Lecture 0 – Introduction

AAA705: Software Testing and Quality Assurance

Jihyeok Park

2026 Fall

Course Information

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** By appointment via e-mail
  - **Office:** 507, Jung Woonoh IT & General Education Center
  - **Email:** jihyeok_park@korea.ac.kr

Course Information

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** By appointment via e-mail
  - **Office:** 507, Jung Woonoh IT & General Education Center
  - **Email:** jihyeok_park@korea.ac.kr

- **Class:** AAA705: Software Testing and Quality Assurance

Course Information

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** By appointment via e-mail
  - **Office:** 507, Jung Woonoh IT & General Education Center
  - **Email:** jihyeok_park@korea.ac.kr
- **Class:** AAA705: Software Testing and Quality Assurance
- **Lectures:** 09:00–12:00, Mon. and Wed. @ 304 애기능생활관

Course Information

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** By appointment via e-mail
  - **Office:** 507, Jung Woonoh IT & General Education Center
  - **Email:** jihyeok_park@korea.ac.kr

- **Class:** AAA705: Software Testing and Quality Assurance

- **Lectures:** 09:00–12:00, Mon. and Wed. @ 304 애기능생활관

- **Homepage:** https://plrg.korea.ac.kr/courses/aaa705/

Schedule

Week	Date	Contents
1	09/02	Introduction
2	09/07	Combinatorial Testing
	09/09	Random Testing
3	09/14	Coverage Criteria
	09/16	Search Based Software Testing (SBST)
4	09/21	Dynamic Symbolic Execution (DSE)
	09/23	Mutation Testing
5	09/28	Regression Testing
	09/30	Fault Localization
6–7	10/05–10/14	**No Offline Lectures (Homework)**
8	10/19	Testing Oracles
	10/21	Course Review

Grading

- **4 Homework Assignments: 80%**
  - Each assignment is worth **20%**:
    - Homework 1 (due on Sep. 28) – Coverage Measurement
    - Homework 2 (due on Oct. 5) – Test Generation
    - Homework 3 (due on Oct. 12) – Mutation Testing and Minimization
    - Homework 4 (due on Nov. 2) – Fault Localization
  - Submit your homework on **LMS**.

Grading

- **4 Homework Assignments: 80%**
  - Each assignment is worth **20%**:
    - Homework 1 (due on Sep. 28) – Coverage Measurement
    - Homework 2 (due on Oct. 5) – Test Generation
    - Homework 3 (due on Oct. 12) – Mutation Testing and Minimization
    - Homework 4 (due on Nov. 2) – Fault Localization
  - Submit your homework on **LMS**.
- **Attendance: 20%**
  - Please use **LMS** to attend the class with the code provided.
  - **All or nothing:** $\geq 2/3$ of the checks $\rightarrow$ **20%**, otherwise **0%**.
  - **No** attendance check in the first week; it starts on **Sep. 7**.

Course Materials

- **Self-contained lecture notes.**

https://plrg.korea.ac.kr/courses/aaa705/

Course Materials

- **Self-contained lecture notes.**

  https://plrg.korea.ac.kr/courses/aaa705/

- **References**: this course is developed with reference to the following materials.
  - **"Introduction to Software Testing (2nd Ed.)"** by Paul Ammann and Jeff Offutt.
  - **"Why Programs Fail (2nd Ed.)"** by Andreas Zeller.
  - **"CS453: Automated Software Testing"** by **Shin Yoo** @ KAIST.

Contents

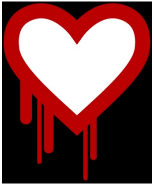
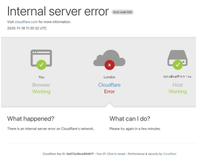
1. Why Software Testing?

2. Terminologies in Software Testing
   Types of Software Quality
   Faults vs. Errors vs. Failures
   More Terminologies

3. Software Testing Techniques

Errors in Safety-Critical Software

Unexpected faults in **safety-critical software** cause serious problems:

			
Rocket Ariane 5 (1996)	**Security** Heartbleed (2014)	**Global IT** CrowdStrike (2024)	**Cloud** Cloudflare (2025)

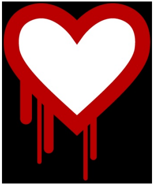
Errors in Safety-Critical Software

Unexpected faults in **safety-critical software** cause serious problems:

Rocket

Ariane 5

(1996)



Security

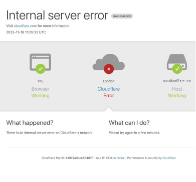
Heartbleed

(2014)

Global IT

CrowdStrike

(2024)



Cloud

Cloudflare

(2025)

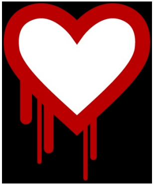
Then, how can we **prevent** such software faults?

