CSCE 747 - Unit Testing Laboratory

You have been hired to test our new calendar app! Congrats!(?)

This program allows users to book meetings, adding those meetings to calendars maintained for rooms and employees. It will actively prevent multiple bookings, and will manage the busy and open status for employees and rooms.

The system enables the following high-level functions:

- Booking a meeting
- Booking vacation time
- Checking availability for a room
- Checking availability for a person
- Printing the agenda for a room
- Printing the agenda for a person

Normally, actions are conducted through the driver provided by the main method in the PlannerInterface class. As a tester, you - of course - have full access to the source code to employ in testing the system.

Documentation for the code is attached.

You are to do the following:

1. Spend 10-15 minutes formulating a test plan.
    a. Given the above features and the code documentation, plan out a series of test cases to ensure that these features can be performed without error.
    b. Think about what the “testable units” are.
        i. Your tests may use any of the classes in the system, and may be at the method, class, or system level.
    c. Make sure you think about both the normal execution and illegal inputs and actions that could be performed.
        i. Think of as many things that could go wrong as you can! For instance, you will probably be able to add a normal meeting, but can you add a meeting for February 35th? Try it out.
2. Write tests in the jUnit framework.
    a. If a test is supposed to cause an exception to be thrown. Make sure you check for that exception.
    b. Make sure that your expected output is detailed enough to ensure that - if something is supposed to fail - that it fails for the correct reasons.
3. Turn in your test plan and unit tests.

Test Plan

jUnit Basics

JUnit is a Java-based toolkit for writing executable tests.

- Choose a target from the code base.

```
public class Calculator {
  public int evaluate (String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

- Write a “testing class” containing a series of unit tests centered around testing that target.
    - Each test is denoted **@test**

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
    calculator = null;
  }
}
```

```
@Test
public void test<MethodName><TestingContext>() {
        //Define Inputs
        try{ //Try to get output.
        }catch(Exception error){
                fail("Why did it fail?");
        }
        //Compare expected and actual values through assertions or through if statements/fails
}
```

- @Before annotation defines a common test initialization method:

```
@Before
public void setUp() throws Exception
{
        this.registration = new Registration();
        this.registration.setUser(“ggay”);
}
```

- @After annotation defines a common test tear down method:

```
@After
public void tearDown() throws Exception
{
        this.registration.logout();
        this.registration = null;
}
```

- @BeforeClass defines initialization to take place before any tests are run.

```
@BeforeClass
  public static void setUpClass() {
        myManagedResource = new
                ManagedResource();
  }
```

- @AfterClass defines tear down after all tests are done.

```
 @AfterClass
  public static void tearDownClass() throws IOException {
    myManagedResource.close();
    myManagedResource = null;
  }
```

- Assertions are a "language" of testing - constraints that you place on the output.
    - assertEquals, assertArrayEquals
        - Compares two items for equality.
        - For user-defined classes, relies on `.equals` method.
            - Compare field-by-field
            - `assertEquals(studentA.getName(), studentB.getName())` rather than `assertEquals(studentA, studentB)`

```
@Test
public void testAssertEquals() {
   assertEquals("failure - strings are not equal", "text", "text");
}
```

- assertArrayEquals compares arrays of items.

```
@Test
public void testAssertArrayEquals() {
        byte[] expected = "trial".getBytes();
        byte[] actual = "trial".getBytes();
        assertArrayEquals("failure - byte arrays
                not same", expected, actual);
}
```

- assertFalse, assertTrue
  - Take in a string and a boolean expression.
  - Evaluates the expression and issues pass/fail based on outcome.
  - Used to check conformance of solution to expected properties.

```
@Test
public void testAssertFalse() {
   assertFalse("failure - should be false", (getGrade(studentA,
“CSCE747”).equals(“A”));
}
@Test
public void testAssertTrue() {
        assertTrue("failure - should be true", (getOwed(studentA) > 0));
}
```

- assertNull, assertNotNull
  - Take in an object and checks whether it is null/not null.
  - Can be used to help diagnose and void null pointer exceptions.

```
@Test
public void testAssertNotNull() {
   assertNotNull("should not be null", new Object());
}

@Test
public void testAssertNull() {
   assertNull("should be null", null);
}
```

- assertSame,assertNotSame
  - Checks whether two objects are clones.
  - Are these variables aliases for the same object?
    - assertEquals uses .equals().
    - assertSame uses ==

```
@Test
public void testAssertNotSame() {
   assertNotSame("should not be same Object", studentA, new Object());
}

@Test
public void testAssertSame() {
    Student studentB = studentA;
   assertSame("should be same", studentA, studentB);
}
```