

---

# **Chapter 8 - Software Testing**

**MI Qing (Lecturer)**

Telephone: 15210503242

Email: [miqing@bjut.edu.cn](mailto:miqing@bjut.edu.cn)

Office: 410, Information Building

# Topics Covered

---

- ✧ Overview of Software Testing
- ✧ Static Testing VS Dynamic testing
- ✧ Black Box Testing VS White Box Testing
- ✧ Manual Testing VS Automated Testing
- ✧ Development Testing VS User Testing
- ✧ Unit Testing VS Integration Testing VS System Testing

---

# Overview of Software Testing

# Do I need to do testing?

---

- ✧ Everyone who develops software does testing.
- ✧ The only question is whether testing is conducted **haphazardly by random trial-and-error** or **systematically with a plan**.



# Skills Required to Become a Software Tester

---

## ✧ Non-Technical Skills

- Analytical skills
- Communication skills
- Time management skills
- GREAT attitude

## ✧ Technical Skills

- Knowledge and hands-on experience of any defect tracking tool
- Knowledge and hands-on experience of automation tool
- Basic knowledge of Database
- Basic knowledge of Linux commands

# Terms

---

**错误 (Error)**：在软件生存期内的不希望或不可接受的人为错误，其结果是导致软件缺陷的产生。

**缺陷 (Defect)**：软件缺陷是存在于软件产品之中的那些不希望或不可接受的偏差，其结果是软件运行于某一特定条件时出现故障。

**故障 (Fault)**：软件运行过程中出现的一种不希望或不可接受的内部状态，若无适当措施（容错）加以及时处理，便产生软件失效。

**失效 (Failure)**：软件运行时产生的一种不希望或不可接受的外部行为结果。

# Terms

举例  
爱国者导弹



计时用系统时钟  
并以整数表达

软件  
系统

缺少防  
范措施

1991年2月，一次拦截失  
利造成28名美国士兵丧生

缺陷

激活

故障

演变

失效

寄存器存储导致误差  
(0.000000095)<sub>10</sub>

$$0.000000095 \times 100\text{h} \times 60 \times 60 \times 10 = 0.34\text{s}$$

# Software Testing Goals

---

- ✧ Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
  - To demonstrate to the developer and the customer that the software meets its requirements.
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification.

## Validation testing

A successful test shows that the system operates as intended.

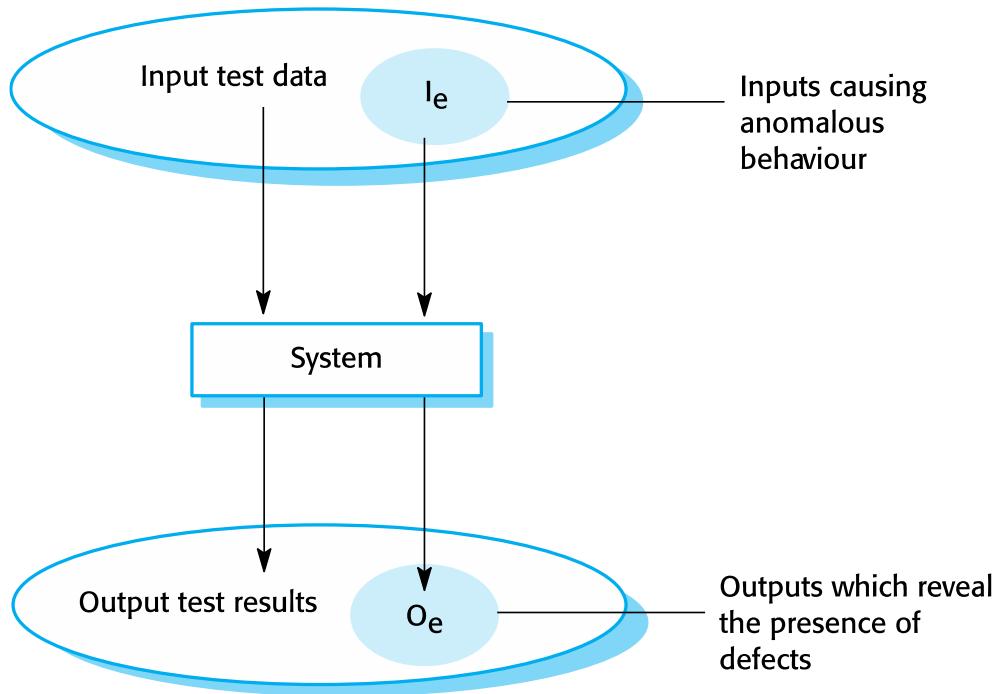
## Defect testing

A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.



# Overview of Software Testing

---



- ✧ A test case is a particular choice of input data to be used in testing a program and the expected output or behavior.
- ✧ A test is a finite collection of test cases.

The goal is to try as many “critical” input combinations as possible, given the time and resource constraints.

# Psychology of Testing

---

- ✧ Human beings tend to be highly goal-oriented, and establishing the proper goal has an important psychological effect on them.

If our goal is to demonstrate that a program has no errors, then we will tend to select test data that have a low probability of causing the program to fail.

**VS**

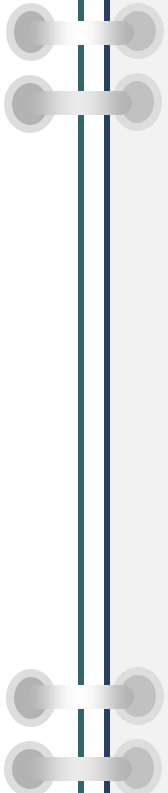
If our goal is to demonstrate that a program has errors, our test data will have a higher probability of finding errors.

*Testing shows the presence, not the absence of bugs.*

Edsger W. Dijkstra, 1972

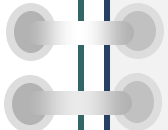
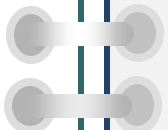
# Software Testing Principles

---

- 
1. A necessary part of a test case is a definition of the expected output or result.
  2. A programmer should avoid attempting to test his or her own program.
  3. A programming organization should not test its own programs.
  4. Any testing process should include a thorough inspection of the results of each test.
  5. Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.

# Software Testing Principles

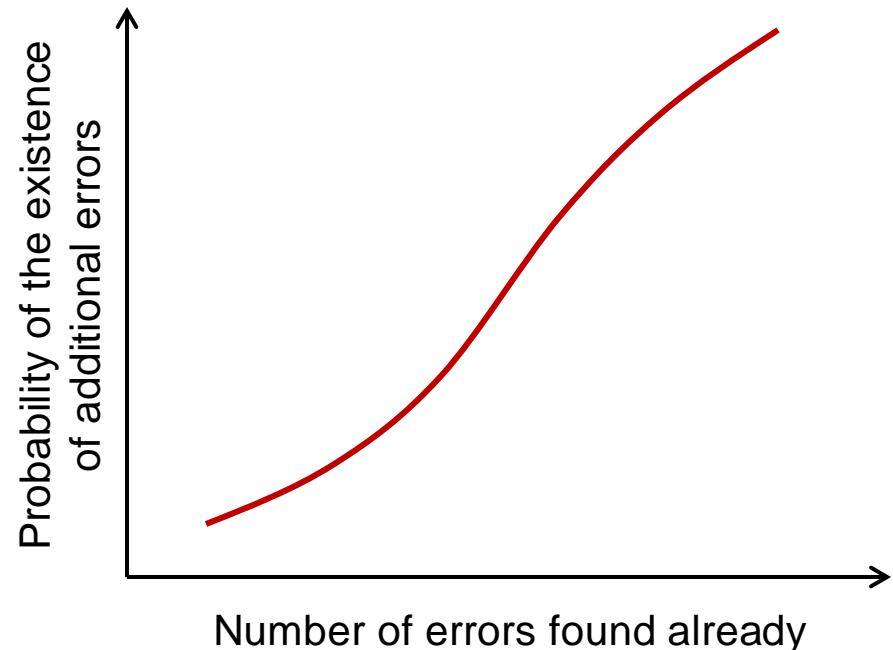
---

- 
- 
6. Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
  7. Avoid throwaway test cases unless the program is truly a throwaway program.
  8. Do not plan a testing effort under the tacit assumption that no errors will be found.
  9. The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
  10. Testing is an extremely creative and intellectually challenging task.

# Defect Clustering

---

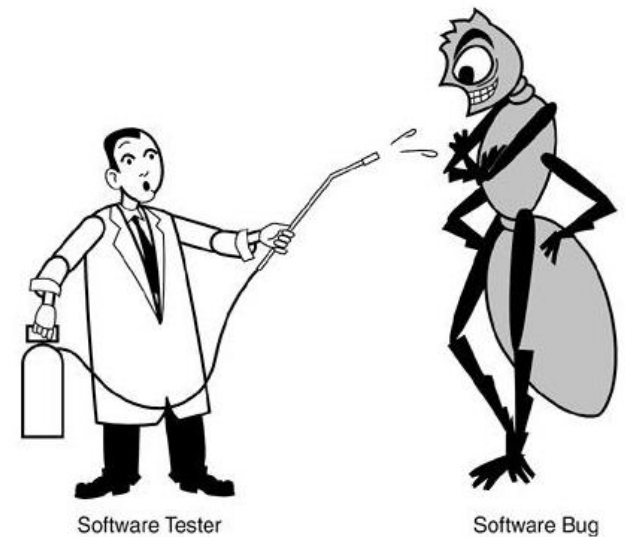
- ✧ Defect clustering means a small number of modules containing most of the defects.
  - Basically, the defects are not distributed uniformly across the entire application.
- ✧ Defect clustering is based on “Pareto Principle”. It means that 80% of the defects found are due to 20% of the modules in the application.



# Pesticide Paradox

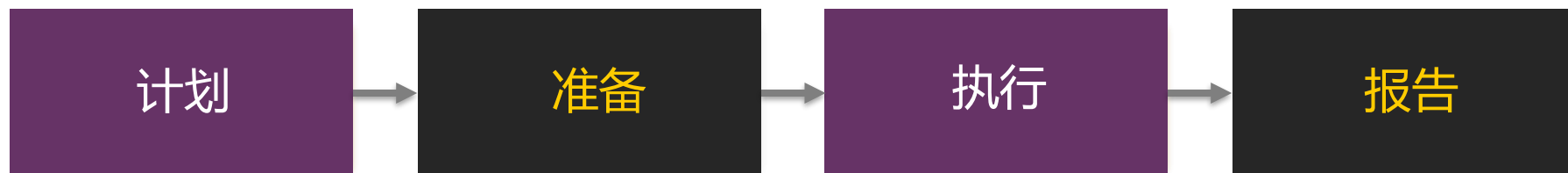
---

- ✧ Pesticide paradox means that if the same set of test cases are executed again and again over the period of time then these set of tests are not capable enough to identify new defects in the system.
- ✧ The set of test cases needs to be regularly reviewed and revised. If required, a new set of test cases can be added and the existing test cases can be deleted if they are not able to find any more defects from the system.