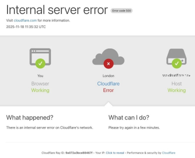
Errors in Safety-Critical Software

Unexpected faults in **safety-critical software** cause serious problems:


June 4, 1996: Ariane-5 explodes after lift off

Today in History: June 4, 1996: Ariane-5 explodes after lift off





Rocket	**Security**	**Global IT**	**Cloud**
Ariane 5	Heartbleed	CrowdStrike	Cloudflare
(1996)	(2014)	(2024)	(2025)

Then, how can we **prevent** such software faults?

Can we **automatically check** whether a program does not have any software faults?

Detecting Software Faults

How do we know whether a software is correct?

Detecting Software Faults

How do we know whether a software is correct?

vs.

Empiricists – Francis Bacon

Rationalists – René Descartes

It is correct because I **TESTED** *several times but no error was found!*

It is correct because I formally **PROVED** *that no error exists!*

Detecting Software Faults

We can use various **analysis** techniques to detect software faults.

An **analyzer** is a program that takes a **program** and a **property** as inputs and determines whether the program **satisfies** the property.

Detecting Software Faults

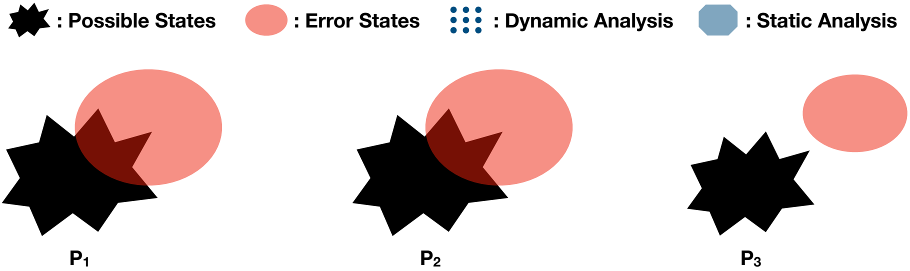
We can use various **analysis** techniques to detect software faults.

An **analyzer** is a program that takes a **program** and a **property** as inputs and determines whether the program **satisfies** the property.

We can categorize them into two groups:

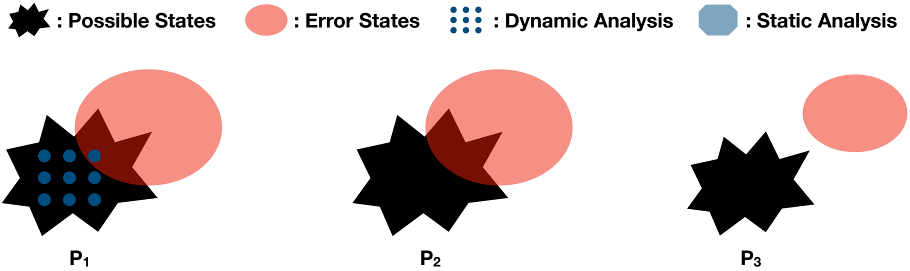
- **Dynamic analyzers** analyze programs by **executing** them.
- **Static analyzers** analyze programs **without executing** them.

Dynamic Analysis vs. Static Analysis



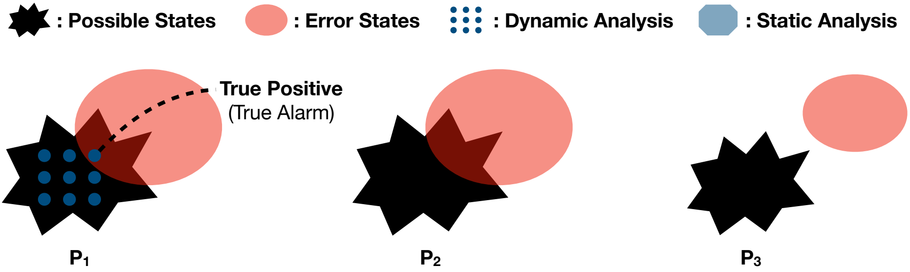
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



