Course notes for CS220 are maintained in this file. They will be uploaded after classes, about once per week.

1 Introduction

The Software Design sequence

- Design algorithms
  - High-level problem-solving skills
- Implement algorithms as programs
  - Java - A modern programming language
  - Organize data and instructions
  - In both algorithms and programs, we must use low-level and precise logic
    - No ambiguity allowed
  - Debugging and testing
- Understand what programs will do
- Communicate technical information about your programs
- Learn how to operate as a technical professional
Software Design I

The six things a program can do

1. Get input
2. Give output
3. Do arithmetic
4. Update a stored value
5. Test a condition, and select an alternative
6. Repeat a group of actions

Four ways Java will help you organize your work

1. Grouping related data together
2. Defining sequences of operations
3. Associating data with operations relevant to the particular data
4. Naming these groups, sequences and associations for easy and repeated use

Software Design II

• A deeper look at inheritance and object-oriented design
• Exceptions
• Recursion
• Linear (list) data structures
• Describing and tracing the effects of programs
• Programming with files and directories
• Multi-dimensional arrays
• Debugging and testing
  – Knowing what "correct" means, and how to tell if your code meets that standard
  – This class will have a strong focus on test-driven development
• Skills-based class
• Many things we learn will build on what we’ve already studied
  – Including all of Software Design I
• Practice is essential
  – Expect to work on CS220 every day
Assignments
Each assignment is...

• Posted to the course website

• Submitted through AutoLab
  –  https://euryale.cs.uwlax.edu/courses/cs220-fa17-jmaraist/
  –  Remember the campus VPN vpn.uwlax.edu

• You should have received an email with account information on Monday
  –  Check your spam filter
  –  If you registered on Friday or after, I may not have your info
  –  Email by 2pm if you still need one, and I’ll make additional accounts this afternoon

• Tomorrow’s lab will step you through a first AutoLab submission

Textbooks
There is no required text for this class, but you may be happier with a reference of some sort and/or a source of practice exercises. Some options:

• The CS120 online book, *Programming in Java*, zyBooks
  –  Can print sections/chapters as well
  –  Subscribe:
    ∗  Sign up at zyBooks.com
    ∗  Enter zyBook code: UWLAXCS220MaraistFall2017
    ∗  Click Subscribe

• *Java: A Beginner’s Guide*, Herbert Schildt, Oracle Press
  –  Several faculty recommend this book as a second reference for CS120

• O’Reilly has been reputable for reference books

2 JUnit and test-driven development

JUnit

• For specifying and running functional tests in Java

• A separate test for every method
  –  Use Java annotations to mark the test

• The JUnit executable finds test methods, runs them, reports the results

• Eclipse will alert you to test failures
JUnit example test subject
From the JUnit wiki:

- A simple class

```java
public class Calculator {
    public int evaluate(final String expression) {
        int sum = 0;
        for (String summand: expression.split("\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
}
```

JUnit example test class

- Test Calculator with

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class CalculatorTest {
    @Test public void evaluatesExpression() {
        final Calculator calculator = new Calculator();
        final int sum = calculator.evaluate("1+2+3");
        assertEquals(6, sum);
    }
}
```

- The @Test annotation: how JUnit finds tests

- Method takes no parameters

- The class holding the tests has a zero-argument constructor
  - Which is implicitly present if we give no constructor at all

- Assertions
  - Methods provided by JUnit for asserting things which should be true
  - Parameter order: description, expected value, then actual value
    * Always give the description!
  - For real-valued tests, an additional tolerance parameter `delta`
  - Pointer equality vs. `equals`
JUnit Assert JavaDoc


org.junit

Class Assert

java.lang.Object

extends Object

public class Assert

extends Object

A set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly: Assert.assertEquals(...), however, they read better if they are referenced through static import:

import static org.junit.Assert.*;
...
  assertEquals(...);

JUnit Assert methods


| static void | assertEquals(String message, double expected, double actual, double delta) | Asserts that two doubles are equal to within a positive delta. |
| static void | assertEquals(String message, float expected, float actual, float delta) | Asserts that two floats are equal to within a positive delta. |
| static void | assertEquals(String message, long expected, long actual) | Asserts that two longs are equal. |
| static void | assertEquals(String message, Object[] expecteds, Object[] actuals) | Deprecated. use assertArrayEquals |
| static void | assertEquals(String message, Object expected, Object actual) | Asserts that two objects are equal. |

JUnit Assert methods


| static void | assertEquals (Object object) | Asserts that an object is null. |
| static void | assertNotNull (Object message, Object object) | Asserts that an object is null. |
| static void | assertNull (Object expected, Object actual) | Asserts that two objects refer to the same object. |
| static void | assertNotNull (String message, Object expected, Object actual) | Asserts that two objects refer to the same object. |

JUnit and exceptions

Can also specify tests which we expect to fail
• The Java API says that `ArrayList.get(0)` will throw an `IndexOutOfBoundsException` when the list is empty.

• We verify this behavior with:

```java
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

• Documented on Test JavaDoc page `junit.org/junit4/javadoc/latest/org/junit/Test.html`

**ArrayList.get exceptions**

```java
public E get(int index)
```

Returns the element at the specified position in this list.

