哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2 Testing and Test-First Programming

软件测试与测试优先的编程

Wang Zhongjie
rainy@hit.edu.cn

March 5, 2024

Objective of this lecture

- Understand the value of testing, and know the process of test-first programming 认可“测试”的价值，搞清楚“测试优先”的哲理
- Be able to design a test suite for a method by partitioning its input and output space, finding its boundaries, and choosing good test cases 学会用等价划分和边界值分析方法为模块设计测试用例
- Be able to judge a test suite by measuring its code coverage 可用工具度量一组测试用例对代码的“覆盖度”
- Understand and know when to use blackbox vs. whitebox testing, unit tests vs. integration tests, and automated regression testing 各种各样的测试，都有些初步了解

Outline

- **Software Testing**
- **Test Case 测试用例**
- **Test-First Programming / Test-Driven Development (TDD) 测试优先的编程/测试驱动开发**
- **Unit Testing 单元测试**
- **Automated Unit Testing with JUnit 使用JUnit进行自动化单元测试**
- **Black-box Testing 黑盒测试**
  - Choosing Test Cases by Partitioning 等价类划分
  - Include Boundaries in the Partition 边界值分析
- **White-box Testing * 白盒测试**
- **Coverage of Testing 覆盖度**
- **Integration Testing * 集成测试**
- **Automated Testing and Regression Testing * 回归测试**
- **Documenting Your Testing Strategy 在程序中文档化测试策略**

确保程序正确性/健壮性的最普遍的手段：测试

1 设计测试用例
2 用JUnit写测试程序
3 自动化测试过程

Reading

- **MIT 6.031：03**
- **CMU 17-214：Sep 05**
- **代码大全：第22章**
- **软件工程--实践者的研究方法：第17-18章**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1 Software Testing

What is testing?

- **Software testing** is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. 提高软件质量的重要手段
  - It is the process of executing a program or application with the intent of finding bugs (errors or other defects), and verifying that the software product is fit for use. 确认是否达到可用级别(用户需求)
  - It involves the execution of a software component to evaluate one or more properties of interest. 关注系统的某一侧面的质量特性

What is testing?

- Even with the best validation, **it's very hard to achieve perfect quality in software.** 即使是最好的测试，也无法达到100%的无错误

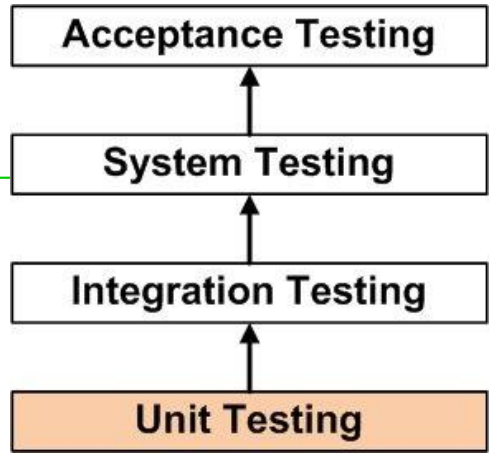
- Some typical **residual defect rates** 残留缺陷率 (bugs left over after the software has shipped) per kloc (1000 Lines of Code):
  - 1-10 defects/kloc: Typical industry software.
  - 0.1-1 defects/kloc: High-quality validation. The Java libraries might achieve this level of correctness.
  - 0.01-0.1 defects/kloc: The very best, safety-critical validation. NASA and companies like Praxis can achieve this level.
- **This can be discouraging for large systems.** For example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs! 触目惊心

Test Characteristics

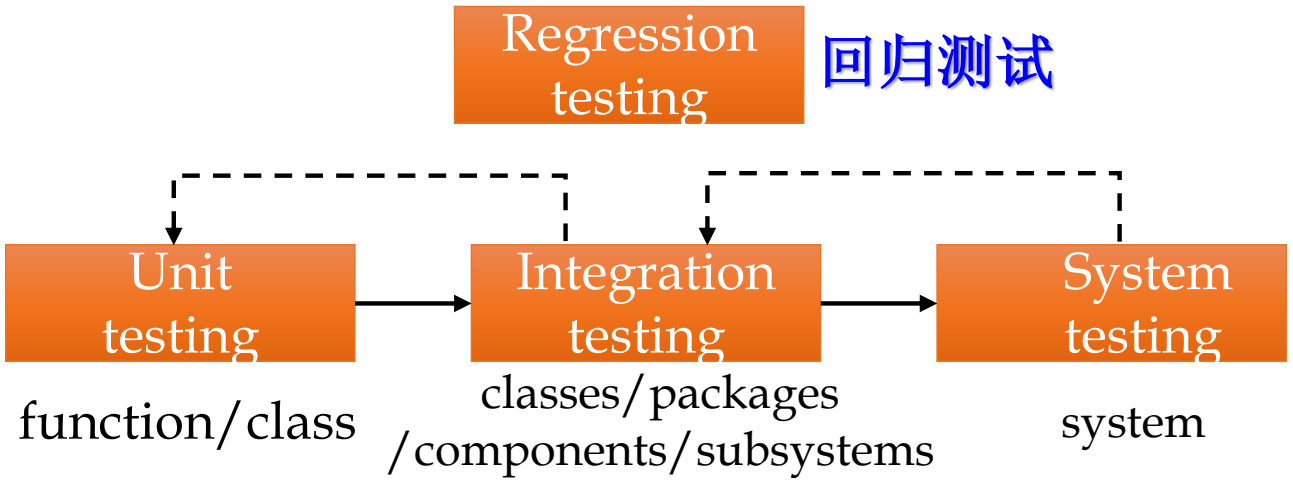
- **Testing's goal runs counter to the goals of other development activities. The goal is to find errors.** 测试跟其他活动的目标相反：破坏、证错、“负能量”
- **Testing can never completely prove the absence of errors.** 再好的测试也无法证明系统里不存在错误

- **What is a good test?**
  - A good test has a high probability of finding an error 能发现错误
  - A good test is not redundant 不冗余
  - A good test should be “best of breed” 最佳特性
  - A good test should be neither too simple nor too complex 别太复杂也别太简单

Testing levels



- **Unit testing** 单元测试**:** refers to tests that verify the functionality of a specific section of code, usually at the function level.
- **Integration testing** 集成测试**:** the combined execution of two or more classes, packages, components, subsystems that have been created by multiple programmers or programming teams.
- **System testing** 系统测试**:** to test a completely integrated system to verify that the system meets its requirements, which executes the software in its final configuration.



Other testing types

- **Installation testing**
- **Compatibility testing**
- **Smoke and sanity testing**
- **Regression testing**
- **Acceptance testing**
- **Alpha testing / Beta testing**
- **Performance/load/scalability**
- **Usability testing**
- **Accessibility testing**
- **Security testing**
- **and so on…**

Static vs. Dynamic testing

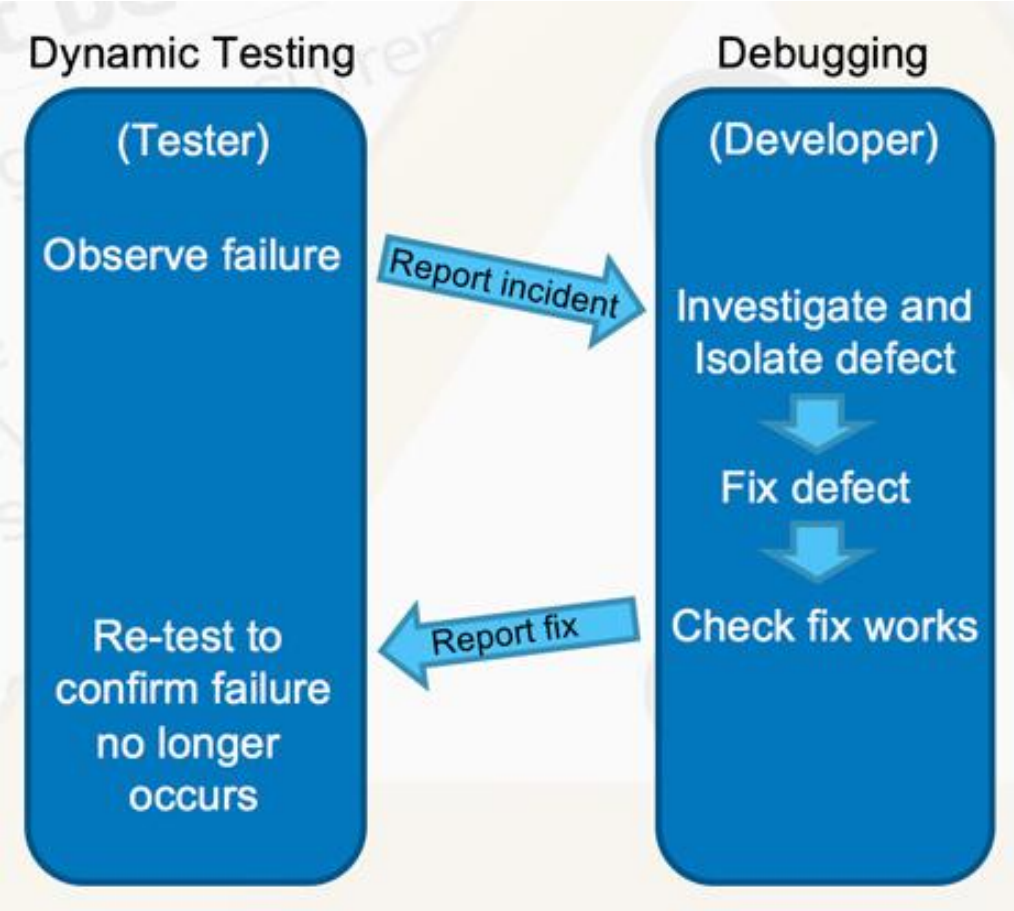
- **Static testing is performed without actually executing programs.**
  - Static testing is often implicit, as proofreading, plus when programming tools/text editors check source code structure or compilers (pre-compilers) check syntax and data flow as static program analysis.
  - Reviews, walkthroughs, or inspections are referred to as static testing.

静态测试 vs 动态测试：靠眼睛看 vs 撸起袖子使劲干

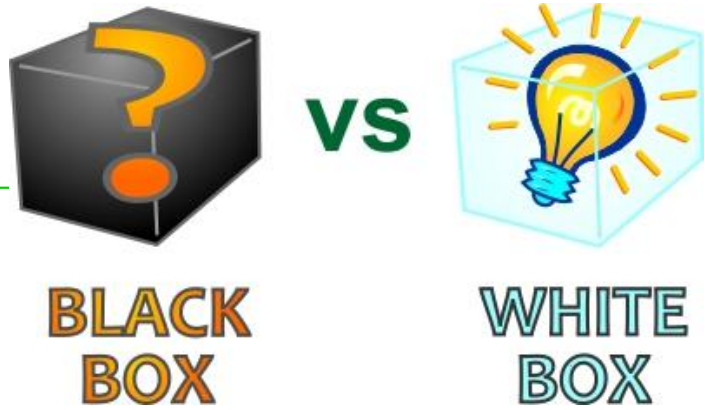
- **Dynamic testing describes the testing of the dynamic behavior of code, which actually executes programmed code with a given set of test cases.**
  - Dynamic testing may begin before the program is 100% complete in order to test particular sections of code and are applied to discrete functions or modules.
  - Typical techniques for this are either using stubs/drivers or execution from a debugger environment.

Testing vs. Debugging

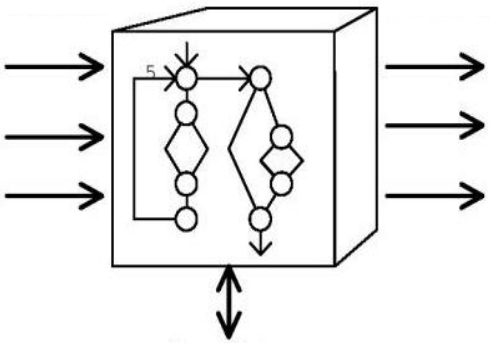
- **Testing is a means of detecting errors.** 测试：发现是否存在错误
- **Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.** 调试：识别错误根源，消除错误
- **This section deals exclusively with testing.**
- **Debugging is discussed in detail in Chapter 6.4.**



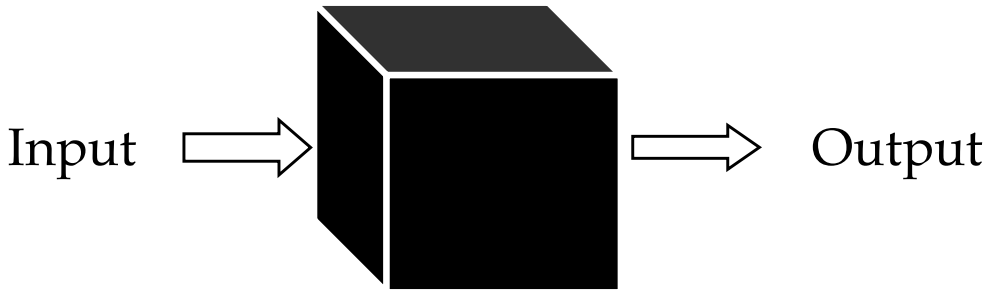
White-box vs. black-box testing



- **White-box testing** tests internal structures or workings of a program by seeing the source code. 白盒测试：对程序内部代码结构的测试
- **Black-box testing** treats the software as a "black box", examining functionality without any knowledge of internal implementation, without seeing the source code. 黑盒测试：对程序外部表现出来的行为的测试



White-box testing



Black-box testing

Why Software Testing is Hard

- **Exhaustive testing is infeasible:** The space of possible test cases is generally too big to cover exhaustively. 穷举+暴力=不可能
  - Imagine exhaustively testing a 32-bit floating-point multiply operation, a*b. There are 2^64 test cases!
- **Haphazard testing ("just try it and see if it works") is less likely to find bugs,** unless the program is so buggy that an arbitrarily-chosen input is more likely to fail than to succeed. 靠偶然测试没意义
  - It also doesn't increase our confidence in program correctness.
- **Random or statistical testing doesn't work well for software.** Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole production lot. 基于样本的统计数据对软件测试意义不大—软件与产品的巨大差异
  - This is only true for physical artifacts, but it's not true for software.

Why Software Testing is Hard

- **Software behavior varies discontinuously and discretely across the space of possible inputs.** 软件行为在离散输入空间中差异巨大
  - The system may seem to work fine across a broad range of inputs, and then abruptly fail at a single boundary point. 大多数正确，少数点出错
  - The famous **Pentium FDIV bug** affected approximately 1 in 9 billion divisions, discovered in 1994.
  - Stack overflows, out of memory errors, and many bugs tend to happen abruptly, and always in the same way, not with probabilistic variation. bug出现往往不符合特定概率分布
- **That's different from physical systems** 无统计分布规律可循
  - There is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point (so that statistical testing will observe some failures even before the point is reached).

Software test cases must be chosen carefully and systematically

Pentium FDIV bug

- https://en.wikipedia.org/wiki/Pentium_FDIV_bug
- **The Pentium FDIV bug was a computer bug that affected the floating point unit (FPU) of the early Intel Pentium processors.**
- **Because of the bug, the processor might return incorrect binary floating point results when dividing a number.**
  - 奔腾浮点除错误是Intel旧版本Pentium浮点运算器FPU的一个错误，起源于浮点除指令。
- **In December 1994, Intel recalled the defective processors.**
- **In January 1995, Intel announced "a pre-tax charge of $475 million against earnings, ostensibly the total cost associated with replacement of the flawed processors.“**
- **最初找出bug的原始记录：http://www.trnicely.net/#PENT**

Ariane 5 launch vehicle

- **In the 1990s, the Ariane 5 launch vehicle, designed and built for the European Space Agency, self-destructed 37 seconds after its first launch.**
- **The reason was a control software bug that went undetected.**
  - The Ariane 5's guidance software was reused from the Ariane 4, which was a slower rocket.
  - When the velocity calculation converted from a 64-bit floating point number (a double in Java terminology, though this software wasn't written in Java) to a 16-bit signed integer (a short), it overflowed the small integer and caused an exception to be thrown.
  - The exception handler had been disabled for efficiency reasons, so the guidance software crashed.
  - Without guidance, the rocket crashed too. The cost of the failure was $1 billion.

