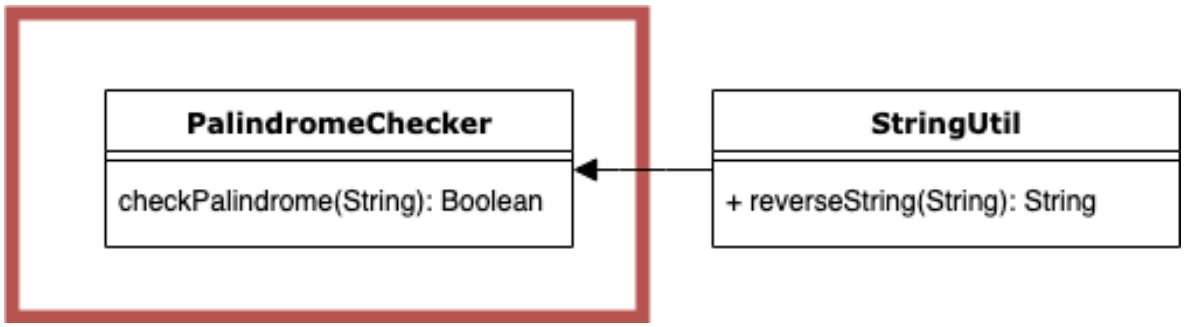
Testing - An Example



Suppose we have a method in a class we want to test. In this example the method is called "checkPalindrome" and expects a String. If the String is the same no matter if read from the front or the back. To reverse the String a Class called "StringUtil" is used.

Example:
Input: "ANNA"
Output: true

Input: "QUACK"
Output: false

The Code in checkPalindrome(String) could look like this (Not handling special characters):

```
public class PalindromeChecker {

    private final StringUtil stringUtil;

    public PalindromeChecker (StringUtil stringUtil) {
        this.stringUtil = stringUtil;
    }

    public boolean checkPalindrome(String original) {
        String reverse = stringUtil.reverseString(original);
        System.out.println(reverse);
        return original.equalsIgnoreCase(reverse);
    }

}
```

To test this specific method (and nothing else) we need to "mock" the StringUtil class. To test the checkPalindrome(String) method we don't actually care about the specific implementation in the StringUtil class. The implementation of the StringUtil class should be tested separately with its own tests. The mock enables us to completely ignore the behaviour of the StringUtil class and dictate its behaviour in our specific case.

Here is an example of what a test class for the checkPalindrome(String) could look like:

```
public class PalindromeCheckerTest {

    private PalindromeChecker classUnderTest;
    private StringUtil stringUtilMock;

    @Before
```

```
    public void setUp() {
        // Create an empty hull of a class, we can control the behaviour of every
method
        stringUtilMock = mock(StringUtil.class);
        classUnderTest = new PalindromeChecker(stringUtilMock);
    }

    @Test
    public void testCheckPalindrome_True() {
        //Setup, what needs to be available to run the method
        String word = "This doesn't matter";
        String reverse = "This neither";

        //Mocking, how should the mocked class(es) behave
        given(stringUtilMock.reverseString(word)).willReturn(reverse);
        given(stringUtilMock.equalStrings(word, reverse)).willReturn(true);

        //Call, actually call the method to be tested
        boolean isPalindrome = classUnderTest.checkPalindrome(word);

        //Assert, check if the result is as expected
        assertTrue(isPalindrome);
    }

    @Test
    public void testCheckPalindrome_False() {
        //Setup, what needs to be available to run the method
        String word = "This doesn't matter";
        String reverse = "This neither";

        //Mocking, how should the mocked class(es) behave
        given(stringUtilMock.reverseString(Mockito.any())).willReturn(reverse);
        given(stringUtilMock.equalStrings(word, reverse)).willReturn(false);

        //Call, actually call the method to be tested
        boolean isPalindrome = classUnderTest.checkPalindrome(word);

        //Assert, check if the result is as expected
        assertFalse(isPalindrome);
    }
```

For a more general test, in the “given(…)” Statement the specific variables “word” and “reverse” could be replaced by Mockito.any() or Mockito.anyString(). This means that the Mock will return the value from the “willReturn(…)” statement no matter the Input. So it doesn’t wait specifically for «This doesn’t matter» but accepts any input and returns the specified object (reverse).

You can think of the mock as an empty hull oft he actual class. If you do not mock a called method it will simply return a predefined value (null or false) depending on the return type oft he mocked method.

Why mocking is important

Mocking is important because you want to test every method / class as individually in your Unit-Tests. This way if you or another developer change something in a specific class and this change breaks something, the test for this specific class will alert you. This way you also know where you have to look for the mistake. If every case is considered and other classes are mocked correctly it is very easy to find a problematic piece of code.

Tests where you test the whole process (without mocking) are called “integration” tests. These often take longer to run and can for example test a specific business case of a process. There are generally fewer integration tests in a Code base than Unit-tests because these cover multiple classes but are less specific and often don’t check all the specific cases of every single method.
The problem with integration tests is, that you can often not immediately see where a bug occurred, and you can spend hours just debugging to find the change that made the integration test fail.

With Unit test the point of failure can be identified a lot quicker and more specific cases can be covered.

A third but (for us) less relevant kind of tests are UI tests. These can be automated or manual. UI Tests are generally a good idea but can cover even less specific cases per method. These tests often don’t cover any logic but just test UI for its functionality because this cannot be tested as thoroughly by unit and integration tests.

These three categories of tests can be visualized as a “Pyramid of Testing”: