Principles of Software Construction: Objects, Design, and Concurrency

Specification, Testing, and Builds

Jonathan Aldrich    Charlie Garrod
Administrivia

- Homework 0 due at 11:59pm tonight
- Document summarizing Java basics posted
Review: Steps in the Design Process

- Precondition: understand functional requirements
  - Pre- and post-condition specifications for IntSet
- Precondition: understand quality attribute requirements
- Design a logical architecture
- Design a behavioral model
- Responsibility assignment
- Interface design
- Algorithm and data structure design
- Writing code
- ...
Review: In the Previous Lecture…

- We broke the Tree Simulation design process into steps

- We studied an example of the last step, detailed design
  - A data structure for mathematical sets

- We discussed a number of design principles
  Can you name the benefits, and how achieved?
  - Representation hiding →
    - Object-oriented interfaces →
    - Abstract data types →
Review: In the Previous Lecture…

• We broke the Tree Simulation design process into steps

• We studied an example of the last step, detailed design
  ▪ A data structure for mathematical sets

• We discussed a number of design principles
  Can you name the benefits, and how achieved?
  ▪ Representation hiding → ease of changing representation
    • Mechanisms: public vs. private, constructors

  ▪ Object-oriented interfaces → interoperability of implementations
    • Use interface as type, especially for binary methods
    • Dispatch provides interoperability

  ▪ Abstract data types → performance from shared representation
    • Using a class type instead of an interface type, esp. in binary methods
    • The type tells you the representation – can access private fields
    • Tradeoff: lose interoperability advantages
Steps in the Design Process

• Precondition: understand functional requirements
  ▪ Pre- and post-condition specifications for IntSet

• Precondition: understand quality attribute requirements

• Design a logical architecture

• Design a behavioral model

• Responsibility assignment

• Interface design

• Algorithm and data structure design

• Writing code

• … What goes here?
Steps in the Design Development Process

- Precondition: understand functional requirements
  - Pre- and post-condition specifications for IntSet
- Precondition: understand quality attribute requirements
- Design a logical architecture
- Design a behavioral model
- Responsibility assignment
- Interface design
- Algorithm and data structure design
- Writing code

- Testing code
  - Unit testing, JUnit, Coverage, EclEmma
- Automated builds and continuous integration
  - Ant and TravisCI

Note – possibly before writing code!
This lecture

- 214: managing complexity, from programs to systems
  - Threads and concurrency
  - Object-oriented programming
  - Analysis and modeling
  - Design
Today’s Lecture: Learning Goals

- Review principles for detailed design
- Basic specification concepts (review from 15-122)
  - preconditions and postconditions
- Testing principles and practices
  - design effective unit test suites
  - write tests in JUnit
- Specification and code coverage
  - how they differ
  - what each is useful for
  - evaluate of code coverage using EclEmma
- Continuous integration
  - benefits in software development
  - use ant and TravisCI
This is a bug
Is this a bug?

graph.getDistance(rachel, kramer);

> -1
Is this a bug?  **NO!**

```java
class Graph {
    /** @return the distance between p1 and p2 * in the graph. Returns -1 if p1 and p2 * are unconnected. */
    void getDistance(Person p1, Person p2);
}

graph.getDistance(rachel, kramer);

> -1
```

We need **specifications** to determine whether or not code behaves correctly.
Specifications

- **Contain**
  - Functional behavior
  - Erroneous behavior
  - Quality attributes

- **Desirable attributes**
  - Complete
    - Does not leave out any desired behavior
  - Minimal
    - Does not require anything that the user does not care about
  - Unambiguous
    - Fully specifies what the system should do in every case the user cares about
  - Consistent
    - Does not have internal contradictions
  - Testable
    - Feasible to objectively evaluate
  - Correct
    - Represents what the end-user(s) need
A Real Specification

