howest hogeschool

Object Oriented Architectures and Secure Development

Unit testing

Frédéric Vlummens
Matthias Blomme
Mattias De Wael

howest
hogeschool

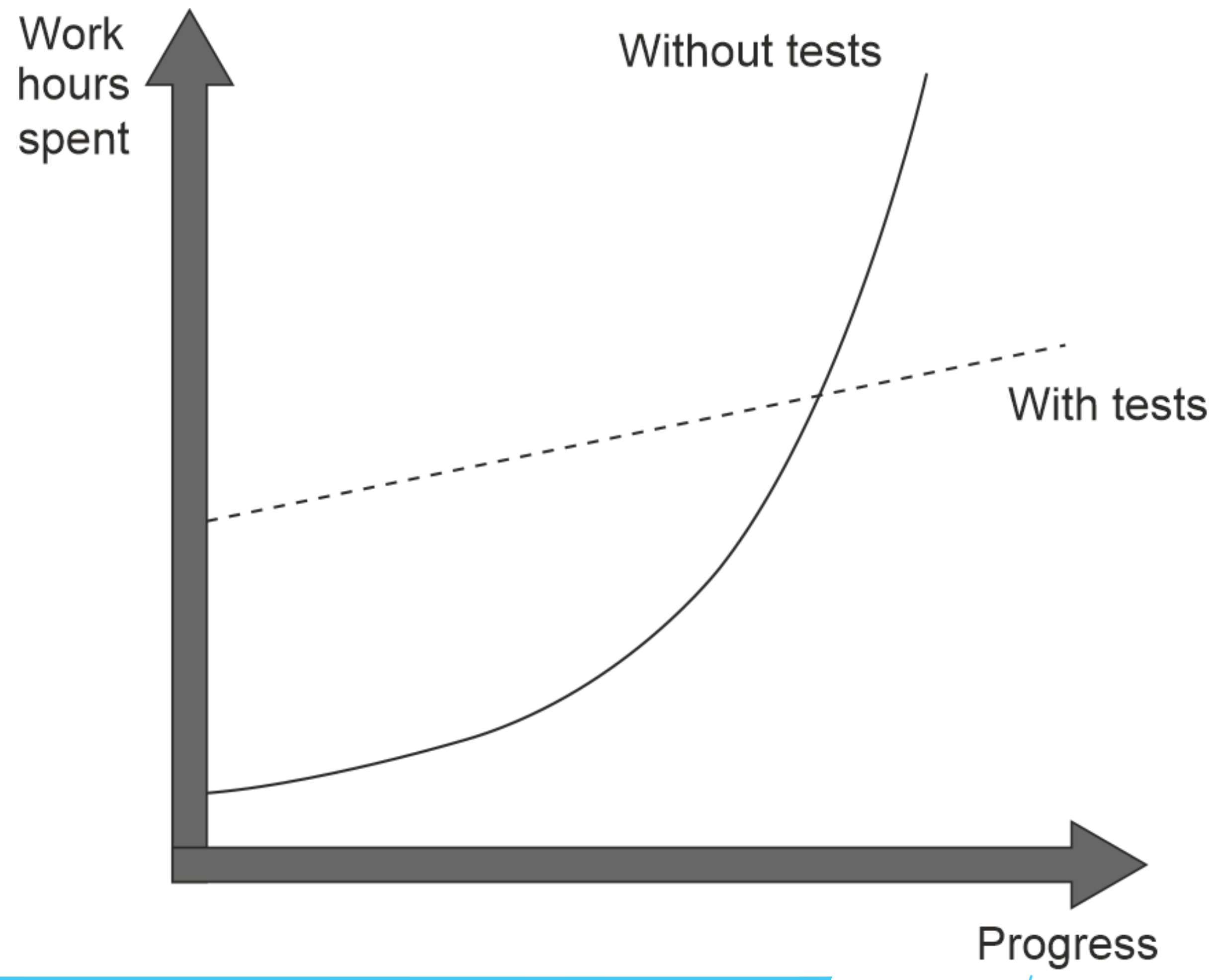
Unit testing

the goal of unit testing

The goal of unit testing

- Enable sustainable growth of the software project.
- Avoid the stagnation phase and maintain the development pace over time.
- Trust that your changes won't lead to regressions/bugs.
- A tool that provides insurance against a vast majority of **regressions**.

It's quite easy to grow a project, especially when you start from scratch.
It's much harder to sustain this growth over time.



All tests are not created equal.

- Each test has a cost and a benefit component.
- You need to carefully weigh one against the other.
- Keep only tests of positive net value in the suite and get rid of all others.
- Some of the unit tests costs:
  - **Refactoring** the test when you refactor the underlying code
  - **Running** the test on each code change
  - **Dealing** with false alarms raised by the test
  - **Spending** time reading the test when you’re trying to understand how the underlying code behaves

What are the other parts of a typical code base?

- Infrastructure code
- External services and dependencies, such as the database and third-party systems
- Code that glues everything together

What makes a successful test suite.

1. It’s integrated into the development cycle.

2. It targets only the most important parts of your code base.
    1. **Testing business logic gives you the best return on your time investment.**

3. It provides maximum value with minimum maintenance costs.

Unit testing

What is a unit test?

Definition of a unit test

- There are a lot of definitions of a unit test.
- In essence each definition includes:
    - A unit test is an automated test that
    - Verifies a small piece of code (a unit)
    - Does it quickly,

Small piece of code?

- Tests shouldn’t verify ***units of code*.**
- Rather, they should verify ***units of behavior.***
- Something that is meaningful for the **problem domain** and a **business person**.
- The number of classes it takes to implement such a unit of behavior is irrelevant.
- The unit could span across multiple classes or only one class, or even take up just a tiny method.

Isolated tests?

- In these slides isolation means:
  - Unit tests themselves should be run in isolation from each other.
  - Ability to run the tests in parallel, sequentially, and in any order, without affecting each other's outcome.
- Isolation is achieved by identifying and acting upon (shared) dependencies.

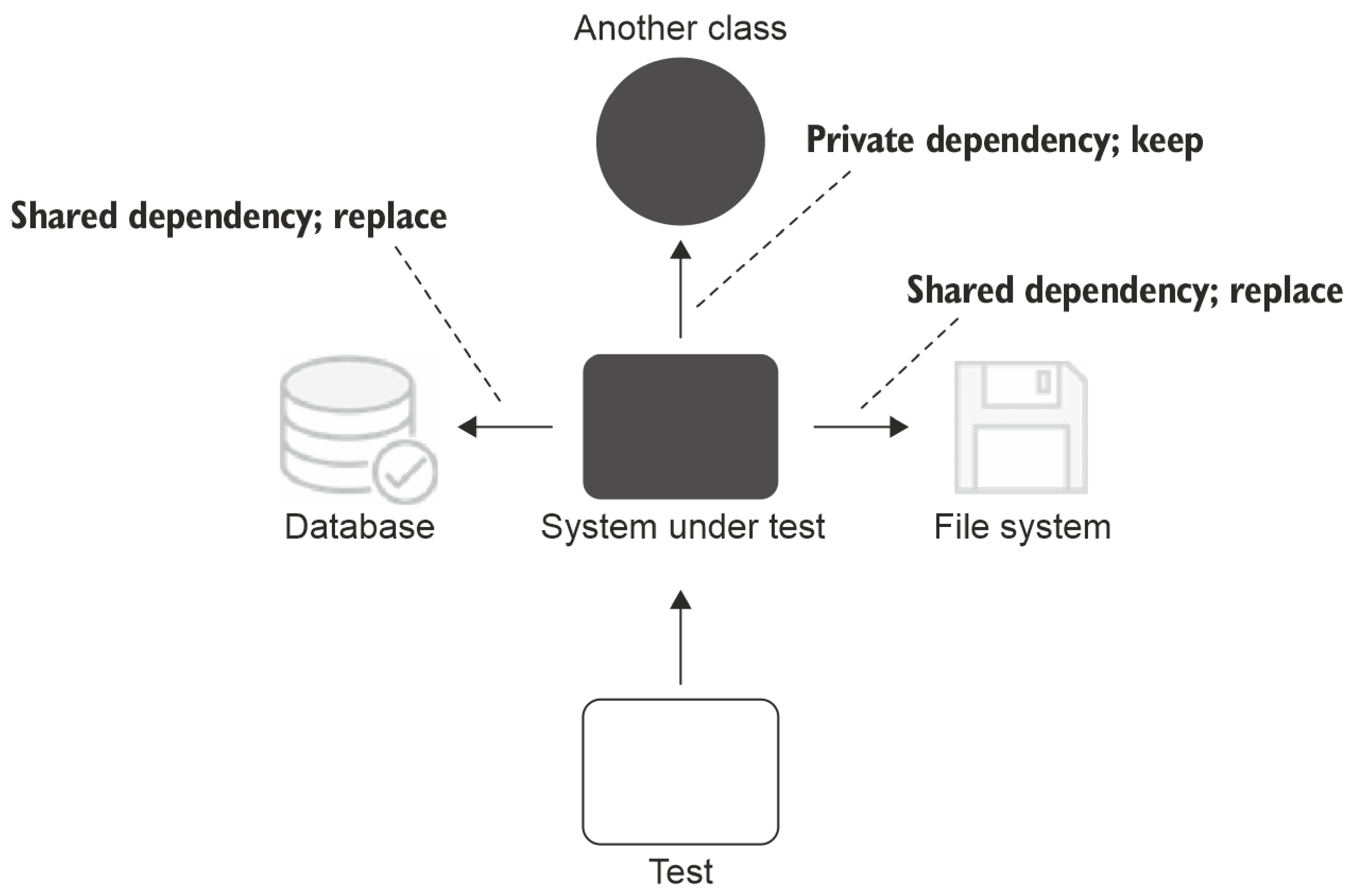
Shared, private, and out-of-process dependencies

- ***Shared dependency***
  - a dependency that is shared **between tests.**
  - e.g. a static mutable field.
    A change to such a field is visible across all unit tests running within the same process.
  - Another example: a database.

- ***Private dependency*** is a dependency that is not shared.

- ***Out-of-process dependency*** is a dependency that runs outside the application's execution process.
  - An *out-of-process dependency* corresponds to a shared dependency in most cases.
  - e.g. a database is both out-of-process and shared. But if you launch that database in a Docker container or virtual machine before each test run, that will make this dependency out-of-process but not shared.
    - Tests no longer work with the same instance of it.
  - e.g. a **read-only** database is also out-of-process but not shared, even if it's reused by tests.

Shared, private, and out-of-process dependencies

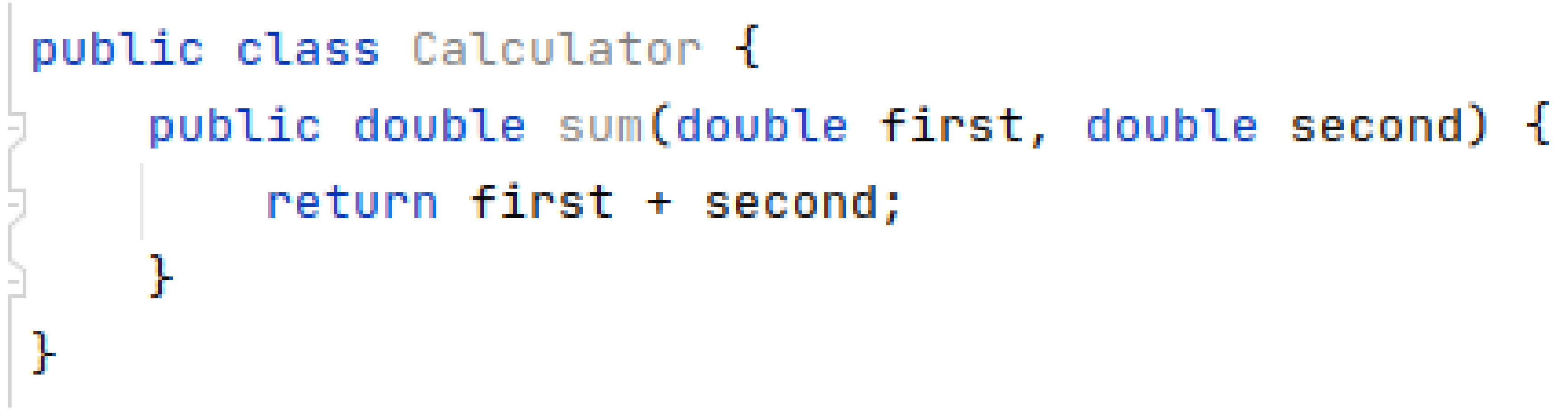


howest hogeschool

Unit testing

The anatomy of a unit test.

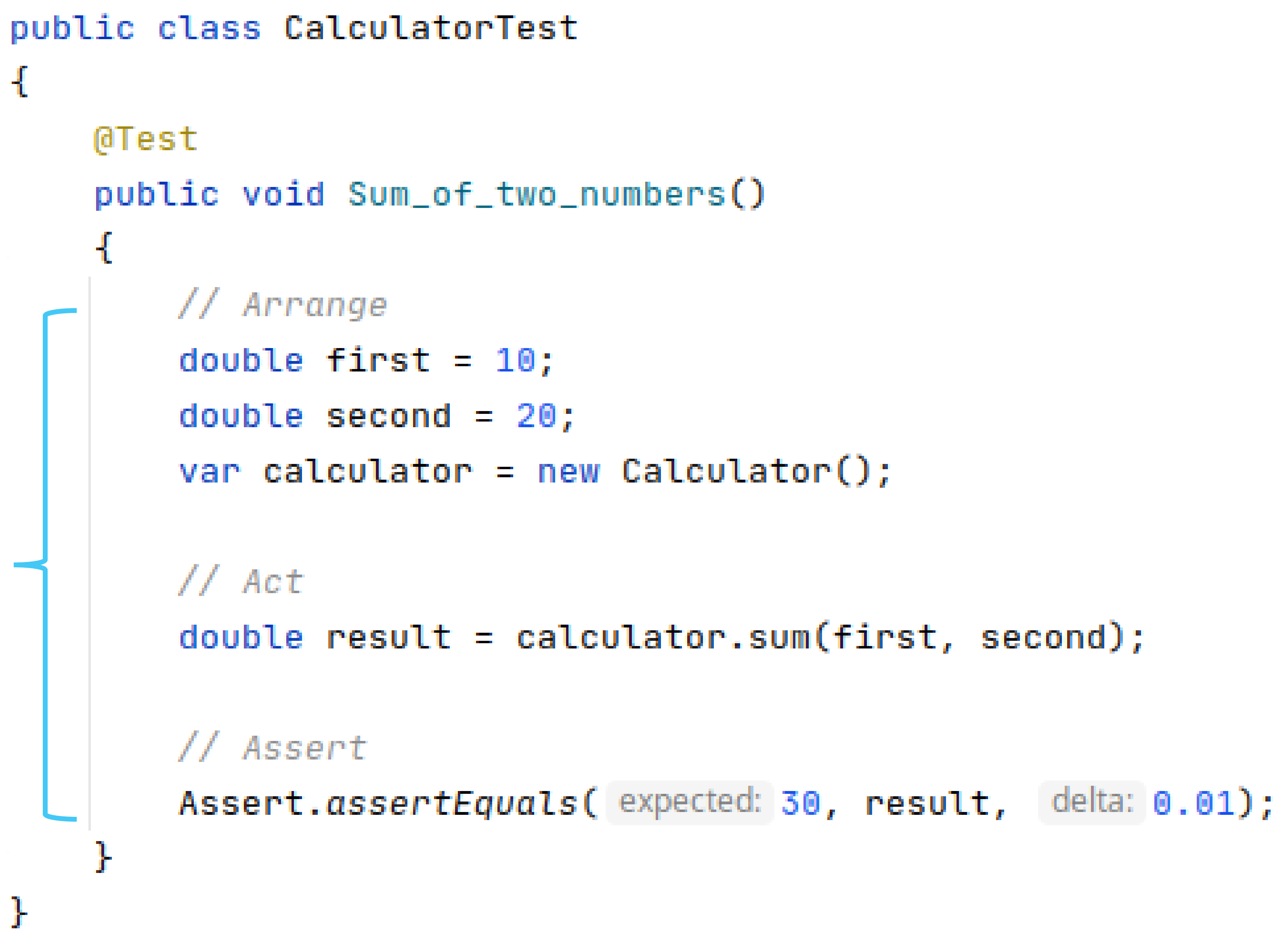
Using the AAA pattern (something called given, when, then)



```
public class Calculator {
    public double sum(double first, double second) {
        return first + second;
    }
}
```

- ***Arrange section:*** bring the system under test (SUT) and its dependencies t**o a desired state**.
- ***Act section:*** call methods on the SUT, pass the prepared dependencies, and capture the output value (if any).
- ***Assert section***, you verify the outcome. The outcome may be represented by the return value, the final state of the SUT and its collaborators, or the methods the SUT called on those collaborators.

AAA pattern



```
public class CalculatorTest
{
    @Test
    public void Sum_of_two_numbers()
    {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        Assert.assertEquals( expected: 30, result,  delta: 0.01);
    }
}
```

Avoid if statements in tests.

- A unit test should be a simple sequence of steps with no branching.
- An if statement indicates that the test verifies too many things at once. Such a test, therefore, should be split into several tests. But unlike the situation with multiple AAA sections.
- There are no benefits in branching within a test. You only gain additional maintenance costs: if statements make the tests harder to read and understand.

How large should each section be?

