Joint Degree Program of Fudan University and University College Dublin (UCD)

SOFT620020.01
Advanced Software Engineering

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io

Course Outline

Date	Topic
Sep. 09	Introduction
Sep. 16	**Testing Overview**
Sep. 23	Guided Random Testing
Sep. 30	Search-Based Testing
Oct. 12	Performance Analysis
Oct. 14	Presentation 1
Oct. 21	Security Testing
Oct. 28	Compiler Testing

Date	Topic
Nov. 04	Mobile Testing
Nov. 11	Delta Debugging
Nov. 18	Presentation 2
Nov. 25	Bug Localization
Dec. 02	Automatic Repair
Dec. 09	Symbolic Execution
Dec. 16	Big Code Analysis
Dec. 23	Presentation 3

Why Software Testing is Needed?

Railway Traffic Accident at Yong Wen Line
Two trains crashed and 40 people died.

ATM Machine Accident in Guangzhou
Only 1 RMB was deducted from the account after withdrawing 1000 RMB from the ATM.

Discussion 1 – Software Accidents

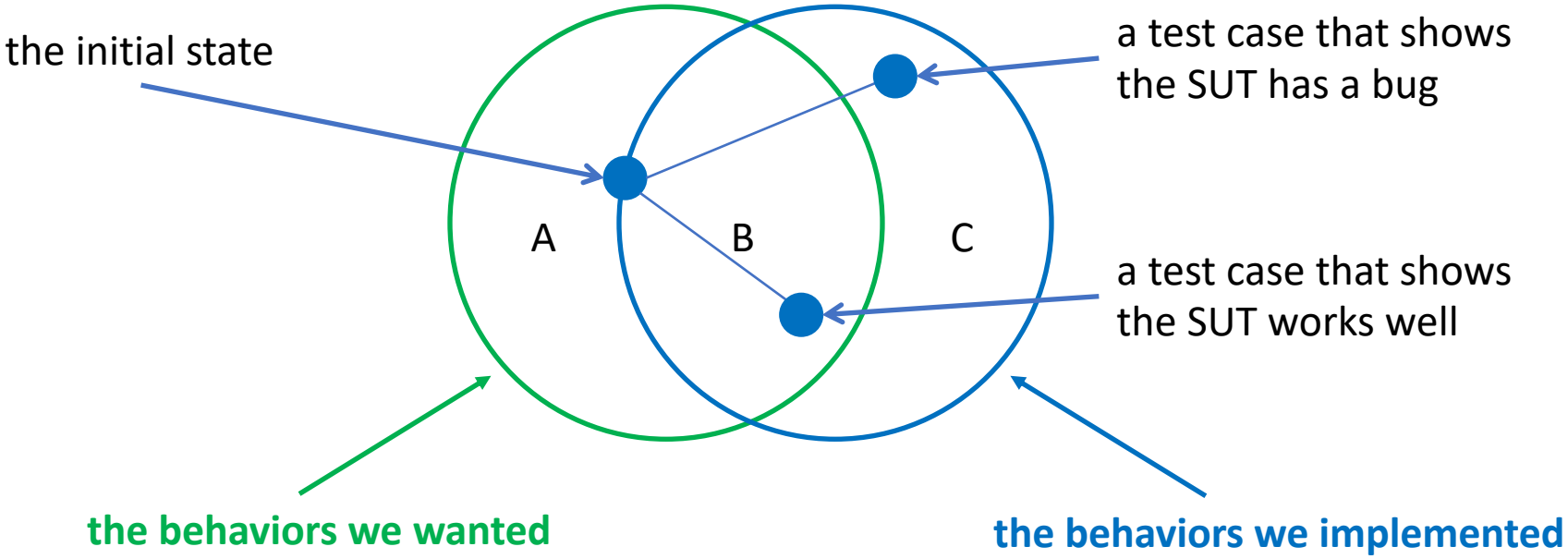
Do you know any accident that was caused by software defects?

- The ticketing system of the Olympic Game in 2008 suffered a performance degradation and was offline.

The ticketing center made an apology to the public.

Software Testing

- Testing is a verification and validation process that executes a **system under test** (SUT) with a set of **test cases** to decide whether the SUT works **correctly** or **unexpectedly**
  - Verification: are we building the system right?
  - Validation: are we building the right system?



