**MOCKS AND STUBS**

**Test Doubles:** A test double is an object that can stand in for a real object in a test, similar to how a stunt double stands in for an actor in a movie.

**Different Types of Test Doubles:**

1. **Fake:** Fakes are objects that have working implementations, but not same as production one. Usually they take some shortcut and have simplified version of production code.
2. **Dummy:** A Dummy is just an object that you pass to satisfy a constructor, it will not have any method implemented and it shouldn’t.
3. **Spy:** Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
4. **Mock:** Mocks are objects that register calls they receive. In test assertion we can verify on Mocks that all expected actions were performed.
5. **Stub:** Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don’t want to involve objects that would answer with real data or have undesirable side effects.

**Mock:**

Mocks are objects that register calls they receive. In test assertion we can verify on Mocks that all expected actions were performed.

**Example:**

Sending emails is a way to talk to outside world. We need to check the if the email Service is working correctly or not we can’t have a email sent every time we run our tests.

The steps to that we need to follow are:

* Create an interface
* Create a mock implementing the interface
* Write our test

1. Create an interface:

public interface EmailService {

void send(EmailBody emailBody);

}

1. Create a mock implementing the interface:

public class MockEmailService implements EmailService {

private EmailBody lastSentEmail;

@Override

public void send(EmailBody emailBody){

lastSentEmail=emailBody;

}

public void verify(EmailBody emailBody) {

assertEquals(emailBody,lastSentEmail);

}

}

3.Write our test:

@Test

void shouldSendWelcomeEmail() {

MockEmailService emailService=new MockEmailService();

UserService userService = new UserService(null,emailService);

userService.sendWelcomeEmail("hello@gmail.com");

EmailBody expectedEmail = new EmailBody("Welcome","Welcome to the portal","hello@gmail.com");

emailService.verify(expectedEmail);

}

**STUB:**

Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don’t want to involve objects that would answer with real data or have undesirable side effects.

**Example:**

1. **Create an Interface:**

package co.interleap.mocks;

public interface UserRepository {

User findByEmail(String email) throws NotFoundException;

}

1. Stub implementation of interface:

package co.interleap.mocks;

public class StubUserRepository implements UserRepository {

@Override

public User findByEmail(String email) throws NotFoundException {

return new User("9373940583","hello@gmail.com","sanju");

}

}

1. Write test:

@Test

void shouldSendRegisteredPhoneNumberIfUserAccountExists() {

MockEmailService emailService=new MockEmailService();

StubUserRepository userRepository=new StubUserRepository();

UserService userService = new UserService(userRepository,emailService);

userService.sendRegisteredPhoneNumber("hello@gmail.com");

EmailBody expectedEmail = new EmailBody("Account Details",

"Here is your Registered Phone Number: \n" + "9373940583",

"hello@gmail.com");

emailService.verify(expectedEmail);

}

**Handling Exception:**

public User findByEmail(String email) throws NotFoundException {

if ("world@gmail.com".equals(email))

throw new NotFoundException();

return new User("9373940583", "hello@gmail.com", "sanju");

}

}

@Test

void sendAccountNotFoundEmailForUnregisteredUsersWhenTryingToGetRegisteredPhoneNumber() throws NotFoundException {

MockEmailService emailService = new MockEmailService();

StubUserRepository userRepository = new StubUserRepository();

UserService userService = new UserService(userRepository, emailService);

userService.sendRegisteredPhoneNumber("world@gmail.com");

EmailBody expectedEmail = new EmailBody("Account Not Found",

"We do not have a registered account matching your email address",

"world@gmail.com");

emailService.verify(expectedEmail);

}

**MOCKITO**

Mockito is a mocking framework that tastes really good. It lets you write beautiful tests with a clean & simple API. Mockito doesn’t give you hangover because the tests are very readable and they produce clean verification errors.

**Some of the features:**

* Mocks concrete classes as well as interfaces
* Little annotation syntax sugar - @Mock
* Verification errors are clean - click on stack trace to see failed verification in test; click on exception's cause to navigate to actual interaction in code. Stack trace is always clean.
* Allows flexible verification in order (e.g: verify in order what you want, not every single interaction)
* Supports exact-number-of-times and at-least-once verification
* Flexible verification or stubbing using argument matchers (anyObject(), anyString() or refEq() for reflection-based equality matching)
* Allows creating custom argument matchers or using existing hamcrest matchers

**MOCK TEST USING MOCKITO:**

Mock objects are used to verify object behavior during a test. By object behavior I mean we check that the correct methods and paths are exercised on the object when the test is run.

