CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
SOFTWARE TESTING

Autumn 2011

Testing

- "[T]he means by which the presence, quality, or genuineness of anything is determined; a means of trial." –dictionary.com
- A *software test* executes a program to determine whether a property of the program holds or doesn't hold
- A test *passes* [*fails*] if the property **holds** [**doesn't hold**] on that run
- A *test suite* – a set of systematically-designed software tests – executes a program to increase confidence about whether specific properties of the program hold or don't hold
  - The collection of passed and failed tests as a whole provides information beyond each individual test
  - Just as the result of a single coin flip tells little about fairness while a longer sequence can tell more

Software Quality Assurance (QA)
Testing plus other activities including

- Static analysis (assessing code without executing it)
- Proofs of correctness (theorems about program properties)
- Code reviews (people reviewing others' code)
- Software process (placing structure on the development lifecycle)
- …and many more ways to find problems and to increase confidence

No single activity or approach can guarantee software quality

Kinds of Testing

- Unit Testing: does each unit (class, method, etc.) do what it supposed to do?
- Integration Testing: do you get the expected results when the parts are put together?
- Validation Testing: does the program satisfy the requirements?
- System Testing: does it work within the overall system?

Today

- Absolute basics of unit testing, which is our primary focus in 331, using `RandomHello` as an example
- Some examples of JUnit – a Java unit testing mechanism that is nicely integrated into Eclipse
- **Later lectures:** more on testing

Some other testing buzzwords: alpha, beta, fuzz, random, mutation, symbolic, black & white box, coverage (statement/edge/path), model-based … and *many* more …

Unit testing

A. Choose input data ("test inputs")
B. Define the expected outcome ("oracle")
C. Run the unit ("SUT" or "software under test") on the input and record the results
D. Examine results against the oracle

Specification
Precondition | Postcondition
Implementation
= oracle?

Black box
Must choose inputs *without knowledge* of the implementation

White box
Can choose inputs *with knowledge* of the implementation

It's not black-and-white, but…

Black box
Must choose inputs *without knowledge* of the implementation

- Has to focus on the behavior of the SUT
- Needs an oracle
  - Or at least an expectation of whether or not an exception is thrown

White box
Can choose inputs *with knowledge* of the implementation

- Black-box++
- Common use: coverage
- Basic idea: if your test suite never causes a statement to be executed, then that statement might be buggy

sqrt example

```
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)
```

What are some values or ranges of x that might be worth testing

- `x < 0` (exception thrown)
- `x ≥ 0` (returns normally)
- around `x = 0` (boundary condition)
- perfect squares (`sqrt(x)` an integer), non-perfect squares
- `x < sqrt(x)`, `x > sqrt(x)`
- Specific tests: say `x = {-1, 0, 0.5, 1, 4}`

Subdomains

- Many executions reflect the same behavior – for `sqrt`, for example, the expectation is that
  - all `x < 0` inputs will throw an exception
  - all `x ≥ 0` inputs will return normally with a correct answer
- By testing any element from each *subdomain*, the intention is for the single test to represent the other behaviors of the subdomain – *without testing them!*
- Of course, this isn't so easy – even in the simple example above, what about when x overflows?

Testing `RandomHello`

- "Create your first Java class with a main method that will randomly choose, and then print to the console, one of five possible greetings that you define."
- We'll focus on the method `getGreeting`, which randomly returns one of the five greetings
- We'll focus on *black-box testing* – we will work with no knowledge of the implementation
- And we'll focus on unit testing using the JUnit framework
- Intermixing, with any luck, slides and a demo

Does it even run and return?

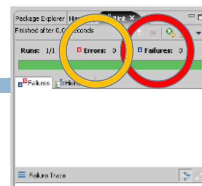
- If `getGreeting` doesn't run and return without throwing an exception, it cannot meet the specification

JUnit tag "this is a test"	`@Test`
name of test	`public void test_NoException(){`
Run `getGreeting`	`RandomHello.getGreeting();`
JUnit "test passed" (doesn't execute if exception thrown)	`assertTrue(true);`
	`}`

Tests should have descriptive (often very long) names

A unit test is a (stylized) program! When you're writing unit tests (and many other tests), you're programming!

