COMS W4156 Advanced Software Engineering (ASE)

September 20, 2022

Agenda

1. Team formation
2. JUnit demo
3. finish Unit Testing

Team Formation

Please speak up if you have not found a team or have a team with any size different from 4 or 5 members

4 members is strongly preferred since it nicely supports pair programming (covered later in course)

Resubmit assignment if your team changes (courseworks will accept until 11:59 tonight, I'll extend if needed)

Agenda

1. Team formation
2. JUnit demo
3. finish Unit Testing

JUnit Demo: Ashutosh Sameer

Learning More About JUnit

There are numerous tutorials (e.g., SoftwareTestingHelp) and lots of other documentation about JUnit online

Posting useful (non-obvious) tips to the class on Ed Discussion counts towards your Participation grade. You can post anonymously wrt other students, but do not make your post anonymous to teaching staff because then we cannot give you credit!

Agenda

1. Team formation
2. JUnit demo
3. finish Unit Testing

Running the Tests

After choosing a collection of valid and invalid inputs for one or more units under test, how do we run the tests?

Let's say we have a method that takes all its inputs as parameters and provides one output as its return value: write code that calls the method with the input parameters, receives the output, and then checks the output

- ➢ The "manual" version is create a tiny main() program just to call the method under test.
- ➢ Don't do it! 💩 Instead use a unit testing framework

If the method reads/writes local shared state, such as class variables, an object and its instance variables, or an in-memory data structure like a stack or a tree: need to arrange for inputs from those sources

- ➢ The "manual" version is again create a tiny main() program, which now calls constructors and initializes fields, and possibly calls other methods in the same class, before calling the method under test.
- ➢ Don't do it! 💩 Instead use a unit testing framework

What is “Local” Shared State?

I made up this term to mean state shared only among units that are part of the same module
(e.g., methods in the same class)

In an object-oriented language, this includes instance variables (object fields) and class variables (static fields)

“Global” shared state is application state shared with other code outside the module and environment state shared even outside the program, such as files, databases, devices, etc.

There may also be “global” inputs/outputs from/to UI, network, or accessing the underlying platform (e.g., time of day, random number generator)

“Global” shared state and inputs/outputs are usually faked during unit testing (mocking)

Unit Testing Frameworks

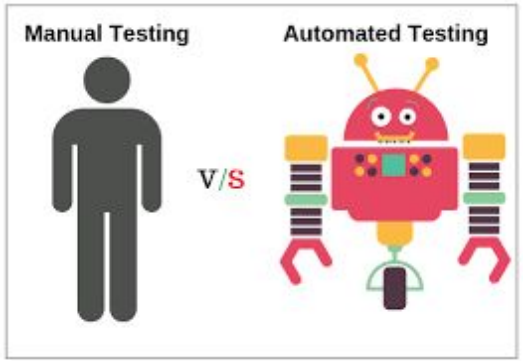
Instead of “manual” unit testing, use some unit testing framework for your programming language, e.g., googletest for C++, junit for Java

Unit testing frameworks always provide “test runners” and usually provide facilities for writing unit tests like assertions and fixtures

Unit testing frameworks often play nicely with mocking frameworks, for faking “global” inputs, outputs, and shared state

Automated testing reduces human tedium and error,
and is reproducible and faster so more likely to be done frequently

➢ Automated testing refers to *running* the tests, not *writing* tests (although templates may be provided), developers still write tests



Test Runner

Executes a test suite in some order and reports results

Usually does not stop when a test fails - instead runs all the tests and reports all the results

Ordering might be built-in or specified by the tester - why does test order matter? (test ordering issues covered later in semester)

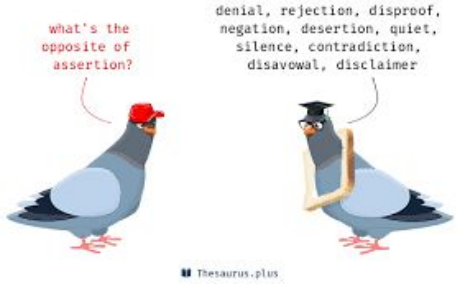
Chart	See children	Build Number ⇒ Package-Class-Testmethod names ⇓	16	15	14	13	12	11	10	9	8	7	6	5
☐	⊖	org.common.samplea	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
☐	⊖	SampleATest	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
☐		testA	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
☐		testB	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
☐		testC	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
☐		testD	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
☐	⊕	org.common.sampleb	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
☐	⊖	org.common.samplec	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
☐	⊖	SampleDTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
☐		testA	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
☐		testB	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	N/A
☐		testC	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	N/A
☐		testD	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A

Facilities for Writing Unit Tests

Assertions = notation for checking the results (outputs and side-effects) of executing a test. Usually supports checking for true/false, checking whether an exception has been raised, and comparing a result to a known value

Test fixture = *setup* code to execute before running one or more tests, *teardown* code to run after those tests. Setup can create objects, open files or network/database connections (often involves mocks), etc. Teardown destroys objects, closes files and connections, etc.

Assertions



- **Arrange**: set up the context and prepare a bunch of objects to run the unit under test

- **Act**: call the unit under test

- **Assert**: check that outputs and side effects of the unit under test are as expected

arrange/act/assert or given/when/then

Assertions express requirements that the unit under test is expected to meet

What Is Wrong With These Assertions?

```
@Test
public void getEmployeesGreaterThan25() {

  // arrange
  Employee mark = new Employee ("Mark", 19);
  Employee nina = new Employee ("Nina", 26);
  Employee jules = new Employee ("Jules", 31);
 clearDatabase();
  insertEmployee(mark);
  insertEmployee(nina);
  insertEmployee(jules);
  Predicate<Employee> constraint = e -> e.getAge() > 25;

  // act
  List<Employee> result = getEmployees(constraint);

  // assert
  assertEquals(result.size(), 2);
  assertEquals(result.get(0).getName(), "Nina");
  assertEquals(result.get(0).getAge(), 26);
  assertEquals(result.get(1).getName(), "Jules");
  assertEquals(result.get(1).getAge(), 31);
}
```

```
for (Employee e : result) {
  assertTrue(e.getAge() > 25);
}
```

```
boolean first = result.get(0).getAge() > 25;
boolean second = result.get(1).getAge() > 25;
assertTrue(first && second);
```

```
assertTrue(result.get(0).getAge() > 25);
assertTrue(result.get(1).getAge() > 25);
```

Good Assertions

Tests are software too, they may have bugs and they need to be robust, maintainable and effective

Good assertions check what is precisely requested from the code under test, as opposed to checking an overly precise or an overly loose condition

Don’t write assertions which check conditions that you could split to multiple assert statements, always check the simplest condition possible

Write assertions that check requirements, not implementation details of the unit under test. You do not want to update a test just because the unit under test has changed for reasons unrelated to the test

Fixtures

A test fixture is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. Typically applied to a test class with multiple test cases, not just a single test case

- Preparation of input data and setup/creation of fake or mock objects
- Loading a database with a specific, known set of data
- Copying a specific known set of files

Test fixtures also clean up after themselves!

What Does This Do?

```
import java.io.Closeable;
import java.io.IOException;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class TestFixturesExample {
  static class ExpensiveManagedResource implements Closeable {
    @Override
    public void close() throws IOException {}
  }

  static class ManagedResource implements Closeable {
    @Override
    public void close() throws IOException {}
  }

  @BeforeClass
  public static void setUpClass() {
    System.out.println("@BeforeClass setUpClass");
    myExpensiveManagedResource = new ExpensiveManagedResource();
  }

  @AfterClass
  public static void tearDownClass() throws IOException {
    System.out.println("@AfterClass tearDownClass");
    myExpensiveManagedResource.close();
    myExpensiveManagedResource = null;
  }
```

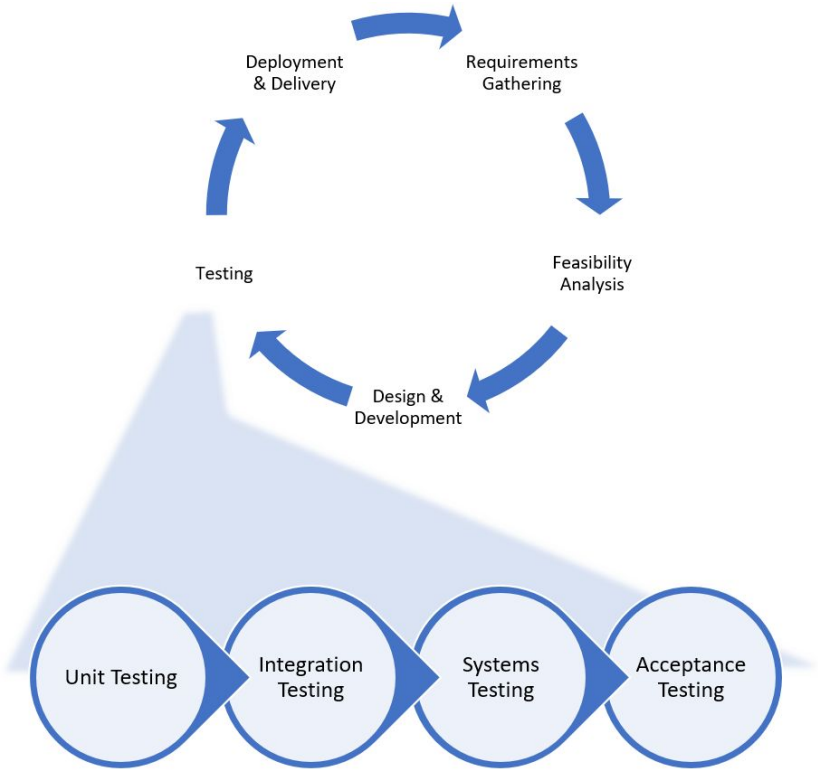
```
  private ManagedResource myManagedResource;
  private static ExpensiveManagedResource myExpensiveManagedResource;

  private void println(String string) {
    System.out.println(string);
  }

  @Before
  public void setUp() {
    this.println("@Before setUp");
    this.myManagedResource = new ManagedResource();
  }

  @After
  public void tearDown() throws IOException {
    this.println("@After tearDown");
    this.myManagedResource.close();
    this.myManagedResource = null;
  }

  @Test
  public void test1() {
    this.println("@Test test1()");
  }

  @Test
  public void test2() {
    this.println("@Test test2()");
  }
}
```

What Does This Do?

```
import java.io.Closeable;
import java.io.IOException;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class TestFixturesExample {
  static class ExpensiveManagedResource implements Closeable {
    @Override
    public void close() throws IOException {}
  }

  static class ManagedResource implements Closeable {
    @Override
    public void close() throws IOException {}
  }

  @BeforeClass
  public static void setUpClass() {
    System.out.println("@BeforeClass setUpClass");
    myExpensiveManagedResource = new ExpensiveManagedResource();
  }

  @AfterClass
  public static void tearDownClass() throws IOException {
    System.out.println("@AfterClass tearDownClass");
    myExpensiveManagedResource.close();
    myExpensiveManagedResource = null;
  }

  private ManagedResource myManagedResource;
  private static ExpensiveManagedResource myExpensiveManagedResource;

  private void println(String string) {
    System.out.println(string);
  }

  @Before
  public void setUp() {
    this.println("@Before setUp");
    this.myManagedResource = new ManagedResource();
  }

  @After
  public void tearDown() throws IOException {
    this.println("@After tearDown");
    this.myManagedResource.close();
    this.myManagedResource = null;
  }

  @Test
  public void test1() {
    this.println("@Test test1()");
  }

  @Test
  public void test2() {
    this.println("@Test test2()");
  }
}
```

```
@BeforeClass setUpClass
@Before setUp
@Test test2()
@After tearDown
@Before setUp
@Test test1()
@After tearDown
@AfterClass tearDownClass
```

Lots More Testing Later



Equivalence partitions and Boundary analysis

Mocking

API Testing

Integration and System Testing

Coverage

Test Suite Ordering

Test Oracles

More…

Next Class

Software Reuse

APIs

Upcoming Assignments

Preliminary Project Proposal: due September 26, next Monday - Tentative mentors will be assigned after we read your preliminary proposals

Revised Project Proposal: due October 3 - meet with your Mentor to discuss your preliminary proposal *before* submitting revision

Ask Me Anything