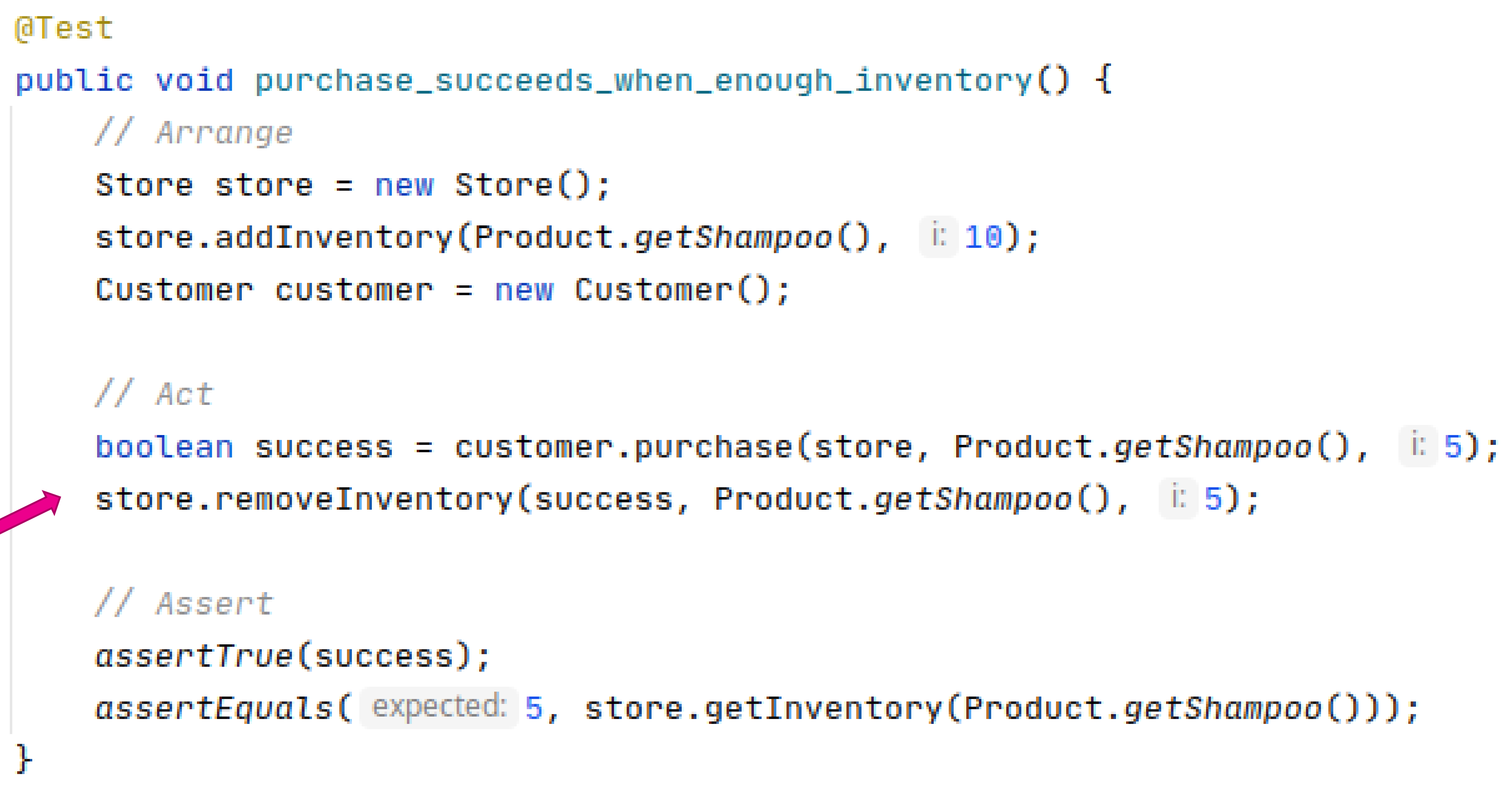
- The arrange section is the largest
    - The *arrange* section is usually the largest of the three. It can be as large as the *act* and *assert* sections combined.

- Watch out for act sections that are larger than a single line
    - The *act* section is normally just a single line of code. If the *act* consists of two or more lines, it could indicate a problem with the SUT’s public API.

Multiple act statements

If multiple acts are needed:

- Alert! Check your public api.
- In this example purchasing should also indirectly remove it from the store.
- What if the developer forgets to remove the



```java
@Test
public void purchase_succeeds_when_enough_inventory() {
    // Arrange
    Store store = new Store();
    store.addInventory(Product.getShampoo(),  i: 10);
    Customer customer = new Customer();

    // Act
    boolean success = customer.purchase(store, Product.getShampoo(),  i: 5);
    store.removeInventory(success, Product.getShampoo(),  i: 5);

    // Assert
    assertTrue(success);
    assertEquals( expected: 5, store.getInventory(Product.getShampoo()));
}
```

This test reveals that the removeInventory should be encapsulated in de purchase operation.

Reusing arrange sections

- It’s important to know how and when to reuse code between tests.

- Reusing code between *arrange* sections is a good way to shorten and simplify your tests.

- Different solution
  - Wrong solution: initialize the arrage section in the test’s constructor (or the method marked with a @BeforeAll(Each).

  - A better solution: private factory methods.

Wrong: extracting the arrange code into the test constructor

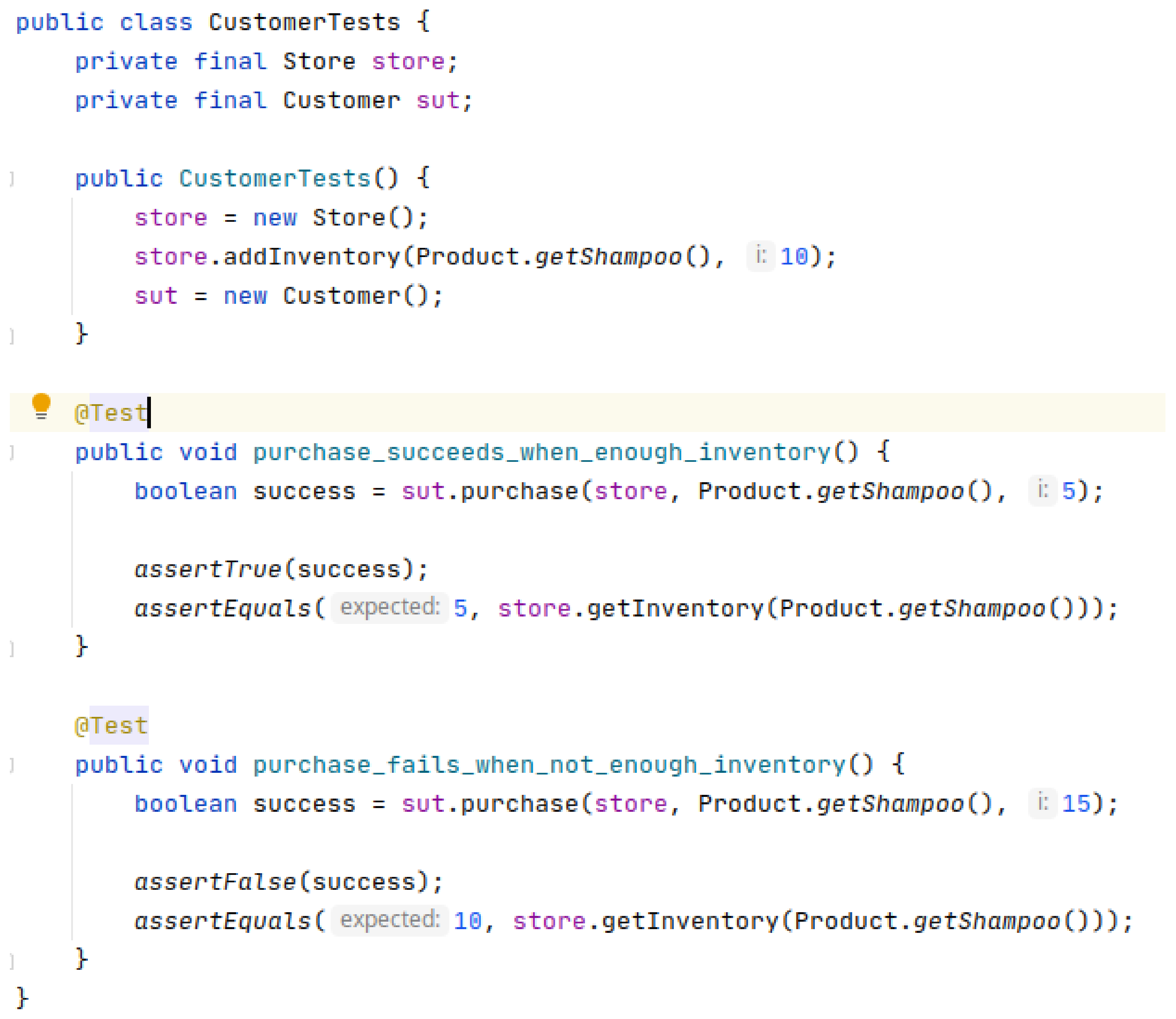
The two tests have common configuration logic.
Their *arrange* sections are the same and thus can be fully extracted into CustomerTests's constructor or @before method.

The tests themselves no longer contain arrangements.

This approach reduces the amount of test code.