Running JUnit tests



- There are many ways to run JUnit test method, test classes, and test suites
- Generally, select the method, class or suite and **Run As >> JUnit Test**
- A green bar says "all tests pass"
- A red bar says at least one test failed or was in error
- The failure trace shows which tests failed and why

- A failure is when the test doesn't pass – that is, the oracle it computes is incorrect
- An error is when something goes wrong with the program that the test didn't check for (e.g., a null pointer exception)

Does it return one of the greetings?

- If it doesn't return one of the defined greetings, it cannot satisfy the specification

```
@Test
public void testDoes_getGreeting_returnDefinedGreeting() {
  String rg = RandomHello.getGreeting();
  for (String s : RandomHello.greetings) {
    if (rg.equals(s)) {
      assertTrue(true);
      return;
    }
  }
  fail("Returned greeting not in greetings array");
}
```

A JUnit test class

13

```
import org.junit.*;
import static org.junit.Assert.*;
public class RandomHelloTest() {
  @Test
  public void test_ReturnDefinedGreeting() {
    …
  }
  @Test
  public void test_EveryGreetingReturned() {
    …
  }
  …
}
```

Don't forget that Eclipse can help you get the right `import` statements – use Organize Imports (Ctrl-Shift-O)

- All `@Test` methods run when the test class is run
- That is, a JUnit test class is a set of tests (methods) that share a (class) name

CSE 331 Autumn 2011

Does it return a random greeting?

14

```
@Test
public void testDoes_getGreetingNeverReturnSomeGreeting() {
  int greetingCount = RandomHello.greetings.length;
  int count[] = new int[greetingCount];
  for (int c = 0; c < greetingCount; c++)
    count[c] = 0;
  for (int i = 1; i < 100; i++) {
    String rs = RandomHello.getGreeting();
    for (int j = 0; j < greetingCount; j++)
      if (rs.equals(RandomHello.greetings[j]))
        count[j]++;
  }
  for (int j = 0; j < greetingCount; j++)
    if (count[j] == 0)
      fail(j+"th [0-4] greeting never returned");
  assertTrue(true);
}
```

Run it 100 times

If even one greeting is never returned, it's unlikely to be random ($\sim 1-0.8^{100}$)

CSE 331 Autumn 2011

What about a sleazy developer?

15

```
if (randomGenerator.nextInt(2) == 0) {
   return(greetings[0]);
} else
   return(greetings[randomGenerator.nextInt(5)]);
```

- Flip a coin and select **either** a random **or** a specific greeting
- The previous "is it random?" test will almost always pass given this implementation
- But it doesn't satisfy the specification, since it's not a random choice

CSE 331 Autumn 2011

Instead: Use simple statistics

16

```
@Test
public void test_UniformGreetingDistribution() {
 // …count frequencies of messages returned, as in
 // …previous test (test_EveryGreetingReturned)

  float chiSquared = 0f;
  float expected = 20f;
  for (int i = 0; i < greetingCount; i++)
    chiSquared = chiSquared +
            ((count[i]-expected)*
             (count[i]-expected))
                         /expected;
  if (chiSquared > 13.277) // df 4, pvalue .01
    fail("Too much variance");
}
```

CSE 331 Autumn 2011

A JUnit test suite

17

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  RandomHelloTest.class,
  SleazyRandomHelloTest.class
})
public class AllTests {
  // this class remains completely
  // empty, being used only as a
  // holder for the above
  // annotations
}
```

- Define one suite for each program (for now)
- The suite allows multiple test classes – each of which has its own set of `@Test` methods – to be defined and run together
- Add `tc.class` to the `@Suite.SuiteClasses` annotation if you add a new test class named `tc`
- So, a JUnit test suite is a set of test classes (which makes it a set of a set of test methods)

CSE 331 Autumn 2011

JUnit assertion methods

18

…causes the current test to fail…	
`fail()`	`immediately`
`assertTrue(tst)`	`if tst is false`
`assertFalse(tst)`	`if test is true`
`assertEquals(expected, actual)`	`if expected does not equal actual`
`assertSame(expected, actual)`	`if expected != actual`
`assertNotSame(expected, actual)`	`if oracle == actual`
`assertNull(value)`	`if value is not null`
`assertNotNull(value)`	`if value is null`

- Can add a failure message: `assertNull("Ptr isn't null", value)`
- **`expected`** is the oracle – remember this is the first (leftmost) param
- The table above only describes when to fail – what happens if an assertion succeeds? Does the test pass?

CSE 331 Autumn 2011

ArrayIntList: example tests

19

```
@Test
public void testAddGet1() {
  ArrayIntList list = new
               ArrayIntList();
  list.add(42);
  list.add(-3);
  list.add(15);
  assertEquals(42, list.get(0));
  assertEquals(-3, list.get(1));
  assertEquals(15, list.get(2));
}
```

```
@Test
public void testIsEmpty() {
  ArrayIntList list = new
                 ArrayIntList();
  assertTrue(list.isEmpty());
  list.add(123);
  assertFalse(list.isEmpty());
  list.remove(0);
  assertTrue(list.isEmpty());
}
```

- High-level concept: test behaviors in combination
  - Maybe `add` works when called once, but not when call twice
  - Maybe `add` works by itself, but fails (or causes a failure) after calling `remove`

A few hints: data structures

20

- Need to pass lots of arrays? Use array literals

```
public void exampleMethod(int[] values) { ... }
...
exampleMethod(new int[] {1, 2, 3, 4});
exampleMethod(new int[] {5, 6, 7});
```

- Need a quick **`ArrayList`**?

```
List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
```

- Need a quick set, queue, etc.? Many take a list

```
Set<Integer> list = new HashSet<Integer>(
                            Arrays.asList(7, 4, -2, 9));
```


A few general hints

21

- Test one thing at a time per test method
  - 10 small tests are much better than one large test
- Be stingy with **`assert`** statements
  - The first **`assert`** that fails stops the test – provides no information about whether a later assertion would have failed
- Be stingy with logic
  - Avoid **`try/catch`** – if it's supposed to throw an exception, use **`expected=`** ... if not, let JUnit catch it

Test case dangers

22

- Dependent test order
  - If running Test A before Test B gives different results from running Test B then Test A, then something is likely confusing and should be made explicit
- Mutable shared state
  - Tests A and B both use a shared object – if A breaks the object, what happens to B?
    - This is a form of dependent test order
    - We will explicitly talk about invariants over data representations and testing if the invariants are ever broken

More JUnit (but not in detail today)

23

- Timeouts – don't want to wait forever for a test to complete
- Testing for exceptions

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
  ArrayIntList list = new ArrayIntList();
  list.get(4); // this should raise the exception
}              // and thus the test will pass
```

- Setup [teardown] – methods to run before [after] each test case method [test class] is called

One view of testing

24

Testing by itself does not improve software quality. Test results are an indicator of quality, but in and of themselves, they don't improve it. Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. What you eat before you step onto the scale determines how much you will weigh, and the software development techniques you use determine how many errors testing will find. If you want to lose weight, don't buy a new scale; change your diet. If you want to improve your software, don't test more; develop better.

Steven C McConnell *Code Complete: A Practical Handbook of Software Construction.* ISBN: 1556154844

Next steps

- Assignment 1: on the web now, due Friday 11:59PM
- Section Thursday: Javadoc, JUnit and Eclipse – in your regularly scheduled rooms
- Lectures: equality (W), ADTs (F & M)