HW Server Survey: http://goo.gl/forms/k6ReIdoPNG

Or just go to the Announcements page, the link is there

As a thank you for filling up the survey, we give an extra late day to use on HW4 or later

Testing, cont., Equality and Identity

Based on material by Michael Ernst, University of Washington

Announcements

- HW4 is due October 20th
  - LONG!!!
  - Several tasks:
    1. design the ADT following the ADT methodology (Mutable vs. immutable
    2. Operations (creators, mutators, etc.) + their specs
    3. write a test suite based on those specs (before coding, test-first principle)
    4. then code
    5. then augment test suite based on actual implementation and measure coverage
  - When done, Submit in Homework Server

Outline

- Testing
  - Strategies for choosing tests
    - Black box testing
    - White box testing
  - Equality and identity
  - Quiz 5 at the end of class

Why Is Testing So Hard?

```
// requires: 1 <= x,y,z <= 10000
// returns: computes some f(x,y,z)
int proc(int x, int y, int z)
```

- Exhaustive testing would require 1 trillion runs! And this is a trivially small problem
- The key problem: choosing test suite
  - Small enough to finish quickly
  - Large enough to validate program

Testing Strategies

- Test case: specifies
  - Inputs + pre-test state of the software
  - Expected result (outputs and post-test state)
- Black box testing:
  - We ignore the code of the program. We look at the specification (roughly, given some input, was the produced output correct according to the spec?)
  - Choose inputs without looking at the code
- White box (clear box, glass box) testing:
  - We use knowledge of the code of the program (roughly, we write tests to “cover” internal paths)
  - Choose inputs with knowledge of implementation
Black Box Testing Heuristics

- Paths in specification
  - A form of equivalence partitioning
- Equivalence partitioning
- Boundary value analysis
  - Arithmetic (Inputs: smallest/largest values)
  - Objects (Inputs: null objects, circular list, aliasing)

White Box Testing

- Ensure test suite covers (covers means executes) "all of the program"
- Measure quality of test suite with % coverage
- Assumption: high coverage implies few remaining errors in program
- Focus: features in code not described in specification
  - Performance optimizations
  - Alternate algorithms (paths) for different cases

White Box Testing: Control-flow-based Testing

- Control-flow-based white box testing:
  - Extract a control flow graph (CFG)
  - Test suite must cover (execute) certain elements of this control flow graph
- Idea: Define a coverage target and ensure test suite covers target
  - Targets: CFG nodes, branch edges, paths, etc.
  - Coverage target approximates "all of the program"

Aside: Control Flow Graph (CFG)

- Assignment \( x = y + z \) \( \Rightarrow \) node in CFG: \( x = y + z \)
- If-then-else
  - if (b) S1 else S2 \( \Rightarrow \)
  - CFG for S1
  - CFG for S2
  - end-if

Aside: Control Flow Graph (CFG)

- Loop
  - while (b) S \( \Rightarrow \)
  - CFG for S
Aside: Control Flow Graph (CFG)

Draw the CFG for this code:

```java
1 s = 0;
x = 0;
2 while (x<y) {
3     x = x+3;
y = y+2;
4     if (x+y<10)
5         s = s+x+y;
6     else
7         s = s+x-y;
8 }
9 res = s;
```

Statement Coverage

- Traditional target: statement coverage. Write test suite that covers all statements, or in other words, all nodes in the CFG

- Intuition: code that has never been executed during testing may contain errors
  - Often this is the "low-probability" code

Example

- Suppose that we run two test cases
  - Test case #1: follows path 1-2-exit (e.g., we never take the loop)
  - Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)

- Problems?

Branch Coverage

- Target: write test cases that cover all branch edges at predicate nodes
  - True and false branch edges of each if-then-else
  - The two branch edges corresponding to the condition of a loop
  - All alternatives in a SWITCH statement
  - In modern languages, branch coverage implies statement coverage!
    - i.e., a test suite that achieves 100% branch coverage achieve 100% statement coverage

Example

- We need to cover the red branch edges
  - Test case #1: follows path 1-2-exit
  - Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit

- What is % branch coverage?
Example

```java
static int min(int a, int b) {
    int r = a;
    if (a <= b)
        r = a;
    return r;
}
```

- Let’s test with `min(1,2)`
- What is the statement coverage?
- What is the branch coverage?

Code Coverage in Eclipse

Other White Box Heuristics

- White box equivalence partitioning and boundary value analysis
- Loop testing
  - Skip loop
  - Run loop once
  - Run loop twice
  - Run loop with typical value
  - Run loop with max number of iterations
  - Run with boundary values near loop exit condition
- Branch testing
  - Run with values at the boundaries of branch condition

Outline

- Reference equality
- “Value” equality with `.equals`
- Equality and inheritance
- `.equals` and `.hashCode`
- Equality and mutation
- Implementing `.equals` and `.hashCode` efficiently
- Equality in ADTs

Equality

- Simple idea:
  - 2 objects are equal if they have the same value
- Many subtleties
  - Same reference, or same value?
  - Same rep or same abstract value?
    - Remember the HW3 questions
  - Equality in the presence of inheritance?
  - Does equality hold just now or is it eternal?
  - How can we implement equality efficiently?

Equality: `==` and `.equals`

```java
class Point {
    int x; // x-coordinate
    int y; // y-coordinate
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public boolean equals(Point other) {
        return this.x == other.x && this.y == other.y;
    }
}
```

Java uses the reference model for class types

- `a == b`?
- `true or false? a.equals(b)`?
- `true or false? b.equals(c)`?
Equality: \(==\) and \(\text{equals}\)

- In Java, \(==\) tests for reference equality. This is the strongest form of equality.
- Usually we need a weaker form of equality, value equality.
- In our Point example, we want \(a\) to be “equal” to \(b\) because the \(a\) and \(b\) objects hold the same value.
- Need to override \(\text{Object.equals()}\).

Properties of Equality

- Equality is an equivalence relation.
  - Reflexive \(a.\text{equals}(a)\)
  - Symmetric \(a.\text{equals}(b) \iff b.\text{equals}(a)\)
  - Transitive \(a.\text{equals}(b) \land b.\text{equals}(c) \implies a.\text{equals}(c)\)

- Is reference equality an equivalence relation?
  - Yes

Object.equals method

- \(\text{Object.equals}\) is very simple:

```java
public class Object {
    public boolean equals(Object obj) {
        return this == obj;
    }
}
```

Object.equals Javadoc spec

Indicates whether some other object is “equal to” this one. The \(\text{equals}\) method implements an equivalence relation:

- It is reflexive: for any non-null reference value \(x\), \(x.\text{equals}(x)\) should return true.
- It is symmetric: for any non-null reference values \(x\) and \(y\), \(x.\text{equals}(y)\) should return true if and only if \(y.\text{equals}(x)\) returns true.
- It is transitive: for any non-null reference values \(x, y,\) and \(z\), if \(x.\text{equals}(y)\) returns true and \(y.\text{equals}(z)\) returns true, then \(x.\text{equals}(z)\) should return true.
- It is consistent: for any non-null reference values \(x\) and \(y\), multiple invocations of \(x.\text{equals}(y)\) consistently return true or consistently return false, provided no information used in \(\text{equals}\) comparisons on the objects is modified.

Parameters:
- \(\text{obj}\) - the reference object with which to compare.

Returns:
- true if this object is the same as the \(\text{obj}\) argument;
- false otherwise.

See Also:
- \(\text{hashCode()}, \text{HashMap}\)
Adding equals

```java
public class Duration {
    private final int min;
    private final int sec;
    public Duration(int min, int sec) {
        this.min = min;
        this.sec = sec;
    }
    Duration d1 = new Duration(10,5);
    Duration d2 = new Duration(10,5);
    System.out.println(d1.equals(d2)); // prints?
}
```

First Attempt to Add equals

```java
public class Duration {
    public boolean equals(Duration d) {
        return this.min == d.min && this.sec == d.sec;
    }
    Duration d1 = new Duration(10,5);
    Duration d2 = new Duration(10,5);
    System.out.println(d1.equals(d2)); Yields what?
}
```

What About This?

```java
public class Duration {
    public boolean equals(Duration d) {
        return this.min == d.min && this.sec == d.sec;
    }
    Object d1 = new Duration(10,5);
    Object d2 = new Duration(10,5);
    System.out.println(d1.equals(d2)); Yields what?
}
```

A Correct equals

```java
@Override
public boolean equals(Object o) {
    if (! (o instanceof Duration) )
        return false;
    Duration d = (Duration) o;
    return this.min == d.min && this.sec == d.sec;
}
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2)); Yields what?
```

Outline

- Reference equality
- “Value” equality with .equals
- Equality and inheritance
- equals and hashCode
- Equality and mutation
- Implementing equals and hashCode efficiently
- Equality and ADTs

Add a Nano-second Field

```java
public class NanoDuration extends Duration {
    private final int nano;
    public NanoDuration(int min, int sec, int nano){
        this.super(min, sec); // initializes min&sec
        this.nano = nano;
    }
}
```

What if we don’t add NanoDuration.equals? (Assume Duration.equals as in slide 34)

Fall 15 CSCI 2600, A Milanova (example due to Michael Ernst)
First Attempt at `NanoDuration.equals`

```java
public boolean equals(Object o) {
    if (! (o instanceof NanoDuration) )
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nd.nano == nano;
}
```

Duration `d1 = new NanoDuration(5,10,15)`;
Duration `d2 = new Duration(5,10)`;
`d1.equals(d2)` Yields what?
`d2.equals(d1)` Yields what?

Possible Fix for `NanoDuration.equals`

```java
public boolean equals(Object o) {
    if (! (o instanceof Duration) )
        return false;
    if (! (o instanceof NanoDuration) )
        return super.equals(o); // compare without nano
    NanoDuration nd = (NanoDuration) o;
    return super.equals(o) && nd.nano == nano;
}
```

- Does it fix the symmetry bug?
- What can go wrong?

Possible Fix for `NanoDuration.equals`

Duration `d1 = new NanoDuration(10,5,15)`;
Duration `d2 = new Duration(10,5)`;
Duration `d3 = new NanoDuration(10,5,30)`;
`d1.equals(d2)` Yields what? `d1` -- 10 5 15
`d2.equals(d3)` Yields what? `d2` -- 10 5
`d1.equals(d3)` Yields what? `d3` -- 10 5 30

`equals` is not transitive!

One Solution: Checking Exact Class, Instead of `instanceof`

```java
class Duration {
    public boolean equals(Object o) {
        if (o == null) return false;
        if (!o.getClass().equals(getClass()))
            return false;
        Duration d = (Duration) o;
        return d.min == min && d.sec == sec;
    }
}
```

- Problem: every subclass must implement `equals`;
sometimes, we want to compare distinct classes!

Another Solution: Composition

```java
public class NanoDuration {
    private final Duration duration;
    private final int nano;
    // ...
}
```

Composition does solve the `equals` problem:
Duration and NanoDuration are now unrelated, so we'll never compare a Duration to a NanoDuration.
Problem: Can't use NanoDuration instead of Duration. Can't reuse code written for Duration.

A Reason to Avoid Subclassing Concrete Classes. More later

- In the JDK, subclassing of concrete classes is rare. When it happens, there are problems

- One example: `Timestamp extends Date`
  - Extends Date with a nanosecond value
  - But Timestamp spec lists several caveats
    - E.g., `Timestamp.equals(Object)` method is not symmetric with respect to `Date.equals(Object)`
    (the symmetry problem we saw on the previous slide)
Abstract Classes
- Prefer subclassing abstract classes
  - Just like in real life, “Superclasses” in real life cannot be instantiated
- There is no equality problem if superclass cannot be instantiated!
  - E.g., if Duration were abstract, the issue of comparing Duration and NanoDuration never arises

Outline
- Reference equality
- “Value” equality with .equals
- Equality and inheritance
- equals and hashCode
- Equality and mutation
- Implementing equals and hashCode efficiently
- Equality and ADTs

The int hashCode Method
- hashCode computes an index for the object (to be used in hashtables)
- Javadoc for Object.hashCode():
  - “Returns a hash code value of the object. This method is supported for the benefit of hashtables such as those provided by HashMap.”
- Self-consistent: o.hashCode() == o.hashCode() ...
  as long as o does not change between the calls
- Consistent with equals() method: a.equals(b) => a.hashCode() == b.hashCode()

The Object.hashCode Method
- Object.hashCode’s implementation returns a distinct integer for each distinct object, typically by converting the object’s address into an integer
- hashCode must be consistent with equality
  - equals and hashCode are used in hashtables
  - If hashCode is inconsistent with equals, the hashtable behaves incorrectly
- Rule: if you override equals, override hashCode; must be consistent with equals

Implementations of hashCode
Remember, we defined Duration.equals(Object)
public class Duration {
  Choice 1: don’t override, inherit hashCode from Object
  Choice 2: public int hashCode() { return 1; }
  Choice 3: public int hashCode() { return min; }
  Choice 4: public int hashCode() { return min+sec; }
}

hashCode Must Be Consistent with equals
- Suppose we change Duration.equals
  // Returns true if o and this represent the same number of // seconds
  public boolean equals(Object o) {
    if (!(o instanceof Duration)) return false;
    Duration d = (Duration) o;
    return 60*min+sec == 60*d.min+d.sec;
  }
- Will min+sec for hashCode still work?
Outline of today's class

- Reference equality
- “Value” equality with `.equals`
- Equality and inheritance
- `equals` and `hashCode`
- Equality and mutation
- Implementing `equals` and `hashCode` efficiently
- Equality and ADTs

Equality, Mutation and Time

- If two objects are equal now, will they always be equal?
- In mathematics, the answer is “yes”
- In Java, the answer is “you chose”
- The Object spec does not specify this
- For immutable objects
  - Abstract value never changes, equality is eternal
- For mutable objects
  - We can either compare abstract values now, or
  - be eternal (can’t have both since value can change)

StringBuffer Example

- StringBuffer is mutable, and takes the eternal approach
  - StringBuffer s1 = new StringBuffer("hello");
  - StringBuffer s2 = new StringBuffer("hello");
  - System.out.println(s1.equals(s1)); // true
  - System.out.println(s1.equals(s2)); // false
- `equals` is just reference equality (==). This is the only way to ensure eternal equality for mutable objects

Date Example

- Date is mutable, and takes the “compare values now” approach
  - Date d1 = new Date(0); // Jan 1, 1970 00:00:00 GMT
  - Date d2 = new Date(0);
  - System.out.println(d1.equals(d2)); // true
  - d2.setTime(1); // a millisecond later
  - System.out.println(d1.equals(d2)); // false

Behavioral and Observational Equivalence

- Two objects are “behaviorally equivalent” if there is no sequence of operations that can distinguish them
- Two objects are “observationally equivalent” if there is no sequence of observer operations that can distinguish them
  - We are excluding mutators
  - Excluding `==`

Equality and Mutation

- We can violate rep invariant of a Set container (rep invariant: there are no duplicates in set) by mutating after insertion
  - Set<Date> s = new HashSet<Date>();
  - Date d1 = new Date(0);
  - Date d2 = new Date(1);
  - s.add(d1);
  - s.add(d2);
  - d2.setTime(0); // mutation after d2 already in the Set!
  - for (Date d : s) { System.out.println(d); }

Fall 15 CSCI 2600, A Milanova
Equality and Mutation

- Be very careful with elements of Sets
- Ideally, elements will be immutable objects, because immutable objects guarantee behavioral equivalence
- Java spec for Sets warns about using mutable objects as set elements

Same problem applies to keys in maps

Equality and Mutation

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Sets assume hash codes don’t change

Equality and Mutation

Redefining equals and hashCode makes most sense for immutable, “value”, objects

E.g., String, RatNum

Be careful with equals and hashCode on mutable objects

From spec of Object.equals: It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

Implementing equals Efficiently

equals can be expensive!

How can we speed-up equals?

public boolean equals(Object o) {
    if (this == o) return true;
    // class-specific prefiltering (e.g.,
    // compare file size if working with files)
    // Lastly, compare fields (can be expensive)
    return true;
}

Example: A Naive RatPoly.equals

public boolean equals(Object o) {
    if (o instanceof RatPoly) {
        RatPoly rp = (RatPoly) o;
        int i=0;
        while (i<Math.min(rp.c.length,c.length)) {
            if (rp.c[i] != c[i]) // Assume int arrays
                return false;
            i = i+1;
        }
        if (i != rp.c.length || i != c.length) return false;
        return true;
    } else
        return false;
}

Example: Better equals

public boolean equals(Object o) {
    if (o instanceof RatPoly) {
        RatPoly rp = (RatPoly) o;
        if (rp.c.length != c.length)
            return false; // prefiltering
        for (int i=0; i < c.length; i++) {
            if (rp.c[i] != c[i])
                return false;
        }
        return true;
    } else
        return false;
}
Implementing `hashCode`

```java
// returns: the hashCode value of this String
public int hashCode() {
    int h = this.hash; // rep. field hash
    if (h == 0) { // caches the hashcode
        char[] val = value;
        int len = count;
        for (int i = 0; i < len; i++) {
            h = 31*h + val[i];
        }
        this.hash = h;
    }
    return h;
}
```

This works only for immutable objects!

Outline of today’s class

- Reference equality
- “Value” equality with `.equals`
- Equality and inheritance
- `equals` and `hashCode`
- Equality and mutation
- Implementing `equals` and `hashCode` efficiently
- Equality and ADTs

Rep Invariant, AF and Equality

- With ADTs we compare abstract values, not rep
- Usually, many valid reps map to the same abstract value
  - If Concrete Object (rep) and Concrete Object’ (rep’) map to the same Abstract Value, then Concrete Object and Concrete Object’ must be equal
- A stronger rep invariant shrinks the domain of the AF and simplifies `.equals`

```
class LineSegment {
    // Rep invariant:
    // x1<x2  
    float x1,y1;
    float x2,y2;
    ...
}
```

```
class LineSegment {
    // Rep invariant:
    // !(x1=x2 && y1=y2)  
    // x1<x2 ||
    float x1,y1;
    float x2,y2;
    ...
}
```

```
class LineSegment {
    // equals must  
    // return true for  
    // equals is simpler:
    // {x1:1,y1:2,x2:4,y2:5}  
    // {4,5,1,2} is not  
    // and {4,5,1,2}  
    // valid rep anymore
}
```

Example: Line Segment

Other Examples

- RatNum
- RatPoly