Software Testing

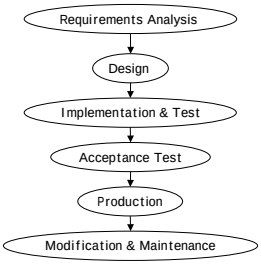
General Motors crash test dummies

Week 5
CS 212 – Spring 2008

Announcements

- Part 2 Assignment
  - Compiler Project Part 2 is online
  - GBA Project Part 2 was delayed, but is now online
  - Both are due Sunday, March 9

Recall: The Waterfall Model



- This model is idealized
  - True development is never entirely sequential
  - There is feedback from each stage of the process

Another Model for Software Development

- This is a diagram from a website promoting *extreme programming* (http://www.extremeprogramming.org/)

Some Features of *Extreme Programming*

- All code is written in response to a *user story* (4x6 card describing requirements)
- Start with smallest set of useful features; release early and often
- Simple design
  - Use simplest possible design that gets the job done
- Continuous testing
  - Tests are written *before* programming
  - When the tests are passed, the job is done
- Continuous integration
  - New code is added daily, but *all* tests must be passed
- *Pair programming*
  - Two programmers at one machine

Pair Programming

- Two programmers share one computer
  - One is the *driver*
    - Controls keyboard and mouse
    - Does all the writing of code
  - The other is the *observer*
    - Watches and guides
    - Focuses on strategic issues (e.g., how this module fits with others)
    - Is usually the *better or more experienced* programmer
- Claim: pair programming is *more productive* than having two separate programmers
- I've never tried it, but you might want to try this with your group

The Tower of Babel (1563). Pieter Bruegel the Elder

Software Testing

- Testing is an important part of software development
  - True for the Waterfall Model
  - Especially true for Extreme Programming (XP)
- XP is an example of TDD (Test Driven Development)
  - Tests are written *first*
  - As soon as all tests are passed, the project is *done*

TDD Cycle

1. Create a test
   - Done during initial development
   - Also done when new features are added
2. Run all tests and see the new one fail
   - Make sure it fails for expected reason
3. Write some code
   - Goal is just to pass the test, even if code is inelegant
   - No extra (and untested) functionality should be added
4. Run automated tests and see all tests succeed
5. Refactor (i.e., consolidate, re-arrange, and clean up) code
   - Tests are re-run as changes are made

Repeat

Test Driven Development

- Benefits
  - By focusing on test-cases, programmer is concerned with interface and not implementation
  - All code written is covered by a test
    - Programmer (and later users) can have greater confidence in the code
  - Unexplained failure can sometimes be "fixed" more easily by reverting to previous version (that passed all tests) than by debugging
- Limitations
  - More code must be produced (i.e., the code for the tests)
    - But total implementation time is claimed to be shorter
  - Systems with complex input/output can be hard to test (e.g., GUIs or Relational Databases)
- For more detail on TDD
  - Wikipedia article on TDD
  - *Test Driven Development: By Example*, a book by Kent Beck

Unit vs. Integration Testing

- Unit testing
  - Testing of a single module
  - If a unit fails to match its specification then it is considered to be incorrect
- Integration testing
  - Testing of the entire program
  - Failure here may imply that the specifications are incorrect
  - Integration testing is usually harder than unit testing

Black Box vs. White Box vs. Glass Box Testing

- Black Box testing implies that the tests are based entirely on the public interface
  - This is the kind of testing encouraged by TDD
- Glass Box testing implies that tests are based on the internal code
  - Such tests can become useless if the implementation changes
- Some authors make a distinction between White Box and Glass Box testing
  - White Box implies you can modify the internal state of the object being tested
  - Glass Box implies you can see the internal state, but not modify it

Tools for Testing

- Goal: automate as much of the testing as we can
  - Some parts can't be automated
    - Process of developing test cases is difficult and usually cannot be fully automated
  - But we can automate the testing process itself
    - Both DrJava and Eclipse include facilities for using JUnit (http://www.junit.org)
      - Simplifies the process of writing unit tests
- Can make use of *drivers* and *stubs*
  - A driver
    - Calls the unit being tested and keeps track of how it performs
  - A stub
    - Simulates a program-part that is called by the unit being tested
  - Both can interact with a file or with a person
    - Example: a driver can read calling parameters from a file and save test results to another file

JUnit

- JUnit is an open source framework for writing and running repeatable tests
  - Original version was written by Erich Gamma and Kent Beck
  - There are now similar tools for several languages
    - C++ (CppUnit)
    - Fortran (fUnit)
    - Python (PyUnit, now unittest)
    - ...
- Makes it easy to write and manage various tests

JUnit Example

- Derived from an article by Antonio Goncalves [http://www.devx.com/Java/Article/31983]

```
package calc;

public class Calculator {

    private static int result;

    public void add(int n) { result = result + n;}

    public void subtract (int n) {
        result = result - 1;        // Bug!
    }

    public void multiply(int n) {
    }        // not ready yet

    public void divide(int n) {result = result/n;}

    public void square(int n) {result = n * n;}

    public void squareRoot(int n) {
        for (; ;) ;             //Bug : loops forever
    }

    public void clear() {result = 0;}

    public void switchOn() {
        // Beep and do other calculator stuff
        result = 0;
    }

    public void switchOff() {
        // Beep and switch off
    }

    public int getResult() {return result;}
}
```

JUnit Annotations

- The current version of JUnit (JUnit 4) uses annotations to communicate with the testing framework
  - @Test
    the following method is a test method
  - @Before
    the following method should be run before each test
  - @After
    the following method should be run after each test
  - @Ignore
    the following @Test method should be ignored

Code for Testing the Calculator

```
package mytests;

import calc.Calculator;
import org.junit.*;
import static org.junit.Assert.*;

public class CalculatorTest {

  private static Calculator calculator =
          new Calculator();

  @Before
  public void clearCalculator() {
    calculator.clear();
  }

  @Test
  public void add() {
    calculator.add(1); calculator.add(1);
    assertEquals(calculator.getResult(), 2);
  }

@Test public void subtract() {
  calculator.add(10); calculator.subtract(2);
  assertEquals(calculator.getResult(), 8);
}

@Test public void divide() {
  calculator.add(8); calculator.divide(2);
  assert calculator.getResult() == 5;  // Bug!
}

@Test(expected = ArithmeticException.class)
public void divideByZero() {
  calculator.divide(0);
}

@Ignore("not ready yet")
@Test
public void multiply() {
  calculator.add(10); calculator.multiply(10);
  assertEquals(calculator.getResult(), 100);
} }
```

Running the Tests

- To run from the command window (assuming your CLASSPATH is set correctly)
  - java org.junit.runner.JUnitCore <test class name>
  - For our example
    - java org.junit.runner.JUnitCore mytests.CalculatorTest
- The result is a listing showing each failed test
  - For our example, 4 tests with 2 failures
- There are other features; for example, we can test for too much time:

```
@Test(timeout = 1000)
public void squareRoot() {
    calculator.squareRoot(2);
}
```

Coding Quality

- Pareto Principle
  - Named for Vilfredo Pareto, Italian economist, late 1800's
  - An 80/20 rule that shows up often
    - 80% of complaints are about 20% of the products
    - 80% of the decisions are completed during 20% of a meeting
- Software version: 80% of software defects occur in just 20% of the modules
- NSA study [Drake, IEEE Computer, 1996] on 25 million lines of code
  - 70-80% of problems were due to 10-15% of modules
  - 90% of all defects were in modules containing 13% of the code
  - 95% of *serious* defects were from just 2.5% of the code
- Sturgeon's Revelation?

Software Reliability Estimates

- Define R(n) as the probability of not failing in n time steps (e.g., n days)
- Define F(n) as the probability of failing within n time steps
  - Clearly, R(n) + F(n) = 1
- *If* the failure probability for each day is independent of the preceding days then
  - $R(n) = R(1)^n$
  - We can consider each day to be a *trial*
  - This fits the standard definition of a Bernoulli process
  - R(1) = Prob of success = p
  - F(1) = Prob of failure = q

Software Reliability Estimates, Cont'd

- Given p (and q), we can compute the expected time between failures

$E(T) = \sum_t t \cdot \text{Prob}(T=t)$

$E(T) = 1q + 2pq + 3p^2q + 4p^3q + \ldots = q(1 + 2p + 3p^2 + 4p^3 + \ldots)$

$= q/(1-p)^2 = 1/q$

- We can use this information to estimate other quantities
  - Assume that on average failures happen every 4 days
  - What's the probability that the system will run without failure for the next 6 days?
    - We know $4 = 1/q$, so $q = \frac{1}{4}$, and thus $p = \frac{3}{4}$
    - Probability of 6 days in a row without failure = $(\frac{3}{4})^6 \approx .178$
  - This estimate assumes that the failures can be accurately modeled as a Bernoulli process