a[ ]` correct at line 6 (i.e., right before invoking `sell_sort()`)?



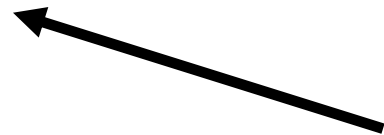
**Line 6** `sell_sort(a, argc);`

```
    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
```

**Line 37** `printf("%d ", a[i]);`

```
    free(a);

    return 0;
}
```



We have known that “`a[0]`” at line 37 is inflected via Hypothesis 1.

# Hypothesis 2: shell\_sort()

---

Hypothesis	The infection does not take place until the procedure <code>shell_sort</code> is about to execute.
Prediction	At Line 6, <code>a[] = [11, 14]; argc = 2</code>
Experiment	Observe <code>a[]</code> and <code>argc</code> at Line 6.
Observation	<code>a[] = [11, 14, 0]; argc = 3.</code>
Conclusion	Hypothesis is <b>rejected</b> .

We now know that both `a[]` and `argc` have been infected.

```
int main(int argc, char *argv[])
{
    int *a;
    int i;
```

*argc* is from the input and has not ever been modified before line 6.

```
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
```

**Line 6** `ell_sort(a, argc);`

We now know that both *a*[] and *argc* have been infected.  
By Hypothesis 2

```
    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
Line 37 printf("%d ", a[i]);
    printf("\n");
```

We have known that “*a*[0]” at line 37 is an infected state by Hypothesis 1.

```
    free(a);

    return 0;
}
```

# Hypothesis 3: argc

---

Hypothesis	Invocation of <code>shell_sort()</code> with <code>size = argc</code> causes the failure.
Prediction	Changing <code>argc</code> to <code>argc - 1</code> at line 6 in code <i>should</i> make the run successful.
Experiment	Change <code>argc</code> to <code>argc - 1</code> at line 6 in code, recompile, and re-run the program with the test case
Observation	The output is now correct.
Conclusion	Hypothesis is <b>confirmed</b> .

# Fixing the Program and Re-Test

```
int main(int argc, char *argv[])
{
    int *a;
    int i;
```

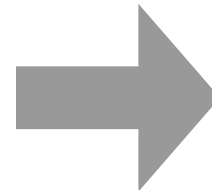
```
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
```

```
    shell_sort(a, argc);
```

```
    shell_sort(a, argc - 1);
```

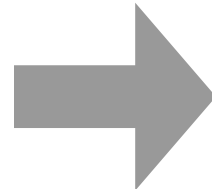
```
}
```

Run the previously failed test cases.  
Run the previously passed test cases as well!



\$ sort 9 8 7

Output: 7 8 9



\$ sort 11 14

Output: 11 14

Note: In C/C++, argc counts the total number of arguments plus 1 (for the program name “sort”).

# Fault Diagnosis

---

- ◆ Cause is “Invoking `shell_sort()` with `argc`”
- ◆ Validated by a pair of experiments:
  - Invoked with `argc`, the failure occurs;
  - Invoked with `argc - 1`, it does not.
- ◆ Side-effect of scientific debugging: we have a *fix*



# Reminder

---

- ◆ The scientific method is **tedious to apply**, but it always works.
  - So, if we have exhausted our own ideas to debug our programs but still failed, try it.
- ◆ A key merit of the scientific method is that we not only locate the bug, but also produce a code fix (i.e., a patch of the code to correct the program!)

# Delta Debugging [17]

---

Input Reduction or Code Reduction

# Motivation of Delta Debugging

---

## ◆ Use case 1:

- A web browser crashed when printing a webpage. If the webpage is large, it is a good idea to make the webpage smaller and at the same time able to crash the web browser before examining the details.

