INTEGRATION TESTING

LEARNING OBJECTIVES

- Understand integration testing in general
- Know different approaches to integration testing
- Handle production and test databases separately
- Automate integration tests using test databases, database testing frameworks, in-memory databases and mocking

INTRODUCTION

Integration testing is testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.
Phase in software testing in which individual software modules are combined and tested as a group that occurs after unit testing and before system testing.
Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.
The purpose of is to verify functional, performance, and reliability requirements placed on major design items

Integration testing is a logical extension of unit testing.
In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit.
In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Assemblages / Groups of units
Test whether all the components within assemblages interact correctly.

If all units work individually, why doubt that they work together?
The point where objects interact is major contributor of bugs!

Integration testing identifies problems that occur when units are combined. By using a test plan that test each unit and ensure the viability of each before combining units, any errors discovered when combining units are likely related to the interface between units. This method reduces the number of possibilities to a far simpler level of analysis.

Focus: Interaction and communication between components Integration, not the functionality of either of them

Problem: The greater the scope of integration testing, the harder it becomes to isolate failures to a specific interface

Example: Code that accesses other objects, services, files, databases or other external resources

Implications: Instantiate objects / Mock objects / Known state

APPROACHES

Integration testing of a software product can be done in a number of ways throughout the development cycle
Different approaches to integration testing are big-bang, top-down, bottom-up, mixed / sandwich and risky-hardest

BIG BANG

Approach where all or most of the units are combined together and tested at the same time
Approach is taken when the entire software is bundled

Individual modules are not integrated until and unless all the modules are ready

Pros

All code is finished before integration testing starts
No need for simulation

Cons

Difficult and time-consuming to trace cause of failure

TOP-DOWN

Approach where top level units are tested first and lower level units are tested step by step after that
Approach is taken when top down development approach is followed
Test Stubs are needed to simulate lower level units which may not be available during the initial phases

Pros

Test cases can be defined in terms of the functionality of the system, starting from GUI
High-level logic and data flow is tested early in the process
Allows program demonstration to the user

Cons

Writing stubs can be difficult
Stubs must allow different possible conditions to be tested
Poor support for early release of limited functionality

BOTTOM-UP

Approach where bottom level units are tested first and upper level units step by step after that
Approach is taken when bottom up development approach is followed
Test drivers are needed to simulate higher level units which may not be available during the initial phases

Pros

Utility modules are tested early in the development process
The need for stubs is minimized
In OO systems xUnit tests can be drivers

Cons

Tests the user interface components last
Poor support for early release of limited functionality

SANDWICH / HYBRID / MIXED

Approach which is a combination of Top-Down and Bottom-Up approaches with layers above and below a middle layer

Pros

Top and bottom layers can be tested in parallel

Cons

Extensive testing of the sub-systems is not performed before the integration

OTHER APPROACHES

Incremental
Functional Incremental

DATABASE

Data Base Layer Testing

Challenges for data access testing
Persistence layer is difficult to test

Persistence layer requires interaction with an external database
Code that accesses the database can be cumbersome
Database access is relatively slow which gives slow tests
A good test is self sufficient and creates all data it needs
Deleting and inserting data for every test may seem like a big time overhead, but it works
Make a setup method that all database tests call to put the database in a known state

Database layer testing considerations

- Production database
- Test database
- SQL Script
- Connector
- Database testing framework
- In memory database
- Mocking

MYSQL / CONNECTOR / SQL SCRIPT

1. Create sql script for test database
2. Create datasource for test database
3. Create connection, execute sql script by executing statements and set database to a known state before tests

DBUNIT

DBUnit is JUnit extension for database testing
Set up database state and expected results with DBUnit framework

- Initialize database into known state before testing
- Import and export data as XML datasets
- Use xml datasets to alter database and define expected

Dependency: org.DBUnit:DBUnit-2.6.0

Create database connection

```
dbConnection = new DatabaseConnection(connection, "cupcakeshop");
dbConnection.getConfig().setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY, new MySqlDataTypeFactory());
dbConnection.getConfig().setProperty(DatabaseConfig.PROPERTY_METADATA_HANDLER, new MySqlMetadataHandler());
```

Load xml dataset

```
xmlDataSet = new FlatXmlDataSetBuilder().build(new FileInputStream("DATASETS/dataset_Init.xml"));
DatabaseOperation.CLEAN_INSERT.execute(dbConnection, xmlDataSet);
```

Get database dataset with tables

```
IDataSet databaseDataSet = new QueryDataSet(dbConnection);
databaseDataSet.addTable("cupcake");
```

Get table

```
databaseTable = databaseDataSet.getTable("cupcaketopping");
```

Assert tables

```
Assertion.assertEquals(xmlTable, databaseTable);
```

H2

Java SQL database
Create embedded, in-memory or network databases

Dependency: com.h2database:h2-1.4.198

Embedded
jdbc:h2:cupcakeshop;
Memory
jdbc:h2:mem:cupcakeshop;DB_CLOSE_DELAY=-1
Network
jdbc:h2:tcp://localhost//data/cupcakeshop

MOCKITO

Use Mockito to mock away datasource, connection, preparedstatement and resultset

RESOURCES

Integration testing
https://www.guru99.com/integration-testing.html
http://tryqa.com/what-is-integration-testing/
https://www.softwaretestinghelp.com/tools/40-best-database-testing-tools/

DBUnit
http://dbunit.sourceforge.net/index.html
http://dbunit.sourceforge.net/intro.html

H2
http://www.h2database.com/html/main.html