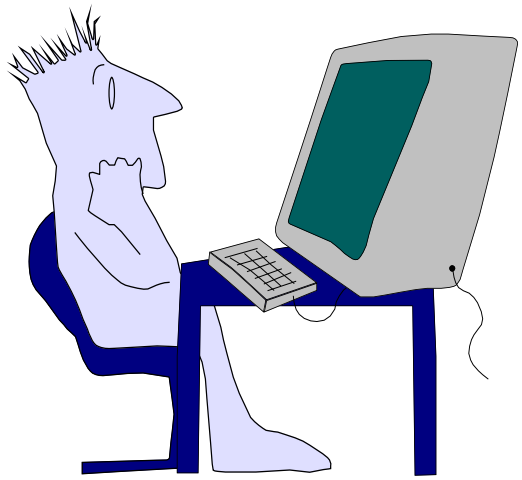
Discussion 2 – Let's Warm Up

Write test cases that can respectively trigger the desired and unexpected behaviors.

```
public int divide (int a, int b) {
    return a/b;
}
```

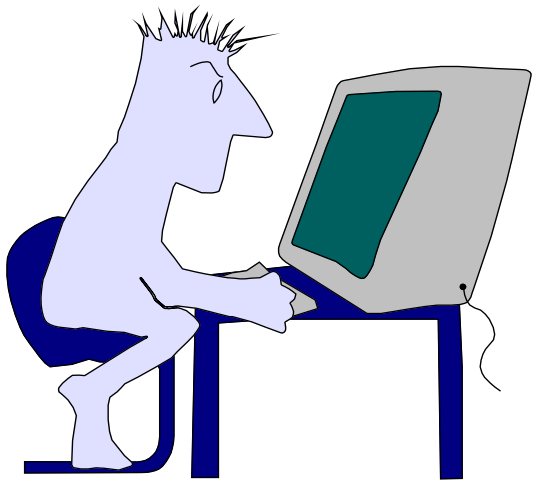
- Test case for the desired behaviour: **a = 8, b = 2**
- Test case for the unexpected behaviour: **a = 8, b = 0**

Who Tests the System?



Developers

- understand the system
- but will test “gently”
- and are driven by “delivery”



Independent Testers

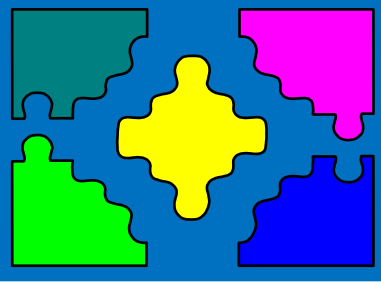
- must learn about the system
- but will attempt to break it
- and are driven by “quality”

Discussion 3 – Testing Classification

- What are the kinds of testing you know?
- Can you try to classify them?

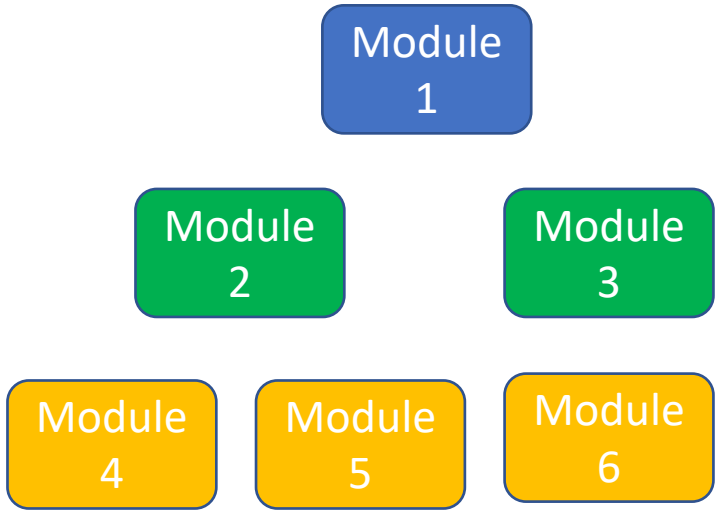
- By the **component level of the SUT**: unit, integration, and system testing
- By the **knowledge of the SUT**: black-box, white-box, and grey-box testing
- By the **object of testing**: functional, **security**, usability, **performance**, compatibility testing, etc.
- By the **type of the SUT**: **mobile**, protocol, **compiler**, smart contract, web testing, etc.
- ……

Unit Testing



- Test individual units/components/modules of a system
  - A unit is often an entire **interface**, but can be an individual **method**
  - Ensure the correctness of a unit, or find bugs in a unit

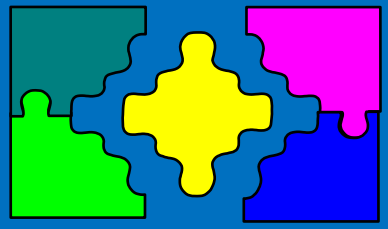
- Advantages
  - Find problems early
  - Facilitate changes
  - Simplify integration
  - Provide live documentation



Unit Testing (cont.)