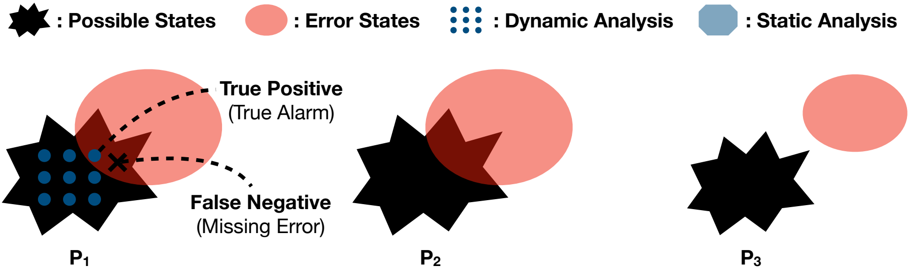
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



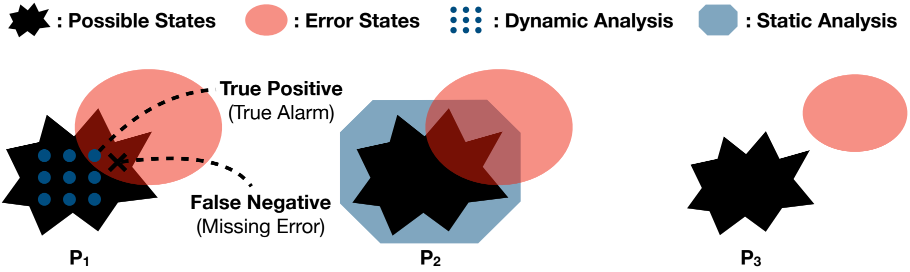
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



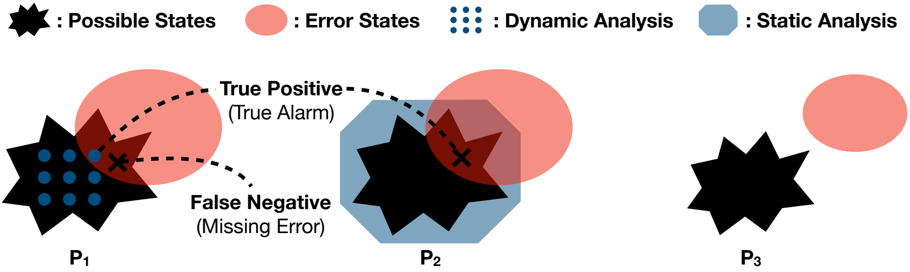
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



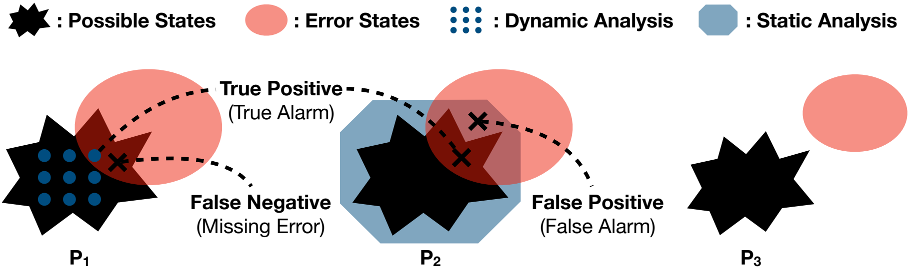
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



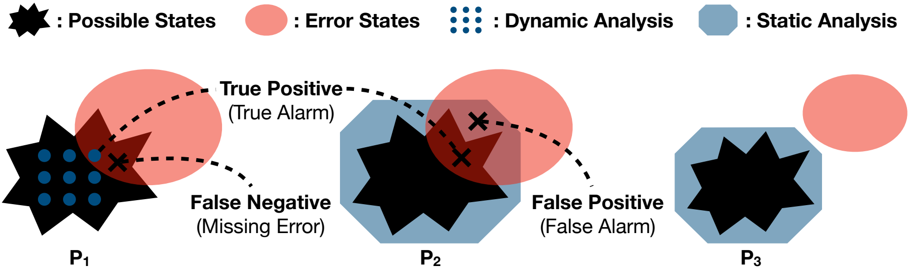
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



