Exercises: Unit Testing

This document defines the exercises for "Java OOP Advanced" course @ Software University.

Problem 1. Database

Create a simple class - **Database**. It should **store Integers**. You should set the initial integers by constructor. Store them **in array**. Your Database should have a functionality to **add**, **remove** and **fetch all stored items**. Your task is to **test the class**. In other words, create the class and **write tests**, so you are sure its methods are working as intended.

Constraints

- Storing array's **capacity** must be **exactly 16 integers**.
  - If the size of the array is not 16 integers long, throw **OperationNotSupportedException**.
- **Add** operation, should **add an element at the next free cell.** (just like a stack)
  - If passed element is null throw **OperationNotSupportedException**.
- **Remove** operation, should support only removing an element **at the last index**. (just like a stack)
  - If you try to remove element from empty Database throw **OperationNotSupportedException**
- **Constructors** should take integers only, and store them in **array.**
- **Fetch method** should return the elements as **array.**

Hint

Do not forget to **test the constructor(s)**. They are methods too!

Problem 2. Extended Database

You already have a class - Database. Now your task is to extend it. It should support, adding, removing and finding People. In other words, it should store People. There should be two types of finding methods - first: findById (long id) and the second one: findByUsername (String username). As you may guess, each person should have its own unique id, and unique username. Your task is to implement these functions and test them.

Constraints

Database should have methods:

- add
  - If there are multiple users with this id, throw **OperationNotSupportedException.**
  - If negative nor null ids are present, throw **OperationNotSupportedException**.
- remove
- findByUsername
  - If no user is present by this username, throw **OperationNotSupportedException**.
  - If username parameter is null, throw **OperationNotSupportedException.**
  - Arguments are all CaseSensitive!
- findById
  - If no user is present by this id, throw **OperationNotSupportedException**.

Hint

Do not forget to test the constructor(s). They are methods too!

Problem 3. Iterator Test

Create a class "ListIterator", it should receive the collection (Strings) which it will iterate, through its constructor. You should store the elements in a List and get them initially through its constructor. If there is null passed to the constructor, throw new **OperationNotSupportedException**. The class should have three main functions:

- **Move** - should move an internal index position to the next index in the list, the method should return true if it successfully moved and false if there is no next index.
- **HasNext** - should return true if there is a next index and false if the index is already at the last element of the list.
- **Print** - should print the element at the current internal index, calling Print on a collection without elements should throw an appropriate exception with the message "**Invalid Operation!**".

By default, the internal index should be pointing to the **0th index** of the List. Your program should support the following commands:

Command	Return Type	Description
Move	boolean	This command should move the internal index to the next index.
Print	void	This command should return the element at the current internal index.
HasNext	boolean	Returns whether the collection has a next element.

Test

Create tests, so you are sure all methods in the class ListIterator are working as intended.

Constraints

- There will always be only **1 Create** command and it will always be the first command passed.
- The last command will always be the only **END** command.

Examples

Input	Output
Create Print END	Invalid Operation!
Create Stefcho Goshky HasNext Print Move Print END	true Stefcho true Goshky

Create 1 2 3 HasNext Move HasNext HasNext Move HasNext END	true true true true true false

Problem 4. **Bubble Sort Test

There is a sorting algorithm - Bubble Sort. You could read this article, to get better the idea: Bubble Sort.

Bubble sort, sometimes referred to as **sinking sort**, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

Create a class "Bubble", and try to implement the sorting method yourslef. Create a Test class and test, is it working as intended. Think about the border cases.

Problem 5. **Integration Tests

When you create tests for a program, you should decouple a method which you test, so you could test a single problem. If a method is coupled to another one or it is doing more job that it is supposed to do, you could not clearly tell, which part is buggy. So when you create test, to determine which exactly is the problem, you separate each operation of the method and test each of them, to be sure, everything is working correctly. But there is another instance of the Unit Testing - Integration testing. When you do integration testing, you are testing the relationships of the models and the functions.

You are a web developer and your boss assigns you a task to implement the backend logic of the following problem.

You have a site with the following two models:

Category has a:

- Name.
- Set of Users.
- (optional) Set of Categories (children)

User has a:

- Name.
- Set of Categories

Functions to implement

- Add Categories
- Remove Category / Categories

- Assign a child Category to a single Category.
- Assign a User to specific Category.

Task

Create tests for all the classes. Test, do they work correctly together. Keep in mind, you should test the border cases which are very interesting in this domain.

Hint

In case you have a child Category and you remove its parent Category, if it has any Users, they must be reassigned to the child Category. If you try to move a child Category from its parent to another, don't forget to remove the relationship from the first one. (e.g. When you move a Category from one parent to another, don't copy the child to the both parents. Instead, create new child with all characteristics of the previous one and assign it to the new parent)

Problem 6. Twitter

You are working for twitter and your boss wants you to make a twitter client for microwave ovens. The main models in your domain should be - a Tweet (e.g. Message) and a Client (e.g. MicrowaveOven). Implement an Interface - Tweet. This interface should have the only functionality - to retrieve a message of a tweet. Create its implementation. The next interface we should create is Client (e.g. MicrowaveOven). When the Client receives the Tweet, it should write it to the console first and send it to the server as next step. Your task is to model the domain and create unit test.

Hint

Don't forget to check the number of invocations of the methods to ensure they are invoked properly.

Problem 7. Hack

It is not considered as good practice to mock foreign objects. For the sake of learning, your task is to ensure the given methods are working as expected:

Methods to Test:

- `Math.Abs()`
- `System.lineSeparator()`
- `Math.Floor()`

Try to mock the classes containing these methods and create tests which are proving they are working correctly.

Problem 8. Custom Linked List

You are given a data structure that needs to be tested. Use the Java file **CustomLinkedList.java** and:

- Create Test Class and Test Methods for **all public members** that need testing.
- Create tests that ensure all methods, getters and setters **work correctly**.
- Use the `@Test(expected = ExpectedException.class)` annotation for methods that are expected to throw exception in case wrong input is entered (those tests don't need assert messages).
- Give **meaningful assert messages** for failed tests.

Problem 9. ZonedDateTime.now().plusDays()

Using Java 8 test the method **`ZonedDateTime.now().plusDays().`** Use mocking to provide the **ZonedDateTime** objects you need.

Some cases to consider:

- Adding a day to the middle of the month, for example 16th June -> 17th June
- Adding a day which will be in the next month (31st July -> 1st August)
- Adding a negative value (-5 days), also check negative values which go on to the previous month
- Adding a day to a leap year (28th February 2008 -> 29th February 2008)
- Check the previous test with non-leap years (28th February 1900 -> 1st Match 1900)

You can also make other tests if you wish.

Problem 10. Tire Pressure Monitoring System

You are given a small project for a system which monitors the pressure in car tires. Your task is to write unit tests for the system. You will need to use mocking in order to pass dependencies. Think about the corner cases of the project.