COURSEWARE

- Professional Skills
- Agile Fundamentals
- Jira
- Git
- Databases Introduction
- Java Beginner
- Maven
- Testing (Foundation)
- Java Intermediate
- HTML
- CSS
- Javascript
- Spring Boot
- Selenium
- Sonarqube
- Advanced Testing (Theory)
  - Integration Testing
  - System Testing
  - Acceptance Testing
  - Behaviour-Driven Development (BDD)
  - Non-Functional Testing
- Cucumber
- MongoDB
- Express
- NodeJS
- React
- Express-Testing
- Networking
- Security
- Cloud Fundamentals

Integration Testing

Contents

- Overview
  - Integration Testing
    - What is Integration Testing
    - Approaches
    - Why do we perform Integration Testing
- Tutorial
  - How to perform Integration Testing
- Exercises

Overview

Below we will be covering the basics of Integration testing, exploring how and why we use this testing approach.

Integration Testing

What is Integration Testing

Integration testing is a testing approach that targets the very fundamental building blocks of an application, the idea is to prove that each 'integration' of the application is functioning as expected.

Approaches

Because the communication between objects is being tested, we have to determine a way we want to follow the communication relationships.

Several strategies are suggested:

- Top-Down

  Top-Down is a means of targeting from the highest level of the system and testing the communication between each object to the lowest level of the system.

- Bottom-Up

  Bottom-Up is a method of targeting from the lowest level of the system to the highest, this might involve the communication and collection of database data.

- Sandwich

  Sandwich testing is a combination of testing `top-down` and `bottom-up` to ensure bi-directional communication has been covered.

- Big Bang

  Big Bang testing bundles all of the systems `units` and runs tests on the whole system, with the primary goal of asserting that the communication between each `unit` at each step completes its expected goal.

  Whilst this might be convenient for small systems, it can be hard to maintain and follow for larger systems.

  > Big Bang testing is very close to `System` testing rather than `Integration`, however, the purpose of Big Bang is to identify and prove the units intercommunication works as expected.

AWS Foundations
AWS Intermediate
Linux
DevOps
Jenkins Introduction
Jenkins Pipeline
Markdown
IDE Cheatsheet

Stubs and Mocking

Just as with unit testing, to isolate the focused targets you may wish you utilise `stubs` or `mocks` to remove class or unit behaviour that is outside the scope of testing.

Why do we perform Integration Testing

To explain why we perform Integration testing, it is valuable to have a system or structure we can perform testing on. In this context, we will be using the `Repository` design pattern.

> If you have designed a SQL "CRUD" system using object-oriented you may be familiar with the `Repository` model/pattern.

If you are not familiar, the example below defines the relationship between a three-layer system.

An example structure:

▶ Expand

The objective of integration testing is to determine that there are no errors between each of this object communications.

> To simplify the example, the terms used will be `Controller`, `Service`, `DAO` (data access object).

We know that the `Controller` communicates to the `Service` object, whilst we might have created unit tests for both we have not proven that they **can** communicate together.

Cross-Object Communication

The focus here is to prove `Controller.printCustomers()`->`Service.printCustomers()`, whilst they appear the same,`Controller.printCustomers()` may use and actively depend on the `Service` object as well as the `Service.printCustomer()` method.

Likewise, we may create these in either a `top-down` or `bottom-up` approach.

Structure:

- `Controller` -> `Service` -> `DAO`

Top-Down Approach

In `top-down` approach we would begin investigating the relationship from the `Controller` -> `Service`, then `Service` -> `DAO`.

> This is because we are focusing on how the `Controller` relates to the `Service`, then the `Service` to the `DAO`.

Bottom-Up Approach

In `Bottom-Up` approach we would begin investigating the relationship from the `Service` -> `DAO`, then we would test `Controller` -> `Service`.

> This is because we are focusing on how the `DAO` relates to the `Service`, then the `Service` to the `Controller`.

Sandwich Testing

The importance is placed on the flow of data as much as the testing approach. In the above examples, the predominate behaviour of the application lends itself well to `top-down` integration.

Communication Flow

This is because it is unlikely that the `DAO` would ever call the `Service`, in these cases however the direction of the communication may be bi-directional, as such, we can see that neither a `top-down` nor `bottom-up` approach would completely capture the bi-directional communication.

This is why some testers might even go as far as to incorporate both approaches to improve coverage, this is known as `Sandwich testing`.

Tutorial

How to perform Integration Testing

There is no reason why Integration testing cannot be performed with `Junit`, it is the scope in what we are testing that is different. As seen below, the object is to test `Controller.printCustomers()`, which relies on `Service.printCustomers()`.

Example:

```
class Controller {
    protected Service myService;
    // Constructor etc here

    public boolean printCustomers(){
        return myService.printCustomers();
    }
}

class Service {
    // Constructor etc here
    public boolean printCustomers(){
        ArrayList<Customers> list;

        try{
            // for loop print list
        }catch(Exception e){
            return false;
        }

        return true;
    }
}
```

As you can observe, we are trying to determine if a condition within `Service.printCustomers()` is returning `true` by calling `Controller.printCustomers()`.

We know that running these classes together should return `true` if the execution of the `Service` method does not throw an error.

If `Service` were to return `false`, we know that the communication between these two classes is flawed somehow and needs investigating.

> Whilst this is a simple example, think about a method in `Controller` that might pass parameters and perform complex behaviour in `Service`, this might lead to potential disaster.

Example:

```
class Controller_INT_TEST {

    private Controller testController;

    @Test
    public void pntCust_TEST() {
        testController = new Controller();

        // assert myService.printCustomers(); has run as expected
        assertTrue(testController.printCustomers());
    }

}
```

Exercises

There are no Exercises with this module.