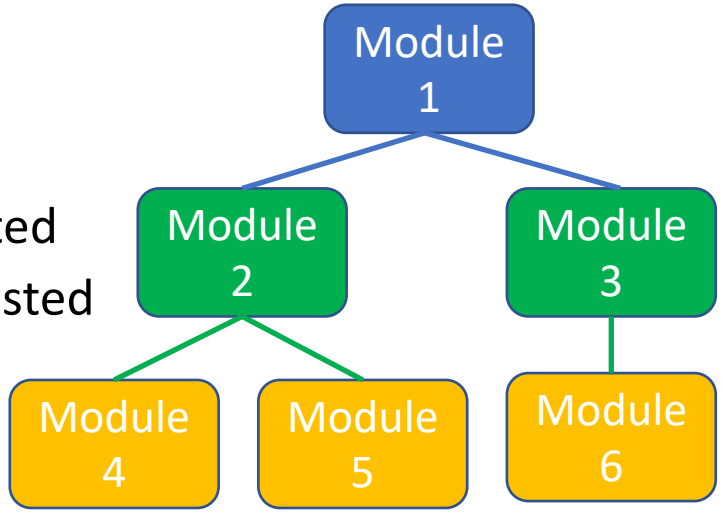
```
public class C1 {
    // assume that both a and b
    // are positive integers
    public int divide (int a, int b) {
        return a/b;
    }
}

@Test
public void testDivide() {
      C1 c1 = new C1();
      assert(c1.divide(8, 2) == 4);
}
```

Integration Testing



- Combine individual units/components/modules of a system, and test them as a group
  - Expose problems in the interface interactions among modules, often caused by **inconsistent assumptions** about interfaces

- Typical approaches
  - Big-bang: difficult to locate bugs
  - Top-Down: lower modules less tested
  - Bottom-Up: higher modules less tested



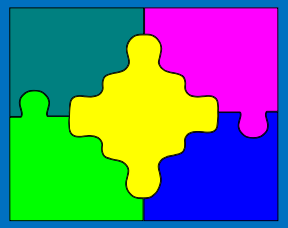
Integration Testing (cont.)

```
public class C1 {
    // assume that both a and b
    // are positive integers
    public int divide (int a, int b) {
        return a/b;
    }
}
```

```
public class C2 {
    // both a and b can be any integers
    public int calculate (int a, int b, int c) {
        C1 c1 = new C1();
        return c1.divide(a, b) * c;
    }
}
```

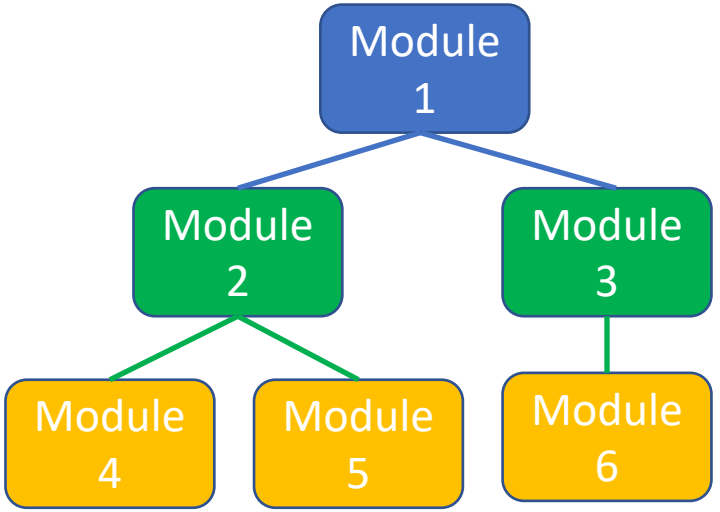
```
@Test
public void testCalculate() {
    C2 c2 = new C2();
    assert(c2.calculate(8, 0, 0) == 0);
}
```