Here is the source code for userservice which is to be tested:

package co.interleap.mocks;

public class UserService {

private final UserRepository userRepository;

private final EmailService emailService;

public UserService(UserRepository userRepository, EmailService emailService) {

this.userRepository = userRepository;

this.emailService = emailService;

}

public void sendWelcomeEmail(String email){

emailService.send(new EmailBody("Welcome", "Welcome to the portal", email));

}

public void sendRegisteredPhoneNumber(String email){

try {

User user = userRepository.findByEmail(email);

EmailBody emailBody = new EmailBody("Account Details",

"Here is your Registered Phone Number: \n" + user.getPhoneNumber(),

email);

emailService.send(emailBody);

} catch (NotFoundException exception) {

EmailBody emailBody = new EmailBody("Account Not Found",

"We do not have a registered account matching your email address",

email);

emailService.send(emailBody);

}

}

}

**The test below creates a mock for the EmailService.**

@Test

void shouldSendWelcomeEmail() {

EmailService emailService = mock(EmailService.class);

UserService userService = new UserService(null, emailService);

userService.sendWelcomeEmail("hello@gmail.com");

EmailBody expectedEmail = new EmailBody("Welcome", "Welcome to the portal", "hello@gmail.com");

verify(emailService).send(expectedEmail);

}

We then call verify on the mocked EmailServie to make sure that the userService calls it's

Send method correctly.

The following line does the checking on the mocked Email Service.

verify(emailService).send(expectedEmail);

This test allows us to show that the email service behaves correctly when sending a mail.

**STUB TEST USING MOCKITO:**

The role of the test stub is to return hardcoded values to the object being tested. These are described as indirect inputs to the test.

In a stub we use the pattern of defining a return value for a method.

Mockito.when(userRepository.findByEmail("hello@gmail.com")).thenReturn(new User("9373940583", "hello@gmail.com", "sanju"));

In a mock we check the behaviour of the object using the following form:

verify(emailService).send(expectedEmail);

Creating a Stub using Mockito for above example:

@Test

void shouldSendRegisteredPhoneNumberIfUserAccountExists() throws NotFoundException {

EmailService emailService = mock(EmailService.class);

UserRepository userRepository = mock(UserRepository.class);

Mockito.when(userRepository.findByEmail("hello@gmail.com")).thenReturn(new User("9373940583", "hello@gmail.com", "sanju"));

UserService userService = new UserService(userRepository, emailService);

userService.sendRegisteredPhoneNumber("hello@gmail.com");

EmailBody expectedEmail = new EmailBody("Account Details",

"Here is your Registered Phone Number: \n" + "9373940583",

"hello@gmail.com");

verify(emailService).send(expectedEmail);

}

**Handling Exception using Mockito:**

@Test

void sendAccountNotFoundEmailForUnregisteredUsersWhenTryingToGetRegisteredPhoneNumber() throws NotFoundException {

EmailService emailService = mock(EmailService.class);

UserRepository userRepository = mock(UserRepository.class);

Mockito.when(userRepository.findByEmail("world@gmail.com")).thenThrow(new NotFoundException());

UserService userService = new UserService(userRepository, emailService);

userService.sendRegisteredPhoneNumber("world@gmail.com");

EmailBody expectedEmail = new EmailBody("Account Not Found",

"We do not have a registered account matching your email address",

"world@gmail.com");

verify(emailService).send(expectedEmail);

}

**REFERENCES**

[**https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/**](https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/)

[**https://www.youtube.com/watch?v=53fncREn8k0&list=PL6Zs6LgrJj3vy7yWpH9xb3Y0I\_pAPrvCU**](https://www.youtube.com/watch?v=53fncREn8k0&list=PL6Zs6LgrJj3vy7yWpH9xb3Y0I_pAPrvCU)

[**https://stackoverflow.com/questions/15976008/using-mockito-to-stub-and-execute-methods-for-testing?newreg=790d2fac4b764077820ce3ae7656e574**](https://stackoverflow.com/questions/15976008/using-mockito-to-stub-and-execute-methods-for-testing?newreg=790d2fac4b764077820ce3ae7656e574)

[**https://www.javacodegeeks.com/wp-content/uploads/2016/09/Mockito-Programming-Cookbook.pdf**](https://www.javacodegeeks.com/wp-content/uploads/2016/09/Mockito-Programming-Cookbook.pdf)

[**https://www.youtube.com/watch?v=3bjGfHx5mrk&list=PLzS3AYzXBoj9Gpuo\_EpyaZ8mGKExfLM8c**](https://www.youtube.com/watch?v=3bjGfHx5mrk&list=PLzS3AYzXBoj9Gpuo_EpyaZ8mGKExfLM8c)