

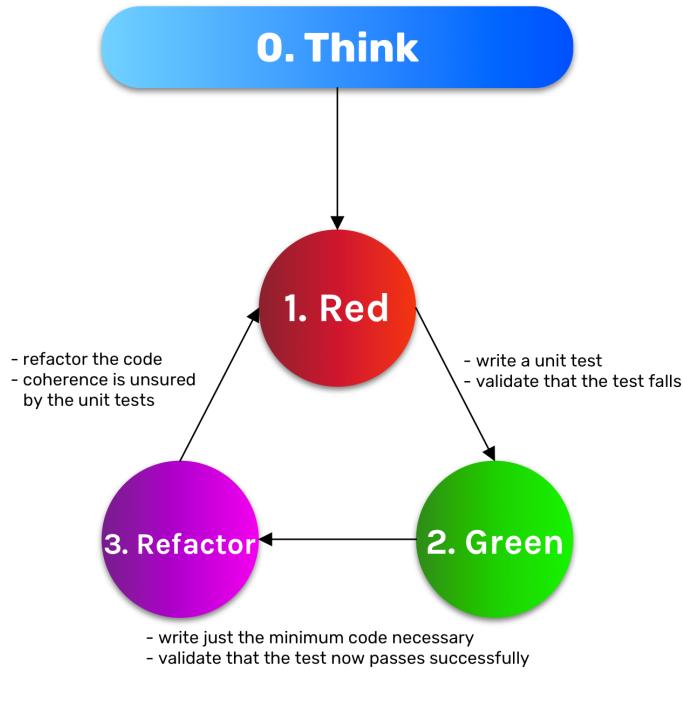


Teoria 4 - AGILE

Unit 4.2. Writing Tests in TDD

Test Driven Development

TDD is an approach that drive the design of software.



First Step: Think.

Think of a small increment that will require fewer than five lines of code and think of a test that will fail unless that behavior is present.

Second Step: Red bar.

Write the test in terms of the class' behavior and its public interface, run it and watch the new fail.

Third Step: Greenbar.

Write just the enough production code to get the test to pass. Run again and watch all tests pass.

Fourth Step: Refactor.

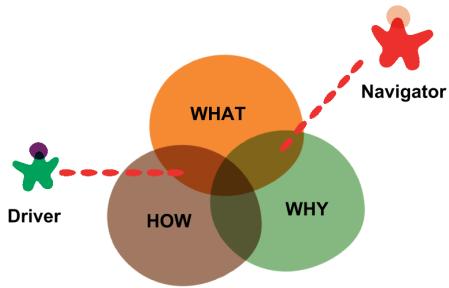
Review the code for improvements and apply small refactorings. Run again and watch all tests pass.

Before Starting

Pair Programming

- Before starting, form pairs (Pair Programming).
- In Pair Programming two people work at the same keyboard (more powerful and funny).
- It is found that pairing takes about 15% more effort than one individual working alone, but produces results more quickly and with 15% fewer defects.

- The **driver** works on the tactical challenges of creating rigorous, syntactically correct code without worrying about the big picture.



- The **navigator** thinks about the strategic issues without being distracted by the details of coding and prepares the next step, as for instance:

- What other tests do we need to write?
- How does this code fit into the rest of the system?
- Is there duplication we need to remove?
- Can the code be more clear?
- Can the overall design be better?

- Pair on everything you'll need to maintain.
- Allow pairs to form fluidly rather than assigning partners.
- Switch partners when you need a fresh perspective.
- Switch partners several times per day.

- Sit comfortable, side by side.
- Produce code through collaboration. Don't critique.
- Switch driver and navigator roles frequently.

Some TDD Principles

TDD principles to take into account before applying TDD

1. TDD Principle: **Automated Test**
 - How do you test your software: Write an automated test.
2. TDD Principle: **Test First**
 - When should you write your tests? Before you write the code that is to be tested.

Benefits of Test First

- Test first proves that your code actually works.
 - You get a low-level regression-test.
 - You can improve the design without breaking it.
 - It's more fun to code with them than without.
 - They demonstrate concrete progress.
 - Unit tests are a form of documentation.
- Test-first forces you to plan before you code.
 - Test-first reduces the cost of bugs.
 - It's even better than code inspections.
 - It systematizes the structured part of coding.
 - Unit tests make better designs.
 - It's faster than writing code without tests!

First Step: Think

First Step: Think.

Think of a small increment that will require fewer than five lines of code and think of a test that will fail unless that behavior is present.

The small increment may be a fragment of a user story or something to do from the to-do list (from previous cycles).

0. Think

1. Red

As an Account Manager I
per customer so that I can
customers are most profitable

To do:

- \$5 + 10 CHF = \$10 if CHF:USD is 2:1
- \$5 * 2 = \$10
Make amount private
Money rounding?
Dollar side-effects?

User Stories

NO DOCUMENTACIÓ

def. User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

They typically follow a simple template:

As a <type of user>, I want <some goal> so that <some reason>

User stories are often written on index cards or sticky notes, stored in a shoe box, and arranged on walls or tables to facilitate planning and discussion.

As a financial controller I need to see net profit and profit margin per customer per year for our top 10 customers, so that I can identify the characteristics of the most profitable customers

- **User stories** differ from use cases in:

- Although both deliver business value, user stories are kept smaller in scope (no more than 10 days of development work). A user story may be one single scenario of a use case.
- Use cases are more complete than user stories.
- Use cases are often permanent artifacts (documentation) whereas user stories are not intended to outlive the iteration.
- Use cases usually include user interface details, despite admonishments to avoid them.

TDD Principles

Some TDD principles to take into account in the First Step

1. TDD Principle: Test List

- What should you test? Before begin, write a list of all the tests you know you will have to write. Add to it as you find new potential tests

To do:

- $\$5 + 10 \text{ CHF} = \10 if CHF:USD is 2:1
- $\$5 * 2 = \10

2. TDD Principle: One Step Test

- Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

To do:

- $\$5 + 10 \text{ CHF} = \10 if CHF:USD is 2:1
- **$\$5 * 2 = \10**

Example

Story 1: Buy a parking ticket. A car driver walks to the pay station to buy parking time. He enters several valid coins (5, 10, and 25 cents) as payment. For each payment of 5 cents he receives 2 minutes parking time. On the pay station's display he can see how much parking time he has bought so far. Once he is satisfied with the amount of time, he presses the button marked "Buy". He receives a printed receipt, stating the number of minutes parking time he has bought. The display is cleared to prepare for another transaction.

Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

Story 3: Reject illegal coin. A driver has entered 50 cents total and the display reads "20". By mistake, he enters a 1 euro coin which is not a recognized coin. The pay station rejects the coin and the display is not updated.

- TDD Principle: **Test List** (other suggestions may be possible)

Test List:

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- read display
- buy produces valid receipt
- cancel resets pay station

- TDD Principle: **One Step Test**

Test List:

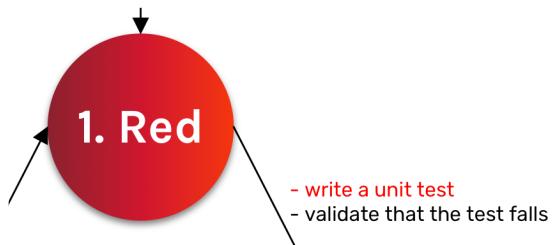
- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- read display
- buy produces valid receipt
- cancel resets pay station

Second Step: Writing Tests in TDD

Second Step: Red bar.

Write the test in terms of the class' behavior and its public interface, run it and watch the new fail.

In this step, we focus on writing a test. But, what kind of tests we must write?



Types of Tests

DevelopmentTests

Unit Tests.

A Unit Test is a method that instantiates a small portion of our application and verifies its behavior (methods) independently from other parts. Unit Tests are the relevant tests to TDD.

Integration Tests.

Integration tests demonstrate that different parts of a system work together in the real-life environment.

System Tests.

They are the biggest integration tests. They may test from the GUI to the DB.

Client / Business Analyst Tests

Acceptance Tests.

They check if functional and non-functional requirements are fulfilled. These tests may deal with the GUI.

Some examples:

- **Example 1:**

Performance Test (when performance has been defined as a non-functional requirement).

- **Example 2:**

The product X with the price 50 € has a final price of 55 € after applying the Y tax.

Functional Tests.