# Process of Software Testing

---



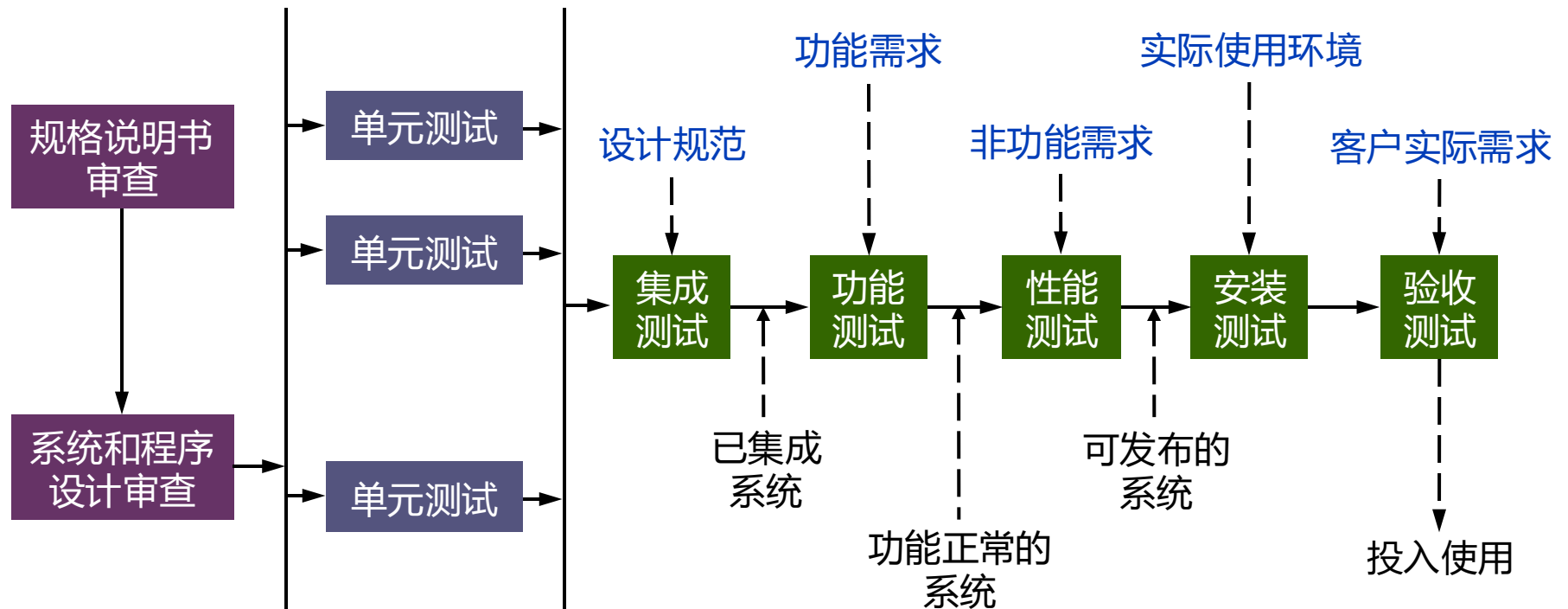
- 识别测试需求
- 分析质量风险
- 拟定测试方案
- 制定测试计划

- 组织测试团队
- 设计测试用例
- 开发工具和脚本
- 准备测试数据

- 获得测试版本
- 执行和实施测试
- 记录测试结果
- 跟踪和管理缺陷

- 分析测试结果
- 评价测试工作
- 提交测试报告

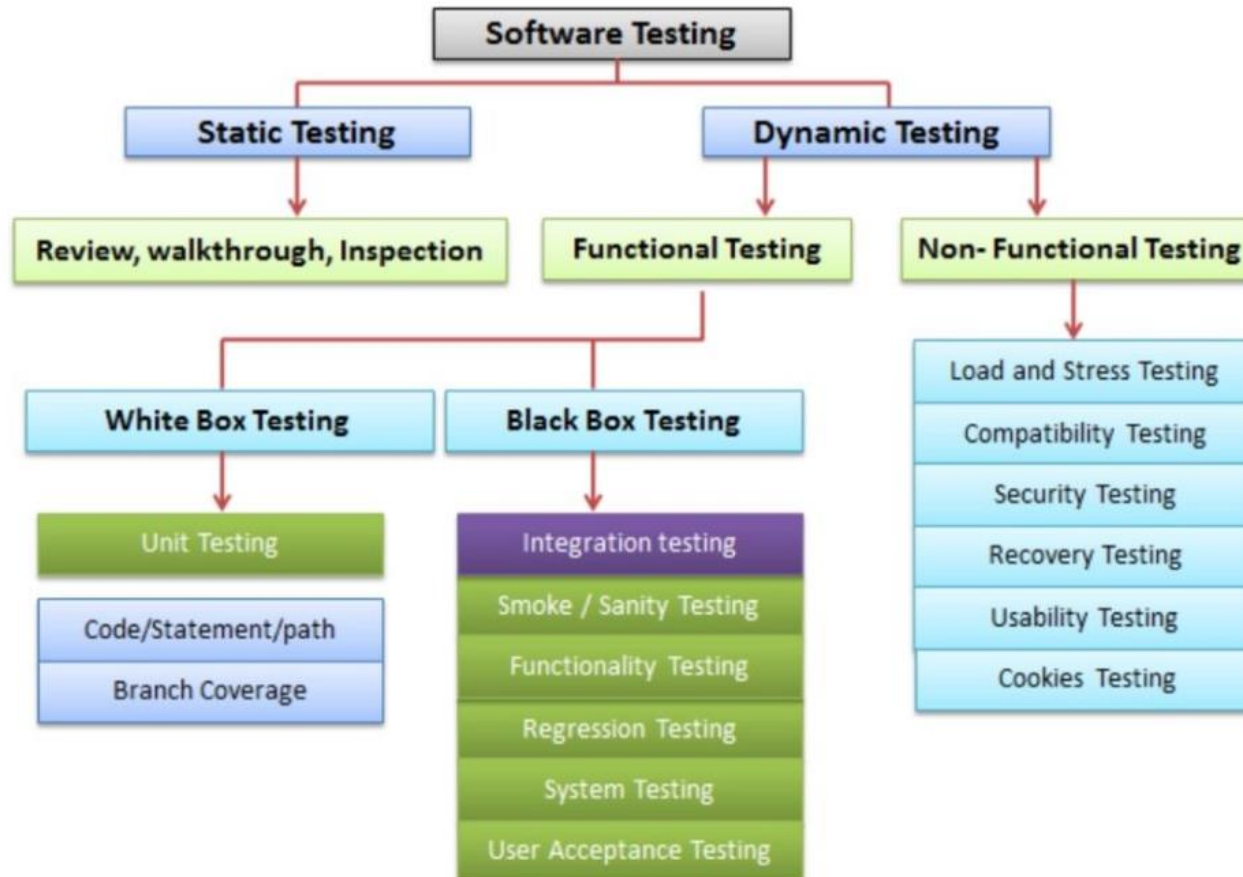
# Phases of Software Testing





# Types of Software Testing

---



---

# Static Testing VS Dynamic testing

# Static Testing

---

## ✧ Walkthroughs

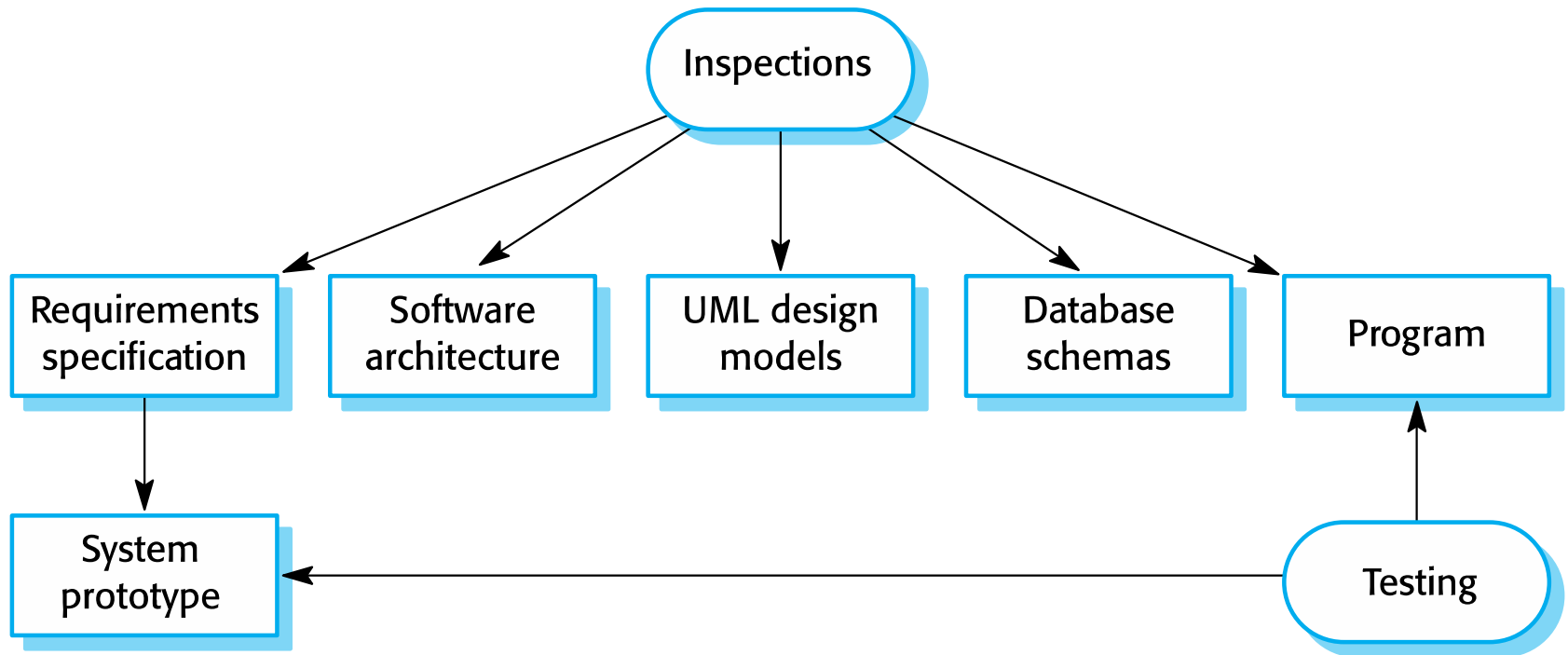
- Walkthrough is a method of conducting informal group/individual review. A small set of paper test cases is prepared. Each test case is mentally executed; that is, the test data are “walked through” the logic of the program.

## ✧ Inspections

- A code inspection is a set of procedures and error-detection techniques for group code reading. These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation. They may be applied to any representation of the system (requirements, design, configuration data, test data).

# Inspections and Testing

---



# Advantages of Inspections

---

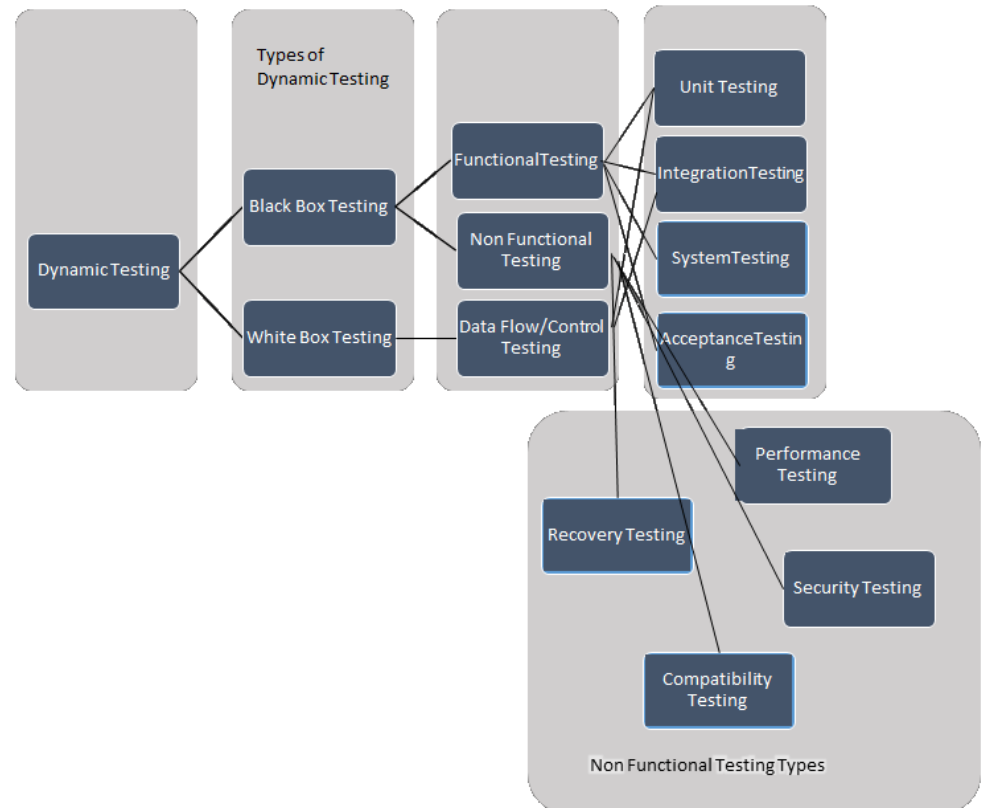
- ✧ During testing, errors can mask (hide) other errors. Because inspection is a static process, **you don't have to be concerned with interactions between errors.**
- ✧ **Incomplete versions of a system can be inspected without additional costs.** If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- ✧ As well as searching for program defects, **an inspection can also consider broader quality attributes** of a program, such as compliance with standards, portability and maintainability.

# Dynamic Testing

✧ Dynamic Testing is a software testing method used to test the **dynamic behavior of software code**.

- White box testing
- Black box testing

✧ The main purpose of dynamic testing is to test software behavior with dynamic variables and finding weak areas in software runtime environment.



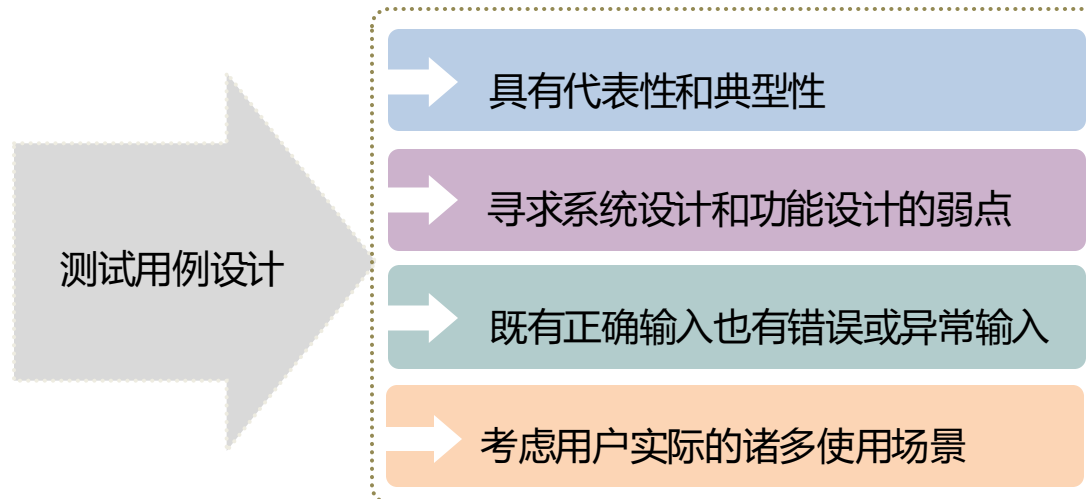
---

# Black Box Testing VS White Box Testing

# Test-Case Design

---

- ✧ Test-case design is so important because complete testing is impossible. The obvious strategy, then, is to try to make tests as complete as possible.
- ✧ Given constraints on time and cost, the key issue of testing becomes: **what subset of all possible test cases has the highest probability of detecting the most errors?**





# Test-Case Design

---

- ✧ The recommended procedure is to **develop test cases using the black box methods and then develop supplementary test cases with white box methods.**

## **White box testing**

Statement coverage  
Decision coverage  
Condition coverage  
Decision/condition coverage  
Multiple-condition coverage

+

## **Black box testing**

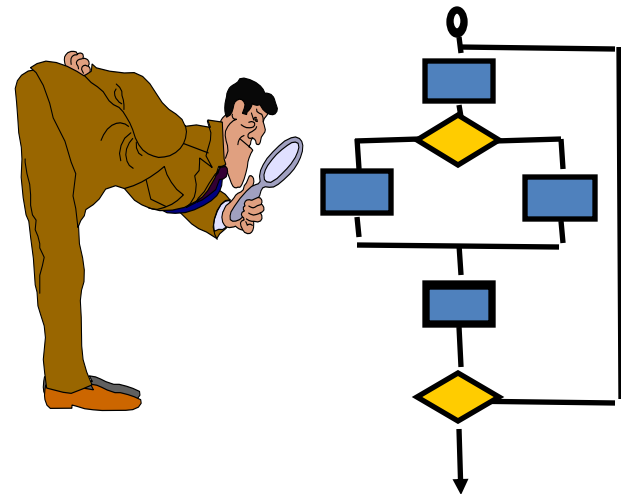
Equivalence partitioning  
Boundary value analysis  
Error guessing

- ✧ It is recommended to use a combination of most, if not all, of them to design a rigorous test of a program, since each method has distinct strengths and weaknesses.

# White Box Testing

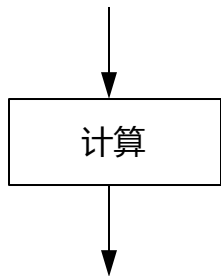
---

- ✧ White box testing is software testing technique in which **internal structure, design and coding of software are tested** to verify flow of input-output and to improve design, usability and security.
  - White box testing is concerned with the degree to which test cases exercise or cover the logic of the program.
- ✧ Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today.

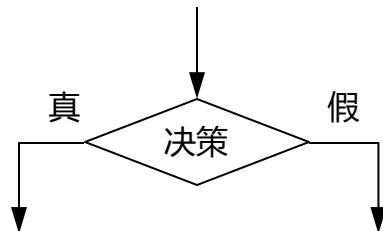


# Control Flow Graph

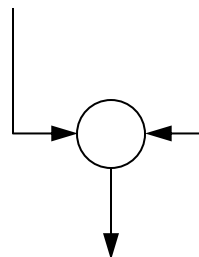
```
public void foo(int A, int B, int X) {  
    if(A > 1 && B == 0) {  
        X = X / A;  
    }  
    if(A == 2 || X > 1) {  
        X = X + 1;  
    }  
}
```



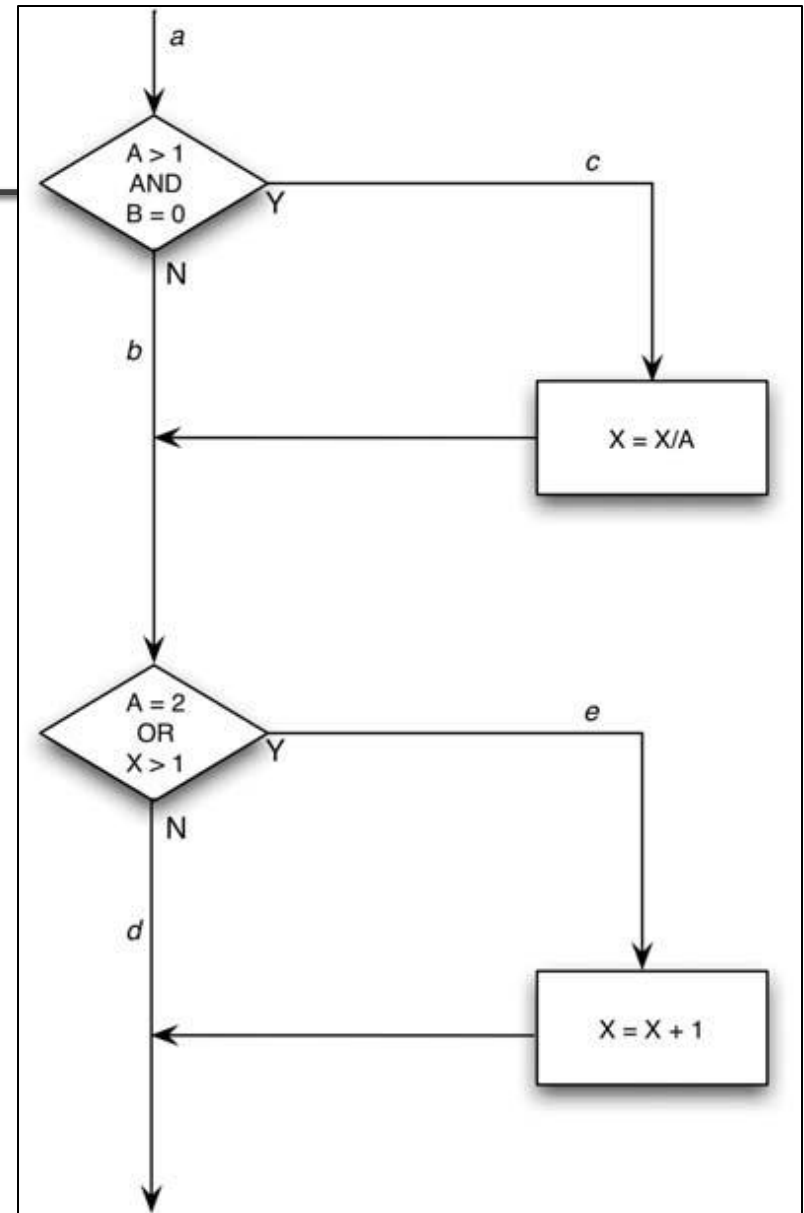
顺序计算



判断节点



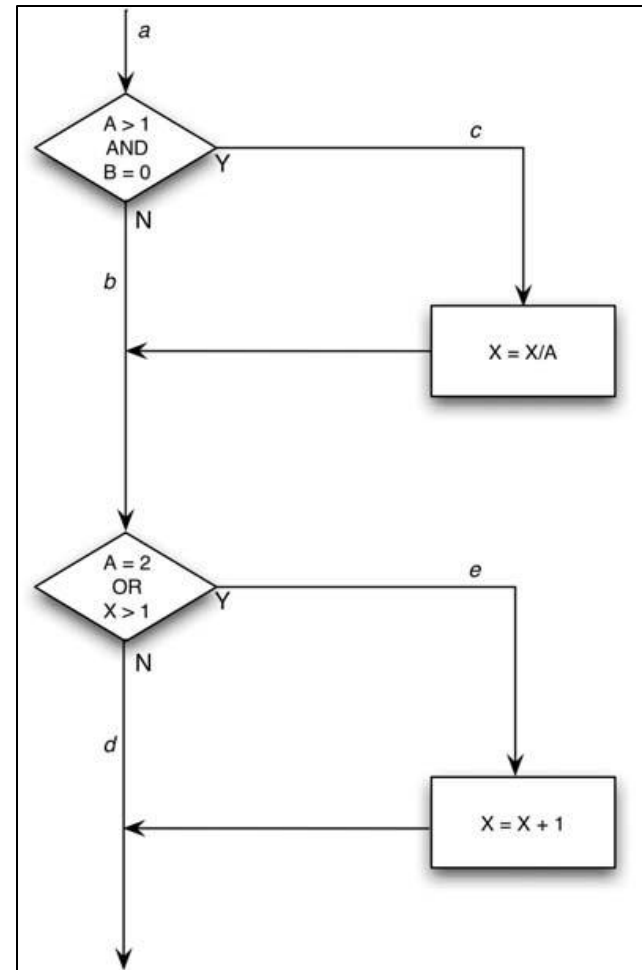
合并节点



# Statement Coverage

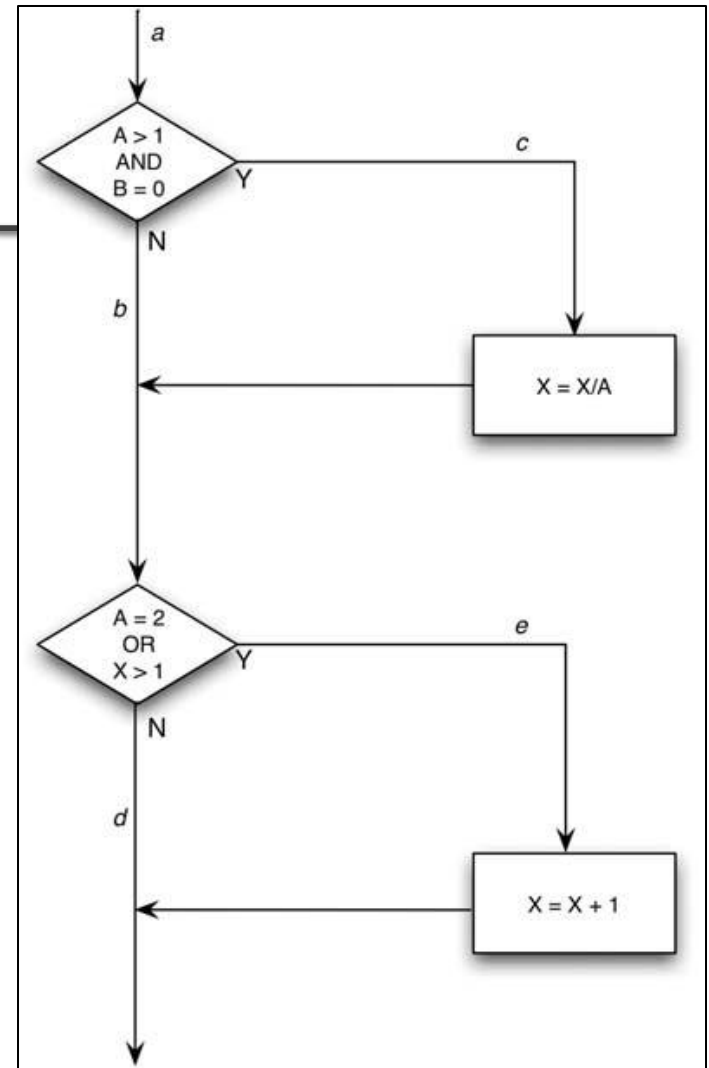
- ✧ This technique requires **every possible statement** in the code to be tested at least once during the testing process.
- ✧ You could execute every statement by writing a single test case that traverses path *ace*.
  - $A=2, B=0, X=3$

The statement coverage criterion is so weak that it generally is useless.



# Decision Coverage

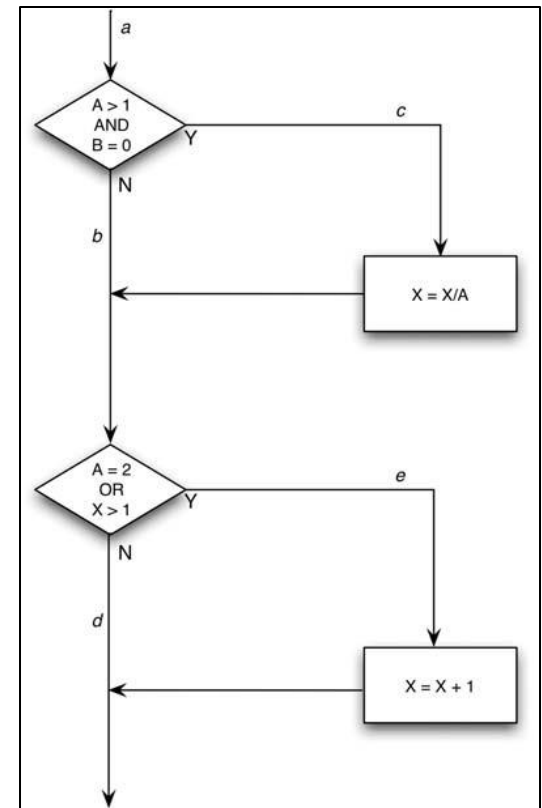
- ✧ This technique checks **every possible path** (if-else and other conditional loops) of a software application.
  - In other words, each branch direction must be traversed at least once.
- ✧ Decision coverage can be met by two test cases covering paths *ace* and *abd* or, *acd* and *abe*.
  - $A=3, B=0, X=3$
  - $A=2, B=1, X=1$



Decision coverage usually can satisfy statement coverage.