This technique has two significant drawbacks:

- **It introduces high coupling between tests.**
- **It diminishes test readability.**



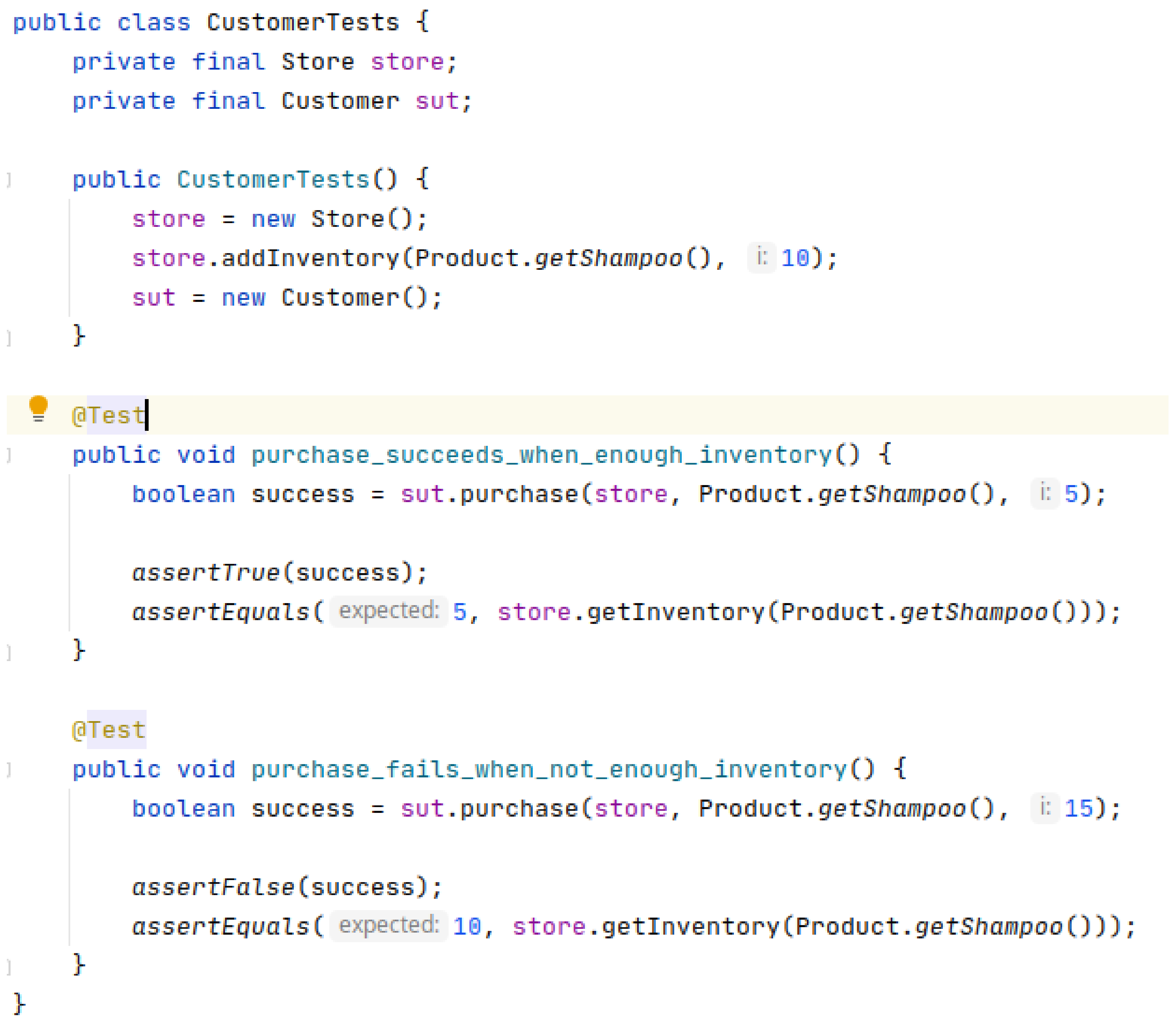
```java
public class CustomerTests {
    private final Store store;
    private final Customer sut;

    public CustomerTests() {
        store = new Store();
        store.addInventory(Product.getShampoo(),  i: 10);
        sut = new Customer();
    }

    @Test
    public void purchase_succeeds_when_enough_inventory() {
        boolean success = sut.purchase(store, Product.getShampoo(),  i: 5);

        assertTrue(success);
        assertEquals( expected: 5, store.getInventory(Product.getShampoo()));
    }

    @Test
    public void purchase_fails_when_not_enough_inventory() {
        boolean success = sut.purchase(store, Product.getShampoo(),  i: 15);

        assertFalse(success);
        assertEquals( expected: 10, store.getInventory(Product.getShampoo()));
    }
}
```

Wrong: extracting the arrange code into the test constructor

This technique has two significant drawbacks:

- **It introduces high coupling between tests.**
  - Changing 10 to 15 would break some tests...
  - The modification of one test should not affect other tests.
  - Avoid sharing shared state!

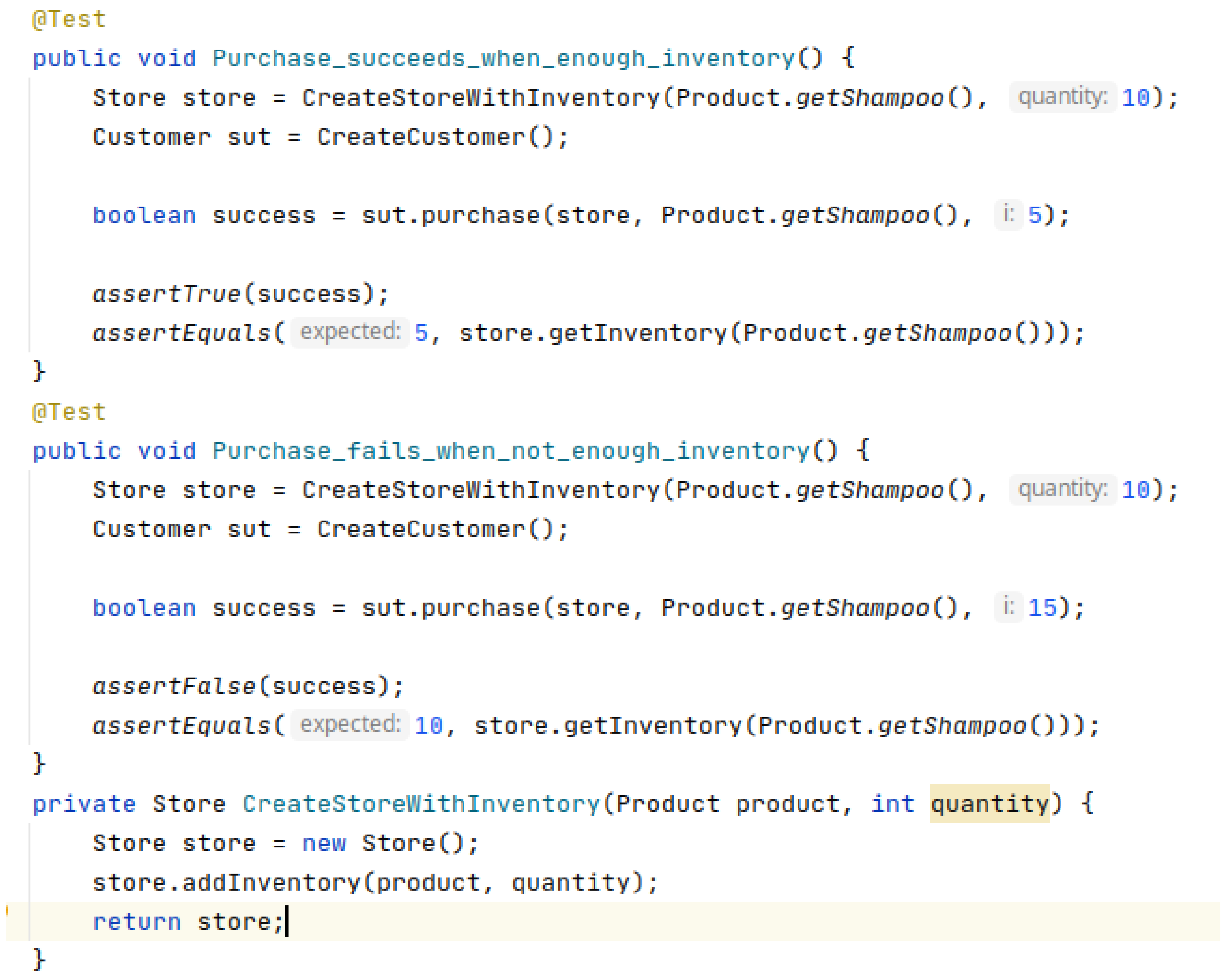
- **It diminishes test readability.**
  - No more full picture by looking at the test itself. Examing different places in the class to understand what the test method does is necessary.



```
public class CustomerTests {
    private final Store store;
    private final Customer sut;

    public CustomerTests() {
        store = new Store();
        store.addInventory(Product.getShampoo(),  i: 10);
        sut = new Customer();
    }

    @Test
    public void purchase_succeeds_when_enough_inventory() {
        boolean success = sut.purchase(store, Product.getShampoo(),  i: 5);

        assertTrue(success);
        assertEquals( expected: 5, store.getInventory(Product.getShampoo()));
    }

    @Test
    public void purchase_fails_when_not_enough_inventory() {
        boolean success = sut.purchase(store, Product.getShampoo(),  i: 15);

        assertFalse(success);
        assertEquals( expected: 10, store.getInventory(Product.getShampoo()));
    }
}
```

A better way: private factory methods.