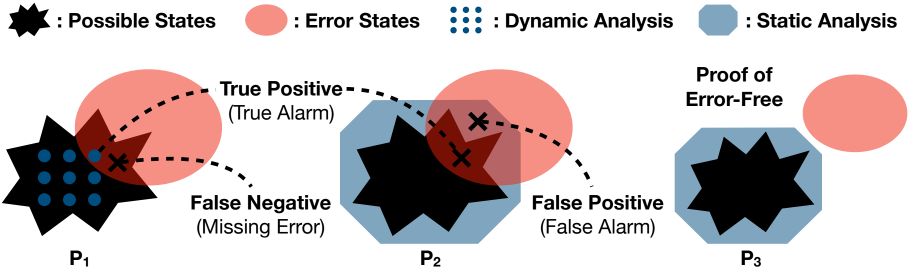
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



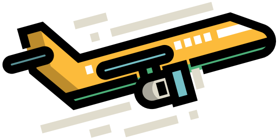
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Dynamic Analysis vs. Static Analysis



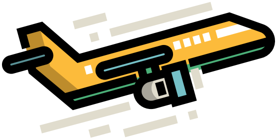
Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

Why Software Testing?



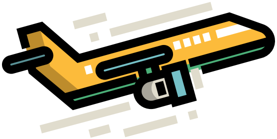
- Imagine you have two choices when boarding a airplane:
  - While an airplane A has **never been proven** to have any run-time errors, it has been **tested** with a finite number of test flights.
  - While an airplane B has been **formally verified** to have no run-time errors, it has **never been tested** in the real world.

Why Software Testing?



- Imagine you have two choices when boarding a airplane:
  - While an airplane A has **never been proven** to have any run-time errors, it has been **tested** with a finite number of test flights.
  - While an airplane B has been **formally verified** to have no run-time errors, it has **never been tested** in the real world.
- Some people may choose A, while others may choose B.

Why Software Testing?



- Imagine you have two choices when boarding a airplane:
  - While an airplane A has **never been proven** to have any run-time errors, it has been **tested** with a finite number of test flights.
  - While an airplane B has been **formally verified** to have no run-time errors, it has **never been tested** in the real world.
- Some people may choose A, while others may choose B.
- In addition, some properties only can be **tested** but not **verified** (e.g., energy consumption, usability, etc.).

Contents

1. Why Software Testing?

2. Terminologies in Software Testing
   - Types of Software Quality
   - Faults vs. Errors vs. Failures
   - More Terminologies

3. Software Testing Techniques

Software testing *is an* **empirical**, **technical** *investigation conducted to provide stakeholders with information about the* **quality** *of the product or service under test.*

— Cem Kaner (2006)

Types of Software Quality: Dependability

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

Types of Software Quality: Dependability

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.

Types of Software Quality: Dependability

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.
- **Reliability**: the software should have a **high probability** of being **correct** for a period of time.

Types of Software Quality: Dependability

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.
- **Reliability**: the software should have a **high probability** of being **correct** for a period of time.
- **Safety**: the software should pose **no risk** of any kind of **hazard** (loss of life, injury, etc.).

Types of Software Quality: Dependability

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.
- **Reliability**: the software should have a **high probability** of being **correct** for a period of time.
- **Safety**: the software should pose **no risk** of any kind of **hazard** (loss of life, injury, etc.).
- **Robustness**: the software should reasonably **remain dependable** even if surrounding **environment changes**.

Types of Software Quality: Performance

Apart from dependability, the software should meet certain **performance** expectations.

Types of Software Quality: Performance

Apart from dependability, the software should meet certain **performance** expectations.

- For example, execution time, network throughput, memory usage, number of simultaneous users, etc.

Types of Software Quality: Performance

Apart from dependability, the software should meet certain **performance** expectations.

- For example, execution time, network throughput, memory usage, number of simultaneous users, etc.
- **Hard to thoroughly test** due to the heavy reliance on the **execution environment** and **usage patterns**.

Types of Software Quality: Usability

The software should be **usable**.

Types of Software Quality: Usability

The software should be **usable**.

- In general, there is no universally accepted criterion for **usability**.

Types of Software Quality: Usability

The software should be **usable**.

- In general, there is no universally accepted criterion for **usability**.
- Usability testing usually involves **user studies**, such as **focus groups**, **beta-testing**, **A/B testing**, etc.

Types of Software Quality: Ethics

The software should be **ethical**.

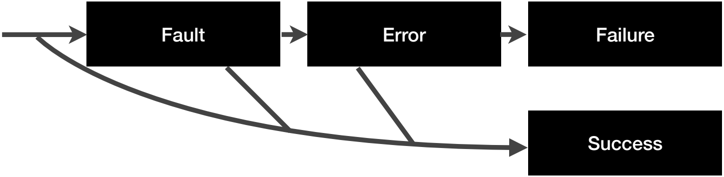
Types of Software Quality: Ethics

The software should be **ethical**.

- Typically, this is applied to AI/ML based systems.
- **[FSE'17]** S. Galhotra, Y. Brun, and A. Meliou. "*Fairness testing: testing software for discrimination.*"
- **[TOSEM'24]** Z. Chen, J. M. Zhang, M. Hort, M. Harman, and F. Sarro. "*Fairness testing: a comprehensive survey and analysis of trends.*"
- **[FSE'26]** X. Li, Z. Chen, J. M. Zhang, Y. Xiao, T. Li, W. Sun, Y. Liu, Y. Lou, and X. Liu. "*Fairness testing of large language models in role-playing.*"

Faults vs. Errors vs. Failures

The purpose of testing is to **detect** and **remove** faults, errors, and failures.



From **IEEE Standard 729-1983**, IEEE Standard Glossary of Software Engineering Terminology[1]

- **Fault**: an anomaly in the software that may lead to an error.
- **Error**: a runtime effect of executing a fault, which may cause a failure.
- **Failure**: a manifestation of an error external to the software.

[1] https://ieeexplore.ieee.org/document/7435207

Faults vs. Errors vs. Failures

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {
  let result = 0;
  for (let i = 0; i < arr.length; i++) {
    // fault: `i` should be fixed to `arr[i]`
    result += i;
  }
  return result;
}
```

It is a **fault** but **not an error** until the function is executed.

```
// the faulty statement is not reached at runtime (no error)
assert(sum([]) === 0);
```

Faults vs. Errors vs. Failures

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {
  let result = 0;
  for (let i = 0; i < arr.length; i++) {
    // fault: `i` should be fixed to `arr[i]`
    result += i;
  }
  return result;
}
```

It is an **error** with the following input but **not a failure** because the output is **coincidentally correct**.

```
// the faulty statement is reachable at runtime (error)
// the output is coincidentally correct (no failure)
assert(sum([4, -2, 1]) === 3);
```

Faults vs. Errors vs. Failures

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {
  let result = 0;
  for (let i = 0; i < arr.length; i++) {
    // fault: `i` should be fixed to `arr[i]`
    result += i;
  }
  return result;
}
```

It is a **failure** with the following input because the output is **incorrect**.

