AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

# Correctness and Testing

## Correctness

Here's our linear search implementation from before:

```
public int search(int[] a, int target) {
   int i = 0;
   while ((i < a.length) && (a[i] != target))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

What we would like to say: "**This is correct.**"

A difficult claim to make!


Dijkstra


Hoare

What we sometimes end up saying:


**60% of the time, it works every time.**

## Doing better

**Use a systematic, disciplined approach to developing software.**

A “personal software process” is outside the scope of this course, but time management, goal setting, and scheduling milestones will go a long way.

A personal workflow and a “daily build” mentality are good tools too.

**Write clean, simple code.**

Make the code as simple as possible. Adopt a “clean straight line” aesthetic for your code.

**Write rigorous, thorough tests.**

Your code should be well-tested before you submit it for grading.

Writing good tests requires skill, practice, and experience.

## Clean, simple code

*"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." – Tony Hoare*

## Code structure

Code structure affects clarity, understandability, and, at least indirectly, correctness.

```
public int search(int[] a, int target) {
   int i = 0;
   while ((i < a.length) && (a[i] != target))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

```
public int search(int[] a, int target) {
   for (int i = 0; i < a.length; i++) {
      if (a[i] == target)
         return i;
   }
   return -1;
}
```

Which version is easier for you to understand and reason about?

Which version just looks better to you?

## Code structure

Code structure affects clarity, understandability, and, at least indirectly, correctness.

```
public static int min1(int a, int b, int c) {
   if ((a < b) && (a < c)) {
      return a;
   }
   if ((b < a) && (b < c)) {
      return b;
   }
   return c;
  }
```

Which version is easier for you to understand and reason about?

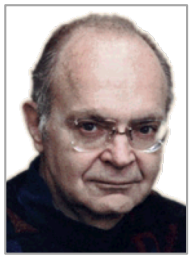
Which version just looks better to you?

Notice the **error** in **min1** and be sure that you understand how to correct it.

```
public static int min2(int a, int b, int c) {
   if (a < b) {
      if (a < c) {
         return a;
      }
      else if (c < a) {
         return c;
      }
      else {
         return a;
      }
   }
   else {
      if (b < c) {
         return b;
      }
      else if (c < b) {
         return c;
      }
      else {
         return b;
      }
   }
}
```

## Rigorous, thorough tests

*"Testing can be used to show the presence of bugs but never to show their absence." – Edsger Dijkstra*



*"Beware of bugs in the above code; I have only proved it correct, not tried it." – Donald Knuth*

## Perspectives on testing

[Adapted from *Software Testing Techniques*, 2nd Edition, by Boris Beizer, Van Nostrand Reinhold, 1990]


**Phase 0** Testing == Debugging

**Phase 1** Testing is an act whose purpose is to show that the software works.

**Phase 2** Testing is an act whose purpose is to show that the software does not work.

*This isn't a completely bad way of thinking about things. If this is all you can muster for this course, no problem. At least you'll have the right mindset: break something.*

**Phase 3** Testing is an act whose purpose isn't to show or prove anything, but to reduce the perceived risk of failure to an acceptable level.

**Phase 4** Testing is not an act; rather, it is a mindset that involves development and coding practices along with a systematic approach to exercising software.

## Test case design

We should write test cases for each of the following categories:

| | |
|---|---|
| **Illegal Cases** | illegal states for data, data structure, etc. |
| **Boundary Cases** | beginning, end, “edges” of data ranges, data structures, etc. |
| **Typical Cases** | normal or expected range of data, normal states, “middle” cases |
| **Special Cases** | unusual or exceptional values or states, perhaps not commonly considered |

## Example: search

```
/**
 * This method uses the linear search algorithm to return
 * the index of the first occurrence of target in a.
 *
 * @param   a  the array to be searched through
 * @param   target  the value to be searched for
 * @return  the index of the first occurrence of target in a
 *          or -1 if a does not contain target
 */
public static int search(int[] a, int target) { ... }
```

| Illegal Cases | Boundary Cases | Typical Cases | Special Cases |
|---|---|---|---|
| a == null | a.length = 1, 2 | a.length = N | duplicates |
| a.length = 0 | max/min int values | | order |

found v. not found

## Example: search

**Illegal Cases**

a == null

a.length = 0

```
@Test(expected = IllegalArgumentException.class)
public void testSearch_array_null() {
   LinearSearch.search(null, 1);
}


@Test
public void testSearch_array_len_0() {
   int[] a = new int[0];
   try {
      LinearSearch.search(a, 1);
      Assert.fail("Did not throw IllegalArgumentException.");
   }
   catch (IllegalArgumentException e) {
      Assert.assertTrue(true); // pass
   }
   catch (Exception e) {
      Assert.fail("Threw incorrect exception.");
   }
}
```