## ◆ Use case 2:

- Tools like Appium [16] or Selenium can help to apply the same sequence of UI events to mobile apps or native apps
  - They are effective tools
  - **Question:** How to integrate with the fuzzing tool like AFL? (as Seeds)
- If the input data sequence is long, it would be better if we could shorten the input data sequence but reproduce these crashes

# Example 1:

## Printing this HTML page crash Mozilla web browser

H1

```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug_severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

← The crash is due to this statement

H2

This HTML page has 40k+ characters

Still produces the same crash after deleting line segments  
H1 and H2

# Input File Reduction

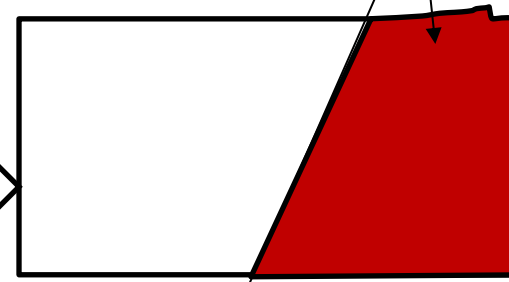
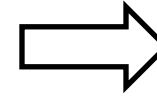
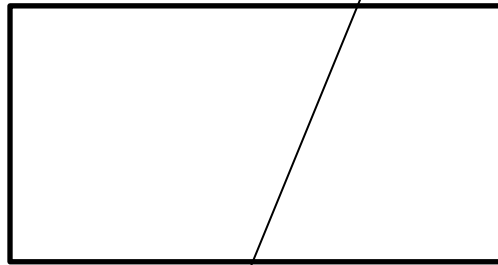
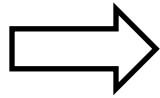
---

- ◆ In the process, ensure we can still crash the web browser by printing the page.
- ◆ Remove single character one at a time: 40k tests
  - Too fine-grained. Look at too many test cases --- tried.
- ◆ More sensible strategy:
  - First: cut away large parts
  - Then: iteratively cut out the other parts bit by bit.

This part still crash the program

# Illustration

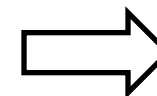
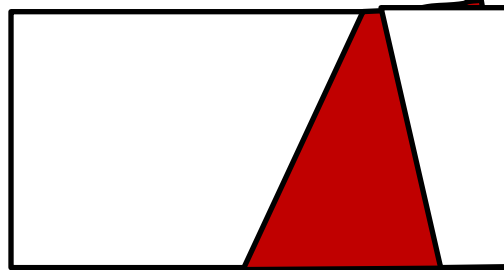
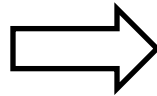
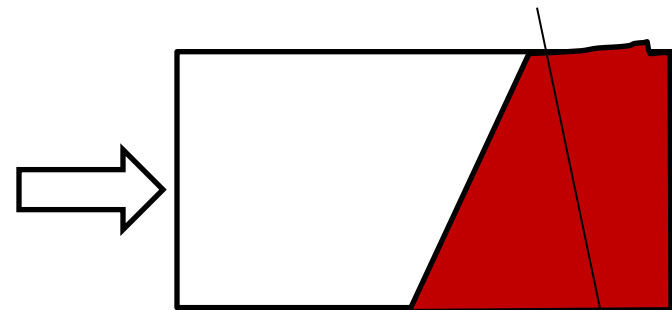
Cut the web page here and there



The failure causing input

Set up the 1st hypothesis

Test the 1st hypothesis



Input with smaller size

Set up the 2nd hypothesis

Test the 2nd hypothesis

more iterations

✂ Strategy: Try by reducing firstly by half (and if not work) then randomly

# Reduction

