COMS W4156 Advanced Software Engineering (ASE)

September 13, 2022

Agenda

1. Maven demo
2. Unit Testing

Maven Demo: Nihar Maheshwari

Learning More About Maven

There are numerous tutorials (e.g., Baeldung) and lots of other documentation about Maven 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!

Build for Java

Maven

Ant

Gradle

sbt

Build for C++

Ninja

GNU Make

CMake

Bazel

Learning More About Build and Package Management Tools

You will need to learn some build/package tool for your team project

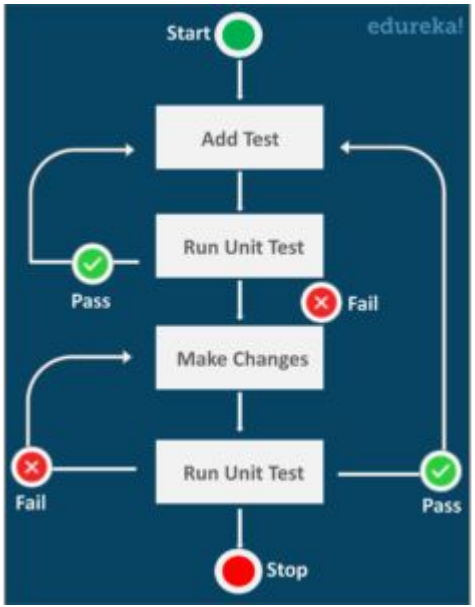
Software engineers spent an immense amount of time consulting documentation

Read [Today Was a Good Day: The Daily Life of Software Developers]

Agenda

1. Maven demo
2. Unit Testing

Unit Testing



What is a “unit”? The smallest piece of code that can be logically isolated for execution

Depends on the programming language, but generally a method, function, procedure, subroutine (not a statement or expression)

Small interconnected sets of such units, such as classes, modules, files, are typically tested during unit testing as well as individual units

Interfaces between these sets are tested later, known as “integration testing” (covered later in semester)

Example Unit Tests

```
public class TestBadCode {

    public int calculateFoo(int x, int y, boolean increment){
        if(increment)
            x++;
            x *=2;
        x += y;
        return x;
    }

    @Test
    public void test() {
        assertEquals(calculateFoo(3, 5, true), 13);
        assertEquals(calculateFoo(3, 5, false), 8);
    }

}
```

calculateFoo is the “unit under test”

This is one test class consisting of two test cases, with inputs { 3, 5, true } and { 3, 5, false }

Which of these tests pass?

Note this is Java code, not Python

Unit Testing 2

When does unit testing happen? Ideally all changed code is unit-tested before commit (e.g., git pre-commit hook), and the commit aborted if any tests fail

Unit testing is nearly always automated by a unit testing framework, where we create a set of unit tests, push a button, and wait

Choosing Inputs for Unit Tests

Baseline - one normal valid input (or set of inputs), one abnormal valid input, one invalid input for every unit (or almost every unit, maybe not getters and setters)

Better - Baseline plus one input in middle of every equivalence partition, 2-3 inputs at every boundary between equivalence partitions

Best? - Better plus “cover” every branch plus every loop 0, 1, 2, and many/max (if known) times around the loop

Equivalence partitions, boundary analysis, and coverage discussed later in course

How to Choose Baseline Test Inputs

Typical valid inputs	Enter age: _____25_____
Atypical valid inputs	Enter age: _____114_____
Invalid input value	Enter age: _____9000_____
Invalid input format	Enter age: _____3.14159_____

Invalid Input Formats

String inputs with valid vs. invalid *formatting*

- Printing vs. non-printing characters
- Upper vs. lower case
- Space, tab, newline characters (whitespace)
- Any characters with special meaning to the application (or to the implementation, e.g., programming language, SQL)
- Null/empty string
- Any string above maximum buffer size (the buffer is the data storage for reading the string input

➢ The user will not know these rules unless the software tells them

Where do Invalid Inputs come from?

Users

Network

Devices

Databases

Files

...

Example: Choosing Test Inputs

Consider a simple calendar program where the user can enter an arbitrary date and the program returns the corresponding day of the week. The user is prompted to enter the month, the day, and the year in separate fields.

The calendar represents months as integers 1 to 12, days as integers 1 to 31, and years as integers 1 to 9999 (AD).

How should we choose test cases for this program?

Choosing Test Inputs

Is there anything special about the days 29, 30 or 31 that we should test?

Should we test the month, day, or year 0?

What about testing with negative integers?

Should we test with month 13? What about day 32?

Should we test with non-numeric input?

➢ Note more than one atypical valid input and more than one kind of invalid input

Test ***Every*** Entry Point

Units (methods, functions) typically have parameters or arguments provided in call

But parameters are not the only entry points!

- ➔ Global variables
- ➔ Return values and side-effects from API calls are also inputs
- ➔ Data entered by user via GUI or CLI are also inputs
- ➔ Data retrieved from databases, files, devices, etc. are also inputs

When testing units *in isolation*, need some way to specify the inputs for these other kinds of entry points - test doubles (mocks) will be discussed next week

Running the Tests

Now that we have chosen some 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

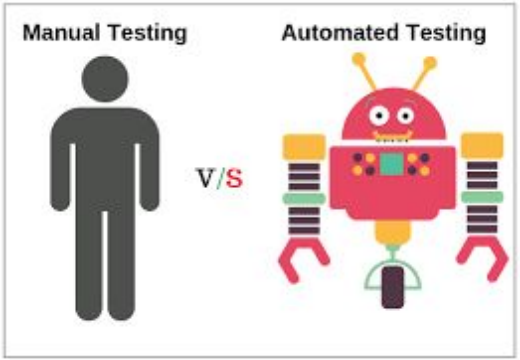
Instead of “manual” unit testing, use some unit testing framework for your programming language, e.g., googletest for C++, junit for Java

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



Coming Soon

Thursday:

Guest Speaker!

if time permits: finish unit testing

next Tuesday:

JUnit demo

finish unit testing and/or APIs

Upcoming Assignments

Team Formation: due September 19, next Monday!

Preliminary Project Proposal: due September 26, one week later - Mentors will be assigned after we receive your preliminary proposals

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

Ask Me Anything