They are the subset of acceptance tests that check functional requirements.

Example 2 is a functional test.

TDD Tests

TDD Tests are the Tests used in the context of TDD.

On the surface, **TDD tests** are very similar to Unit Tests. We can use a unit testing framework such as JUnit to create both types of tests.

The purpose of a **Unit Test** is to give a developer confidence that a particular part of their application behaves in the way that the developer expects.

Instead, the purpose of a **TDD test** is to drive the design of an application. A TDD test is used to express what application code should do before the application code is actually written.

Unit testing in XP is often unlike classical unit testing, because in XP you're usually not testing each unit in isolation.

TDD Tests Characteristics (FIRST):

- **Fast.**

TDD tests run in memory, which makes them very fast.

- **Independent.**

A TDD test should not depend on other unit tests.

- **Repeatable.**

TDD tests doesn't change the state of the system.

They don't change the DB, send emails or create files.

- **Self-Verifying:**

A good unit TDD test fails or passes unambiguously.

When a test runs green, we have high confidence that we can ship the code to the next level. If a tests fail, we don't proceed until it gets fixed.

- **Timely.**

Tests written first specify the behavior that you're about to build into the code.

Structure of TDD Tests (Four-Phase Test pattern):

1. Definition of the state prior to the test (fixture setup in JUnit)
2. Interaction with the system under test (SUT)
3. Determine whether the expected outcome has been obtained (Assertions in JUnit).
4. Put the world back into the state in which we found it (fixture teardown in JUnit).

TDD Principles

1. Fake It

What is your first implementation once a test is broken? Return a constant. Once the tests are running, gradually transform it.

2. Assert First

When should you write the asserts?

Try writing them first.

3. Triangulation

How do you most conservatively drive abstraction with test?

Abstract only when have two or more examples.

4. Isolated Test

How should the running of tests affect one another? Not at all.

5. Obvious Implementation

How do you implement simple operations?

Just implement them.

6. Break

What do you do when you feel tired or stuck?

Take a break

7. Evident Data

How do you represent the intent of the data?

Include expected, actual results in the test, and make their relationship apparent.

You are writing tests for the reader, not just for the computer.

8. Representative Data

What data do you use for your tests?

Select a small set of data where each element represents a conceptual aspect or a special computational processing.

9. Evident Tests

How do we avoid writing defective tests?

By keeping the testing code evident, readable, and as simple as possible.

10. Do Over

What do you do when you are feeling lost?

Throw away the code and start over.

> Example

ITERATION ONE

Test List:

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- read display
- buy produces valid receipt
- cancel resets pay station

Write a test case in which the unit under test is the addPayment method of the pay station; the input value is 5 cents and the expected output is that we can read 2 minutes parking time as the value to be displayed.

- Step 1: Quickly add a test

- Apply TDD principle: **Assert First**

```
public class TestPayStation {  
    /* Entering 5 cents should make the display report 2 minutes parking time. */  
    @Test  
    public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {  
        ...  
        assertEquals( "Should display 2 min for 5 cents", 2, ps.readDisplay() ); } }
```

```
public class TestPayStation {  
    /* Entering 5 cents should make the display report 2 minutes parking time. */  
    @Test  
    public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {  
        PayStation ps = new PayStationImpl();  
        ps.addPayment( 5 );  
        assertEquals( "Should display 2 min for 5 cents", 2, ps.readDisplay() ); }}
```

The Test **fails!**

- Step 2: Run all tests and see the new one fail

- Provide a class that implements the PayStation interface (often called temporary stub) where all method bodies are empty except those that have a return value: these return 0 or null.

```
public class PayStationImpl implements PayStation {  
    public void addPayment( int coinValue ) throws IllegalCoinException { }  
    public int readDisplay() { return 0; }  
    public Receipt buy() { return null; }  
    public void cancel() { } }
```

```
public class TestPayStation {  
    @Test  
    public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {  
        PayStation ps = new PayStationImpl();  
        ps.addPayment( 5 );  
        assertEquals( "Should display 2 min for 5 cents", 2, ps.readDisplay() ); }}
```