```
1 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
2 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
3 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
4 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
5 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
6 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
7 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
8 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
9 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
10 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
11 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
12 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
13 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
14 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
15 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
16 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
17 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
18 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
19 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
20 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
21 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
22 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
23 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
24 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
25 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> ✓
26 <SELECT_NAME="priority"_MULTIPLE_SIZE=7> X
```

Try 1, crash.

Try a part of 1 (e.g., 2, 3, 4), no crash, continue, ... find 5

Try a part of 5 (e.g., 6). Lucky, crash found when 6.

Try a part of 6 (e.g., 7, 8, 9, 10). Crash found when 10.

Try a part of 10 (e.g., 11 to 16). Crash found when 16.

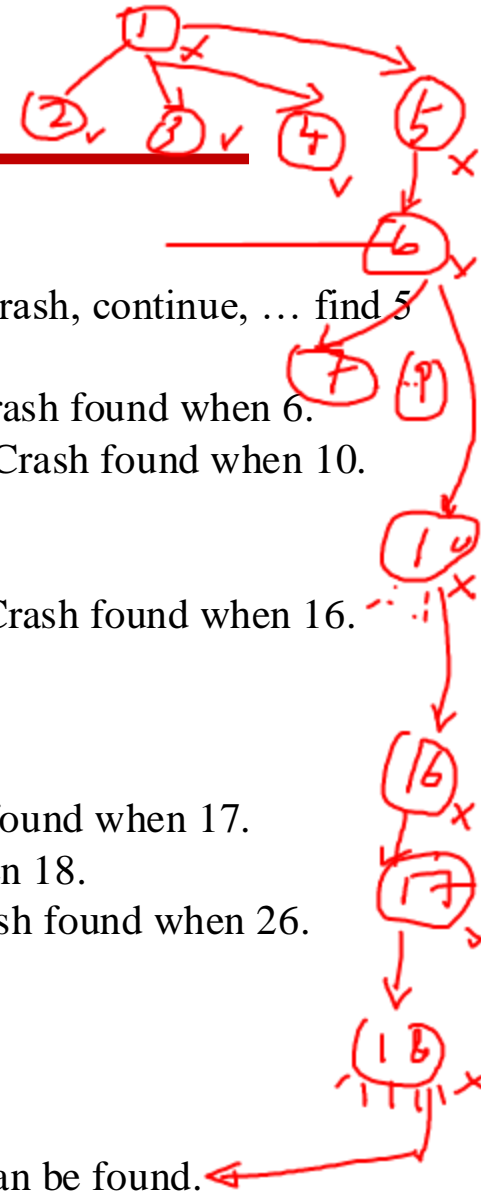
Try a part of 16 (e.g., 17), crash found when 17.

Try a part of 17, crash found when 18.

Try a part of 18 (e.g., 19-26), crash found when 26.

Try a part of 26, no more crash can be found.

So, the case 26 is the reduced input.



# Example 2:

## bug.c to crash gcc V2.95.2 (Intel/Linux)

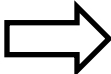
```
#define SIZE 20

double mult(double z[], int n)
{
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}

void copy(double to[], double from[], int count)
{
    int n = (count + 7) / 8;
    switch (count % 8) do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    } while (--n > 0);
    return mult(to, 2);
}

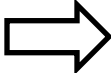
int main(int argc, char *argv[])
{
    double x[SIZE], y[SIZE];
    double *px = x;
    while (px < x + SIZE)
        *px++ = (px - x) * (SIZE + 1.0);
    return copy(y, x, SIZE);
}
```

Which part of this program crashes the gcc compiler?  
=> Remove half of the program, try compiling the program again, and follow the strategy on the last slide.  
=> Remove program segments while making the program syntactically correct



```
#define SIZE 20

double mult(double z[], int n)
{
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```



```
double mult(double z[],int n)
{ int i,j;
  for(;;){
    i=i+j+1;
    z[i]=z[i]*(z[0]+1.0);
  }
  return z[n];
}
```



# Example 2:

## bug.c to crash gcc V2.95.2 (Intel/Linux)

---

Crash the gcc compiler

(a) failing program

```
#define SIZE 20
```

```
double mult(double z[], int n)
{
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Does not crash the gcc compiler

(b) passing program

```
#define SIZE 20
```

```
double mult(double z[], int n)
{
    int i, j;