```
// the output is incorrect (failure)
assert(sum([3, 7, 4]) === 14);
```

More Terminologies

- **Test Input**: a set of inputs that are used to test a program.

More Terminologies

- **Test Input**: a set of inputs that are used to test a program.
- **Test Oracle**: a mechanism to determine whether the program behaves correctly.

More Terminologies

- **Test Input**: a set of inputs that are used to test a program.
- **Test Oracle**: a mechanism to determine whether the program behaves correctly.
- **Test Case**: a pair of a test input and a test oracle.

More Terminologies

- **Test Input**: a set of inputs that are used to test a program.
- **Test Oracle**: a mechanism to determine whether the program behaves correctly.
- **Test Case**: a pair of a test input and a test oracle.
- **Test Suite**: a set of test cases.

More Terminologies

- **Test Input**: a set of inputs that are used to test a program.
- **Test Oracle**: a mechanism to determine whether the program behaves correctly.
- **Test Case**: a pair of a test input and a test oracle.
- **Test Suite**: a set of test cases.
- **Test Effectiveness**: the ability of a test suite to detect faults or achieve other testing objectives.

More Terminologies

PLRG

- **Test Input**: a set of inputs that are used to test a program.
- **Test Oracle**: a mechanism to determine whether the program behaves correctly.
- **Test Case**: a pair of a test input and a test oracle.
- **Test Suite**: a set of test cases.
- **Test Effectiveness**: the ability of a test suite to detect faults or achieve other testing objectives.
- **Testing** vs. **Debugging**: testing is the process of detecting faults, while debugging is the process of fixing faults.

Contents

1. Why Software Testing?

2. Terminologies in Software Testing
   - Types of Software Quality
   - Faults vs. Errors vs. Failures
   - More Terminologies

3. Software Testing Techniques

Problem – Sampling the Input Space

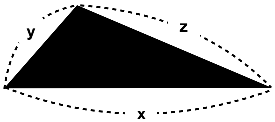
- **Exhaustive Testing**: Can we test a program with all possible inputs?

Problem – Sampling the Input Space

- **Exhaustive Testing**: Can we test a program with all possible inputs? In theory, **Yes!**

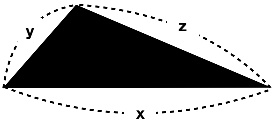
Problem – Sampling the Input Space

- **Exhaustive Testing**: Can we test a program with all possible inputs? In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns whether they can form a **triangle** and, if so, **its type**.



Problem – Sampling the Input Space

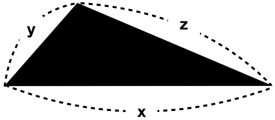
- **Exhaustive Testing**: Can we test a program with all possible inputs? In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns whether they can form a **triangle** and, if so, **its type**.



  - How many possible inputs are there?

Problem – Sampling the Input Space

- **Exhaustive Testing**: Can we test a program with all possible inputs? In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns whether they can form a **triangle** and, if so, **its type**.



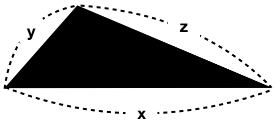
  - How many possible inputs are there?

$$2^{32} \times 2^{32} \times 2^{32} = 2^{96} \approx 7.9 \times 10^{28}$$

  - Approximated number of stars in the universe: 10^{24}

Problem – Sampling the Input Space

- **Exhaustive Testing**: Can we test a program with all possible inputs? In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns whether they can form a **triangle** and, if so, **its type**.



  - How many possible inputs are there?

$$2^{32} \times 2^{32} \times 2^{32} = 2^{96} \approx 7.9 \times 10^{28}$$

  - Approximated number of stars in the universe: 10^{24}
- Testing allows only a **sampling** of an enormous **input space**. The difficulty lies in how to come up with **effective sampling**.

Problem – Test Oracle

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).

Problem – Test Oracle

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).

- How to define the **oracle**?

Problem – Test Oracle

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).
- How to define the **oracle**?
- Without an explicit oracle, we can only detect a small subset of faults. (e.g., **crash**, **unintended infinite loop**, **division by zero**, etc.)

Problem – Test Oracle

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).
- How to define the **oracle**?
- Without an explicit oracle, we can only detect a small subset of faults. (e.g., **crash**, **unintended infinite loop**, **division by zero**, etc.)
- We need to **define** or **infer** the oracle for testing.

Software Testing Techniques

- There is no fixed recipe for software testing.

Software Testing Techniques

- There is no fixed recipe for software testing.
- We need to understand the pros and cons of each testing technique.

Software Testing Techniques

- There is no fixed recipe for software testing.
- We need to understand the pros and cons of each testing technique.
- There are two major categories of testing techniques:
  - **Black-box Testing**: testing **without** knowing the internal structure of the program.
  - **White-box Testing**: testing **with** the knowledge of the internal structure of the program.

Black-box Testing

- **Combinatorial Testing**
  - Tester utilizes **input specifications** to generate test cases.
- **Random Testing**
  - Tester **randomly** selects test cases from the input space.
  - It can be used for **white-box testing** as well.

White-box Testing

Sometimes called **structural testing** because it uses the **internal structure** of the program to derive test cases.

- **Coverage Criteria**
  - The adequacy of a test suite is measured in terms of the **coverage** of the program's internal structure.
- **Search Based Software Testing (SBST)**
  - A technique that uses **meta-heuristic search** algorithms to maximize/minimize a certain **fitness function**.
- **Dynamic Symbolic Execution (DSE)**
  - A technique that systematically explores the input space using **symbolic execution** with **dynamic analysis**.

General Techniques

- **Mutation Testing**
  - A technique that evaluates the quality of a test suite by introducing **artificial faults** to the program.
- **Regression Testing**
  - A technique that ensures that a change in the program does not introduce new faults.
- **Fault Localization**
  - A technique that identifies the **location** of a fault in the program.
- **Metamorphic Testing**
  - A technique that tests a program using **metamorphic relations**.
- **Differential Testing**
  - A technique that tests a program by comparing the outputs of **multiple implementations**.

Next Lecture

- Combinatorial Testing

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr