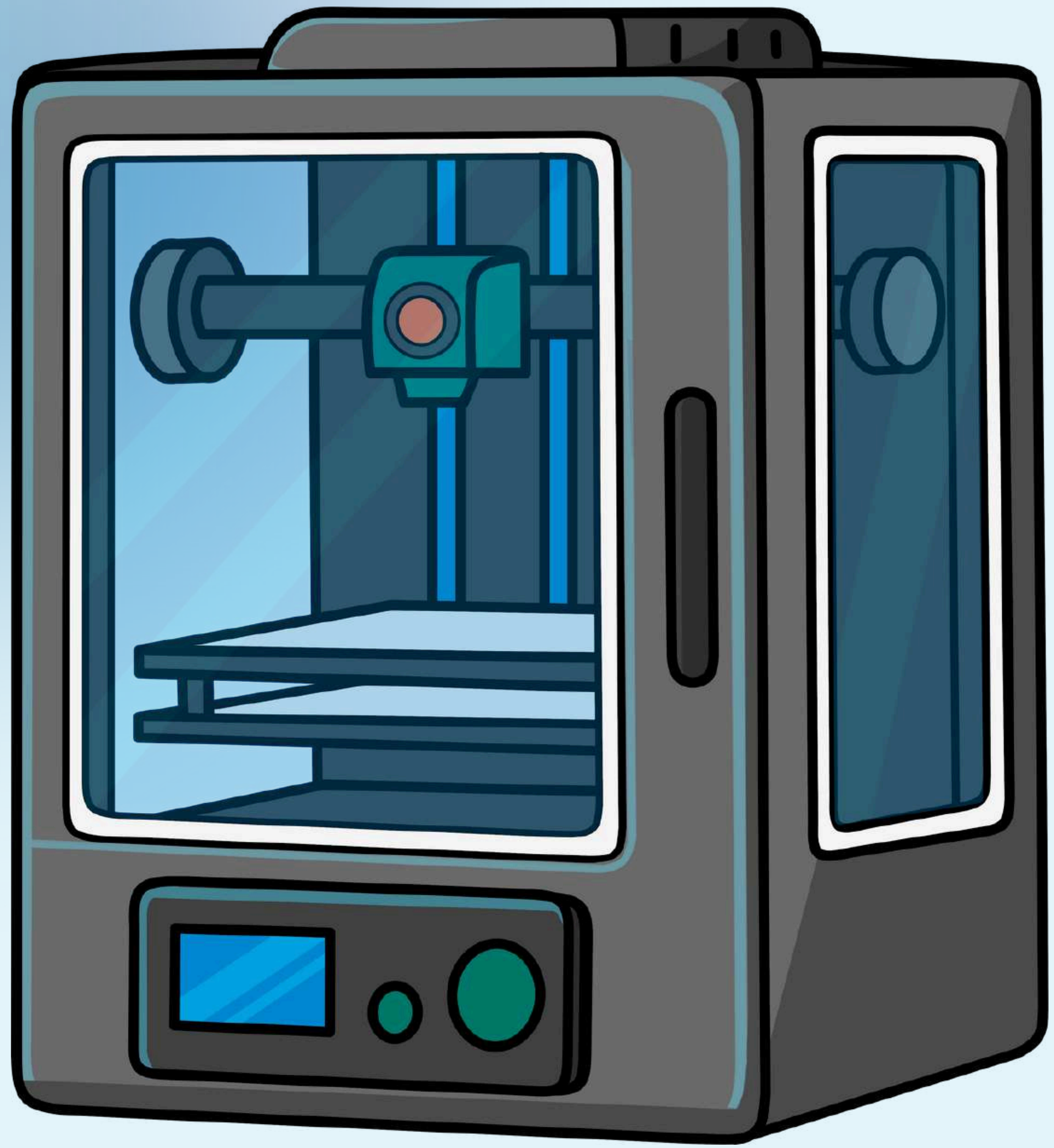
JUnit & Unit Testing

Nazanin Foroutan
AP - Fall 2025 - Dr. Vahidi Asl



Every Product Needs Testing

- No product is perfect from the start
- A bug found late = very high cost
- Famous failures:
  - Samsung Note 7 – battery explosions
  - Ford Pinto – fuel tank fire hazard
- Importance of early defect detection

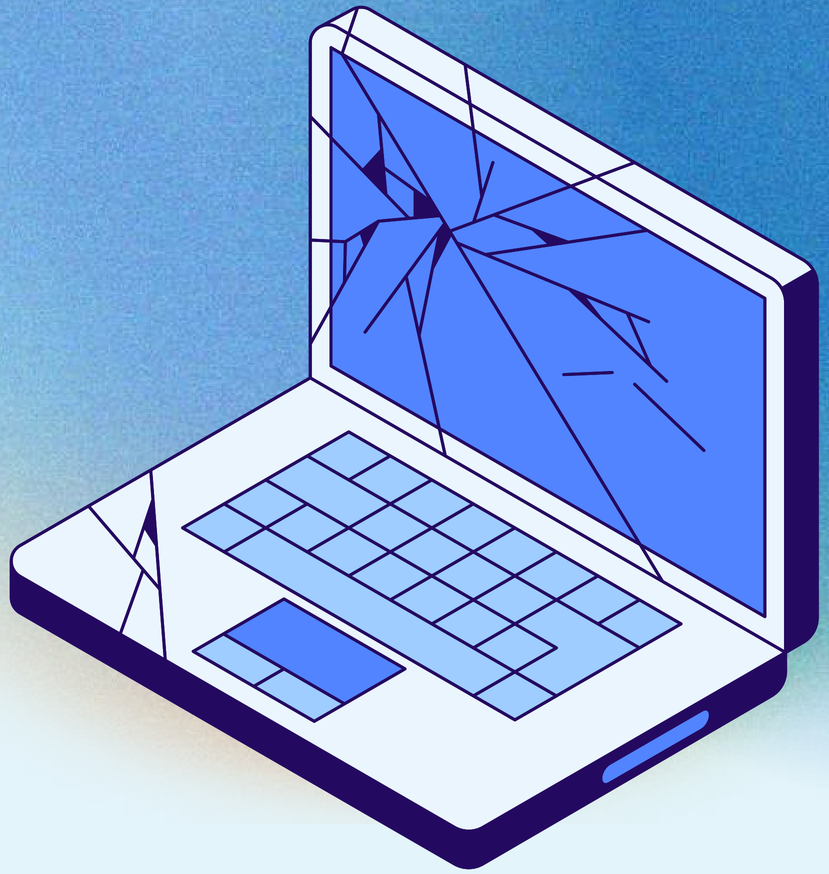
Product Lifecycle & the Role of Testing

- Design → Development → Testing → Release
- Testing ensures reliability before launch
- Testing reduces risk and uncertainty

Software Is Also a “Product”

- Not physical, but it behaves
- Daily dependence on software
- Software failures can cost more than physical failures

Catastrophic Software Failures



Ariane 5 rocket explosion - type conversion error

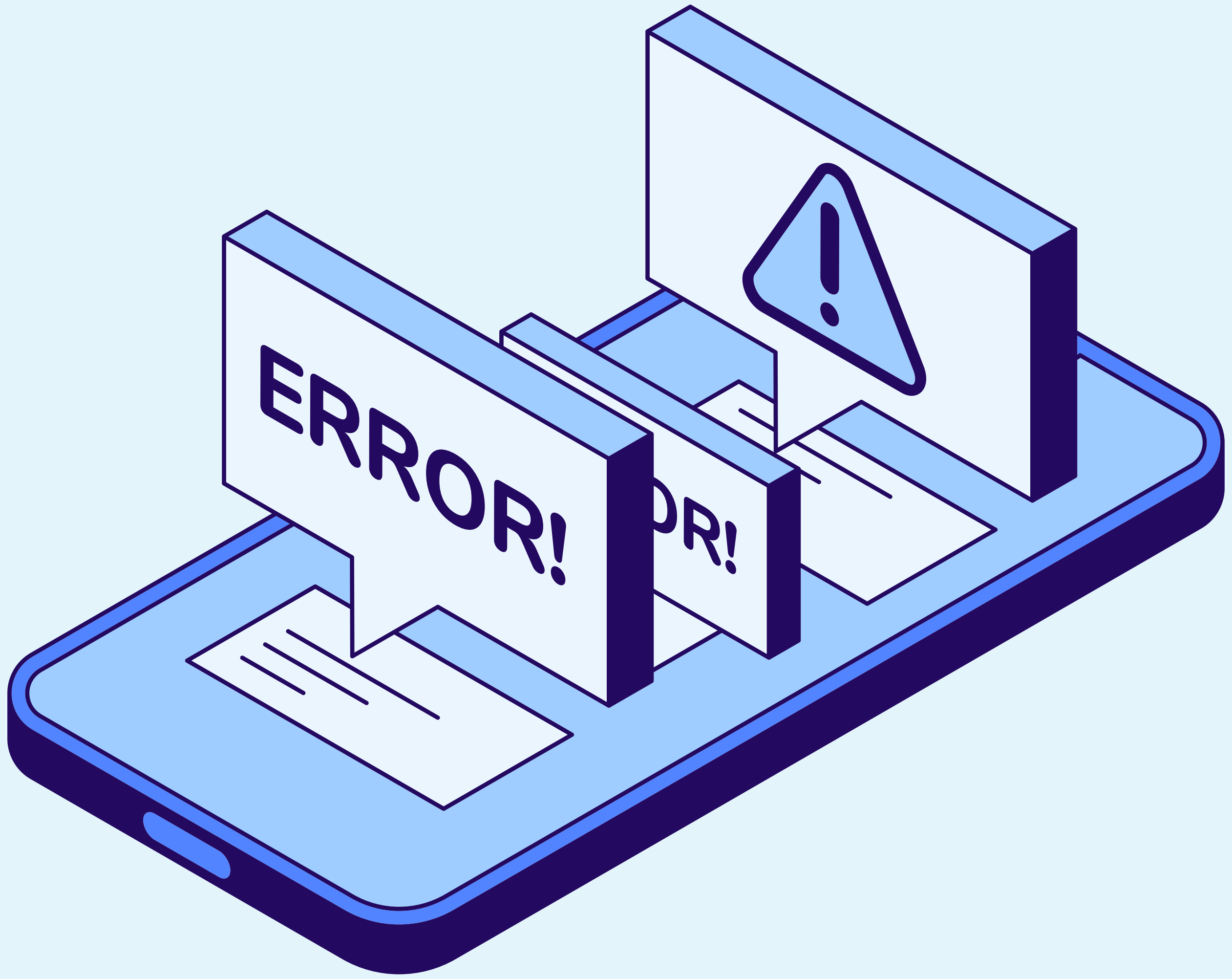
A number was converted incorrectly (64-bit → 16-bit), causing an overflow and destroying the rocket.
Lesson: Always test data boundaries and type conversions.

Knight Capital - $440M Loss in 45 Minutes

Old, untested code accidentally activated and executed huge unintended trades.
Lesson: Regression tests prevent old bugs from coming back.

Mars Climate Orbiter - Unit Mismatch

One team used metric units, another used imperial, causing navigation failure and loss of the spacecraft.
Lesson: Test integrations carefully, especially units and shared data.



Why Is Software Error-Prone?

- High complexity
- Dynamic behavior
- Many dependencies
- Error growth is exponential with project size

Qualities of Good Software

Functional Requirements

Functional qualities describe what the software should do - the correctness of behavior.

Correctness
The software produces the right outputs for the given inputs.
It behaves exactly as the requirements specify.

