HARBIN INSTITUTE OF TECHNOLOGY

Chapter 7: Software Construction for Robustness
7.5 Testing and Test-First Programming

Ming Liu

February 28, 2019

Outline

- **Introduction to Software Testing**

- **Test Case**

- **Test-First Programming**

- **Unit Testing**

- **Automated Unit Testing with JUnit**

- **Black-box Testing**
  - Choosing Test Cases by Partitioning
  - Include Boundaries in the Partition

Outline

- **White-box Testing**

- **Coverage of Testing**

- **Integration Testing**

- **Automated Testing and Regression Testing**

- **Documenting Your Testing Strategy**

- **Summary**

Objective of this lecture

- Understand the value of testing, and know the process of test-first programming;

- Be able to design a test suite for a method by partitioning its input and output space and choosing good test cases;

- Be able to judge a test suite by measuring its code coverage; and

- Understand and know when to use blackbox vs. whitebox testing, unit tests vs. integration tests, and automated regression testing.

1 Introduction to Software Testing

Software Validation

- **Testing** is an example of a more general process called *validation*.

- **The purpose of validation** is to uncover problems in a program and thereby increase your confidence in the program's correctness.

- **Validation includes:**

  - **Formal reasoning** about a program, usually called *verification*. Verification constructs a formal proof that a program is correct. Verification is tedious to do by hand, and automated tool support for verification is still an active area of research. (out of scope of this course)

  - **Code review**. Having somebody else carefully read your code, and reason informally about it, can be a good way to uncover bugs. It's much like having somebody else proofread an essay you have written. (to be discussed in Chapter 9)

  - **Testing**. Running the program on carefully selected inputs and checking the results.

Software Validation

- Even with the best validation, **it's very hard to achieve perfect quality in software.**

- Here are some typical **residual defect rates** (bugs left over after the software has shipped) per kloc (one thousand lines of source 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!

Test Characteristics

- **Testing's goal runs counter to the goals of other development activities. The goal is to find errors.**

- **Testing can never completely prove the absence of errors.**

- **Testing by itself does not improve software quality.**

- **Testing requires you to assume that you'll find errors in your code.**

- **A good test has a high probability of finding an error**

- **A good test is not redundant**

- **A good test should be "best of breed"**

- **A good test should be neither too simple nor too complex**

Testing levels

- **Unit testing: refers to tests that verify the functionality of a specific section of code, usually at the function level (in OO: class level).**

- **Integration testing: the combined execution of two or more classes, packages, components, subsystems that have been created by multiple programmers or programming teams.**

- **System testing: to test a completely integrated system to verify that the system meets its requirements, which executes the software in its final configuration.**

Regression testing

Unit testing — Integration testing — System testing

function/class   classes/packages /components/subsystems   system
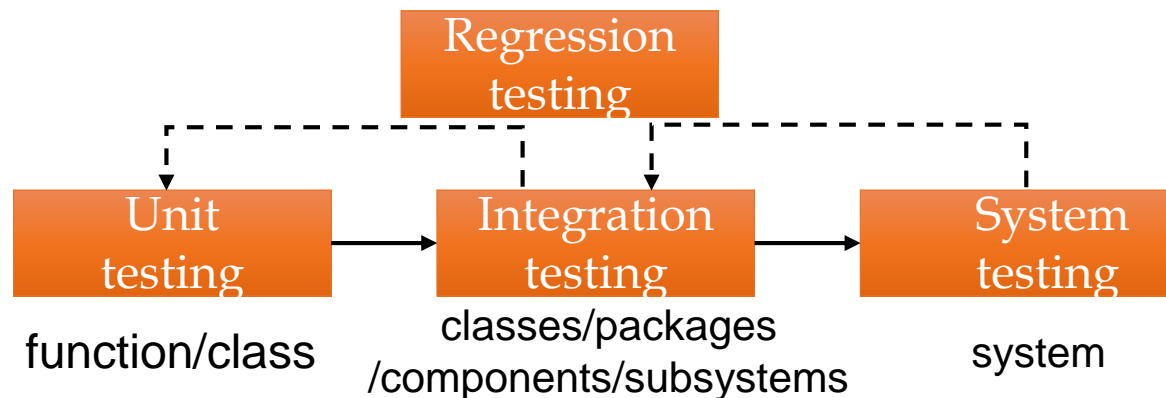
Other testing types

- **Installation testing**

- **Compatibility testing**

- **Smoke and sanity testing**

- **Regression testing**

- **Acceptance testing**

- **Alpha testing**

- **Beta testing**

- **Performance/load/scalability testing**

- **Usability testing**

- **Accessibility testing**

- **Security testing**

- **...**

The whole topic of testing is much larger than the subject of testing during software construction.
**Other topics for test specialists can be studied in other "software engineering" related courses.**

White-box vs. black-box testing

- **White-box testing (also known as clear box testing, glass box testing, transparent box testing and structural testing) tests internal structures or workings of a program by seeing the source code.**

- **Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation, without seeing the source code.**

White-box testing

Input ⟹ ⟹Output
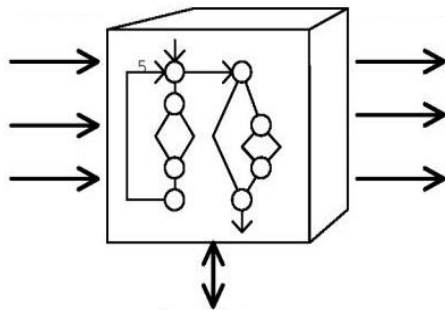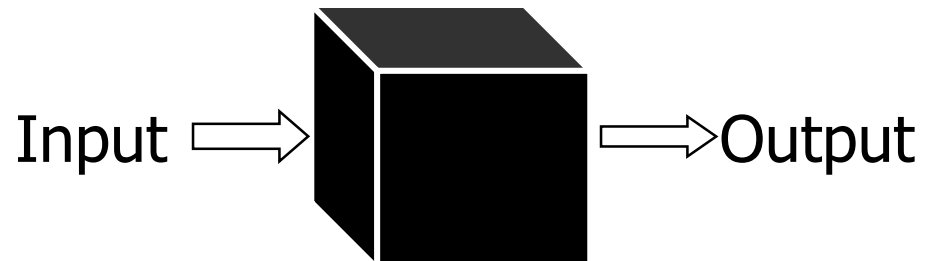
Black-box testing

Testing vs. Debugging

- **Some programmers use the terms "testing" and "debugging" interchangeably, but careful programmers distinguish between the two activities.**

- **Testing is a means of detecting errors.**

- **Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.**

- **This section deals exclusively with testing. Debugging is discussed in detail in 7.3.**

Why Software Testing is Hard

- **Exhaustive testing is infeasible:** The space of possible test cases is generally too big to cover exhaustively. 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. 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. This is only true for physical artifacts, but it's not true for software.

