Testing

Jules White jules@dre.vanderbilt.edu
Institute for Software Integrated Systems (ISIS) Vanderbilt University, Nashville, TN

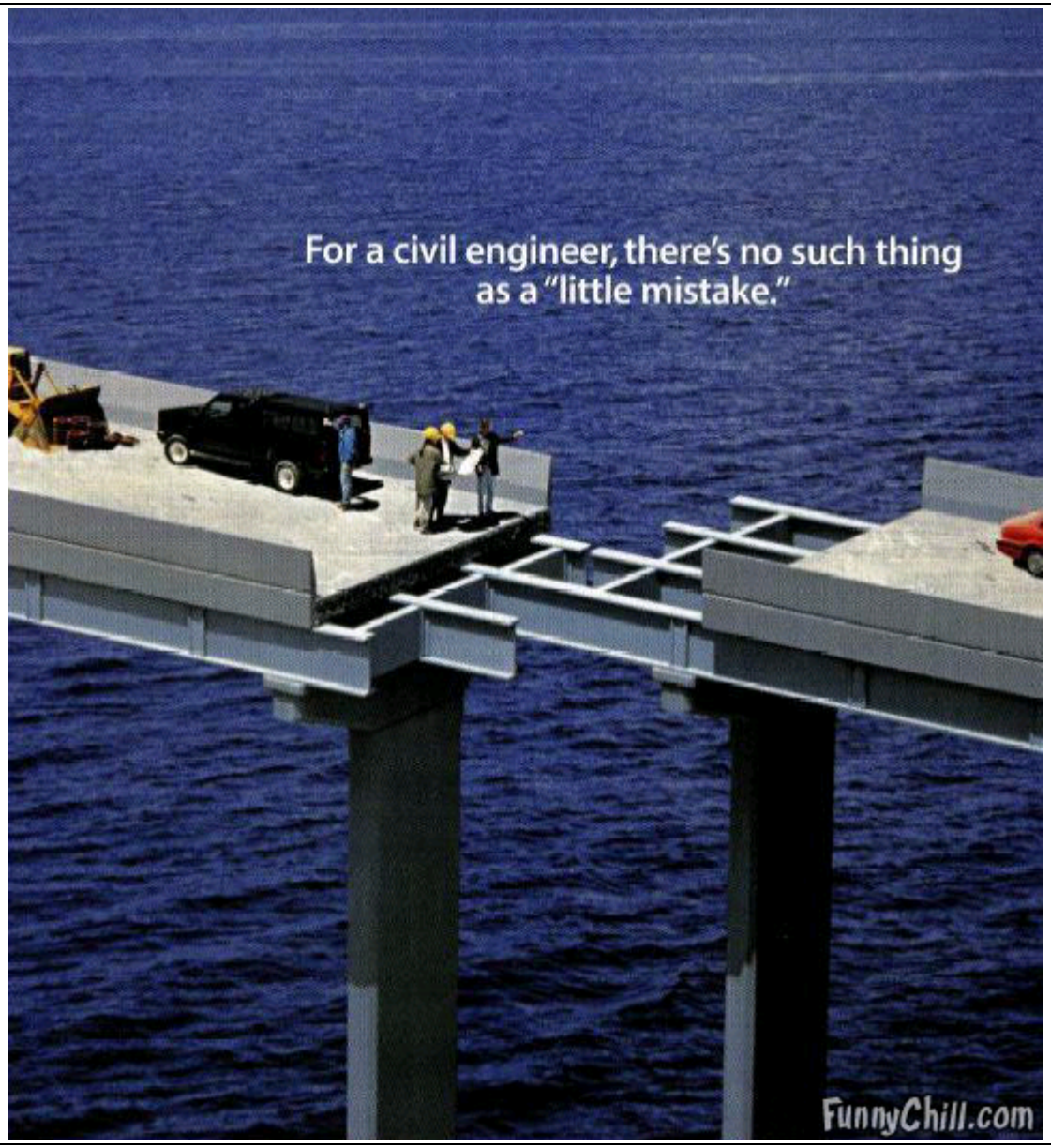
Testing

Program testing can be used to show the presence of bugs, but never to show their absence!

--Dijkstra

How Do We Test?

- Generally, we use languages that are too complex to validate that an implementation is correct
- To test, we want to ensure that we expose the conditions where the code is likely to fail
  - What input/output values cause problems
  - What execution paths cause problems
  - What combinations of input/output values cause problems
- There are far too many possible combinations of factors to test every possible input/output and execution path combination


For a civil engineer, there's no such thing as a "little mistake."
FunnyChill.com

Why are Civil Engineers Beating Us?

- They completely model the project before they start
- They use languages that have well understood and measurable properties
  - They can validate/verify the correctness of the design itself
- We use languages that do not have well understood and easily measurable properties
  - E.g. C, and those other platform-independent assembly languages
- We cannot verify that a design is correct without implementing it

What if We Built Bridges Like This?

- We draw a rough sketch of what the bridge should look like
- We hand the sketch to a contractor and ask him/her how much it will cost
- The contractor builds the bridge
  - The bridge is behind schedule
- Not until the bridge is built do we begin testing it
  - We drive some cars across…see if it collapses
  - We drive more cars across
  - We drive a truck across…and declare victory
  - And…the bridge collapses 2 weeks later when it rains

Unfortunately….

- This scenario is how we test software….
- In the future, hopefully, we will all use model-driven development (we will talk about this soon)

- Testing software is about trying to make sure that you have executed the code in a sufficiently large number of scenarios to be fairly sure that it works

Testing Review

- Everything is tested as soon as possible
  - Unit Testing
  - Regression Testing
  - Integration Testing

- The later a defect is discovered, the more expensive it is to correct

Static vs. Dynamic Testing

- Static testing analyzes the code to determine if it is correct
  - Code reviews, walkthroughs, static analysis tools
- Dynamic testing executes the code in different ways to discover problems
  - Unit tests, integration tests, etc.

Test Coverage

- An important part of testing is the notion of *Test Coverage*
- Test coverage can be defined in different ways:
  - What % of the possible execution paths do the tests cause to be executed
  - What % of the range of input/output values are exercised by the tests
  - What % of the different control flow branches get executed
  - What % of the program statements are executed
  - Etc…

Test Coverage

- What % of the possible execution paths do the tests cause to be executed
- Does our test cause both foo() and bar() to be executed?

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
        bar(size);
    }
}
```

Test Coverage

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
          bar(size);
    }
}

public class FooTest {
    public void testFoo(){
          Foo foo = new Foo();
          foo.bar(1);
          foo.foo(2);
    }
}
```

Do we have complete coverage from an execution point of view?

Test Coverage

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
          bar(size);
    }
}

public class FooTest {
    public void testFoo(){
          Foo foo = new Foo();
          foo.bar(1);
          foo.foo(2);
    }
}
```

No!!!
We never execute the path:
Foo.bar(size > 2)->foo()->bar()..

Test Coverage

- What % of the possible range of input/output values do we cover?

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
        bar(size);
    }
}
```

Test Coverage

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
          bar(size);
    }
}

public class FooTest {
    public void testFoo(){
          Foo foo = new Foo();
          foo.bar(1);
          foo.foo(2);
    }
}
```

Do we have complete coverage from an I/O value point of view?

Equivalence Partitioning

- Testing every possible integer value for size would be expensive…

```
public void bar(int size) {
      if(size > 2){foo(size)};
}
```

Equivalence Partitioning

- Equivalence partitioning reduces the number of test cases and chooses the right test cases to simplify testing
- We find the boundary areas for the input parameters to create partitions
  - Size = 2
  - Size < 2
  - Size > 2

```
public void bar(int size) {
      if(size > 2){foo(size)};
}
```

Test Coverage

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
          bar(size);
    }
}

public class FooTest {
    public void testFoo(){
          Foo foo = new Foo();
          foo.bar(0); // < 2
          foo.bar(2); // = 2
          foo.bar(5); // > 2
          …
    }
}
```

Covering all partitions for Foo.bar(..)

Boundary Value Analysis

- Most defects occur at the boundaries of valid and invalid input values
- We use equivalence partitioning to find the boundaries and then we carefully test right at the boundaries
- A “clean” test case and a “negative” test case are run at each boundary
  - Clean test case should be just within bounds and produce a valid result
  - Negative test case should be just outside bounds and produce a warning, *expected* exception, etc.

Test Coverage

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
          bar(size);
    }
}

public class FooTest {
    public void testFoo(){
          Foo foo = new Foo();
          foo.bar(2); // = 2
          foo.bar(3); // > 2
          …
    }
}
```

Better test ranges for Foo.bar(..)

Fuzz Testing

- It may be hard to figure out where the boundaries are
- We may make mistakes when delineating the boundaries

- Fuzz Testing randomly generates test data to try and find faults
  - Very simple
  - High payoff if it works
  - May be difficult to design a good “fuzzer”

Test Coverage

```
public class Foo {
    public void bar(int size) {
        if(size > 2){foo(size)};
    }
    public void foo(int size){
          bar(size);
    }
}

public class FooTest {
    public void testFoo(){
          Foo foo = new Foo();
          for(int i=0; i < 10; i++){
              foo.foo((int)Math.rint(Math.random()));
           }
           …
    }
}
```

Fuzzing…

Black Box Testing

- Black Box Testing assumes that we don't know anything about the internal implementation of the software
- All testing is done based on the interface provided by the software

Black Box Testing

- Advantages:
  - As long as the interface doesn't change, we don't have to rewrite the test
  - Simplifies the testing process
  - Tests can sometimes be reused across classes that expose the same interface (i.e. the classes implement the same Java interface)

Black Box Testing

- Disadvantages:
  - May be hard to find the partitions and boundaries
  - May provide poor coverage of the code
  - Prevents the developer from gaining insight on what could potentially break the code

White Box Testing

- White Box Testing assumes knowledge of the internal implementation of the software
- All testing is done based on how the class is implemented

White Box Testing

- Advantages:
  - Easier to ensure execution path, branching, etc. coverage
  - May be easier to determine partitions and boundaries

White Box Testing

- Disadvantages:
  - Changes to implementation force changes in the tests
  - Hard to perform test-driven development
    - If you haven't implemented the spec, how can you write a test for it that assumes implementation knowledge?
  - Difficult to reuse tests since they are tied to implementation details

Grey Box Testing

- A hybrid of black/white box testing
- Use knowledge of the internal details of the code to design the tests, find boundaries, etc.
- Write tests that only utilize the external interfaces of the classes
  - Tests are reusable and not tightly-coupled to internal implementation
  - Easier to ensure coverage
  - Still may need to make changes when implementation changes

Testing Behavior

- What is the right way to test the following code:

```
public class AccountUpdater {
    public void deposit(int amt, Account act) {
        Transaction t = new Transaction(DEPOSIT, amt);
        act.addTransaction(t);
        act.setBalance(act.getBalance() + amt);
    }
}

public class Account {….}
public class Transaction {…}
```

Testing Behavior

```
public class AccountTester {
    public void test() {
        Account act = new Account("Brian", 2000);
        AccountUpdater updater = new AccountUpdater();
        updater.deposit(100,act);
        assert(act.getBalance() == 2100);
    }
}
```

Testing Behavior

```
public class AccountTester {
    public void test() {
        Account act = new Account("Brian", 2000);
        AccountUpdater updater = new AccountUpdater();
        updater.deposit(100,act);
        assert(act.getBalance() == 2100);
    }
}
```

If this test fails, where did the error come from?

Testing Behavior

```
public class AccountTester {
    public void test() {
        Account act = new Account("Brian", 2000);
        AccountUpdater updater = new AccountUpdater();
        updater.deposit(100,act);
        assert(act.getBalance() == 2100);
    }
}
```

This test will not distinguish between errors in AccountUpdater and Account?

Testing Behavior

```
public class AccountTester {
    public void test() {
        Account act = new Account("Brian", 2000);
        AccountUpdater updater = new AccountUpdater();
        updater.deposit(100,act);
        assert(act.getBalance() == 2100);
    }
}
```

What if we haven't implemented Account yet? We don't want to wait to test!!!!

Mock Objects

- We create a simplified dummy object to ensure that the AccountUpdater has the appropriate behavior.

```
public class AccountDepositMockObject implements Account {
   private boolean transactionAdded_ = false;
   private boolean balanceUpdated_ = false;

   public void addTransaction(Transaction t){
      transactionAdded_ = (t != null && t.amt == 100);
   }
   public void setBalance(int b){
      balanceUpdated_ = (b == 2100);
   }
   public void getBalance(){
      return 2000;
   }
   public int getDepositAmount(){ return 100;}
  …
}
```

Mock Objects

```
public class AccountTester {
    public void test() {
        AccountDepositMockObject act =
                new AccountDepositMockObject();
        AccountUpdater updater = new AccountUpdater();
        updater.deposit(act.getDepositAmount());
        assert(act.transactionAdded());
        assert(act.balancedUpdated());
    }
}
```

Mock Objects

```
public class AccountTester {
    public void test() {
        AccountDepositMockObject act =
                new AccountDepositMockObject();
        AccountUpdater updater = new AccountUpdater();
        updater.deposit(act.getDepositAmount());
        assert(act.transactionAdded());
        assert(act.balancedUpdated());
    }
}
```

If the test fails, it must be because of the AccountUpdater

Mock Objects

```
public class AccountTester {
    public void test() {
        AccountDepositMockObject act =
                new AccountDepositMockObject();
        AccountUpdater updater = new AccountUpdater();
        updater.deposit(act.getDepositAmount());
        assert(act.transactionAdded());
        assert(act.balancedUpdated());
    }
}
```

We can also test the AccountUpdater before Account is fully implemented

Fault Injection

- Force an error to occur and see that it is handled properly
- Two general types:
  - Runtime Fault Injection → Example: turn off the database that the application is attached to and see what happens
  - Compile-Time Fault Injection → Example: create a database driver that throws an exception when a query is executed and compile it into the application

Non-functional Testing

- Does the system do what it is supposed to do with the appropriate non-functional qualities
  - Is the code secure?
  - Does the code execute fast enough?
  - Does the code consume too much memory?
  - Etc.

Performance Testing

- Determining how the system performs under a particular workload

- Common types of performance testing:
  - Load testing → test the system under a specific workload to understand how it reacts
  - Stress testing → push the system towards its breaking point to determine robustness
  - Endurance or Soak Testing → allow the system to run for an extended period of time (find memory leaks, etc.)
  - Spike Testing → create sudden peaks in workload to see how the system responds

System Testing

- Does the system meet its requirements?

System IntegrationTesting

- Does the system work when it is integrated with other systems:
  - Do Vendor1 and Vendor2's systems work properly in conjunction with each other
  - Does the system work when integrated with the real legacy database rather than a test database instance
  - Etc.

JUnit

- A Java unit/regression testing framework
- A standardized TestCase implementation
  - Similar to what you produced for Asgn 1
- A test execution framework to standardize the running of tests and the reporting of results

Example

```
import junit.framework.TestCase;


public class FooTest extends TestCase {

   public void testFoo(){
       assertTrue(3 == 3);
   }
}
```

Example

```
import junit.framework.TestCase;


public class FooTest extends TestCase {

   public void testFoo(){
        assertTrue(3 == 3);
   }
}
```

When the TestCase is run, every method that starts with “test” is called

Example

```
import junit.framework.TestCase;


public class FooTest extends TestCase {

   public void testFoo(){
        assertTrue(3 == 3);
   }
}
```

TestCase provides assertion methods to specify test conditions

Example

```
import junit.framework.TestCase;


public class FooTest extends TestCase {
    private Foo foo_;
    public void setUp(){
        foo_ = new Foo();
    }

   public void testFoo(){
        assertTrue(foo.foo());
   }
}
```

setUp() can be used to initialize resources for testing

Example

```
import junit.framework.TestCase;


public class FooTest extends TestCase {
    private Foo foo_;
    public void setUp(){
        foo_ = new Foo();
    }
    public void tearDown(){
        foo_.dispose()
    }
   public void testFoo(){
        assertTrue(foo.foo());
   }
}
```

tearDown() is called after the TestCase is run to release resources

Example

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

   public static Test suite() {
        TestSuite suite = new TestSuite(
                        "Test for
   org.foo.test.dsml.emf.testdsml");
        //$JUnit-BEGIN$
        suite.addTestSuite(FooTest.class);
        //$JUnit-END$
        return suite;
   }

}
```

TestCases can be aggregated to form Tests

Example

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

   public static Test suite() {
        TestSuite suite = new TestSuite(
                        "Test for
   org.foo.test.dsml.emf.testdsml");
        //$JUnit-BEGIN$
        suite.addTestSuite(FooTest.class);
        //$JUnit-END$
        return suite;
   }

}
```

A single static method is created to instantiate and identify the set of tests (TestSuite)

Version Control System

- A version control system is a repository for source code that tracks changes over time
- Code is checked in and out of the repository
- Each time code is checked in or “committed,” a new version number is created
- Code can only be committed if it is “up to date”
  - e.g. has the latest version number
- If code is not up to date, developers may be forced to merge versions

Version Control System

- Most version control systems allow developers to create “branches”
- Changes in a branch are separate from the main branch or “trunk” or “head” of the version control system
- Eventually, branches must be merged back into the trunk by reconciling differences

Version Control System

- Common version control systems:
  - CVS
  - Subversion
  - ClearCase
  - Visual Source Safe
  - GIT
- All serious development projects use a version control system

Nightly Builds

- A Nightly Build is used to discover errors in a project's source code
  - A build server is setup that runs autonomously
  - The head revision of the code is autonomously checked out from version control
  - The code is compiled
  - All regression tests are run
  - In the morning, project managers track down developers who checked in broken code
- Breaking a build is considered very poor form
- You should always thoroughly test your code before committing it to version control

Another Neat Tool (ANT)

- ANT was designed as a cross-platform build tool
  - ANT can be used as a general automation framework
- ANT is written in Java
- ANT runs an “ANT script” that gives it a series of “tasks” to perform

2-second XML Review

- XML is a supposedly *human readable* language for capturing data in a platform independent way
- An XML document forms a tree structure
  - The tree structure is called the Document Object Model (DOM)

2-second XML Review

- A node in the tree is defined by a tag:

```
<root>
  <a>
      <b/>
      <b>My Text</b>
   </a>
</root>
```

2-second XML Review

- Nodes can have attributes associated with them:

```
<root>
  <a name="foo">
      <b/>
      <b>My Text</b>
   </a>
</root>
```

Continuous Integration

- Continuous integration is a technique designed to catch bugs as soon as they are introduced into the system
- Each time a file is committed into the version control system, the continuous integration system checks out the latest version of the code, builds, and tests it
- If the build or tests fails, the culprit is known….