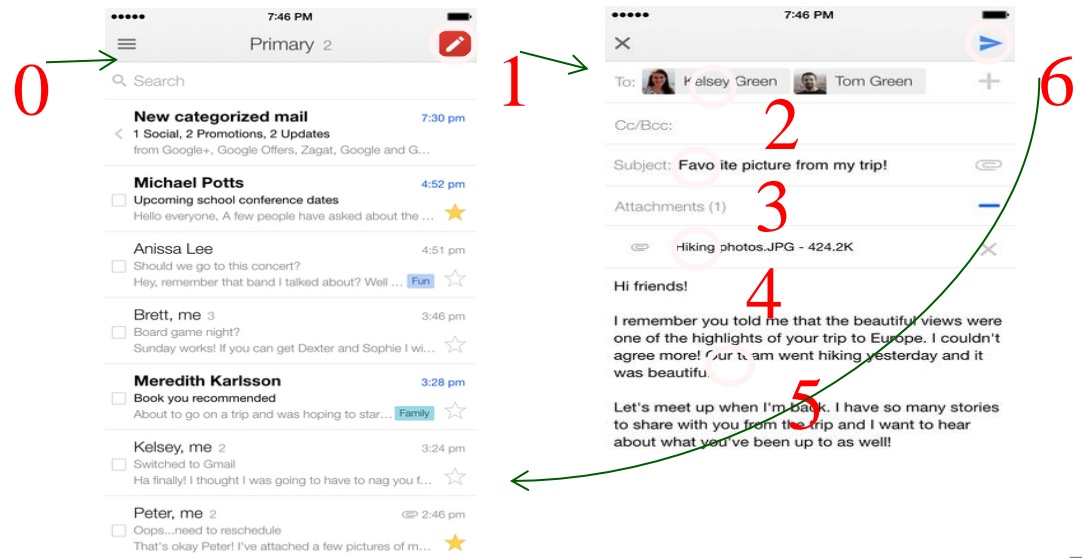
    i = 0;
    for (j = 0; j < n; j++) {
        i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

Actual bug in the gcc compiler:

inlining the expression  $i + j + 1$  in the array accesses  $z[i]$  on the following line.

# Example 3: Shorten Input Data Sequences for Android App?

- ◆ Testing android app generating a long sequence of events
- ◆ Reduce the input event sequence by taking the hierarchical interaction session into account
- ◆ E.g., purge out 2-5 while keeping <0, 1, 6> and crash the app



# Example 3: Shorten Input Data Sequences for Android App?

- ◆ Can reduce the input data sequence to a few events to trigger the same crash and not too slow (spending a few seconds to 1000 seconds in experiment)

Subjects	Yahtzee																	K9mail																								DalvikExplorer					
No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
Original	168	196	153	214	131	100	101	110	121	121	126	128	111	134	123	107	110	171	156	587	216	621	269	184	193	275	216	366	266	217	261	272	215	220	202	154	216	150	269	180	36	75	117	307	950	1525	
LHDD	7	8	7	8	7	8	7	7	8	8	8	7	8	7	7	9	8	7	7	6	6	9	6	4	6	6	5	17	6	11	10	9	6	5	5	5	9	6	7	13	4	4	4	4	4	4	

Subjects	DalvikExplorer					WeightChart					Ringdroid					Tippy					SyncMyPic					WhoHasMyStuff																				
No.	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92
Original	1359	397	404	265	294	16	46	521	514	239	765	448	220	885	449	550	314	646	706	70	76	63	31	102	129	162	46	99	45	426	89	154	177	462	27	44	132	78	306	353	472	50	217	679	149	231
LHDD	4	4	13	4	4	5	6	8	9	7	8	14	9	7	7	7	6	8	7	4	4	6	4	4	4	4	4	4	4	5	4	4	5	9	4	3	6	6	5	6	6	5	6	7	5	6

# Delta Debugging [17] in Nutshell

<https://dzone.com/articles/debugging-step-step-delta>

- ◆ Double-click to open this Word document to see a buggy Quicksort program written in Java.
- ◆ Run it online, say at: <https://www.jdoodle.com/online-java-compiler/>

Input: 4, 12, 9, 14, 3, 10, 17, 11, 8, 7, 4, 1, 6, 19, 5, 21, 2, 3

Output: 1, 3, 2, 5, 3, 4, 4, 6, 7, 8, 9, 10, 11, 12, 14, 17, 19, 21

The output numbers are not in a sorted order. **Failure observed.**

So, the input triggers a failure. We call it as a **failure-inducing input**. [Try Delta Debugging for 30 mins](#) to find a smaller failure-inducing input (see the next slide for process)