- By extracting the common initialization code into private factory methods, you can also shorten the test code, **but at the same time keep the full context of what's going on in the tests.**
- The private methods don't couple tests to each other.
- Allow the tests to specify how they want the **fixtures** to be created.
- **Fixture:** objects or data that must be in a fixed state before the sut can be tested.



```
@Test
public void Purchase_succeeds_when_enough_inventory() {
    Store store = CreateStoreWithInventory(Product.getShampoo(),  quantity: 10);
    Customer sut = CreateCustomer();

    boolean success = sut.purchase(store, Product.getShampoo(),  i: 5);

    assertTrue(success);
    assertEquals( expected: 5, store.getInventory(Product.getShampoo()));
}
@Test
public void Purchase_fails_when_not_enough_inventory() {
    Store store = CreateStoreWithInventory(Product.getShampoo(),  quantity: 10);
    Customer sut = CreateCustomer();

    boolean success = sut.purchase(store, Product.getShampoo(),  i: 15);

    assertFalse(success);
    assertEquals( expected: 10, store.getInventory(Product.getShampoo()));
}
private Store CreateStoreWithInventory(Product product, int quantity) {
    Store store = new Store();
    store.addInventory(product, quantity);
    return store;
}
```

Unit testing

The four pillars of a good unit test.

Four pillars of a good unit test.

In the first slides we saw criteria for a good ***test suite*****.**

A good ***unit test*** has the following four attributes:

1. Protection against regressions
2. Resistance to refactoring
3. Fast feedback
4. Maintainability

For a test to be valuable each at least a little of each pillar should be available.

1) Protection against regressions

Remember: a regression == software bug

The larger the code base, the more exposure it has to potential bugs.
That’s why it’s crucial to develop a good protection against regressions.
Without such protection, it is not possible to sustain the project growth in a long run.
Without it there will be an ever-increasing number of bugs.

1) Protection against regressions: how to messure

- The amount of code that is executed during the test.
  - Large amount of business logic code that gets executed results in a higher chance to reveal a regression.
    - Assuming the test contains usefull assertions.

- The complexity of that code
  - Complex business code is more important than boilarplate code.
  - Bugs in business logic are the most damaging.

- The code's domain significance
  - Testing business logic getters setters doesn't give a big change to reveal software bugs!

2) Resistance to refactoring

The degree to which a test can sustain a refactoring of the underlying application code without failing.

Situation: *a new feature is developed, everything works great and all the tests are passing.*
As a devloper you decided to clean up the code by doing some refactoring. The feature is now more clean and still works as expected. Except one thing, the tests are failing. The problem is that the tests are written in such a way that they turn red with any modification of the underlying code. And they do that regardless of whether you actually break the functionality itself.

This is called a false positive (false alarm)

2) Resistance to refactoring: why it's so important.

Remember the goal of unit testing is to **enable sustainable project growth**.

The mechanism by which the tests enable sustainable growth is that they allow you to add new features and conduct regular refactorings without introducing regressions.

- *Tests provide an early warning when you break existing functionality.*
    - Such early warnings, allows the fix of an issue long before the faulty code is deployed to production.

- *Trustability that code changes won't lead to regressions.*
    - Without trust, there is a change developpers will hesitate to refactor (clean up) the code.

False positives intefere with benefits.

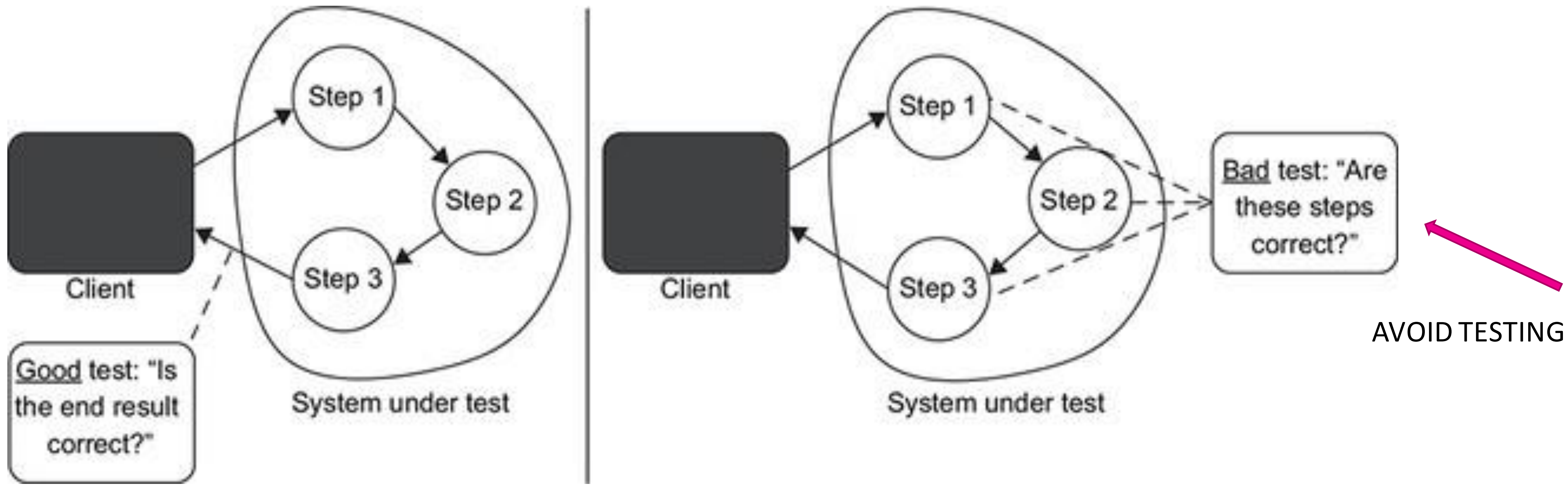
2) Resistance to refactoring: causes

- The number of false positives is directly related to the way the test is structured.

- The more the test is coupled to the implementation details of the system under test (SUT), the more false alarms it generates.

- The only way to reduce the chance of getting a false positive is to decouple the test implementation details.

- **How**?
  - Make sure the test verifies the **end result** the SUT delivers **not the steps** it takes to do that.



AVOID TESTING STEPS!

3) Fast feedback

- The faster the tests, the more tests the test suite can contain without taking too much time.

- With tests that run quickly, you can drastically shorten the feedback loop to the point where the tests begin to warn you about bugs as soon as you break the code.

- Slow tests delay the feedback and potentially prolong the period during which the bugs remain unnoticed, thus increasing the cost of fixing them.
    - Slow tests discourage frequently running the test, and therefore lead to wasting more time moving in a wrong direction.

4) Maintainability

This metric consists of two major components:

- **How hard it is to understand the test.**
  - The fewer lines of code in the test, the more readable the test is.
  - Easier to change a small test when needed.
  - Assuming you don't try to compress the test code artificially just to reduce the line count.
    - Don't cut corners when writing tests; treat the test code as a first-class citizen.

- **How hard it is to run the test.**
  - If the test works with out-of-process dependencies, time is needed to keeping those dependencies operational: reboot the database server, resolve network connectivity issues, and so on.