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**.

- 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.

- **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.

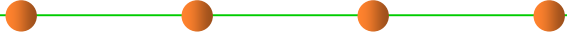- **Not only "make it fail", but also "fail fast".**

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}**

  – 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 .

Characteristics of good test case

- **Most likely to catch the wrong**

- **Not repetitive and not redundant**

- **The most effective in a group of similar test cases**

- **Neither too simple nor too complicated**

Test suite

- **A test suite, less commonly known as a 'validation suite', is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors.**

- **A test suite often contains:**
  - Detailed instructions or goals for each collection of test cases
  - Information on the system configuration to be used during testing.
  - A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.
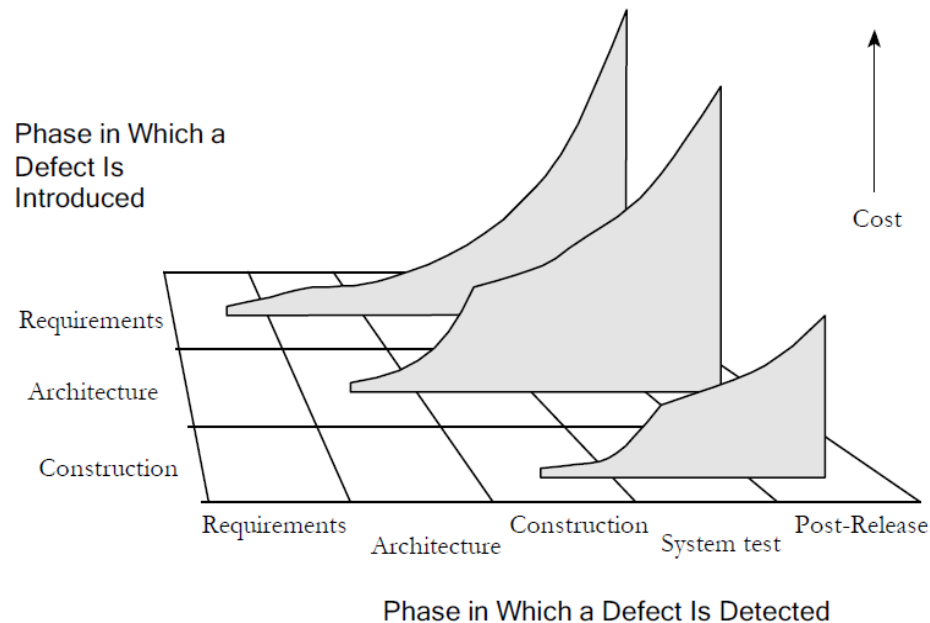
3 Test-First Programming

Test-First Programming

- **Test-first programming: Write the tests before you write the code.**

- **Motivation**
  - Test early and often.

  - 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.

- **In test-first-programming, the development of a single function proceeds in this order:**
  - Write a specification for the function.

  - Write tests that exercise the specification.

  - Write the actual code. Once your code passes the tests you wrote, you're done.

Test First or Test Last?

- **The defect-cost increase graph shows that 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.**

Phase in Which a
Defect Is
Introduced

Cost

Requirements

Architecture

Construction

Requirements   Architecture   Construction   System test   Post-Release

Phase in Which a Defect Is Detected

Test First or Test Last?

- **Many reasons encourage developers to write test cases first:**
  - To detect defects earlier and correct them more easily
  - To lead to a deeper and earlier understanding of the product requirements, ensures the effectiveness of the test code, and maintains a continual focus on software quality.
  - To write test cases first exposes requirements problems sooner, before the code is written, because it's hard to write a test case for a poor requirement.
  - To ensure that the application is written for testability, as the developers must consider how to test the application from the outset rather than adding it later.
 - To ensure that tests for every feature get written.

4 Unit Testing

Unit Testing

- **Unit testing focuses verification effort on the smallest unit of software design—the software component or module (classes in OOP).**

  - Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.

  - 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.

  - The unit test focuses on the internal processing logic and data structures within the boundaries of a component.

Unit-test considerations

- **The module interface is tested to ensure that information properly flows into and out of the program unit under test.**

- **Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.**

- **All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.**

- **Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.**

- **Finally, all error-handling paths are tested.**

Module

Interface
Local data structures
Boundary conditions
Independent paths
Error-handling paths

Test cases

Unit-test

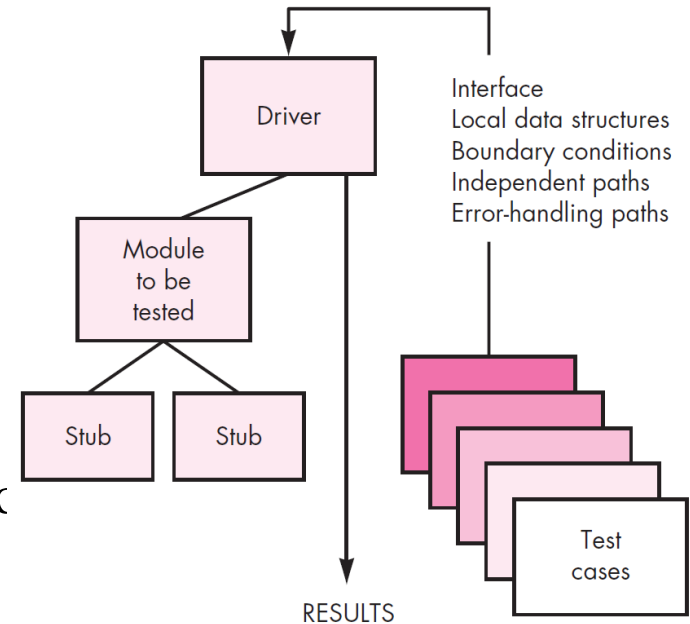Unit-test
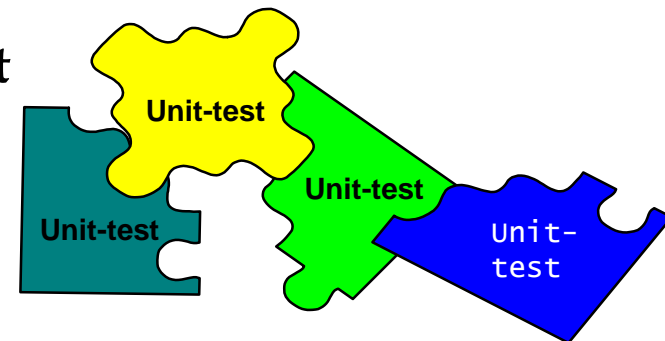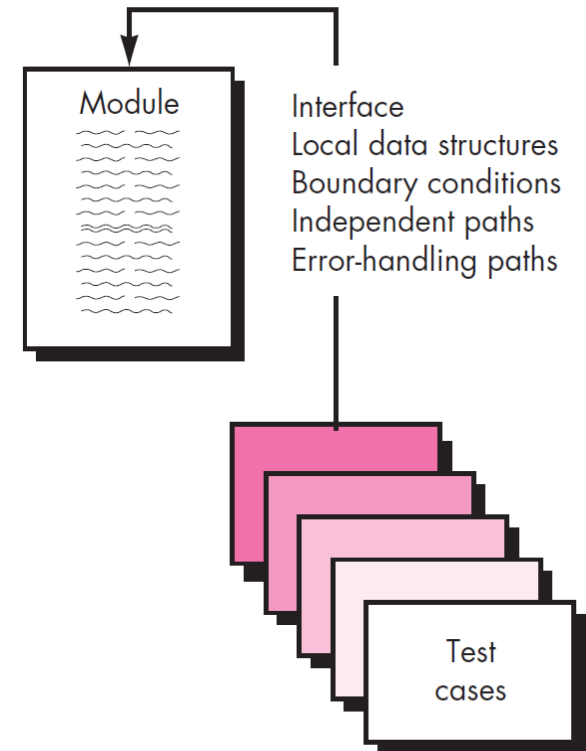
Unit-test

Unit-test

Unit-test procedures

- **Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated.**

- **A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.**
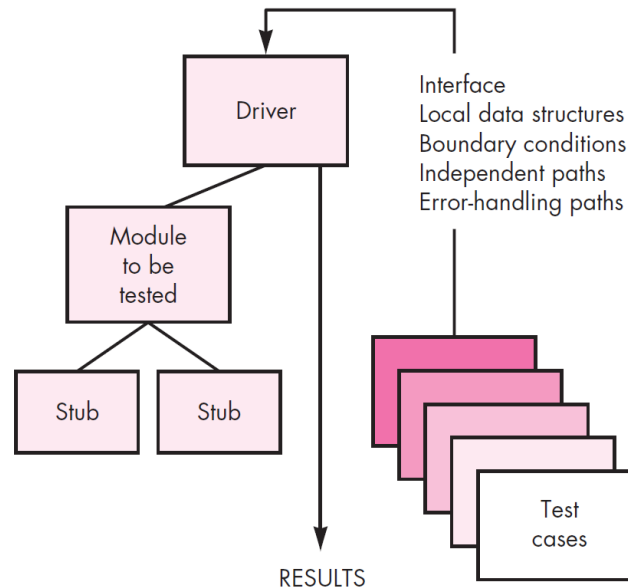
Unit-test procedures

- **Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.**

- **Driver :** In most applications a driver is nothing more than 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 or "dummy ubprogram" uses the subordinate module's interface, may do minimal data manipulation, prints 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 the Java programming language.**

- **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 that originated with SUnit.**

- **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**

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()` **above might look like this when implemented for JUnit:**

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

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

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

JUnit

- **Note that the order of the parameters to assertEquals is important.**
  - The first parameter should be the expected result, usually a constant, that the test wants to see.
  - The second parameter is the actual result, what the code actually does.
  - All the assertions supported by JUnit follow this order consistently: expected first, actual second.

- **If an assertion in a test method fails, then that test method returns immediately, and JUnit records a failure for that test.**

- **A test class can contain any number of @Test methods, which are run independently when you run the test class with JUnit.**

- **Even if one test method fails, the others will still be run.**
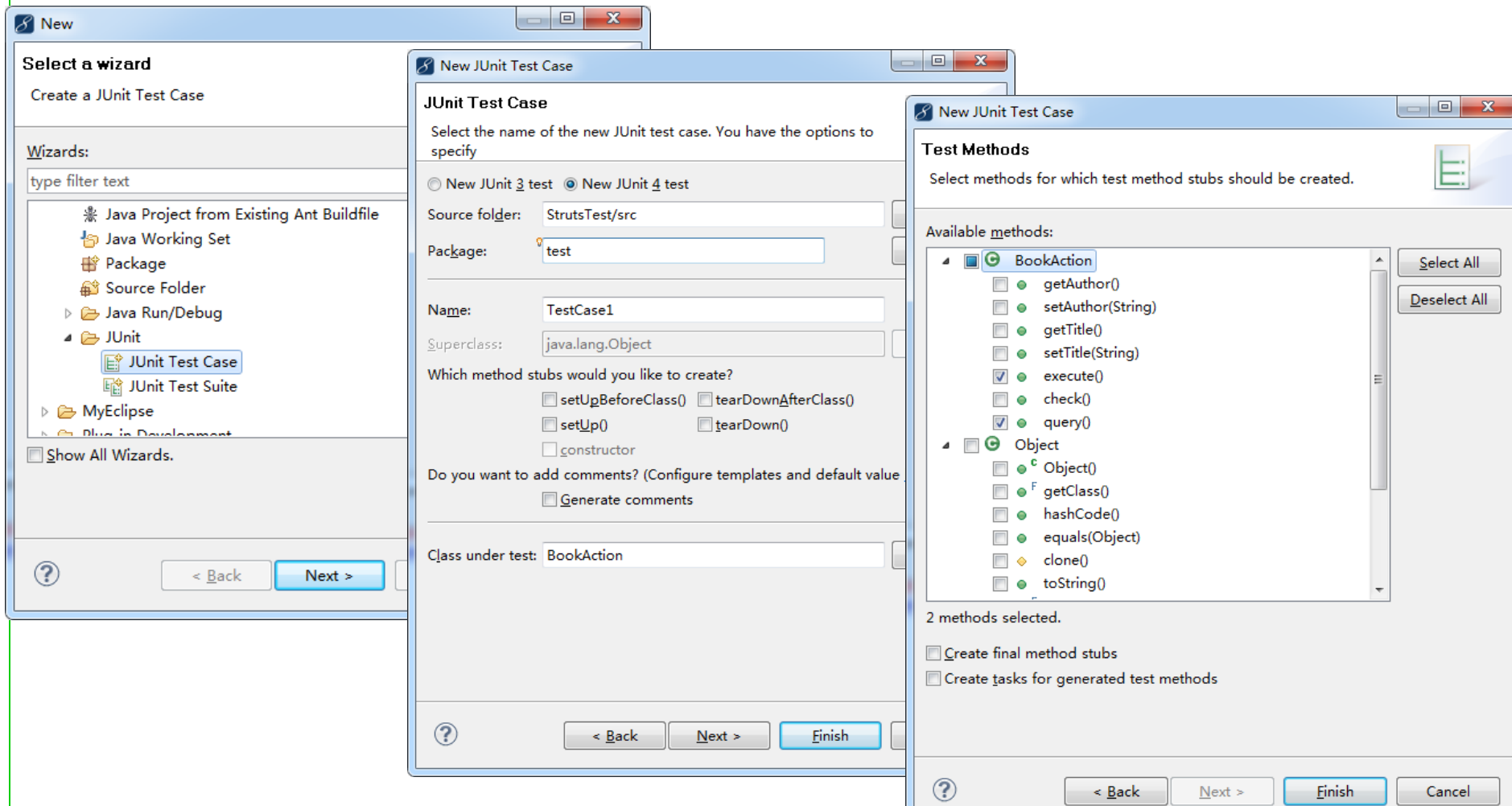
JUnit

- **The class structure of JUnit**
  - TestCase class: a test case, corresponding test code is written in it.
  - TestSuite class: a package of some TestCases which can be run as a whole.
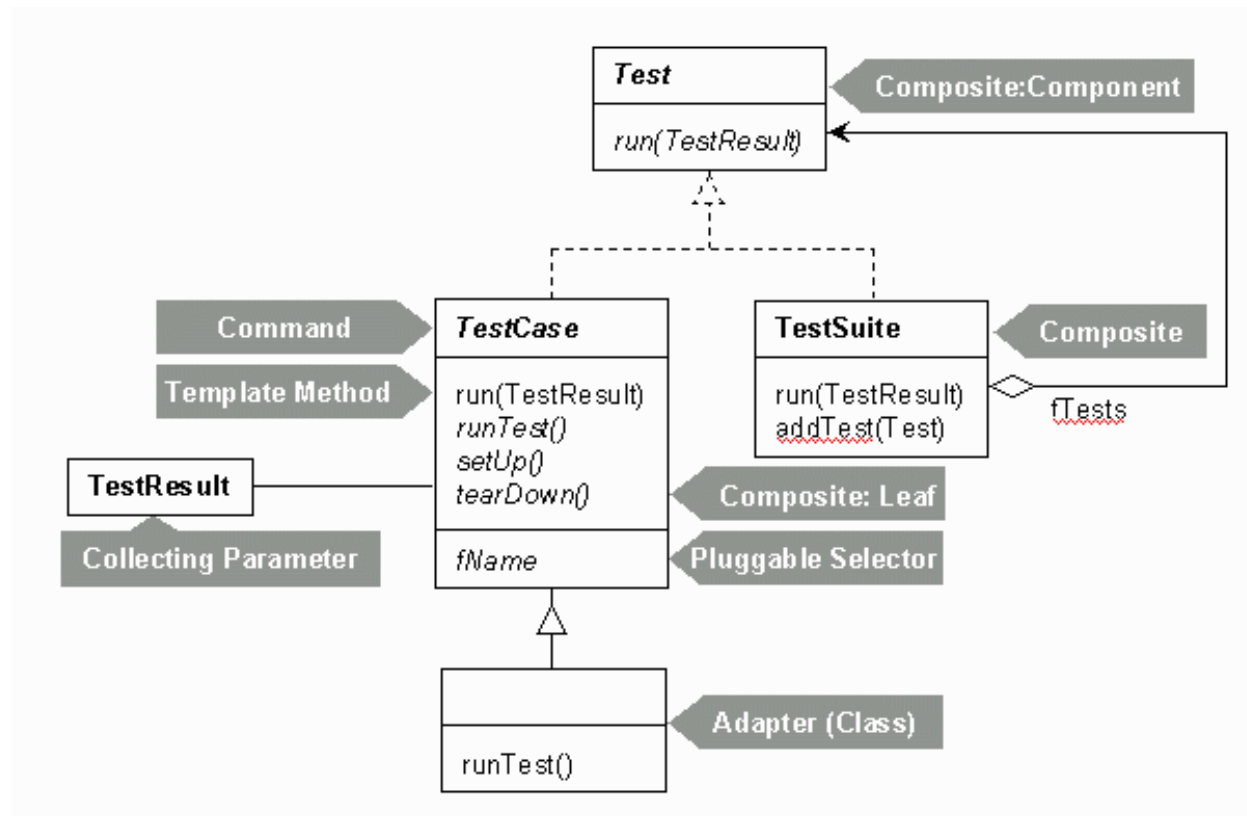
JUnit

- **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

```
Program to be tested：

public class Calcuator {
    public double add (double n1, double n2) {
        return n1 + n2;
    }
}
```

```
Test case：

import junit.framework.TestCase;

public class TestCalcuator extends TestCase {
    public void testAdd(){
        Calcuator calcuator=new Calcuator();
        double result=calcuator.add(1,2);
        assertEquals(3,result,0);
    }
}
```

Input

Expected result

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.**

- **Black-box testing attempts to find errors in the following categories:**
  - (1) incorrect or missing functions,
  - (2) interface errors,
  - (3) errors in data structures or external database access,
  - (4) behavior or performance errors, and
  - (5) initialization and termination errors.

Input ⟹ ⟹ Output
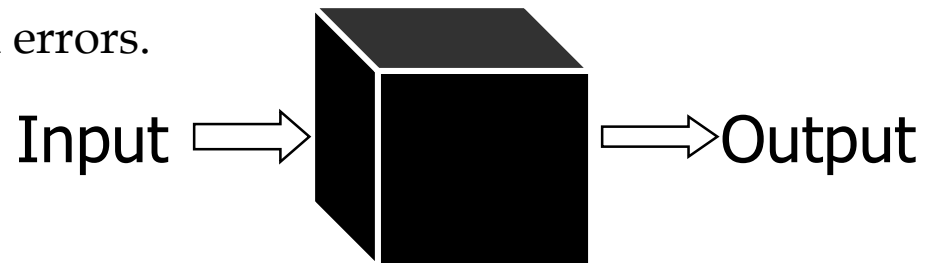
Test cases for black-box testing

- **Test cases for black-box testing are built around specifications and requirements, i.e., what the application is supposed to do.**

