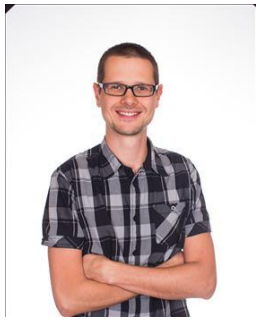

Mockito was created by **Szczepan Faber in 2007**. Dissatisfied with the complexity of existing mocking frameworks, **Faber** initiated the **Mockito** project to provide a more straightforward and user-friendly approach to creating **mock objects** for unit testing in Java. The first production use of **Mockito** was in early **2008** during a project for ***The Guardian* in London,** where **Faber** was part of a **ThoughtWorks team**

Mockito is a popular **Java framework** that simplifies unit testing by enabling developers to create **mock objects.** These mock objects **mimic** the behavior of **real components**, allowing for isolated testing of classes without depending on their actual implementations.

This is especially beneficial when testing classes that interact with external systems, databases, or other complex components.

1. Why Use Mockito?

- **Isolate components** by mocking dependencies.
- **Avoid real database calls, API requests, or expensive operations.**
- **Verify method calls and interactions.**
- **Simulate different scenarios using stubbing.**

Key Concepts in Mockito:

1. **Mocking and Stubbing:**
   - **Mocking:** This involves creating **fake versions of real objects** to simulate their behavior.

     It's particularly useful when **real objects** are impractical to include in tests due to complexity or external dependencies.

     For example, instead of using a **real database connection**, you can **mock the database service to return predefined data.**

   - **Stubbing:** This is the process of **specifying the behavior** of mock objects. You can define return values for specific method calls or specify exceptions that should be thrown under certain conditions. For instance:

```
// Creating a mock List
List<String> mockList = mock(List.class);

// Stub method behavior
when(mockList.get(0)).thenReturn("Hello Mockito
```

     In this example, whenever get(0) is called on mockList, it will return **"Hello, Mockito!".**

2. **Verifying Interactions:**

   Mockito allows verification of interactions between objects. This means you can check whether specific methods were called on a mock object, how many times they were called, and with what arguments.

   This is crucial for ensuring that the class under test interacts with its dependencies as expected. For example:

```
verify(mockList).get(0);
```

This line verifies that the get(0) method was called on mockList.

Verification Methods:

Method	Description
verify(mock).method()	Ensures method was called **at least once**.
verify(mock, times(n)).method()	Ensures method was called **n times**.
verify(mock, never()).method()	Ensures method **was never called**.
verify(mock, atLeast(n)).method()	Ensures method was called **at least n times**.
verify(mock, atMost(n)).method()	Ensures method was called **at most n times**.

3. **Argument Matching:**

   When verifying interactions or stubbing methods, you can specify argument matchers to handle flexible or generic inputs.

   Mockito provides built-in matchers like **any(), eq(), and isA()** to match arguments of various types. For instance:

```
when(mockList.get(anyInt())).thenReturn("Matched!");
```

Here, **anyInt()** is an argument matcher that matches any integer argument, so any call to get with an integer argument will return "**Matched**!".

common Argument Matchers:

Matcher	Description
anyInt()	Matches any int value.
anyString()	Matches any String value.
any(Class<T>)	Matches any object of a given class.
eq(value)	Matches exactly the specified value.

4. **Handling Void Methods:**

   Mocking void methods requires a different approach since they don't return values. Mockito provides methods like **doNothing(), doThrow(), and doAnswer()** to define the behavior of void methods when they are called. For example:

```
// Stub the clear method to do nothing
        doNothing().when(mockList).clear();

        // Use the mock object
        mockList.clear();

        // Verify that the clear method was called once
        verify(mockList, times(1)).clear();
```

In this example, **doNothing()** specifies that when **clear()** is called on **mockList,** it should do nothing. The subsequent verification ensures that **clear()** was indeed called.

Summary

JUnit is a widely-used testing framework in Java that allows developers to write and run tests to ensure their code behaves as expected. It provides annotations like @Test to denote test methods and various assertion methods to validate expected outcomes.

Mockito is a **mocking framework** that **complements JUnit** by enabling the creation of mock objects. These mock objects simulate the behavior of real dependencies, allowing developers to **isolate** the unit of work from its external dependencies during testing.

This is particularly useful when the actual implementations are complex, not yet available, or involve external systems.

How They Work Together:

When writing unit tests, you might encounter scenarios where the class under test interacts with other classes or systems. Using Mockito alongside JUnit allows you to:

- **Mock Dependencies:** Replace real implementations with mock objects to test the class in isolation.
- **Define Behavior:** Specify the behavior of these mock objects using methods like **when(...).thenReturn(...).**
- **Verify Interactions:** Ensure that certain methods were called on the mock objects with expected arguments using **verify(...).**