The Test **fails!**

- Step 3: Make a little change
 - Apply the TDD principle: **Fake it!**

```
public class PayStationImpl implements PayStation {
    public void addPayment( int coinValue ) throws IllegalCoinException { }
    public int readDisplay() { return 2; } ←
    public Receipt buy() { return null; }
    public void cancel() { } }
```

The Test **passes!**

- We have actually tested quite a few of our initial test cases.
- Our test case passes but the production code is incomplete! It will only pass this single test case! This is the correct and smallest possible implementation of this single test case.
- Apply the TDD principle: **Triangulation** in the next iteration (two or more examples before generalization).
- Status at the end of Iteration 1
 - Test List:
 - accept-legal-coin
 - 5-cents-should-give-2-minutes-parking-time
 - reject-illegal-coin
 - read-display
 - buy-produces-valid-receipt
 - cancel-resets-pay-station
 - 25 cents = 10 min
- Preparing Iteration 2
 - Applying the TDD principle: **One Step Test**
 - Test List:
 - accept-legal-coin
 - 5-cents-should-give-2-minutes-parking-time
 - reject-illegal-coin
 - read-display
 - buy-produces-valid-receipt
 - cancel-resets-pay-station
 - 25 cents = 10 min ←

ITERATION TWO

- Step 1: Quickly add a test
 - Apply the TDD principle: **Triangulation**

```
public class TestPayStation {
    @Test
    public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
        PayStation ps = new PayStationImpl()
        ps.addPayment( 5 );
        assertEquals( "Should display 2 min for 5 cents", 2, ps.readDisplay() );
        ps.addPayment( 25 );
        assertEquals( "Should display 12 min for 30 cents", 12, ps.readDisplay() ); } }
```

- **Bad Test!!!**

- Avoid interference between two test cases (5 cents for 2 min and 25 cents for 12 min).
- Apply TDD principle: **Isolated Test**

```
public class TestPayStation {
    @Test
    public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
        PayStation ps = new PayStationImpl();
        ps.addPayment( 5 );
        assertEquals( "Should display 2 min for 5 cents", 2, ps.readDisplay() ); }

    @Test
    public void shouldDisplay10MinFor25Cents() throws IllegalCoinException {
        PayStation ps = new PayStationImpl();
        ps.addPayment( 25 );
        assertEquals( "Should display 10 min for 25 cents", 10, ps.readDisplay() ); } }
```

- Step 2: Run all tests and see the new one fail

```
public class PayStationImpl implements PayStation {
    public void addPayment( int coinValue ) throws IllegalCoinException { }
    public int readDisplay() { return 2; }
    public Receipt buy() { return null; }
    public void cancel() {} }
```

```
public class TestPayStation {
    ...
    @Test
    public void shouldDisplay10MinFor25Cents() throws IllegalCoinException {
        PayStation ps = new PayStationImpl();
        ps.addPayment( 25 );
        assertEquals( "Should display 10 min for 25 cents", 10, ps.readDisplay() ); } }
```

Test **fails!** `shouldDisplay10MinFor25Cents()` should display 10 min for 25 cents but it displays 2 min.

- Step 3: Make a little change

- Generalizing

```
public class PayStationImpl implements PayStation {
    private int insertedSoFar;

    public void addPayment( int coinValue ) throws IllegalCoinException {
        insertedSoFar += coinValue; }

    public int readDisplay() { return insertedSoFar / 5 * 2; }

    public Receipt buy() { return null; }

    public void cancel() { } }
```

- Step 4: Run all tests and see them all succeed

Test **passes!**

- Step 5: Refactor to remove the duplication

```
public class TestPayStation {
    PayStation ps;
    @Before
    public void setUp() { ps = new PayStationImpl(); }
    @Test
    public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
        ps.addPayment( 5 );
        assertEquals( "Should display 2 min for 5 cents", 2, ps.readDisplay() );
    }
    @Test
    public void shouldDisplay10MinFor25Cents() throws IllegalCoinException {
        ps.addPayment( 25 );
        assertEquals( "Should display 10 min for 25 cents", 10, ps.readDisplay() );
    }
}
```

- The last test case contains the assertion `assertEquals(10, ps.readDisplay())`.
 - Where did '10' come from?
- Apply TDD principle: **Evident Data**

```
assertEquals( "Should display 10 min for 25 cents", 25/5*2,
    ps.readDisplay() ); }
```