Putting on Your Testing Hat

- Testing requires having the right attitude. When you're coding, your goal is to make the program work, but as a tester, you want to **make it fail**. 要转变心态，用“让其出错”和“尽快出错”作为写高质量代码的日常法宝
  - That's a subtle but important difference. It is all too tempting to treat code you've just written as a precious thing, a fragile eggshell, and test it very lightly just to see it work.

    抛弃这样的想法：我的代码是宝贝，可不能总让它出错！

- **Instead, you have to be brutal.** A good tester wields a sledgehammer and beats the program everywhere it might be vulnerable, so that those vulnerabilities can be eliminated. 学会对自己的代码更暴力些！

Not only “make it fail”, but also “fail fast”.

2 Test Case

What is test case?

- **A test case, is a set of test inputs, execution conditions, and expected results. i.e., test case = {test inputs + execution conditions+ expected results} 测试用例：输入+执行条件+期望结果**
  - A test case is developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
  - A test case could simply be a question that you ask of the program. The point of running the test is to gain information, for example whether the program will pass or fail the test.
  - Test case is the cornerstone of Quality Assurance whereas they are developed to verify quality and behavior of a product.
- **E.g., test cases:** `{2,4}, {0,0}, {-2,4}` **for program** `y=x^2`

Test cases are valuable assets 资产 in your project

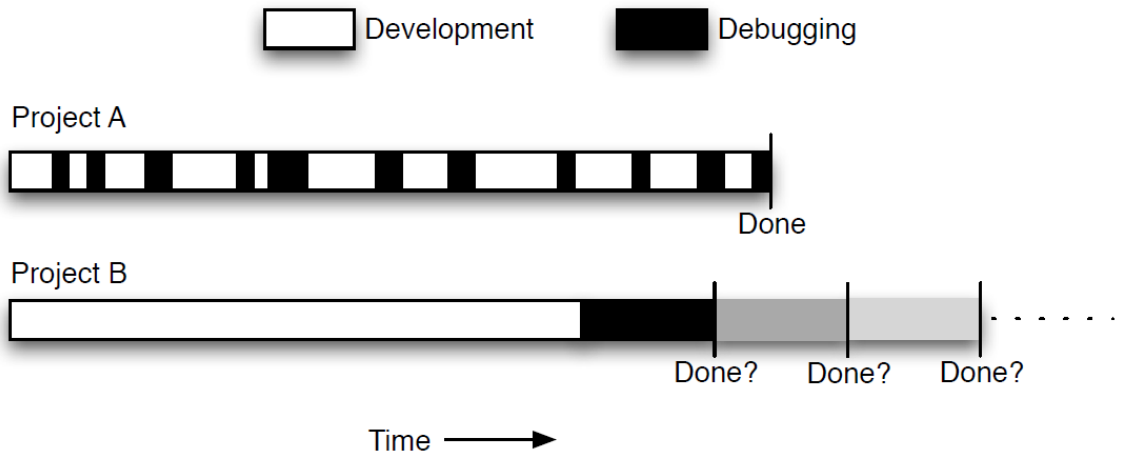
```java
public final class GenericMathTest {
    @Test
    public void testMean() {
        final int[] intTestValues = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
        final int intTestResult = 8;
        final double[] doubleTestValues = new double[intTestValues.length];
        final double doubleTestResult = intTestResult + 0.03;
        for (int i = 0; i < intTestValues.length; i++) {
            doubleTestValues[i] = intTestValues[i] + 0.03;
        }
```

Characteristics of good test case

- **Most likely to catch the wrong** 最可能发现错误
- **Not repetitive and not redundant** 不重复、不冗余
- **The most effective in a group of similar test cases** 最有效
- **Neither too simple nor too complicated** 既不简单也不复杂

- **若干相关学术研究：**
  - 根据spec和代码，自动生成测试用例
  - 测试用例的最小化
  - 回归测试的最小用例集

3 Test-First Programming



Test-First Programming 测试优先的编程

- **Test-first programming: Write the tests before you write the code.**
- **Motivation**
  - Test early and often. → Make it fail, fail it fast!
  - Don't leave testing until the end, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code.
  - It's far more pleasant to test your code as you develop it. "测试代码"比写代码更有成就感！
- **Process:**
  - Write a specification for the function. 先写spec
  - Write tests that exercise the specification. 再写符合spec的测试用例
  - Write the actual code. Once your code passes the tests you wrote, you're done. 写代码、执行测试、有问题再改、再执行测试用例，直到通过它

Test-First Programming

- **Spec describes the input and output behavior of the function.**
  - It gives the types of the parameters and any additional constraints on them (e.g. `sqrt()`'s parameter must be nonnegative).
  - It also gives the type of the return value and how the return value relates to the inputs.
  - In code, the specification consists of the method signature and the comment above it that describes what it does.

写测试用例，就是理解、修正、完善你的spec设计的过程

- **Writing tests first is a good way to understand the specification.**
  - The specification can be buggy, too — incorrect, incomplete, ambiguous, missing corner cases.
  - Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec.

What is specification?

Details of specification will be learned in Chapter 2

Specification of a function is the description of the function's behavior:

- **the types of parameters**
- **type of return value**
- **constraints and relationships between them.**

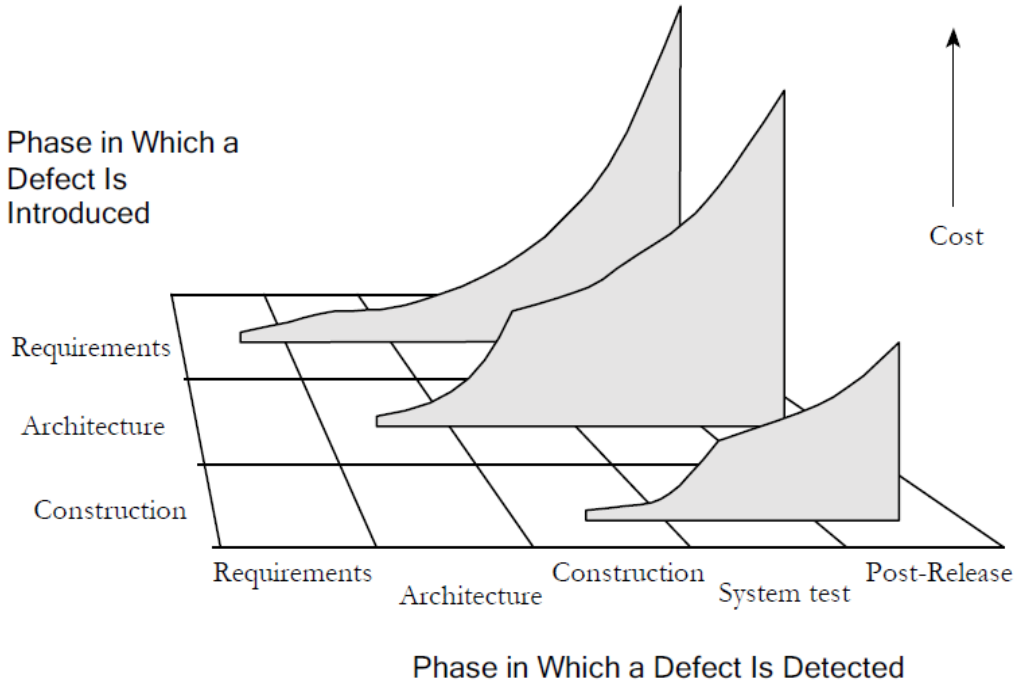
```
/**
 * Returns the element at the specified position of this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant time.
 * In some implementations, it may run in time proportional to the
 * element position.
 *
 * @param index position of element to return; must be non-negative and
 *              less than the size of this list.
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

postcondition

precondition

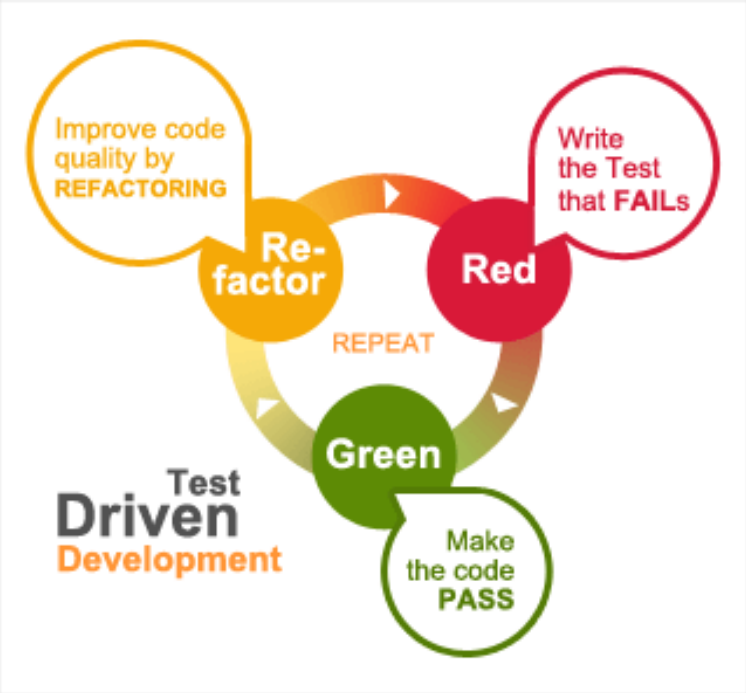
Test First or Test Last?

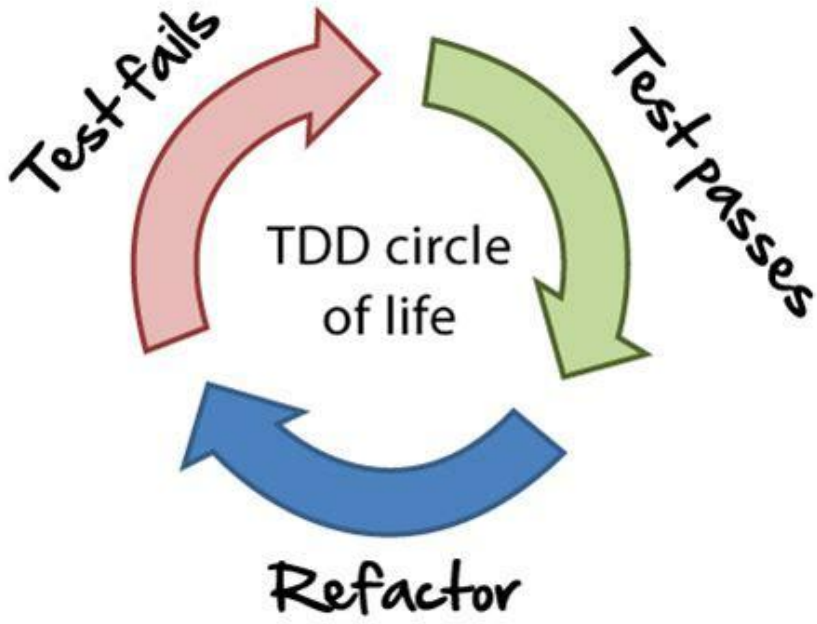
- **Debugging and associated rework takes about >50 percent of the time spent in a typical software development cycle.**
- **The defect-cost increase graph suggests that writing test cases first will minimize the amount of time between when a defect is inserted into the code and when the defect is detected and removed. 先写测试会节省大量的调试时间**



Test-driven development (TDD)

- **Test-driven development (TDD)** is a development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only.
- **It is opposed to software development that allows software to be added that is not proven to meet requirements.**

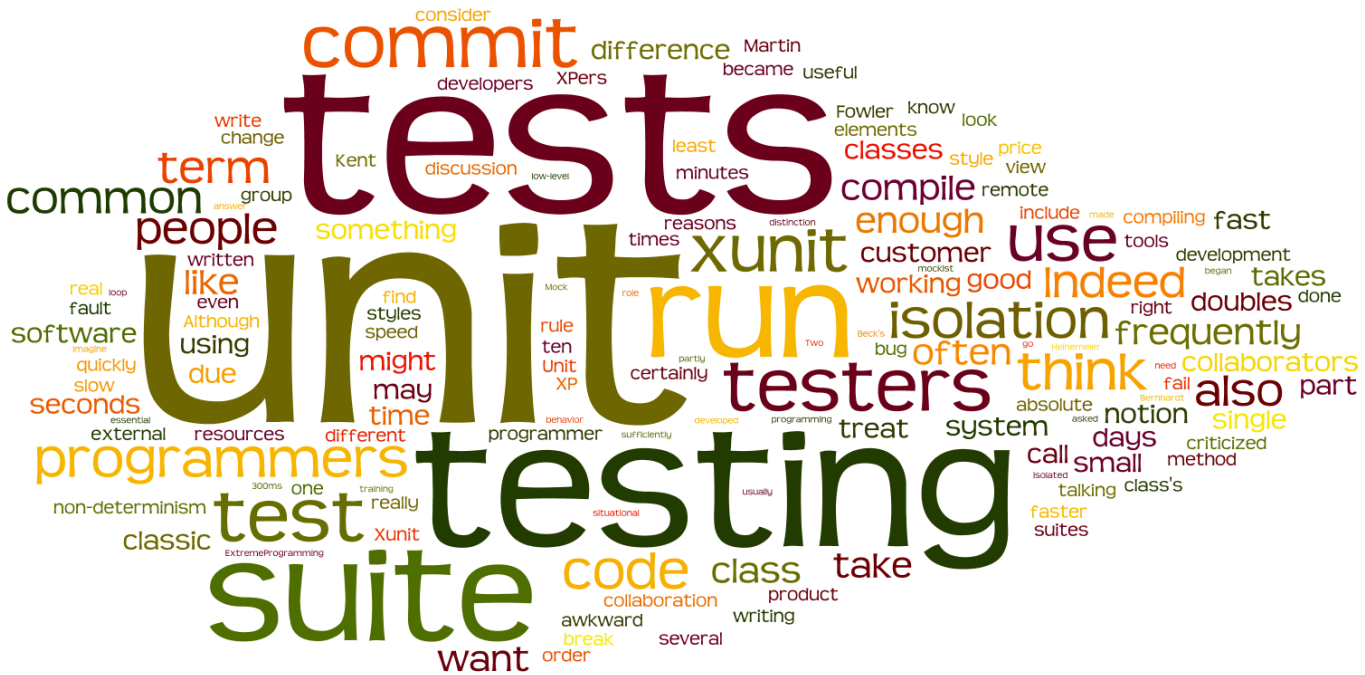




4 Unit Testing

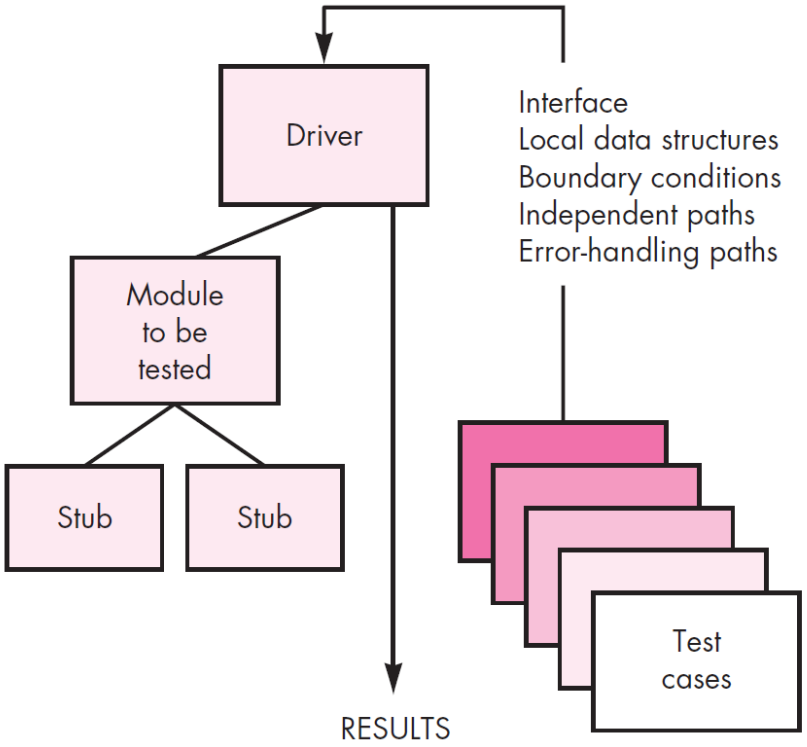
Unit Testing 单元测试

- **Unit testing focuses verification effort on the smallest unit of software design—the software component or module. 针对软件的最小单元模型开展测试，隔离各个模块，容易定位错误和调试**
  - Testing modules in isolation leads to much easier debugging.
  - When a unit test for a module fails, you can be more confident that the bug is found in that module, rather than anywhere in the program.



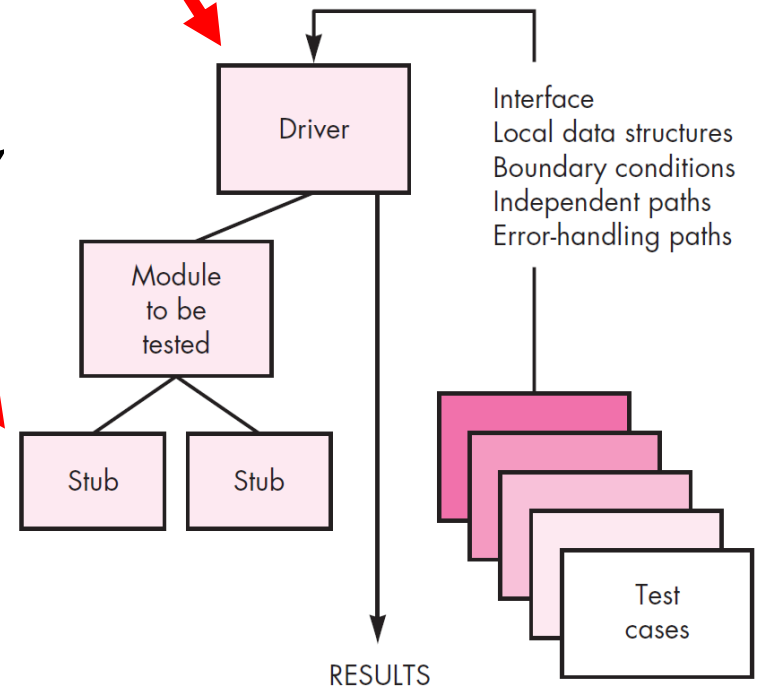
Unit-test procedures

- **Unit testing** is normally considered as an adjunct to the coding step.
  - Test-first programming!
- **A review of design information provides guidance for establishing test cases that are likely to uncover errors. Each test case should be coupled with a set of expected results.**



Unit-test procedures

- **Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.**
- **Driver :** A "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- **Stubs:** serve to replace modules that are subordinate (invoked by) the component to be tested.
  - A stub uses the subordinate module's interface, may do minimal data manipulation, verification of entry, and returns control to the module undergoing testing.



5 Automated Unit Testing with JUnit

A Popular unit test framework: JUnit

- **JUnit is a widely-adopted unit testing framework for Java.**
  - JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit.

- **JUnit is linked as a JAR at compile-time; the framework resides under package `junit.framework` for JUnit 3.8 and earlier, and under package `org.junit` for JUnit 4 and later.**
  - A research survey performed in 2013 across 10,000 Java projects hosted on GitHub found that JUnit, (in a tie with slf4j-api), was the most commonly included external library. Each library was used by 30.7% of projects.

http://www.junit.org

http://junit.sourceforge.net/javadoc/

Junit test case

- **A JUnit unit test is written as a method preceded by the annotation `@Test`.**
- **A unit test method typically contains one or more calls to the module being tested, and then checks the results using assertion methods like `assertEquals`, `assertTrue`, and `assertFalse`.**
- **For example, the tests we chose for `Math.max()` might look like this when implemented for JUnit:**

```
@Test
public void testALessThanB() {
    assertEquals(2, Math.max(1, 2));
}
```

expected actual

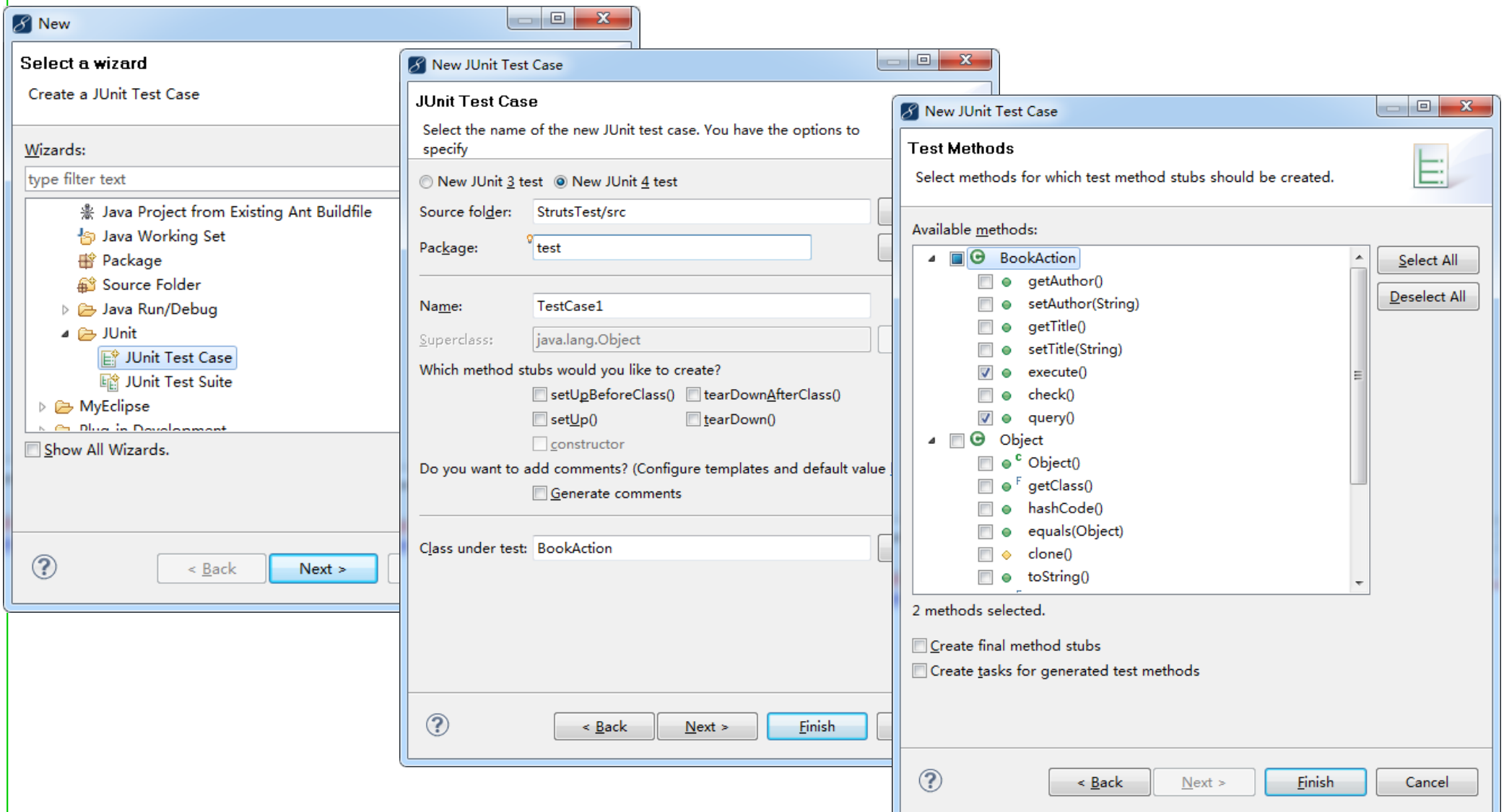
```
@Test
public void testBothEqual() {
    assertEquals(9, Math.max(9, 9));
}

@Test
public void testAGreaterThanB() {
    assertEquals(-5. Math.max(-5, -6));
}
```

To show that this is a test

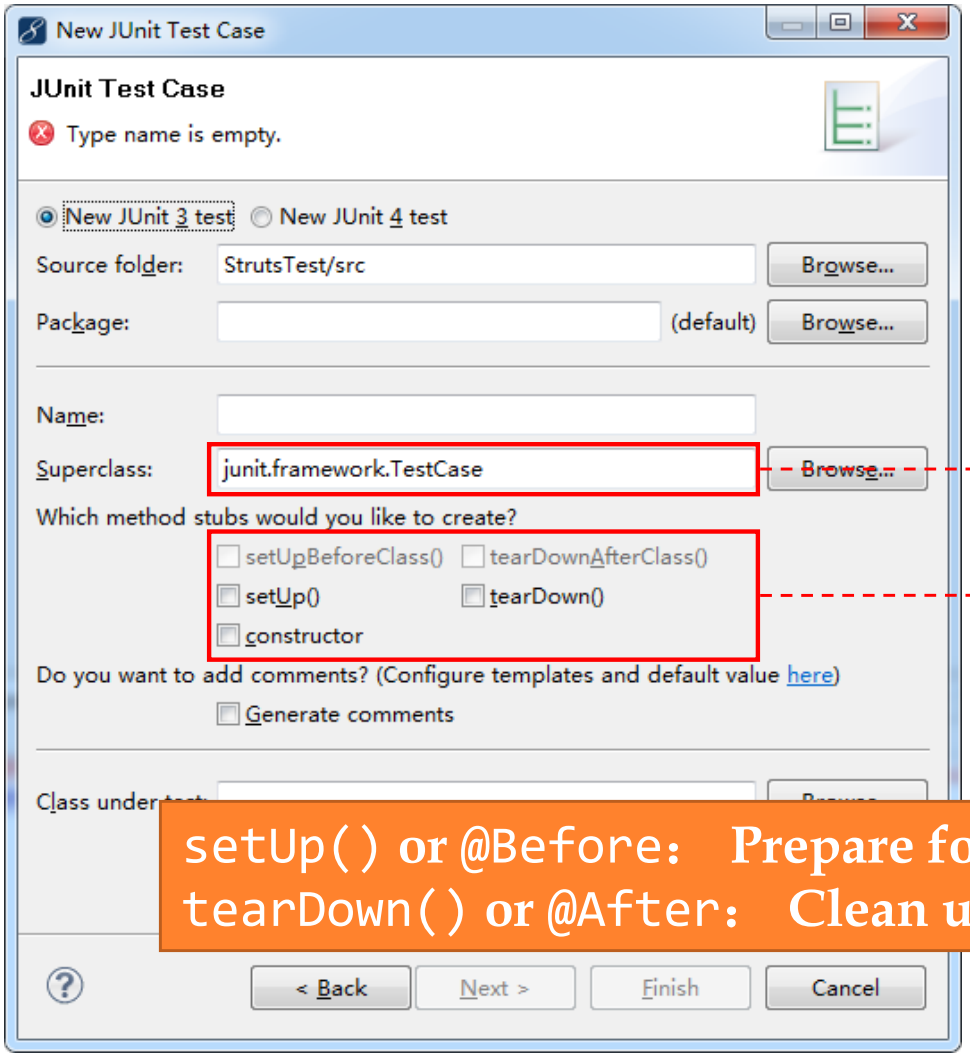
JUnit

- **To create a new JUnit test case or test suite in an existing project**

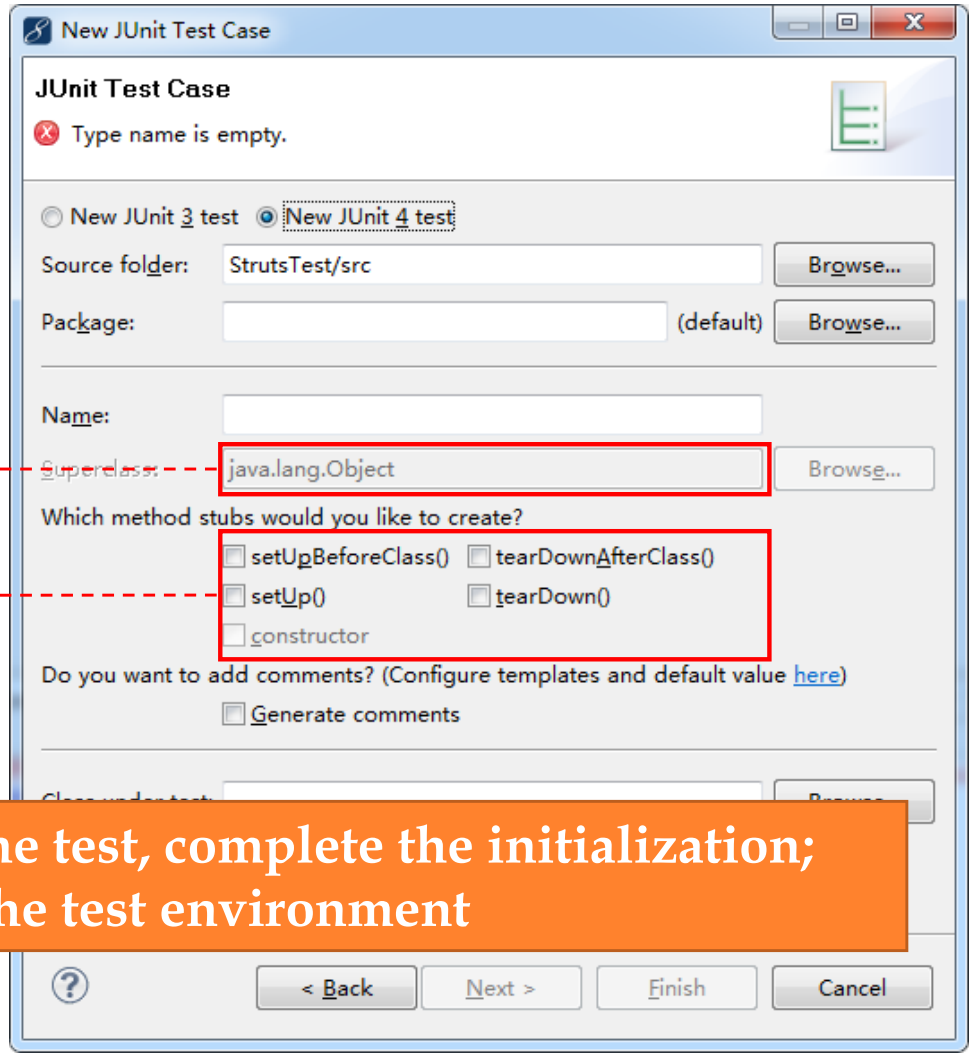


JUnit

- **JUnit3:**



- **JUnit4:**



`setUp()` or `@Before`: Prepare for the test, complete the initialization;
`tearDown()` or `@After`: Clean up the test environment

JUnit 4 example

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}

------

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

A more complex JUnit test

```
@Test
public void lookupEmailAddresses() {
   assertThat(new CartoonCharacterEmailLookupService().
         getResults("looney"), allOf(
            not(empty()),
            containsInAnyOrder(
                allOf(instanceOf(Map.class), hasEntry("id", "56"),
                              hasEntry("email", "roadrunner@fast.org")),
                allOf(instanceOf(Map.class), hasEntry("id", "76"),
                              hasEntry("email", "wiley@acme.com"))
            )
        ));
    }
```

assertXXX in JUnit

