

# LECTURE 4: TESTING

Dr. Ehab Essa  
Computer Science Department,  
Mansoura University

# 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. Nevertheless, small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system, or the bytecode interpreter in a virtual machine, or the filesystem in an operating system.

**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.

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

# WHY SOFTWARE TESTING IS HARD

Here are some approaches that unfortunately don't work well in the world of software.

**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. Physical systems can use many tricks to speed up time, like opening a refrigerator 1000 times in 24 hours instead of 10 years. These tricks give known failure rates (e.g. mean lifetime of a hard drive), but they assume continuity or uniformity across the space of defects. This is true for physical artifacts.

# TEST-FIRST PROGRAMMING

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.

**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.

# TEST-FIRST PROGRAMMING

In test-first-programming, you write tests before you even write any code. 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.

The specification 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 describes how the return value relates to the inputs.

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.

# CHOOSING TEST CASES BY PARTITIONING

Designing an effective test suite is a challenging and interesting design problem

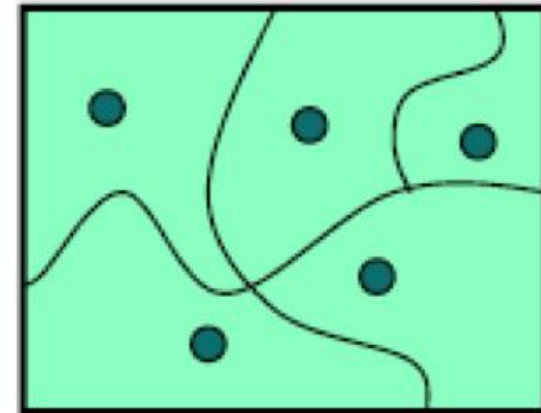
The goal is to create a **small set of tests** that can run quickly, yet is **comprehensive enough** to detect as many potential bugs as possible.

In other words, we seek **maximum coverage with minimal effort**.

To do this, we divide the **input space into subdomains**, each consisting of a set of inputs. Taken together the subdomains completely cover the input space, so that every input lies in at least one subdomain. Then **we choose one test case from each subdomain, and that's our test suite**.

The idea behind subdomains is to partition the input space into sets of similar inputs on which the program has similar behavior. Then we use one representative of each set.

We can also partition the output space into subdomains.



# EXAMPLE: SQRT(X)

For example, suppose you are testing a function `sqrt(x)`:

The input space is all possible numbers for  $x$ .

You can divide it into subdomains such as:

- Negative numbers (invalid inputs)
- Zero
- Positive numbers (valid inputs)

Each of these groups forms a **subdomain**.

Since the program likely behaves differently in each (e.g., throws an exception for negatives, returns 0 for 0, and returns  $\sqrt{x}$  for positives),

we choose **one representative test case** from each subdomain — such as  $x = -1$ ,  $x = 0$ , and  $x = 4$ .

Together, these test cases **cover all logical behaviors** without exhaustively testing every possible input value.

# EXAMPLE BIGINTEGER.MULTIPLY: PARTITIONING

Example: BigInteger.multiply()

Let's look at an example. 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)
```

For example, here's how it might be used:

```
BigInteger a = ...;
```

```
BigInteger b = ...;
```

```
BigInteger ab = a.multiply(b);
```

# EXAMPLE BIGINTEGER.MULTIPLY: PARTITIONING

`multiply : BigInteger × BigInteger → BigInteger`

So we have a two-dimensional input space, consisting of all the pairs of integers (a,b). Now let's partition it. Thinking about how multiplication works, 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

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 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: PARTITIONING

So this will produce  $7 \times 7 = 49$  partitions that completely cover the space of pairs of integers.

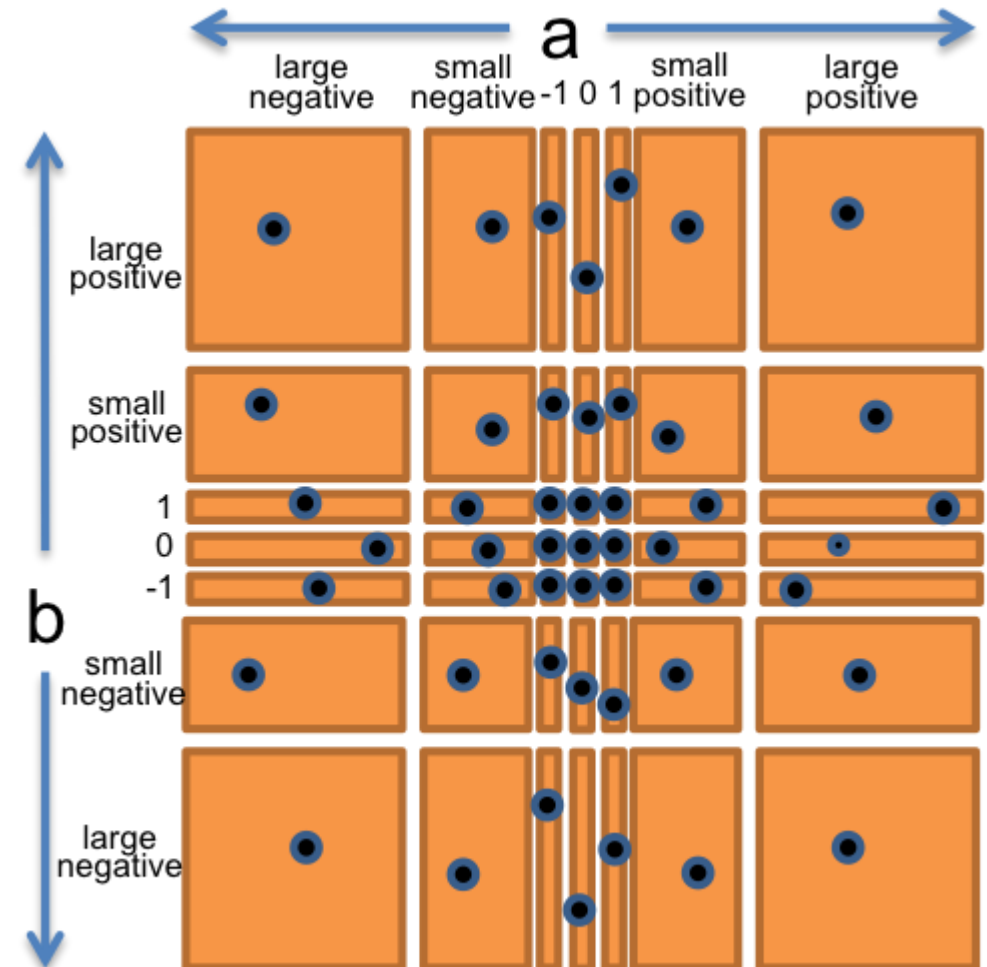
To produce the test suite, we would pick an arbitrary pair  $(a,b)$  from each square of the grid, for example:

$(a,b) = (-3, 25)$  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.



# EXAMPLE MAX(): PARTITIONING

Let's look at 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)
```

Mathematically, this method is a function of the following type:

$\text{max} : \text{int} \times \text{int} \rightarrow \text{int}$

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  $a > b$

# INCLUDE BOUNDARIES IN THE PARTITION

**Bugs often occur at boundaries** between subdomains. Some examples:

- **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

Why do bugs often happen at boundaries?

- off-by-one mistakes (like writing  $\leq$  instead of  $<$ , or initializing a counter to 0 instead of 1).
- some boundaries may need to be handled as special cases in the code.
- Discontinuities — when an int overflows beyond Integer.MAX\_VALUE, it wraps around to a negative value, causing unexpected results.

# INCLUDE BOUNDARIES IN THE PARTITION

To thoroughly test the `max()` method, we can refine our partitioning based on both **the relationship between the two inputs** and their **individual value ranges**:

Partition into:

- *relationship between a and b*

- $a < b$
- $a = b$
- $a > b$

- *value of a*

- $a = 0$
- $a < 0$
- $a > 0$
- $a = \text{minimum integer}$
- $a = \text{maximum integer}$

- *value of b*

- $b = 0$
- $b < 0$
- $b > 0$
- $b = \text{minimum integer}$
- $b = \text{maximum integer}$

**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  $a = b$ ,  $a = 0$ ,  $b = 0$
- $(\text{Integer.MIN\_VALUE}, \text{Integer.MAX\_VALUE})$  covers  $a < b$ ,  $a = \text{minint}$ ,  $b = \text{maxint}$
- $(\text{Integer.MAX\_VALUE}, \text{Integer.MIN\_VALUE})$  covers  $a > b$ ,  $a = \text{maxint}$ ,  $b = \text{minint}$

# TWO EXTREMES FOR COVERING THE PARTITION

After partitioning the input space, we can choose how exhaustive we want the test suite to be:

## Full Cartesian product.

- **Every legal combination of the partition dimensions is covered by one test case.**
- This is what we did for the multiply example, and it gave us  $7 \times 7 = 49$  test cases.
- 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  $3 \times 5 \times 5 = 75$  test cases.
- In practice **not all of these combinations are possible**, however. 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.

## 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 5 test cases if carefully chosen.
- That's the approach we took above, which allowed us to choose 5 test cases.

# EXAMPLE

```
/**
 * 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 start parameter? Check all that apply.

- ☐ 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()

# BLACKBOX AND WHITEBOX TESTING

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

- That's what we've been doing in our examples so far. 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.

When conducting whitebox testing, ensure your test cases aren't based on specific code behaviors not outlined in the specifications.

For example, if the spec says “throws an exception if the input is poorly formatted,” then your test shouldn't check specifically for a `NullPointerException` just because that's what the current implementation does.

# DOCUMENTING YOUR TESTING STRATEGY

```
/**
 * 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.
 */
```

Document how each test case was chosen, including white box tests:

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

// ... other test cases ...
```

# COVERAGE

One way to judge a test suite is to ask **how thoroughly it exercises the program**. This notion is called **coverage**. There are three common kinds of coverage:

- **Statement coverage:** is every statement run by some test case?
- **Branch coverage:** for every **if or while statement** in the program, are both the true and the false direction taken by some test case?
- **Path coverage:** is every possible combination of branches — every path through the program — taken by some test case?

In industry,

- 100% statement coverage is a common goal.
- 100% branch coverage is highly desirable, and safety critical industry code has even more arduous criteria.
- 100% path coverage is infeasible, requiring exponential-size test suites to achieve.

A standard approach to testing is to add tests until the test suite achieves adequate statement coverage: i.e., so that every reachable statement in the program is executed by at least one test case.

In practice, **statement 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;
- In black box testing, you measure the coverage of your black box test first, and then add more test cases until all important statements are logged as executed.

# UNIT TESTING AND INTEGRATION TEST

A well-tested program will have tests for every individual module (where a module is a method or a class) that it contains.

A test that tests an individual module, in isolation if possible, is called **a unit test**.

- 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.

**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.

# AUTOMATED TESTING AND REGRESSION TESTING

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

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.

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

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*.

# TEST-FIRST DEBUGGING (BEST PRACTICE)

## Test-First Debugging (Best Practice)

- When a bug is found:
- **Write a test** that reproduces the bug.
- Run the test to confirm it fails.
- Fix the bug.
- Run the test again — it should now pass.
- Keep the test forever as a regression test.
- This approach guarantees that the bug never silently comes back again.