- Status at the end of Iteration 2

Test List:

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- read display
- buy produces valid receipt
- cancel resets pay station
- 25 cents = 10 min
- enter two or more legal coins

- Preparing Iteration 3

Test List:

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin ←
- read display
- buy produces valid receipt
- cancel resets pay station
- 25 cents = 10 min
- enter two or more legal coins

ITERATION 3

- Step 1: Quickly add a test

```
public class TestPayStation {
    ...
    @Test(expected=IllegalCoinException.class)
    public void shouldRejectIllegalCoin() throws IllegalCoinException {
        ps.addPayment(17);
    }
}
```

- Step 2: Run all tests and see the new one fail

Of course, the Test **fails!**

- Step 3: Make a little change

```
public class PayStationImpl implements PayStation {
    private int insertedSoFar;

    public void addPayment( int coinValue ) throws IllegalCoinException {
        switch ( coinValue ) {
            case 5: break;
            case 25: break;
            default:
                throw new IllegalCoinException("Invalid coin: "+coinValue);
        }
        insertedSoFar = coinValue;
    }

    public int readDisplay() { return insertedSoFar / 5 * 2; }

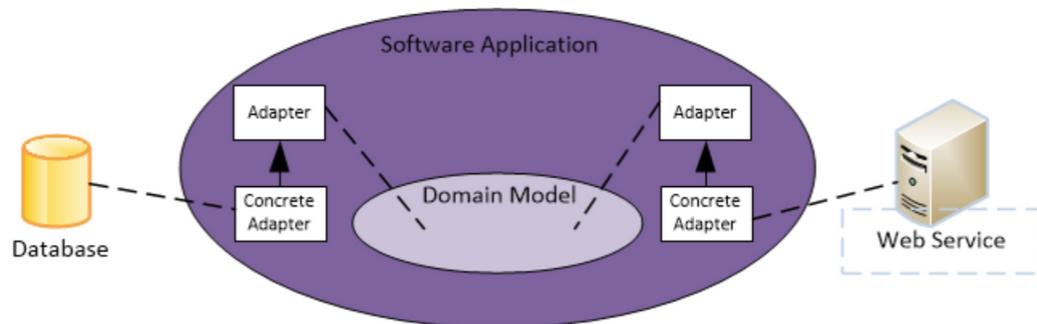
    public Receipt buy() { return null; }

    public void cancel() { }
}
```

- Step 4: Run all tests and see them all succeed
 - All Tests pass!
 - But ... the production code is still incomplete. I do not test for coin value 10 and therefore it is considered an illegal coin.
 - Apply TDD principle: **Representative Data** in the next iteration.

Writing Tests for External Dependencies

- TDD tests (as unit tests) aren't enough. At some point, our code has to talk to the outside world (databases, web services, etc...). Tests that do these things are **Integration Tests**.
- We shouldn't need many integration tests. The number of this type of tests should be proportional to the number of external dependencies. If we need more, it is a symptom of a bad design.
- Instead, we may use TDD tests to drive the design of the interaction between our code and the external dependencies.
- TDD tests drive the design of the interaction with the adapters (using mocks or stubs).
- TDD tests drive the design of the interaction between the concrete adapters and the external dependency (using, for instance, a in-memory database).
- Integration tests validate the real interaction



- Sometimes the classes we want to design have complex behaviors, extensive state, and tight relationships. This makes the definition of tests difficult and time consuming.

- There are four types of objects that can be used in place of a real object for testing purposes:
 - **Dummy:** Object that has no implementation as an argument of a method called on the SUT.
 - **Fake:** Object with a lighter-weight implementation that a SUT depends on.
 - **Stubs:** A test-specific object that feeds the desired indirect inputs into the SUT.
 - **Mocks:** A test-specific object that a SUT depends on that verifies it is being used correctly by the SUT.
- **Dummy Object.**