# Condition Coverage

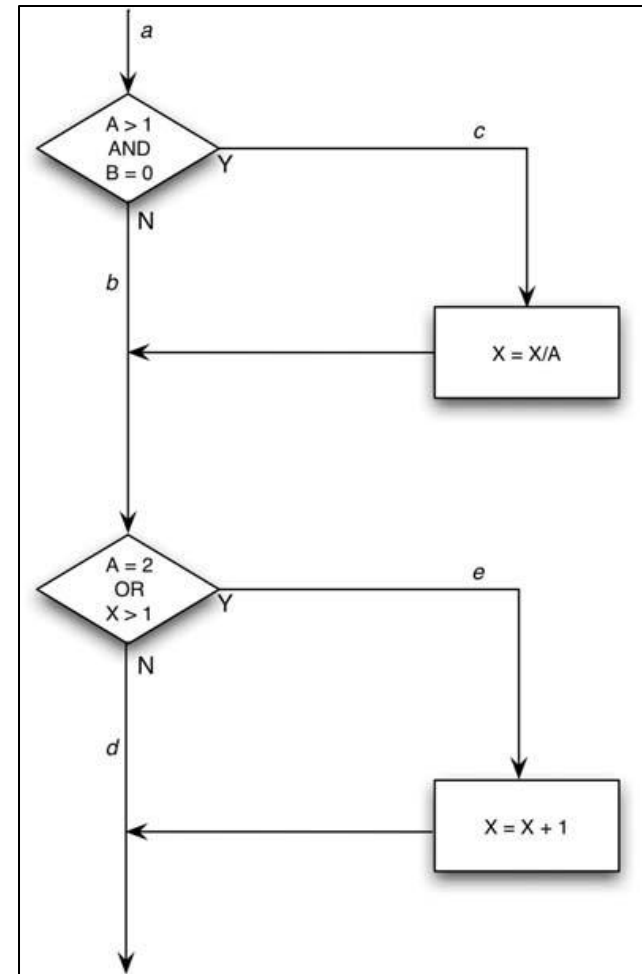
- ✧ In this case, you write enough test cases to ensure that **each condition in a decision** takes on all possible outcomes at least once.
- ✧ There are four conditions:  $A > 1$ ,  $B = 0$ ,  $A = 2$ , and  $X > 1$ . Hence, enough test cases are needed to force the situations where  $A > 1$ ,  $A \leq 1$ ,  $B = 0$ , and  $B \neq 0$  are present at point a and where  $A = 2$ ,  $A \neq 2$ ,  $X > 1$ , and  $X \leq 1$  are present at point b.
  - $A = 1$ ,  $B = 0$ ,  $X = 3$
  - $A = 2$ ,  $B = 1$ ,  $X = 1$



# Decision/Condition Coverage

✧ Decision/condition coverage requires sufficient test cases such that **each condition in a decision** takes on all possible outcomes at least once, and **each decision** takes on all possible outcomes at least once.

- $A=2, B=0, X=4$
- $A=1, B=1, X=1$



# Multiple-Condition Coverage

---

- ✧ This criterion requires that you write sufficient test cases such that **all possible combinations of condition outcomes in each decision** are invoked at least once.

1. $A > 1, B = 0$	5. $A = 2, X > 1$
2. $A > 1, B < > 0$	6. $A = 2, X \leq 1$
3. $A \leq 1, B = 0$	7. $A < > 2, X > 1$
4. $A \leq 1, B < > 0$	8. $A < > 2, X \leq 1$

$A = 2, B = 0, X = 4$       Covers 1, 5

$A = 2, B = 1, X = 1$       Covers 2, 6

$A = 1, B = 0, X = 2$       Covers 3, 7

$A = 1, B = 1, X = 1$       Covers 4, 8

- ✧ A set of test cases satisfying the multiple-condition criterion also satisfies the decision coverage, condition coverage, and decision/condition coverage criteria.



# Advantages and Disadvantages

---

## ✧ Advantages of white box testing:

- Forces test developer to reason carefully about implementation.
- Reveals errors in "hidden" code.
- Spots the **Dead Code** or other issues with respect to best programming practices.

## ✧ Disadvantages of white box testing:

- Expensive as one has to spend both time and money to perform white box testing.
- In-depth knowledge about the programming language is necessary to perform white box testing.

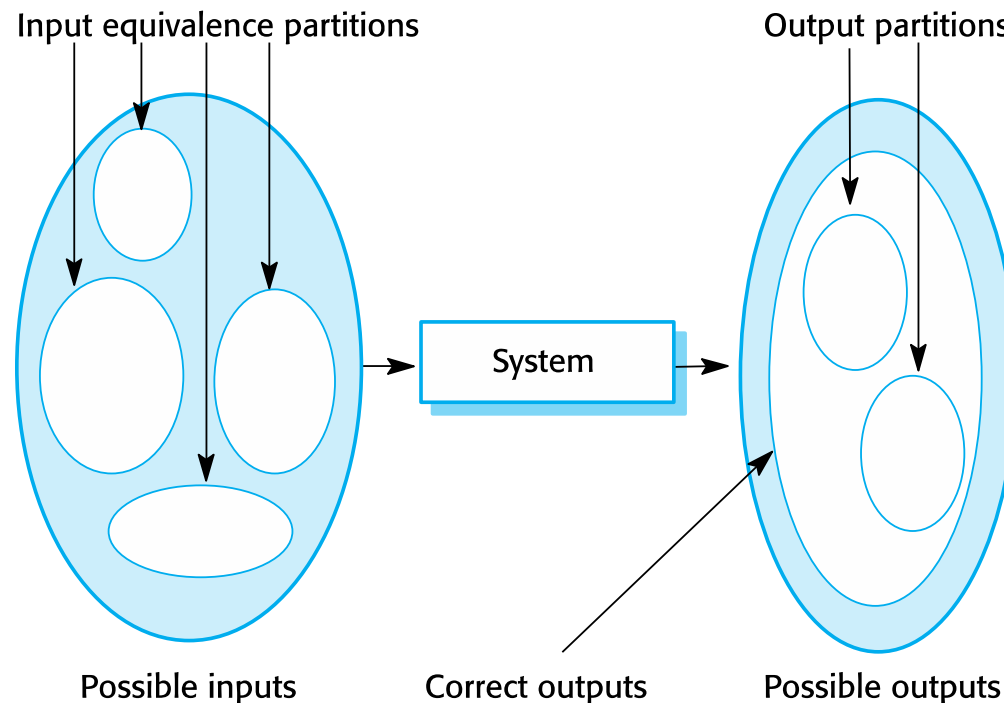
# Black Box Testing

---

- ✧ Black box testing is a high level of testing that focuses on the behavior of the software. It involves testing from an external or end-user perspective.
- ✧ A well-selected test case should have two properties:
  - It reduces, by more than a count of one, the number of other test cases that must be developed to achieve some predefined goal of “reasonable” testing.
  - It covers a large set of other possible test cases. That is, it tells us something about the presence or absence of errors over and above this specific set of input values.

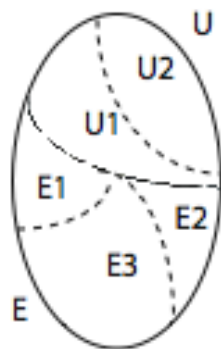
# Equivalence Partitioning

- ✧ Identifying the equivalence classes: identify groups of inputs that have common characteristics and should be processed in the same way.



# Types of Equivalence Class

- ✧ **有效等价类**是对规格说明有意义、合理的输入数据构成的集合。利用有效等价类，能够检验程序是否实现了规格说明中预先规定的功能和性能。
- ✧ **无效等价类**是对规格说明无意义、不合理的输入数据构成的集合。利用无效等价类，可以发现程序异常处理的情况，检查被测对象的功能和性能的实现是否有不符合规格说明要求的地方。



- E 表示所有正常和合法的输入
- U 表示所有异常和非法的输入

# Equivalence Partitioning

---

- ✧ Given an input or external condition, identifying the equivalence classes is largely **a heuristic process**.
  - If an input condition specifies **a range of values**, identify one valid equivalence class and two invalid equivalence classes.
  - If an input condition specifies **a set of input values**, and there is reason to believe that the program handles each differently, identify a valid equivalence class for each and one invalid equivalence class.
  - If an input condition specifies **a “must-be” situation**, identify one valid equivalence class and one invalid equivalence class.
- ✧ If there is any reason to believe that the program does not handle elements in an equivalence class identically, split the equivalence class into smaller ones.

# Examples

---

1) 程序的输入参数  $x$  是小于100大于10的整数。

1个有效等价类:  $10 < x < 100$

2个无效等价类:  $x \leq 10$  和  $x \geq 100$

2) 姓名是长度不超过20的非空字符串，且只由字母组成，数字和其他字符都是非法的。

1个有效等价类: 满足了上述所有条件的字符串

3个无效等价类:

- 空字符串
- 长度超过20的字符串
- 包含了数字或其它字符的字符串

# Examples

---

3) 某程序根据不同的学历分别计算岗位工资，其中学历可以是专科、本科、硕士、博士等四种类型。

4个有效等价类：专科、本科、硕士、博士

1个无效等价类：其他学历

```
4) struct student {  
    string name;  
    string course[100];  
    int grade[100];  
}
```

对复合数据类型中的每个元素进行等价类划分，再将这些等价类进行组合，最终形成对软件整个输入域的划分。

# Identifying the Test Cases

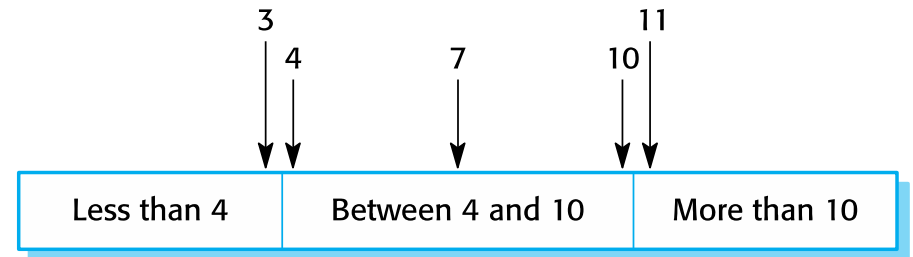
---

- ✧ We then make use equivalence classes to identify the test cases. The process is as follows:
  - Assign a unique number to each equivalence class.
  - Until all valid equivalence classes have been covered by (incorporated into) test cases, write a new test case **covering as many** of the uncovered valid equivalence classes as possible.
  - Until your test cases have covered all invalid equivalence classes, write a test case that **covers one**, and only one, of the uncovered invalid equivalence classes.

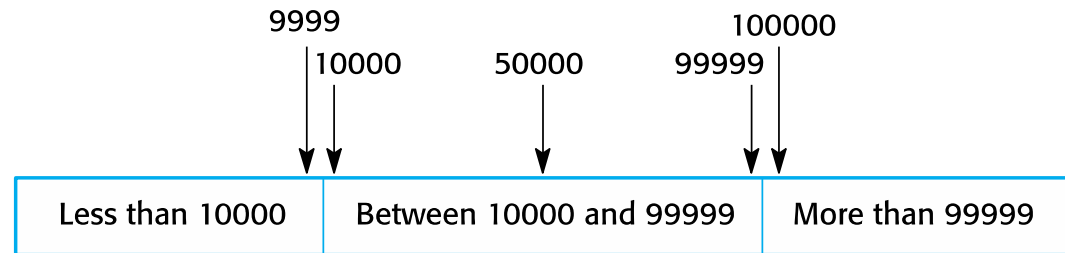


# Boundary Value Analysis

- ✧ Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not.
- ✧ Boundary conditions are those situations directly **on, above, and beneath the edges** of input equivalence classes and output equivalence classes.



Number of input values



Input values

# Error Guessing

---

- ✧ Given a particular program, some people surmise, both by intuition and experience, certain probable types of errors and then write test cases to expose those errors.
  - The basic idea is to enumerate a list of **possible errors or error-prone situations** and then write test cases based on the list.
  - Another idea is to identify test cases associated with **assumptions that the programmer might have made** when reading the specification.



# Black Box Testing VS White Box Testing

---

Parameter	Black Box testing	White Box testing
<b>Base of Testing</b>	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
<b>Usage</b>	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Testing is best suited for a lower level of testing like Unit Testing, Integration testing.
<b>Programming knowledge</b>	Programming knowledge is not needed to perform Black Box testing.	Programming knowledge is required to perform White Box testing.
<b>Tested by</b>	Performed by the end user, developer, and tester.	Usually done by tester and developer.
<b>Time</b>	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.

---

# Manual Testing VS Automated Testing

# Manual Testing VS Automated Testing

---

- ✧ In practice, the testing process usually involves a mixture of manual and automated testing.
  - In **manual testing**, a tester runs the program with some test data and compares the results to their expectations. They note and report discrepancies to the program developers.
  - In **automated testing**, the tests are encoded in a program that is run each time the system under development is to be tested.
- ✧ Unfortunately, testing can never be completely automated as automated tests can only check that a program does what it is supposed to do.

