Algorithms

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

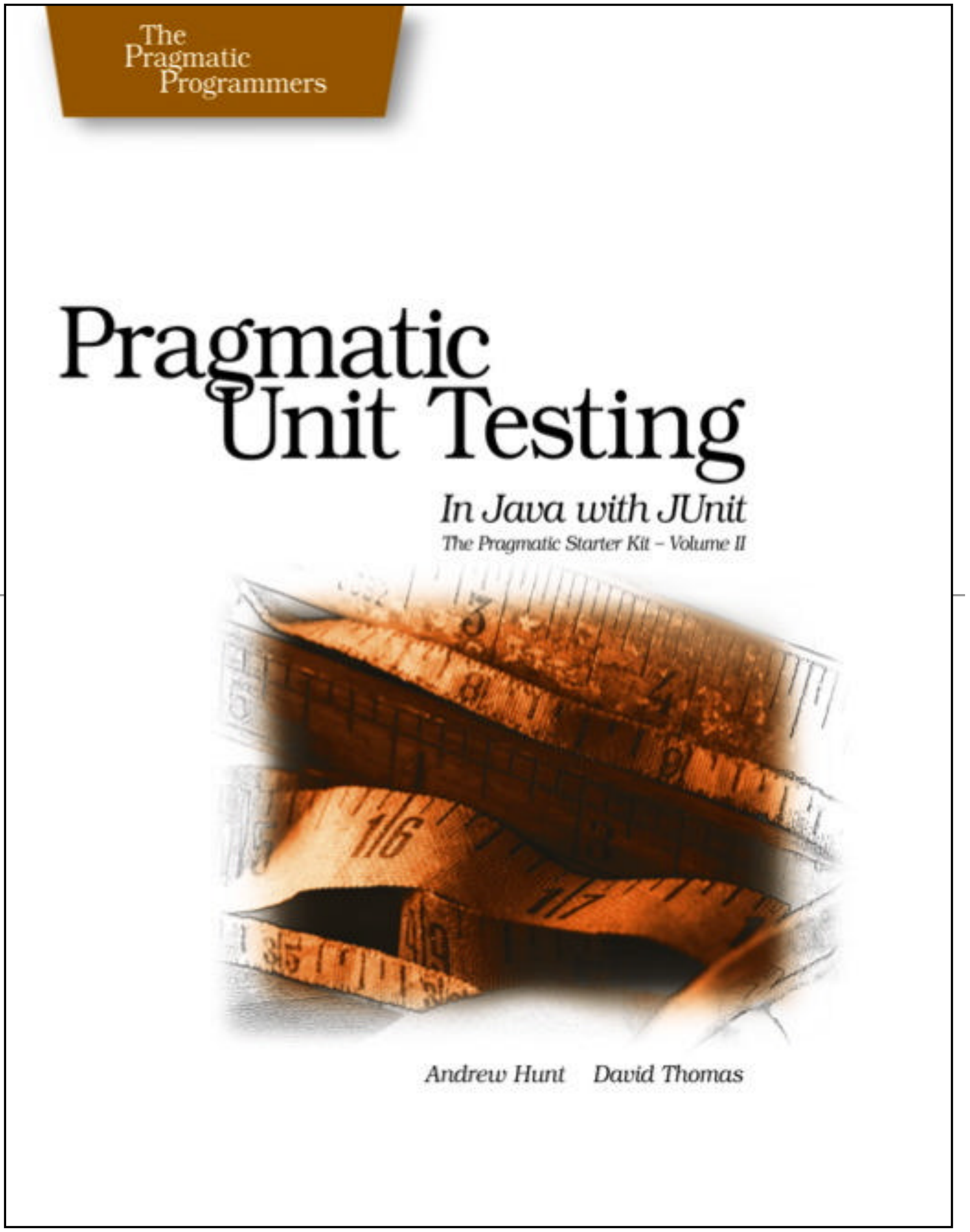
http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

CORRECT Boundary Conditions



Correct Thinking

- The underlying question to be constantly considered is:
    - *What can go wrong?*
- Once you think of something that could go wrong, write a test for it. Once that test passes, again ask
    - *What else can go wrong?*
- and so on.

C.O.R.R.E.C.T.

- **C**onformance - Does the value conform to an expected format?
- **O**rdering - Is the set of values ordered or unordered as appropriate?
- **R**ange - Is the value within reasonable minimum and maximum values?
- **R**eference - Does the code reference anything external that isn't under direct control of the code itself?
- **E**xistence - Does the value exist (e.g., is non-null, nonzero, present in a set, etc.)?
- **C**ardinality - Are there exactly enough values?
- **T**ime (absolute and relative) - Is everything happening in order? At the right time? In time?

Conformance

- When data in a specific format is expected -consider what will happen if the data does not conform to the structure.
- Eg and email address :

  name@somewhere.com

  firstname.lastname@subdomain.somewhere.com

  firstname.lastname%somewhere@subdomain.somewhere.com

  firstname
- How will code react to each of these?
- Similarly, if code is producing data to a specific format, test must verify that the generated data conforms to desired format

Ordering

- Position of one piece of data within a larger collection.
- A search routine should be tested for conditions where the search target is first or last
- For a sort routine, what might happen if the set of data is already ordered? Or sorted in precisely reverse order?

```
public void testOrder ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 8, 7 }));
  assertEquals(9, Largest.largest(new int[] { 8, 9, 7 }));
  assertEquals(9, Largest.largest(new int[] { 7, 8, 9 }));
}

public void testDups ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 7, 9, 8 }));
}

public void testOne ()
{
  assertEquals(1, Largest.largest(new int[] { 1 }));
}

public void testNegative ()
{
  int[] negList = new int[] { -9, -8, -7 };
  assertEquals(-7, Largest.largest(negList));
}

public void testEmpty ()
{
  try
  {
    Largest.largest(new int[] {});
    fail("Should have thrown an exception");
  }
  catch (RuntimeException e)
  {
    assertTrue(true);
  }
}
```