**Specified by:**
`get` in interface `List<E>`

**Specified by:**
`get` in class `AbstractList<E>`

**Parameters:**
index - index of the element to return

**Returns:**
the element at the specified position in this list

**Throws:**
`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())

**Test annotation arguments**

From `junit.org/junit4/javadoc/latest/org/junit/Test.html`,
Test-driven development

In this class we will learn the discipline of test-driven development (TDD)

- When coding, repeat the following steps:
  1. Write a failing test case
  2. Get it to compile
  3. Get it to pass
  4. Simplify and remove duplication
- Once we have a test passing, we make sure that it keeps on passing

TDD Example

- Working on a financial application
- Need a class to make decisions as to whether certain commodities should be traded
  - A very mathematical question, so we’ll need to support various operations
- Specifically, we need to calculate the first statistical moment about a particular point
  - Don’t have a method for it yet
  - But our stats experts gave us a simple example for this test:

```java
@Test public void testFirstMoment() {
    final InstrumentCalculator calc = new InstrumentCalculator();
    calc.addElement(1.0);
    calc.addElement(2.0);
    assertEquals(7,
```
"First moment about 2.0 for {1.0,2.0} within tolerance", -0.5, calc.firstMomentAbout(2.0), TOLERANCE);

Making testFirstMoment compile

- This won’t even compile right now
  - We might already have a class InstrumentCalculator, but we haven’t written firstMomentAbout yet!
  - So we add a stub for the method
- We don’t want to pass by coincidence, so we make it return an absurd value

```java
public class InstrumentCalculator {
    // ... keeping what’s already here

    public double firstMomentAbout(double point) {
        return Double.NaN;
    }
}
```

- In our projects, we’ll usually throw an exception

Making testFirstMoment pass

- The algorithm for calculating the first moment is standard, so we look it up and implement it

```java
public double firstMomentAbout(double point) {
    double numerator = 0.0;
    for(final double element : getElements()) {
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Write another failing test case

- The code we just added makes one test pass, but it’s not hard to conceive of cases which will fail
- There’s a division in the algorithm: are we safe against division by zero?
  - And what should happen when we call firstMomentAbout() with an empty data set?
- Write another test for this case!

```java
@Test(expected = InvalidBasisException.class)
public void testEmptyFirstMoment() {
    new InstrumentCalculator().firstMomentAbout(0.0);
    fail("Expected InvalidBasisException");
}
```

- fail is another JUnit method
Making `testEmptyFirstMoment` compile

What do we need to do to make `testEmptyFirstMoment` compile?

- If `InvalidBasisException` is not already part of `InstrumentCalculator`'s package, we must create it
- Otherwise it compiles

Making `testEmptyFirstMoment` pass

- We’ll need to throw an `InvalidBasisException` when there are zero elements
- So a revised `firstMomentAbout`:

```java
public double firstMomentAbout(double point) {
    if (getElements().isEmpty()) {
        throw new InvalidBasisException();
    }

    double numerator = 0.0;
    for (final double element : getElements()) {
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

- Now the test passes!

Write a failing test case

- Our next task is to write a routine for the second statistical moment about a point.
- So we write a test for this case. Our stats experts again gave us a simple example:

```java
@Test public void testSecondMoment() {
    final InstrumentCalculator calc = new InstrumentCalculator();
    calc.addElement(1.0);
    calc.addElement(2.0);
    assertEquals(
        "Second moment about 2.0 for {1.0,2.0} within tolerance",
        0.5, calc.secondMomentAbout(2.0), TOLERANCE);
}
```

Making `testSecondMoment` compile

- The problem is again that we do not define the method we are now testing
- And again we make it compile by adding a vacuous definition of the method. We’ll just copy `firstMomentAbout` and change the name:
public double secondMomentAbout(double point) {
    if (getElements().isEmpty()) {
        throw new InvalidBasisException();
    }

    double numerator = 0.0;
    for(final double element : getElements()) {
        numerator += element - point;
    }
    return numerator / elements.size();
}

Making testSecondMoment pass

• Unsurprisingly, the code for the first moment does not satisfy the second moment’s test!

• But the algorithm for the second moment is very similar, and we only need to make one change: from

    numerator += element - point;

    to

    numerator += Math.pow(element - point, 2.0);

• And now it passes!

Remove duplication

• This time around there’s definitely duplication — we have two methods that are almost completely identical!

• In fact, the algorithm for any of the statistical moments has only the same variation that we see here

• So the best way to remove this duplication is with a more general private method nthMomentAbout which the others call

    public double nthMomentAbout(double point, double n) {
        if (getElements().isEmpty()) {
            throw new InvalidBasisException();
        }

        double numerator = 0.0;
        for(final double element : getElements()) {
            numerator += Math.pow(element - point, n);
        }
        return numerator / elements.size();
    }

    public double firstMomentAbout(double point) {
return nthMomentAbout(point, 1.0);
}

public double secondMomentAbout(double point) {
    return nthMomentAbout(point, 2.0);
}

• We already have tests in place, so we can be confident in this change

It’s OK to duplicate!

• This example seems draconian
  – Adding methods that we know are wrong!
  – Copying a method outright!
• But the point of TDD is that we are freed from worrying about more than one thing at a time
  – We might be setting up a test
  – Or we might be writing code for a new feature, but never at the same time as setting up a test
  – Or we might be refactoring away some horrible duplication, but never at the same time as setting up a test or writing new code
  – Do one thing at a time, and do it right
• The tests we build up make later changes and additions much less risky

How we’ll learn TDD

• How does a specification become a series of tests?
  – Literally, we go sentence by sentence, translating the entire thing into a set of tests
• In the first two projects
  – I’ll give you the tests for each step
  – You’ll submit code making those tests — and only those tests — pass
• Over the semester, you’ll take responsibility for both the test and the primary code