# Manual Testing VS Automated Testing

---

- ✧ 在系统功能逻辑测试、验收测试、适用性测试、涉及物理交互性测试时，多采用黑盒测试的手工测试方法；
- ✧ 单元测试、集成测试、系统负载或性能、稳定性、可靠性测试等比较适合采用自动化测试；
- ✧ 对那种不稳定软件的测试、开发周期很短的软件、一次性的软件等不适合自动化测试；
- ✧ 工具本身并没有想象力和灵活性，一般自动化测试只能发现15~30%的缺陷，而手工测试可以发现70~85%的缺陷；自动化测试工具在进行功能测试时，其准确的含义是回归测试工具。

---

# Development Testing VS User Testing

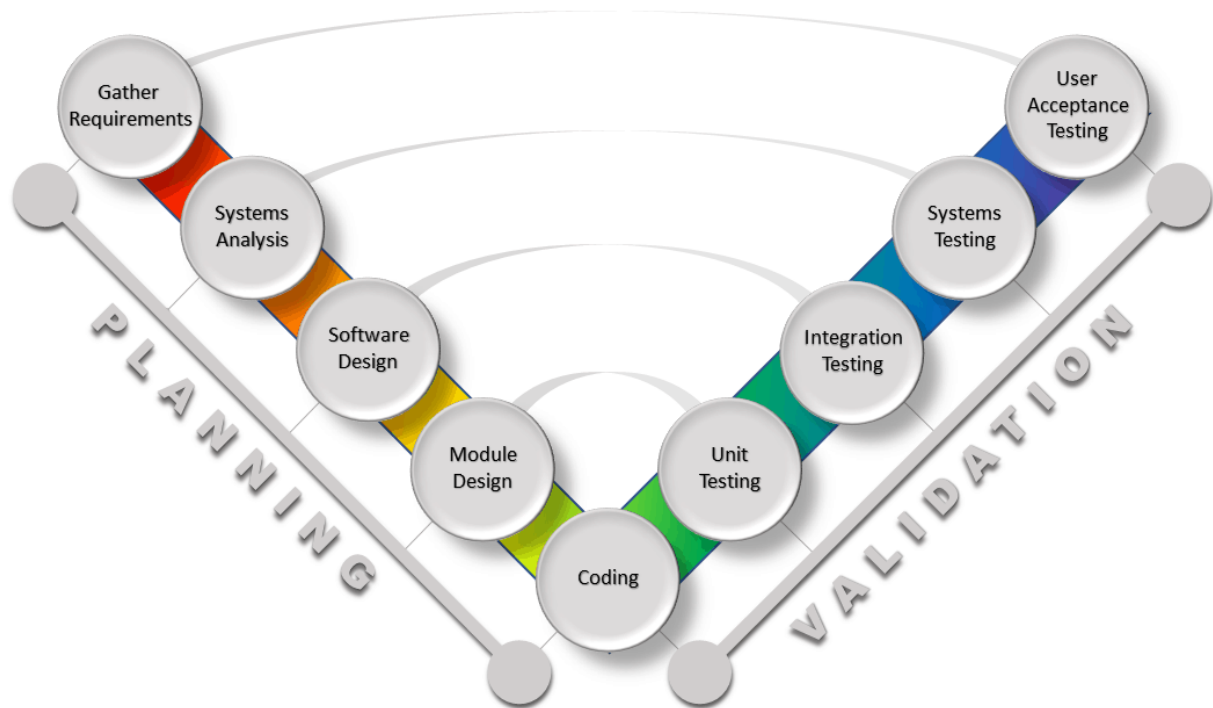
# Development Testing VS User Testing

## ✧ Development testing

- The system is tested **during development** to discover bugs and defects.

## ✧ User testing

- Users or potential users of a system test the system **in their own environment**.





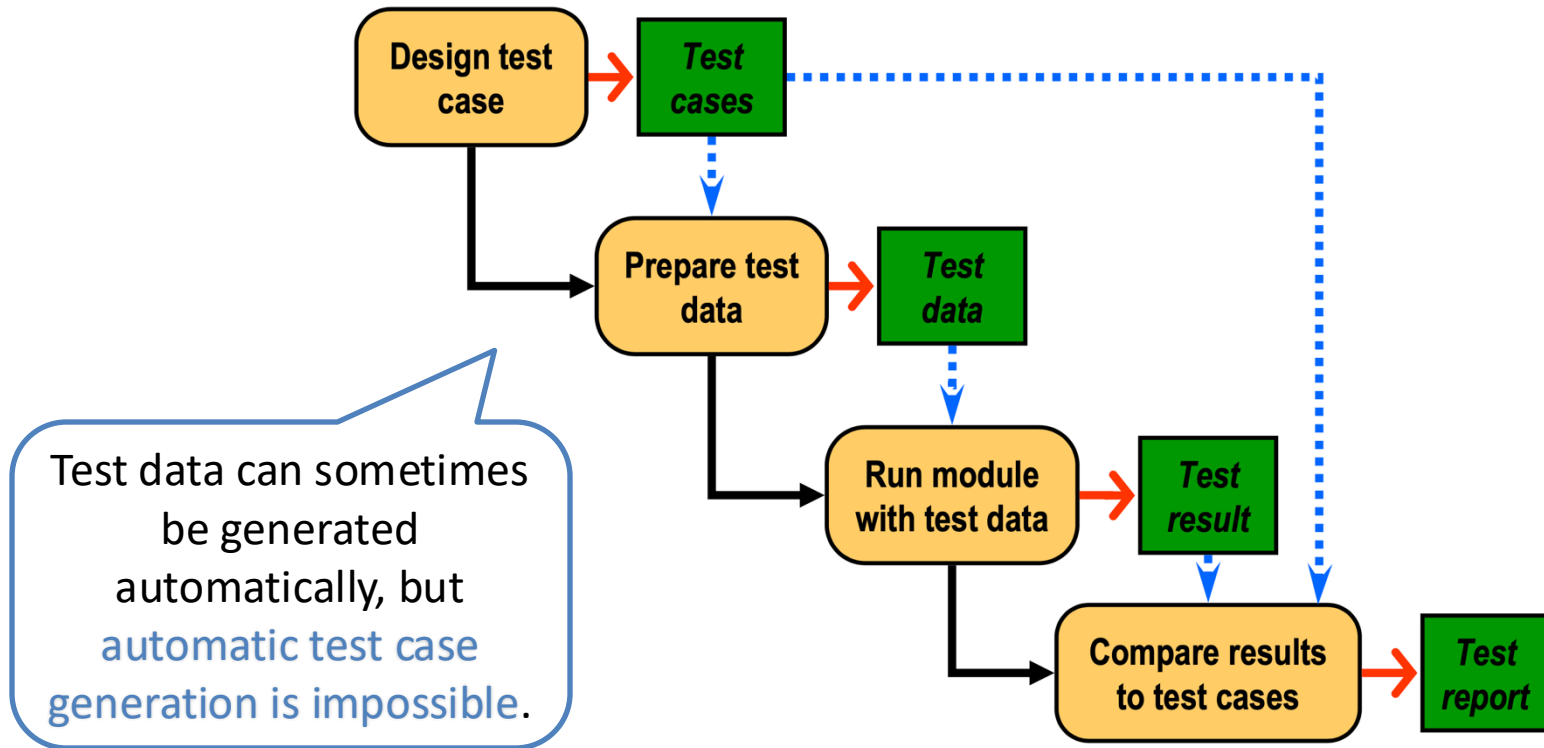
# Development Testing

---

- ✧ Development testing includes all testing activities that are carried out by the team developing the system.
  - **Unit testing**: where individual program units or object classes are tested.
  - **Integration testing**: where individual software modules are combined and tested as a group.
  - **System testing**: where a complete integrated system is tested to evaluate the system's compliance with its specified requirements.

# Unit Testing

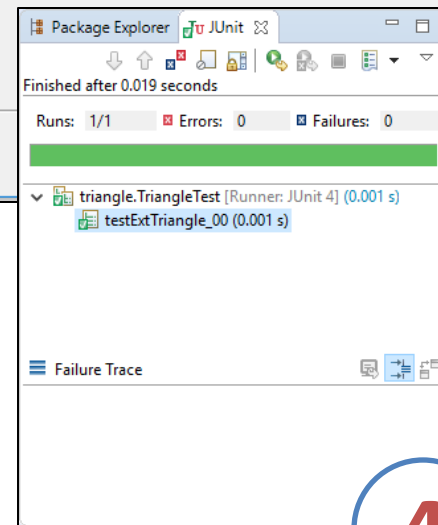
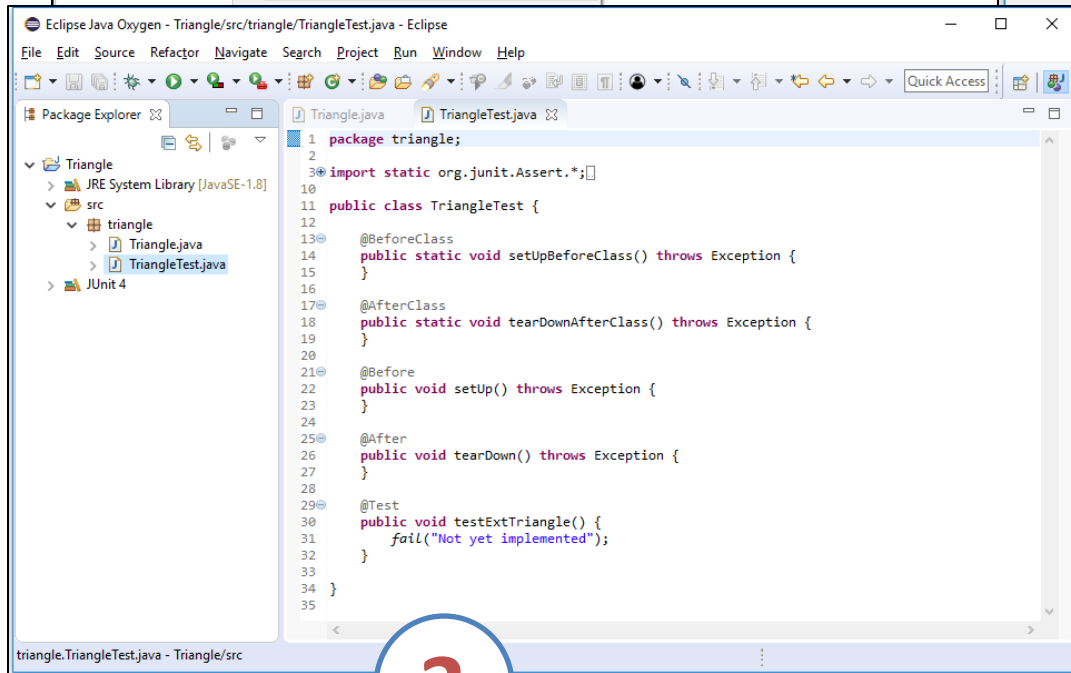
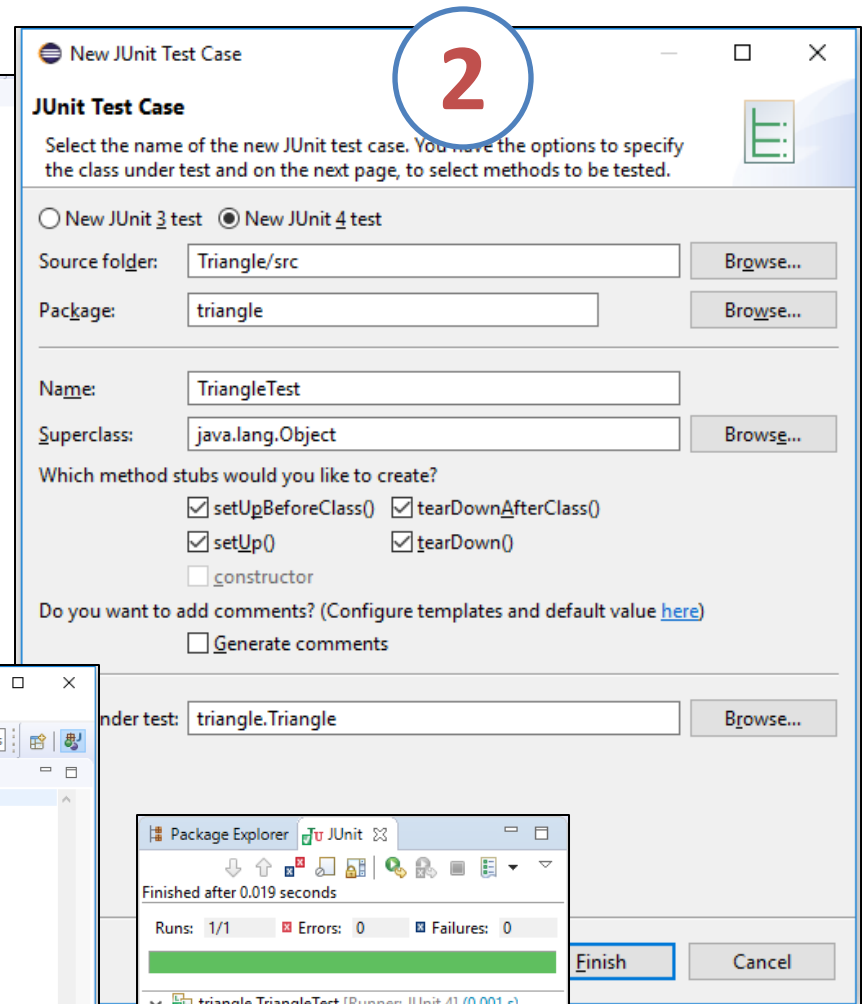
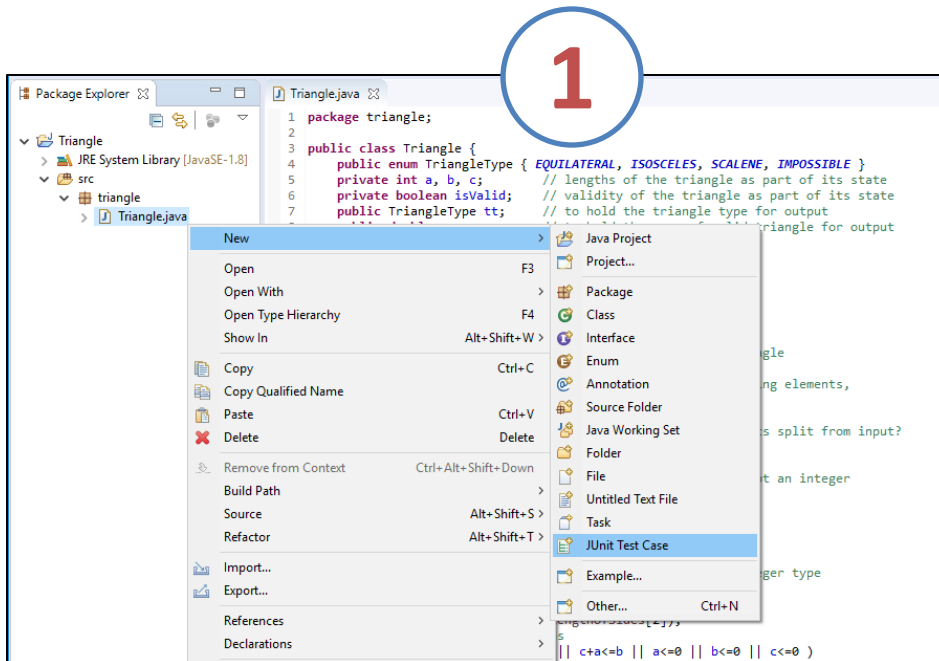
- ✧ Unit testing is the process of testing program components, such as methods or object classes.