**Delta debugging** [17] is to remove some parts of a failure-inducing input to see the reduced input produces the same failure. If yes, treat the reduced input as a failure-inducing input and continue the incremental removal, otherwise, start again from a failure-inducing input.

```
import java.io.*;
import java.util.*;

public class QuickSort
{
    public static void quicksort(int numbers[], int low, int high) {
        int i = low, j = high;
        // Get the pivot element from the middle of the list
        int pivot = numbers[(low + (high-low)/2)];

        // Divide into two lists
        while (i <= j) {
            while (numbers[i] < pivot) {
                i++;
            }
            while (numbers[j] > pivot) {
                j--;
            }
            if (i <= j) {
                exchange(numbers, i, j);
            }
            i++;
            j--;
        }
        // Recursion
        if (low < j)
            quicksort(numbers, low, j); //low
        if (i < high)
            quicksort(numbers, i, high);
    }

    private static void exchange(int numbers[], int i, int j) {
        int temp = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = temp;
        int k = 1;
    }

    public static void main(String argv[])
    {
        int A[] = {4, 12, 9, 14, 3, 10, 17, 11, 8, 7, 4, 1, 6, 19, 5, 21, 2, 3};

        quicksort(A, 0, A.length - 1);

        for (int i=0; i < A.length; i++) System.out.print(A[i] + " ");
        System.out.println();
    }
}
```

this image is a Word document

# In-Class Exercise on Delta Debugging

<https://dzone.com/articles/debugging-step-step-delta>

---

1. Cut the failure-inducing input by half: 4, 12, 9, 14, 3, 10, 17, 11, 8, and get output: 3, 12, 4, 8, 9, 10, 11, 14, 17, i.e., wrong!
2. Since 4, 12, 9, 14, 3, 10, 17, 11, 8 can produce an error, cut it into half again, say using 4, 12, 9, 14, which the output is correct: 4, 9, 12, 14.
3. Our guess in Step 2 is wrong, so we try the other half as input, which is 3, 10, 17, 11, 8, obtaining the output: 3, 8, 10, 11, 17, but the program outputs correctly.
4. Our next input is to reduce our **last failure-inducing** input by cutting it into two parts **randomly**, e.g., cutting the last two numbers. The input is 4, 12, 9, 14, 3, 10, 17. The output is erroneous: 4, 10, 9, 12, 3, 14, 17.
5. We repeat the above process to find a smaller and smaller failure-inducing input to produce the same failure condition (i.e., not in a sorted order).
6. After many rounds, we will get 4, 3, 5, 2, 3 as a failure-inducing input.
  - If you choose a different random split, you may get a smaller input to reproduce the failure

**Keep your results for your revision in a later time**

# Progress Illustration for the Exercise

## [the cutting atop slide 78]

[4, 12, 9, 14, 3, 10, 17, 11, 8, 7, 4, 1, 6, 19, 5, 21, 2, 3]

Output: [1, 3, 2, 5, 3, 4, 4, 6, 7, 8, 9, 10, 11, 12, 14, 17, 19, 21]. ✗

Divide, and try the first half

try the second half

[4, 12, 9, 14, 3, 10, 17, 11, 8]

Output: [3 12 4 8 9 10 11 14 17]. ✗

try the second half

Divide, and try the first half

[3, 10, 17, 11, 8]

Output: [3 8 10 11 17]. ✓ correct.

We guess wrong

[4, 12, 9, 14]

Output: [4 9 12 14]. ✓ correct. We guess wrong

What level of failure you seen?

- The output is unsorted? ❶
- Any one of “3, 2” and “5, 3”? ❷

[7, 4, 1, 6, 19, 5, 21, 2, 3]

Output: [1 2 4 5 3 6 7 19 21]. ✗

Divide, and try the first half

many rounds

Can't reduce further. So, this is the result [\* , \* , \* , \*]

# References and Resources

---

1. Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, New York, NY, USA, 259-269. DOI: <https://doi.org/10.1145/3238147.3238177>
2. Anca Deak, Tor Stålhane, and Guttorm Sindre. 2016. Challenges and strategies for motivating software testing personnel. Inf. Softw. Technol. 73, C (May 2016), 1-15. DOI=<http://dx.doi.org/10.1016/j.infsof.2016.01.002>
3. David Martin, John Rooksby, Mark Rouncefield, and Ian Sommerville. 2007. 'Good' Organisational Reasons for 'Bad' Software Testing: An Ethnographic Study of Testing in a Small Software Company. In Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 602-611. DOI=<http://dx.doi.org/10.1109/ICSE.2007.1>
4. Omission fault: <http://www.exampler.com/testing-com/writings/omissions.html>
5. Android monkey: <https://developer.android.com/studio/test/monkey>
6. Konstantin Böttinger, Patrice Godefroid, Rishabh Singh: Deep Reinforcement Fuzzing. IEEE Symposium on Security and Privacy Workshops 2018: 116-122. <https://arxiv.org/pdf/1801.04589.pdf>
7. Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In Proceedings of the 13th International Workshop on Automation of Software Test (AST '18). ACM, New York, NY, USA, 34-37. DOI: <https://doi.org/10.1145/3194733.3194742>

# References and Resources

---

8. American fuzzy lop (AFL)
  - Wikipedia page: [https://en.wikipedia.org/wiki/American\\_fuzzy\\_lop\\_\(fuzzer\)](https://en.wikipedia.org/wiki/American_fuzzy_lop_(fuzzer))
  - Tutorial to run AFL: <https://fuzzing-project.org/tutorial3.html>
  - Official website and source code of AFL: <http://lcamtuf.coredump.cx/afl/>
9. Additional tutorials on AFL: <https://github.com/ThalesIgnite/afl-training>
  - Needs 3-4 hours to complete
10. Tobias Ospelt. AFL --- American fuzzy lop: a short introduction:  
<http://floyd.ch/download/introduction-fuzzing-with-afl.pdf>
  - Include the commands and the usage of these commands
11. Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15). IEEE Computer Society, Washington, DC, USA, 429-440. DOI: <https://doi.org/10.1109/ASE.2015.89>
  - The code is here: <http://bear.cc.gatech.edu/~shauvik/androtest/>



# References and Resources

---

12. Faster Fuzzing: Reinitialization with Deep Neural Models <https://arxiv.org/pdf/1711.02807.pdf>
  - Using deep learning to generate seeds to guide AFL to explore program paths
13. Caroline Lemieux and Koushik Sen. FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage. In Proceedings of ASE'18.
  - <http://www.carolemieux.com/fairfuzz-ase18.pdf>
  - Source code: <https://github.com/carolemieux/afl-rb>
14. Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). ACM, New York, NY, USA, 1032-1043. DOI: <https://doi.org/10.1145/2976749.2978428>
  - Source code: <https://github.com/mboehme/aflfast>
15. Bo Jiang, Yuxuan Wu, Teng Li, and W. K. Chan. 2017. SimplyDroid: efficient event sequence simplification for Android application. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), 297-307.
  - Source code: <https://github.com/gongbell/SimplyDroid>
16. Appium
  - introduction: <https://hackernoon.com/an-introduction-to-appium-the-best-open-source-mobile-app-automation-tool-940f5eaabae9>
  - Source code <http://appium.io/>
  - Tutorials: <http://www.automationtestinghub.com/appium-tutorial/>

# References and Resources

---

17. Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. IEEE Trans. Softw. Eng. 28, 2 (February 2002), 183-200. DOI=<http://dx.doi.org/10.1109/32.988498>
  - This is the delta debugging paper
18. Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous Deployment at Facebook and OANDA. In Proceedings of ICSE 2016 companion, pages 21-30, 2016. <http://dx.doi.org/10.1145/2889160.2889223>
19. Andreas Zeller. Why Programs Fail: A Guide to Systematic Debugging, 2<sup>nd</sup> edition. ISBN-13: 978-0123745156. Morgan Kaufmann
20. Patrice Godefroid.. Fuzzing: Hack, Art, and Science. Communications of the ACM, February 2020, Vol. 63 No. 2, Pages 70-76 <https://cacm.acm.org/magazines/2020/2/242350-fuzzing/fulltext#R8>

