Section 4: Graphs and Testing

Agenda
- Graphs
- Java assertions
- Internal vs. external testing
- Representation invariants in real code

Graphs

- Nodes
- Children of A
- Edges

Slides adapted from Alex Mariakakis, with material from Krysta Yousoufian, Mike Ernst, and Kellen Donohue
Graphs

Parents of D

Path from A to C

Shortest path from A to C?

Shortest path from A to B?

Practical Graphs

Petr Simecek’s FB Mining

Practical (?) Graphs

Chris Webb’s Eurovision Voting
Java Asserts

Demo!

Enabling Java Asserts

- Right click the .java file you are running
- Go to “Run As” → “Run Configurations”
- Click on the “Arguments” tab
- Enter “-ea” under “VM arguments”

Assertions vs. Exceptions

- Assertions should check for things that should never happen
- Exceptions should check for things that might happen
- “Exceptions address the robustness of your code, while assertions address its correctness”

Java Asserts

- assert(someValue); where someValue is a non-obvious boolean
- someValue should always be true unless something is broken
- Asserts do not run unless specifically enabled
- Our autograder will enable them, so remember to enable asserts!

Internal vs. External Testing

- Internal: JUnit
  - How you decide to abstract the object
  - Checked with implementation tests
  - If it’s something you’re testing about your implementation that might not be true for everyone’s, it’s internal.
- External: test script
  - How the client uses the object
  - Checked with specification tests
  - If it’s something that should be true for anybody’s implementation from the same spec, it’s external.

A JUnit Test Class

- A method with @Test is flagged as a JUnit test
- All @Test methods run when JUnit runs

```java
import org.junit.*;
import static org.junit.Assert.*;
public class TestSuite {
    ...
    @Test
    public void TestName1() {
        ...
    }
    ...
}
```
JUnit Value Checking

- assertEquals, assertNull, assertNotSame, etc...
- These are not the same as Java assert
- Verify that a value matches expectations:
  - assertEquals(42, meaningOfLife());
  - assertTrue(list.isEmpty());
- If the value isn’t what it should be, the test fails
  - Test immediately terminates
  - Other tests in the test class are still run as normal
  - Results show details of failed tests

Checking for Exceptions

- Verify that a method throws an exception when it should
- Test passes if specified exception is thrown, fails otherwise
- Only time it’s OK to write a test without a form of asserts

```java
@Test(expected=IndexOutOfBoundsException.class)
public void testGetEmptyList() {
    List<String> list = new ArrayList<String>();
    list.get(0);
}
```

Setup and Teardown

```java
public class Example {
    List empty;

    @Before
    public void initialize() {
        empty = new ArrayList();
    }

    @Test
    public void size() {
        ...
    }

    @Test
    public void remove() {
        ...
    }
}
```

Don’t Repeat Yourself

```java
public class Example {
    List empty;

    @Before
    public void initialize() {
        empty = new ArrayList();
    }

    @Test
    public void size() {
        ...
    }

    @Test
    public void remove() {
        ...
    }
}
```
#1: Be descriptive

- When a test fails, JUnit tells you:
  - Name of test method
  - Message passed into failed assertion
  - Expected and actual values of failed assertion
- The more descriptive this information is, the easier it is to diagnose failures

<table>
<thead>
<tr>
<th>Level of goodness</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>testAddDaysWithinMonth()</td>
</tr>
<tr>
<td>Not so good</td>
<td>testAddDays1(), testAddDays2()</td>
</tr>
<tr>
<td>Bad</td>
<td>test1(), test2()</td>
</tr>
<tr>
<td>Overkill</td>
<td>TestAddDaysOneDayAndThenFiveDaysStartingOnJanuaryTwentySeventhAndMakeSureItRollsBackToJanuaryAfterRollingToFebruary()</td>
</tr>
</tbody>
</table>

Let's put it all together!

public class DateTest {

    ... 

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }

    ...
Let's put it all together!

```java
public class DateTest {

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
```

#2: Keep tests small

- Ideally, test one thing at a time
  - "Thing" usually means one method under one input condition
  - Not always possible – but if you test `x()` using `y()`, try to test `y()` in isolation in another test
- Low-granularity tests help you isolate bugs
  - Tell you exactly what failed and what didn't
- Only a few (likely one) assert statements per test
  - Test halts after first failed assertion
  - Don't know whether later assertions would have failed

#3: Be thorough

- Consider each equivalence class
  - Items in a collection: none, one, many
- Consider common input categories
  - `Math.abs()`: negative, zero, positive values
- Consider boundary cases
  - Inputs on the boundary between equivalence classes
    - `Person.isMinor()`: age < 18, age == 18, age > 18
- Consider edge cases
  - -1, 0, 1, empty list, arr.length, arr.length-1
- Consider error cases
  - Empty list, null object

Other Guidelines

- Test all methods
  - Constructors are exception to the rule
- Keep tests simple
  - Minimize `if/else, loops, switch, etc`
  - Don't want to debug your tests!
- Tests should always have at least one assert
  - Unless testing that an exception is thrown
  - Testing that an exception is not thrown is unnecessary
    - `assertTrue(true)` doesn't count!
- Tests should be isolated
  - Not dependent on side effects of other tests
  - Should be able to run in any order

JUnit Summary

- Tests need `failure atomicity` so we know exactly what failed
  - Each test should have a descriptive name
  - Assert methods should have clear messages to know what failed
  - Write many small tests, not one big test
- Test for expected errors / exceptions
- Choose a descriptive assert method, not always `assertTrue`
- Choose representative test cases from equivalent input classes
- Avoid complex logic in test methods if possible
- Use helpers, `@Before` to reduce redundancy between tests

External Testing

- This is for your client (us!)
- HW5 and on, class specifications are no longer provided; everyone's might be different
- So how do we test your code?
Test Script Language

- Text file with one command listed per line
- First word is always the command name
- Remaining words are arguments
- Commands will be translated to some method(s) in your code

```java
CheckRep: Linked List

public void insert(element e) {
    checkRep();
    // Do things!
    checkRep();
}

private void checkRep() {
    // Checks invariants: ordered linked list dependency
    check list not null quick!
    if(Thisprogram.DEBUG)
        check list elements order
    check list first.prev is null quick!
    ...
}
```
CheckRep

- When should DEBUG = true?
  - While debugging
- When should DEBUG = false?
  - When submitting code
  - Or, whenever speed is important
    - Performance tests (not for 331!)
    - HW6 grading timeouts
  - While coding, sometimes (you don’t want to wait thirty minutes to test every time you change code)