## Example: min of three

```
/**
 * Returns the minimum values of its three parameters.
 *
 * @param a the first value
 * @param b the second value
 * @param c the third value
 * @return  the minimum of a, b, and c
 *
 */
public static int min(int a, int b, int c) { . . .}
```

| Illegal Cases | Boundary Cases | Typical Cases | Special Cases |
|---|---|---|---|
| None | Integer.MIN_VALUE<br>Integer.MAX_VALUE | random int values | duplicates<br>order |

## Example: min of three

**Special Cases: Accounting for duplicates and order**

Let 0 = min, 1 = not min

| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| ~~1~~ | ~~1~~ | ~~1~~ |

Each row corresponds to one or more test cases.

```
// 000
@Test
public void test_min1_1() {
   int expected = 2;
   int actual = MinOfThree.min1(2, 2, 2);
   Assert.assertEquals(expected, actual);
}

// 001
@Test
public void test_min1_2() {
   int expected = 2;
   int actual = MinOfThree.min1(2, 2, 4);
   Assert.assertEquals(expected, actual);
}
```

*Note that the 001 test is the only one that will expose the error in min1.*

## Example: min of three

**Special Cases: Accounting for duplicates and order**

Let 0 = min, 1 = not min

| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| ~~1~~ | ~~1~~ | ~~1~~ |

Each row corresponds to one or more test cases.

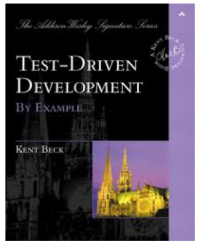
```
// 011
@Test
public void test_min1_4() {
   int expected = 2;
   int actual = MinOfThree.min1(2, 4, 6);
   Assert.assertEquals(expected, actual);
}

 // 011
@Test
public void test_min1_5() {
   int expected = 2;
   int actual = MinOfThree.min1(2, 6, 4);
   Assert.assertEquals(expected, actual);
}
```

## Test early and often

*More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded – indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest. – Boris Bezier*

Bezier's observation is the basis of an approach to writing code called **Test-Driven Development** (TDD).



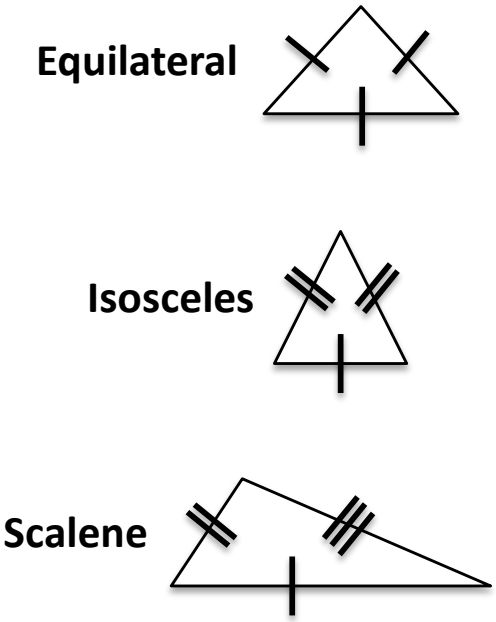

*Kent Beck*

Test-driven development or test-first programming is based on a valuable, common sense idea:

***Plan how to break something before you build it.***

## Test-first example

```
/**
  * Classifies an implied Euclidean triangle. This method takes
  * three ints, representing the length of the three sides
  * of a possible triangle and returns the appropriate string:
  * "equilateral", "isosceles", "scalene", or "not a triangle".
  *
  * @param a  length of one side of a possible triangle
  * @param b  length of one side of a possible triangle
  * @param c  length of one side of a possible triangle
  * @return   a string representing the appropriate classification
  *           "equilateral", "isosceles", "scalene", or "not a triangle"
  *
  */
  public static String triangle(int a, int b, int c) {
     String result = "";
     return result;
  }
```




In Euclidean geometry, a triangle is uniquely defined by any three non-collinear points. These points are the vertices of the triangle, and the line segments that connect each pair of vertices form the sides of the triangle. Note that since the length of the sides represent the distance between the two connected vertices, the sum of the length of any two sides must be greater than the length of the third side.