- **Test cases are generally derived from external descriptions of the software, including specifications, requirements and design parameters. Although the tests used are primarily functional in nature, non-functional tests may also be used.**
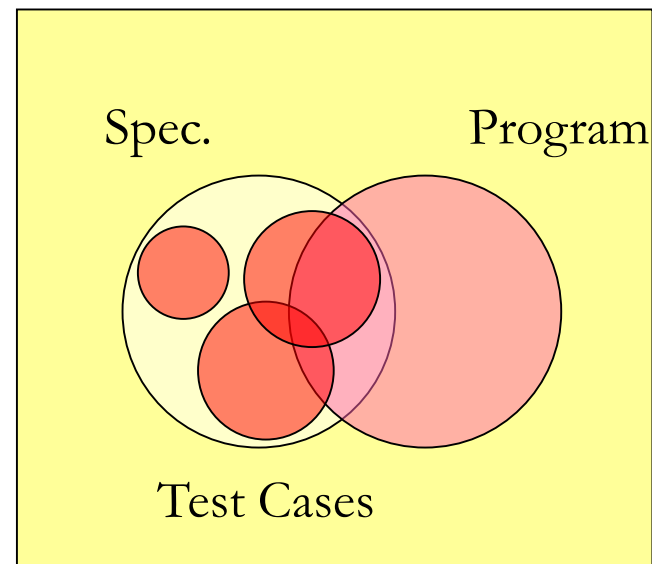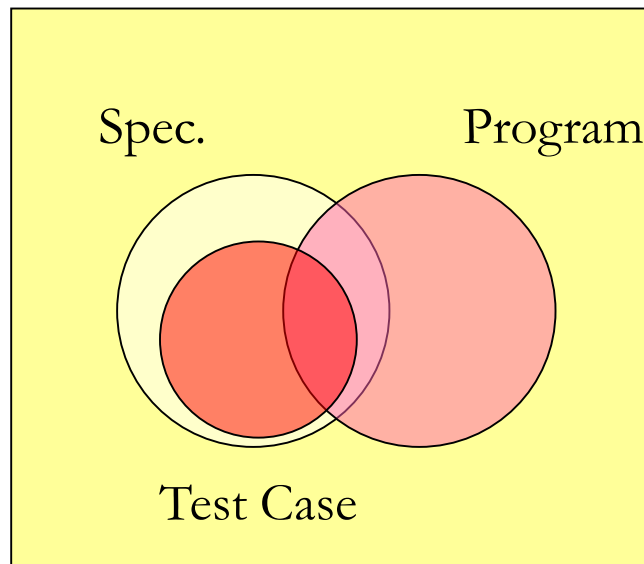
Test design techniques for black-box testing

- **Typical black-box test design techniques include:**
  - Equivalence partitioning
  - Boundary value analysis
  - Decision table testing
  - All-pairs testing
  - Cause–effect graph
  - Error guessing
  - State transition testing
  - Use case testing
  - User story testing
  - Domain analysis
  - Combining technique
 - …

Input ⟹ Output
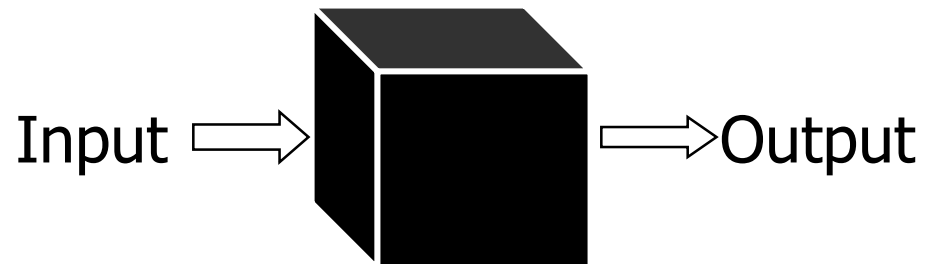
6.1 Choosing Test Cases by Partitioning

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.**

  - 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.

- **We can also partition the equivalence classes (similar outputs on which the program has similar behavior) if we need to ensure our tests will explore different parts of the output space.**
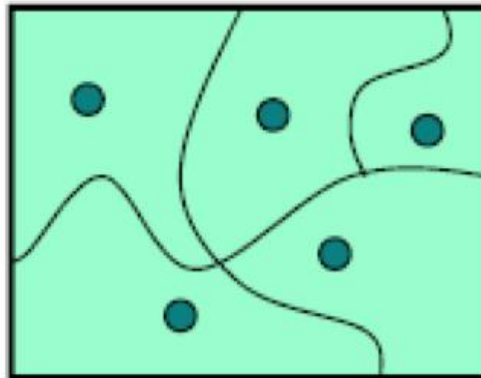
Guidelines for Equivalence Partitioning

- **Equivalence classes may be defined according to the following guidelines:**
  - If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
  - If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
  - If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
  - If an input condition is Boolean, one valid and one invalid class are defined.

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:**

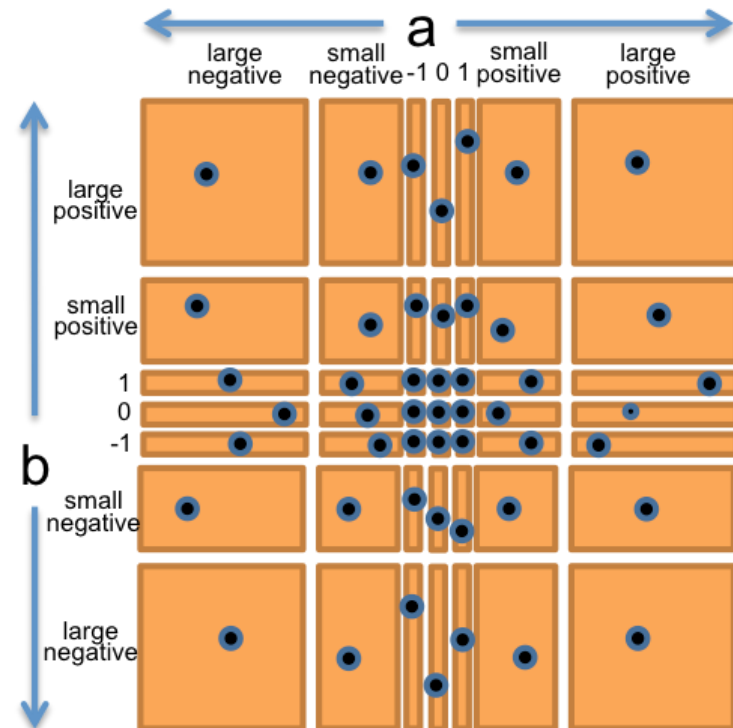$$\texttt{BigInteger} \times \texttt{BigInteger} \rightarrow \texttt{BigInteger}$$

- **So we have a two-dimensional input space, consisting of all the pairs of integers** `(a,b)`.

- **We might start with these partitions:**
  - `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.**

  - `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.**

- **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)

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.**
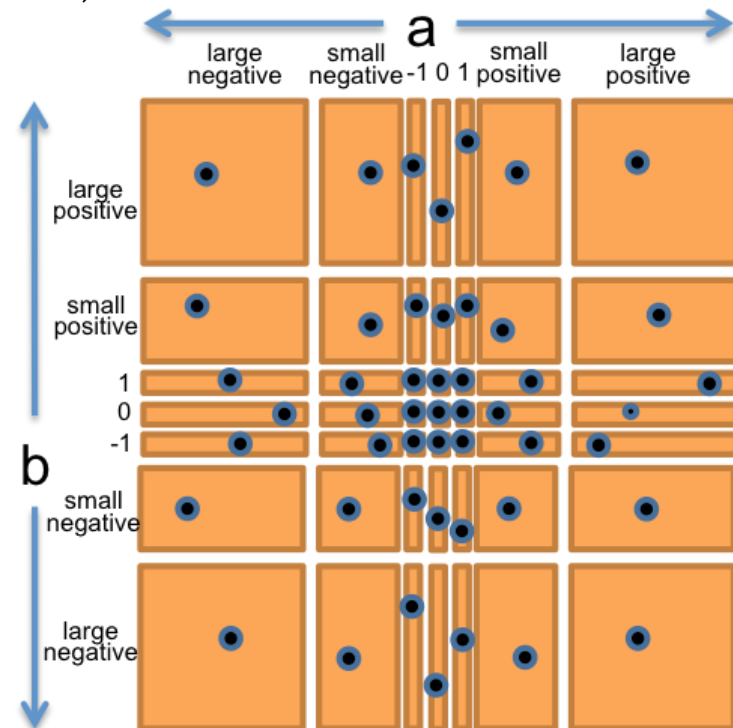
Two Extremes for Covering the Partition

- **Full Cartesian product**
  - Every legal combination of the partition dimensions is covered by one test case.
  - For the max example that included boundaries, which has two dimensions with 7 parts and 7 parts respectively, it would mean up to $7 \times 7 = 49$ test cases.

- **Cover each part.**
  - Every part of each dimension is covered by at least one test case, but not necessarily every combination.
  - With this approach, the test suite for max might be as small as 7 test cases if carefully chosen. That's the approach we took above, which allowed us to choose 7 test cases.

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":**

  - 0 is a boundary between positive numbers and negative numbers

  - The 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

- **It is for this reason that boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.**

  - Boundary value analysis is 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.

  - Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.
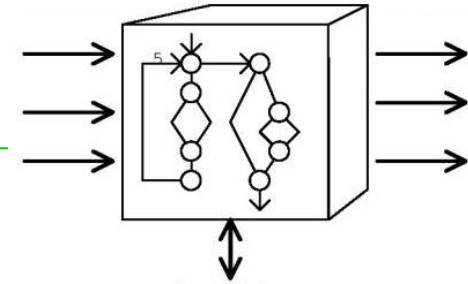
7 White-box Testing

Black-box vs. White-box testing

- **Blackbox testing means choosing test cases only from the specification, not the implementation of the function.**

  – 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.**

  – 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

- **In 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.**

- **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.**
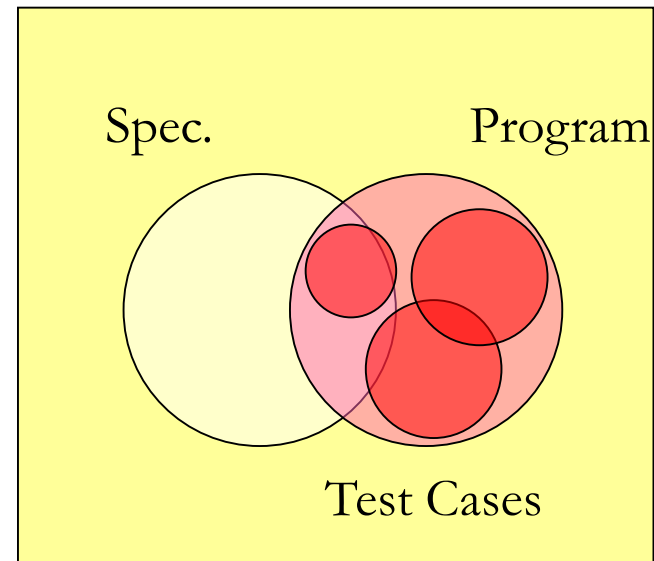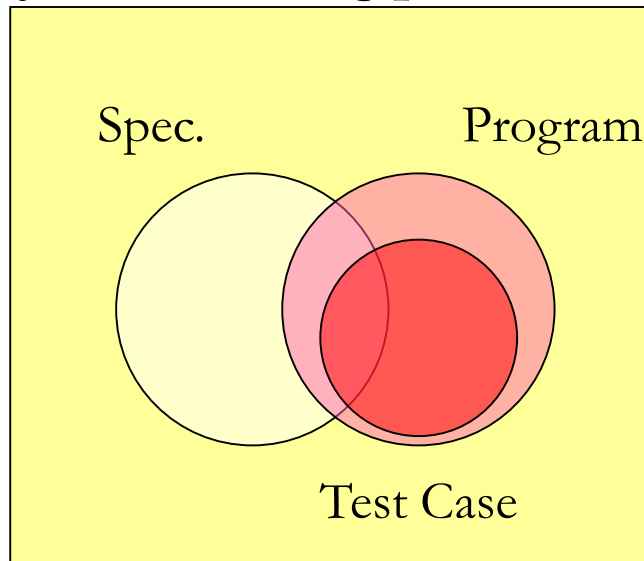
White-box testing

- **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.

8 Coverage of Testing

Code coverage

- **The most thorough white-box method is to cover every path in the program, but because the program generally contains a loop, so the number of paths is great.**

- **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.

循环≤20次
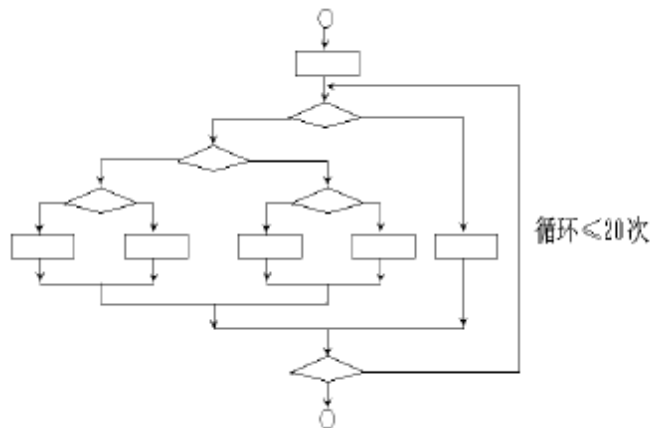
Code coverage

- **Basic coverage criteria**

  - Function coverage – Has each function (or subroutine) in the program been called?

  - Statement coverage – Has each statement in the program been executed?

  - Decision or Branch coverage – Has each branch of each control structure (such as in if and case statements) been executed? For example, given an if statement, have both the true and false branches been executed? Another way of saying this is, has every edge in the program been executed?

  - Condition or predicate coverage – Has each Boolean sub-expression evaluated both to true and false?

  - Condition/decision coverage requires that both decision and condition coverage be satisfied.

  - Multiple condition coverage requires that all combinations of conditions inside each decision are tested.

  - Path coverage: is every possible combination of branches — every path through the program — taken by some test case?

EclEmma

- **A good code coverage tool for Eclipse is [EclEmma](http://www.eclemma.org)**

  - 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.

- If you saw this result from your coverage tool, your next step would be to come up with a test case that causes the body of the while loop to execute, and add it to your test suite so that the red lines become green.

9 Integration Testing

From unit testing to integration testing

- **Unit tests** test a single module (like a method or a class) in isolation.
  - 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.

- In contrast to a unit test, an **integration test** tests a combination of modules, or even the entire program.
  - If all you have are integration tests, then when a test fails, you have to hunt for the bug. It might be anywhere in the program.
  - Integration tests are still important, because a program can fail at the connections between modules.
    - For example, one module may be expecting different inputs than it's actually getting from another module.
  - But if you have a thorough set of unit tests that give you confidence in the correctness of individual modules, then you'll have much less searching to do to find the bug.

What to be tested in Integration Testing?

- **Once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing.**
  - Data can be lost across an interface;
  - One component can have an inadvertent, adverse effect on another;
  - Subfunctions, when combined, may not produce the desired major function;
  - Individually acceptable imprecision may be magnified to unacceptable levels;
  - Global data structures can present problems.
  - Sadly, the list goes on and on.

Example: testing a web search engine

```
/** @return the contents of the web page downloaded from url
 */
public static String getWebPage(URL url) {...}


/** @return the words in string s, in the order they appear,
 *          where a word is a contiguous sequence of
 *          non-whitespace and non-punctuation characters
 */
public static List<String> extractWords(String s) { ... }
```

```
/** @return an index mapping a word to the set of URLs
 *          containing that word, for all webpages in the input set
 */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    for (URL url : urls) {
        String page = getWebPage(url);
        List<String> words = extractWords(page);
        ...
    }
    ...
}
```

Example: testing a web search engine

- **Test process:**
  - Unit tests just for getWebPage() that test it on various URLs
  - Unit tests just for extractWords() that test it on various strings (to use a stub of getWebPage() to isolate the test)
  - Unit tests for makeIndex() that test it on various sets of URLs (integrating two functions together)
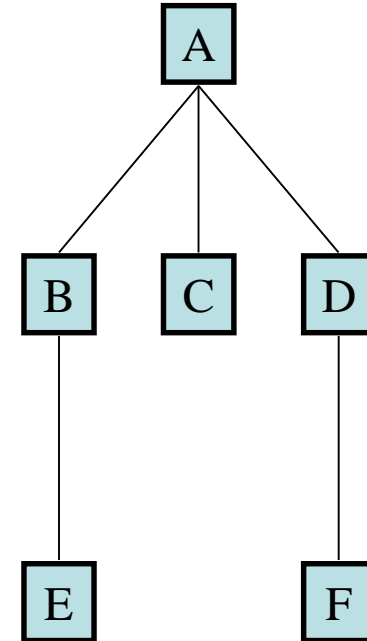
Nonincremental vs. Incremental Testing

- **Nonincremental integration that is to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole.**

  - A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

- **Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.**

  - Top-down integration
  - Bottom-up integration

Nonincremental integration

- **E.g.:**
  - Test A (with stubs for B, C, D)
  - Test B (with driver for A and stub for E)
  - Test C (with driver for A)
  - Test D (with driver for A and stub for F)
  - Test E (with driver for B)
  - Test F (with driver for D)
  - Test (A, B, C, D, E, F)

```
        A
      / | \
     B  C  D
     |     |
     E     F
```

Top-down integration
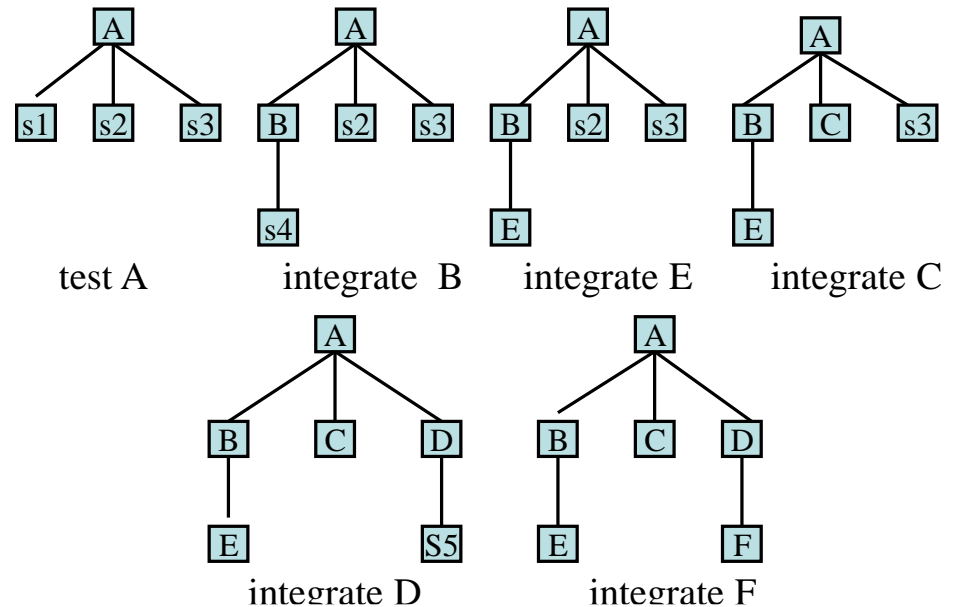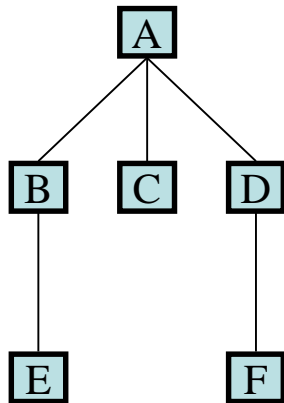
- **Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manr**
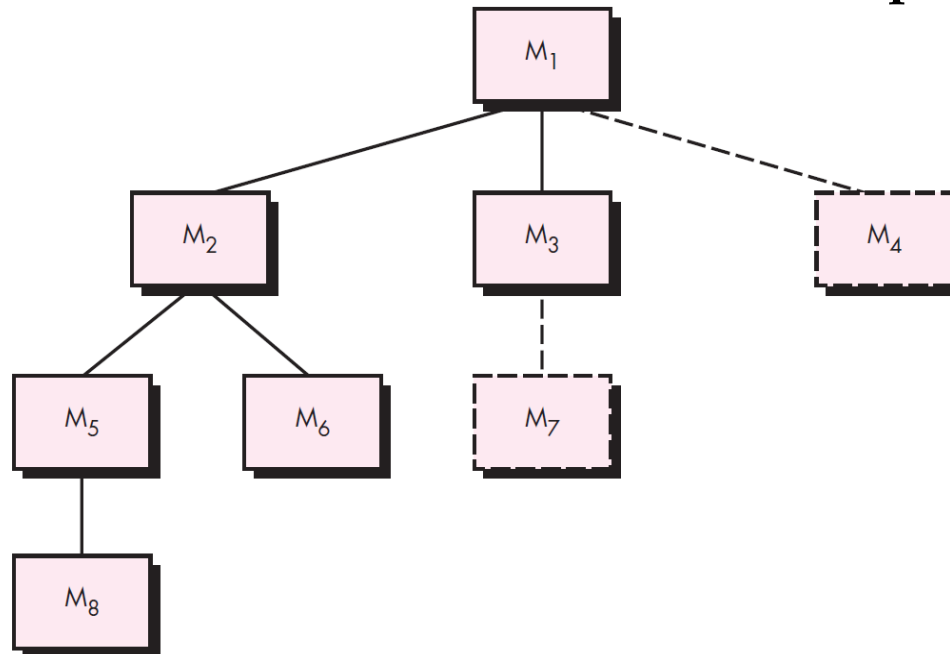
Top-down integration

- **Top-down integration testing: depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.**

test A   integrate B   integrate E   integrate C

integrate D   integrate F

Incremental integration--Top-down integration

- **Top-down integration testing: Breadth-first integration incorporates all componentsdirectly subordinate at each level, moving across the structure horizontally.**

  - Test A (with stubs for B,C,D)
  - Test A;B (with stubs for E,C,D)
  - Test A;B;C (with stubs for E,D)
  - Test A;B;C;D (with stubs for E,F)
 - Test A;B;C;D;E (with stubs for F)
  - Test A;B;C;D;E;F

```
              A
            / | \
           B  C  D
           |     |
           E     F
```
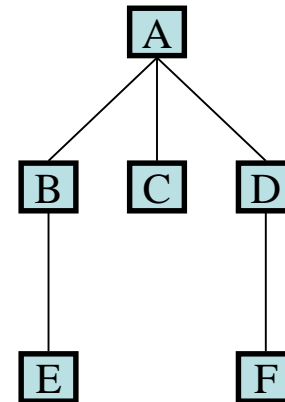
Bottom-up integration

- **Bottom-up integration. Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.**

- **A bottom-up integration strategy may be implemented with the following steps:**

  - 1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

  - 2. A driver (a control program for testing) is written to coordinate test case input and output.

  - 3. The cluster is tested.

  - 4. Drivers are removed and clusters are combined moving upward in the program structure.

Bottom-up integration

- **E.g.,**
  - Test E (with driver for B)
  - Test C (with driver for A)
  - Test F (with driver for D)
  - Test B;E (with driver for A)
  - Test D;F (with driver for A)
  - Test (A;B;C;D;E;F)
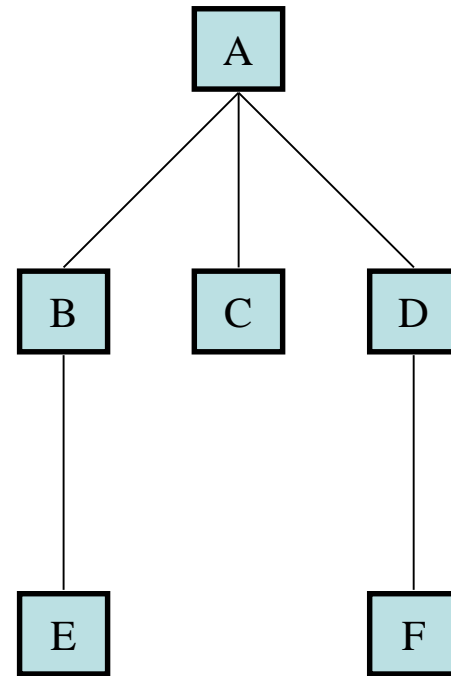
Sandwich integration

- **Sandwich integration is an approach to combine top-down integration with bottom-up integration.**

- **B is selected as the middle layer**
  - Test A (with stubs for B,C,D)
  - Test B (with driver for A and stubs for E,F)
  - Test C (with driver for A)
  - Test D (with driver for A and stub for G)
  - Test A;B (with stubs for E,F,C,D)
  - Test A;B;C (with stubs for E,F,D)
  - Test A;B;C;D (with stubs for E,F,G)
  - Test E (with driver for B)
  - Test F (with driver for B)
  - Test B;E;F (with driver for A)
  - Test G (with driver for D)

Top-down
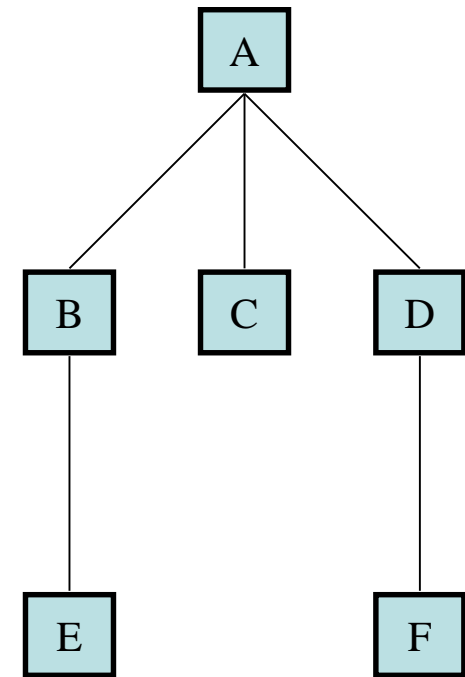
Bottom-up

Top-down integration vs. bottom-up integration

- **Top-down integration**
  - Pros
    - It's possible to examine the program's main control and decision-making mechanism as soon as possible, so can find errors earlier
    - a complete function of software can be realized and displayed(depth-first integration)
    - less drive modules required
  - Cons
    - The number of stub modules required is huge;
    - In the test of higher-level modules, the low-level processing is using stub modules instead, which can not reflect the real situation, and important data can not be sent back to the upper module in time, so the test is not sufficient;

Integration Testing

- **bottom-up integration**
  - Pros
    - There is no need of stub modules
    - Design of test case is relatively simple
  - Cons
    - When the last module is integrated, the overall image of software can be seen. It is difficult to establish confidence as soon as possible.

10 Automated Testing and Regression Testing

Automated testing

- Nothing makes tests easier to run, and more likely to be run, than **complete automation**.

- **Automated testing** means running the tests and checking their results automatically.

  - 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.

  –

Regression testing

- Once you have test automation, it's very important to rerun your tests when you modify your code.

- 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**.

11 Documenting Your Testing Strategy

Documenting Testing Strategy

- **Unit testing strategy is a complementary document of ADT's design.**

- **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.**

An example

```
/**
 * 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:

```
// 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

February 28, 2019