System Testing



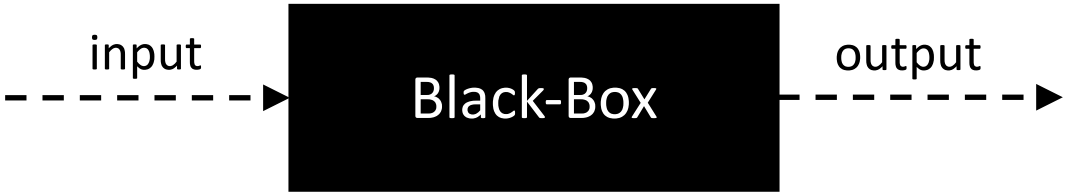
- Test a complete, integrated system to evaluate the system's compliance with its specified requirements

- Approaches
  - Functional testing
  - Security testing
  - Performance testing
  - ……



Black-Box Testing

- Test a system without any knowledge about the code and its internal structure
  - Testers are aware of what the system is supposed to do but are not aware of how it does it

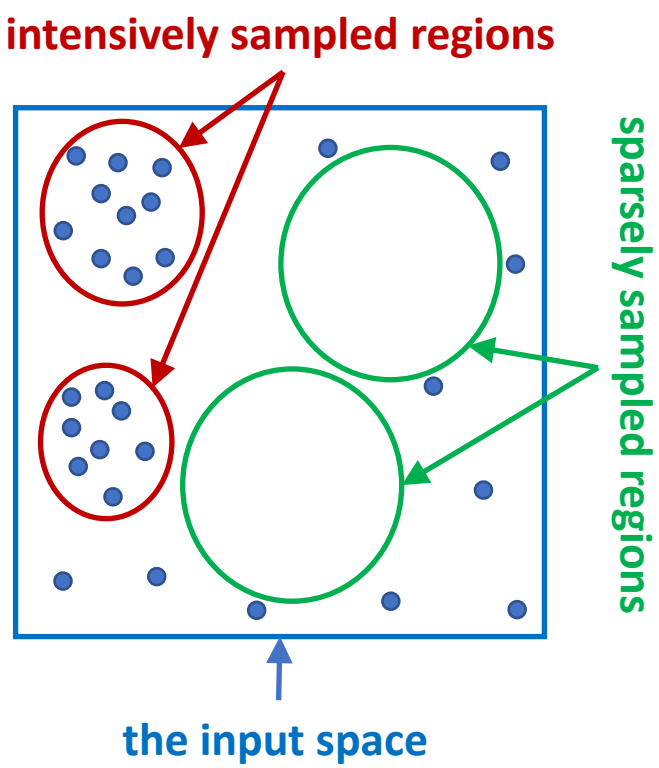


Discussion 4 – Triangle Problem

How do you generate black-box test cases for the triangle problem?

- The program reads **three positive integers** as inputs. These integers represent the lengths of the three sides of a triangle.
- The purpose of this program is to display a message which states whether the triangle is **scalene** (i.e., no two sides are equal), **isosceles** (i.e., two sides are equal) or **equilateral** (i.e., all three sides are equal).

Black-Box Testing – Random Testing



- Test a system by generating random, independent inputs
  - Efficient, and cheap to use
  - Ununiform sampling
  - Redundant test cases
  - Hard to cover corner cases

Black-Box Testing – Equivalence Partitioning

- Divide the input space of a system into partitions of (**valid** and **invalid**) equivalent classes according to the input conditions, and then derive test cases from the partitions

Input Condition	Valid Equivalent Classes	Invalid Equivalent Classes
Three sides are positive integers	(1) a > 0 (2) b > 0 (3) c > 0	(4) a <= 0 (5) b <= 0 (6) c <= 0
Scalene triangles	(7) a < b + c (8) b < a + c (9) c < a + b	(10) a >= b + c (11) b >= a + c (12) c >= a + b
Isosceles triangles	(13) a = b, a != c (14) a = c, a != b (15) b = c, b != a	
Equilateral triangles	(16) a = b = c	

Black-Box Testing – Equivalence Partitioning (cont.)

- Divide the input space of a system into partitions of (**valid** and **invalid**) equivalent classes according to the input conditions, and then derive test cases from the partitions

Test Input	Expected Output	Covered Equivalent Classes
a = 50, b = 60, c = 70	Scalene triangle	(1) (2) (3) (7) (8) (9)
a = 50, b = 50, c = 50	Equilateral triangle	(16)
a = 50, b = 50, c = 10	Isosceles triangle	(13)
a = 50, b = 10, c = 50	Isosceles triangle	(14)
a = 10, b = 50, c = 50	Isosceles triangle	(15)
a = 0, b = 0, c = 0	Invalid input	(4) (5) (6)
a = 80, b = 30, c = 40	Invalid input	(10)
a = 30, b = 80, c = 40	Invalid input	(11)
a = 30, b = 40, c = 80	Invalid input	(12)

Discussion 5 – Let's Fight!