We test the Invoice that requires a Customer, Address and City objects. But the behavior we test should not access the Customer at all. So we create a Customer dummy object.

```
public void testInvoice_addLineItem_noECS() {
    final int QUANTITY = 1;
    Product product = new Product(getUniqueNumberAsString(),
        getUniqueNumber());
    State state = new State("West Dakota", "WD");
    City city = new City("Centreville", state);
    Address address = new Address("123 Blake St.", city, "12345");
    Customer customer = new Customer(getUniqueNumberAsString(),
        getUniqueNumberAsString(),
        address);
    Invoice inv = new Invoice(customer);
    // Exercise
    inv.addItemQuantity(product, QUANTITY);
    // Verify
    List<LineItem> lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    assertLineItemsEqual("", expItem, actual);
}

public void testInvoice_addLineItem_DOO {
    final int QUANTITY = 1;
    Product product = new Product("Dummy Product Name",
        getUniqueNumber());
    Invoice inv = new Invoice(new DummyCustomer());
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Exercise
    inv.addItemQuantity(product, QUANTITY);
    // Verify
    List<LineItem> lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    assertLineItemsEqual("", expItem, actual);
}

public class DummyCustomer implements ICustomer {
    public DummyCustomer() {
        // Real simple; nothing to initialize!
    }

    public int getZone() {
        throw new RuntimeException("This should never be called!");
    }
}
```

- **Fake Object.**
 - We test the `CalculateTotalSalary` method with a collection of employees that were read from a database. However in unit tests you don't want to access a database. So you create a fake employees list.

```
public int CalculateTotalSalary(IList<Employee> employees) { }

IList<Employee> fakeEmployees = new List<Employee>();
```

- **Stub Object.** Test to send an email message if we failed to fill an order. We can't send emails to customers during testing. So we create a stub of the email system.

```
public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent()); }
```

```
public interface MailService {
    public void send (Message msg); }

public class MailServiceStub implements MailService {
    private List<Message> messages = new ArrayList<Message>();
    public void send (Message msg) {
        messages.add(msg); }
    public int numberSent() {
        return messages.size(); } }
```

- **Mock Object.** Test to send an email message if we failed to fill an order. We can't send emails to customers during testing. So we create a mock to test the behavior of the mail service system (using jMock)

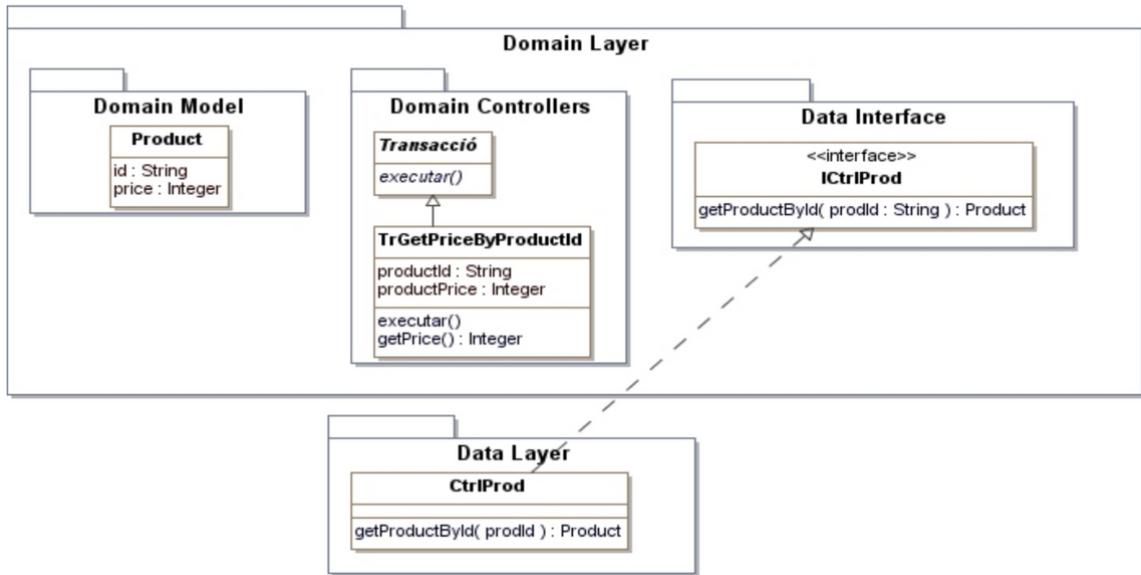
```
public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory").withAnyArguments().
        will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy()); }
```

> Example

- Write test cases to design the interaction between a software system (specifically, a transaction `TrPriceByProductId`) and a database that provides the data to this transaction.



- Write a test case in which the SUT is the `execute()` method of the `TrPriceByProductId` class.
- This method has to access to the data layer (`CtrlProduct` adapter) to get the price of a product by the product id.
- We use a mock to represent the real `CtrlProduct` class.
- Steps to define the test:
 - Creation of a mock object for the `CtrlProduct`.
 - Creation of an object of the `TrPriceByProductId` class.
 - Injection of the mock object into the `TrPriceByProductId` object.
 - Definition of the expectations for the mock object (invocation to the mock object to get the data).
- Execution of the method (SUT).
- Verification of the expectations and the state.

- Test case for `TrPriceByProductId::execute()` using **Mockito**

```

10 public class TestTrPriceByProductId {
11
12     //Creation of a mock object for the CtrlProduct
13@     @Mock
14     ICtrlProd mockCP;
15
16@     @Before
17     public void setup() {
18         MockitoAnnotations.initMocks(this);
19     }
20@     @Test
21     public void testTVPriceIs200Euros() {
22         //Creation of an object of the TrPriceByProductId class
23         TrPriceByProductId tr = new TrPriceByProductId("TV");
24         //Injection of the mock object into the TrPriceByProductId object
25         tr.setCp(mockCP);
26
27         //expectations
28         when(mockCP.getProductById("TV")).thenReturn(new Product("TV", 200));
29
30         //exercise
31         tr.execute();
32
33         //verify
34         verify(mockCP).getProductById("TV");
35         assertEquals("TV price is 200 euros", 200, tr.getPrice());
    }

```

- `TrPriceByProductId::execute()` method (**mock injection**)

```

1 package com.junitmockito.example;
2
3 public class TrPriceByProductId {
4     private ICtrlProd cp;
5
6     private String productId;
7     private int productPrice;
8
9
10@    public TrPriceByProductId(String id) {
11        productId = id;
12    }
13
14@    public void execute() {
15        Product p = cp.getProductById(productId);
16        productPrice = p.getProductPrice();
17    }
18@    public int getPrice() {
19        return productPrice;
20    }
21
22@    public ICtrlProd getCp() {
23        return cp;
24    }
25
26@    public void setCp(ICtrlProd cp) {
27        this.cp = cp;
28    }
29
30 }

```

- Write a test case in which the unit under test is `CtrlProd::getProductById()` method.

- This test drive the design of the interaction between the concrete adapters (CtrlProd) and the database external dependency (using, for instance, a in-memory array).
- We use a fake object to represent the CtrlProduct class.
- Steps to define the test:
 - Creation of a fake object for the CtrlProduct.
 - Execution of the method getProductById.
 - Verification of the state returned.
- Test case for `CtrlProd::getProductById()`

```

1 package com.junitmockito.example2;
2
3 import static org.junit.Assert.*;
4
5 public class TestGetProductById {
6
7     private CtrlProd cp;
8
9     @Before
10    public void setup() {
11        cp = new CtrlProd();
12    }
13
14    @Test
15    public void testGetProductById() {
16        Product p = cp.getProductById("DVD");
17        assertEquals("DVD price is 100 euros", 100, p.getProductPrice());
18    }
19 }
```

- Fake object CtrlProd

```
1 package com.junitmockito.example2;
2
3 import com.junitmockito.example.Product;
4
5 public class CtrlProd implements ICtrlProd {
6
7     private Product[] inMemoryDB = new Product[3];
8
9     public CtrlProd(){
10         Product p1 = new Product ("TV", 200);
11         Product p2 = new Product ("DVD", 100);
12         Product p3 = new Product ("PC", 800);
13         inMemoryDB[0] = p1;
14         inMemoryDB[1] = p2;
15         inMemoryDB[2] = p3;
16     }
17
18     @Override
19     public Product getProductById(String prodId) {
20         Boolean found = false;
21         int i = 0;
22         int j = 0;
23
24         while ((i < inMemoryDB.length) && (!found)) {
25             if (inMemoryDB[i].getProductId().equals(prodId)) {
26                 j = i;
27                 found = true;
28             }
29             i++;
30         }
31         return inMemoryDB[j];
32     }
33 }
```