# Unit Testing Automation

---

- ✧ Whenever possible, **unit testing should be automated** so that tests are run and checked without manual intervention.
  - In automated unit testing, you make use of a test automation framework (such as **JUnit**) to write and run your program tests.
- ✧ Advantages:
  - Reduce the tedious, routine and repetitive tasks
  - Free tester to do the more creative test case design job
  - Much easier re-use of previous test cases
  - Better management of the test suite (set of test cases)
  - Encourage more frequent testing with more test cases



# Example: Calc and CalcTest

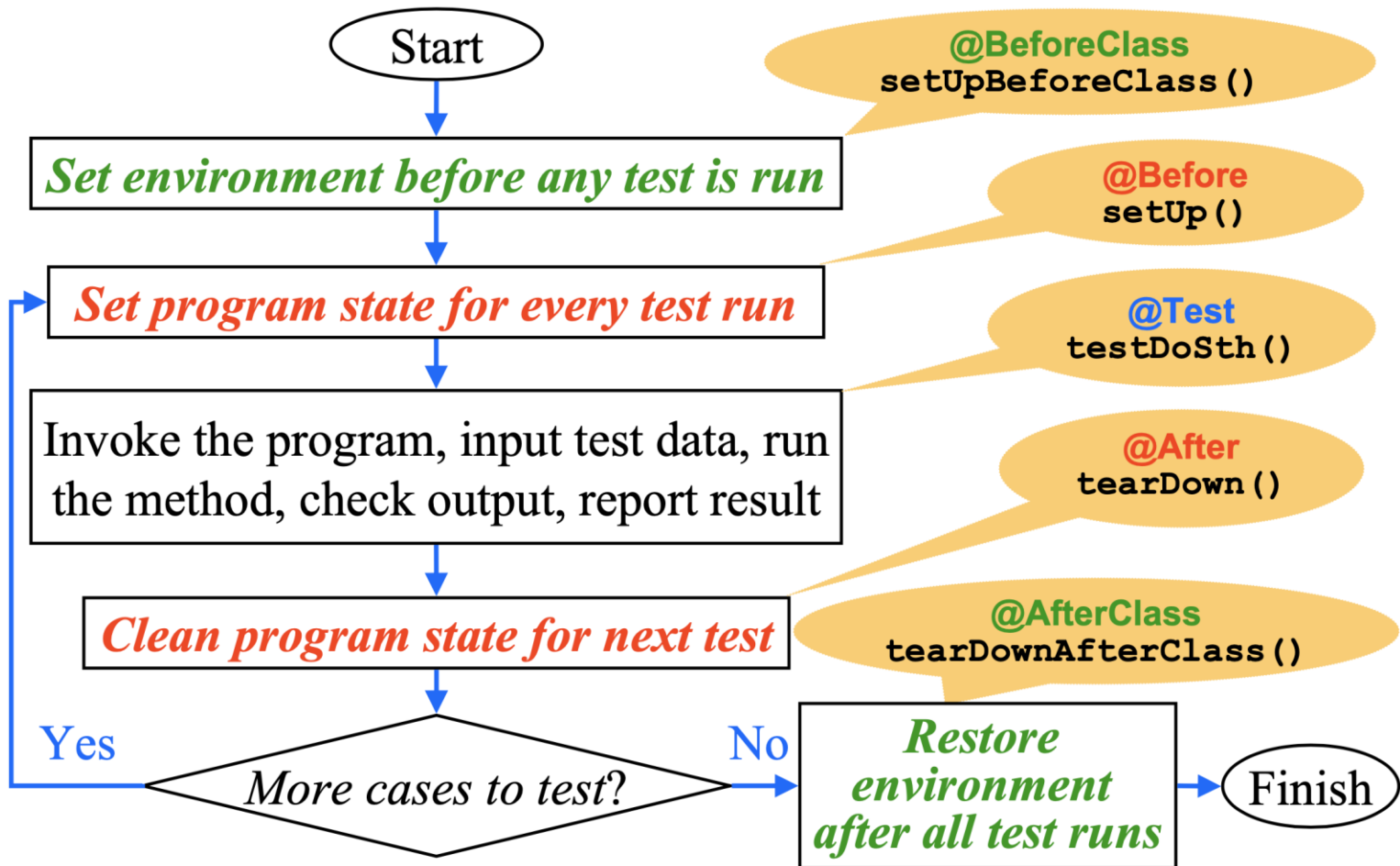
---

## Calc vs CalcTest

- A Java program **Calc** is typically coded in a file, say, **Calc.java**
- Program logic is usually embedded in methods, say, **doSth()**
- To test **Calc**, we need a driver program, say, **CalcTest**, coded in a file, say, **CalcTest.java**
- To test the method **doSth()**, we need a *Test Method*, say, **testDoSth()**



# Logic for Executing JUnit Tests



# Calc.java

---

```
package calc;

public class Calc {

    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }

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

}
```

# CalcTest.java

---

```
package calc;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class CalcTest {

    private Calc c;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
        c = new Calc();
    }

    @After
    public void tearDown() throws Exception {
    }
}
```

- To test the Calc class, we may first generate a **template class** called CalcTest by using Eclipse.
- The generated test class and methods will be automatically marked with **selected special annotations** defined in the JUnit package which is imported from the org.junit namespace.



# CalcTest.java

---

```
@Test
public void testAdd() {
    int result = c.add(10, 2);
    assertEquals("Testing add method", 12, result, 0.0001);
}

@Test
public void testSubtract() {
    int result = c.subtract(10, 2);
    assertEquals("Testing subtract method", 8, result, 0.0001);
}

@Test
public void testMultiply() {
    int result = c.multiply(10, 2);
    assertEquals("Testing multiply method", 20, result, 0.0001);
}

@Test
public void testDivide() {
    int result = c.divide(10, 2);
    assertEquals("Testing divide method", 5, result, 0.0001);
}
```

- The code of the CalcTest class generated by Eclipse is only a template. It has to be **manually completed by filling in specific code** to test the Calc class.

<https://junit.org/junit4/javadoc/latest/org/junit/Assert.html>

# xUnit: A Family of Unit Test Tools

---

- ✧ JUnit is the first generic framework designed to support automation of unit testing of Java programs.
- ✧ Soon afterward, JUnit was ported to support many other languages:
  - CUnit (C): <http://cunit.sourceforge.net/>
  - VUnit (Visual Basic): <https://simplyvbunit.sourceforge.net/>
  - NUnit (all .Net languages): <https://nunit.org/>
  - SUnit (Smalltalk), CppUnit (C++), runit (Ruby) and PyUnit (Python)

# Do Unit Testing First

---

## ✧ Advantages:

- Search the bugs at where they may appear

- When you test a system of many modules and find a bug, it is usually not easy to pinpoint in which module the bug is.

- Remove the bugs as early as possible

- The latter the bugs are removed, the more difficult and costly to do so.

- Introduce parallelism into the program testing process

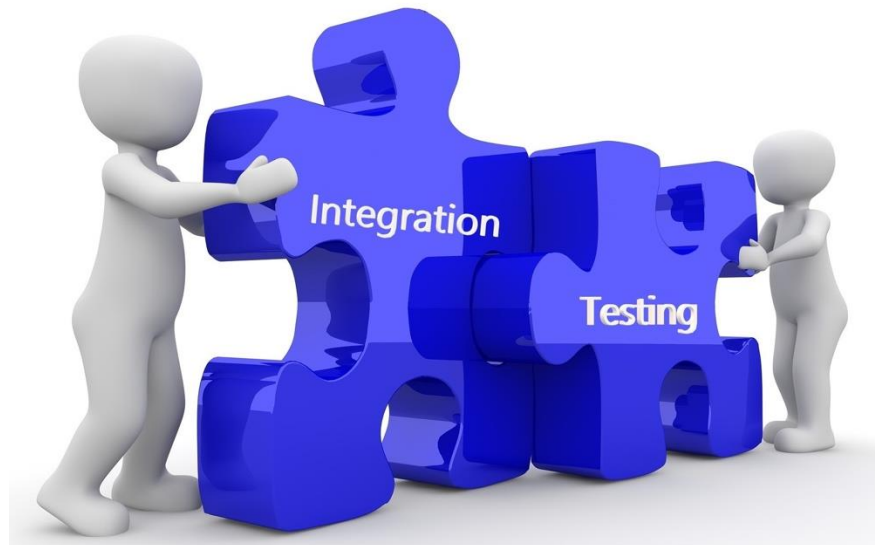
- It is possible to test multiple modules simultaneously.

✧ But many people violate these principles, skip unit and even integration testing, and jump to system testing.

# Integration Testing

---

- ✧ Integration testing is a level of software testing where individual units are combined and tested to verify if they are working as they intend to.
  - The purpose is to **expose defects in the interaction between software modules when they are integrated.**



# Interface Errors

---

## ✧ Interface misuse

- A calling component calls another component and makes an error in its use of its interface.

## ✧ Interface misunderstanding

- A calling component embeds assumptions about the behaviour of the called component which are incorrect.

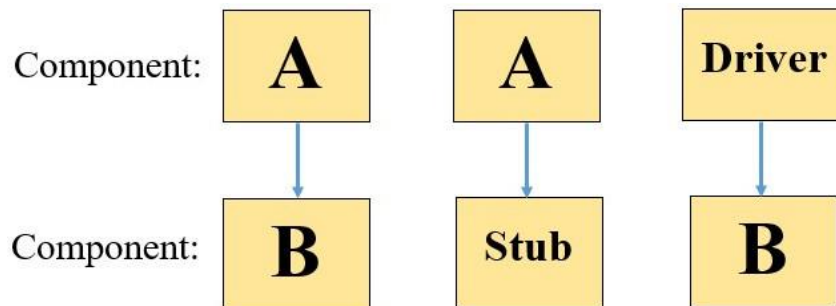
## ✧ Timing errors

- The called and the calling component operate at different speeds and out-of-date information is accessed.

# Interface Testing Guidelines

✧ Some general guidelines for interface testing are:

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.



- **Stub**: A stub is called from the software component to be tested.
- **Driver**: A driver calls the component to be tested.

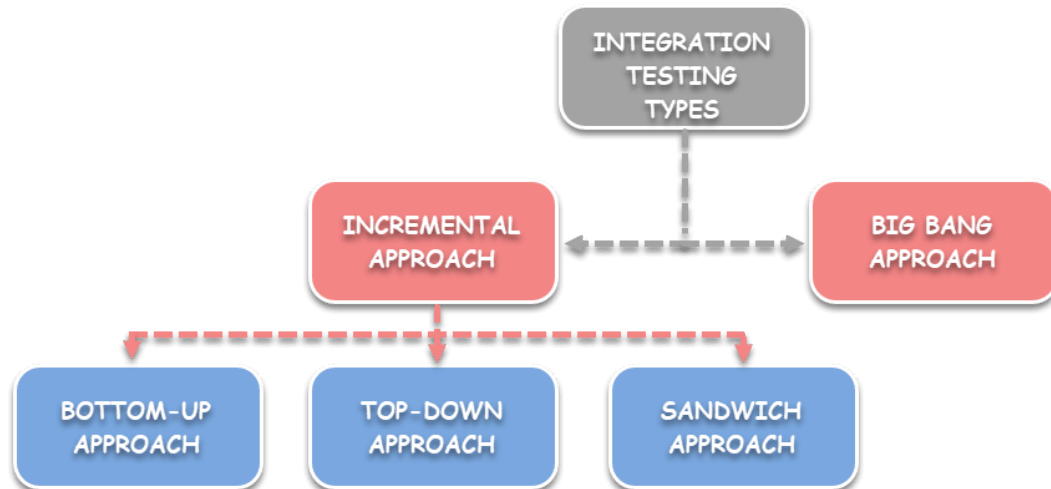
# Integration Testing Approaches

## ✧ 爆炸式集成

- 将所有通过单元测试的模块一次性地集成到一起进行测试，也称为一次性组装或整体拼装。

## ✧ 递增式集成

- 将要测试的模块逐渐集成到已经测试好的模块里面，边集成边测试，测完一部分再连接一部分。



# Big-Bang Approach

---

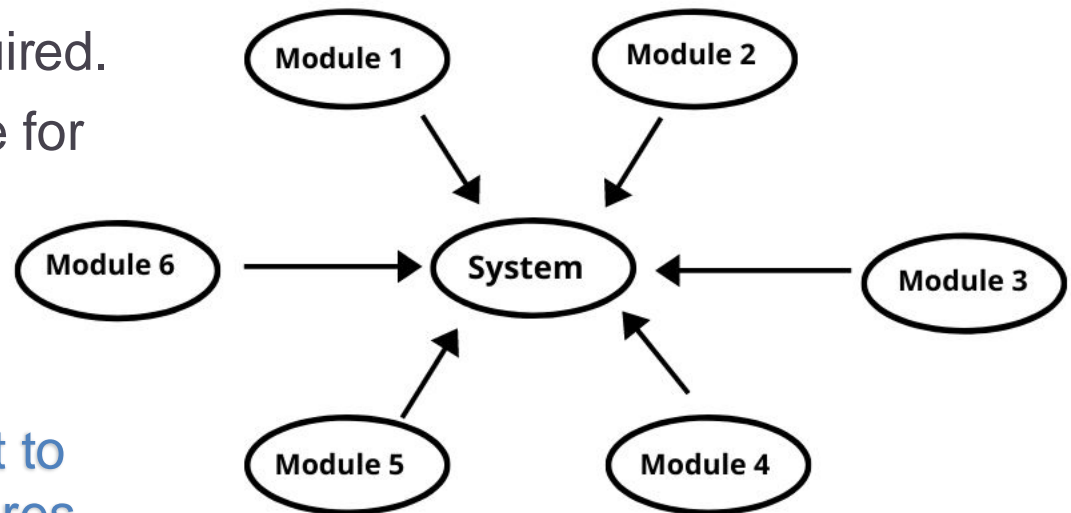
✧ Once all the modules are developed and tested individually, they are integrated once and tested together.

✧ Advantages:

- Only one round is required.
- It is very much suitable for smaller systems.

✧ Disadvantages:

- In general it is time consuming and **difficult to trace the cause of failures.**

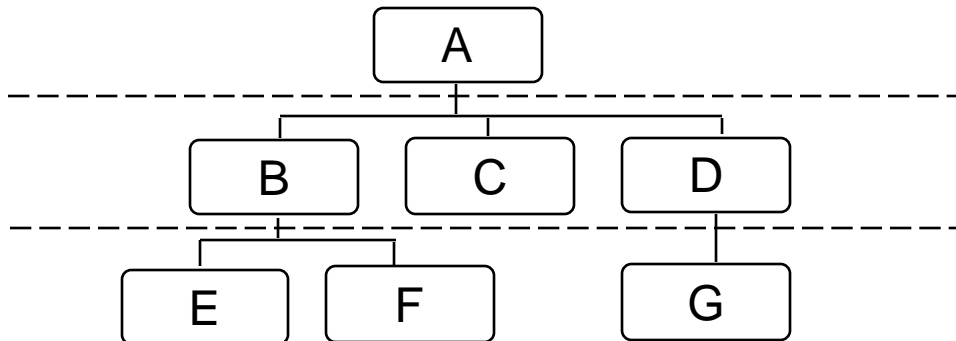




# Top-Down Approach

---

- ✧ Testing takes place from top to bottom, following the control flow or architectural structure (e.g. **starting from the GUI or main menu**).
- ✧ The functionality of the modules at a lower level is simulated with the help of stubs.



**Breadth First Strategy:**

A - B - C - D - E - F - G

**Depth First Strategy:**

A - B - E - F - C - D - G

# Bottom-Up Approach

---

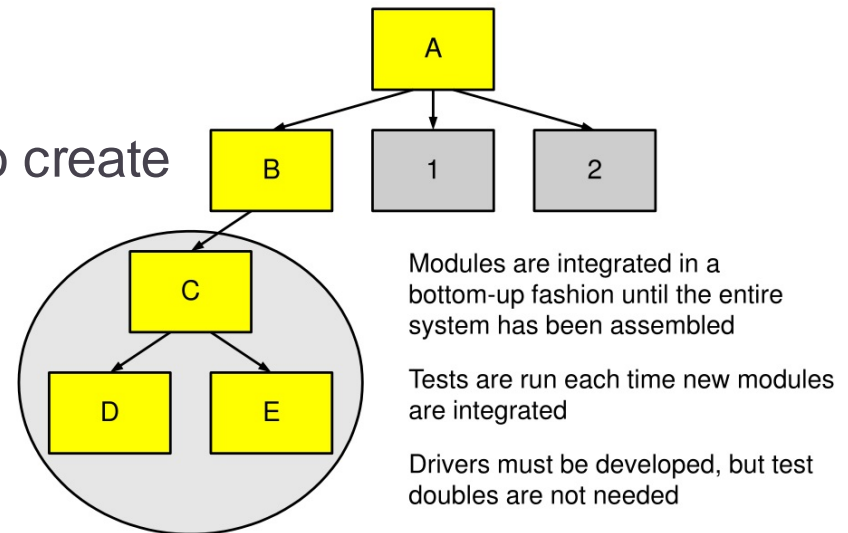
✧ Testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers.

✧ Advantages:

- Test conditions are much easy to create

✧ Disadvantages:

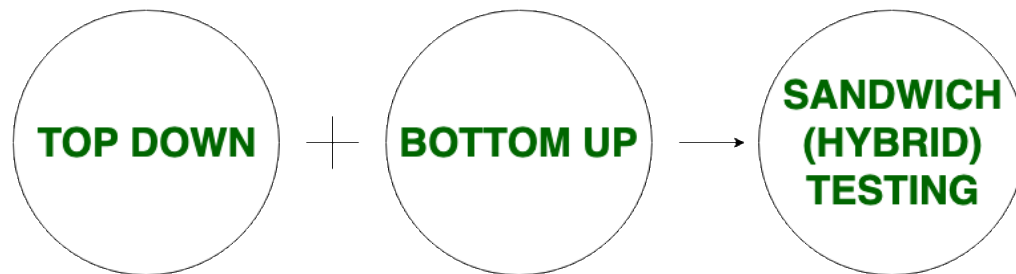
- Requires several drivers
- Data flow is tested very late
- Poor support for early release
- Key interfaces defects are detected late



# Sandwich Approach

---

- ✧ Sandwich approach is the combination of bottom-up approach and top-down approach, so it uses the advantage of both approaches.
- ✧ The system is considered to be made up of **three layers**:
  - A layer in the middle which will be the target of testing
  - A layer above the target layer and a layer below the target layer
  - Testing begins from the outer layer and converges at the middle layer



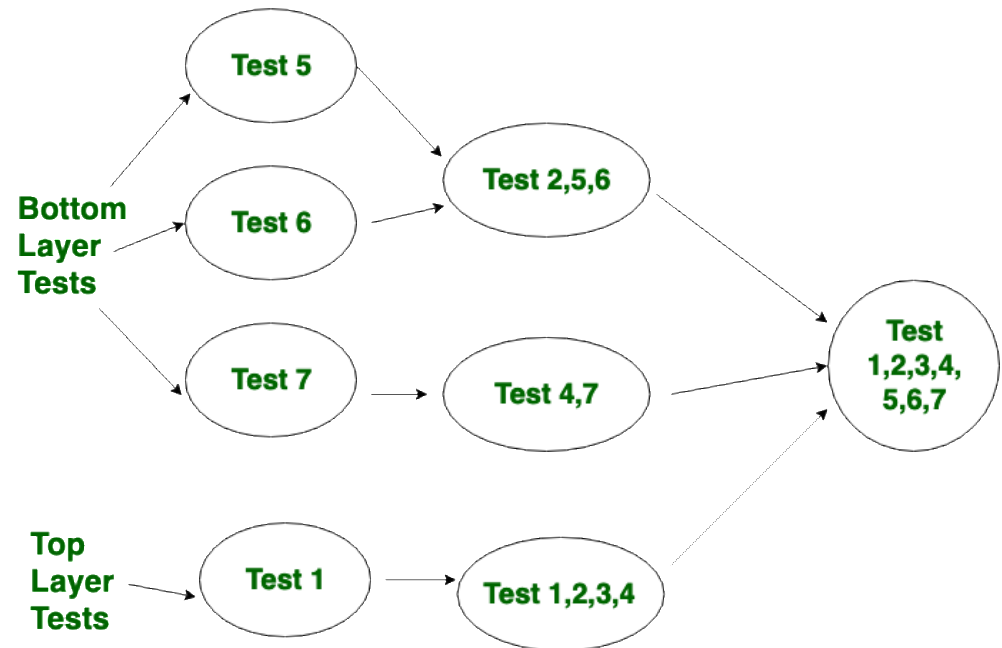
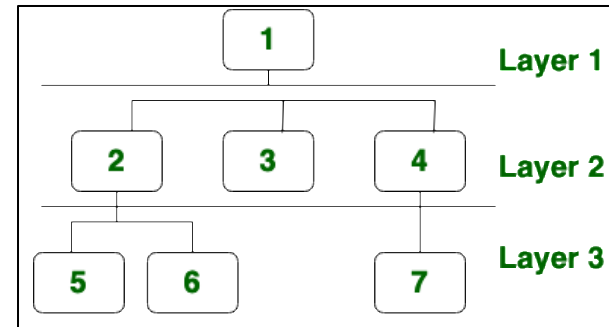
# Sandwich Approach

## ✧ Advantages:

- It allows parallel testing.
- It is a time saving approach.
- It is used in very large projects.

## ✧ Disadvantages:

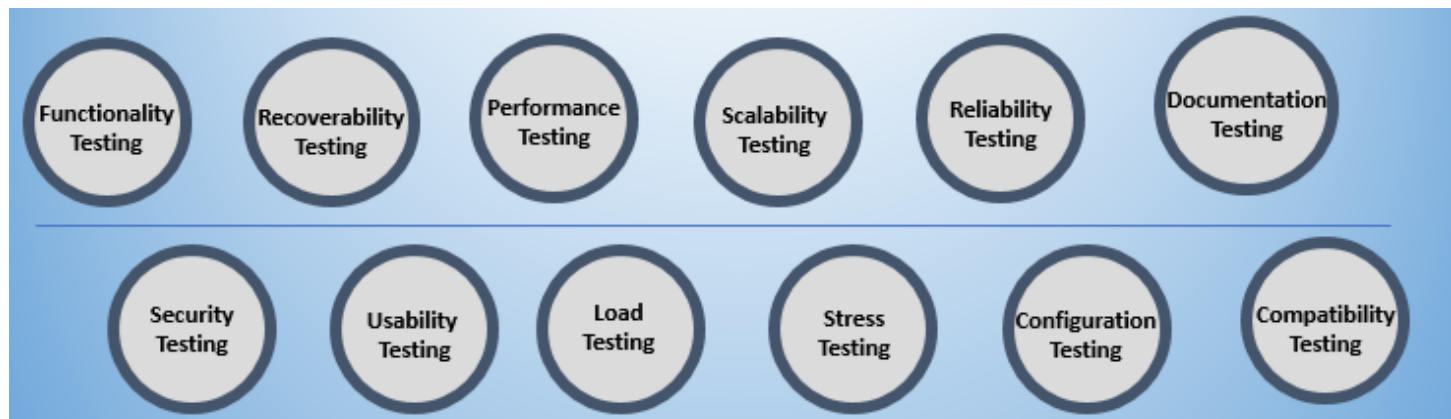
- It is very costly.
- The need of stubs and drivers is very high.



# System Testing

---

- ✧ System testing is testing conducted on a complete integrated system to **evaluate the system's compliance with its specified requirements**.
  - System testing can be considered as a **black-box** test technique.
  - System testing is usually carried out by a team that is independent of the development team in order to measure the quality of the system unbiased.



# Types of System Testing

---

## ✧ Functionality Testing

- This testing makes sure that the functionality of a product is working as per the requirements specification, within the capabilities of the system.

## ✧ Recoverability Testing

- This testing determines whether operations can be continued after a disaster or after the integrity of the system has been lost.

## ✧ Performance Testing

- This testing is carried out to test the speed, scalability, stability and reliability of the software product or application.

# Types of System Testing

---

## ✧ Scalability Testing

- This testing makes sure the system's scaling abilities in various terms like user scaling, geographic scaling, and resource scaling.

## ✧ Reliability Testing

- This testing makes sure the system can be operated for a longer duration without developing failures.

## ✧ Documentation Testing

- This testing makes sure that system's user guide and other help topics documents are correct and usable.

# Types of System Testing

---

## ✧ Security Testing

- This testing makes sure that the system does not allow unauthorized access to data and resources.

## ✧ Usability Testing

- This testing makes sure that the system is easy to use, learn and operate.

## ✧ Load Testing

- This testing is carried out to determine the behavior of a system or software product under extreme load.



# Types of System Testing

---

Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point.

## ✧ Stress Testing

- This testing checks the robustness of the system under the varying loads.

## ✧ Configuration Testing

- This testing determines minimal and optimal H/W and S/W configuration.

## ✧ Compatibility Testing

- This testing is done to ensure that the developed product is compatible with different browsers, hardware platforms, operating system and databases as per the requirement document.

