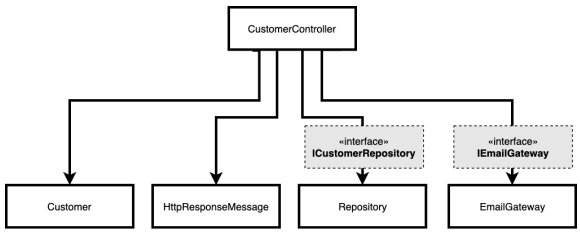
Break.{IT}

Home->Tasks->Active

Task: EmailGWIntegration Previous Tasks

# Integration Testing: Mocks and Stubs

The following hierarchy of components represents a structure of dependencies between components.




Some of the previous components are simple and can be implemented with low effort. Hence, it's better to code and integrate them, instead of using **stubs** (described in the presentation). **Which components require stubs?**

The **rule of thumb** is to use stubs for

- **not implemented components** that can be **replaced by *doubles* which can provide canned answers** equivalent to those offered by the real component;

- **external subsystems** out of the programmer's control.

Let's suppose that the *Repository* and the *EmailGateway* are the most complex components from the hierarchy. Shall we create stubs for each of them to test the *CustomerController* or wait until their implementations are available?

EmailGateway

The *EmailGateway* provides an interface with low complexity, but its implementation may be complicated.

```
public interface IEmailGateway {
    public void sendSpecialGreetings(Customer customer);
    public void sendRegularGreetings(Customer customer);
}
```

We only have one test condition to validate

- Are emails been sent according to the city of the customer?

Finally, gateway email components depend on external infrastructure (e.g., SMTP server) that's out of programmer's control, which may not be available or configured for testing purposes.

The previous arguments combined, make the gateway an excellent candidate for a *stub.*

## Repository

Similarly to the *EmailGateway* component, the *Repository* has a low complex interface.

```
public interface ICustomerRepository {
    public Customer getCustomerByEmail(String s);
    public void save(Customer customer);
}
```

However, one stub for the Repository component should hold canned answers that makes the implementation effort of the stub higher, namely

- Is the save method verifying for existing emails when saving new customers?

- Is the customer saved in the database after invoking the save method?

Finally, since the repository is under the programmer's control, he/she may decide that the real component contribution is higher than the stub counterpart for testing purposes.

The next step is to integrate the repository with the real *CustomerController*.

## Mocks or Stubs?

The EmailGateway interface has two methods. Each method sends a specific greeting email. **How to test whether the correct greeting message has been sent to the customer?**

A **mock object is used to testify the method invocation**, but without checking the EmailGateway's state, **it's impossible to determine the greeting message sent**. In this scenario, we **need a *stub*** instead of a *mock* object.

### Exercise 1

Import the components' implementation into a **new project**.

Implement the *EmailGatewayStub*. You can generate the methods' skeleton from the interface automatically, using the options of the *IntelliJ's Code* menu.

**Create a new Test Case** (named *EmailGWIntegration*) and **add a new test** that:

-**creates** a new **customer**

-**asserts** each of the **customer's fields** stored in the **repository**

-**sends** a **greeting message** to the customer

-**asserts** the number of emails sent by the gateway

The *EmailGatewayStub* should implement the following method to validate the last assertion.

```
public static int getSentEmails(){
    return this.sentEmails;
}
```

**The created *stub* doesn't look like a *stub*. Why?**

## Exercise 2

**That's true!!! Our stub is actually a mock object.**

It's **not possible to test important objects'** state using **mock objects**. As an example, we **didn't evaluate** the correctness of the **message** sent to the customer. A fault in the implementation of the EmailGateway may lead this component to switch greeting messages. **We need a *stub* to validate such a condition.**

**How to create a stub that asserts the greeting message sent to the customer?**

We need to **evaluate the object's state** to obtain the list of sent messages.

```
public static ArrayList<String> getMessagesSent() {
    return messagesSent;
}
```

We also have to **change the test case** to **evaluate the correctness of the message** sent to the customer.

Now, you can **submit the JAR file for testing**.

## Exercise 3

One last challenge. The test case implementation is brittle, as it depends on the update of the list of messages sent every time one of the ***send**** methods is invoked.

```
@Override
public void sendRegularGreetings(Customer customer) {
   //...
   messagesSent.add("Hello Dear Customer" + customer.getName());
}
```

**Tests can be broken** if the programmer adds **new greeting formats** without updating the list of messages sent. On top of that, we have **code duplication**. How to solve this design issue?

### Tip

*"All problems in computer science can be solved by another level of indirection (the fundamental theorem of software engineering)"*

### Specification

### Deadline
2020-04-22 23:55

Submit for Testing | Submit for Evaluation