Objectives

- Testing

Review: Software Development

- From Monday

CLASSPATH

Classpath

- Tells the compiler or JVM where to look for user-defined classes and packages
  - Often when using third-party libraries
- Similar to PYTHONPATH
- Typically know it needs to be set when there are class not found error messages

Setting the Classpath

- Can specify classpath in command line

```
javac -cp path/to/myjavaclasses MyClass.java
java –cp path/to/myjavaclasses MyClass
```

- Can specify the classpath environment variable
  - Edit your .bash_profile OR
  - Set in terminal

```
CLASSPATH=$CLASSPATH:path/to/myjavaclasses
echo $CLASSPATH
```

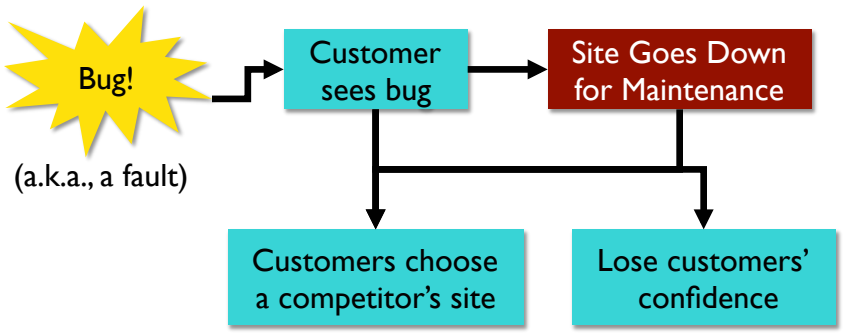
Current value of CLASSPATH

- In Eclipse, you can "Configure Build Path" for a project

SOFTWARE TESTING PROCESS

Why Test Programs?

- Consider an online bookstore



Microsoft Windows Vista Testing

- Beyond their internal testing …
  - 5 million people beta tested
  - 60+ years of performance testing
  - 1 Billion+ Office 2007 sessions
- Still, users found correctness, stability, robustness, and security bugs

Type 1 Bugs: Compile-Time

- Syntax errors
  - Missing semicolon, parentheses
- Compiler notifies of error
- Cheap, easy to fix

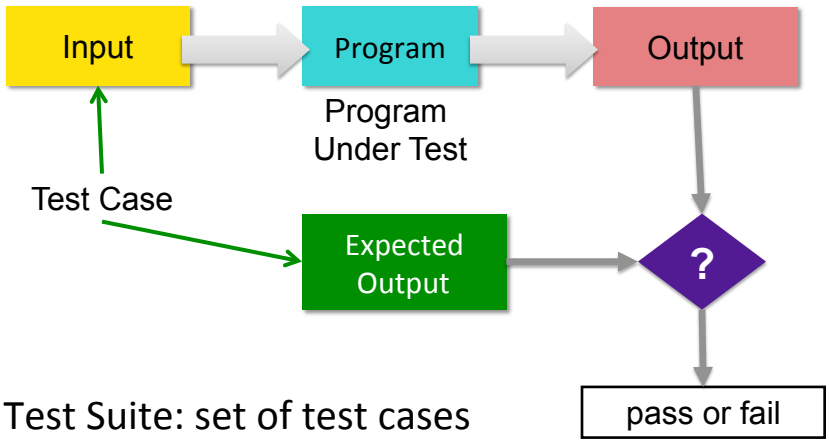
Type 2 Bugs: Run-Time

- Usually logic errors
- Expensive to locate, fix

Aside: Objections to "Bug" Terminology

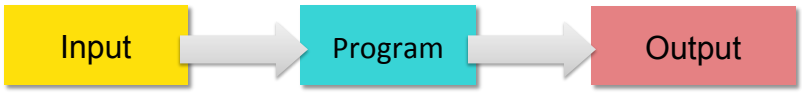
- "Bug"
  - Sounds like it's just an annoyance
    - Can simply swat away
  - Minimizes potential problems
  - Hides programmer's responsibility
- Alternative terms
  - **Defect**
  - **Fault**

Software Testing Process



- Test Suite: set of test cases

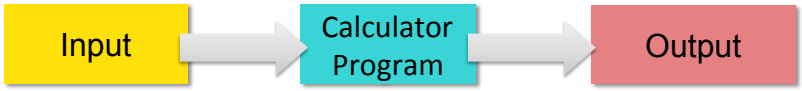
Software Testing Process



- Tester plays devil's advocate
  - ***Hopes*** to reveal problems in the program using "good" test cases
  - Better tester finds than a customer!

How is **testing** different from **debugging**?

How Would You Test a Calculator Program?



Operands, operators, expected output	adds, subtracts, multiplies, divides	Numerical Answer

- What test cases: input and expected output?

Example Test Cases for Calculator Program

- Basic Functionality
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Order of operations
- Invalid Input
  - Letters, not-operation characters (&,$, …)
- "Tricky" Cases
  - Divide by 0
  - Negative Numbers
  - Long sequences of operands, operators
  - VERY large, VERY small numbers

Types of Testing
(Non-Exhaustive)

- Black-box testing
- White-box testing
- Non-functional testing
- Acceptance testing

Ideas or definitions of any of these?

Types of Testing
(Non-Exhaustive)

- Black-box testing
  - Test *functionality* (e.g., the calculator)
  - No knowledge of the code
  - Examples of testing: boundary values
- White-box testing
  - Have access to code
  - **Goal**: execute *all* code
- Non-functional testing
  - Performance testing
  - Usability testing (HCI)
  - Security testing
  - Internationalization, localization
- Acceptance testing
  - Customer tests to decide if accepts product

Levels of Testing

- Unit
  - Tests minimal software component, in isolation
  - For us, Class-level testing
  - Web: Web pages (Http Request)
- Integration
  - Tests interfaces & interaction of classes
- System
  - Tests that completely integrated system meets requirements
- System Integration
  - Test system works with other systems, e.g., third-party systems

UNIT TESTING

Why Unit Test?

- Verify code works as intended in isolation
- Find defects ***early*** in development
  - Easier to test small pieces
  - Less cost than at later stages

Why Unit Test?

- Verify code works as intended in isolation
- Find defects ***early*** in development
  - Easier to test small pieces
  - Less cost than at later stages
- As application evolves, new code is more likely to break existing code
  - Suite of (small) test cases to run after code changes
  - Also called **regression** testing

Some Approaches to Testing Methods

- Typical case
  - Test typical values of input/parameters
- Boundary conditions
  - Test at boundaries of input/parameters
  - Many faults live "in corners"
- Parameter validation
  - Verify that parameter and object bounds are documented and checked
  - Example: pre-condition that parameter isn't null

➥ All black-box testing approaches

Another Use of Unit Testing: Test-Driven Development

- A development style, evolved from Extreme Programming
- Idea: write tests first *without code bias*
- The Process:
  1. Write tests that code/new functionality should pass
     - Like a specification for the code (pre/post conditions)
     - All tests will initially *fail*
  2. Write the code and verify that it passes test cases
     - Know you're done coding when you pass **all** tests

How do you know you're "done" in traditional development?

What assumption does this make?

Software Testing Issues

- How should you test? How often?
  - Code may change frequently
  - Code may depend on others' code
  - A lot of code to validate
- How do you know that an output is correct?
  - Complex output
  - Human judgment?
- What caused a code failure?

➥ Need a *systematic, automated, repeatable* approach

Characteristics of Good Unit Testing

- **Automatic**
- **Thorough**
- **Repeatable**
- **Independent**

Why are these characteristics of good (unit) testing?

JUNIT

JUnit Framework

- A framework for unit testing Java programs
  - Supported by Eclipse and other IDEs
  - Developed by Erich Gamma and Kent Beck
- Functionality
  - Write tests
    - Validate output, automatically
  - Automate execution of test suites
  - Display pass/fail results of test execution
    - Stack trace where fails
  - Organize tests, separate from code
- But, you still need to come up with the tests!

Testing with JUnit

```
tests
   └ CDTest
     DVDTest
     MediaItemTest
```

- Typical organization:
  - Set of testing classes
  - Testing classes packaged together in a **`tests`** package
    - Separate package from code testing
- A test class typically
  - Focuses on a specific class
  - Contains methods, each of which represents another test of the class

Structure of a JUnit Test

1. Set up the test case (optional)
   - Example: Creating objects
2. Exercise the code under test
3. Verify the correctness of the results
4. Teardown (optional)
   - Example: reclaim created objects

Annotations

- Testing in JUnit 4: uses **annotations**
- Provide data about a program that is not part of program itself
- Have no direct effect on operation of the code
- Example uses:
  - @Override: method declaration is intended to override a method declaration in parent class
    - If method does not override parent class method, compiler generates error message
  - Information for the compiler to suppress warnings (@SupressWarnings)

Tests are Methods

- Mark your testing method with `@Test`
  - From `org.junit.Test`

```
public class CalculatorTest {

      @Test
      public void addTest() {
            …
      }
}
```

Class for testing the Calculator class

A method to test the "add" functionality

- Convention: Method name describes what you're testing

Assert Methods

- Variety of assert methods available
- If fail, throw an exception
- Otherwise, test keeps executing
- All `static void`
- Example:
  `assertEquals(Object expected, Object actual)`

```
@Test
public void addTest() {
      …
      assertEquals(4, calculator.add(3, 1));
}
```

Assert Methods

- To use asserts, need *static* import:

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

  - `static` allows us to not have to use classname
- More examples
  - `assertTrue(boolean condition)`
  - `assertSame(Object expected, Object actual)`
    - Refer to same object
  - `assertEquals(double expected, double actual, double delta)`

Example Uses of Assert Methods

```
@Test
public void testEmptyCollection() {
    Collection collection = new ArrayList();
    assertTrue(collection.isEmpty());
}
```

`assertEquals(double expected, double actual, double delta)`

```
@Test
public void testPI() {
    final double ERROR_TOLERANCE = .01;
    assertEquals(Math.PI, 3.14, ERROR_TOLERANCE);
}
```

Will fail if ERROR_TOLERANCE = .001

Set Up/Tear Down

- May want methods to set up objects for every test in the class
  - Called **fixtures**
  - If have multiple, no guarantees for order executed

```
@Before
public void prepareTestData() { ... }

@Before
public void setupMocks() { ... }

@After
public void cleanupTestData() { ... }
```

Executed before **each** test method

Example Set Up Method

```
private CD testCD;

@Before
public void setUp() {
     testCD = new CD("CD title", 100, 1997,
                     "CD Artist", 11);
}
```

`@Before` Executed before **each** test method
Can use `testCD` in test methods

Expecting an Exception

- Handling Error Cases
  - Sometimes an exception *is* the expected result

Add an "expected" attribute:

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
   ArrayList emptyList = new ArrayList();
   Object o = emptyList.get(0);
}
```

Test case passes iff exception thrown

Set Up/Tear Down For Class

- May want methods to set up objects for set of tests
  - Executed once before any test in class executes

```
@BeforeClass
public static void
setupDatabaseConnection() { ... }

@AfterClass
public static void
teardownDatabaseConnection() { ... }
```

JUNIT IN ECLIPSE

Using JUnit in Eclipse

- Eclipse can help make our job easier
  - Automatically execute tests (i.e., methods)
  - We can focus on coming up with tests

Using JUnit in Eclipse

- In Eclipse, go to your `MediaItem` project
- Create a new JUnit Test Case (under Java)
  - Use JUnit **4**
    - Add junit to build path
  - Put in package `media.tests`
  - Name: `DVDTest`
  - Choose to test `DVD` class
    - Select `setUp` and `tearDown`
    - Select methods to test
- Run the class as a JUnit Test Case

Example

- Test method that gets the length of the DVD
  - Revise: Add code to `setUp` method that creates a DVD
- Notes
  - Replaying all the test cases: right click on package
  - FastView vs Detached
  - Hint: CTL-Spacebar to get auto-complete options

Unit Testing & JUnit Summary

- Unit Testing: testing smallest component of your code
  - For us: class and its methods
- JUnit provides framework to write test cases and run test cases automatically
  - Easy to run again after code changes
- JUnit Resources available from Course Page's "Resource" Link, under Java
  - API
  - Tutorials