Unit testing

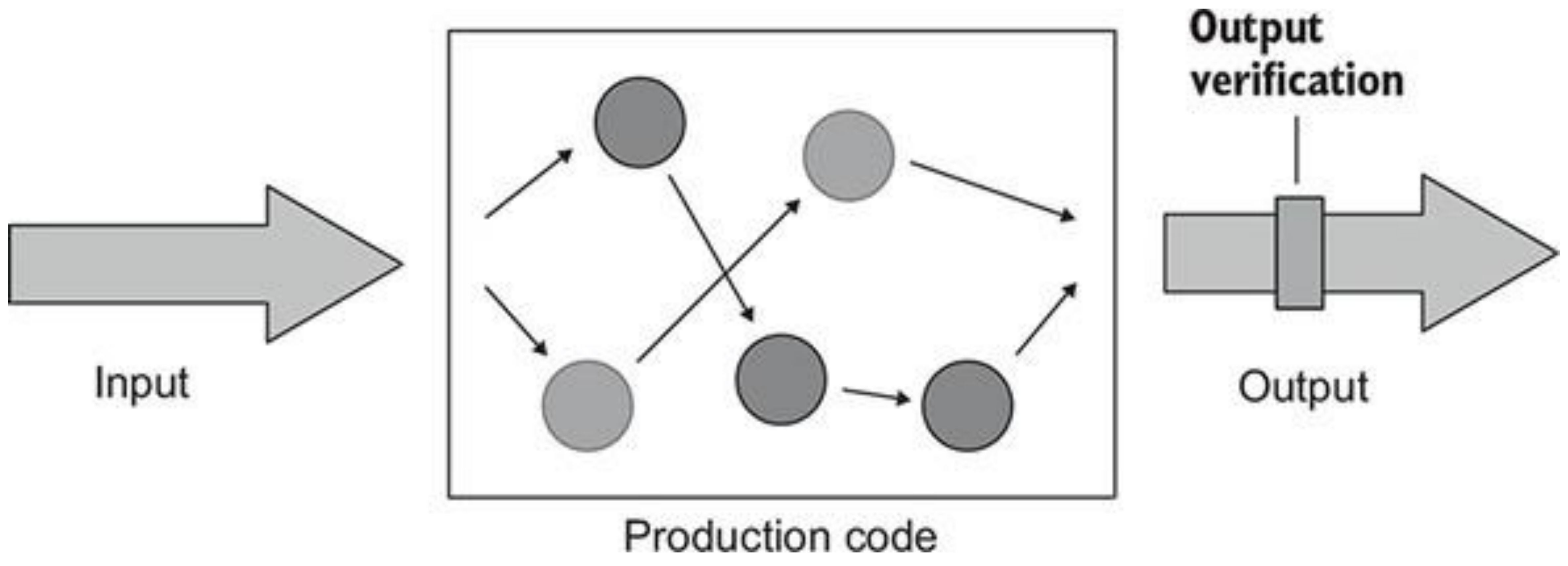
Styles of unit testing

Three styles of unit testing

- Output-based testing
- State-based testing
- Communication-based testing

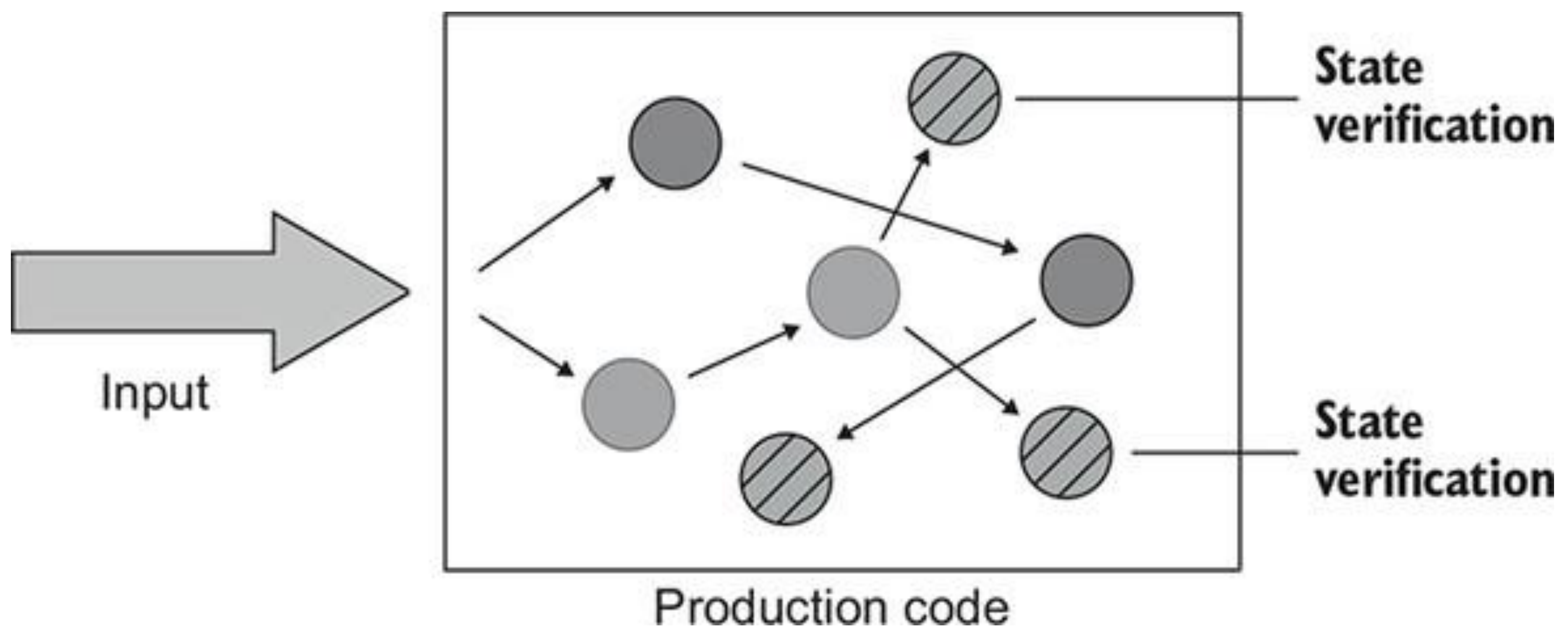
Output based testing

- Input is provided to the system under test (SUT) and the test checks the output it produces.
- Only applicable to code that doesn’t change a global or internal state.
- The only component to verify **is its return value.**



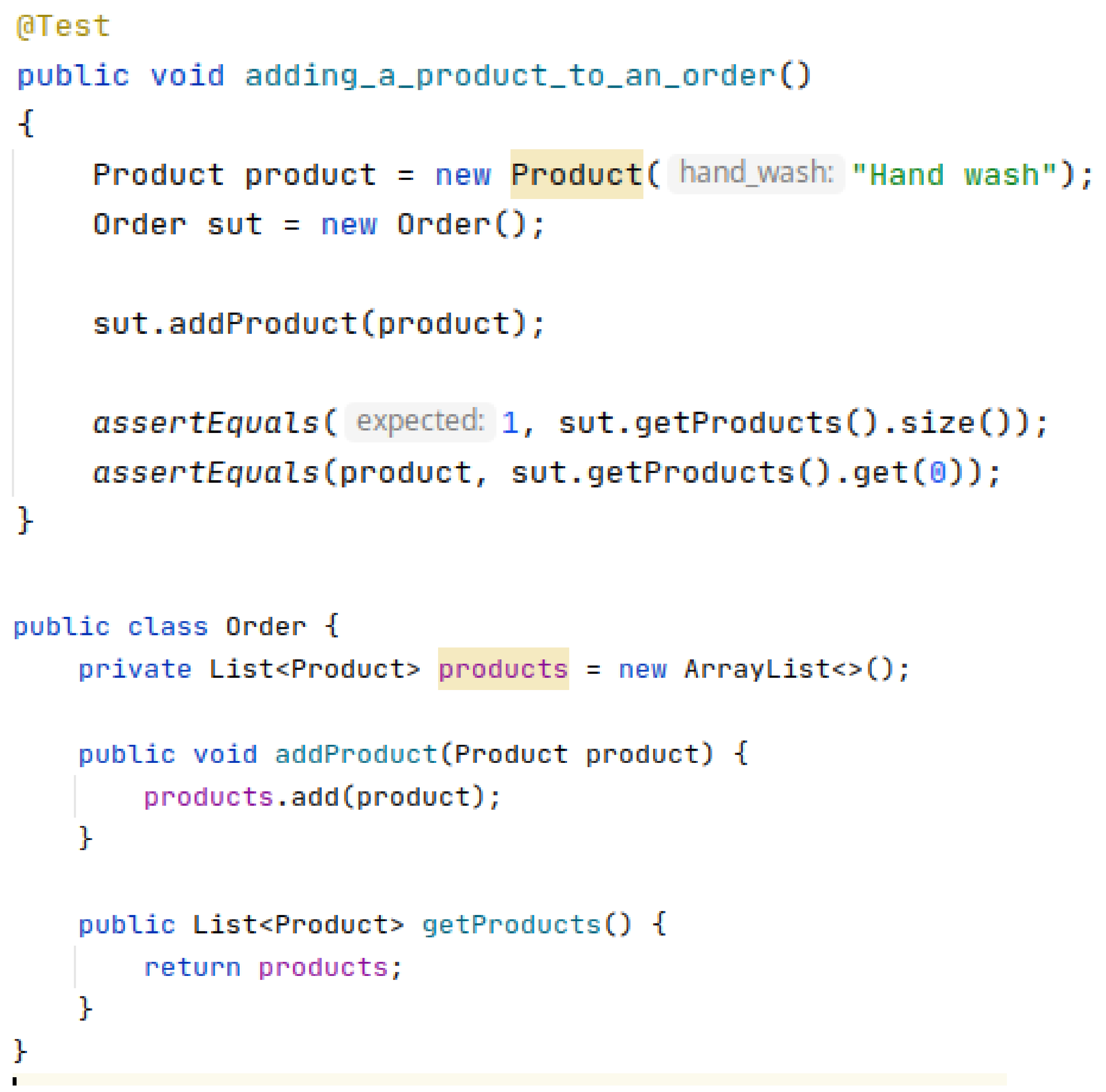
State based testing

• Verifying the state of the system after an operation is complete.
• The term *state* can refer to the state of the SUT itself, or one of its collaborators, or of an out-of-process dependency, such as the database or the filesystem.



State based testing

- The test verifies the products collection after the addition is completed.
- Unlike the example of output-based testing the outcome of addProduct() is the change made to the order's state.

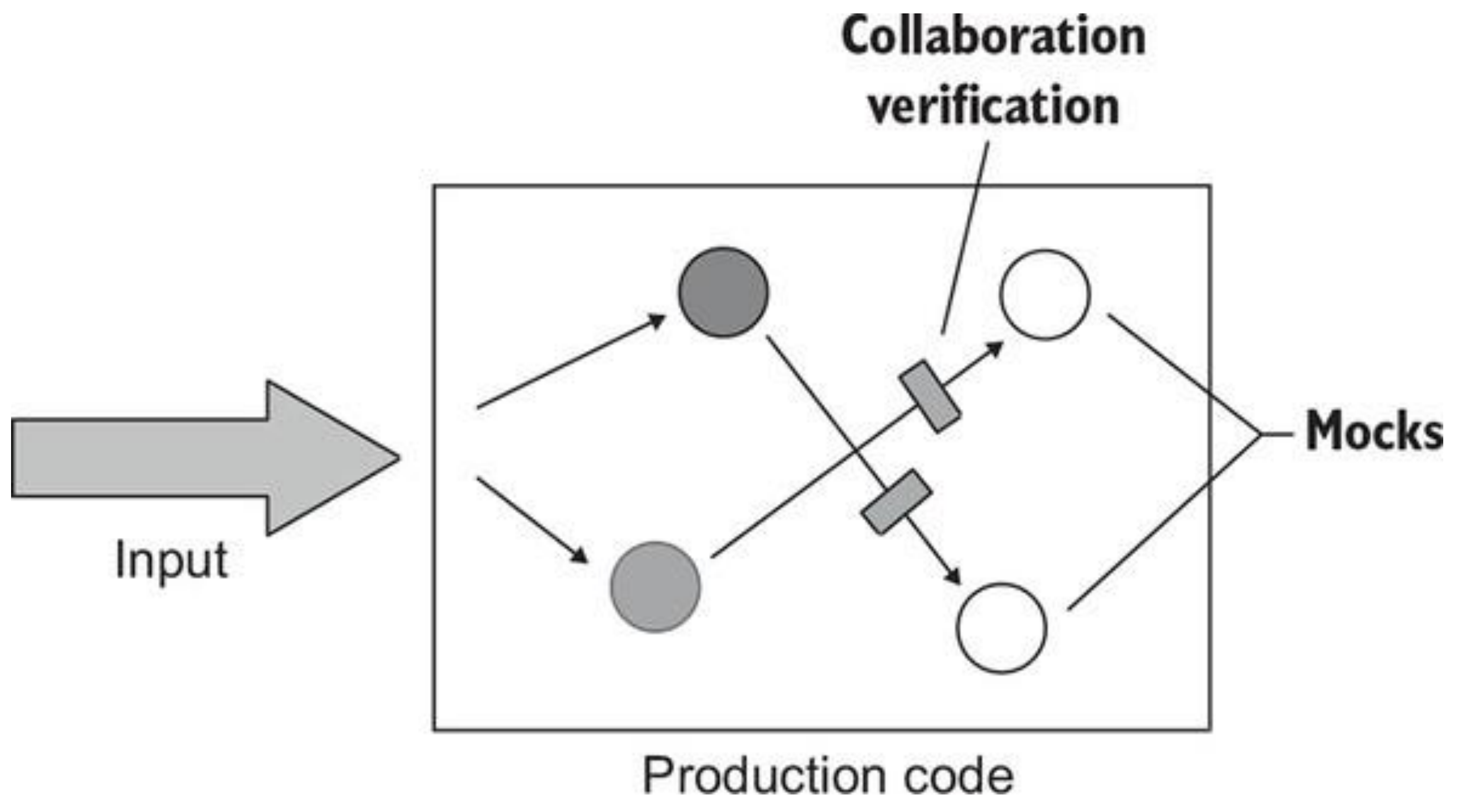


```java
@Test
public void adding_a_product_to_an_order()
{
    Product product = new Product( hand_wash: "Hand wash");
    Order sut = new Order();

    sut.addProduct(product);

    assertEquals( expected: 1, sut.getProducts().size());
    assertEquals(product, sut.getProducts().get(0));
}
```

```java
public class Order {
    private List<Product> products = new ArrayList<>();

    public void addProduct(Product product) {
        products.add(product);
    }

    public List<Product> getProducts() {
        return products;
    }
}
```

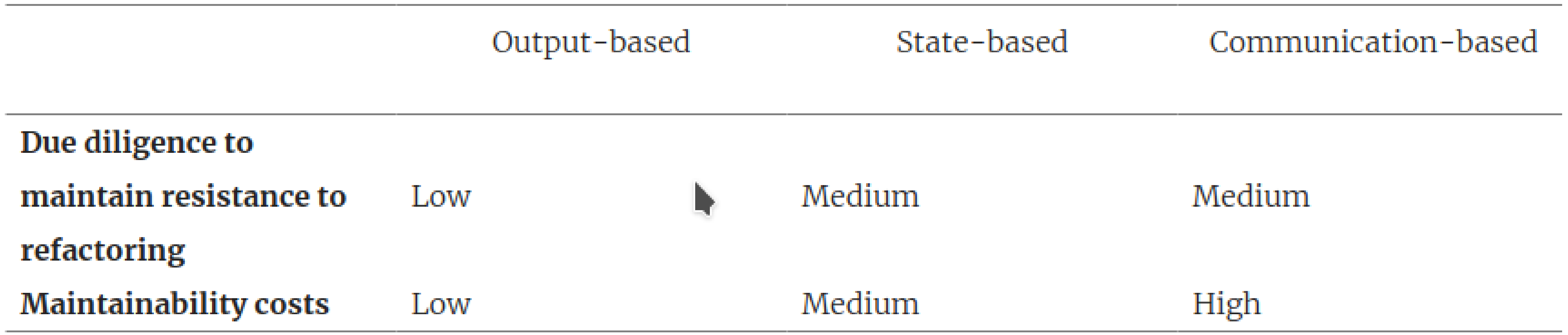
Communication based testing

This style uses mocks to verify communications between the system under test and its collaborators



Testing styles comparison

Choose output based testing if possible, then state-based …



	Output-based	State-based	Communication-based
Due diligence to maintain resistance to refactoring	Low	Medium	Medium
Maintainability costs	Low	Medium	High

Unit testing

Some anti-patterns

What is an anti-pattern?

An anti-pattern is a common solution
to a recurring problem that looks appropriate on the surface
but leads to problems further down the road.

Anti-pattern: unit testing private methods.

- Private methods and test fragility
  - Exposing private methods leads to coupling tests to implementation details and, ultimately, damaging your tests’ resistance to refactoring
  - Test these methods indirectly through public methods.

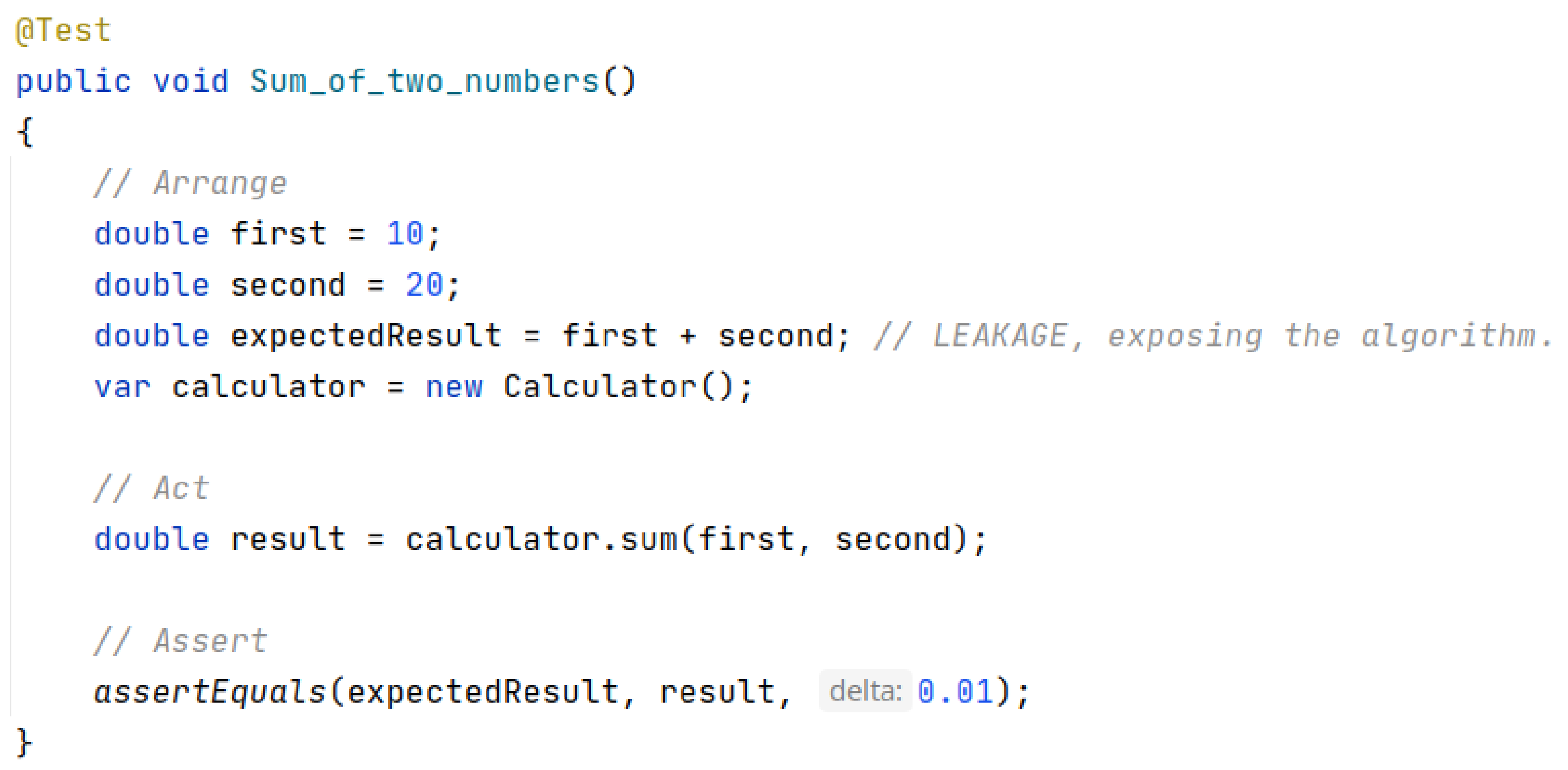
- Private method and coverage
  - Sometimes, the private method is too complex, and testing it through a public method doesn’t provide sufficient coverage.
  - Assuming the behaviour to test contains enough coverage then:
    - The uncovered code is dead? It's code that actually should be removed.
    - There is a missing abstraction?
      - Maybe the private code reveals that it should actually be a seperate class.

Anti-pattern: exposing private state.

- Exposing private state for the sole purpose of unit testing.
- Widening the public API surface for the sake of testability is a bad practice.
  - Tests should interact with the system under test (SUT) exactly
    the same way as the production code and shouldn't have any special privileges.

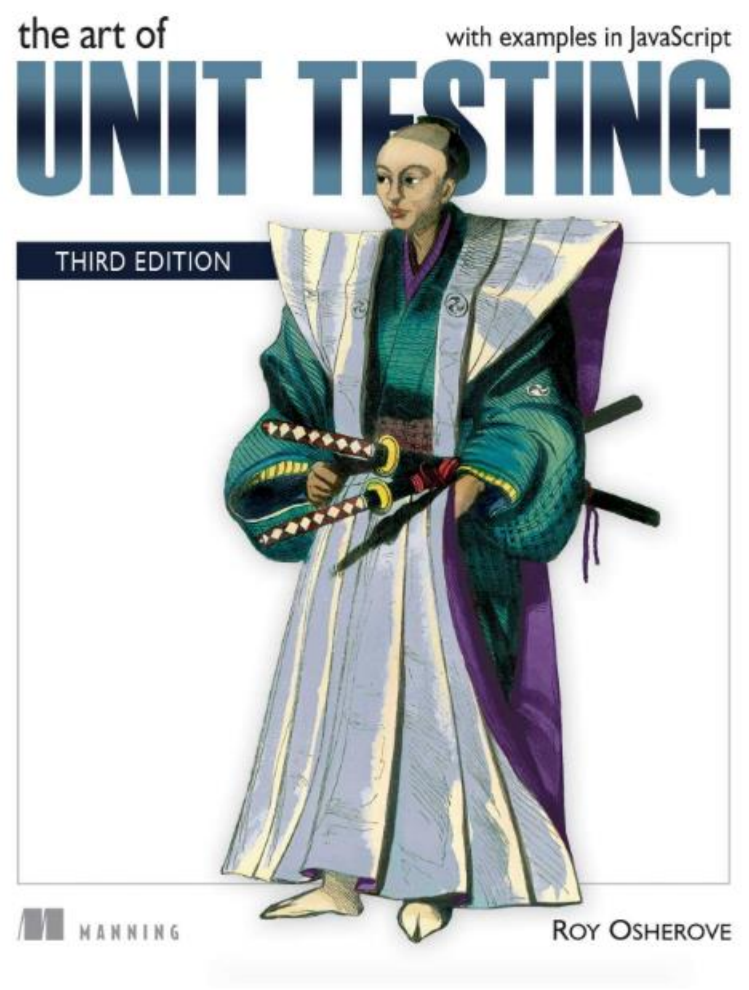
Anti-pattern: leaking domain knowledge to tests

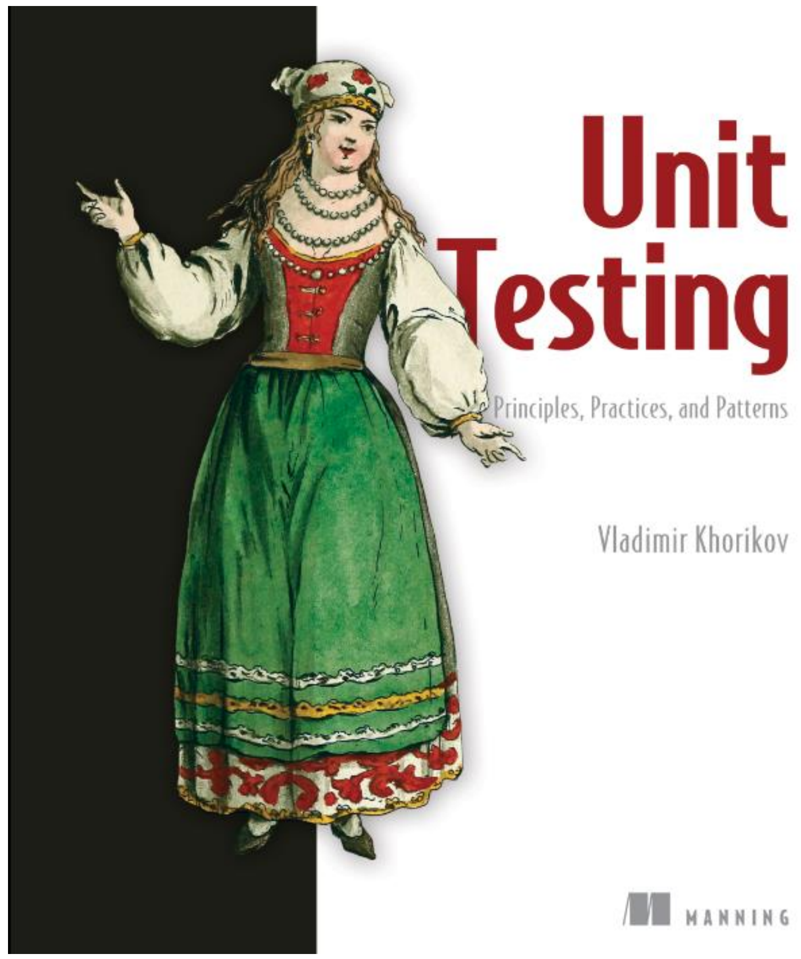
- Exposing complex algorithms of domain knowledge into a test.
  - Use hardcoded values as a result of such algorithm.

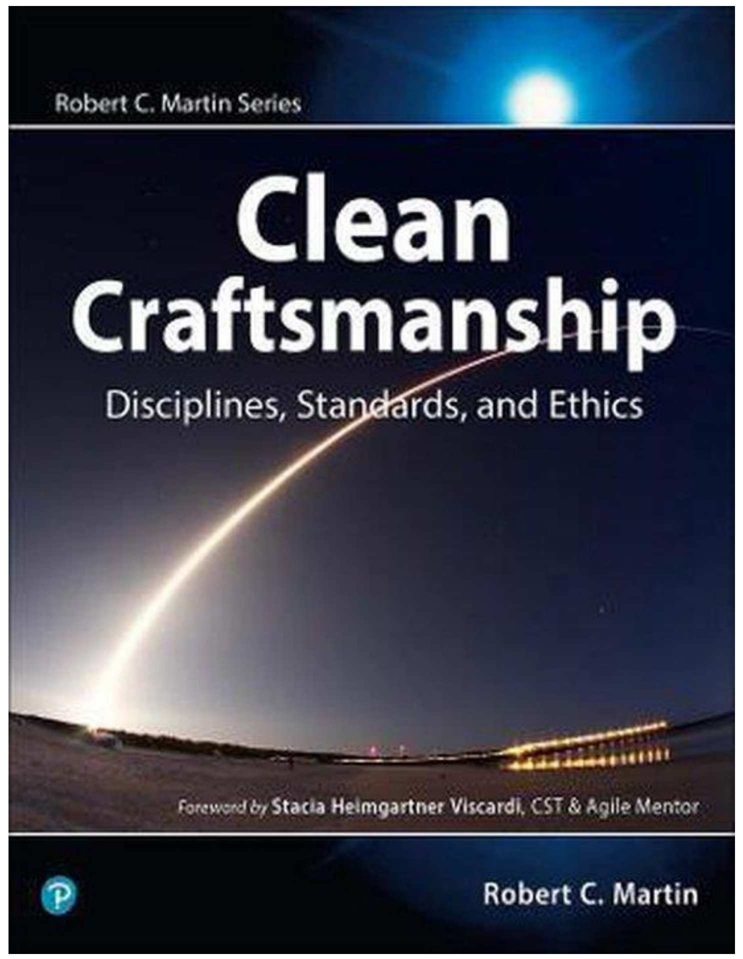


```java
@Test
public void Sum_of_two_numbers()
{
    // Arrange
    double first = 10;
    double second = 20;
    double expectedResult = first + second; // LEAKAGE, exposing the algorithm.
    var calculator = new Calculator();

    // Act
    double result = calculator.sum(first, second);

    // Assert
    assertEquals(expectedResult, result,  delta: 0.01);
}
```

Some books


the art of
with examples in JavaScript
UNIT TESTING
THIRD EDITION
MANNING
ROY OSHEROVE


Unit Testing
Principles, Practices, and Patterns
Vladimir Khorikov
MANNING


Robert C. Martin Series
Clean Craftsmanship
Disciplines, Standards, and Ethics
Foreword by Stacia Heimgartner Viscardi, CST & Agile Mentor
Robert C. Martin