```
assertArrayEquals("failure - byte arrays not same", expected, actual);
assertEquals("failure - strings are not equal", "text", "text");
assertFalse("failure - should be false", false);
assertNotNull("should not be null", new Object());
assertNotSame("should not be same Object", new Object(), new Object());
assertNull("should be null", null);
assertSame("should be same", aNumber, aNumber);
assertTrue("failure - should be true", true);

assertThat("albumen", both(containsString("a")).and(containsString("b")));
assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
assertThat("good", allOf(equalTo("good"), startsWith("good")));
```

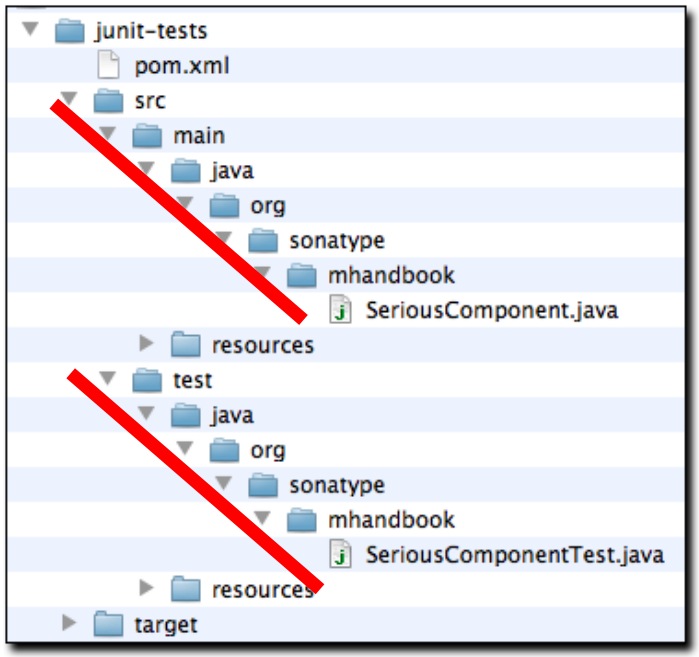
查看实际值是否满足指定的条件，条件使用Hamcrest Matchers匹配符进行匹配

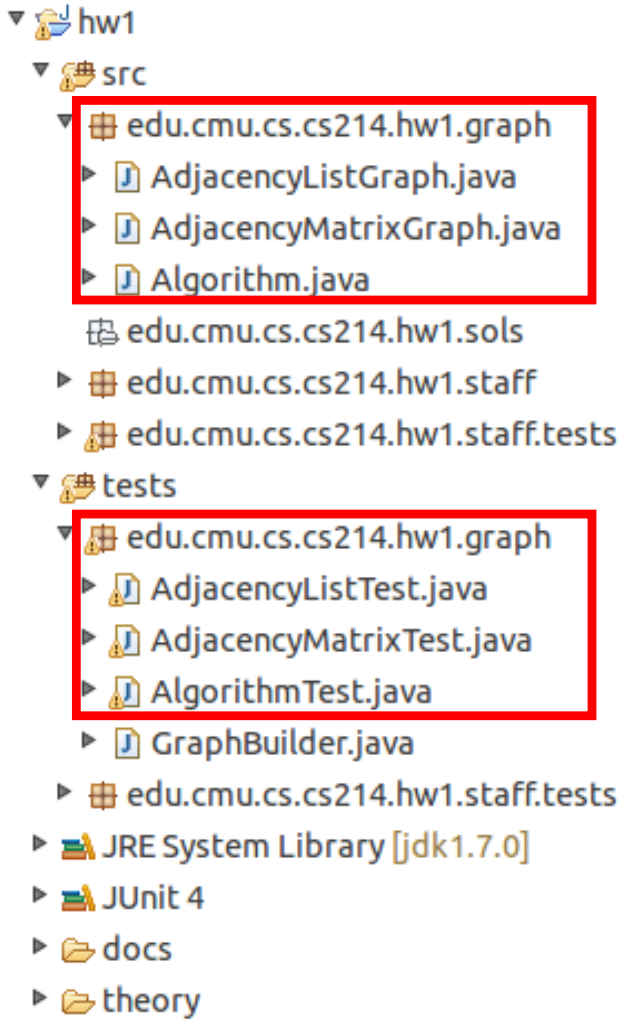
Complete list can be found in

https://github.com/junit-team/junit4/wiki/Assertions

JUnit Test organization

- **Have a test class `FooTest` for each public class `Foo`**
- **Have a `source` directory and a `test` directory**
  - Store `FooTest` and `Foo` in the same package
  - Tests can access members with default (package) visibility

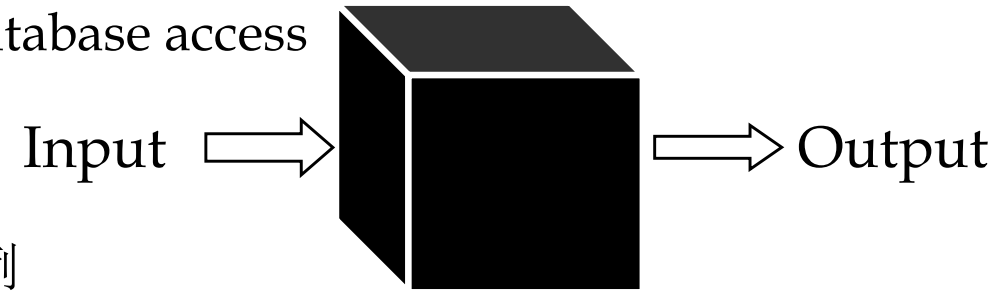


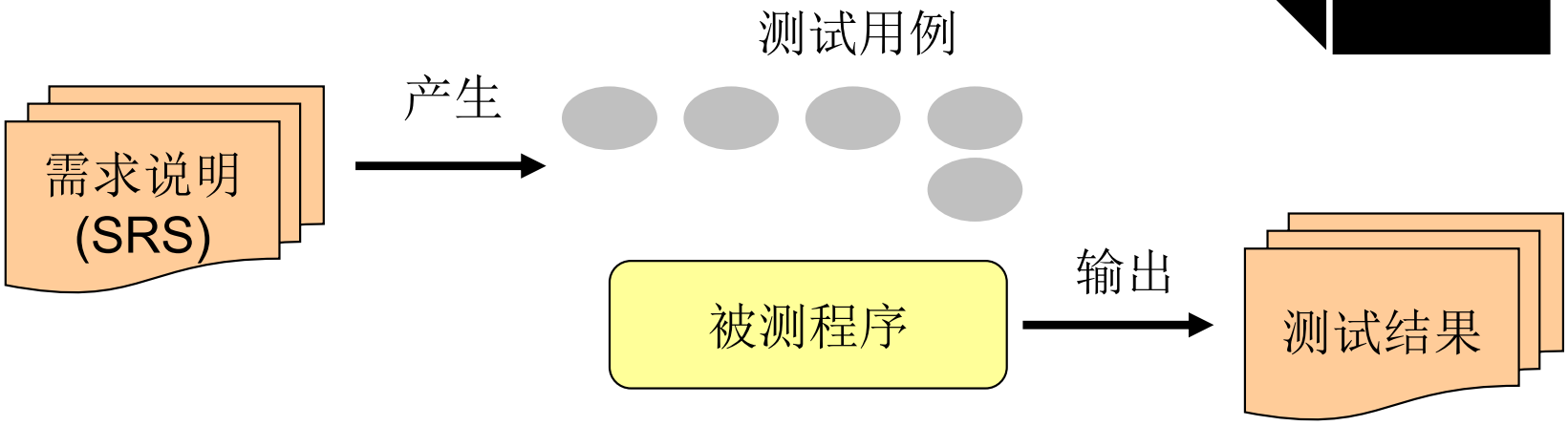


6 Black-box Testing

Black-box testing

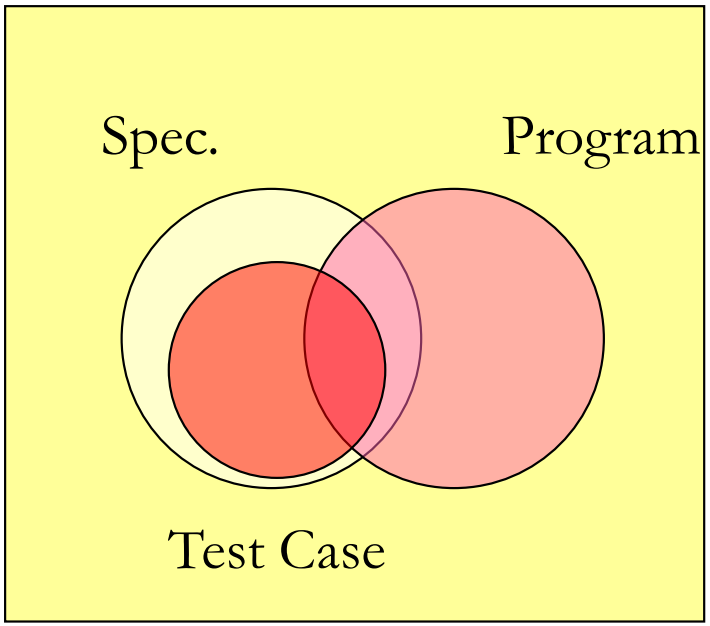
- **Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. 黑盒测试：用于检查代码的功能，不关心内部实现细节**
- **Black-box testing attempts to find errors in the following types:**
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external database access
  - Behavior or performance errors
  - Initialization and termination errors

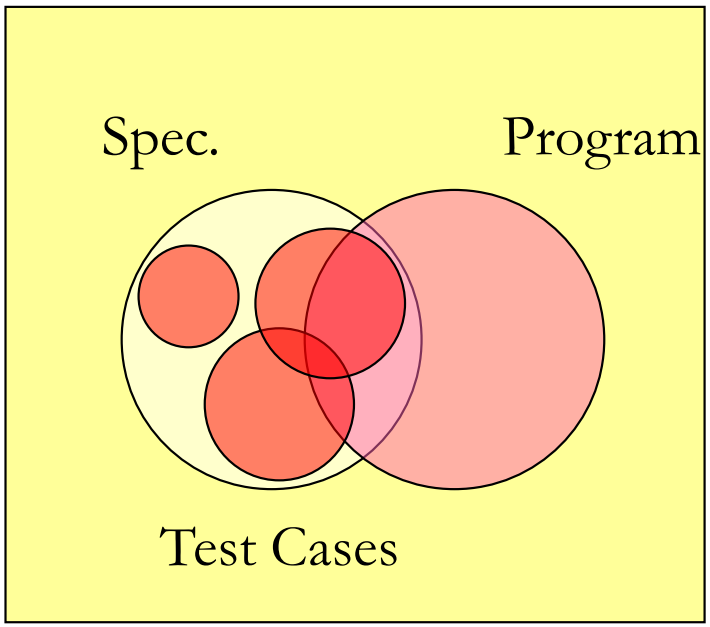




Test cases for black-box testing

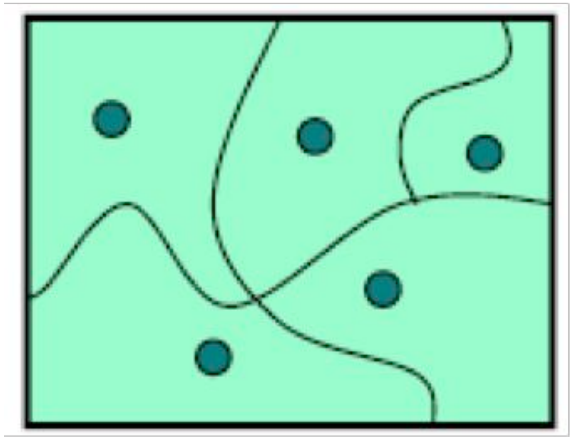
- **Test cases for black-box testing are built around specifications and requirements, i.e., what the application is supposed to do. 检查程序是否符合规约**
  - Test cases are generally derived from external descriptions of the software, including specifications, requirements and design parameters.
  - Pick a set of test cases that is small enough to run quickly, yet large enough to validate the program. 用尽可能少的测试用例，尽快运行，并尽可能大的发现程序的错误





哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

6.1 Choosing Test Cases by Partitioning

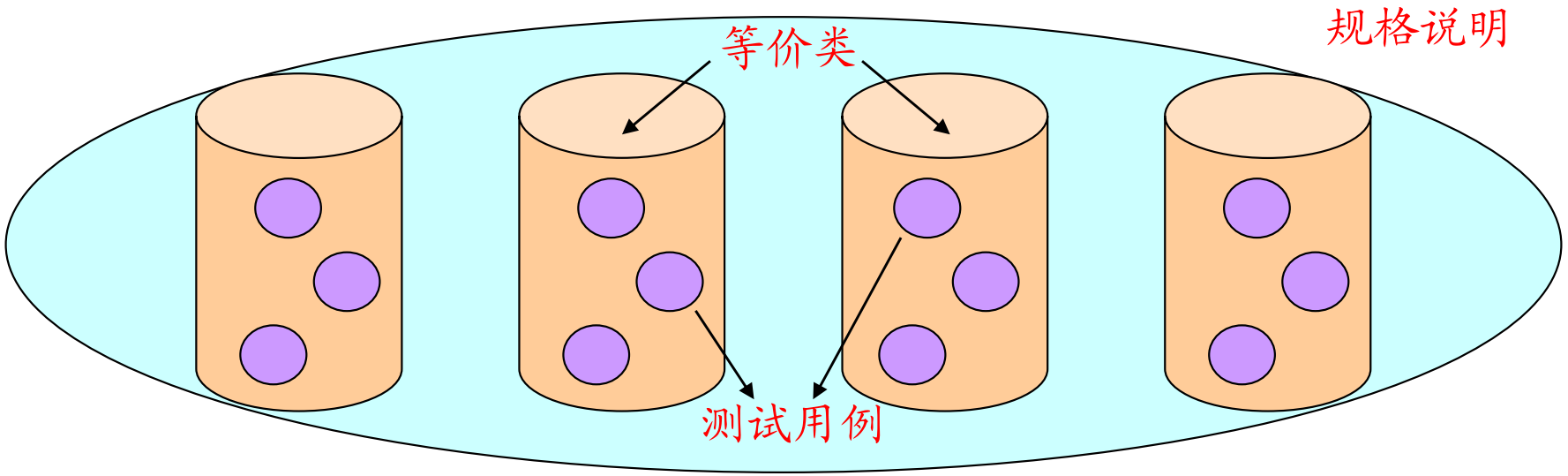


Equivalence Partitioning 等价类划分

- **Equivalence partitioning** is a testing method that divides the input domain of a program into classes of data from which test cases can be derived. 基于等价类划分的测试：将被测函数的输入域划分为等价类，从等价类中导出测试用例。
- **Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.** 针对每个输入数据需要满足的约束条件，划分等价类
  - If a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present. 对称、传递、自反
- **An equivalence class represents a set of valid or invalid states for input conditions.** 每个等价类代表着对输入约束加以满足/违反的有效/无效数据的集合
  - Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

Equivalence Partitioning

- **The idea behind equivalence classes** is to partition the input space into sets of similar inputs on which the program has similar behavior, then use one representative of each set. 基于的假设：相似的输入，将会展示相似的行为。故可从每个等价类中选一个代表作为测试用例即可
  - This approach makes the best use of limited testing resources by choosing dissimilar test cases, and forcing the testing to explore parts of the input space that random testing might not reach. 从而可以降低测试用例数量



Guidelines for Equivalence Partitioning

- **Equivalence classes may be defined according to the guidelines:**
  - If an input condition specifies a range, one valid and two invalid equivalence classes are defined. 输入数据限定了数值范围，则…
  - If an input condition requires a specific value, one valid and one invalid equivalence classes are defined. 输入数据指明了特定的值，则…
  - If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined. 输入数据确定了一组数值，则…
  - If an input condition is Boolean, one valid and one invalid class are defined. 输入数据是Y/N，则…

n是正奇数：
正数：>0、<0
奇数：奇数、偶数
奇数隐含着整数：整数、非整数

输入的学号no需满足的条件：
- 长度为10位：10、>10、<10
- 以118开头：以此开头、以其他开头
- 之后两位数应为03/36/37：03、36、37、其他

Example: `BigInteger.multiply()`

- **`BigInteger` is a class built into the Java library that can represent integers of any size, unlike the primitive types `int` and `long` that have only limited ranges.**
- **`BigInteger` has a method multiply that multiplies two `BigInteger` values together:**

```
/**
* @param val another BigInteger
* @return a BigInteger whose value is (this * val).
*/
public BigInteger multiply(BigInteger val)

E.g.,
BigInteger a = ...;
BigInteger b = ...;
BigInteger ab = a.multiply(b);
```

Example: `BigInteger.multiply()`

- **We should think of multiply as a function taking two inputs:**

  `BigInteger` × `BigInteger` → `BigInteger`

- **So we have a two-dimensional input space 二维输入空间, consisting of all the pairs of integers `(a,b)`.**
- **We might start with these partitions 从正负的角度对二维空间进行等价类划分**
  - `a` and `b` are both positive
  - `a` and `b` are both negative
  - `a` is positive, `b` is negative
  - `a` is negative, `b` is positive

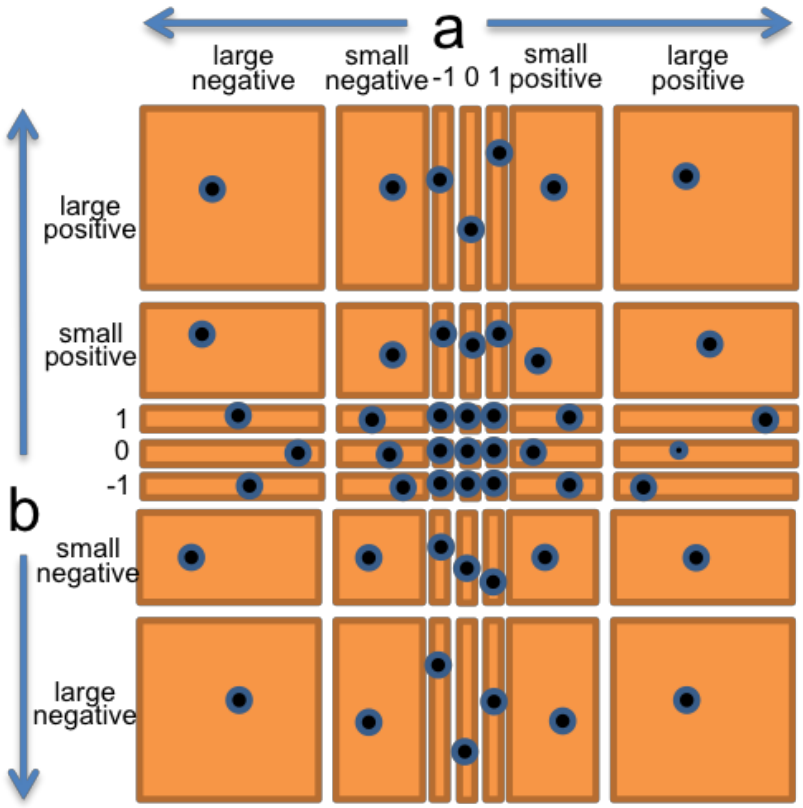
Example: `BigInteger.multiply()`

- **There are also some special cases for multiplication that we should check: 0, 1, and -1.** 需要考虑输入数据的特殊情况
  - `a` or `b` is `0`, `1`, or `-1`
- **Finally, as a suspicious tester trying to find bugs, we might suspect that the implementor of `BigInteger` might try to make it faster by using `int` or long internally when possible, and only fall back to an expensive general representation (like a list of digits) when the value is too big.** 考虑输入的上限：很大的数是否仍正确？
- **So we should definitely also try integers that are very big, bigger than the biggest `long`.**
  - a or b is small
  - the absolute value of `a` or `b` is bigger than `Long.MAX_VALUE` (the biggest possible primitive integer in Java, which is roughly 2^63)

Boundary Value Analysis (BVA)

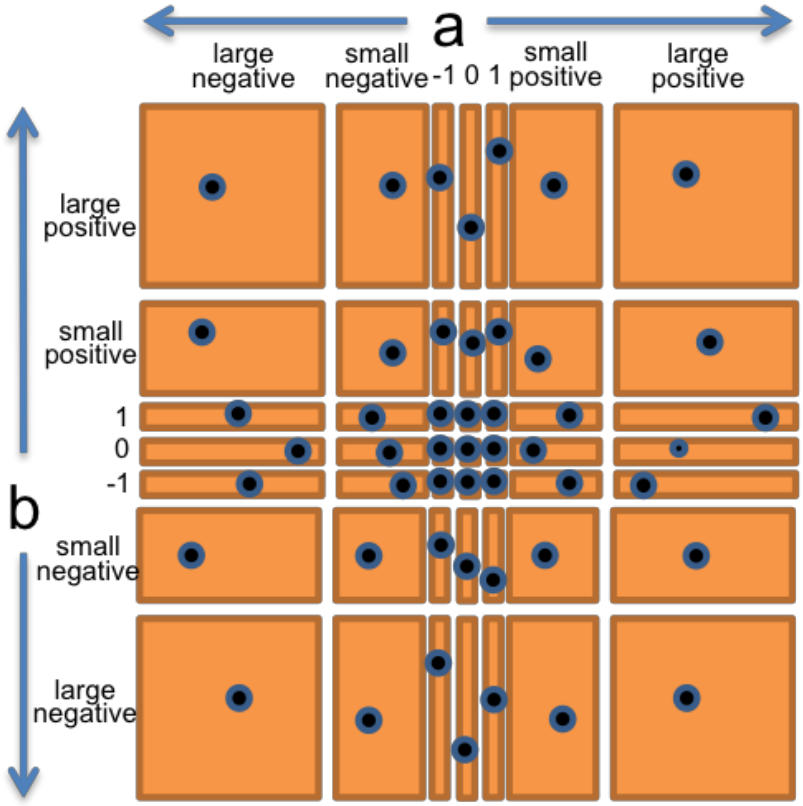
Example: `BigInteger.multiply()`

- **Let's bring all these observations together into a straightforward partition of the whole `(a,b)` space.**
- **We'll choose `a` and `b` independently from:**
  - `0`
  - `1`
  - `-1`
  - small positive integer
  - small negative integer
  - huge positive integer
  - huge negative integer
- **So this will produce `7*7=49` partitions that completely cover the space of pairs of integers.**



Example: `BigInteger.multiply()`

- **To produce the test suite, we would pick an arbitrary pair (a,b) from each square of the grid, for example:**
  - `(a,b) = (-3, 25)` to cover (small negative, small positive)
  - `(a,b) = (0, 30)` to cover (0, small positive)
  - `(a,b) = (2^100, 1)` to cover (large positive, 1)
  - etc.
- **The points are test cases that we might choose to completely cover the partition.**

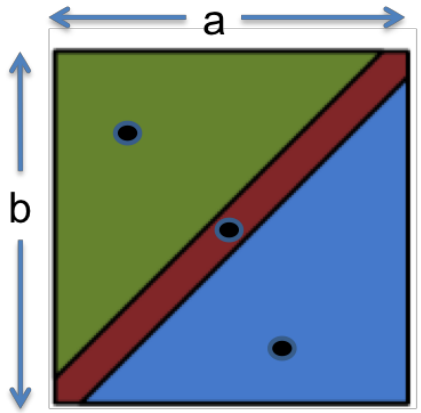


Example: `max()`

- **Another example from the Java library: the integer `max()` function, found in the `Math` class.**

```
/**
* @param a an argument
* @param b another argument
* @return the larger of a and b.
*/
public static int max(int a, int b)
```

  - `max : int × int → int`
  - From the specification, it makes sense to partition this function as:
    - `a < b`
    - `a = b`
    - `a > b`
  - Our test suite might then be:
    - `(a, b) = (1, 2)` to cover `a < b`
    - `(a, b) = (9, 9)` to cover `a = b`
    - `(a, b) = (-5, -6)` to cover `a > b`



Exercises

```
/**
 * Reverses the end of a string.
 *
 *                          012345                    012345
 * For example: reverseEnd("Hello, world", 5) returns "Hellodlrow ,"
 *                               <----->                  <----->
 *
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text    non-null String that will have its end reversed
 * @param start   the index at which the remainder of the input is reversed,
 *                requires 0 <= start <= text.length()
 * @return input text with the substring from start to the end of the string
 *                reversed
 */
public static String reverseEnd(String text, int start)
```

- start = 0, start = 5, start = 100
- start < 0, start = 0, start > 0
- start = 0, 0 < start < text.length(), start = text.length()
- start < text.length(), start = text.length(), start > text.length()

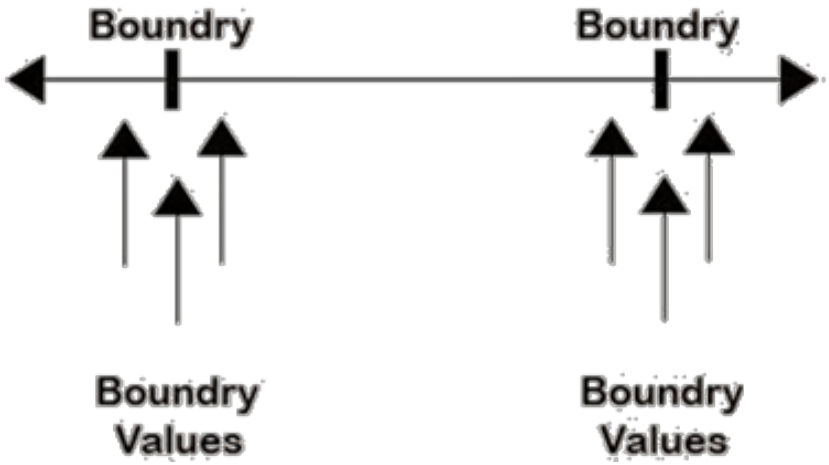
Exercises

```
/**
 * Reverses the end of a string.
 *
 *                            012345                        012345
 * For example: reverseEnd("Hello, world", 5) returns "Hellodlrow ,"
 *                                <----->                       <----->
 *
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text    non-null String that will have its end reversed
 * @param start   the index at which the remainder of the input is reversed,
 *                requires 0 <= start <= text.length()
 * @return input text with the substring from start to the end of the string
 *                reversed
 */
public static String reverseEnd(String text, int start)
```

Which of the following are reasonable partitions for the text parameter?

- text contains some letters; text contains no letters, but some numbers; text contains neither letters nor numbers
- text.length() = 0; text.length() > 0
- text.length() = 0; text.length()-start is odd; text.length()-start is even
- text is every possible string from length 0 to 100

6.2 Include Boundaries in the Partition



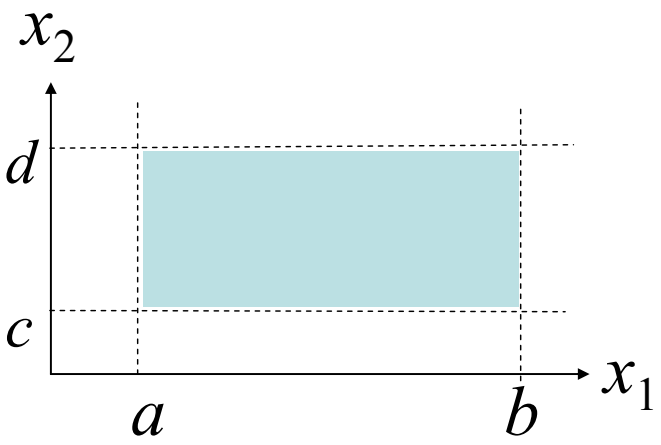
Boundary Value Analysis

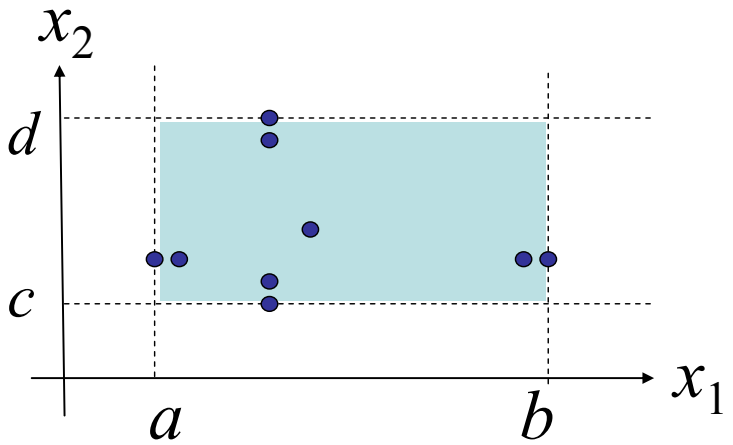
- **A greater number of errors occurs at the boundaries of the input domain rather than in the "center": 大量的错误发生在输入域的"边界"而非中央**
  - 0 is a boundary between positive numbers and negative numbers
  - Maximum and minimum values of numeric types, like `int` and `double`
  - Emptiness (the empty string, empty list, empty array) for collection types
  - The first and last element of a collection
- **Boundary Value Analysis (BVA) has been developed as a testing technique, leading to a selection of test cases on bounding values.**
  - A test-case design technique that complements equivalence partitioning.
  - Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.
- **边界值分析方法是对等价类划分方法的补充**

Why do bugs often happen at boundaries?

- **Reasons**
  - Programmers often make off-by-one mistakes (like writing <= instead of <, or initializing a counter to 0 instead of 1). 程序员经常犯一些大小差1的错误
  - Some boundaries may need to be handled as special cases in the code. 某些边界值是“特殊情况”，需要特殊处理
  - Boundaries may be places of discontinuity in the code's behavior. When an `int` variable grows beyond its maximum positive value, for example, it abruptly becomes a negative number. 程序的行为在边界的地方可能发生“突变”
- **It's important to include boundaries as subdomains in your partition, so that you're choosing an input from the boundary.** 在等价类划分时，将边界作为等价类之一加入考虑

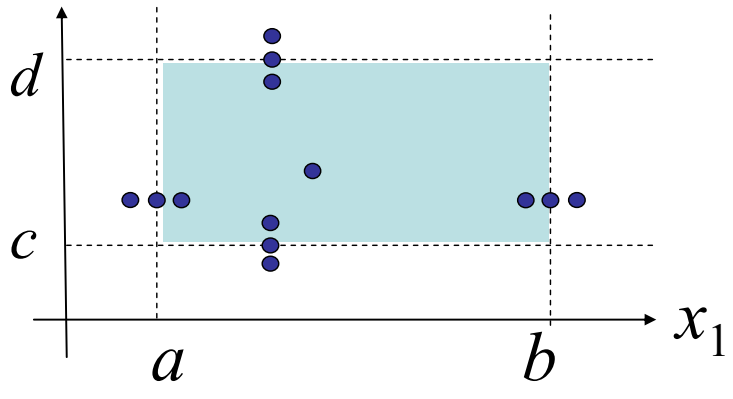
Guidelines for Boundary Value Analysis





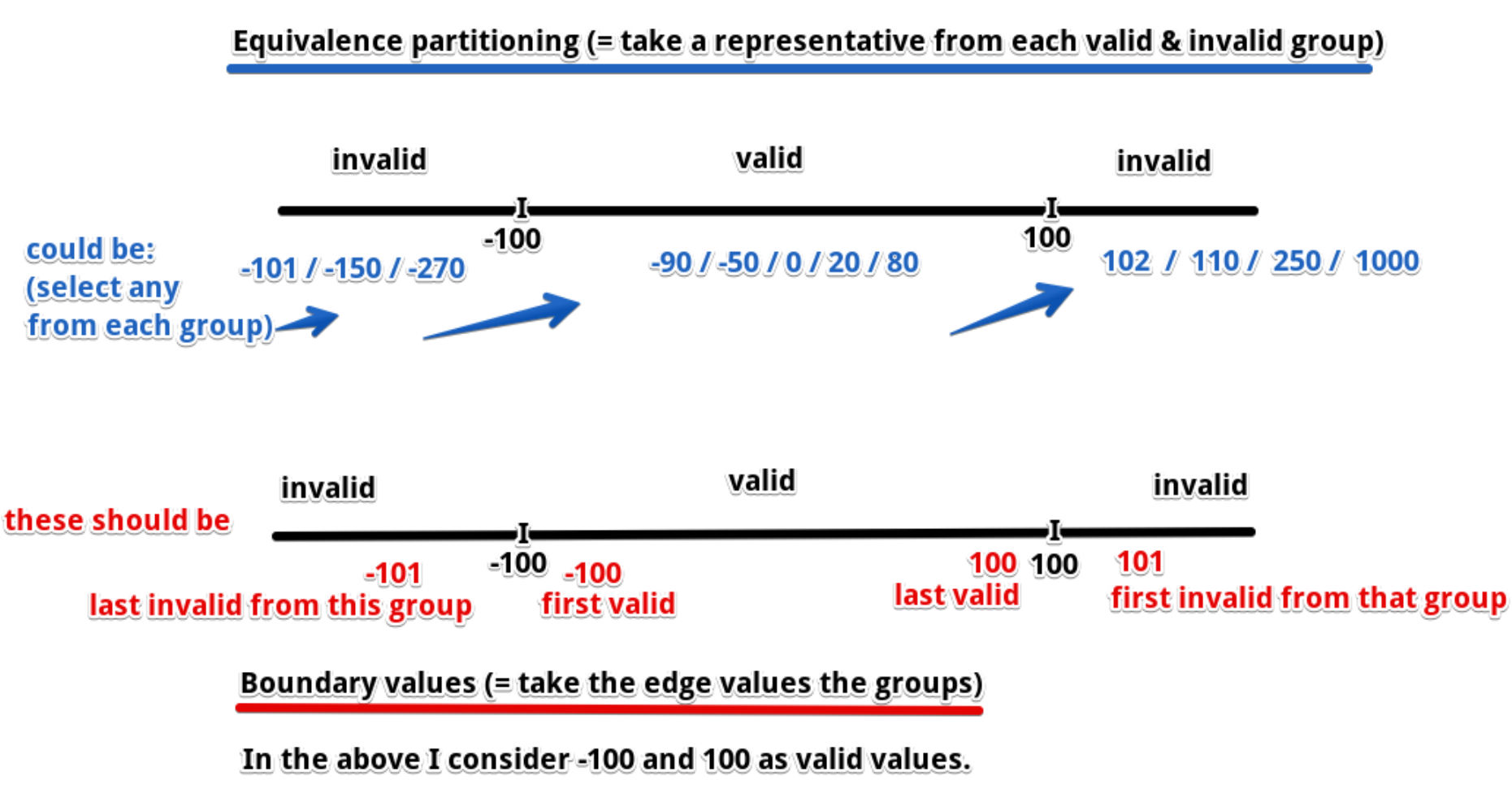
5个测试用例:
最小值；略高于最小值；
正常值；
略低于最大值；最大值；

n个变量，有4n+1个测试用例



在5个测试用例的基础上增如:
略高于最大值；
略低于最小值；

Combine Partitioning and BVA



Example of BVA

AGE Enter Age *Accepts value 18 to 56

BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	**Valid (min, +min, -max, max)**	**Invalid (max +1)**
17	18, 19, 55, 56	57

Name Enter Name *Accepts characters length (6 - 12)

BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	**Valid (min, +min, -max, max)**	**Invalid (max +1)**
5 characters	6, 7, 11, 12 characters	13 characters

Example: `max()`

- **Let's redo** `max : int × int → int.`
- **Partition into:**
  - Relationship between `a` and `b`
    - `a < b`
    - `a = b`
    - `a > b`
  - Value of `a`
    - `a = 0`
    - `a < 0`
    - `a > 0`
    - `a =` minimum integer
    - `a =` maximum integer
  - Value of `b`
    - `b = 0`
    - `b < 0`
    - `b > 0`
    - `b =` minimum integer
    - `b =` maximum integer

Example: `max()`

- **Now let's pick test values that cover all these classes:**
  - `(1, 2)` covers a < b, a > 0, b > 0
  - `(-1, -3)` covers `a > b, a < 0, b < 0`
  - `(0, 0)` covers `a = b, a = 0, b = 0`
  - (`Integer.MIN_VALUE`, `Integer.MAX_VALUE`) covers `a < b, a = minint, b = maxint`
  - (`Integer.MAX_VALUE`, `Integer.MIN_VALUE`) covers `a > b, a = maxint, b = minint`

Two Extremes for Covering the Partition

- **Full Cartesian product** 笛卡尔积：全覆盖 使用笛卡尔积做到完备的测试用例
  - Every legal combination of the partition dimensions is covered by one test case. 多个划分维度上的多个取值，要组合起来，每个组合都要有一个用例
  - For the `max` example that included boundaries, which has three dimensions with 3 parts, 5 parts, and 5 parts respectively, it would mean up to 3 × 5 × 5 = 75 test cases.
  - In practice not all of these combinations are possible. For example, there's no way to cover the combination `a<b, a=0, b=0`, because a can't be simultaneously less than zero and equal to zero. 并非所有组合情况都可能
- **Cover each part** 覆盖每个取值：最少1次即可
  - Every part of each dimension is covered by at least one test case, but not necessarily every combination. 每个维度的每个取值至少被1个测试用例覆盖一次即可
  - The test suite for `max` might be as small as 5 test cases if carefully chosen.

Two Extremes for Covering the Partition

- **Full Cartesian product** 笛卡尔积：全覆盖
- **Cover each part** 覆盖每个取值：最少1次即可

输入的学号no需满足的条件：
- 长度为10位：10、>10、<10
- 以119开头：以此开头、以其他开头
- 之后两位数应为03/36/37：03、36、37、其他

- 前者：测试完备，但用例数量多，测试代价高
- 后者：测试用例少，代价低，但测试覆盖度未必高。
- **Often we strike some compromise between these two extremes, based on human judgement and caution, and influenced by whitebox testing and code coverage tools.**

Two Extremes for Covering the Partition

输入的学号no需满足的条件：

- 长度为10位：10、>10、<10
- 以119开头：以此开头、以其他开头
- 之后两位数应为03/36/37：03、36、37、其他

覆盖每个取值：最少1次即可

等价类	测试用例
长度为10	1190311111
长度>10	11111111111
长度<10	119361111
以119开头	1190311111
不以119开头	11111111111
之后两位是03	1190311111
之后两位是36	119361111
之后两位是37	1193711111
之后两位是其他	11111111111

笛卡尔积：全覆盖

长度=10	以119开头	之后两位是03	1190300000
		之后两位是36	1193600000
		之后两位是37	1193700000
		之后两位是其他	1194000000
	不以119开头	之后两位是03	1200300000
		之后两位是36	1203600000
		之后两位是37	1203700000
		之后两位是其他	1204000000
长度>10	以119开头	之后两位是03	11903000001
		之后两位是36	11936000001
		之后两位是37	11937000001
		之后两位是其他	11940000001
	不以119开头	之后两位是03	12003000001
		之后两位是36	12036000001
		之后两位是37	12037000001
		之后两位是其他	12040000001
长度<10	以119开头	之后两位是03	119030000
		之后两位是36	119360000
		之后两位是37	119370000
		之后两位是其他	119400000
	不以119开头	之后两位是03	120030000
		之后两位是36	120360000
		之后两位是37	120370000
		之后两位是其他	120400000

Two Extremes for Covering the Partition

输入的学号no需满足的条件：

- 长度为10位：10、>10、<10
- 以119开头：以此开头、以其他开头
- 之后两位数应为03/36/37：03、36、37、其他

覆盖每个取值：最少1次即可

等价类	测试用例
长度为10	1190311111
长度>10	11111111111
长度<10	119361111
以119开头	1190311111
不以119开头	11111111111
之后两位是03	1190311111
之后两位是36	119361111
之后两位是37	1193711111
之后两位是其他	11111111111

笛卡尔积：全覆盖

长度=10	以119开头	之后两位是03	1190300000
		之后两位是36	1193600000
		之后两位是37	1193700000
		之后两位是其他	1194000000
	不以119开头	之后两位是03	1200300000
		之后两位是36	1203600000
		之后两位是37	1203700000
		之后两位是其他	1204000000
长度>10	以119开头	之后两位是03	11903000001
		之后两位是36	11936000001
		之后两位是37	11937000001
		之后两位是其他	11940000001
	不以119开头	之后两位是03	12003000001
		之后两位是36	12036000001
		之后两位是37	12037000001
		之后两位是其他	12040000001
长度<10	以119开头	之后两位是03	119030000
		之后两位是36	119360000
		之后两位是37	119370000
		之后两位是其他	119400000
	不以119开头	之后两位是03	120030000
		之后两位是36	120360000
		之后两位是37	120370000
		之后两位是其他	120400000

Two Extremes for Covering the Partition

```
int max(int a, int b)
```

笛卡尔积：全覆盖

a<b	a = 0	b = 0	不存在
		b < 0	不存在
		b > 0	a=0, b=5
		b = minimum integer	不存在
		b = maximum integer	a=0, b=Integer.MAX_VALUE
	a < 0	b = 0	a=-5, b=0
		b < 0	a=-5, b=-4
		b > 0	a=-5, b=5
		b = minimum integer	不存在
		b = maximum integer	a=-5, b=Integer.MAX_VALUE
	a > 0	b = 0	不存在
		b < 0	不存在
		b > 0	a=5, b=6
		b = minimum integer	不存在
		b = maximum integer	a=5, b=Integer.MAX_VALUE
	a = minimum integer	b = 0	a=Integer.MIN_VALUE, b=0
		b < 0	a=Integer.MIN_VALUE, b=-5
		b > 0	a=Integer.MIN_VALUE, b=5
		b = minimum integer	不存在
		b = maximum integer	a=Integer.MIN_VALUE, b=Integer.MAX_VALUE
	a = maximum integer	b = 0	
		b < 0	
		b > 0	
		b = minimum integer	
		b = maximum integer	
a=b	…	…	…
a>b	…	…	…

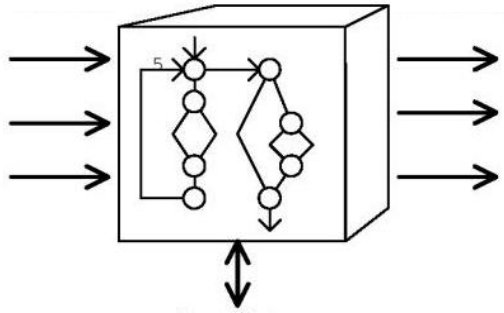
7 White-box Testing

Black-box vs. White-box testing

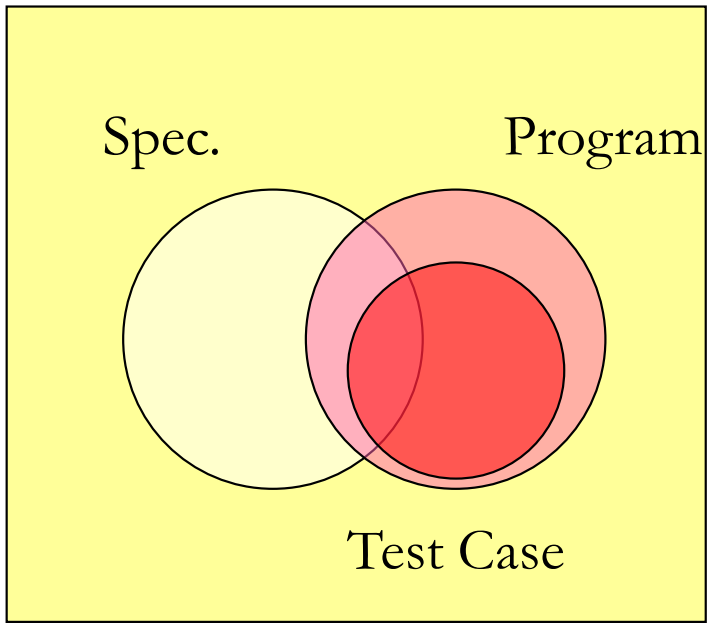
- **Blackbox testing means choosing test cases only from the specification, not the implementation of the function.** 黑盒测试完全从函数spec导出测试用例，不考虑函数内部实现
  - We partitioned and looked for boundaries in `multiply` and `max` without looking at the actual code for these functions.

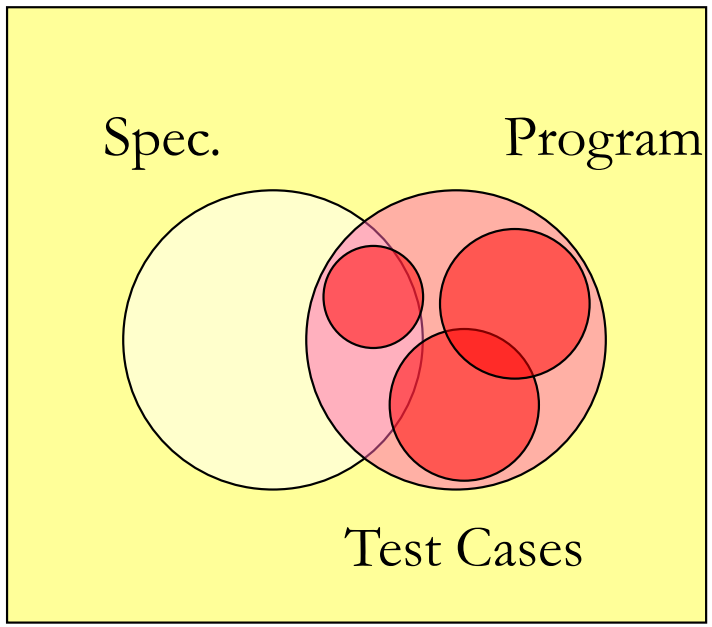
- **Whitebox testing (also called glass box testing) means choosing test cases with knowledge of how the function is actually implemented.** 白盒测试要考虑内部实现细节
  - For example, if the implementation selects different algorithms depending on the input, then you should partition according to those domains.
  - If the implementation keeps an internal cache that remembers the answers to previous inputs, then you should test repeated inputs.

White-box testing

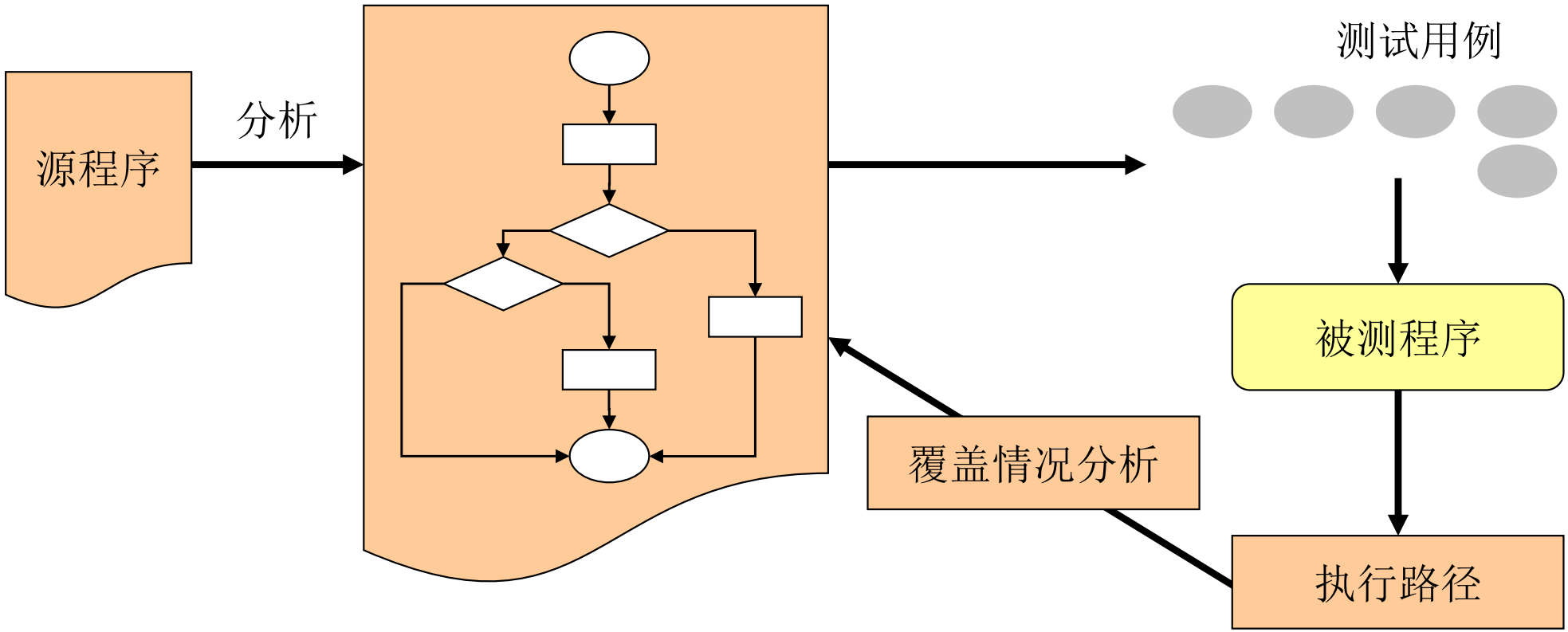


- **An internal perspective of the system, as well as programming skills, are used to design test cases.**
- **The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. 根据程序执行路径设计测试用例**
- **White-box testing can be applied at the unit, integration and system levels of the software testing process. In general, it is performed early in the testing process. 白盒测试一般较早执行**





White-box testing



Exercise

```
/**
 * Sort a list of integers in nondecreasing order.
 * Modifies the list so that
 * values.get(i) <= values.get(i+1) for all 0<=i<values.length()-1
 */
public static void sort(List<Integer> values) {
    // choose a good algorithm for the size of the list
    if (values.length() < 10) {
        radixSort(values);
    } else if (values.length() < 1000*1000*1000) {
        quickSort(values);
    } else {
        mergeSort(values);
    }
}
```

Which of the following test cases are likely to be boundary values produced by white box testing?

```
values = [] (the empty list)
values = [1, 2, 3]
values = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
values = [0, 0, 1, 0, 0, 0, 0]
```

White-box testing

- **Using white-box testing methods, you can derive test cases that**
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides,
  - Execute all loops at their boundaries and within their operational bounds,
  - Exercise internal data structures to ensure their validity.

- **A typical white-box testing method is called "independent/basis path testing"** 独立/基本路径测试：对程序所有执行路径进行等价类划分，找出有代表性的最简单的路径(例如循环只需执行1次)，设计测试用例使每一条基本路径被至少覆盖1次。

8 Coverage of Testing

Code coverage 代码覆盖度

- **Testing should consider the code coverage of the test case for the internal logic of the program.**
- **Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. 代码覆盖度：已有的测试用例有多大程度覆盖了被测程序**
  - A program with high code coverage, measured as a percentage (通常用百分比衡量覆盖度), has had more of its source code executed during testing which suggests it has a lower chance of containing undetected software bugs compared to a program with low code coverage.
  - Many different metrics can be used to calculate code coverage; some of the most basic are the percentage of program subroutines and the percentage of program statements called during execution of the test suite.

代码覆盖度越低，测试越不充分
但要做到很高的代码覆盖度，需要更多的测试用例，测试代价高

Code coverage 代码覆盖度

- **One way to judge a test suite is to ask how thoroughly it exercises the program. This notion is called coverage.**
- **There are many common kinds of coverage:**
  - Function coverage: has each function in program been called? 函数覆盖
  - Statement coverage: is every statement run by some test case? 语句覆盖
  - Branch coverage: for every `if` or `while` or `switch-case` or `for` statement in the program, are both the true and the false direction taken by some test case? 分支覆盖
  - Condition coverage: for every condition in `if/while/for/switch-case` statement, are both the true/false direction taken by some test case? 条件覆盖
  - Path coverage: is every possible combination of branches — every path through the program — taken by some test case? 路径覆盖
  - 感兴趣请自行阅读参考书，学习各类白盒测试方法

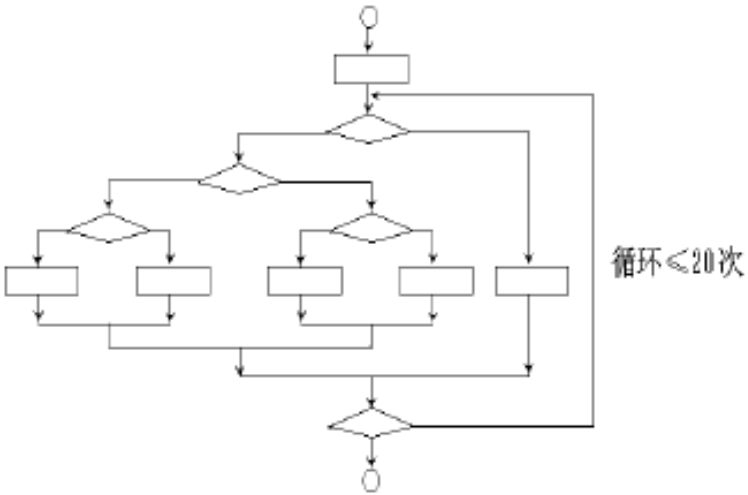
Code Coverage

- **Branch coverage** is stronger (requires more tests to achieve) than **statement coverage**, and **path coverage** is stronger than **branch coverage**. **测试效果：路径覆盖>分支覆盖>语句覆盖**
- In industry, **100% statement coverage** is a common goal, but even that is rarely achieved due to unreachable defensive code (like "should never get here" assertions).
- **100% branch coverage** is highly desirable, and safety critical industry code has even more arduous criteria.
- Unfortunately **100% path coverage** is infeasible, requiring exponential-size test suites to achieve. **测试难度：路径覆盖>分支覆盖>语句覆盖**

- 在不同的软件类型中、不同的公司中，对达到何种标准的覆盖度都有不同的要求

Code coverage 代码覆盖度

- **The most thorough white-box method is to cover every path in the program 路径覆盖, but because the program generally contains a loop, so the number of paths is great. 但路径数量巨大，难以全覆盖**
  - It is almost impossible to execute every path, and we can only try to ensure that the coverage is as high as possible.
- **An example:**
  - A program contains a loop that needs to be executed 20 times. It includes 5^{20} different execution paths. Supposing that it takes 1 ms to test each path, it will take 3170 years to finish testing all the paths.



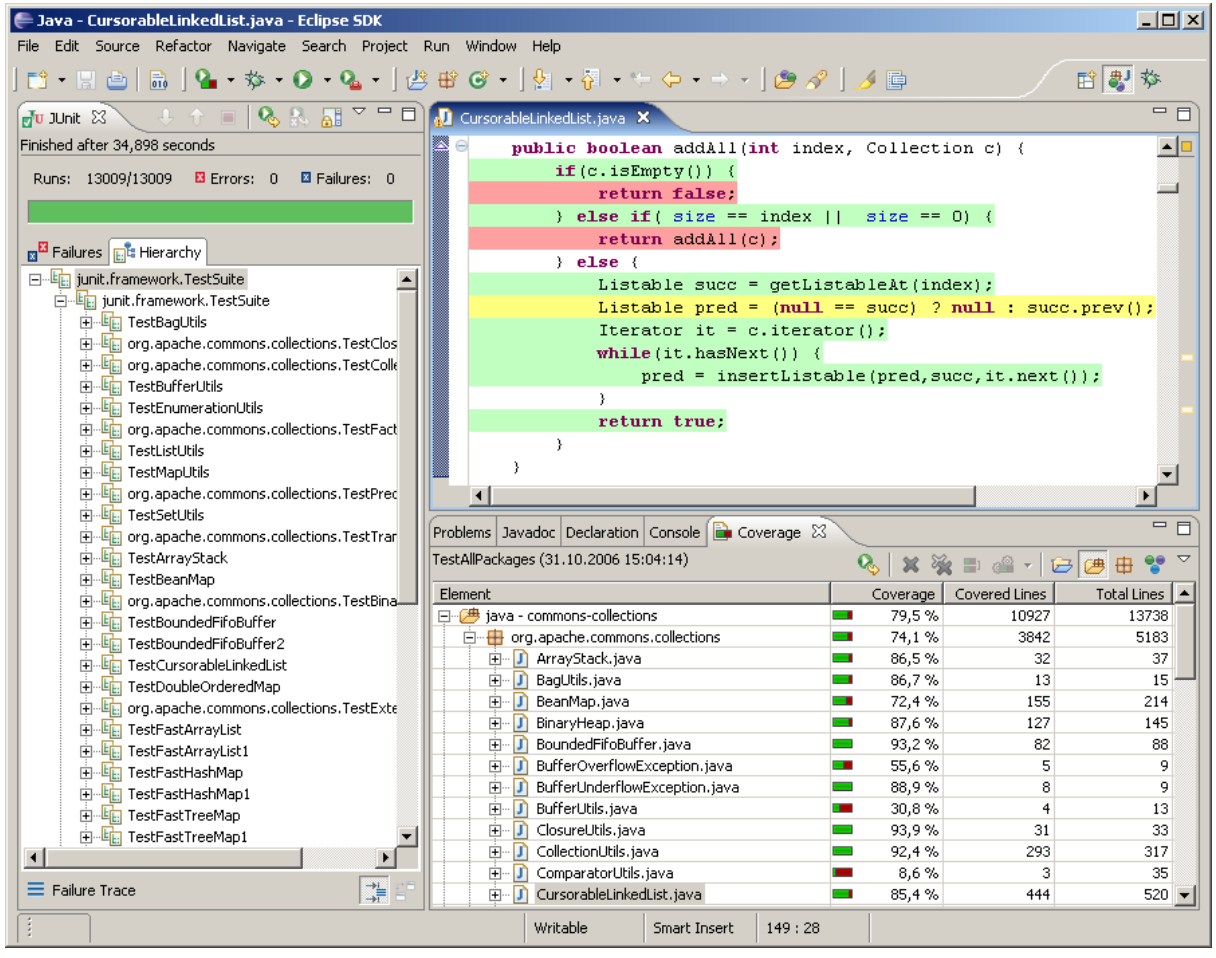
Code Coverage

- A standard approach to testing is to add tests until the test suite achieves adequate coverage: i.e., so that every reachable statement in the program is executed by at least one test case.
- In practice, coverage is usually measured by a code coverage tool, which counts the number of times each statement is run by your test suite.
- With such a tool, white box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed.

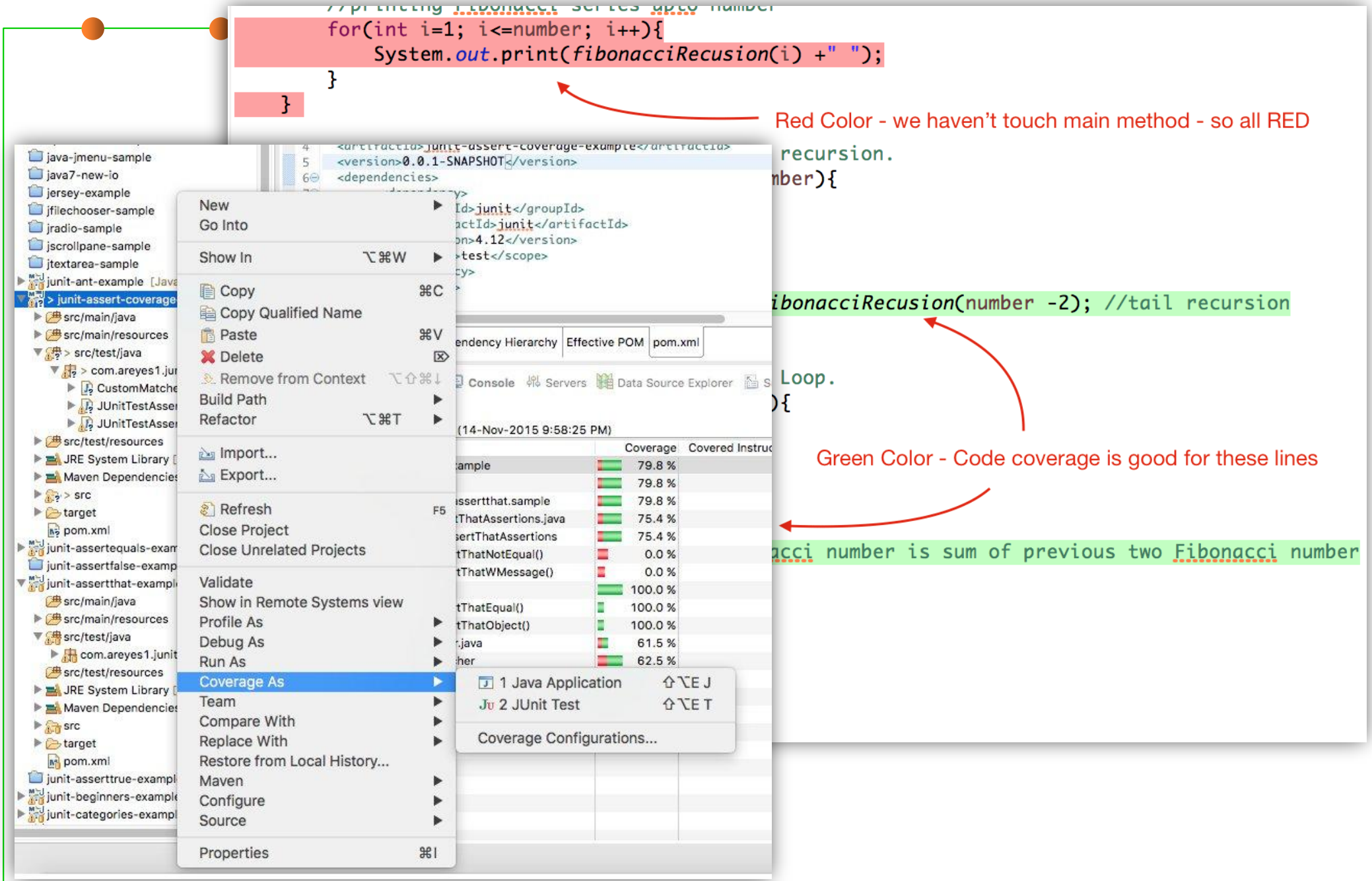
- 实际当中，根据预先设定的覆盖度标准，逐步增加测试用例的数量，直到覆盖度达到标准（例如语句覆盖100%、路径覆盖90%）。

EclEmma

- **A good code coverage tool for Eclipse is EclEmma**
  - http://www.eclemma.org
  - http://www.ibm.com/developerworks/cn/java/j-lo-eclemma
- Lines that have been executed by the test suite are colored **green**, and lines not yet covered are **red**.
- The next step is to come up with a test case that causes the red lines to execute, and add it to your test suite so that the red lines become green.



EclEmma's code coverage analysis



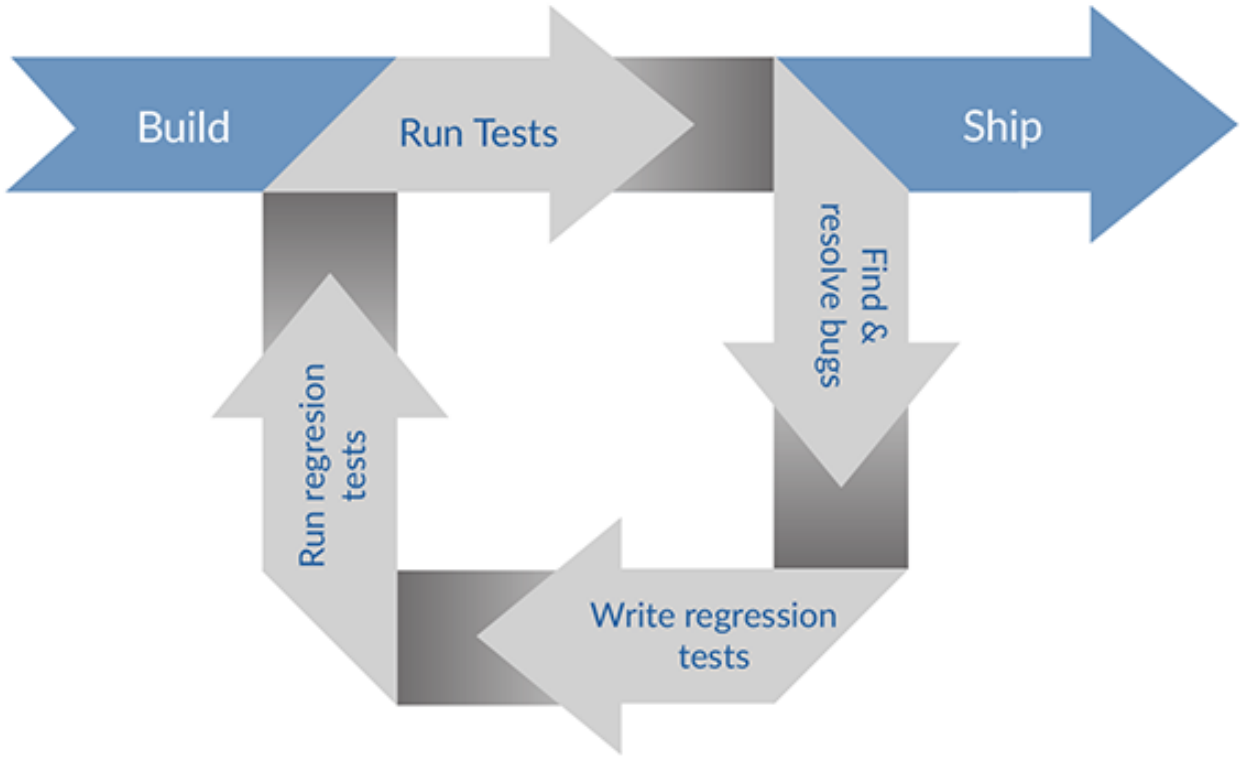
9 Automated Testing and Regression Testing

Automated testing

- Nothing makes tests easier to run, and more likely to be run, than **complete automation**. 手工测试的代价太高，最好达到完全的自动化
- **Automated testing** means running the tests and checking their results automatically. 自动调用被测函数、自动判定测试结果、自动计算覆盖度
  - A test driver should not be an interactive program that prompts you for inputs and prints out results for you to manually check.
  - Instead, a test driver should invoke the module itself on fixed test cases and automatically check that the results are correct.
  - The result of the test driver should be either "all tests OK" or "these tests failed: …"
- A good testing framework, like JUnit, helps you build automated test suites.

Automated testing vs. Automatic test generation

- **Note that automated testing frameworks like JUnit make it easy to run the tests, but you still have to come up with good test cases yourself. 只是“测试用例的自动执行”，并非“自动生成测试用例”**
- **Automatic test generation is a hard problem, still a subject of active computer science research.**



Regression testing 回归测试

- **Once you have test automation, it's very important to rerun your tests when you modify your code.** **回归测试：一旦程序被修改，重新执行之前的所有测试**
  - Software engineers know from painful experience that *any* change to a large or complex program is dangerous.
  - Whether you're fixing another bug, adding a new feature, or optimizing the code to make it faster, an automated test suite that preserves a baseline of correct behavior – even if it's only a few tests – will save your bacon.
  - Running the tests frequently while you're changing the code prevents your program from *regressing* — introducing other bugs when you fix new bugs or add new features.
- Running all your tests after every change is called **regression testing**.

Regression testing

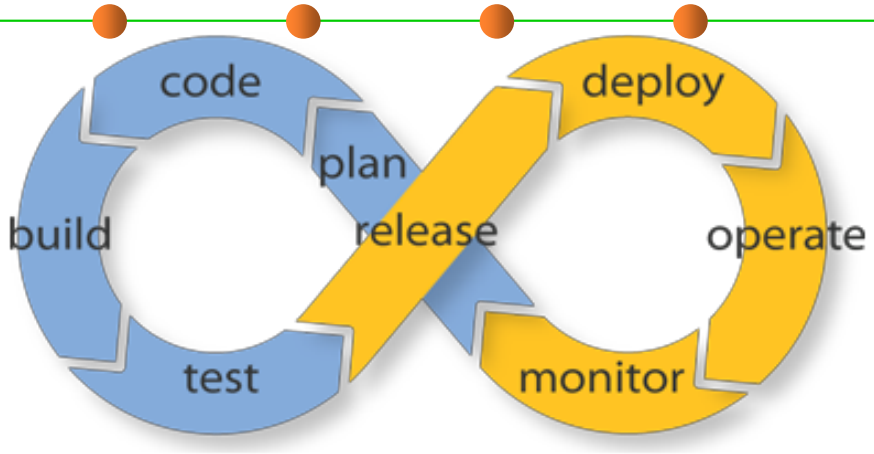
- **Whenever you find and fix a bug, take the input that elicited the bug and add it to your automated test suite as a test case.**
- **This kind of test case is called a *regression test*. This helps to populate your test suite with good test cases.**
- **Remember that a test is good if it elicits a bug — and every regression test did in one version of your code!**
- **Saving regression tests also protects against reversions that reintroduce the bug.**

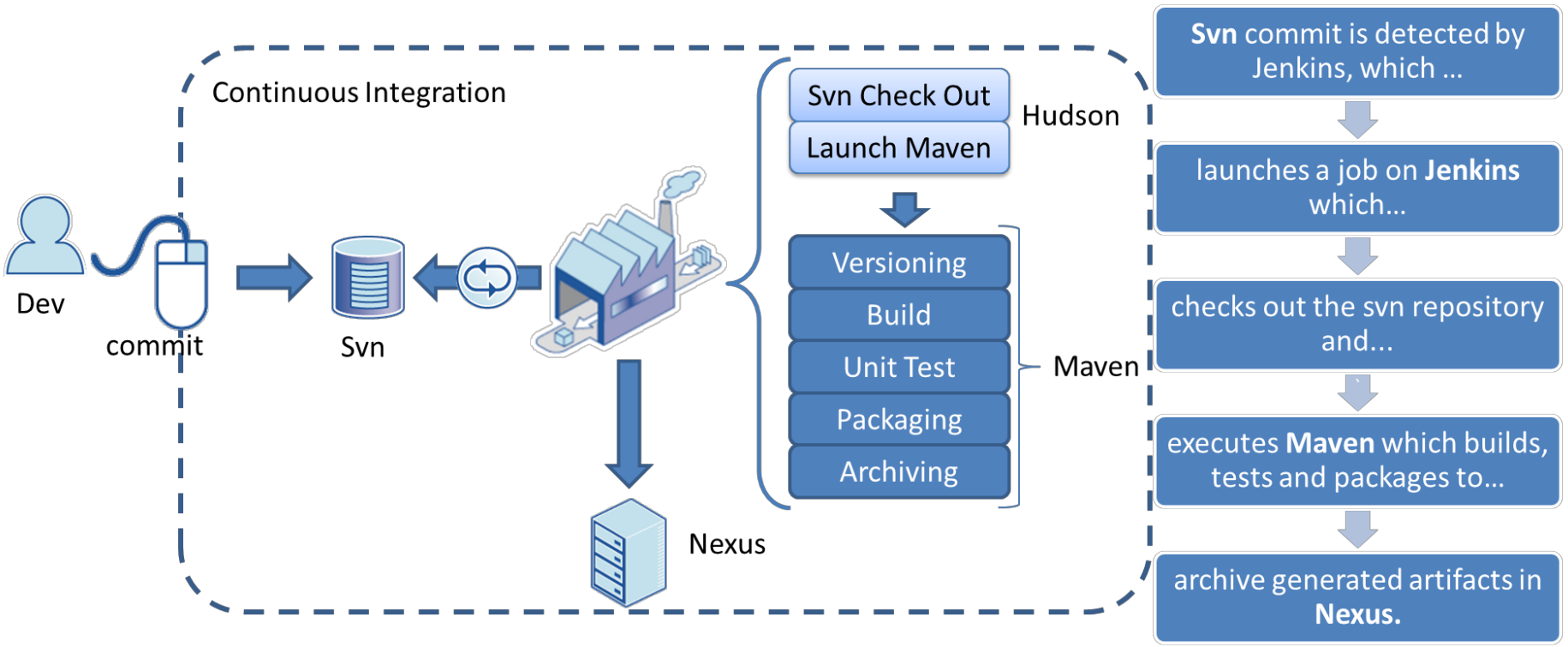
Automated regression testing

- **This idea also leads to *test-first debugging*.**
  - When a bug arises, immediately write a test case for it that elicits it, and immediately add it to your test suite.
  - Once you find and fix the bug, all your test cases will be passing, and you'll be done with debugging and have a regression test for that bug.

  一旦发现bug，要马上写一个可重现该bug的测试用例，并将其加入测试库
- **In practice, automated testing and regression testing, are almost always used in combination.**
  - Regression testing is only practical if the tests can be run often, automatically.
  - Conversely, if you already have automated testing in place for your project, then you might as well use it to prevent regressions.
- **Automated regression testing** is a best-practice of modern software engineering. 自动化回归测试

Continuous Integration





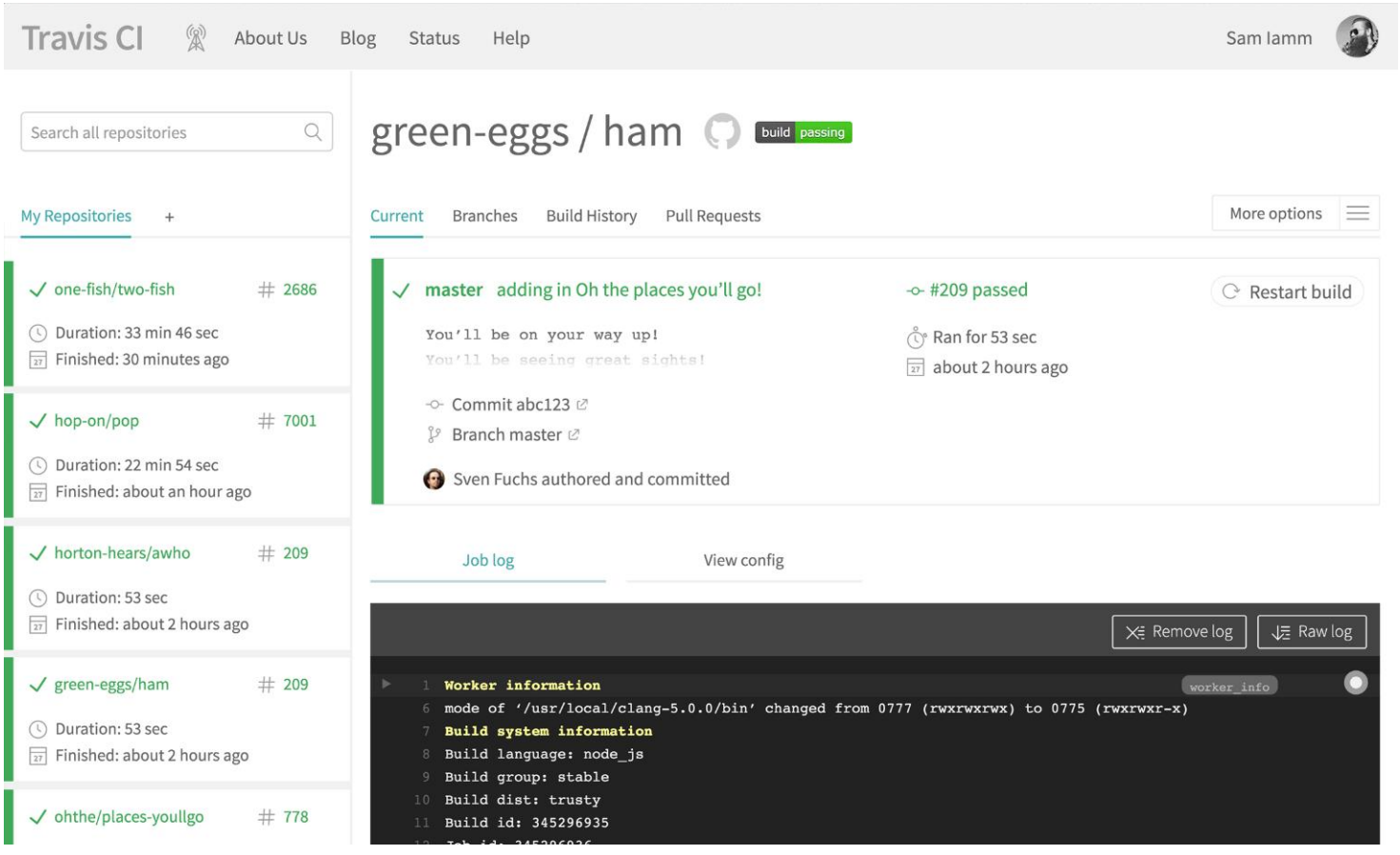
Travis-CI

Travis CI

- https://travis-ci.org

Test and Deploy with Confidence

Easily sync your GitHub projects with Travis CI and you'll be testing your code in minutes!



Exercise

- **Which of the following best defines regression testing?**
  - You should run a suite of tests whenever you change the code.
  - Every component in your code should have an associated set of tests that exercises all the corner cases in its specification.
  - Tests should be written before you write the code as a way of checking your understanding of the specification.
  - When a new test exposes a bug, you should run it on all previous versions of the code until you find the version where the bug was introduced.
- **Which of the following are good times to rerun all your JUnit tests?**
  - Before doing git add/commit/push
  - After rewriting a function to make it faster
  - When using a code coverage tool
  - After you think you fixed a bug

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

10 Documenting Your Testing Strategy

Documenting Testing Strategy

- **Unit testing strategy is a complementary document of ADT's design.** 测试策略（根据什么来选择测试用例）非常重要，需要在程序中显式记录下来
  - Aligning with the idea of test-first programming, it is recommended to write down the testing strategy (such as partitioning and boundary) according to which you design your test cases.
- **The objective is to make code review to check if your testing is sufficient, and to make other developers understand your test.** 目的：在代码评审过程中，其他人可以理解你的测试，并评判你的测试是否足够充分

An example

写Document解释测试用例的来源（也就是写好comment）

