FIT5171 Tutorial 6
Integration Testing

Week 6, 2023

Please do try the questions before coming to the tutorial. Your active participation is the most important!

Integration Testing

1. Suppose we are performing integration testing on a simple Web application for management of personnel, which deals with classes `Person`s, `Expertise`, `Mission`s and `Invitation`s.

   Let's assume *authentication* and *authorisation* has been designed and implemented in the system. With some simplification, the interactions between components for the loading of a `Mission` page is shown below in Figure 1, showing the authentication & authorisation control flow.

   The component **Router** is responsible for routing HTTP requests to the appropriate resources that handles them. As the name suggests, **MissionRoute** is responsible for creating pages to present mission details. Once it is determined that authentication and authorisation are required, the resource consults the **LoginManager** to grant/refuse access based on user input. The **LoginManager** in turn invokes **UserDAO** to load user details given the user name. If access is granted or that access control is not required, the resource then invokes **MissionDAO** to obtain mission details. If the user is denied access, it constructs a page to show the error accordingly. Finally, **MissionRoute** returns the page back to the **Router**.

   Note that the graphs within each component represent *significantly* simplified program graphs, with only some important control structures shown. Nodes in these graphs don't represent individual statements, but rather blocks of statements that can be grouped together logically. The *italic* label beside each node explains the main functionality of that node. Edges represent control flow. Thick edges represent transfer of control (message passing and return) between components.

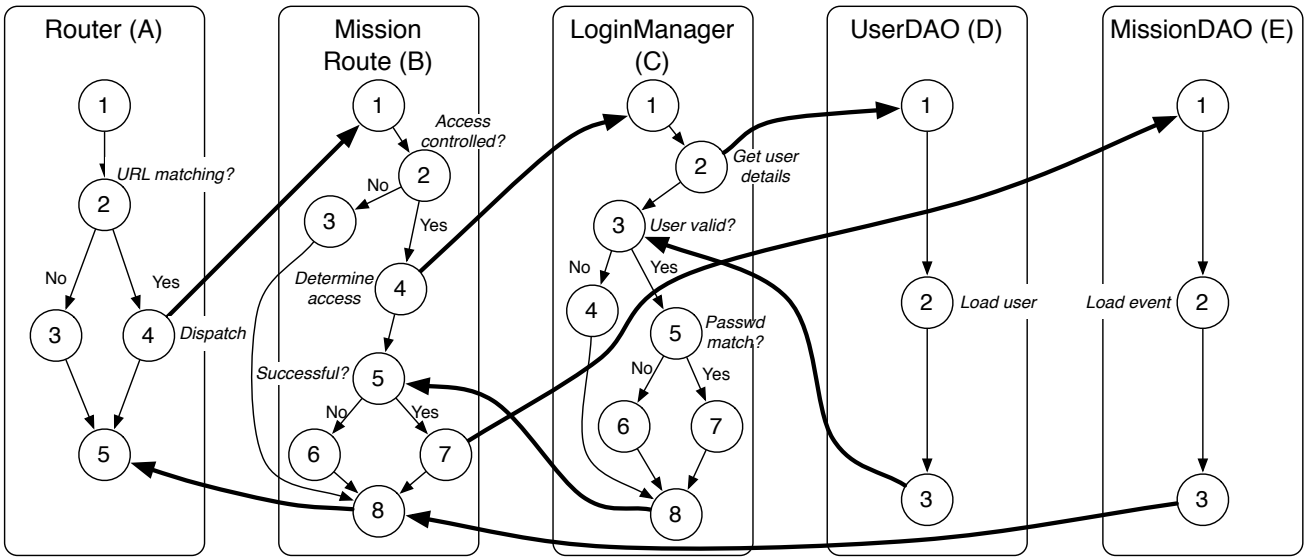


Figure 1: The interactions of some components of a simple Web application that handle `Missions`.

(a) List **all** MM-paths for **all** the 5 components above. You can use letters A, B, C, D and E to represent the 5 components in Figure 1. Recall that an MM-path starts from a *source* node and ends at a *sink* node, with no intervening sink nodes.

For example, the MM-path for component D (`UserDAO`) is $MEP(D, 1) = (1, 2, 3)$. The MM-path for component E (`MissionDAO`) is $MEP(E, 1) = (1, 2, 3)$.

Solution:

The MM-paths for the components are:

MEP(A,1) = (1,2,3,5)
MEP(A,2) = (1,2,4)
MEP(A,3) = (5)
MEP(B,1) = (1,2,3,8)
MEP(B,2) = (1,2,4)
MEP(B,3) = (5,6,8)
MEP(B,4) = (5,7)
MEP(B,5) = (8)
MEP(C,1) = (1,2)
MEP(C,2) = (3,4,8)
MEP(C,3) = (3,5,6,8)
MEP(C,4) = (3,5,7,8)

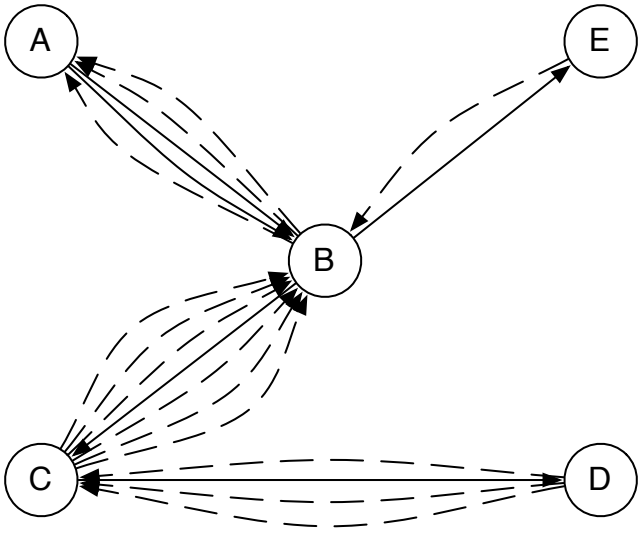
(b) Recall that an MM-path graph is a directed graph whose nodes are MM-paths and edges are transfer of control (message passing & return) between components. Based on the above MM-paths you developed in part (a), do the following:

(1) Draw the MM-path graph for the components.

(2) Calculate the MM-path complexity for the graph you just drew.

Solution:

The MM-path graph can be found below.



The edges between nodes are:

- 5 edges between A and B
- 2 edges between B and E
- 4 edges between C and D
- 7 edges between B and C

The Cyclomatic complexity is $V(G) = \#E - \#V + p = 18 - 5 + 1 = 14$.

2. One of the goals of integration testing is to be able to isolate faults when a test case causes a failure. Consider integration testing for a program written in a procedural programming language. Rate the relative fault isolation capabilities of the following integration strategies.

   A Big bang

   B Decomposition-based top-down integration

   C Decomposition-based bottom-up integration

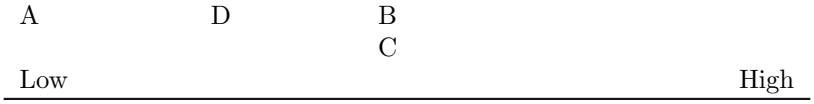
   D Call graph-based neighbourhood integration (radius 1)

   Please also provide rationales for your choices.

   You can show your ratings graphically by placing the letters corresponding to a strategy on a continuum like below. As an example, suppose Strategies X and Y are about equal and not very effective, and strategy Z is very effective.

   Note that this rating is relative and qualitative, so don't agonise over where exactly to put a strategy!

Solution:



Mocking with Mockito

Mocking is an important concept in unit and integration testing, especially for decomposition-based integration testing approaches such as top-down and bottom-up. In these approaches, *drivers* and *stubs* may need to be created to emulate behaviours of components that are needed in integration testing, but have not been implemented.

Mocking is also a widely used tool in practice. For Java, quite a number of open source mocking frameworks have been created to assist the creation of mock objects. In this tutorial, we focus on one such framework, Mockito,[1] and practice how to use it to emulate and verify behaviours of mocked objects. We will do this by extending the assignment 2 code base.

3. Discuss some of the benefits brought by mocking.

Solution:

Some benefits are:

- Be able to create tests before actual implementation, hence enable TDD.
- Be able to create tests that access resources that are inaccessible or expensive to access.
- A team can work in parallel.
- Be able to create prototypes quickly.
- Be able to isolate components to identify defects more easily

[1] `http://mockito.org/`

Suppose that with a design change (remember, changes do happen!), it is decided that an additional layer of abstraction will be added to handle the storage of `User`s that adds additional functionality (e.g., validation) on top of the underlying DAO layer. This new class is called `UserHandler` and it will interact with `UserDAO` to achieve its functionalities.

4. Explain some of the difficulties in TDD for `UserHandler` without mocking.

Solution:

The major difficulties are

- `UserDAO` may not have been created when `UserHandler` needs to be implemented (and tested).
- Even if `UserDAO` has been created, accessing the database may be slow, hence using the real `UserDAO` object may slow down the entire testing process.

5. Study the Mockito documentation to learn how the following is done.
    1. Creating a mock object
    2. Specifying expected return results when a certain method is called
    3. Add partial mocking support to real objects

Solution:

1. Use the `org.mockito.Mockito.mock(...)` method.
2. Use the `org.mockito.Mockito.when(...).thenReturn(...)` sequence of methods.
3. Use the `org.mockito.Mockito.spy(...)` method.

6. Suppose that we want to implement `UserHandler.updatePassword(Long, String)`, which takes 2 parameters: a `Long` value representing a `User`'s ID, and a `String` value representing the new password value. `updatePassword()` returns `true` if the update is successful and `false` otherwise. Two conditions can make an update fail:

   1. A null password is provided as the parameter value, or
   2. The new password is the same as the current one.

   Moreover, if no `User` object can be retrieved with the given ID, `updatePassword` should throw an `SQLException`.

   Implement the above method in `UserHandler` class. Note that this class should make use of `UserDAO` class to access the database.

Solution:

```
package prolist.logics;

import prolist.dataaccess.UserDAO;
import prolist.model.User;

import java.sql.SQLException;

/**
 * @author Yuan-Fang Li
 * @version $Id: $
 */
public class UserHandler {
    private UserDAO userDAO;

    public UserHandler(UserDAO userDAO) {
        this.userDAO = userDAO;
    }

    public boolean updatePassword(Long personId, String password) throws SQLException {
        boolean result = false;

        User user = userDAO.load(personId);

        if (null == user) {
            throw new SQLException("No user found with ID: " + personId);
        }

        if (null != password && !user.getPassword().equals(password)) {
            user.setPassword(password);
            userDAO.update(user);
            result = true;
        }
        return result;
    }

}
```

7. Develop test cases for the `UserHandler` class. Decide whether/when to use Mockito. Suppose you want to test for the following scenarios.

    1. Null person to be retrieved from a given ID.
    2. Null password provided as the parameter value.
    3. New password same as the current one.

Solution:

```
package prolist.logics;

import org.junit.Before;
import org.junit.Test;
import prolist.dataaccess.UserDAO;
import prolist.model.User;

import java.sql.SQLException;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

/**
 * @author Yuan-Fang Li
 * @version $Id: $
 */
public class UserHandlerUnitTest {
    private UserHandler userHandler;
    private UserDAO userDAO;

    @Before
    public void setUp() {
        userDAO = mock(UserDAO.class);
        userHandler = new UserHandler(userDAO);
    }

    @Test
    public void nullPersonUpdateFalse() throws SQLException {
        long id = 0L;
        when(userDAO.load(id)).thenReturn(null);
        try {
            userHandler.updatePassword(id, "password");
            fail("Should have thrown an exception.");
        } catch (SQLException e) {
            assertTrue(e.getMessage().contains("No user found"));
        }
    }

    @Test
    public void nullPasswordUpdateFalse() throws SQLException {
        long id = 0L;
        when(userDAO.load(id)).thenReturn(new User());
        boolean successful = userHandler.updatePassword(id, null);
        assertFalse("Null password", successful);
    }

    @Test
    public void samePasswordUpdateFalse() throws SQLException {
        long id = 0L;
        User user = new User();
        String password = "abcd";
        user.setPassword(password);
        when(userDAO.load(id)).thenReturn(user);
        boolean successful = userHandler.updatePassword(id, password);
        assertFalse("Same password", successful);
```

```
    }
}
```