# Integration Testing VS System Testing

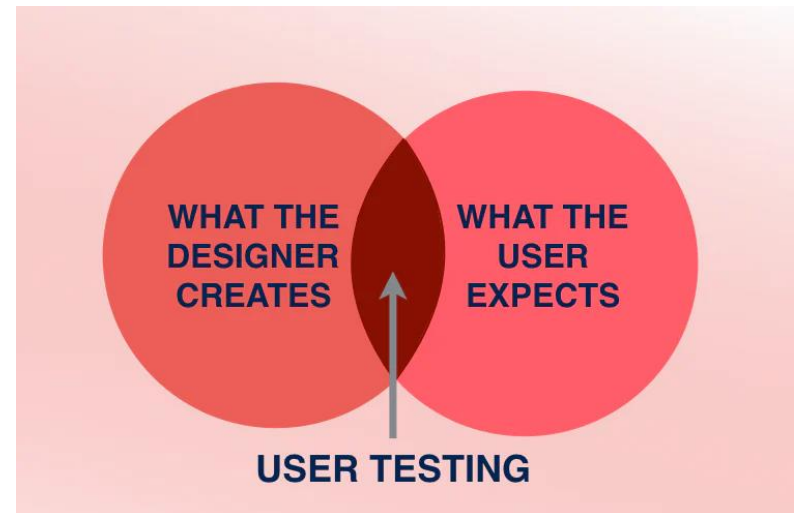
---

Integration Testing	System Testing
Integration testing is mainly done to check how individual modules interact with each other when integrated into a system as a whole.	System testing is mainly done to check if the whole system is working as expected with reference to the specified requirements.
It is conducted after unit testing and will be done each time when a new module is added to the system.	It is conducted at the final level i.e. after the completion of integration testing and just before delivering the system for UAT.
It is a low-level testing.	It is a high-level testing.
Test cases focus on the interface between the system components.	Test cases focus on simulating the real-life scenarios.

# User Testing

---

- ✧ User or customer testing is a stage in the testing process in which **users or customers provide input and advice on system testing.**
- ✧ User testing is essential, even when comprehensive system testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. **These cannot be replicated in a testing environment.**



# Types of User Testing

---

## ✧ Alpha testing

- Users of the software work with the development team to test the software at the developer's site.

## ✧ Beta testing

- A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

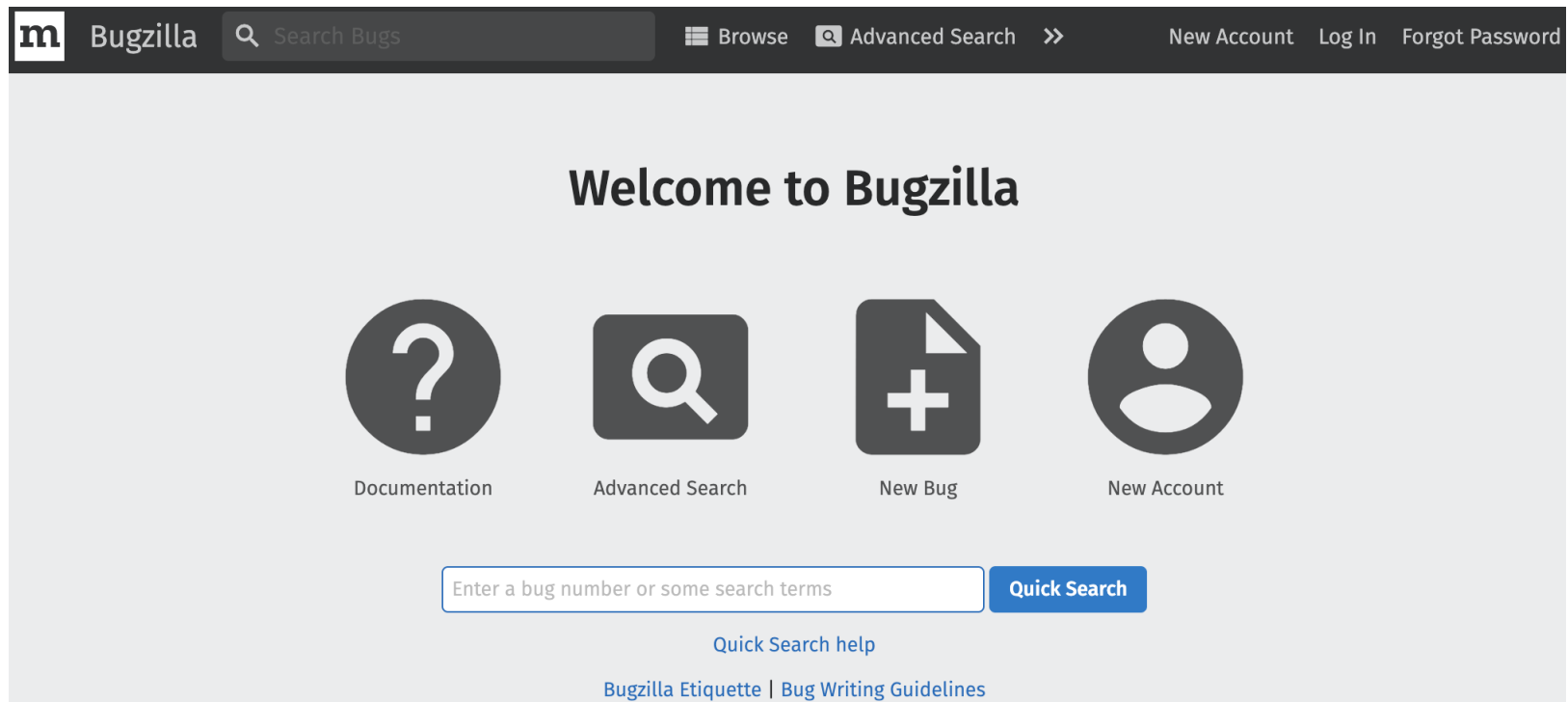
## ✧ Acceptance testing

- Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

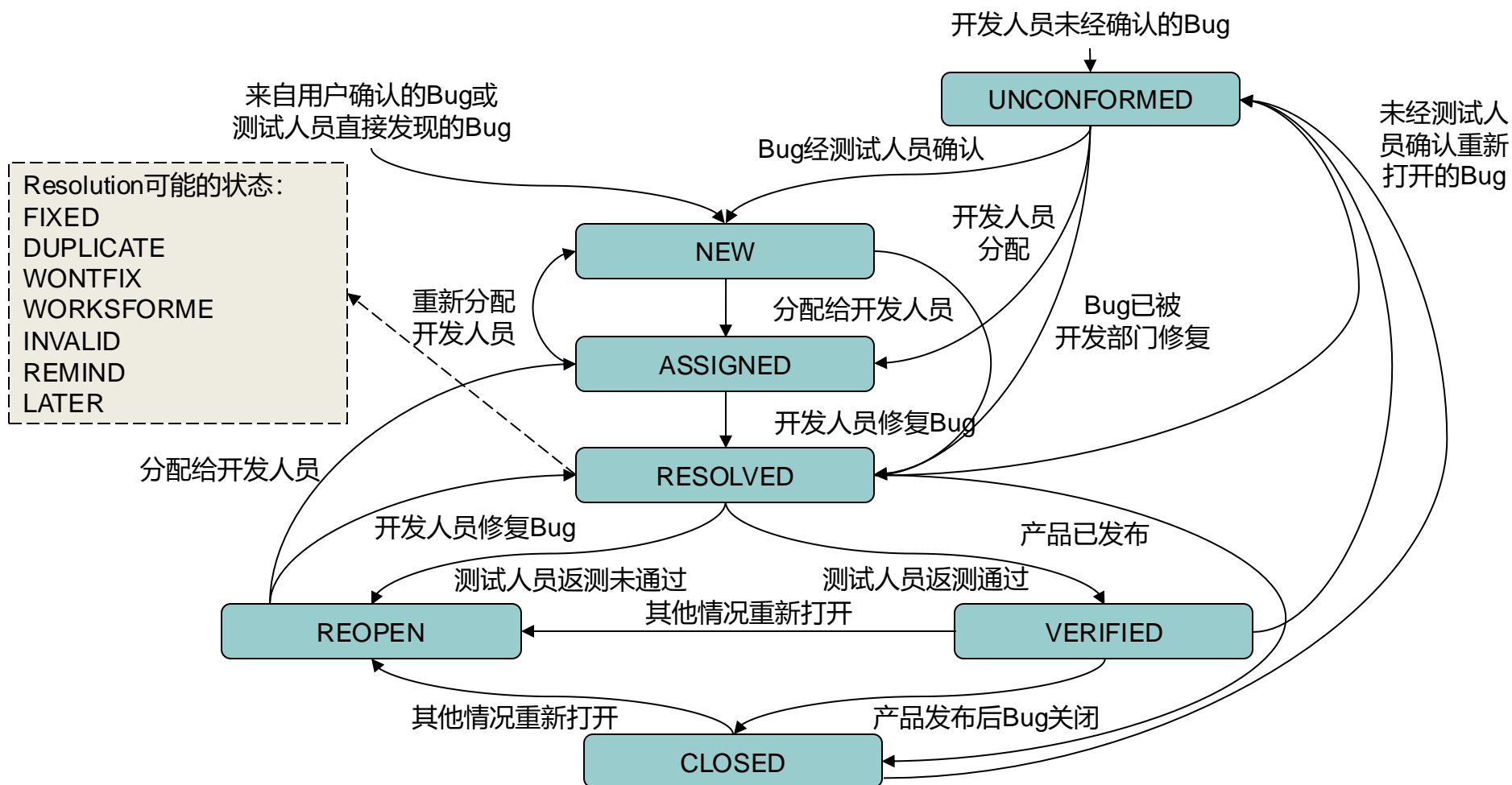
# Bug Tracking System

✧ Bugzilla: <https://www.bugzilla.org/>

✧ JIRA, Mantis, BugFree



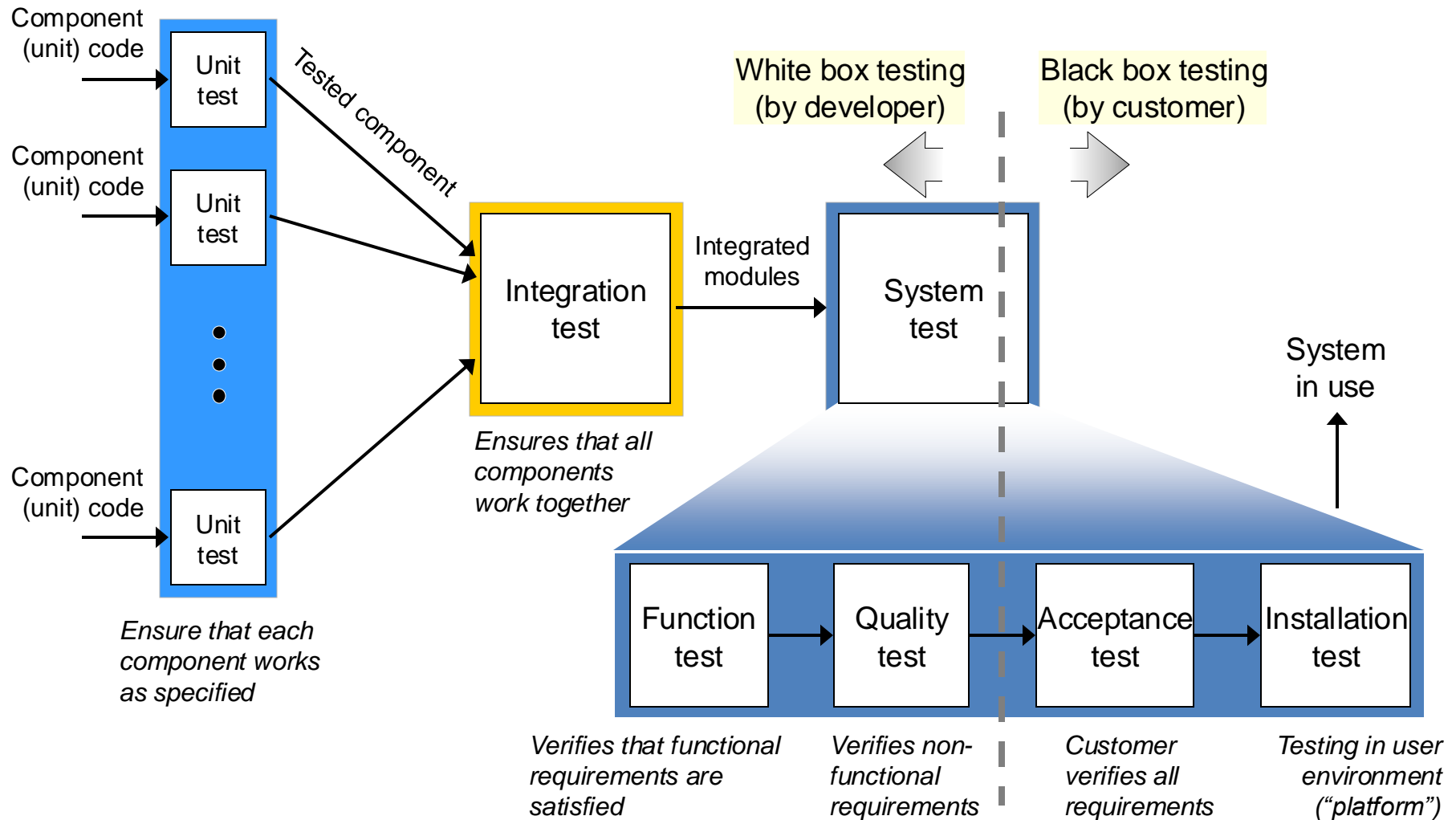
# Life Cycle of a Bug



---

# Summary

# Software Testing





# Software Testing

---

测试类型	对象	目的	测试依据	测试方法
单元测试	模块内部的程序	消除局部模块逻辑和功能上的错误或缺陷	模块逻辑设计 模块外部说明	白盒测试
集成测试	模块间的集成和调用关系	找出和软件设计相关的程序结构、模块调用关系以及模块间接口方面的问题	程序结构	结合使用白盒测试和黑盒测试，较多采用黑盒测试
系统测试	整个系统包括硬件、支持软件、人员等	对整个系统进行一系列整体的、有效性测试	系统结构设计 目标说明书 需求说明书	黑盒测试