```
/**
 * Reverses the end of a string.
 *
 * For example:
 *   reverseEnd("Hello, world", 5)
 *   returns "Hellodlrow ,"
 *
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text    non-null String that will have
 *                its end reversed
 * @param start   the index at which the
 *                remainder of the input is
 *                reversed, requires 0 <=
 *                start <= text.length()
 * @return input text with the substring from
 *               start to the end of the string
 *               reversed
 */
static String reverseEnd(String text, int start)
```

Document the strategy at the top of the test class:

```
/*
* Testing strategy
*
* Partition the inputs as follows:
* text.length(): 0, 1, > 1
* start:         0, 1, 1 < start < text.length(),
*                text.length() - 1, text.length()
* text.length()-start: 0, 1, even > 1, odd > 1
*
* Include even- and odd-length reversals because
* only odd has a middle element that doesn't move.
*
* Exhaustive Cartesian coverage of partitions.
*/
```

Each test method should have a comment above it saying how its test case was chosen, i.e. which parts of the partitions it covers:

并且注释好每一个测试用例cover了哪一个测试部分

```
// covers test.length() = 0,
//        start = 0 = text.length(),
//        text.length()-start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}
```

Summary

Summary of this lecture

- **Test-first programming**. Write tests before you write code.
- **Partitioning and boundaries** for choosing test cases systematically.
- **White box testing and statement coverage** for filling out a test suite.
- **Unit-testing** each module, in isolation as much as possible.
- **Automated regression testing** to keep bugs from coming back.

- **Safe from bugs.** Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, immediately after you introduced them.

The end

March 5, 2024