04834580 Software Engineering (Honor Track) 2025-26

Testing

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

What is Testing?

Definition (IEEE/ISO 24765 [1])

Testing is the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

Testing checks whether software behaves as expected by executing it with concrete inputs and comparing outputs against expected results.

Errors, Faults and Failures

北京大学

- **Error (mistake):** A human action that produces an incorrect result (e.g. misunderstanding a requirement, writing wrong logic).
- **Fault (bug/defect):** A manifestation of an error in the software (e.g. an incorrect statement in the code).
- **Failure:** An observable deviation of the software from its expected behaviour during execution.

Not every fault causes a failure — a fault must be *reached*, *infect* the program state, and *propagate* to an observable output.

Functional and Non-Functional Requirements

北京大学

Functional requirements specify *what* the system should do:

- "The system shall authenticate users with a password"
- "The system shall compute the total price including tax"

Non-functional requirements specify *how well* the system should do it:

- Performance: "Response time ¡ 200ms"
- Reliability: "99.9% uptime"
- Usability, security, scalability, ...

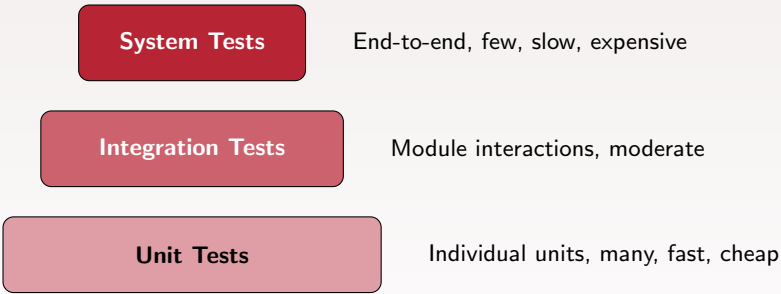
Testing can verify both, but techniques differ: functional testing checks correctness of behaviour, while non-functional testing measures quality attributes (load testing, penetration testing, etc.).

Limitations of Testing

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> — *Edsger W. Dijkstra, 1970 [2]*

- Testing is inherently **incomplete**: we can only check a finite number of inputs, but most programs have an infinite (or astronomically large) input space.
- Passing tests increase *confidence*, not *certainty*.
- Complementary techniques: formal verification, static analysis, code review.

Levels of Testing



Definition

A **unit test** exercises an individual unit (function, method, class) in isolation from the rest of the system.

- Fast (milliseconds per test).
- Dependencies are typically replaced with test doubles (mocks, stubs).
- First line of defence: catches bugs early in development.
- Example: testing that a `sort` function returns elements in order.

Definition

An **integration test** verifies that multiple units work together correctly when combined.

- Tests interactions between modules, services, or layers.
- May involve real databases, file systems, or network calls.
- Slower than unit tests, but catch interface mismatches.
- Example: testing that a REST endpoint correctly queries the database and returns a JSON response.

Definition

A **system test** (end-to-end test) validates the complete, integrated system against its requirements.

- Exercises the full application stack (UI, backend, database, external services).
- Slowest and most brittle, but highest confidence.
- Often automated with tools like Selenium, Playwright, or Cypress.
- Example: simulating a user logging in, adding an item to a cart, and checking out.

Definition

A **basic block** is a maximal sequence of consecutive statements with a single entry point and a single exit point — control flow enters at the top and exits at the bottom without branching (except at the end).

```
def abs_value(x):
    if x < 0:          # Block 1
        x = -x         # Block 2
    return x           # Block 3
```

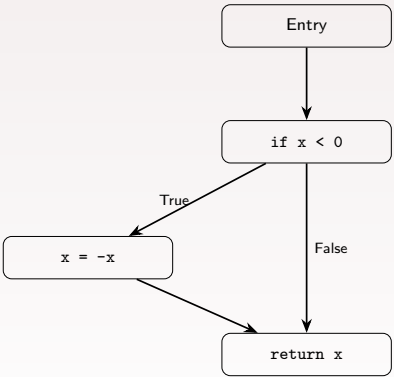
Three basic blocks:

1. Entry + condition: `if x < 0`
2. Then-branch: `x = -x`
3. Exit: `return x`

Definition

A **control flow graph** (CFG) is a directed graph where nodes represent basic blocks and edges represent possible control flow transitions between them.