Compute the equivalent classes of the annual leave problem.

- The program takes the number of working years as an input.
- If the employee has worked for more than 1 year but less than 10 years, the annual leave is 5 days; if the employee has worked for more than 10 years but less than 20 years, the annual leave is 10 days; and if the employee has worked for more than 20 years, the annual leave is 15 days.

- Valid: (1) 0 <= year < 1, (2) 1 <= year < 10, (3) 10 <= year < 20, (4) year >= 20
- Invalid: (5) year < 0

White-Box Testing

- Test a system with full knowledge about the code and its internal structure (e.g., control flow, data flow, and paths)
  - Testers choose inputs to exercise paths through the code and determine the expected outputs

Discussion 6 – Credit Problem

How do you generate white-box test cases for the shopping credit problem? (hint: code coverage)

```
public int getShoppingCredit (double amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 && vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 10) {
        credit += 5;
    }
    return credit;
}
```

White-Box Testing – Coverage Testing

- **Statement coverage**: design a set of test cases that can execute each statement for at least once

amount = 350, vip = 1, quantity = 12; expected output: 40

```
public int getShoppingCredit (double amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 || vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 10) {
        credit += 5;
    }
    return credit;
}
```

the test case fails to detect this bug

White-Box Testing – Coverage Testing

- **Branch coverage**: design a set of test cases that can execute the true and false branch of each conditional for at least once

amount = 350, vip = 1, quantity = 12; expected output: 40

amount = 150, vip = 0, quantity = 7; expected output: 0

```
public int getShoppingCredit (double amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 && vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 8) {      the test case fails to detect this bug
        credit += 5;
    }
    return credit;
}
```

White-Box Testing – Coverage Testing

- **Path coverage**: design a set of test cases that can execute each program execution path for at least once

amount = 350, vip = 1, quantity = 12; expected output: 40
amount = 150, vip = 0, quantity = 7; expected output: 0
amount = 300, vip = 1, quantity = 7; expected output: 30
amount = 150, vip = 1, quantity = 12; expected output: 5

```
public int getShoppingCredit (double amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 && vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 10) {
        credit += 5;
    }
    return credit;
}
```

Discussion 7 – Let's Fight!

Write test cases that satisfy statement coverage, branch coverage and path coverage, respectively.

```
public int exercise(int a, int b, double c) {
    if (a > 0 && b > 0) {
        c = c / a;
    }
    if (a > 2 || c > 1) {
        c = c + 1;
    }
    return b + c;
}
```

White-Box Testing – Mutation Testing

- Mutate (change) certain statements in the source code to introduce potential bugs, and check whether the test cases are able to find the introduced bugs
- **Mutation operators**
  - Replace true with false; replace + with -, *, /; replace > with >=, ==, <=, <; ……
- **Mutant**
  - The program after applying the mutation operators
- **A test case kills a mutant**
  - The output of the original program is different from the output of the mutant after running a test case

White-Box Testing – Mutation Testing

The Original Program

```
public int getShoppingCredit (double
  amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 && vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 10) {
        credit += 5;
    }
    return credit;
}
```

The Mutant

```
public int getShoppingCredit (double
  amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 || vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 10) {
        credit += 5;
    }
    return credit;
}
```

The test case "amount = 350, vip = 1, quantity = 12" fails to kill the mutant

Discussion 8 – Let's fight!

Write a test case that can kill the mutant.

The Original Program

```
public int getShoppingCredit (double
  amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 && vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 10) {
        credit += 5;
    }
    return credit;
}
```

The Mutant

```
public int getShoppingCredit (double
  amount, boolean vip, int quantity) {
    int credit = 0;
    if (amount >= 200 || vip) {
        credit = (int) (amount * 0.1);
    }
    if (amount >= 400 || quantity > 10) {
        credit += 5;
    }
    return credit;
}
```

Grey-Box Testing

- Test a system with partial knowledge about the code and its internal structure

Challenges

- Theoretically, testing can never be complete
  - Tradeoff between completeness and cost
- Automatic test generation

Q&A?

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io