Software Engineering 301:
Software Analysis and Design

Testing:
Basics and automation

Agenda

- Basic concepts
- Test plans
- Automated testing

“Does the software work correctly?”

- What does this phrase mean?
- Many different senses of “correctness” in software
  - Is the needed functionality present?
    - e.g., Mozilla Firefox doesn’t serve as a UML drawing package
  - Do the pieces of the system meet their specification?
    - e.g., a method “plus(a: int, b: int): int” that actually computes product
  - Do the pieces of the system fit together properly?
    - e.g., assumptions about Imperial vs. metric in Mars Climate Orbiter
  - Is the interface usable?
    - e.g., dig through 20 cascading menus to save a file?!?

Determining correctness

- What means do we have at our disposal?

1. Confirming with the users/client that:
   - the functionality is adequate
   - the interface is usable
   - non-functional requirements are met
   - Can be done before, during, and after development
2. "Static analysis" of the source
   - *DON'T CONFUSE THIS WITH ANALYSIS OF THE REQUIREMENTS!*
   - compilers do some of this
   - formal methods: prove that the source works
3. Execution of the program
   - demonstrate that it works: harder than you think!

Determining correctness

- Validation
  - Are we building the desired functionality?
  - static validation: “Here is a sketch of the interface. Does it look reasonable?”
  - dynamic validation: “Hey, there’s no means of saving and loading my work-in-progress!”
- Verification
  - Are we building the functionality correctly?
  - static verification:
    - “Look right here in the code: it says ‘*’ when it should say ‘+’” (*reviews and inspections*)
    - “We’ve mathematically proven that the code meets its specification” (*formal verification*)
  - dynamic verification: *testing*

Terminology

- A *test case* defines specific inputs and the *expected result*
  - Inputs: data passed as parameters, interaction with widgets …
  - Expected result: return a value, throw an exception, cause a different set of widgets to appear …
  - "1" and "2" are specific inputs; "some integer" is not
- A test case might require that set-up be done first
  - Objects initialized, database populated, interface navigated in a particular way …
  - Often called *setup*
- … and have resources cleaned up after
  - Memory freed, files deleted, …
  - Often called *teardown*
- A *test suite* defines a set of test cases

Terminology

- In performing a *test* (i.e., executing a test case), an *actual result* will occur
  - difference from expected result is a *failure*
  - root cause of a failure is a *fault* (i.e., bug)
- During execution of a program, the code containing a fault will eventually be executed
  - this fault may not immediately cause a failure
  - however, the system enters an *erroneous state* (or an *error* has occurred) that will eventually cause a failure
- Separation between error and failure can make detection (and correction) of the fault difficult

Expected vs. actual result

- Consider this code sample

```
/**
 * Return the sum of the input integers
 */
public int plus(int a, int b) {
  return a * b;
}
```

- What is the expected result of “plus(1, 2)”? ***… 3***
  - This assumes that the Javadoc comment is the correct specification for the method!
- But we can see that the method will return 1 * 2 = 2
  - This is the actual result that signals the presence of a fault

Exhaustive testing

- Again consider the “plus” method
  - How can we be sure that it returns the sum correctly for all pairs of inputs?
  - Why don’t we just try them all?
    - in Java, an int can take 2^{32} different values
    - 2 input vars => total combinations = $2^{32} \times 2^{32} = 2^{64}$
    - say, one test requires 10^{-6} s ≈ 2^{-20} s
    - all combinations require 2^{-20} s x $2^{64} = 2^{44}$ s ≈ 22 thousand years!

Exhaustive testing

- In a real system, number of combinations of possible inputs is generally much, much greater than this
- Exhaustive testing is impractical

Test case selection

- If we can’t try every case, can we just try a few?
  - Which ones?
  - It is hard to choose good test cases, but important
- Can we guarantee that we haven’t missed the important cases that will point out bugs?

NO!!!

- Key point: Testing can only prove the *presence* of faults, not their *absence*

Test case selection

- What makes a test case, a good test case?
  - If it demonstrates the presence of a bug, and helps to localize it
- You need to choose where to put your efforts, since you always have limited resources
- What's a cost-effective way of creating test cases?
  - Pick the ones that:
    - are most likely to find bugs, or
    - the situations that would be most serious if a bug were to occur there

Test case selection

- We will start looking at particularly strategies for test case selection in a future class

Other issues

- How do you make sure that problems, once detected, are fixed?
- How do you make sure that one repair does not cause failures that were not present previously?
- How can you make sure that tests are done correctly and repeatably?
- Answers:
  - Bug tracking, regression testing, automated testing
- Are some programs easier to test than others?

Agenda

- ~~Basic concepts~~
- Test plans
- Automated testing

When to test?

- Obviously, you can't **perform** a test until you have an implementation
- However, you can **plan** a test beforehand
  - How: look at descriptions/specifications to…
    - understand what functionality matters
    - determine what inputs to use and what results to expect from these
  - Why: "bug blindness"
    - you will assume that your code is correct
    - test cases written after you have coded are unlikely to notice problems, because your tests will contain the same, false assumptions

Test plans

- Level of formality depends on context
- A test plan describes a set of test cases (and when to run them)
- In this course, a test case should be described as:
  - Name
  - Description of purpose
  - Precise, concrete instructions for performing it
    - Getting the system into the necessary initial state (“set-up”)
    - The steps constituting the real test
    - If automated, the code will provide this
  - **Expected result**
  - How to clean up afterwards (if necessary)
    - Often called “tear-down”
    - Matters if you need to run a sequence
- A means for recording the results will often be involved

