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 `Persons`, `Expertise`, `Missions` and `Invitations`.

   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.

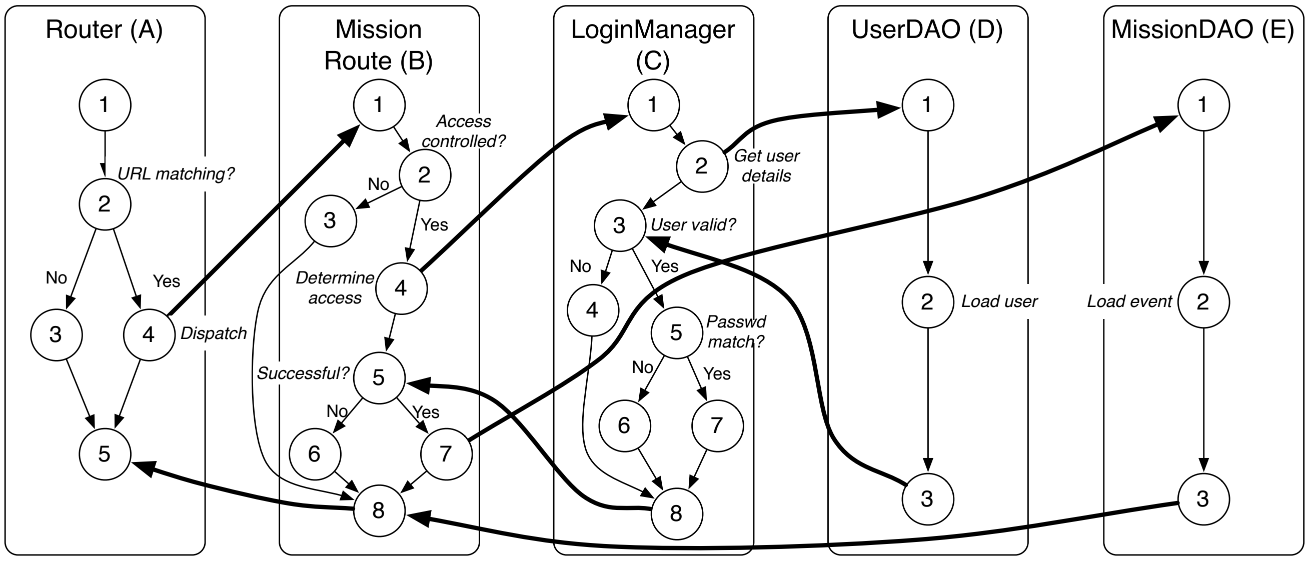


Figure 1: The interactions of some components of a simple Web application that handle `Mission`s.

(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:

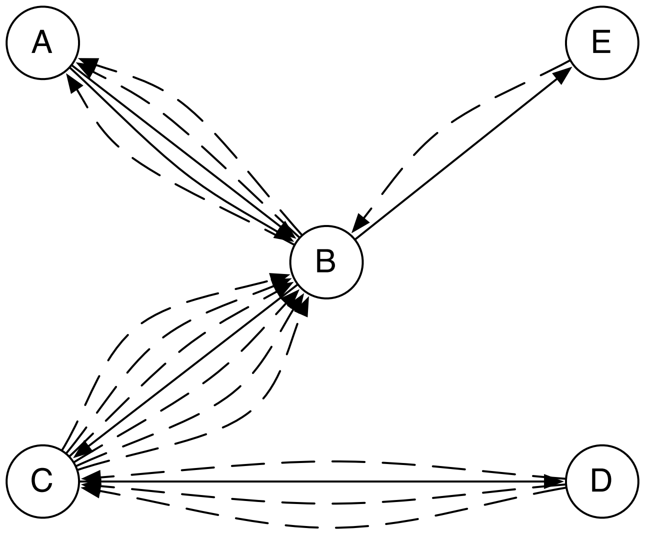
MEP(A,1) =	(1,2,3,5)
MEP(A,2) =	(1,2,4)
MEP(A,3) =	(5)
MEP(B,1) =	(1,2,3,8)
MEP(B,2) =	(1,2,4)
MEP(B,3) =	(5,6,8)
MEP(B,4) =	(5,7)
MEP(B,5) =	(8)
MEP(C,1) =	(1,2)
MEP(C,2) =	(3,4,8)
MEP(C,3) =	(3,5,6,8)
MEP(C,4) =	(3,5,7,8)

(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!

```
 Y
 X                                              Z
 Low                                         High
__________________________________________________
```

Solution:

```
 A            D          B
                         C
 Low                                          High
__________________________________________________
```

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:

```java
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:

```java
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);
    }
}
```

FIT5171 Tutorial 7
Integration Testing

Week 7, 2022

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 Persons, Expertise, Missions and Invitations.

   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.

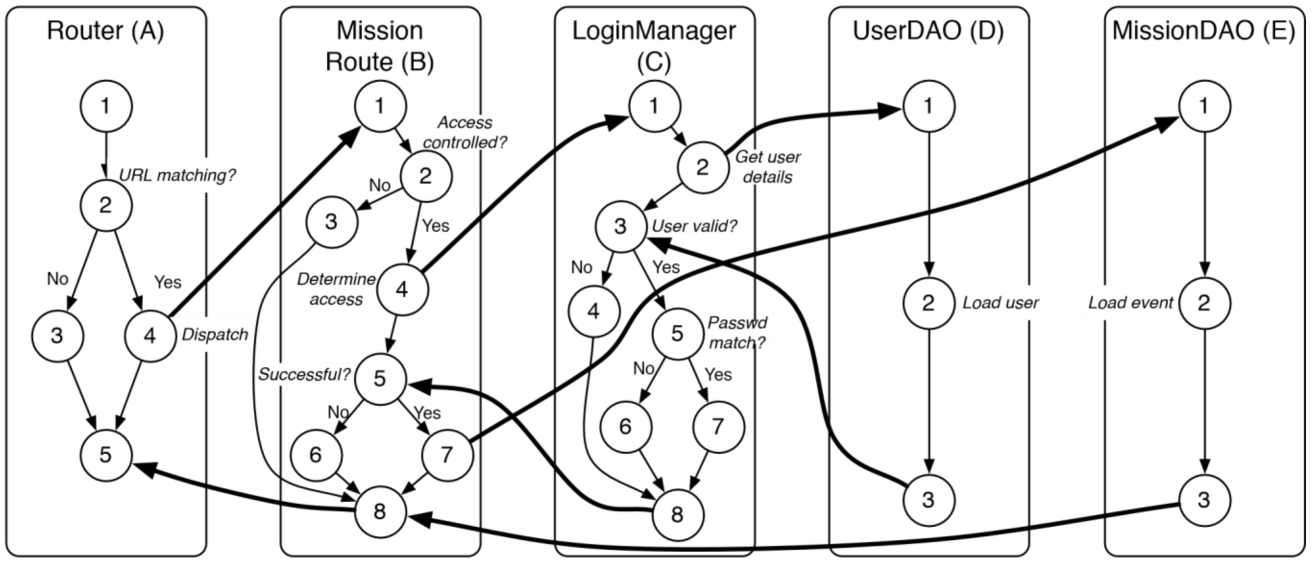


Figure 1: The interactions of some components of a simple Web application that handle Missions.

(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)

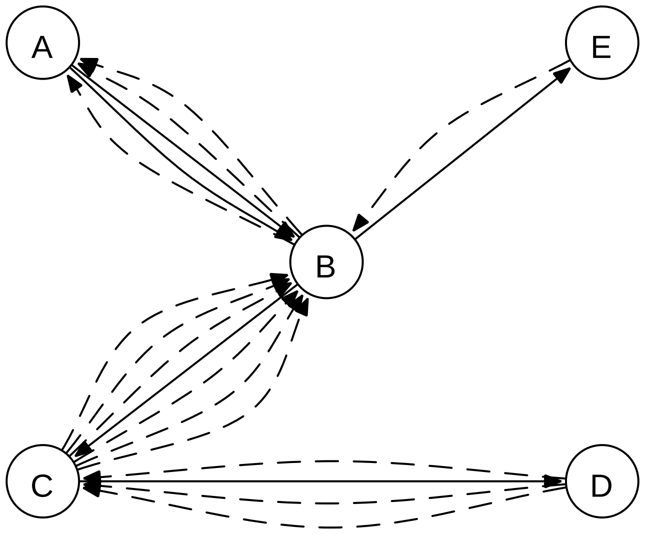
(b) Recall that an MM-path graph is a directed graph whose nodes are MM-paths and edges are transfer of control (message passing & return) between components. Based on the above MM-paths you developed in part (a), do the following:

(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

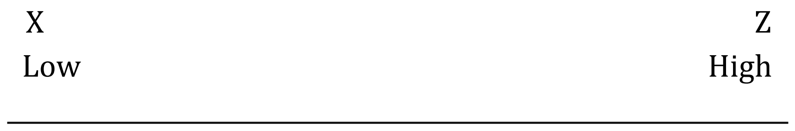
   D Call graph-based neighbourhood integration (radius 1)

   Please also provide rationales for your choices.

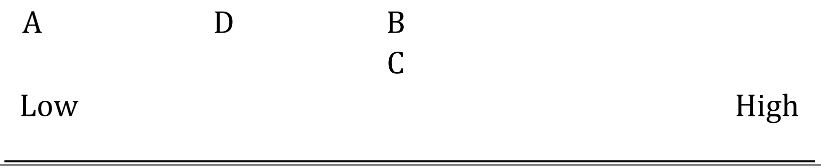
   You can show your ratings graphically by placing the letters corresponding to a strategy on a continuum like below. As an example, suppose Strategies X and Y are about equal and not very effective, and strategy Z is very effective.

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

Y



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

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 Users 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.

[1] http://mockito.org/

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);
        }
}
```

FIT5171 Tutorial 8
Software complexity & metrics

Week 8–9, 2023

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

Weyuker's 9 properties have been proposed to evaluate software metrics. Some of the properties (for example, properties 1, 3, 4 and 8) are quite simple and intuitive. However, some other properties are a bit more complicated and needs further analysis.

In this tutorial we will pick two properties (5, 6) and one software complexity metric from each category (structure, testing and object-oriented) and **informally** prove whether the above properties hold or not. If not, give a counter example.

Structure For structure metrics, we choose the morphology metric Tree Impurity: $TIP = \frac{2(\#E - \#V + 1)}{(\#V - 1)(\#V - 2)}$.

Testing For testing metrics, we choose the simple statement coverage metric C_0.

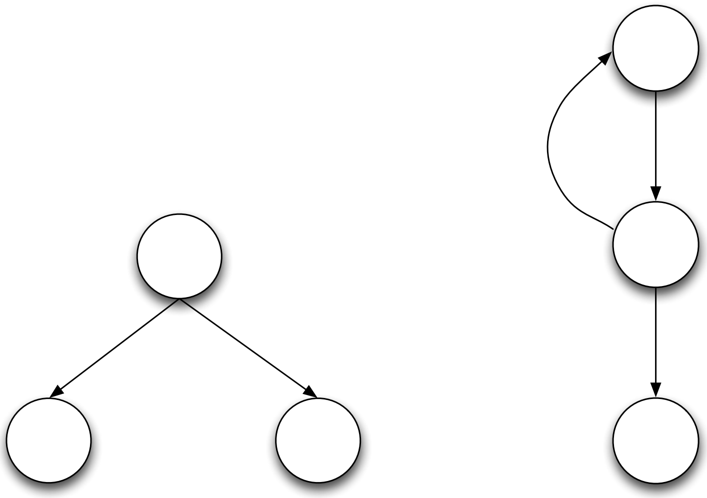
OO For object-oriented metrics we choose the metric Response For a Class: RFC, equal to the number of methods invocable.

We will restrict our discussion to a single language (Java or C#, for example) for simplicity. We also assume that program composition (+) can be either sequence or nesting.

1. Property 5: The complexity of a program segment should be that of the whole program, i.e., $\forall P, Q \bullet M(P) \leq M(P + Q) \wedge M(Q) \leq M(P + Q)$.

(a) Structure metric *TIP*.

Solution:
TIP measures how much a (program) graph deviates from a pure tree (in which each node has at most one parent and there is no cycle). Intuitively, the more deviated a graph is from a tree, the higher the *TIP* value is and the more complex the graph is.
Two example graphs are shown in Figure 1 below.



(a) Graph for P (b) Graph for Q

Figure 1: Two simple graphs with different *TIP* values.

Graph for P above is a pure tree and hence $TIP(P) = 0$. Graph for Q has an extra edge so $TIP(Q)$ is

$$TIP(Q) = \frac{2 * (3 - 3 + 1)}{(3 - 1) * (3 - 2)} = 1 \quad (1)$$

Solution: (continued)

However, if we compose the two program together sequentially, we will have what's shown in Figure 2.

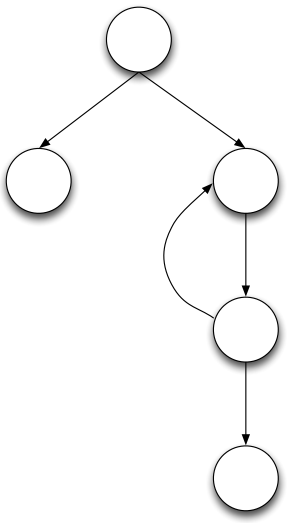


Figure 2: A sequential composition, $P + Q$, of the above two programs.

This graph has 5 nodes and 5 edges, hence the *TIP* value of the above graph in Figure 2, is then

$$TIP(P + Q) = \frac{2 * (5 - 5 + 1)}{(5 - 1) * (5 - 2)} = \frac{1}{6} \tag{2}$$

Compare Formulas (1) and (2), we can see clearly that for this example, $TIP(Q) > TIP(P + Q)$. Hence property 5 does **not** always hold for *TIP*.

(b) Testing metric C_0.

Solution:

If we treat C_0 as a coverage criteria to mean 100% coverage, then clearly property 5 holds for C_0.

However, if we treat C_0 as a coverage metric that measures the percentage of the statements covered by testing, then it becomes more interesting.

For program P, we denote $C_0(P) = \frac{e_P}{L_P}$, where e_P denotes the number of statements covered by testing and L_P denotes the total number of statements of P. Similarly, we have $C_0(Q) = \frac{e_Q}{L_Q}$ and $C_0(P+Q) = \frac{e_{P+Q}}{L_{P+Q}}$.

We observe that $e_{P+Q} = e_P + e_Q$ and that $L_{P+Q} = L_P + L_Q$. Hence, $C_0(P+Q) = \frac{e_P + e_Q}{L_P + L_Q}$.

Let's assume for a particular program P, $C_0(P) = 100\%$ and it has 10 lines of code. Also assume that there exists a program Q with $10,000$ lines of code but coverage 0%. Hence, for the sequential composition of the two, $P + Q$, $C_0(P+Q)$ is then

$$C_0(P+Q) = \frac{10+0}{10+10000} = \frac{1}{1001} < C_0(P) = 100\%$$

Hence, the above counterexample shows that property 5 does **not** always hold for C_0.

(c) OO metric RFC.

Solution:

RFC measures the complexity of a *class* by simply counting the number of methods in this class and all its super classes.

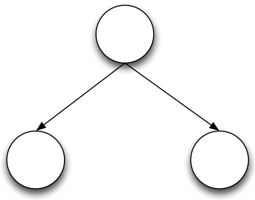
Because RFC measures the complexity of classes, both P and Q must be classes.

If we take sequence composition to be inheritance between classes, *i.e.*, $P + Q$ is class P with added super class Q. Then $RFC(P+Q) \geq RFC(P)$ and $RFC(P+Q) \geq RFC(Q)$, since the class $P + Q$ contains the set of methods that is the union of the sets of methods in both P and Q. Hence, if the composition is sequence composition, then property 5 holds for RFC.

2. Property 6: The complexity of the composition of two programs P and R may not be the same as the composition of programs Q and R, even though P and Q have the same complexity, i.e., $\exists\, P, Q, R \bullet M(P) = M(Q) \wedge M(P + R) \neq M(Q + R)$.

   (a) Structure metric *TIP*.

Solution:
We'll construct two simple trees for programs P and Q that have the same *TIP* value, 0.


(a) Graph for P

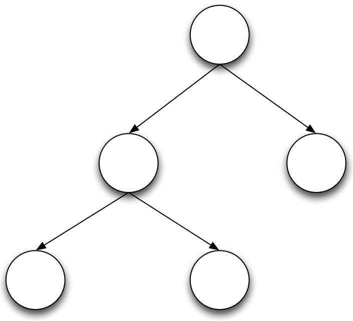

(b) Graph for Q

Figure 3: Two simple graphs with the same *TIP* value of 0.

We'll now try to find a graph for program R to prove property 6. We'll try a simple 2-node graph:

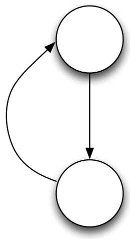
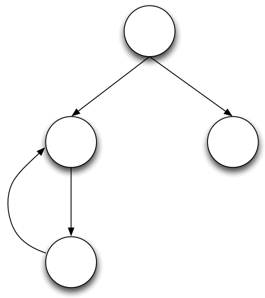


Figure 4: A simple graph for program R.

Solution: (continued)
With sequential composition on the lowest left node, $P+R$ and $Q+R$ then becomes larger graphs and not pure trees shown below in Figure 5.


(a) Graph for $P+R$

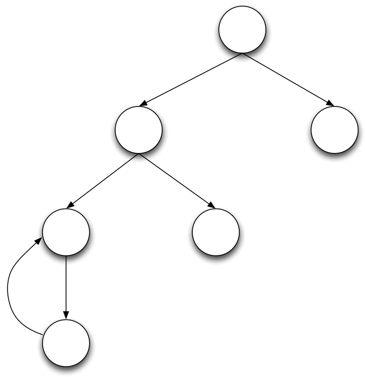

(b) Graph for $Q+R$

Figure 5: Graphs for programs $P+R$ and $Q+R$.

For the two graphs, their respective *TIP* values are:

$$TIP(P+R) = \frac{2\times(4-4+1)}{(4-2)\times(4-1)} = \frac{1}{3}$$
$$TIP(Q+R) = \frac{2\times(6-6+1)}{(6-2)\times(6-1)} = \frac{1}{10}$$

Apparently they're not equal, hence property 6 holds for *TIP*.

(b) Testing metric C_0.

Solution:
Let P be a program with 10 lines of code with 100% statement coverage. Let Q be a program with 5 lines of code also with 100% coverage. In other words, $C_0(P) = C_0(Q)$.
Let R be a program with 10 lines of code with 0% coverage. The sequential composition of R with P and Q then have coverage values as follows:

$$C_0(P+R) = \frac{10+0}{10+10} = 50\%$$
$$C_0(Q+R) = \frac{5+0}{5+10} = 33.3\%$$

They're obviously different. Hence property 6 holds for C_0.

(c) OO metric *RFC*.

Solution:
As the metric *RFC* is an object-oriented metric, we'll make programs P, Q and R classes and $+$ the inheritance operator (subclass). Then $P + R$ is R with P being a super class, similar for $Q + R$.
We'll define three Java classes to represent P and Q, as below.

P:
```
public class A {
  void a() {}
}
```

Q:
```
public class B {
  void b() {}
}
```

R:
```
public class C {
  void b() {}
}
```

Then, $P + R$ and $Q + R$ become

$P + R$:
```
public class C extends A {
  void b() {}
}
```

$Q + R$:
```
public class C extends B {
  void b() {}
}
```

By the definition of *RFC*, it counts all methods invocable in this class and all its super classes. Hence, the metric value for $P + R$ is 2 (`void a()` in `A` and `void b()` in `B`). The metric value for $Q + R$ is 1, since classes `B` and `C` have the same method declaration (`void b()`).
As a result, property 6 holds for *RFC*.

Tutorial 10
System & Object-oriented Testing

Week 10, 2023

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

1. Assume you are given the task of develop a project *ProList*, with the goal to create a simple Web application for proposal management, dealing with `User`s, `Proposal`s and `AuditLog`s.

   Assume the handling of accessing proposal details is implemented with a number of additional objects. With some simplification, the loading of the `Proposal` page is shown below in Figure 1. The component **Router** is responsible for routing HTTP requests to the appropriate *resource*s that handles them. As the name suggests, `ProposalResource` is responsible for creating pages to present proposal details. Once authentication and authorisation are determined to be 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, the resource then invokes `ProposalDAO` to obtain proposal details. If access control is not required, or that the user is denied access, `ProposalResource` constructs a page correspondingly. Finally, `ProposalResource` returns the page back to the `Router`.

   Note that the graphs within each component represent greatly simplified program graphs, with only some important control structures preserved. 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 still represent control flow. Thick edges represent transfer of control (message passing and return) between components.

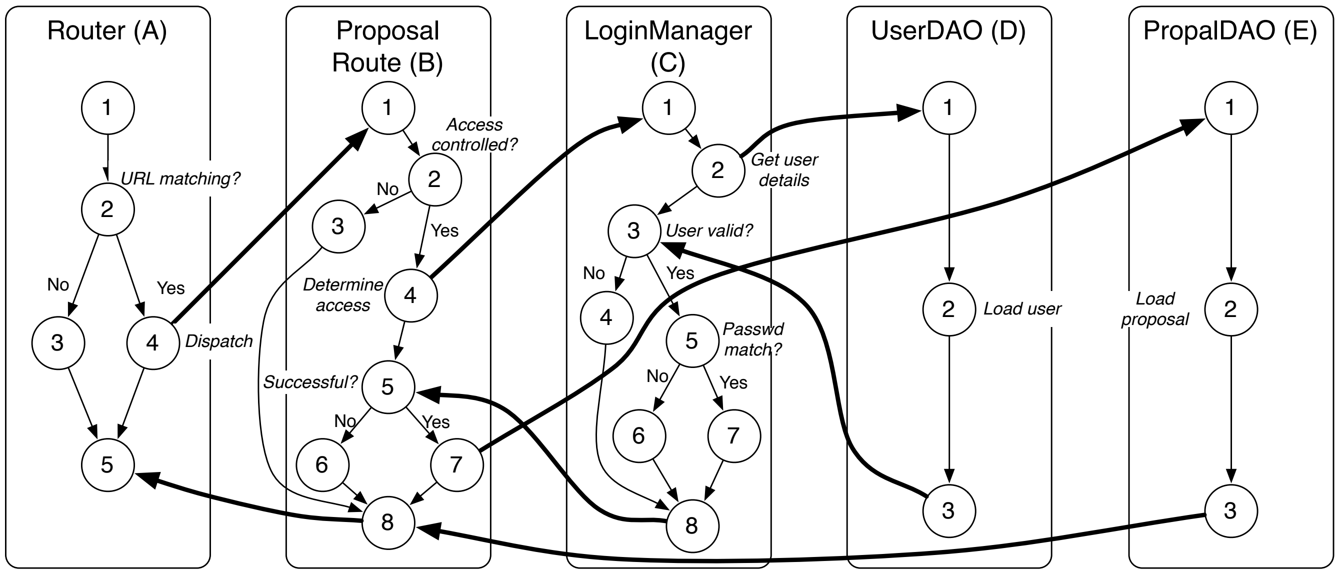


Figure 1: The interactions of some components of the *ProList* system that handle proposal presentation.

(a) Draw a sequence diagram depicting the interactions between the above objects. Note that for simplicity reasons, do not include model objects such as `User` and `Proposal`.

2. Finite state machines can be very useful in testing by capturing system functionality in terms of states, events, transitions, actions and guards.

   Suppose we want to implement a Web service that allows users to create proposals programmatically. A user will specify the user name, password and the details of the proposals to be created. The system creates the proposal upon successfully validating the provided information. A *timeout* function that puts a limit on how long the service can take to create an proposal will also be implemented. That is, the system only waits for a certain amount of time to receive user inputs. When the time is up, the system will go back to the state where it is ready to accept another request.

   Draw a system-level finite state machine depicting this scenario. If you are thinking of input validation, *do not* break it down into validation of different inputs now. Since the system is a long-running system, we will assume that there is no final state. That is, after an proposal is successfully created, the system goes back to the state where it is ready to create another proposal.

Solution:

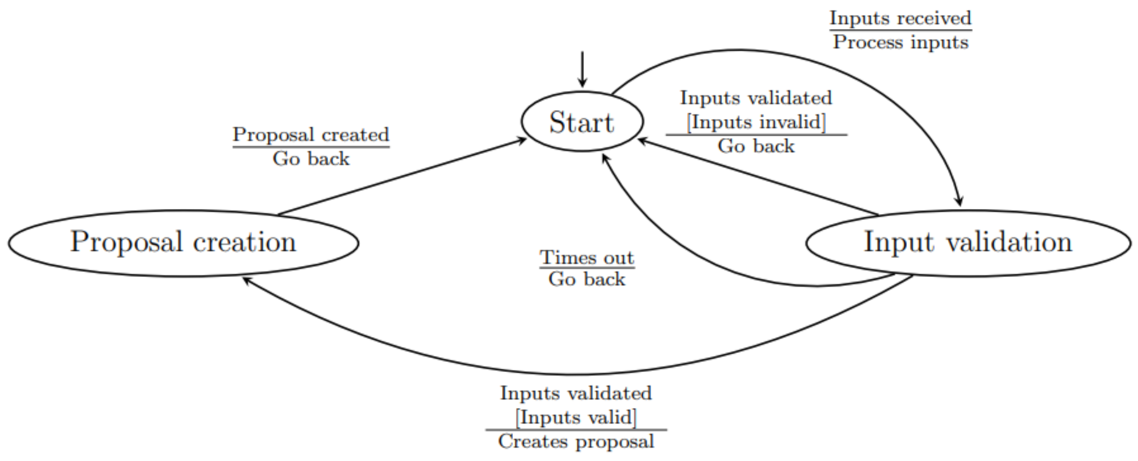


Figure 2

Object-Oriented Testing/ Metric The Java class below contains a number of methods. The method `fizzBuzz` takes a `String` representation of an integer as parameter and returns itself (as a `String`) or some other `String` values. The method `fizzBuzzRange` takes two integers and performs `FizzBuzz` on all the integers within the range.

Listing 1 below shows, on two pages, the class `Foo` in Java. Answer question 3 based on the `Foo` class.

Listing 1: The Java class `Foo`.

```java
public class Foo {

    private int min;
    private int max;

    public String fizzBuzz(String input) {
        int x = Integer.parseInt(input);

        boolean hasFizz = x % 3 == 0;
        boolean hasBuzz = x % 5 == 0;
        if (hasFizz && hasBuzz)
            return "FizzBuzz";
        else if (hasFizz)
            return "Fizz";
        else if (hasBuzz)
            return "Buzz";
        else
            return input;
    }

    public String[] fizzBuzzRange(int low, int high) {
        if (low <= 1)
            throw new IllegalArgumentException("low should be >= 1");
        else if (high > 100)
            throw new IllegalArgumentException("high should be <= 100");

        String[] result = new String[high - low + 1];
        for (int i = low; i <= high; i++)
            result[i - low] = fizzBuzz(Integer.toString(i));

        return result;
    }
```

(Continued overleaf)

```java
    private void findMinMax(int[] array) {
        min = array[0];
        max = array[0];

        for (int i : array) {
            if (min > i)
                min = i;
            else if (max < i)
                max = i;
        }
    }

    public int[] unique(int[] array) {
        findMinMax(array);

        boolean[] set = new boolean[max - min + 1];
        for (int i : array)
            set[i - min] = true;

        int size = 0;
        for (boolean i : set) {
            if (i)
                size++;
        }
        int[] result = new int[size];
        int j = 0;
        for (int i = 0; i < set.length; i++) {
            if (set[i])
                result[j++] = i + min;
        }
        return result;
    }

    public int maxOccurrences(int[] array) {
        findMinMax(array);

        int[] set = new int[max - min + 1];
        for (int i : array)
            set[i - min]++;

        max = set[0];
        for(int i : set) {
            max = max < i ? i : max;
        }
        return max;
    }
}
```

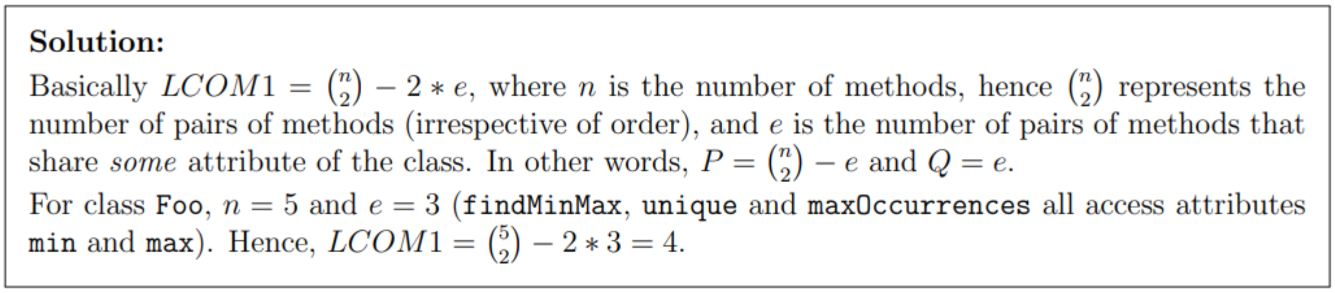
3. The quality of classes in object-oriented languages such as Java can be measured by different object-oriented metrics. Some metrics measure the *lack of cohesion* of a class based on interactions between its methods and attributes. Answer the following questions on the *cohesion* of the class `Foo`.

   (a) Specifically, the metric LCOM1 is defined as

$$LCOM1 = \begin{cases} P - Q, & \texttt{for } P > Q, \\ 0 & \texttt{otherwise} \end{cases}$$

   where for each pair of different methods (order of methods irrelevant), P is incremented by 1 if they do not access any common attribute, otherwise Q is incremented by 1. The initial values of P and Q are both 0.

   Compute the $LCOM1$ value for the class `Foo`. Include all its methods (`public` and `private`) in your calculation.



Solution:
Basically $LCOM1 = \binom{n}{2} - 2 * e$, where n is the number of methods, hence $\binom{n}{2}$ represents the number of pairs of methods (irrespective of order), and e is the number of pairs of methods that share *some* attribute of the class. In other words, $P = \binom{n}{2} - e$ and $Q = e$.
For class `Foo`, $n = 5$ and $e = 3$ (`findMinMax`, `unique` and `maxOccurrences` all access attributes `min` and `max`). Hence, $LCOM1 = \binom{5}{2} - 2 * 3 = 4$.

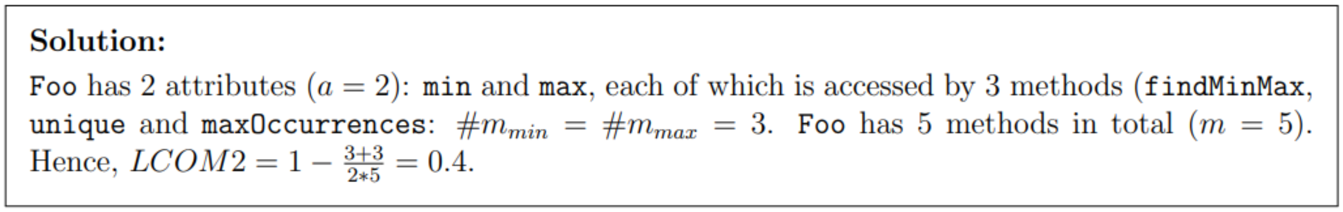
Figure 3

   (b) $LCOM2$ is another lack of cohesion metric for classes. Specifically, $LCOM2$ is defined as

$$LCOM2 = 1 - \frac{\sum\limits_{\text{for each class attribute} A} \#m_A}{m * a}$$

   where m is the number of methods, a is the number of attributes, and $\#m_A$ is the number of methods that access a particular attribute A.

   Compute the $LCOM2$ value for the class `Foo`. Include all its methods (`public` and `private`) in your calculation.



Solution:
`Foo` has 2 attributes ($a = 2$): `min` and `max`, each of which is accessed by 3 methods (`findMinMax`, `unique` and `maxOccurrences`: $\#m_{min} = \#m_{max} = 3$. `Foo` has 5 methods in total ($m = 5$). Hence, $LCOM2 = 1 - \frac{3+3}{2*5} = 0.4$.

Figure 4

FIT5171 Tutorial 3
Discrete Mathematics for Software Testing.

Week 3, 2023

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

Logics & Set Theory

1. Decide if the following predicates are true. If not, give a counter example.
   (a) $\forall A \exists B \bullet C \equiv \exists B \forall A \bullet C$, where A and B are variable declarations and C is a Boolean expression.

Solution: False.
Counterexample: let both A and B be "a human" and C be "is father". Then clearly the LHS and the RHS are not logically equivalent. More formally,

$$\forall x\colon Human \bullet \exists y\colon Human \bullet (x, y) \in isFather$$

is different from

$$\exists y\colon Human \bullet \forall x\colon Human \bullet (x, y) \in isFather$$

   (b) $\forall A, B\colon PN \bullet A \subseteq B \vee B \subseteq A$

Solution: False.
Counter example: let $A = \{0\}$, $B = \{1\}$, clearly $A \nsubseteq B$ nor $B \nsubseteq A$.

Relations & Functions

2. Give a formal definition of the binary relation R, over natural numbers, such that each x is related to y by R if and only if y is greater than the square of x but less than the square of $x + 1$.

Solution: $R \colon N \longleftrightarrow N \bullet \forall x, y \colon N \bullet x \mapsto y \in R \iff x^2 < y < (x+1)^2$

3. Let S be the set of natural numbers between 1 and 15 inclusively. Express R in Q2 as a set of ordered pairs in S.

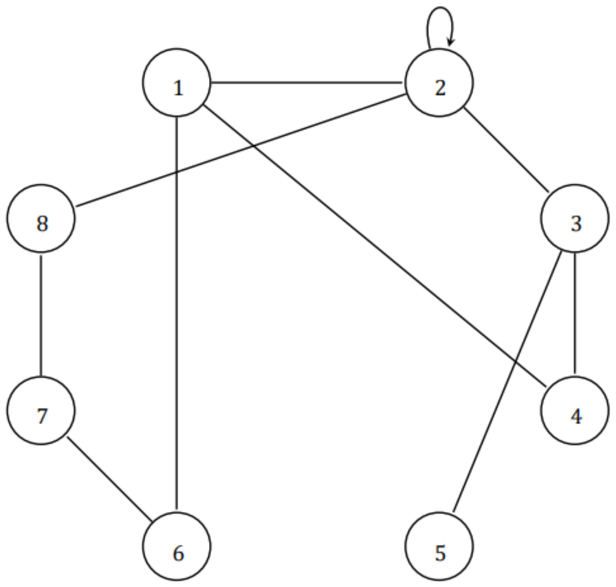
Solution: R = {(1, 2), (1, 3), (2, 5), (2, 6), (2, 7), (2, 8), (3, 10), (3, 11), (3, 12), (3, 13), (3, 14), (3, 15)}

Graph Theory

4. Given an undirected graph G with vertices $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and edges $E = \{12, 14, 16, 22, 23, 28, 34, 35, 67, 78\}$

   (a) Draw G graphically.

Solution:



   (b) Calculate the degree of each node in the graph.

Solution:

Node	1	2	3	4	5	6	7	8
Degree	3	5	3	2	1	2	2	2

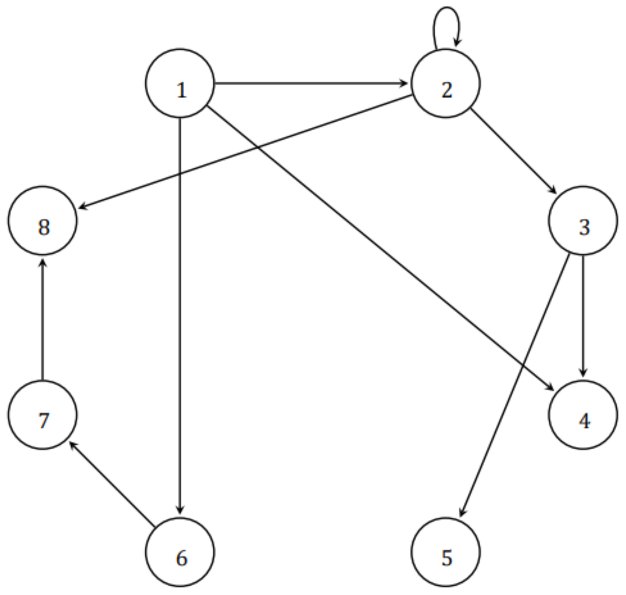
   (c) Calculate the Cyclomatic number of G.

Solution: The Cyclomatic number of of G, $V(G) = \#E - \#V + p = 10 - 8 + 1 = 3$.

5. Let's treat G in Q5 as a directed graph (edge 12 in E represents an edge from node 1 to node 2).

   (a) Draw G graphically.

Solution:



   (b) Calculate the in-degree and out-degree of each node (ignoring edge 22 for this question)

Solution:

Node	1	2	3	4	5	6	7	8
In-degree	0	1	1	2	1	1	1	2
Out-degree	3	2	2	0	0	1	1	0

   (c) Are there any source nodes or sink nodes in G?

Solution:
Source nodes: 1
Sink nodes: 4, 5, 8

   (d) Does this graph contain semi-paths? If so, identify them.

Solution: Yes, (12, 23, 16, 67), (12, 28, 16, 67), and (12, 23, 34, 16, 67, 78) are semi-paths.

(e) Identify a pair of nodes that are 0-connected, 1-connected, 2-connected, and 3-connected, if any (ignoring edge 22 again).

Solution:
0-connected: none.
1-connected: 2 & 6, 3 & 7, etc.
2-connected: 1 & 2, 6 & 7, etc.
3-connected: none.

(f) Work out the reachability matrix for G.

Solution:

	1	2	3	4	5	6	7	8
1	0	1	1	1	1	1	1	1
2	0	1	1	1	1	0	0	1
3	0	0	0	1	1	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

6. Draw a directed graph for each of the following common control constructs: (1) if-then, (2) if-then-else, (3) case switch, (4) while loop, and (5) do-while loop. As an example, the graph of sequence execution is included below in Figure 1.

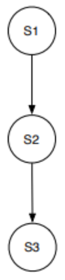


Figure 1: A directed graph showing sequential execution.

Solution: See below for the depictions for the five cases.

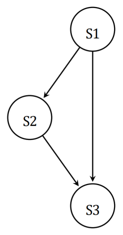


Figure 2: if-then

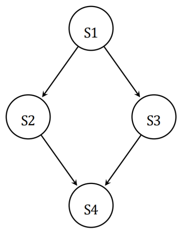


Figure 3: if-then-else

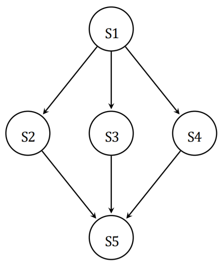


Figure 4: case switch

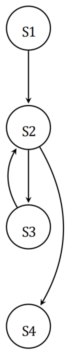


Figure 5: while loop

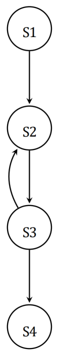


Figure 6: do-while loop

FIT5171 Tutorial 4 and 5
Blackbox Testing

Week 4, 2023

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

1. In Lecture 4 we discussed Boundary Value Testing (BVT) and the (minimum) number of test cases for the "normal" (non-robust) version of BVT for n variables ($4n + 1$). In this question, work out a formula for the number of test cases for each of the following cases and briefly explain why.

   (a) The robust BVT (with additional values $min-$ and $max+$ for each variable).

Solution: $6n + 1$, since we have 2 more test cases for each variable.

   (b) Weak normal equivalence class testing.

Solution: Let C_x denote the equivalence classes of valid values for variable x (C_x is a set of equivalence classes). Then the number of test cases is $max(\#C_{x_i})$, for x_i ranging over all variables. That is, the maximum number of equivalence classes for all variables.

   (c) Weak robust equivalence class testing.

Solution: We assume the same settings as in the previous question. Let I_{x_i} denote the equivalence classes of invalid values for variable x_i. Let n denote the number of variables. the number of tests is $max(\#C_{x_i}) + \sum_{i=1}^{n}(\#I_{x_i})$, for x_i ranging over all variables.

Basically, we include also the total number of test cases in the invalid areas for each variable. Note that cardinality of I_{x_i} may not always be 2 since there may be gaps between valid equivalence classes.

2. In the last lecture we showed a triangle example to demonstrate test case generation for BVT (slide 11). In the example each of the three variables a, b and c is the length of a side from the range $[1, 200]$. Come up with test cases for weak normal equivalence class testing that cover the same expected outputs (isosceles, equilateral, scalene, not a triangle).

Solution: A simple/naïve solution would be to take those 4 expected outputs and use them directly to form equivalence classes:

$$R1 = \{(a, b, c) \mid \text{the triangle with sides } a,\ b \text{ and } c \text{ is isosceles}\}$$

$$R2 = \{(a, b, c) \mid \text{the triangle with sides } a,\ b \text{ and } c \text{ is equilateral}\}$$

$$R3 = \{(a, b, c) \mid \text{the triangle with sides } a,\ b \text{ and } c \text{ is scalene}\}$$

$$R4 = \{(a, b, c) \mid \text{sides } a,\ b \text{ and } c \text{ do not form a triangle}\}$$

Then we can take an arbitrary test case from each of these four cases:

Test case	a	b	c	**Expected output**
WN1	20	20	30	Isosceles
WN2	50	50	50	Equilateral
WN3	30	40	50	Scalene
WN4	30	40	100	Not a triangle

A more insightful solution requires the understanding of the domain. In this case, working out the relationship between the three sides a, b and c. If we take symmetry into consideration, we could come up with the following equivalence classes:

$$D1 = \{(a, b, c) \mid a = b = c\}$$

$$D2 = \{(a, b, c) \mid a = b \wedge a \neq c \wedge c < a + b\}$$

$$D3 = \{(a, b, c) \mid a \neq b \wedge a \neq c \wedge b \neq c \wedge a < b + c \wedge b < a + c \wedge c < a + b\}$$

$$D4 = \{(a, b, c) \mid a = b + c\}$$

$$D5 = \{(a, b, c) \mid a > b + c\}$$

Then base on these 5 classes we can work out the test cases more easily.

3. For the triangle problem above, come up with a decision table for testing.

Solution: The following is one such decision table. Note that for simplicity the value range $[1, 200]$ isn't considered below.

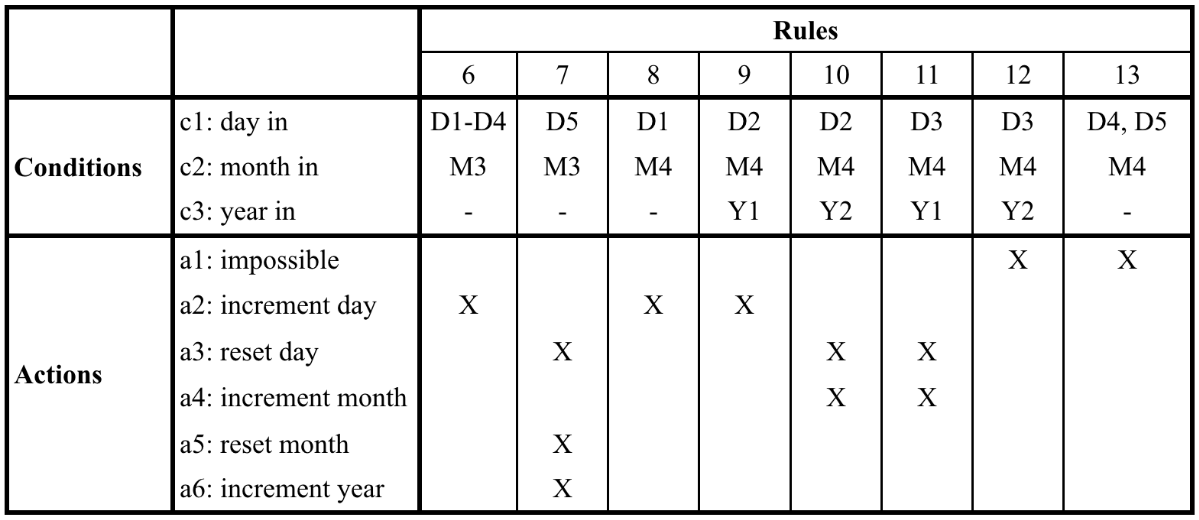
		Rules								
Conditions	c1: a,b,c form a triangle?	F	T	T	T	T	T	T	T	T
	c2: $a = b$?	-	T	T	T	T	F	F	F	F
	c3: $a = c$?	-	T	T	F	F	T	T	F	F
	c4: $b = c$?	-	T	F	T	F	T	F	T	F
Actions	a1: not a triangle	X								
	a2: scalene									X
	a3: isoscles					X		X	X	
	a4: equilateral		X							
	a5: impossible			X	X		X			

Note that if we want to be more specific about the conditions (for example, enumerating conditions when a, b and c do not form a triangle), we could come up with decision tables with more entries.

4. Under the tutorial resources, you will find a pdf document which includes the **NextDate method**, which, given a day, a month, and a year, returns the date of the following day.

   (a) Complete the decision table on slide 37 for NextDate by filling in the missing conditions and associated actions.

Solution:
The following table completes the decision table. Note: Rule 7 is already given in lecture slides.



		Rules							
		6	7	8	9	10	11	12	13
Conditions	c1: day in	D1-D4	D5	D1	D2	D2	D3	D3	D4, D5
	c2: month in	M3	M3	M4	M4	M4	M4	M4	M4
	c3: year in	-	-	-	Y1	Y2	Y1	Y2	-
Actions	a1: impossible							X	X
	a2: increment day	X		X	X				
	a3: reset day		X			X	X		
	a4: increment month					X	X		
	a5: reset month		X						
	a6: increment year		X						

   (b) How many test cases are needed to completely cover the entire decision table?

Solution:
13 test cases.

(c) For the NextDate method, assuming the year variable ranges over [1812, 2016], how many test cases are needed for strong, normal boundary value testing?

Solution:

Since BVT doesn't understand the semantics of variables, the boundary values for the variables are: day: 1 and 31, month: 1 and 12; and year: 1812 and 2016.

NOTE: the below answer (19) is for **weak, robust** boundary value testing! The test cases are given in the table below.

Test case	day	month	year	Expected output
1	0	6	1912	error
2	1	6	1912	2/6/1912
3	2	6	1912	3/6/1912
4	15	6	1912	16/6/1912
5	30	6	1912	1/7/1912
6	31	6	1912	error
7	32	6	1912	error
8	15	0	1912	error
9	15	1	1912	16/1/1912
10	15	2	1912	16/2/1912
11	15	11	1912	16/11/1912
12	15	12	1912	16/12/1912
13	15	13	1912	error
14	15	6	1811	error
15	15	6	1812	16/6/1812
16	15	6	1813	16/6/1813
17	15	6	2015	16/6/1815
18	15	6	2016	16/6/1816
19	15	6	2017	error

As can be seen, there are 19 test cases.

For **strong, normal** boundary value testing, each variable can take on (min, min+, nom, max-, max) values (normal), and all variables are free to take any of the above values (strong). Hence, the total number of test cases is 5^n for n variables. For our example, it is $5^3 = 125$ test cases.

(d) Compare and comment on the effort and effectiveness of BVT and decision table testing.

Solution:

Decision table testing requires significantly more effort in identifying test cases. However it is also more effective in that it covers more scenarios/corner cases than BVT in fewer test cases.