CFGs are used to define **coverage criteria**: how thoroughly tests exercise different parts of the code.



Definition

Statement coverage requires that every statement in the program is executed at least once.

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

```
def foo(x, y):
    z = 0
    if x > 0:
        z = x
    if y > 0:
        z = z + y
    return z
```

Test case: `foo(1, 1)`

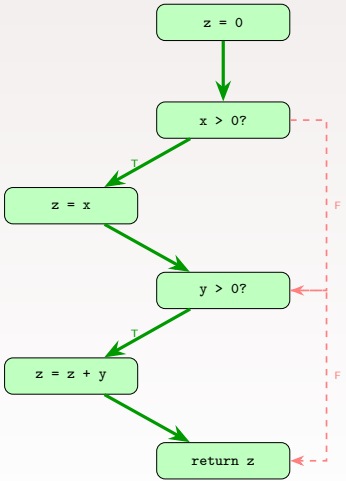
Executes every statement $\Rightarrow$ 100% statement coverage.

But never tests the case where `x <= 0` or `y <= 0`!

With test `foo(1, 1)`, all **nodes** are visited (highlighted), achieving 100% statement coverage.

However, the *False* branches of both conditions are **never taken**.

Statement coverage can miss entire branches of logic.



Definition

Branch coverage (decision coverage) requires that every branch (True/False edge) of every decision point is taken at least once.

$$\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}$$

Branch coverage $\Rightarrow$ statement coverage, but not vice versa.

For `foo(x, y)`:
Test set achieving 100% branch coverage:

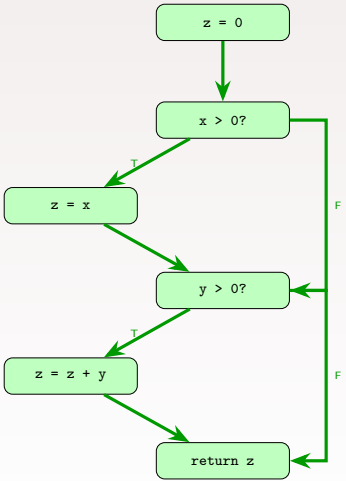
- `foo(1, 1)` — both True
- `foo(-1, -1)` — both False

4 branches total (2 decisions $\times$ 2 outcomes):

- `x > 0`: True, False
- `y > 0`: True, False

With tests `foo(1, 1)` and `foo(-1, -1)`, all **edges** are traversed, achieving 100% branch coverage.

Branch coverage is **strictly stronger** than statement coverage: 100% branch coverage guarantees 100% statement coverage, but not the other way around.



Definition

Path coverage requires that every possible execution path through the program is taken at least once.

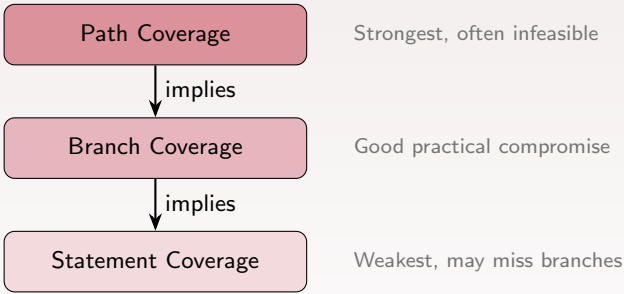
For `foo(x, y)` with two independent `if` statements, there are $2 \times 2 = 4$ paths:

Path	`x > 0`	`y > 0`	Test case
1	T	T	`foo(1, 1)`
2	T	F	`foo(1, -1)`
3	F	T	`foo(-1, 1)`
4	F	F	`foo(-1, -1)`

Path coverage $\Rightarrow$ branch coverage $\Rightarrow$ statement coverage.

Problem: with loops, the number of paths can be **infinite**. Even without loops, n independent conditions yield 2^n paths — exponential growth.

Coverage Criteria Hierarchy



In practice, branch coverage is the most commonly used criterion. Many organizations target 80–90% branch coverage as a reasonable goal.

History of Unit Testing Frameworks

- **1994**: Kent Beck writes **SUnit** for Smalltalk — the first xUnit framework [3].
- **1997**: Kent Beck and Erich Gamma port SUnit to Java, creating **JUnit** [4].
- **2000s**: xUnit pattern spreads to virtually every language:
  - NUnit (C#), CppUnit (C++), PyUnit/unittest (Python), ...
- **2001**: "Test-first" becomes a pillar of Extreme Programming [5].
- **Today**: Unit testing is standard practice; CI/CD pipelines run thousands of tests on every commit.

All xUnit frameworks share the same core architecture: test methods, test classes, assertions, fixtures (setup/teardown), and test runners.

Where is Unit Testing Applicable?

北京大学

Unit testing is most effective for:

- Pure functions and deterministic logic.
- Data transformations and algorithms.
- Business rules and validation logic.
- Parsing and serialization.

Unit testing is harder (but still possible) for:

- GUI/UI code — use integration/end-to-end tests.
- Code with many external dependencies — requires mocking.
- Concurrency and timing-dependent code.
- Non-deterministic systems (randomized algorithms, ML models).

Testing with `println` (ad hoc testing)


```java
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("add(2,3) = " + calc.add(2, 3));
        System.out.println("add(-1,1) = " + calc.add(-1, 1));
        // Manually inspect output...
    }
}
```

Problems:

- Must **manually inspect** output every time.
- No automatic pass/fail — easy to miss regressions.
- Not repeatable or composable.
- Cannot run in CI/CD.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
    @Test
    void testAdd() {
        Calculator calc = new Calculator();
        assertEquals(5, calc.add(2, 3));
        assertEquals(0, calc.add(-1, 1));
    }
}
```

Advantages:

- **Automated** pass/fail — no manual inspection.
- **Repeatable** — run the same tests after every change.
- **Self-documenting** — tests describe expected behaviour.
- Integrates with build tools and CI/CD pipelines.

Anatomy of a Test Method

A well-structured test follows the **Arrange–Act–Assert** pattern:

```
@Test
void testWithdrawal() {
    // Arrange: set up the test scenario
    BankAccount account = new BankAccount(1000);

    // Act: perform the action being tested
    account.withdraw(200);

    // Assert: verify the expected outcome
    assertEquals(800, account.getBalance());
}
```

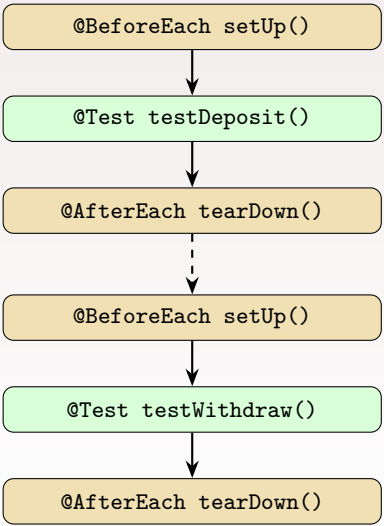
- **Arrange**: Create objects, set up preconditions.
- **Act**: Call the method under test.
- **Assert**: Check the result against expected values.

Test Classes and Fixtures

北京大学

```java
class BankAccountTest {
    private BankAccount account;

    @BeforeEach
    void setUp() {
        account = new BankAccount(1000); // runs before each test
    }

    @AfterEach
    void tearDown() {
        account.close(); // runs after each test
    }

    @Test
    void testDeposit() {
        account.deposit(500);
        assertEquals(1500, account.getBalance());
    }

    @Test
    void testWithdraw() {
        account.withdraw(200);
        assertEquals(800, account.getBalance());
    }
}
```

JUnit Execution Model



Fresh instance for each test

Tests are independent

Order should not matter

Key principle: each test runs in **isolation** — a fresh object is created before each

Class-Level Setup and Teardown

For expensive resources (database connections, servers), use **class-level** fixtures:

```
class DatabaseTest {
    private static Connection conn;

    @BeforeAll
    static void initDatabase() {
        conn = DriverManager.getConnection("jdbc:h2:mem:test");
    }

    @AfterAll
    static void closeDatabase() throws Exception {
        conn.close();
    }

    @BeforeEach
    void cleanTables() {
        conn.createStatement().execute("DELETE FROM users");
    }
    // ... tests ...
}
```

Testing Exceptions

北京大学

Tests can verify that code throws expected exceptions: **JUnit 5:**

```
@Test
void testOverdraft() {
    BankAccount account = new BankAccount(100);

    assertThrows(InsufficientFundsException.class, () -> {
        account.withdraw(200);
    });
}
```

You can also inspect the exception:

```
@Test
void testOverdraftMessage() {
    BankAccount account = new BankAccount(100);

    Exception ex = assertThrows(InsufficientFundsException.class,
        () -> account.withdraw(200));
    assertEquals("Insufficient funds", ex.getMessage());
}
```

Python's standard library includes `unittest`, modelled after JUnit:

```python
import unittest

class TestCalculator(unittest.TestCase):

    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        self.assertEqual(self.calc.add(2, 3), 5)

    def test_add_negative(self):
        self.assertEqual(self.calc.add(-1, 1), 0)

    def tearDown(self):
        pass  # cleanup if needed

if __name__ == '__main__':
    unittest.main()
```

```python
class TestDivision(unittest.TestCase):

    def test_divide_by_zero(self):
        with self.assertRaises(ZeroDivisionError):
            result = 1 / 0

    def test_invalid_input(self):
        with self.assertRaises(ValueError) as context:
            int("not_a_number")
        self.assertIn("invalid literal", str(context.exception))
```

Running tests:

```
python -m unittest test_calculator.py
python -m unittest discover -s tests -p "test_*.py"
```

JUnit 5 Assertion Methods


```
assertEquals(expected, actual);      // equality
assertNotEquals(a, b);               // inequality
assertTrue(condition);               // boolean check
assertFalse(condition);
assertNull(obj);                     // null check
assertNotNull(obj);
assertSame(obj1, obj2);             // reference equality
assertArrayEquals(arr1, arr2);       // array equality
assertThrows(Exception.class, () -> { ... }); // exception
assertTimeout(Duration.ofSeconds(1), () -> { ... });
```

All assertion methods accept an optional **message** parameter:

```
assertEquals(5, calc.add(2, 3), "2 + 3 should equal 5");
```

```
self.assertEqual(a, b)           # a == b
self.assertNotEqual(a, b)        # a != b
self.assertTrue(x)               # bool(x) is True
self.assertFalse(x)              # bool(x) is False
self.assertIs(a, b)              # a is b
self.assertIsNone(x)             # x is None
self.assertIn(a, b)              # a in b
self.assertIsInstance(a, b)      # isinstance(a, b)
self.assertRaises(Exc, func, args)
self.assertAlmostEqual(a, b)     # for floats
self.assertGreater(a, b)         # a > b
self.assertRegex(text, regex)    # regex match
```

Use specific assert methods rather than `assertTrue` — they produce better error messages when tests fail.

Testing Best Practices

- **Test one thing per test method.** Each test should verify a single behaviour — makes failures easy to diagnose.
- **Use descriptive test names.** E.g. `testWithdrawInsufficientFunds` rather than `test3`.
- **Keep tests independent.** No test should depend on the result or side effect of another test.
- **Keep tests fast.** Slow tests discourage frequent execution.
- **Test edge cases.** Empty inputs, boundary values, null/None, negative numbers, overflow.

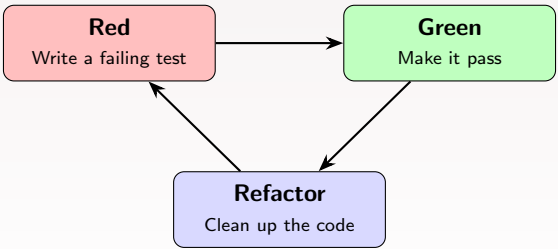
Testing Best Practices (continued)

- **Don't test implementation details.** Test the public API and observable behaviour, not internal state. Refactoring should not break tests.
- **Follow Arrange–Act–Assert.** Consistent structure improves readability.
- **Avoid logic in tests.** Tests should be simple and obvious — no loops, conditions, or complex setup.
- **Use assertions, not** `println`**.** Automated checks, not manual inspection.
- **Run tests frequently.** Ideally on every save/commit (CI/CD).
- **Treat test code as production code.** Refactor, review, and maintain it.

Definition (Kent Beck [6])

Test-Driven Development is a software development process where tests are written *before* the production code that makes them pass.

The TDD cycle (**Red–Green–Refactor**):



TDD Example: Step 1 — Red

Write a test for behaviour that does not exist yet:

```
@Test
void testFizzBuzz() {
    assertEquals("1", fizzBuzz(1));
    assertEquals("2", fizzBuzz(2));
    assertEquals("Fizz", fizzBuzz(3));
    assertEquals("Buzz", fizzBuzz(5));
    assertEquals("FizzBuzz", fizzBuzz(15));
}
```

This test **does not compile** — the method `fizzBuzz` does not exist yet. The test is **red**.

TDD Example: Step 2 — Green

Write the **simplest code** that makes the test pass:

```
String fizzBuzz(int n) {
    if (n % 15 == 0) return "FizzBuzz";
    if (n % 3 == 0)  return "Fizz";
    if (n % 5 == 0)  return "Buzz";
    return String.valueOf(n);
}
```

The test is now **green**. Do not add anything beyond what the tests require.

TDD Example: Step 3 — Refactor

北京大学

Improve the code while keeping all tests green:

- Remove duplication.
- Improve naming.
- Extract methods or classes.
- Simplify logic.

The tests act as a **safety net** — if refactoring breaks something, the tests catch it immediately.

Then the cycle repeats: write the next failing test, make it pass, refactor.

Benefits and Limitations of TDD

北京大学

Benefits:

- Forces thinking about design and interface *before* implementation.
- Produces high test coverage by construction.
- Provides immediate feedback on regressions.
- Results in simpler, more modular code.

Limitations:

- Overhead for exploratory or throwaway code.
- Hard to apply to UI, concurrency, or legacy code without testable interfaces.
- Can lead to over-testing implementation details if not careful.
- Requires discipline and practice.

Tests as Specification

Well-written tests serve as **executable documentation**:

```
class StackTest {
    @Test void newStackIsEmpty() {
        assertTrue(new Stack<>().isEmpty());
    }
    @Test void pushThenPopReturnsSameElement() {
        Stack<Integer> s = new Stack<>();
        s.push(42);
        assertEquals(42, s.pop());
    }
    @Test void pushThenSizeIncrements() {
        Stack<Integer> s = new Stack<>();
        s.push(1);
        assertEquals(1, s.size());
    }
    @Test void popOnEmptyStackThrows() {
        assertThrows(EmptyStackException.class,
            () -> new Stack<>().pop());
    }
}
```

These tests specify the **contract** of `Stack`:

- A new stack is empty.
- Pushing then popping returns the same element (LIFO).
- Pushing increments the size.
- Popping from an empty stack is an error.

This is a form of **specification by example**:

- Concrete, unambiguous, and verifiable — unlike natural language specs.
- Always up-to-date — if the implementation changes and tests pass, the spec is still valid.
- The test suite is a "living document" of the system's behaviour.

Structuring Data in Tests

When testing with multiple input/output pairs, structure your data clearly:

```
class TestRomanNumerals(unittest.TestCase):

    test_cases = [
        (1,    "I"),
        (4,    "IV"),
        (9,    "IX"),
        (40,   "XL"),
        (90,   "XC"),
        (400,  "CD"),
        (1994, "MCMXCIV"),
    ]

    def test_to_roman(self):
        for number, expected in self.test_cases:
            with self.subTest(number=number):
                self.assertEqual(to_roman(number), expected)
```

`subTest` ensures all cases are checked even if one fails, and the failing input is reported.

References I

[1] ISO/IEC/IEEE 24765: Systems and Software Engineering — Vocabulary, 2017.
Available from `https://www.iso.org/standard/71952.html`.

[2] Edsger W Dijkstra.
Notes on structured programming, 1970.

[3] Kent Beck.
Simple Smalltalk testing: With patterns.
The Smalltalk Report, 4(2):16–18, 1994.

[4] Kent Beck and Erich Gamma.
Test infected: Programmers love writing tests.
`https://junit.sourceforge.net/doc/testinfected/testing.htm`, 1998.

[5] Kent Beck.
Extreme programming explained: embrace change.
addison-wesley professional, 2000.

References II

[6] Kent Beck.
Test driven development: By example.
Addison-Wesley Professional, 2022.