System test plans

- At a system level, you should be interested in testing the functionality in a general way
  - Use case model should be clear about setup (preconditions), event sequences, expected values, …

Remember: Vending machine user stories

- As a customer, I want to insert coins to be able to pay for my item.
- As a customer, I want to select my item.
- As a customer, I want to know the cost of an item.
- As a technician, I want to set the prices of items.
- As a technician, I want to open the machine to service it, without injuring anyone.
- … [others too]

Example: Vending machine

- Test 1: Insert quarter happy path
  - Purpose: Check that vending machine responds correctly to a quarter being entered, normal case
  - Setup:
    - Machine contains 1 quarter, 1 loonie, 1 dime, 1 nickel; 1 Coke
    - Machine is on and initialized; door is locked
    - Current value should be 0
  - Action:
    - Insert valid Canadian quarter
  - Expected result:
    - Current value displayed should be 0.25; coin should not be returned; no other change should occur
  - Teardown:
    - Remove the inserted quarter

Example: Vending machine

- Test 2: Washer check
  - Purpose: Check that vending machine responds correctly to a washer being entered, normal case
  - Setup:
    - Machine contains 1 quarter, 1 loonie, 1 dime, 1 nickel; 1 Coke
    - Machine is on and initialized
    - Current value should be 0
  - Action:
    - Insert a washer
  - Expected result:
    - Current value displayed should be 0; washer should be returned; no other change should occur
  - Teardown:
    - None

Example: Vending machine

- See any potential problems with running these tests as described?
  - [Think in terms of doing them quickly, repeatably, consistently,…]

Agenda

- ~~Basic concepts~~
- ~~Test plans~~
- Automated testing

Automated testing

- Problems:
  - during and after a bug fix, tests must be rerun repeatedly
  - after a change, previously passing test cases may now fail (called *regression*)
  - manual tests are tedious, and likely not to be performed consistently
- Solution:
  - automate your tests
- [Additional problem:
  - not all tests are easily automated (e.g., tests requiring physical interactions)]

xUnit

- xUnit is a family of automated testing frameworks
  - JUnit is the variety that targets Java programs
- test fixture
  - set up and tear down shared by a set of test cases
- assertion
  - a check on some condition that has to be true (or false) for the test case to pass
- test case
  - run independently from other test cases
- test suite
  - set of test cases that share a test fixture

Example

```
public class Subscription {
   // subscription total price in cents
   private int price ;
   // length of subscription in months
   private int length ;

   public Subscription(int p, int n) {
     price = p ;
     length = n ;
   }

/**
  * Calculate the monthly subscription
  * price in dollars, rounded up to the
  * nearest cent.
  */
  public double pricePerMonth() {
    double r = (double) price /
                    (double) length ;
    return r ;
  }

 /**
  * Cancels this subscription.
  */
  public void cancel() { length = 0 ; }
}
```

Example

```
import org.junit.* ;
import static org.junit.Assert.* ;

public class SubscriptionTest {
  @Test
  public void pricePerMonthIsInDollars() {
    Subscription S = new Subscription(200,2) ;
    assertTrue(S.pricePerMonth() == 1.0) ;
  }

  @Test
  public void pricePerMonthIsRoundedUp() {
    Subscription S = new Subscription(200,3) ;
    assertTrue(S.pricePerMonth() == 0.67) ;
  }
}
```

These are necessary to give your test code access to the test classes, and JUnit assertions

This is an annotation that tells JUnit that this method is a test case

Input values

Expected result

Example

- After compilation, you tell JUnit to execute the tests in this class:

```
java -cp .;<full path to JUnit.jar> org.junit.runner.JUnitCore SubscriptionTest
```

  - or in Eclipse, you select the test case and “Run as” JUnit test suite
- Result:
  - There were 2 failures (one per test case)
  - This means that the expected results (as embodied in the assertions) did not match the actual results
  - NOTE: Despite each failure, JUnit continues to look for additional test cases and run them too

Example

- Explanation
  - There was an inconsistency between the expected result and the actual result (in two cases)
  - Either the class itself or the test cases (or both) contain an error
    - How do you decide which one has the error?
    - You need to have some independent means of determining what is supposed to happen (the requirements, communication with the stakeholders, a decision on your own part)

Example

- What do you do about the error?
  - If it is in your own code that you have not submitted, fix it
  - Otherwise:
    - Report it (for real projects, even small ones) through a bug tracker (e.g., Bugzilla)
    - Either: Leave it for someone else to fix, or fix it immediately yourself
- Any change you make (to fix a bug, to add functionality, etc.) will probably add more bugs
  - Sometimes, leaving the bug alone is the safest thing to do at the moment

JUnit test suite general format

```
import org.junit.*;
import static org.junit.Assert.*;

public class TestFoobar {
    @BeforeClass public static void setUpClass() throws Exception {
        /* Code executed before the first test method */ }

    @AfterClass public static void tearDownClass() throws Exception {
        /* Code executed after the last test method */ }

    @Before public void setUp() throws Exception {
        /* Code executed before each test method */ }

    @After public void tearDown() throws Exception {
        /* Code executed after each test method */ }

    @Test public void someTest() { /* Do something, make assertion */ }
    @Test public void otherTest() { /* Do something, make assertion */ }
}
```

JUnit more generally

- Order in which test cases will be executed is undefined, so don’t count on it
- Use a set up method to put the system into a known state
- Use a tear down method to clean up the mess from a (potentially failed) test case

Next time

- Testing at different granularities