# Appendix

---

# **Cheat Sheet for Running Java Project in Eclipse**

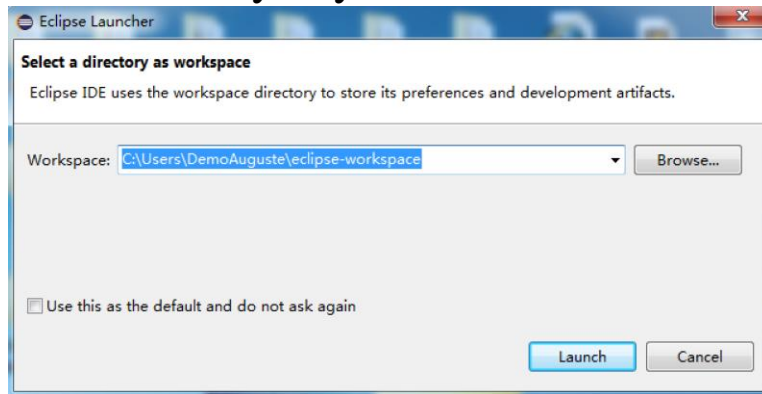
**<https://www.eclipse.org/>**

---

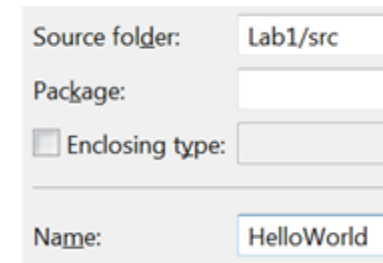
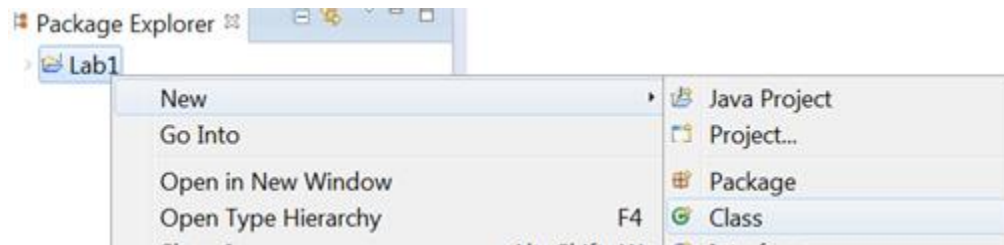
# Cheat Sheet for Running Java Project in Eclipse

## Write, Compile, Execute

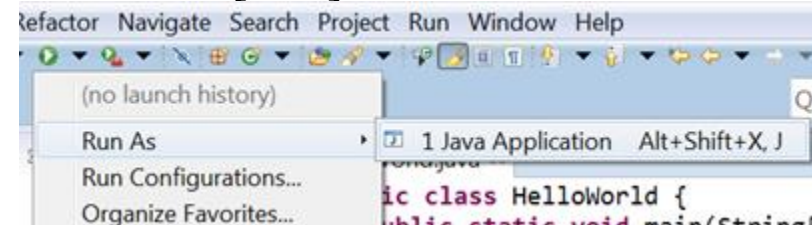
1. Start Eclipse from the CSLab Menu (or install yourself)
2. Select a directory at your H: drive as the workspace
6. Type **HelloWorld** in the Name field (note the Title Case and no space in between), and then press [Finish]



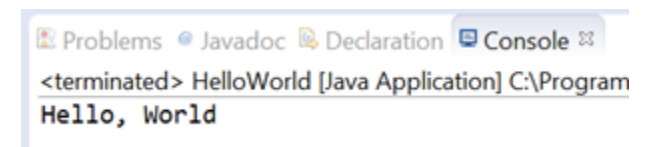
3. Select File->New->Java Project
4. Type “**Lab1**” as the **Project Name** and then press “Finish”.
5. Right click the “Lab1” project. Select New->Class



7. Copy the code in the next slide into the **HelloWorld.java** panel
8. Press the [Run] button on the toolbar



9. Check the output in the Console panel



# *Cheat Sheet for Running Java Project in Eclipse*

## **Hello World**

---

**Note:** filename must match the name of the class.

```
/******  
 * Prints "Hello, World"  
 * Everyone's first Java program.  
 *****/  
  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World");  
    }  
}
```

HelloWorld.java

It is a common practice to use the first comment to describe the purpose of the Java class.