/**
 * Returns the correctly rounded positive square root of a double value.
 * 
 * Special cases:
 * - If the argument is NaN or less than zero, then the result is NaN.
 * - If the argument is positive infinity, then the result is positive infinity.
 * - If the argument is positive zero or negative zero, then the result is the same as the argument.
 * Otherwise, the result is the double value closest to the true mathematical square root of the argument value.
 *
 * @param a a value.
 * @return the positive square root of a. If the argument is NaN or less than zero, the result is NaN.
 */

public static double sqrt(double a) { ...}
Function Specifications

• A function’s contract is a statement of the responsibilities of that function, and the responsibilities of the code that calls it.
  ▪ Analogy: legal contracts
    ▪ If you pay me $30,000
    ▪ I will build a new room on your house
  ▪ Helps to pinpoint responsibility

• Contract structure
  ▪ Precondition: condition the function relies on for correct operation
  ▪ Postcondition: condition the function establishes after running

• (Functional) correctness with respect to the specification
  ▪ If the client of a function fulfills the function’s precondition, the function will execute to completion and when it terminates, the postcondition will be fulfilled

• What does the implementation have to fulfill if the client violates the precondition?
  ▪ A: nothing at all!
  ▪ In practice we often want to specify what happens on error inputs
interface IntSet {
    /** precondition:
    *   postcondition:
    */
    IntSet union(IntSet s);

    /** precondition:
    *   postcondition:
    */
    boolean contains(int i);
}
interface IntSet {
  /** precondition: s is not null
   *  postcondition: returns an IntSet that contains an
   *      element i iff when this or s does
   */
  IntSet union(IntSet s);

  /** precondition: none  ("true" logically)
   *  postcondition: returns true iff i is in the set
   */
  boolean contains(int i);
}
Specifying IntSet (on your own – one solution at the end)

```java
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int i);

    /** precondition:
     *  * postcondition:
     *  *
     */
    boolean isSubsetOf(IntSet s);
}
```
Testing

• Executing the program with selected inputs in a controlled environment

• Goals:
  ▪ Reveal bugs (main goal)
  ▪ Assess quality (hard to quantify)
  ▪ Clarify the specification, documentation
  ▪ Verify contracts

"Testing shows the presence, not the absence of bugs"

Edsger W. Dijkstra 1969
Testing Decisions

- **What to test?**
  - Functional correctness of each method? System behavior? UI?

- **Who tests?**
  - Developers? QA team?

- **When to test?**
  - Before coding? After writing each method? Before shipping?

- **Manual or automated?**
  - Ability to test anything? Ability to repeat tests?

- **When to stop testing?**
  - When all functionality is tested? When all code is tested? When we run out of time or money?
Guidelines for Designing Test Suites

• Write a test for each case in the specification
  ▪ Representative classes of input
  ▪ Invalid classes of input

• Write tests for boundary conditions
  ▪ Off-by-one errors are common

• Write tests for difficult situations
  ▪ Stress tests (extreme input)
  ▪ Situations that require complex reasoning

• Write tests that exercise all of the code
  ▪ Perhaps interesting paths through the code, too

Black-box testing

White-box testing
(glass-box a better term?)
Example (exercise on paper)

```c
/**
 * computes the sum of the first len values of the array
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the array values
 */
int total(int array[], int len);
```

Guideline Reminder

- Write a test for each case in the specification
  - Representative classes of input
  - Invalid classes of input
- Write tests for boundary conditions
  - Off-by-one errors are common
- Write tests for difficult situations
  - Stress tests (extreme input)
  - Situations that require complex reasoning
Example (possible solution)

```java
/**
 * computes the sum of the first len values of the array
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the array values
 */
int total(int array[], int len);
```

- Test empty array
- Test array of length 1 and 2
- Test negative numbers
- Test invalid length (negative or longer than array.length)
- Test null as array
- Test with a very long array
Exercise (on your own)

• Test a priority queue for Strings

```java
public interface Queue {
    void add(String s);
    String getFirstAlphabetically();
}
```

• Write various kinds of test cases
Unit Tests

• Unit tests for small units: functions, classes, subsystems
  ▪ Smallest testable part of a system
  ▪ Test parts before assembling them
  ▪ Intended to catch local bugs

• Typically written by developers

• Many small, fast-running, independent tests
  ▪ Can run on every check-in, or every compile

• Little dependencies on other system parts or environment

• Insufficient but a good starting point, extra benefits:
  ▪ Documentation (executable specification)
  ▪ Design mechanism (design for testability)
JUnit

- Popular unit-testing framework for Java
- Easy to use
- Integration into Eclipse, Ant, other tools
- Can be used to drive design
  - Testability, incrementally adding functionality
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
    @Test
    public void testSanityTest() {
        Graph g1 = new AdjacencyListGraph(10);
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(true, g1.addVertex(s1));
        assertEquals(true, g1.addVertex(s2));
        assertEquals(true, g1.addEdge(s1, s2));
        assertEquals(s2, g1.getNeighbors(s1)[0]);
    }

    @Test
    public void test.... {
        int helperMethod...
    }

    private int helperMethod...
}
JUnit Demo
assert, Assert

- **assert** is a Java statement form that verifies a condition (if checking is turned on)
  - **assert** expression: "Error Message"

- org.junit.Assert is a library that provides many more specific methods
  - static void **assertTrue**(java.lang.String message, boolean condition)
    // Asserts that a condition is true.
  - static void **assertEquals**(java.lang.String message, long expected, long actual);
    // Asserts that two longs are equal.
  - static void **assertEquals**(double expected, double actual, double delta);
    // Asserts that two doubles are equal to within a positive delta
  - static void **assertNotNull**(java.lang.Object object)
    // Asserts that an object isn't null.
  - static void **fail**(java.lang.String message)
    // Fails a test with the given message.
import org.junit.*;  
import org.junit.Before;  
import static org.junit.Assert.assertEquals;  

public class AdjacencyListTest {  
    Graph g;  

    @Before  
    public void setUp() throws Exception {  
        graph = createTestGraph();  
    }  

    @Test  
    public void testSanityTest() {  
        Vertex s1 = new Vertex("A");  
        Vertex s2 = new Vertex("B");  
        assertEquals(3, g.getDistance(s1, s2));  
    }  
}
Test organization conventions

• A test class CTest for each class C
• Two directories: source and test
  ▪ Store CTest and C in the same package
  ▪ Tests can access members with default (package) visibility
• Alternative: (not in 15-214)
  ▪ Store tests in the source directory but in a separate package
JUnit Operation

- TestCase collects multiple tests (in one class)
- TestSuite collects test cases (typically package)
- Tests are methods without parameters or return values

- Test runner knows how to run JUnit tests
  - (uses reflection to find all methods with @Test annotat.)
Run tests frequently

- Run tests before every commit
  - Committing broken code makes teammates unhappy
  - You will be unhappy too if they call your cell on your vacation

- Run tests before trying to understand unfamiliar code
  - If it’s broken, get someone to fix it first

- What if the test suite takes too long to run?
  - Medium sized projects easily have 1000s of test cases and run for minutes
  - Run a subset ("smoke tests") on every commit
  - Run all tests nightly/weekly/whatever ("Nightly build")

- Build tools and continuous integration servers make this convenient
Build and Test Automation

• Build automation: automatically compile a program
  ▪ May be a multi-step process with dependencies
    • Program A uses library B → compile B first
    • Avoid unnecessary recompilation

• Test automation: run tests automatically after a build

• Many tools
  ▪ make
  ▪ ant
  ▪ gradle
  ▪ maven
  ▪ sbt
  ▪ ...

red = changed, recompile
brown = depends on change, recompile
black = no recompilation needed
<project name="MyProject" default="dist" basedir="."/>
  <property name="src" location="src"/>
  <property name="build" location="build"/>

  <target name="compile"
      description="compile the source">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="test" depends="compile"
      description="run tests">
    <junit printsummary="on" haltonfailure="yes">
      ...
    </junit>
  </target>
</project>
Ant demo
Continuous Integration

- Automation server responds on every commit
  - Compiles code
  - Runs tests
  - Reports errors (web page, email, etc.)

- Benefits
  - Immediate feedback about problems
  - Allows developers to make frequent check-ins
  - Keeps code synchronized

- Example tool: TravisCI
TravisCI Demo
Whitebox Testing: Code Coverage

- Organized according to program decision structure
- Touching: statement, branch

```java
public static int binsrch (int[] a, int key) {
    int low = 0;
    int high = a.length - 1;
    while (true) {
        if ( low > high ) return -(low+1);
        int mid = (low+high) / 2;
        if ( a[mid] < key ) low = mid + 1;
        else if ( a[mid] > key ) high = mid - 1;
        else return mid;
    }
}
```

- Will this statement get executed in a test?
- Does it return the correct result?
- Could this array index be out of bounds?
- Does this return statement ever get reached?
Method Coverage

- Trying to execute each method as part of at least one test

```java
public boolean equals(Object anObject) {
    if (isZero())
        return ((IMoney)anObject).isZero();
    if (anObject instanceof IMoney)
        return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```

- Does this guarantee correctness?
Statement Coverage

- Trying to execute **every statement** in at least one test

```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money)
        Money aMoney = (Money)anObject;
    return aMoney.currency().equals(currency())
        && amount() == aMoney.amount();
}
return false;
}
```

- Does this guarantee correctness?
Structure of a Method Under Test

```
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```

Flow chart diagram for junit.samples.money.Money.equals
Statement Coverage

- **Statement coverage**
  - What portion of program statements (nodes) are touched by test cases

- **Advantages**
  - Coverage easily assessed
  - Test suite size linear in size of code

- **Issues**
  - May not exercise code in enough interesting situations

```java
public boolean equals(Object anotherObject) {
    if (anotherObject == zero) {
        if (anotherObject instanceof IMoney)
            return (IMoney)anotherObject.instanceOf().equals().isZero();
    }
    if (anotherObject instanceof Money) {
        Money anotherMoney = (Money)anotherObject;
        return anotherMoney.currency().equals().equals(currency());
    }
    return false;
}
```
Branch Coverage

- **Branch coverage**
  - What portion of **condition branches** are covered by test cases?
    - Consider true and false branches at each choice point

- **Advantages**
  - Coverage easily assessed
  - Test suite size and content derived from structure of control flow

- **Issues**
  - More tests than statement coverage
  - Still may not exercise enough interesting situations
Path Coverage

- **Path coverage**
  - What portion of all possible paths through the program are covered by tests?
    - For loops: must limit to a finite set of iterations (e.g. 0, 1, 2, N)

- **Advantages**
  - Considers all logical combinations

- **Issues**
  - Combinatorial explosion of paths
  - Not necessarily worth the extra tests

```java
public boolean equals(Object anObject) {
    if (anObject instanceof IMoney) {
        return !(anObject instanceof IMoney); // Use instanceof instead of equals
    } else if (anObject instanceof Money) {
        Money mMoney = (Money) anObject;
        return mMoney.currency.equals(currency());
        if (amount() == mMoney.amount());
    } else return false;
}
```
Test Coverage Tooling

- Coverage assessment tools
  - Track execution of code by test cases
- Count visits to statements
  - Develop reports with respect to specific coverage criteria
  - Instruction coverage, line coverage, branch coverage
- Example: EclEmma tool for JUnit tests
EclEmma Demo
Exercise (on your own)

- Write test cases to achieve 100% line coverage but **not** 100% branch coverage

```c
void foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```
“Coverage” is useful but also dangerous

- Examples of what coverage analysis could miss
  - Unusual paths
  - Missing code
  - Incorrect boundary values
  - Timing problems
  - Configuration issues
  - Data/memory corruption bugs
  - Usability problems
  - Customer requirements issues

- Coverage is not a good adequacy criterion
  - Instead, use to find places where testing is inadequate
Toad’s Take-Home Messages

- **Specifications**
  - Defined in terms of preconditions and postconditions
  - Necessary for determining correctness, and reusing code

- **Testing**
  - Finds bugs, but cannot prove their absence
  - Useful as documentation
  - JUnit is a practical tool

- **Build and Test Automation**
  - Runs tests often and automatically
  - Enables finding bugs more quickly

- **Black box: coverage of the specification**
  - The core of good testing practice – but must be done well
  - Representative, boundary, error, and extreme cases

- **White box: coverage of code**
  - Good for finding untested functionality
  - Not an indication that a test suite is adequate
Specifying IntSet *(exercise solution)*

```java
interface IntSet {
    IntSet union(IntSet s);

    boolean contains(int i);

    /** precondition: s is not null
     *   postcondition: returns true iff for every integer i
     *                  such that this.contains(i), we have s.contains(i)
     */
    boolean isSubsetOf(IntSet s);
}
```