## Writing the tests

```
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class TriangleClassifierTest {

   @Test public void triangleTestSide1Zero() {
      String expected = "Not a triangle";
      String actual = TriangleClassifier.triangle(0, 4, 5);
      Assert.assertEquals("triangle(0, 4, 5)", expected, actual);
   }

   @Test public void triangleTestSide1Negative() {
      String expected = "Not a triangle";
      String actual = TriangleClassifier.triangle(-1, 4, 5);
      Assert.assertEquals("triangle(-1, 4, 5)", expected, actual);
   }


}
```

## Writing the tests

```
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class TriangleClassifierTest {

   @Test public void triangleTestSide1TooLong() {
      String expected = "Not a triangle";
      String actual = TriangleClassifier.triangle(10, 4, 5);
      Assert.assertEquals("triangle(10, 4, 5)", expected, actual);
   }

   And you just keep going …

}
```

## Writing the tests

```
public class TriangleClassifierTest {

    /** Not-a-triangle tests */

    /** Equilateral tests */

    /** Isosceles tests */

    /** Scalene tests */

}
```

**9 tests**

**2 tests**

**3 tests**

**2 tests**

**16 total tests**

## Writing the code

```
/** Classifies an implied Euclidean triangle. ... */
public static String triangle(int a, int b, int c) {
   String result = "";
   return result;
}
```

*Run this stub against the tests.*

```
----jGRASP exec: java org.junit.runner.JUnitCore TriangleClassifierTest
JUnit version 4.8.1
.E.E.E.E.E.E.E.E.E.E.E.E.E.E.E.E
Time: 0.012
There were 16 failures:
...
```

**Our code failed all 16 tests, as expected.**

**Next step:** Select particular tests and write just enough code to pass them.

## Writing the code

*Write just enough code to make the method pass the following tests.*

**Not-a-triangle, zero and negative sides (6 tests):**

```
@Test public void triangleTestSide1Zero() {
   String expected = "Not a triangle";
   String actual = TriangleClassifier.triangle(0, 4, 5);
   Assert.assertEquals("triangle(0, 4, 5)", expected, actual);
}
```

```
@Test public void triangleTestSide1Negative() {
   String expected = "Not a triangle";
   String actual = TriangleClassifier.triangle(-1, 4, 5);
   Assert.assertEquals("triangle(0, 4, 5)", expected, actual);
}
```

```
@Test public void triangleTestSide2Zero() {
   String expected = "Not a triangle";
   String actual = TriangleClassifier.triangle(3, 0, 5);
   Assert.assertEquals("triangle(3, 0, 5)", expected, actual);
}
```

```
@Test public void triangleTestSide2Negative() {
   String expected = "Not a triangle";
   String actual = TriangleClassifier.triangle(3, -1, 5);
   Assert.assertEquals("triangle(0, 4, 5)", expected, actual);
}
```

```
@Test public void triangleTestSide3Zero() {
   String expected = "Not a triangle";
   String actual = TriangleClassifier.triangle(3, 4, 0);
   Assert.assertEquals("triangle(3, 4, 0)", expected, actual);
}
```

```
@Test public void triangleTestSide3Negative() {
   String expected = "Not a triangle";
   String actual = TriangleClassifier.triangle(3, 4, -1);
   Assert.assertEquals("triangle(0, 4, 5)", expected, actual);
}
```

## Writing the code

```
 /** Classifies an implied Euclidean triangle. ... */
 public static String triangle(int a, int b, int c) {
    String result = "";

    /** Not a triangle */
    if ((a <= 0) || (b <= 0) || (c <= 0)) {
       result = "Not a triangle";
    }
    return result;
}
```

*Run this version against the tests.*

```
----jGRASP exec: java org.junit.runner.JUnitCore TriangleClassifierTest
JUnit version 4.8.1
...E...E...E.E.E.E.E.E.E.E
Time: 0.013
There were 10 failures:
...
```

**6 down, 10 to go.**

## Writing the code

```
 /** Classifies an implied Euclidean triangle. ... */
 public static String triangle(int a, int b, int c) {
    String result = "";

    /** Not a triangle */
    if ((a <= 0) || (b <= 0) || (c <= 0)) {
       result = "Not a triangle";
    }

    if ((a >= b + c) || (b >= a + c) || (c >= a + b)) {
       result = "Not a triangle";
    }
    return result;
}
```

*Target three triangle inequality tests.*

```
----jGRASP exec: java org.junit.runner.JUnitCore TriangleClassifierTest
JUnit version 4.8.1
.........E.E.E.E.E.E.E
Time: 0.013
There were 7 failures:
...
```

**9 down, 7 to go.**

**Keep going until all tests pass.**

# Writing the code

```
/** Classifies an implied Euclidean triangle. ... */

public static String triangle(int a, int b, int c) {
   String result = "";

   /** Not a triangle */
   if ((a <= 0) || (b <= 0) || (c <= 0)) {
      result = "Not a triangle";
   }
   if ((a >= b + c) || (b >= a + c) || (c >= a + b)) {
      result = "Not a triangle";
   }
   else {
      /** equilateral */
      if ((a == b) && (b == c)) {
         result = "equilateral";
      }
      /** isosceles */
      else if ((a == b) || (b == c) || (a == c)) {
         result = "isosceles";
      }
      /** scalene */
      else {
         result = "scalene";
      }
   }
   return result;
}
```

**All tests pass!**

Still needs cleaning up …