Security
The system prevents unauthorized access and protects data.
Includes authentication, authorization, and safe handling of sensitive operations.

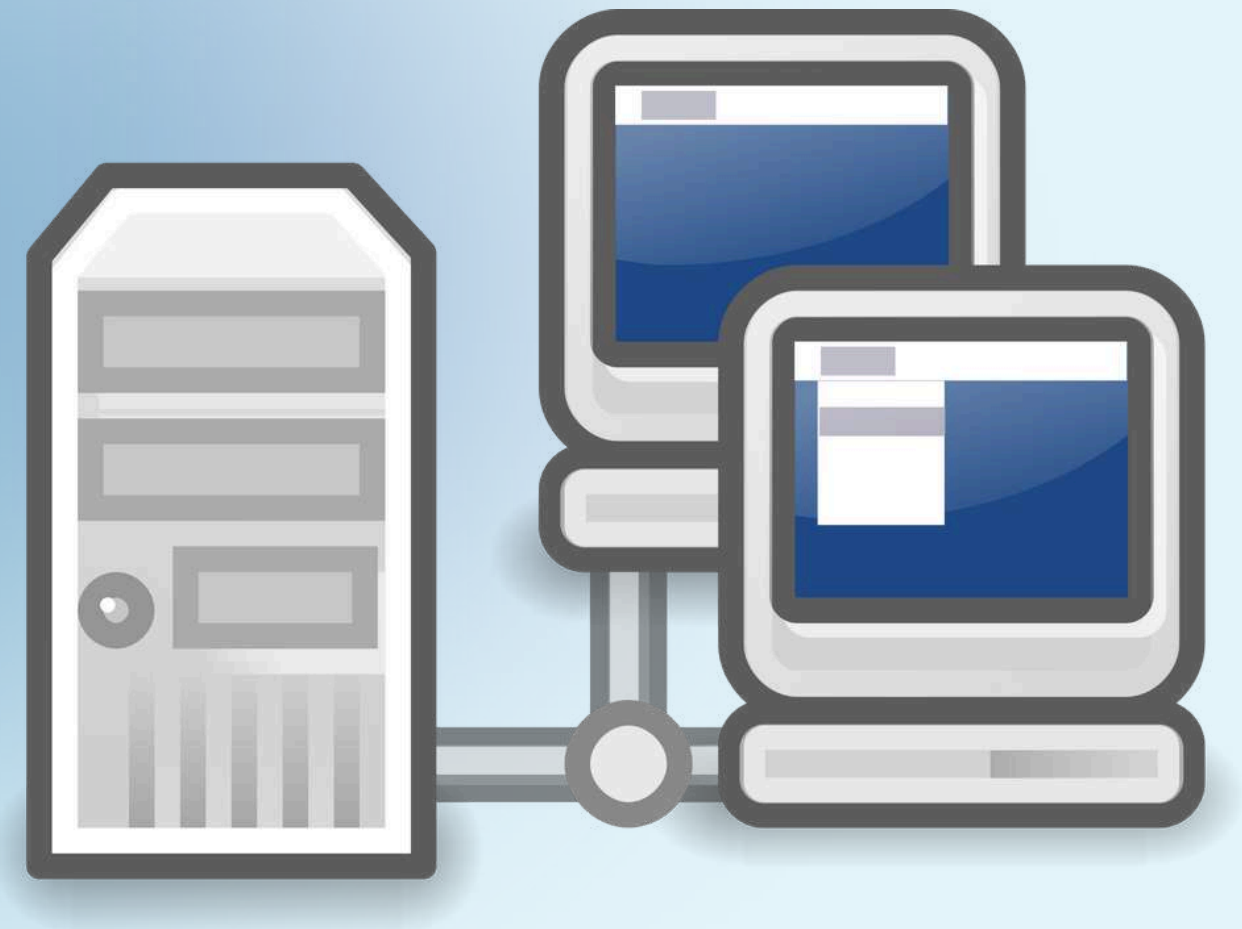
Non-Functional Requirements

Non-functional qualities describe how well the system works - performance, stability, and maintainability.

Reliability
The system runs without failures for long periods.
Consistent and dependable behavior.

Performance
The system responds quickly and uses resources efficiently.
Includes latency, throughput, and memory/CPU usage.

Maintainability
The code is easy to modify, fix, or extend.
Supports clean structure, good naming, and modular design.



Dimensions of Software Testing

Unit Testing

Tests the smallest pieces of code (functions/classes) in complete isolation.

Integration Testing

Checks that multiple components/modules work correctly together.

System Testing

Validates the entire application as a whole in a real-like environment.

Acceptance Testing

Ensures the system meets business requirements and user expectations.

Regression Testing

Verifies that new changes haven’t broken previously working features.

Smoke / Sanity Testing

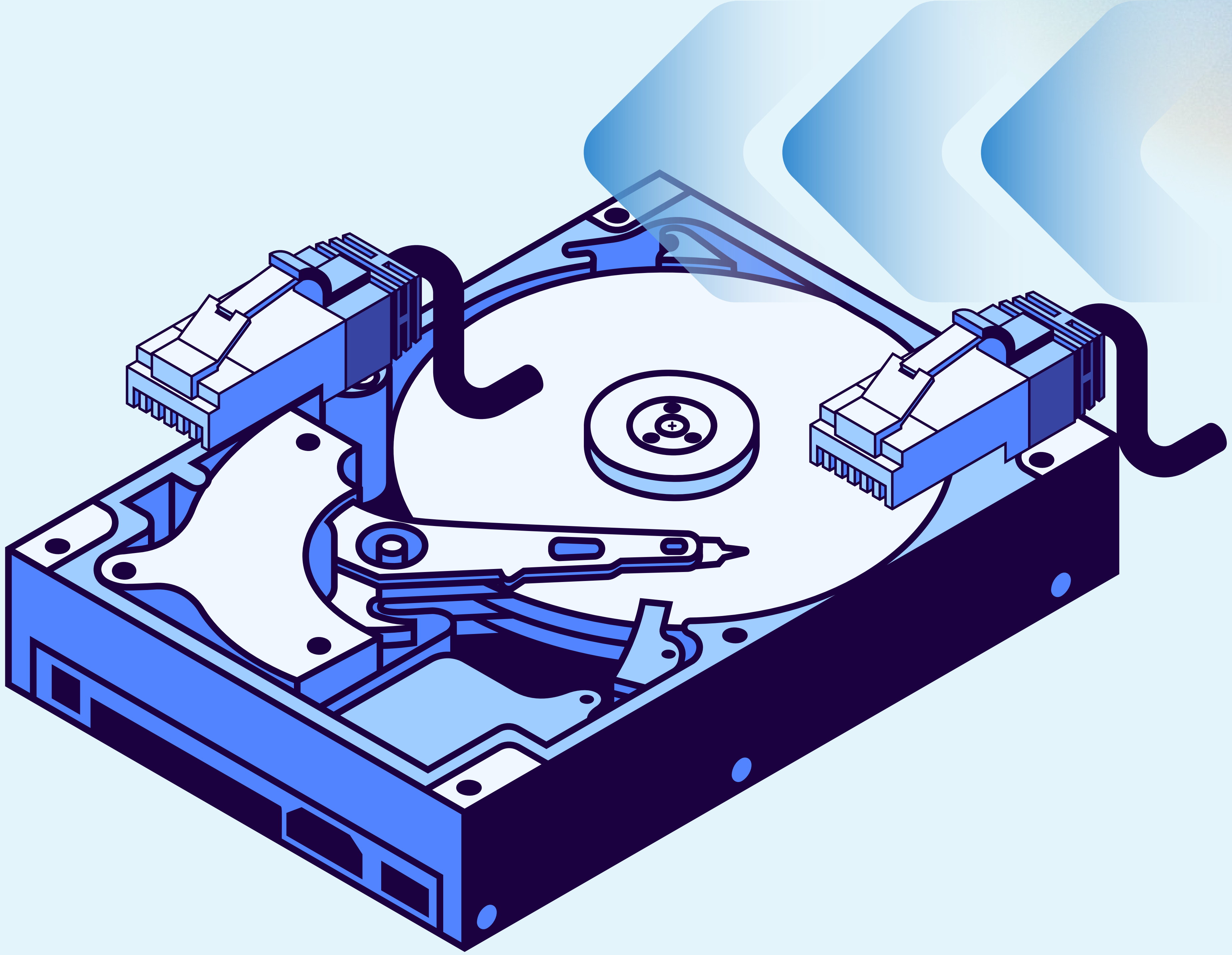
A quick, basic check to confirm that the system is stable enough for deeper testing.

Why Developers Avoid Testing

- Illusion of “moving faster”
- Lack of culture or skill
- Fear of modifying code
- Deadline pressure

What Makes a Good Test?

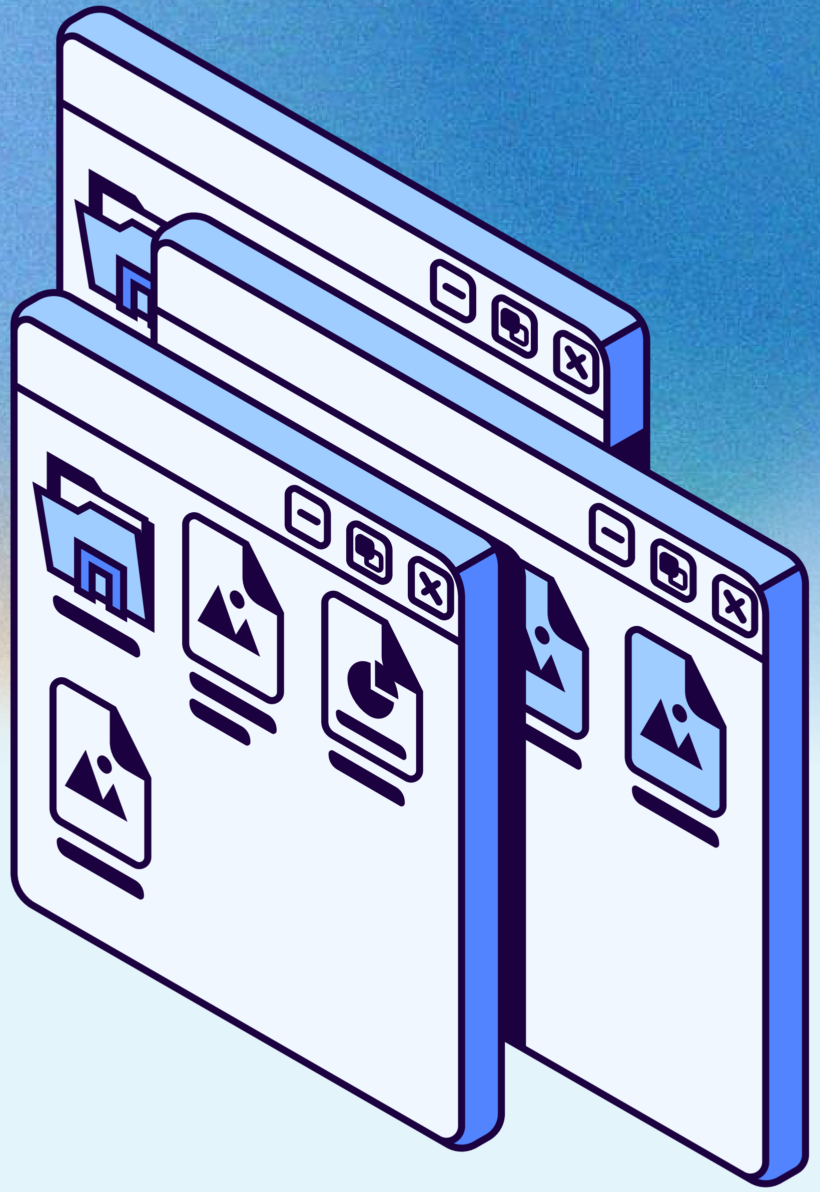
- Clear and simple
- Independent
- Good coverage
- Low maintenance cost
- Fast execution



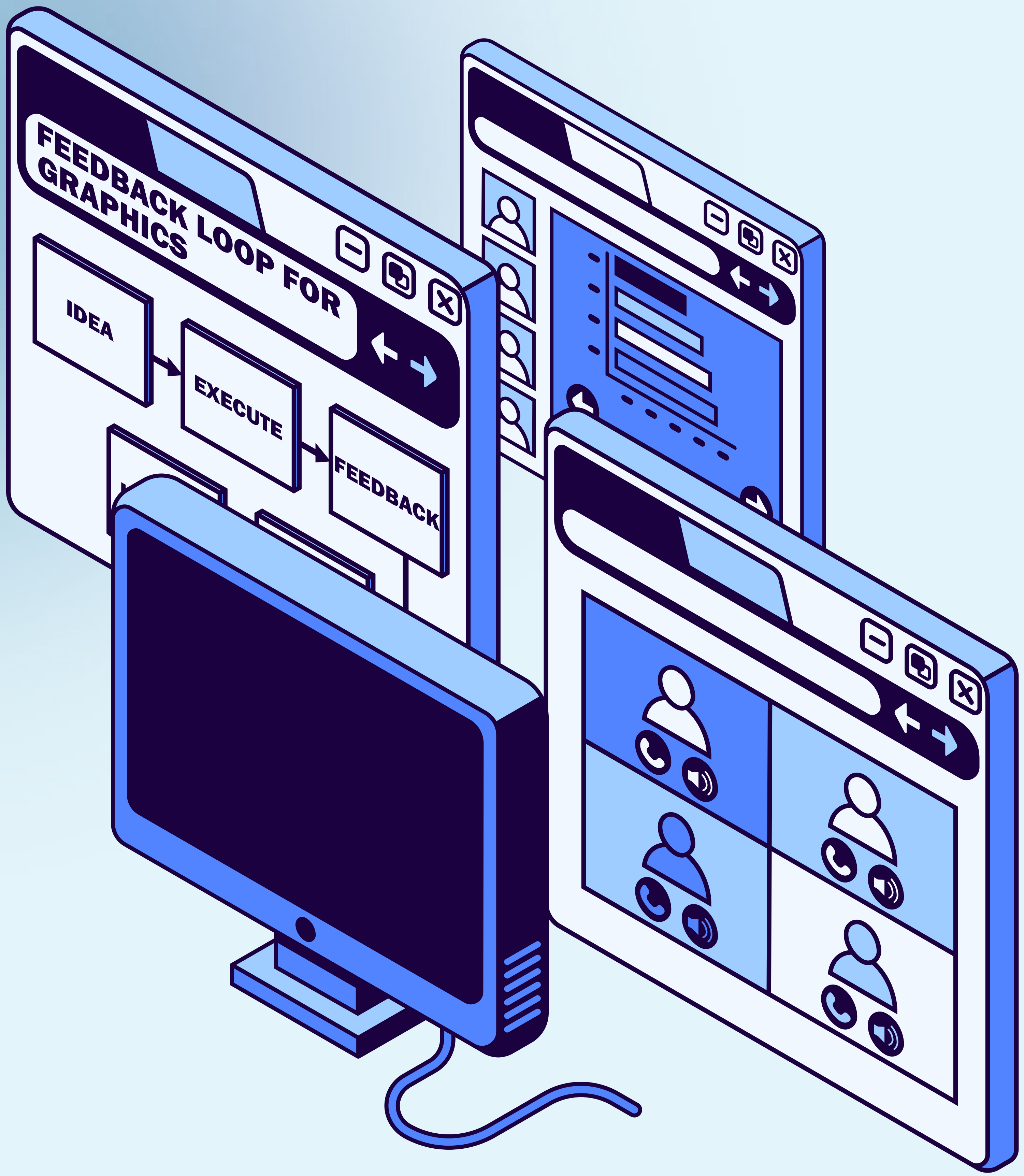
Why Manual Testing Is Not Enough

- Not repeatable
- Slow
- Human error
- Poor coverage
- Not suitable for Agile / CI/CD
- Becomes expensive as project grows

The FIRST Principles



- Fast
- Independent
- Repeatable
- Self-validating
- Timely

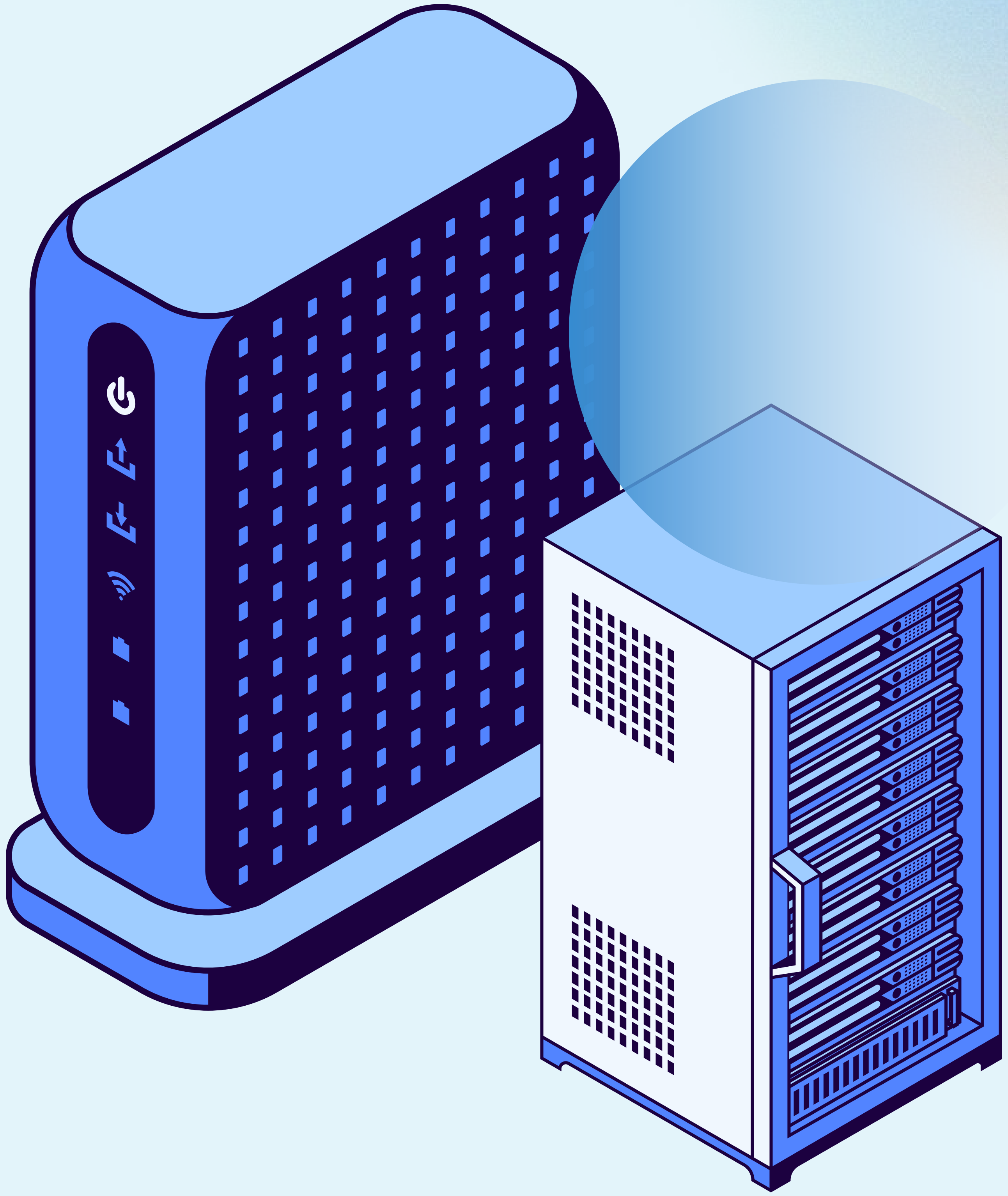


What Is JUnit?

- Most popular testing framework for Java
- Used to write and run unit tests
- Provides annotations like @Test
- Offers rich assertions for checking results
- Integrates with Maven, Gradle, IntelliJ, VS Code, CI/CD

Structure of a JUnit Test

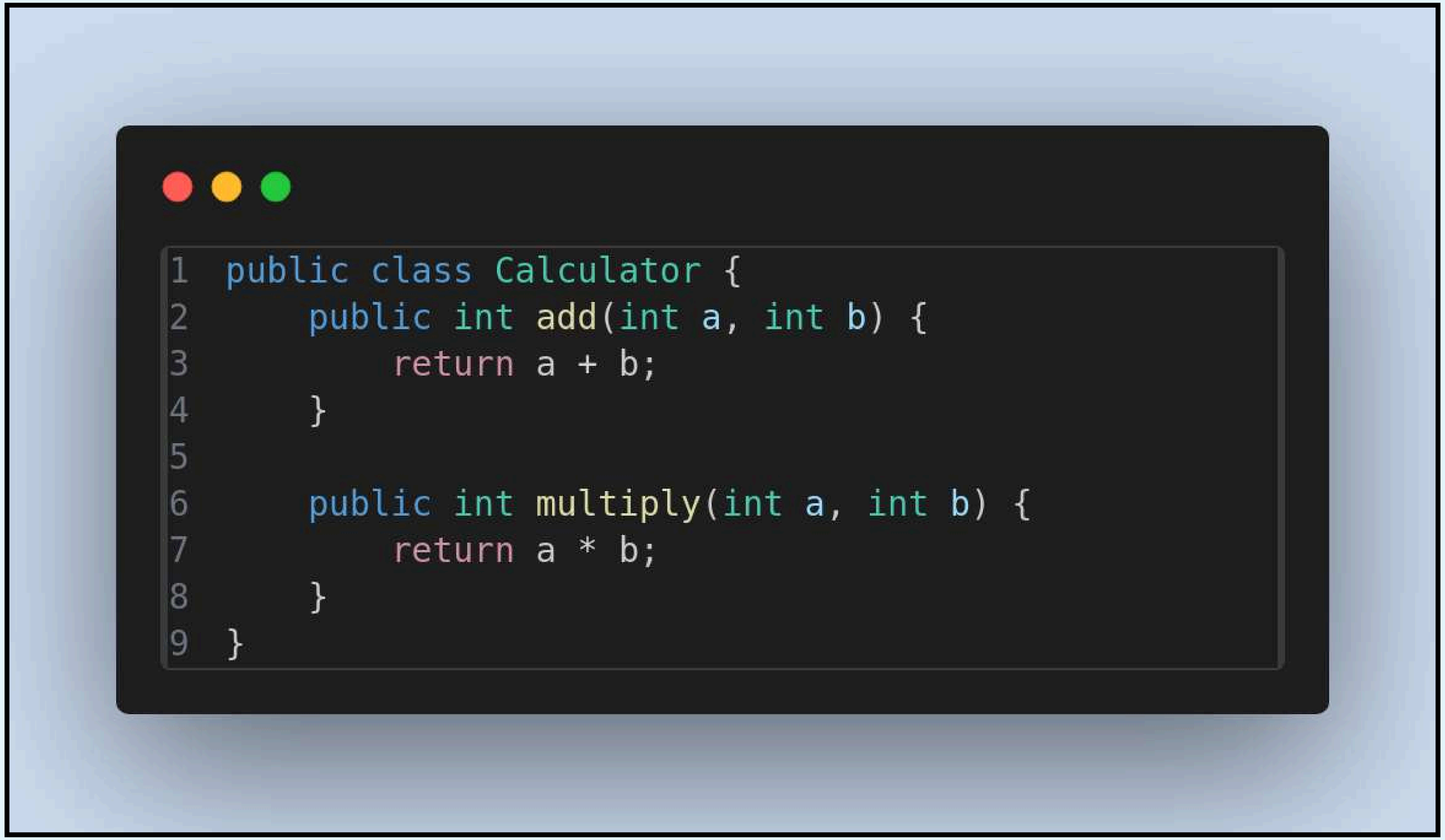
- @Test annotation
- Arrange → Act → Assert pattern
- Clear naming
- Assertions like assertEquals, assertThrows



Test Lifecycle

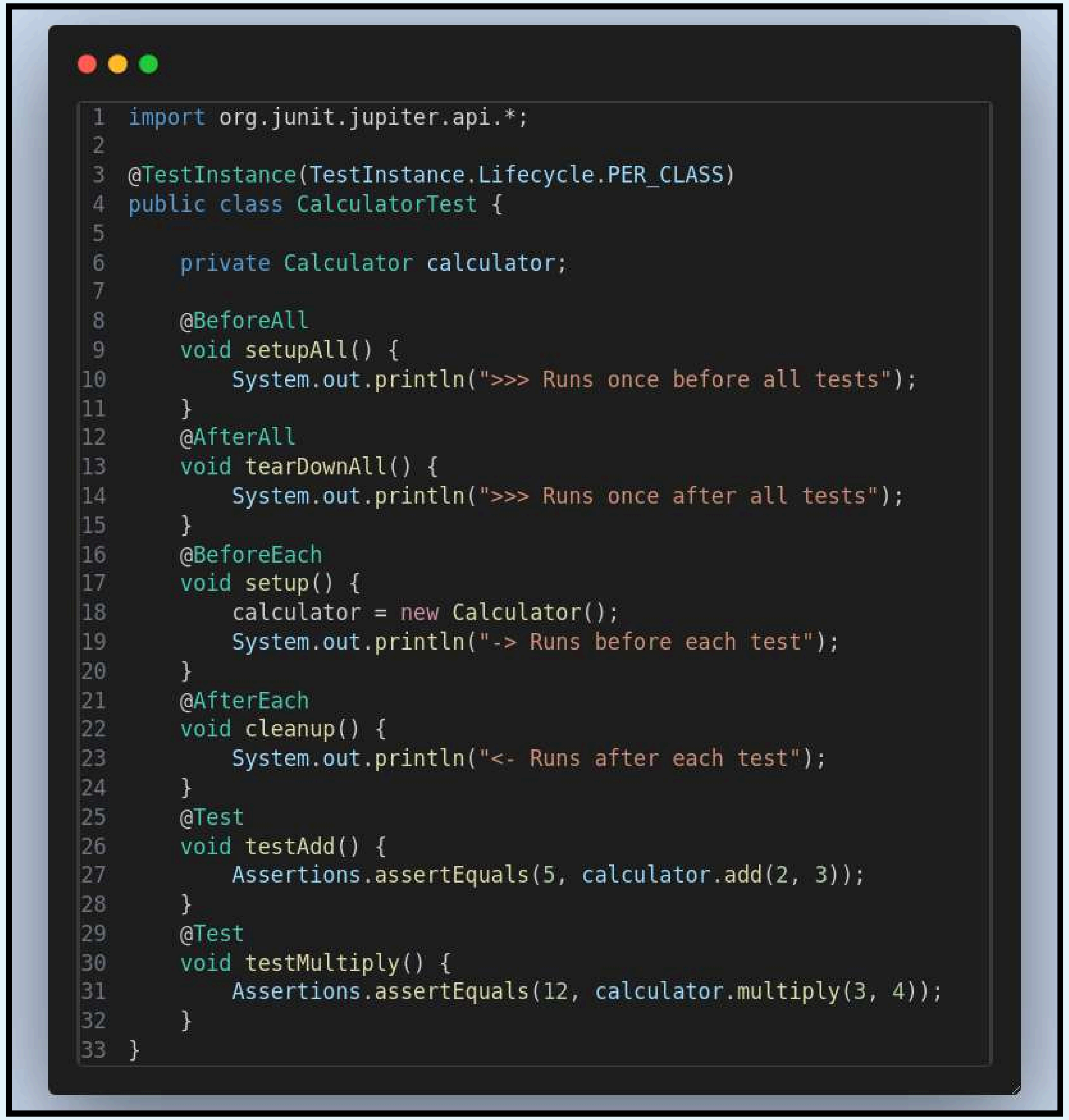
- @BeforeEach
- @AfterEach
- @BeforeAll
- @AfterAll

Example (source code)



```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

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

Example (test)

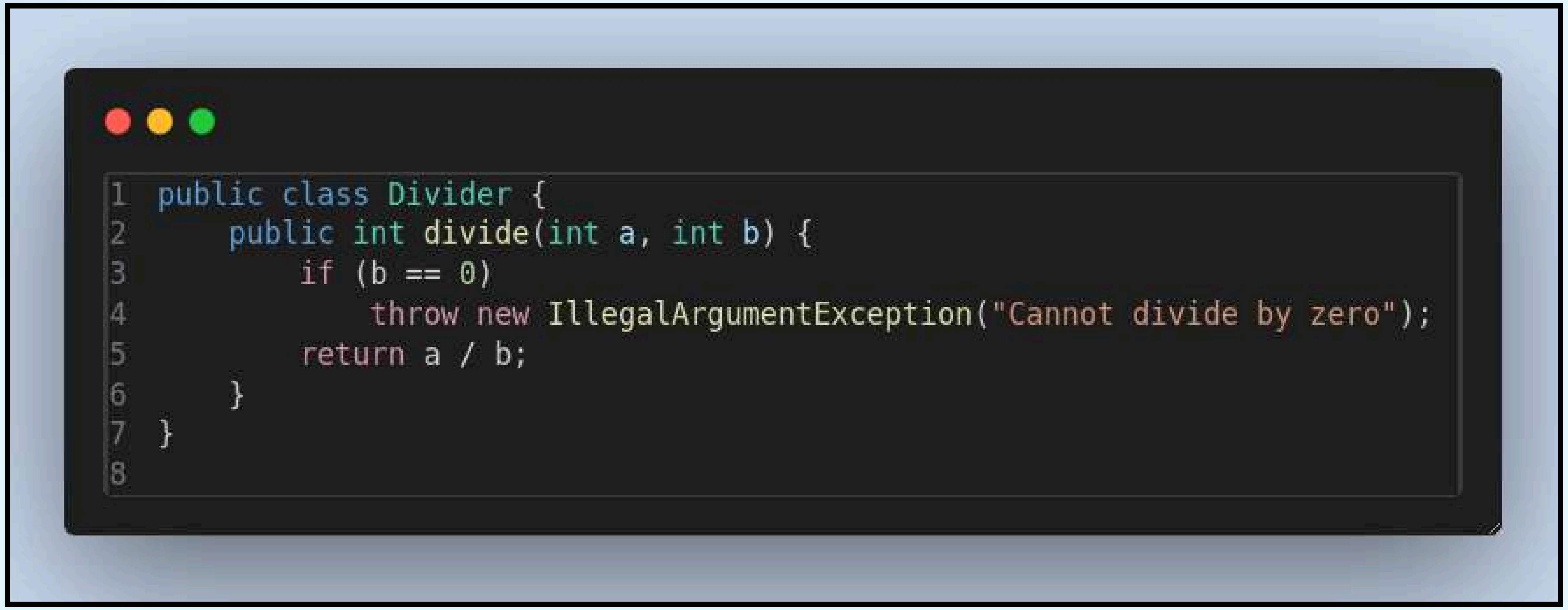


```java
import org.junit.jupiter.api.*;

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class CalculatorTest {

    private Calculator calculator;

    @BeforeAll
    void setupAll() {
        System.out.println(">>> Runs once before all tests");
    }
    @AfterAll
    void tearDownAll() {
        System.out.println(">>> Runs once after all tests");
    }
    @BeforeEach
    void setup() {
        calculator = new Calculator();
        System.out.println("-> Runs before each test");
    }
    @AfterEach
    void cleanup() {
        System.out.println("<- Runs after each test");
    }
    @Test
    void testAdd() {
        Assertions.assertEquals(5, calculator.add(2, 3));
    }
    @Test
    void testMultiply() {
        Assertions.assertEquals(12, calculator.multiply(3, 4));
    }
}
```

Exception Testing

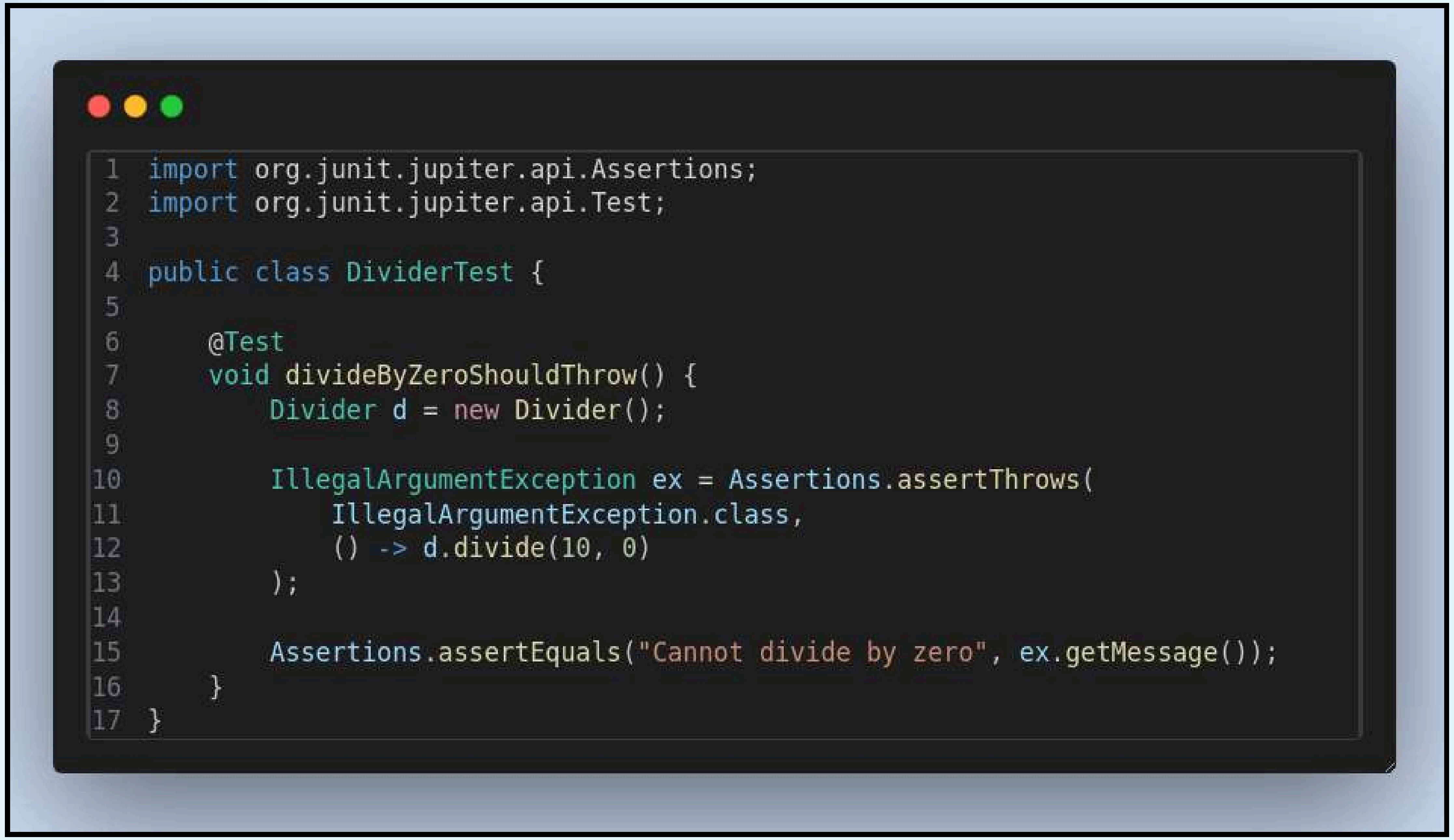
- Use assertThrows()
- Validates behavior under invalid or unexpected conditions
- Ensures proper error handling

Example (source code)



```java
public class Divider {
    public int divide(int a, int b) {
        if (b == 0)
            throw new IllegalArgumentException("Cannot divide by zero");
        return a / b;
    }
}

```

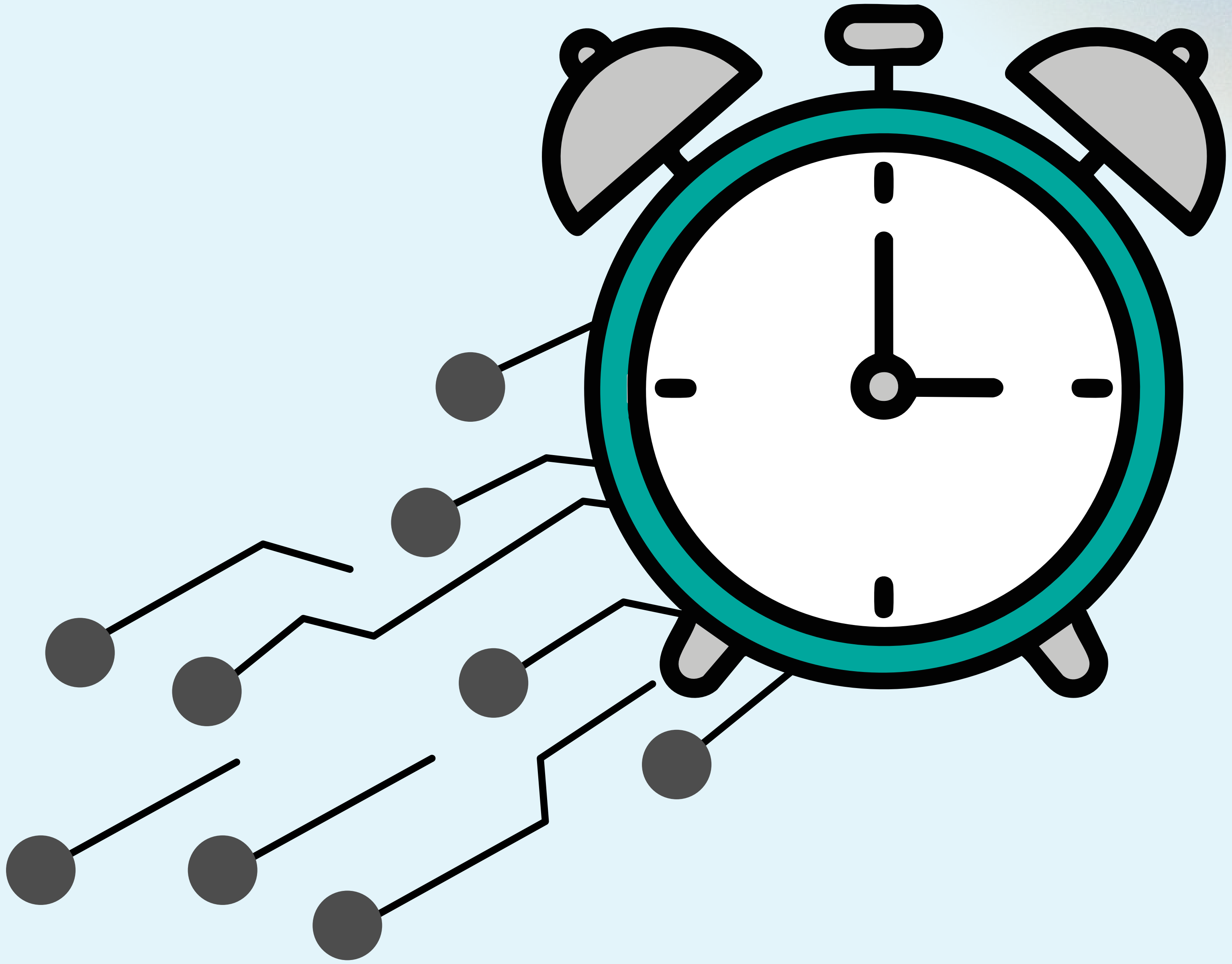
Example (test)



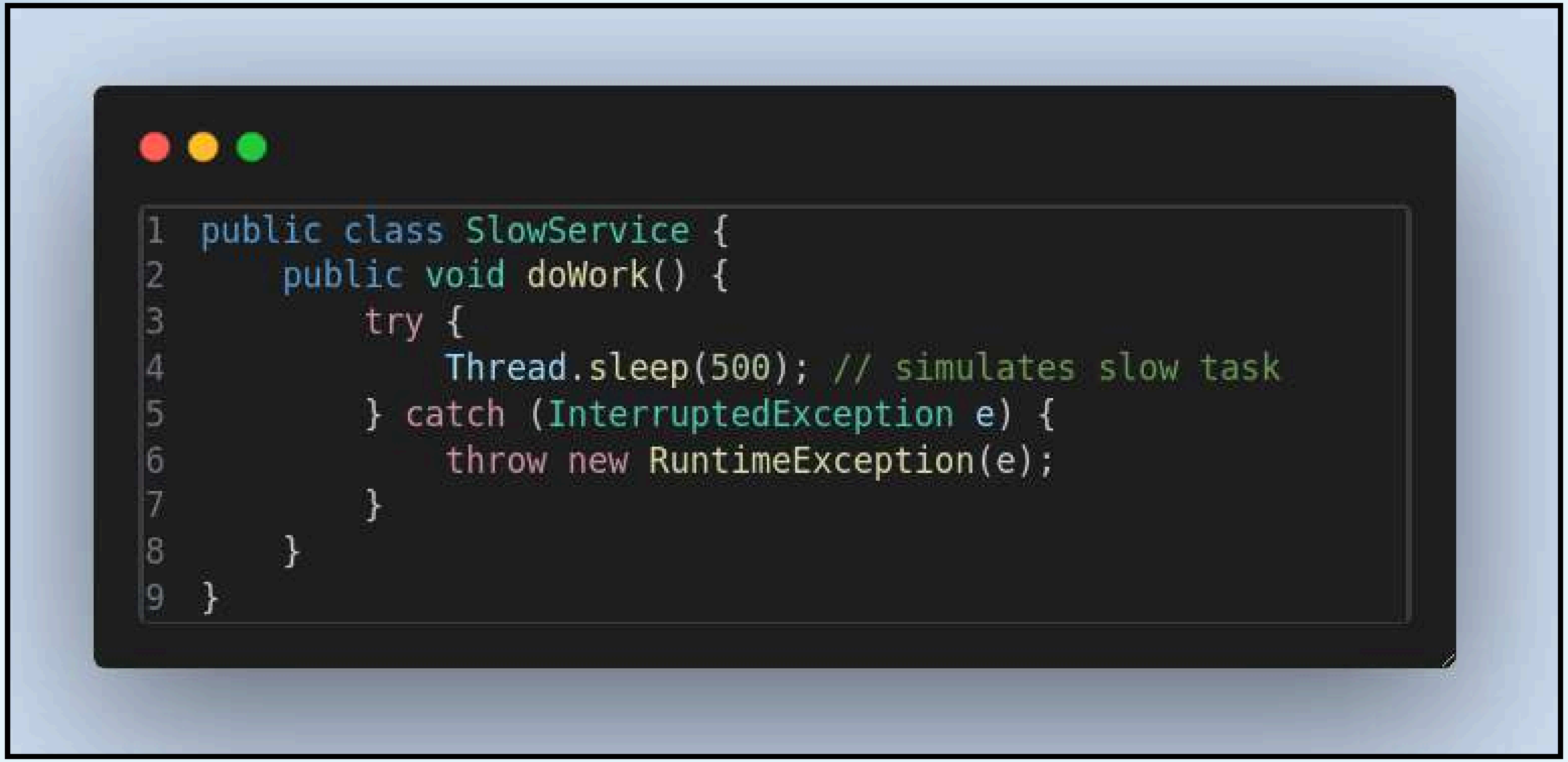
```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class DividerTest {

    @Test
    void divideByZeroShouldThrow() {
        Divider d = new Divider();

        IllegalArgumentException ex = Assertions.assertThrows(
            IllegalArgumentException.class,
            () -> d.divide(10, 0)
        );

        Assertions.assertEquals("Cannot divide by zero", ex.getMessage());
    }
}
```

Timeout Testing

- Use assertTimeout()
- Prevents hanging tests
- Checks basic performance expectations

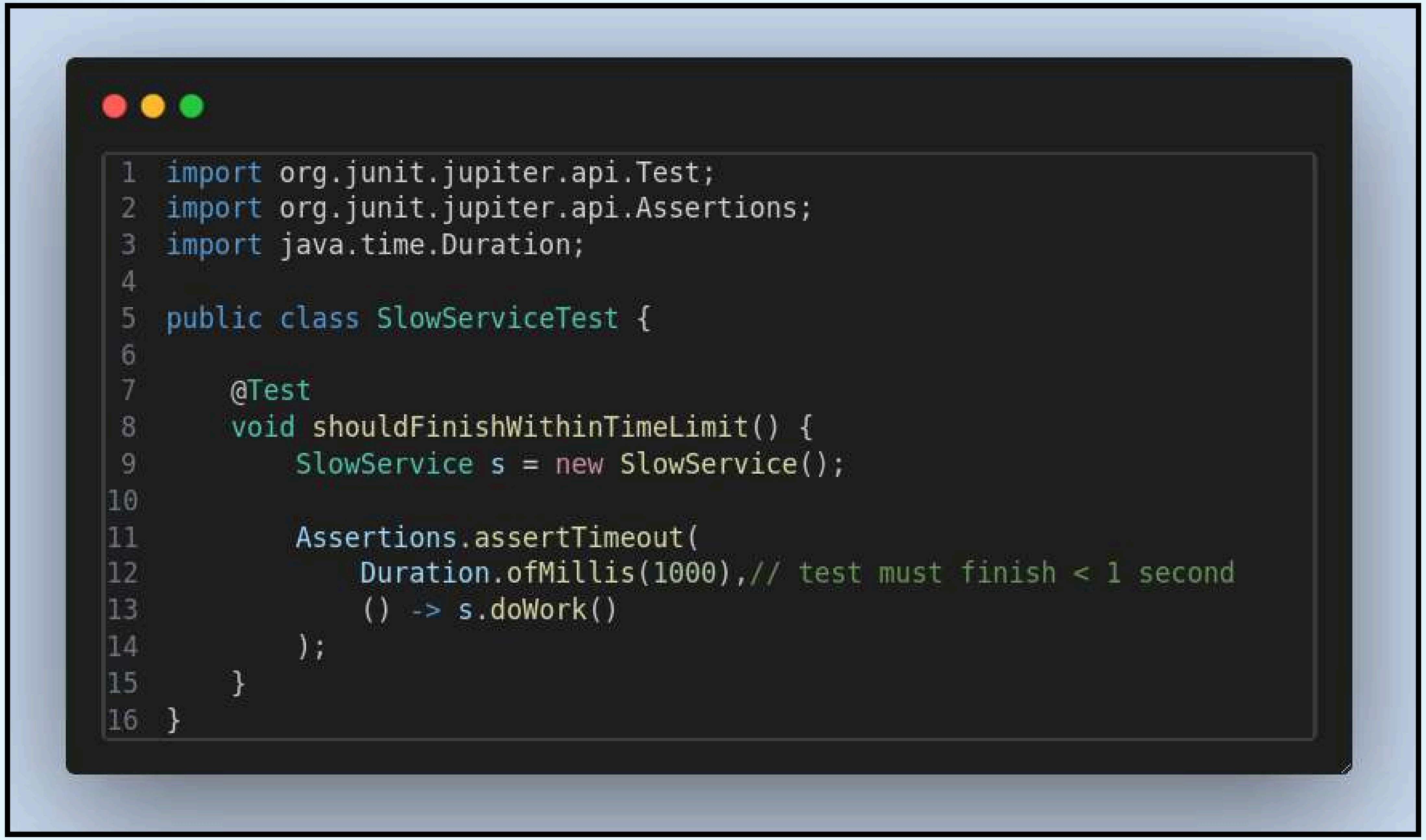


Example (source code)



```java
public class SlowService {
    public void doWork() {
        try {
            Thread.sleep(500); // simulates slow task
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Example (test)

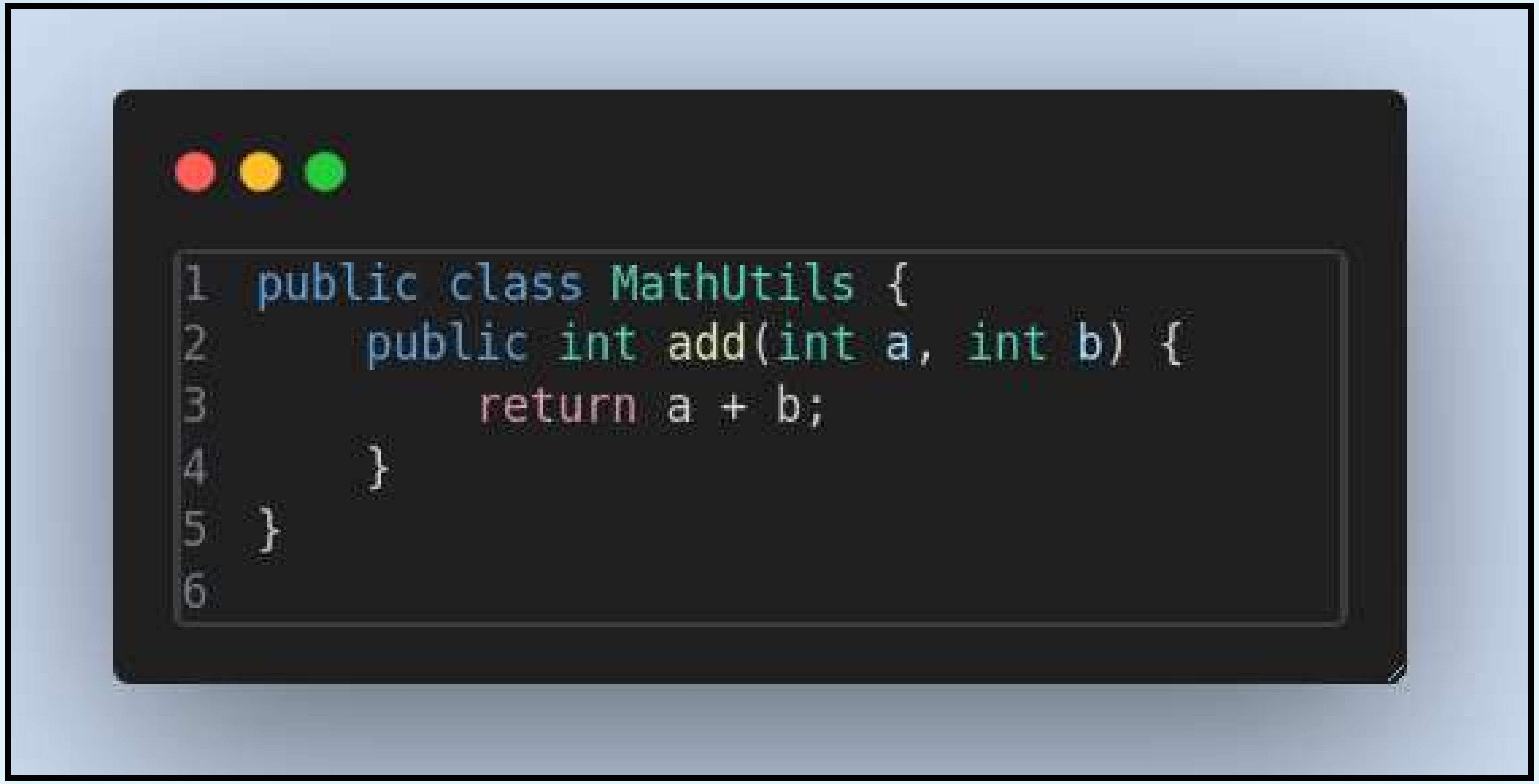


```java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;
import java.time.Duration;

public class SlowServiceTest {

    @Test
    void shouldFinishWithinTimeLimit() {
        SlowService s = new SlowService();

        Assertions.assertTimeout(
            Duration.ofMillis(1000),// test must finish < 1 second
            () -> s.doWork()
        );
    }
}
```

Test Categorization

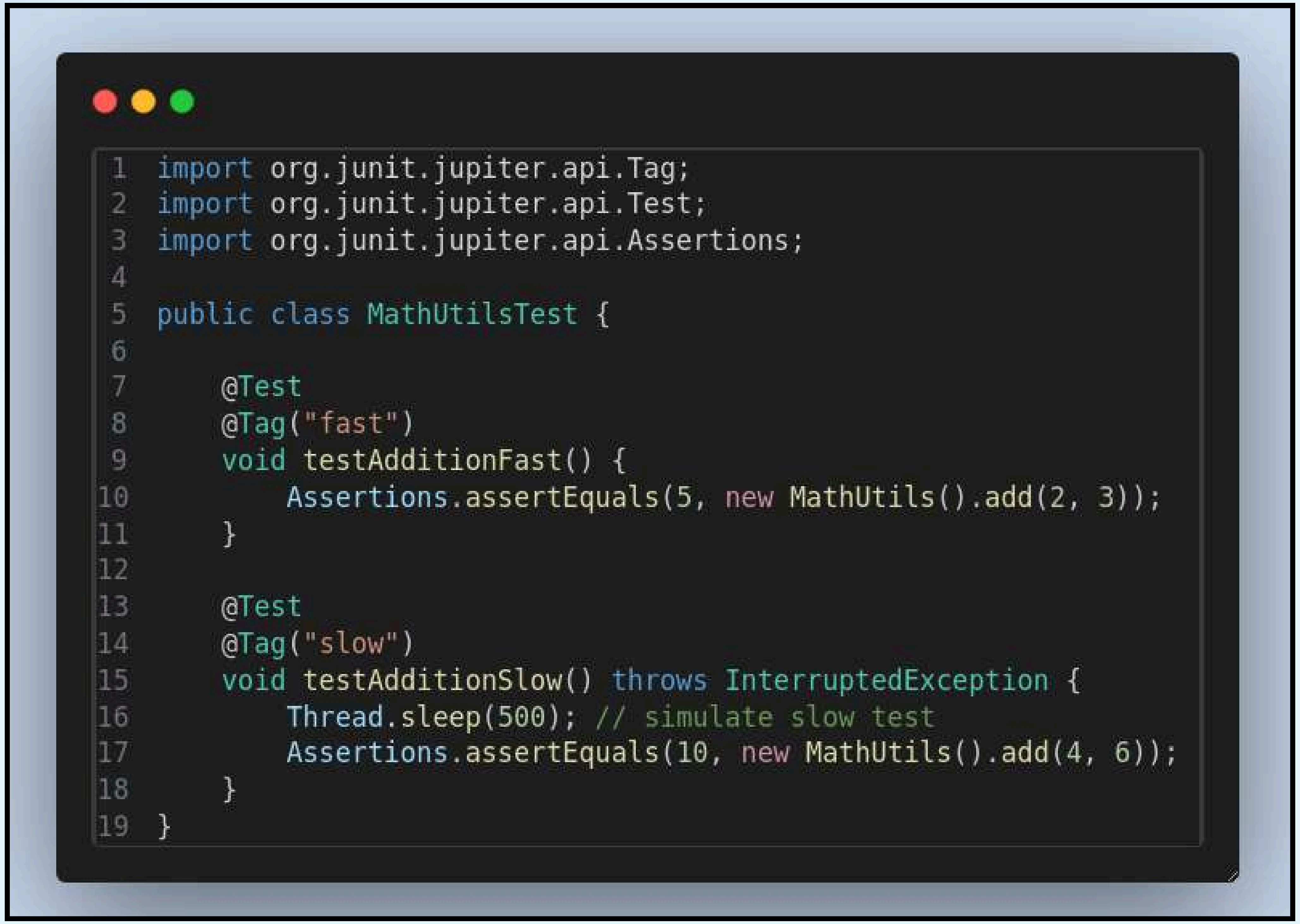
- Use @Tag
- Group tests
- Run specific test groups in CI/CD

Example (source code)

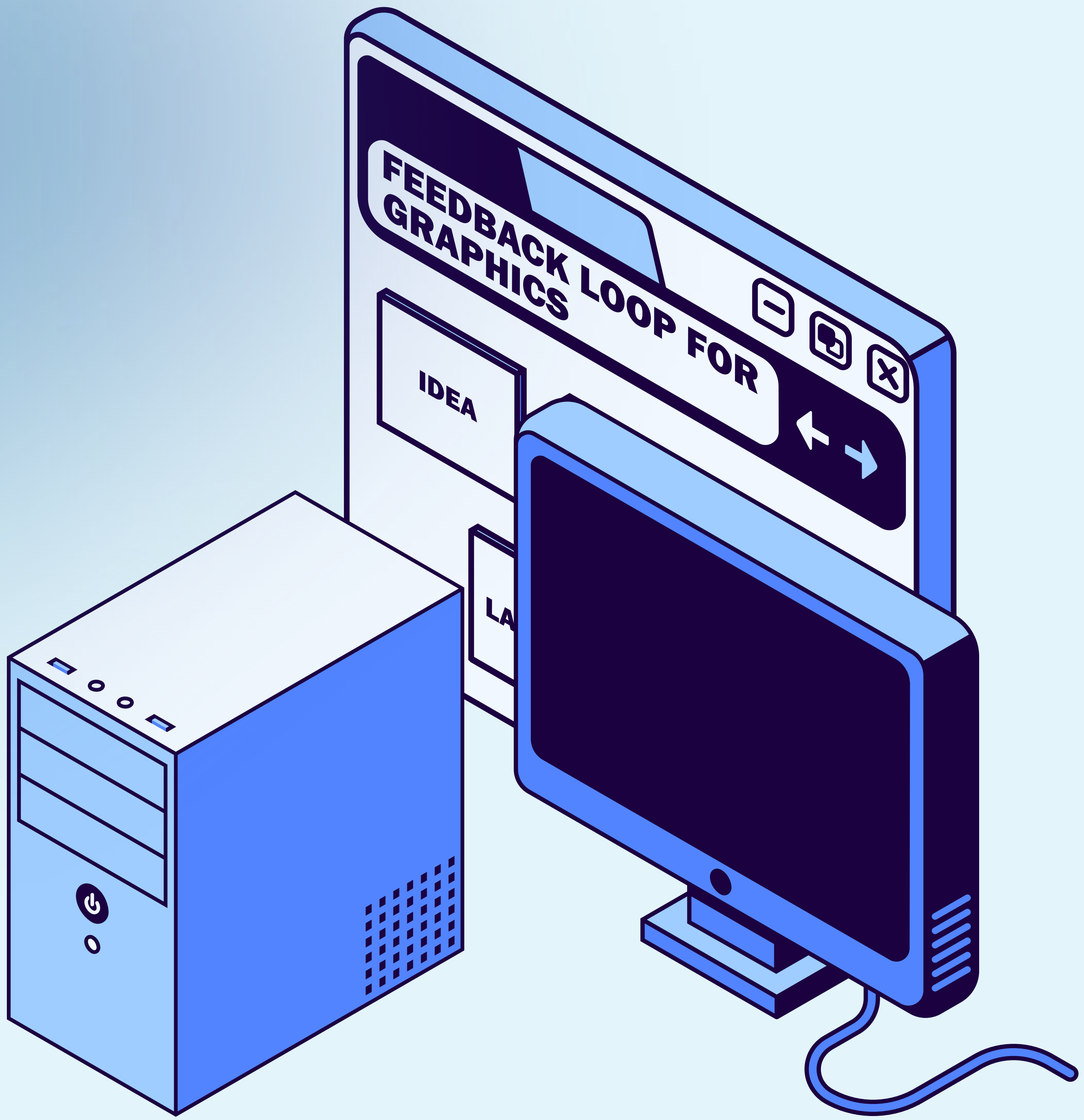


```java
public class MathUtils {
    public int add(int a, int b) {
        return a + b;
    }
}

```

Example (test)

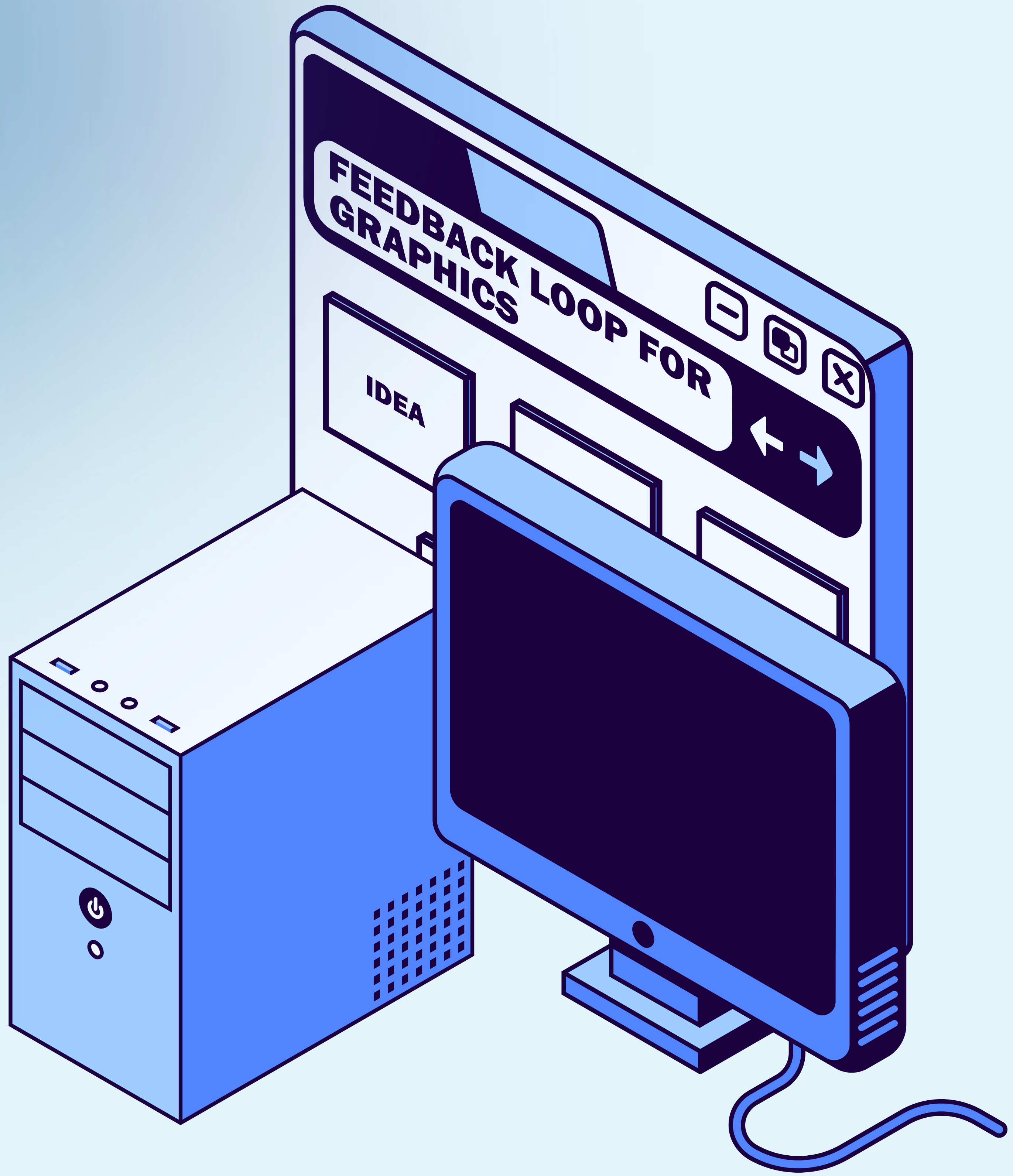


```java
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;

public class MathUtilsTest {

    @Test
    @Tag("fast")
    void testAdditionFast() {
        Assertions.assertEquals(5, new MathUtils().add(2, 3));
    }

    @Test
    @Tag("slow")
    void testAdditionSlow() throws InterruptedException {
        Thread.sleep(500); // simulate slow test
        Assertions.assertEquals(10, new MathUtils().add(4, 6));
    }
}
```



Setting Up JUnit in Maven

- Add dependency to pom.xml
- Use maven-surefire-plugin
- Provides annotations like @Test
- Standard directories:
  - src/main/java
  - src/test/java



Setting Up JUnit in Gradle

- Use testImplementation
- Run tests with Gradle test runner
- Configure source sets if needed

Java Tips for Better Testing

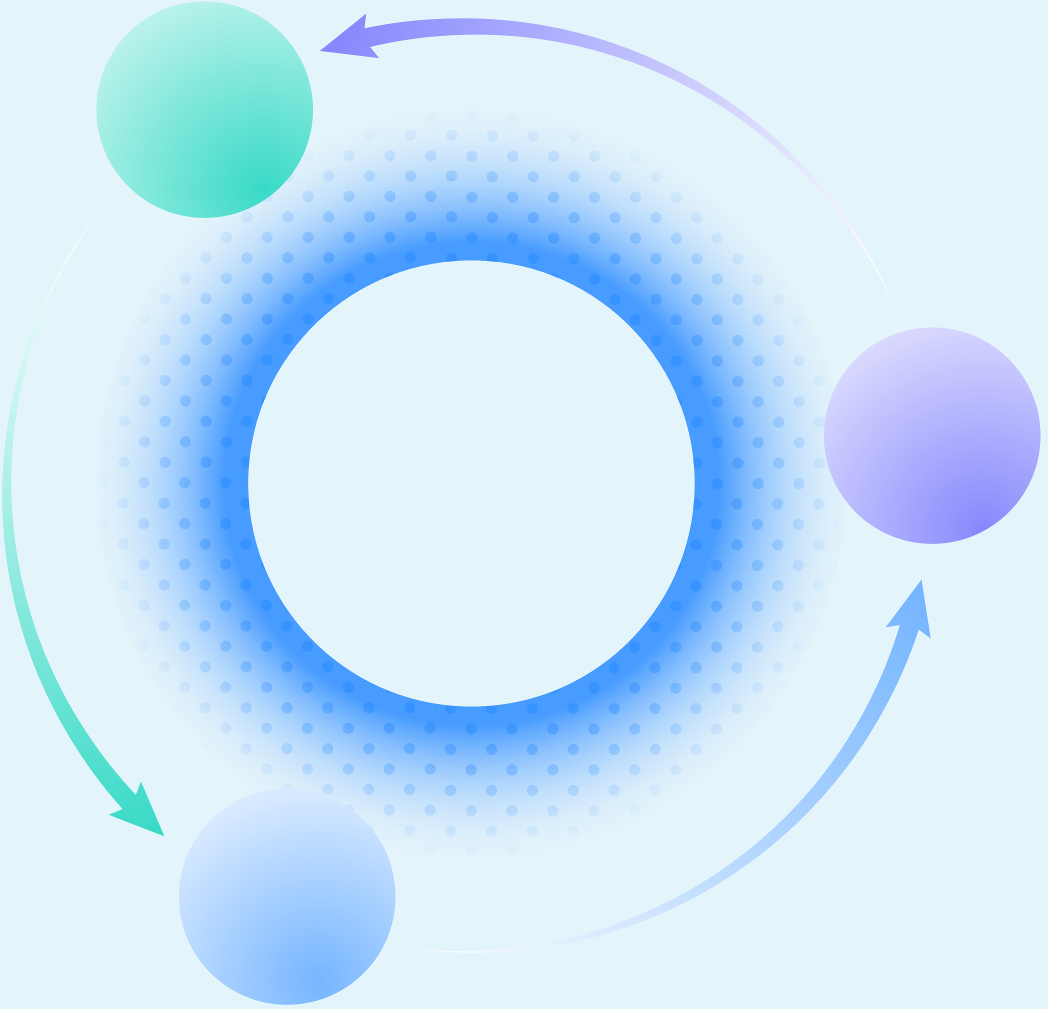
- Use immutability
- Handle exceptions properly
- Implement equals and hashCode correctly
- Be careful with static methods
- Use dependency injection

What Is TDD?

Test-Driven Development (TDD)

- A development method where tests are written before writing code
- Ensures clarity of requirements
- Helps avoid over-engineering
- Produces clean, modular, testable code

The TDD Cycle (Red → Green →Refactor)

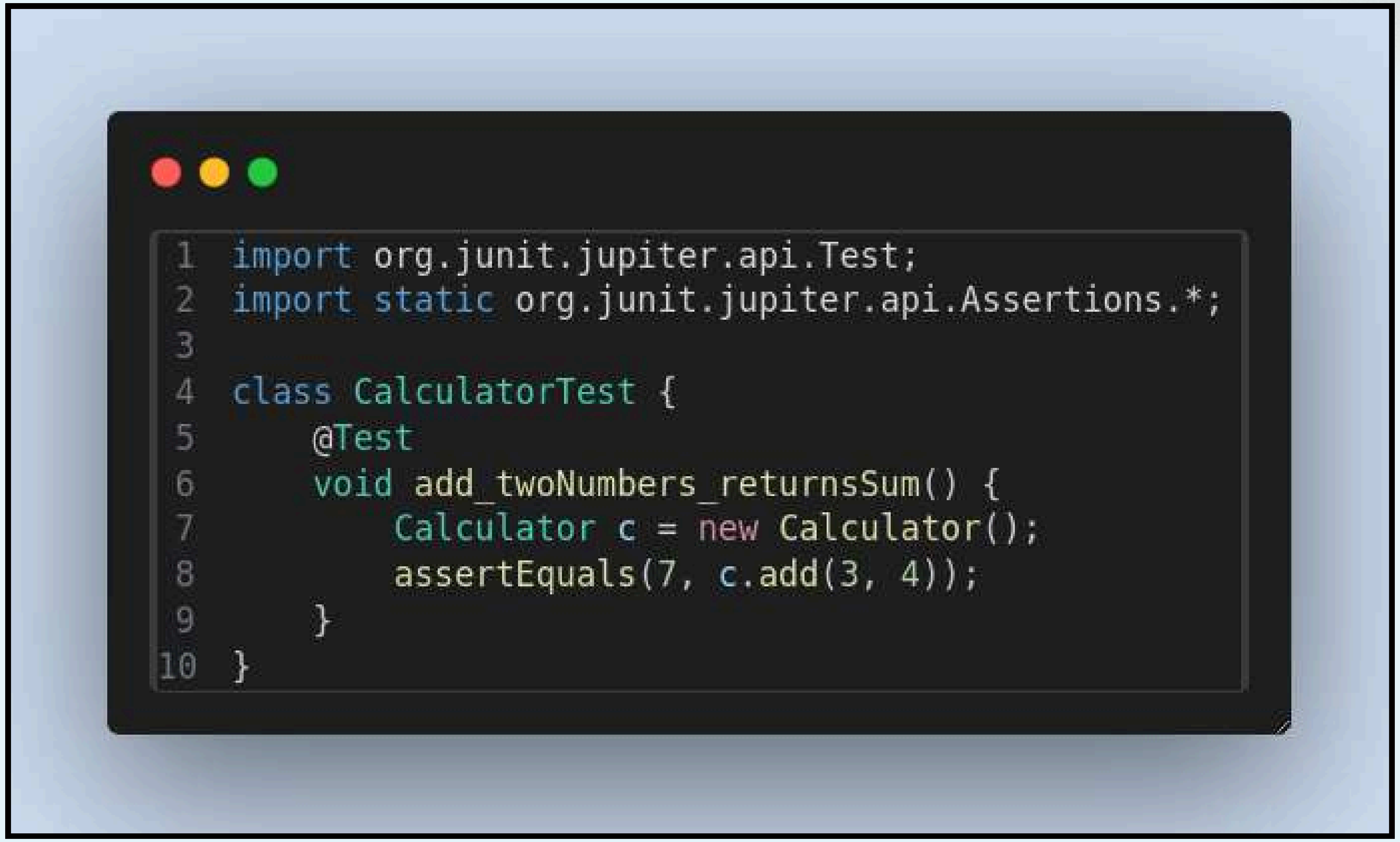


- **Red**: write a failing test
- **Green**: write minimum code to make the test pass
- **Refactor**: improve the code while keeping tests green
- **Repeat for every feature**

Simple TDD Example

(Java + JUnit)

Test (write first):

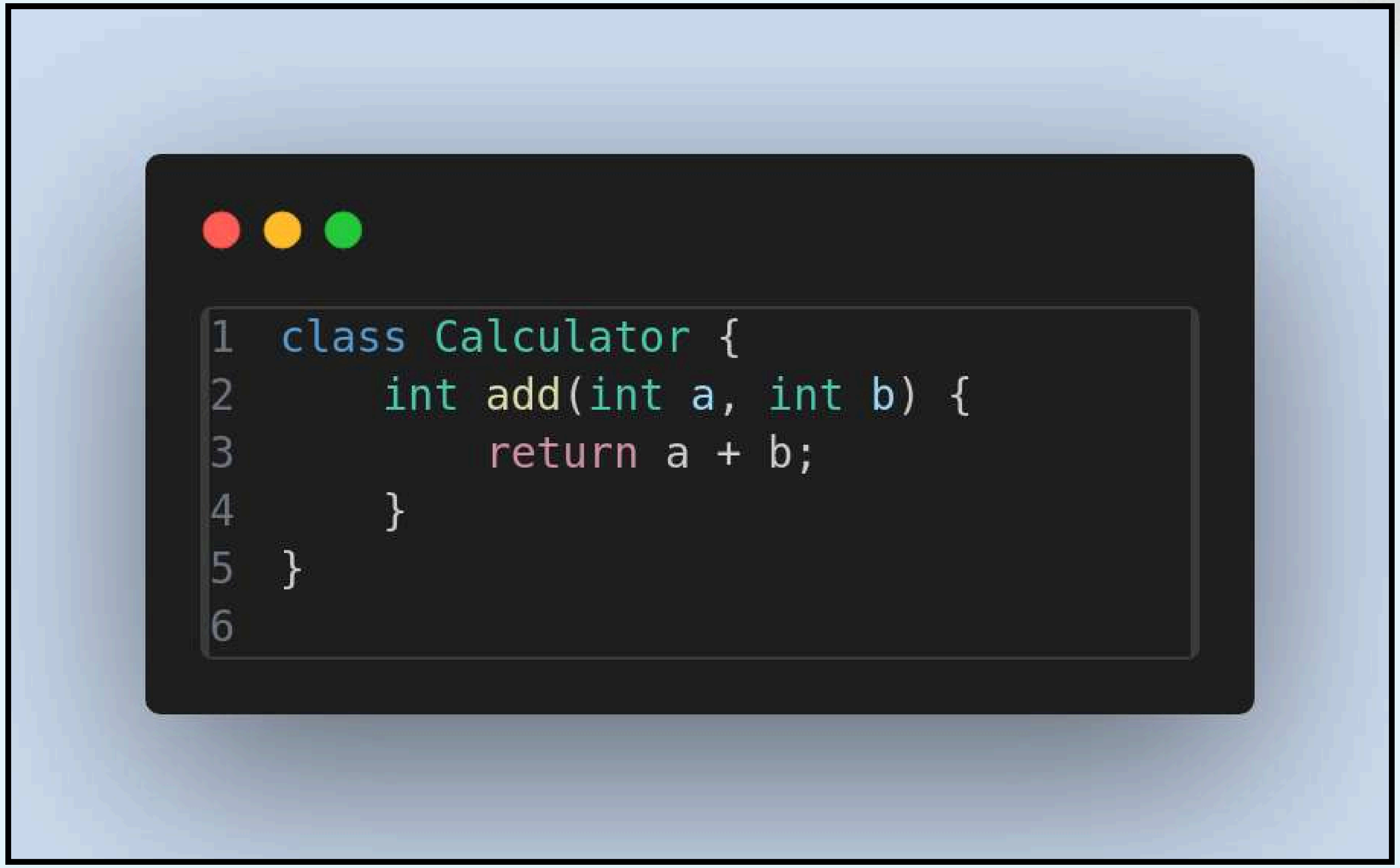


```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {
    @Test
    void add_twoNumbers_returnsSum() {
        Calculator c = new Calculator();
        assertEquals(7, c.add(3, 4));
    }
}
```

Simple TDD Example

(Java + JUnit)

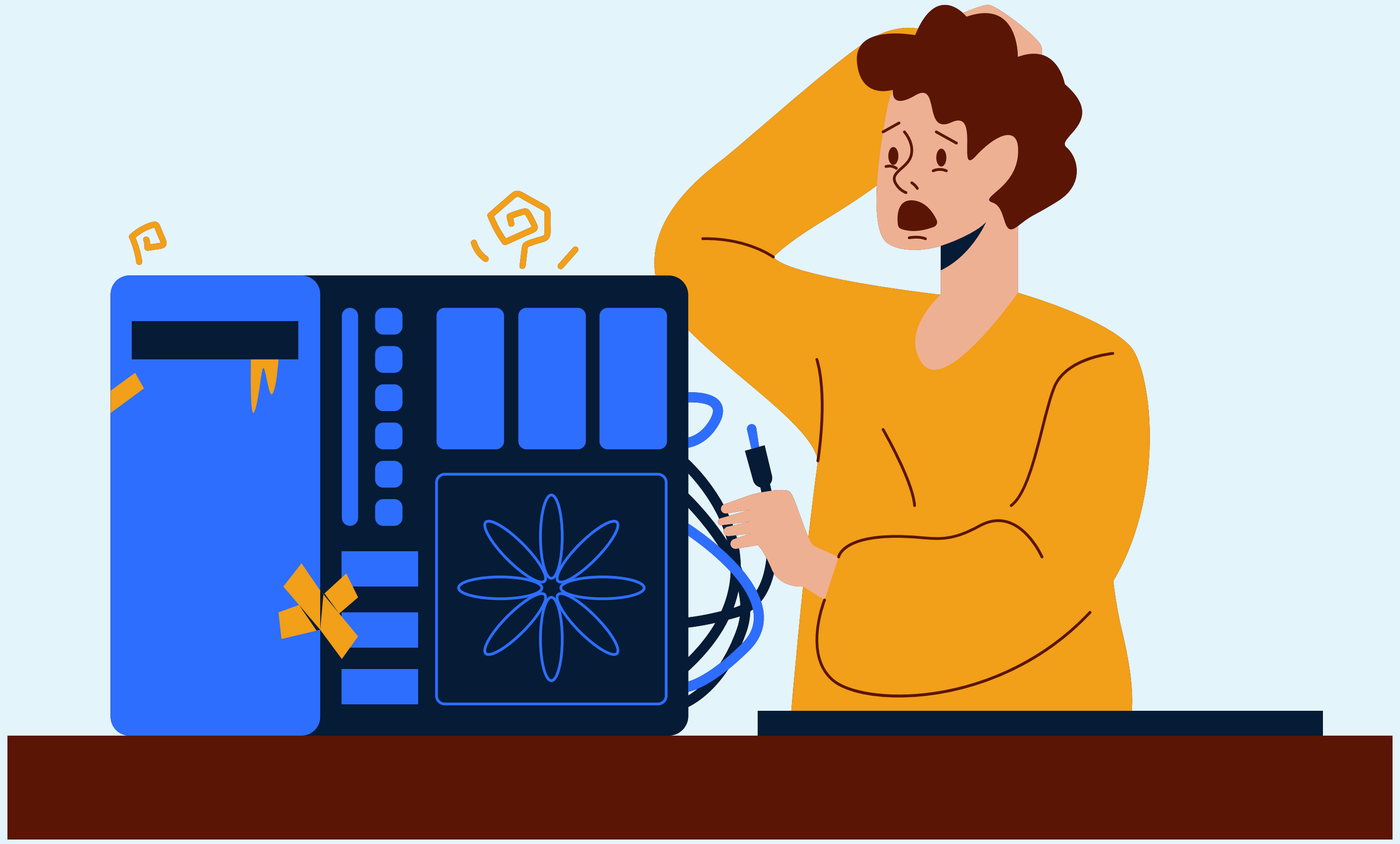
Code
(written after test fails)



```java
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}

```

Why TDD Works

- Forces good design early
- Gives immediate feedback
- Implement equals and hashCode correctly
- Prevents fear of changing code
- Creates a safety net for refactoring
- Reduces debugging time dramatically



Conclusion

- Unit testing builds confidence and catches bugs early
- JUnit makes testing simple, fast, and structured
- TDD improves design, keeps code clean, and reduces rework
- Small, focused tests lead to reliable and maintainable software
- Testing is not extra work – it’s part of writing good code

Thank You!

Thank you for your attention. We hope this presentation enhanced your understanding of unit testing