# *Cheat Sheet for Running Java Project in Eclipse*

## Quadratic Equation

---

**Note:** filename must match the name of the class. Quadratic.java

```
public class Quadratic {  
    public static void main(String[] args) {  
  
        // parse coefficients from command-line  
        double b = Double.parseDouble(args[0]);  
        double c = Double.parseDouble(args[1]);  
  
        double root1 = 0; // the variable to keep the first root  
        double root2 = 0; // the variable to keep the second root  
        // calculate roots  
  
        double discriminant = b*b - 4.0*c;  
        double d = Math.sqrt(discriminant);  
        root1 = (-b + d) / 2.0;  
        root2 = (-b - d) / 2.0;  
  
        // print them out  
        System.out.println(root1);  
        System.out.println(root2);  
    }  
}
```

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

# *Cheat Sheet for Running Java Project in Eclipse*

## **JUnit (a popular mean to run test cases)**

---

### Procedure

1. Right click the project Lab1. New a **package** entitled *Samples* under our project Lab1
2. Move the two classes written into this *Sample* package.
  - View from the Windows Explore to see what the contents of the Lab1\src are?
3. Right click the project *Lab1*. Select new->JUnit Test Case
  - Name the **package** of the new test case as *testcases*
  - Name your test case as **TestXXX**, where **XXX** can be any string, but it must be started with **Test**.
    - E.g., Test001
4. Place the code from the next slide into this test case Test001.
  - Each test method started with the annotation “@Test@” if you use Junit version 4.0 or earlier
5. Select from the menu Run->Run As->JUnit Test Case



# *Cheat Sheet for Running Java Project in Eclipse*

## **The Test001 test case**

We only make use of the automation power of JUnit to ease us to pass input values to our programs developed so far. We have not used the real power of JUnit to help test our programs.

JUnit5

// for **JUnit4**, replace by these two lines  
**import static org.junit.Assert.\*;**  
**import org.junit.Test;**

For JUnit 4, see <https://junit.org/junit4/faq.html>

For JUnit 5, see  
<https://junit.org/junit5/docs/current/user-guide/>

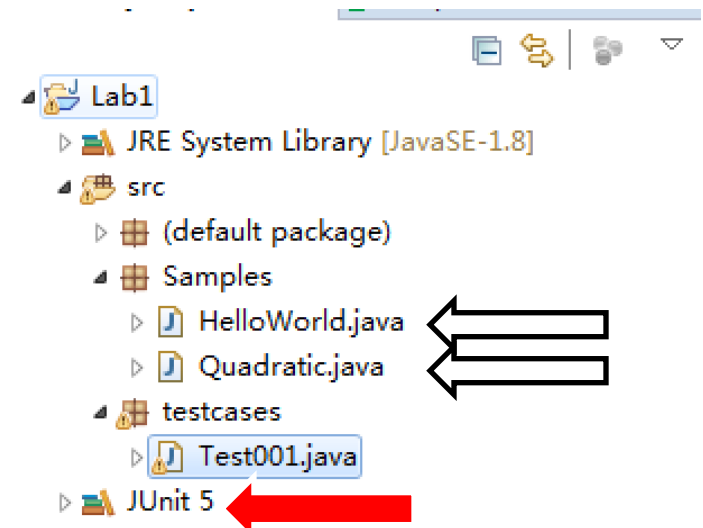
```
package testcases;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

import Samples.*;

public class Test001 {
    @Test
    public void testHelloWord() {
        String[] s1 = null;
        HelloWorld.main(s1);
    }

    @Test
    public void testQuadratic() {
        String[] s1 = { "-3", "2" };
        Quadratic.main(s1);
    }
}
```



check to see  
using **JUnit5** or  
**JUnit4**