Range (1)

- A variable's type may allow it to take on a wider range of values. e.g. age
- Typically do not use a raw native types to store a bounded-integer values e.g Bearing.
- Encapsulating a bearing within a class yields one point in the system that can filter out bad data

```java
public class Bearing
{
  protected int bearing; // 0..359

  public Bearing(int num_degrees)
  {
    if (num_degrees < 0 || num_degrees > 359)
    {
      throw new RuntimeException("Bad bearing");
    }
    bearing = num_degrees;
  }

  public int angleBetween (Bearing anOther)
  {
    return bearing - anOther.bearing;
  }
}
```

Range (2)

- Two sets of x, y co-ordinates.
- Integers, with arbitrary values, with the constraint that the two points must describe a rectangle with no side greater than 100 units.
- Custom assert might be an option:

```
public static final int MAX_DIST = 100;

public void assertPairRange(String message, Point one, Point two)
{
  assertTrue(message,
  Math.abs(one.x - two.x) <= MAX_DIST);
  assertTrue(message,
  Math.abs(one.y - two.y) <= MAX_DIST);
}
```

Reference (1)

- What things does the method under test reference that are outside the scope of the method itself?
  - external dependencies
  - state
  - other conditions
- Eg.
  - a method in a web application to display a customer's account history might require that the customer is first logged on.
  - the method pop() for a stack requires a nonempty stack.
  - shifting the transmission in a car to Park from Drive requires that the car is stopped.

Reference (2)

- If assumptions are made about
  - the state of the class
  - the state of other objects
  - the global application
- then need to test your code to make sure that it is well-behaved if these assumptions are not met.

```
public void testJamItIntoPark()
{
  transmission.select(DRIVE);
  car.accelerateTo(35);
  assertEquals(DRIVE, transmission.getSelect());
  // should silently ignore
  transmission.select(PARK);
  assertEquals(DRIVE,  transmission.getSelect());
  car.accelerateTo(0); // i.e., stop
  car.brakeToStop();
  // should work now
  transmission.select(PARK);
  assertEquals(PARK, transmission.getSelect());
 }
```

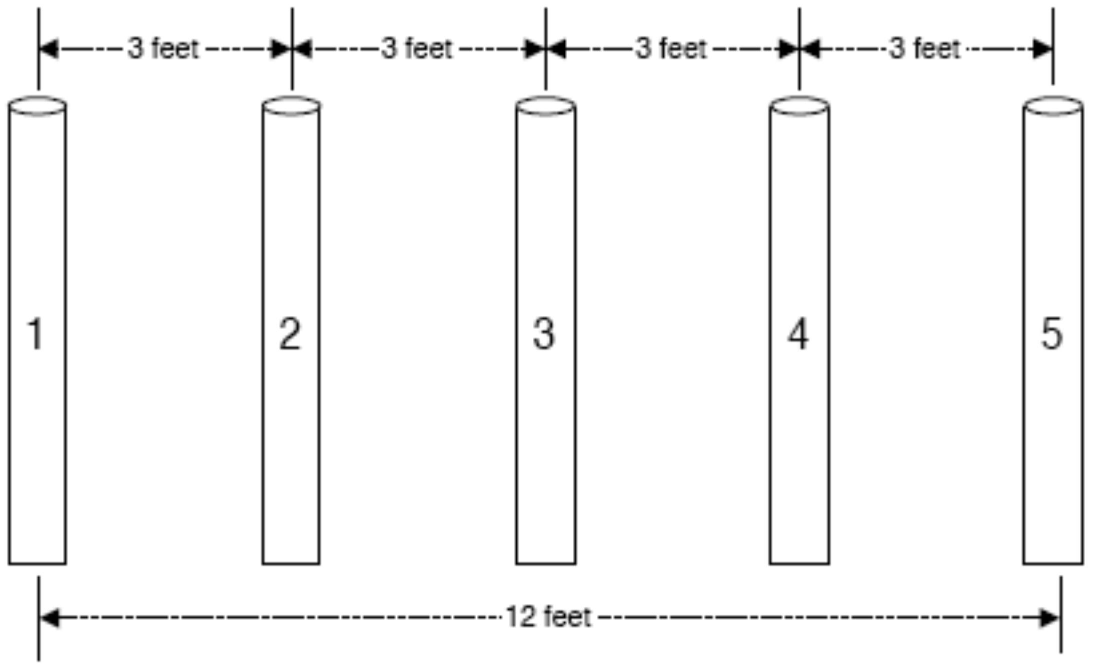
Existence

- Make sure the method under test can stand up to nothing!
  - Network resource, files' URLs, license keys, users, printers may all disappear without notice
- Many Java library methods will throw an exception of some sort when faced with non-existent data.
  - Difficulty: hard to debug a generic runtime exception
  - Exception that reports “Url blank” helps in makes tracking down issue
- Unit test with plenty of nulls, zeros, empty strings etc...

Cardinality (1)

- If you've got 12 feet of lawn that you want to fence, and each section of fencing is 3 feet wide, how many fence posts do you need?

Cardinality (2)



- This problem, and the related common errors, come up so often that they are graced with the name “fencepost errors” or “off-by-one errors”
  - http://en.wikipedia.org/wiki/Off-by-one_error

Cardinality (3)

- Related to *Existence* & *Boundary* - how to make sure there are exactly as many items as needed
- The count of some set of values is most interesting in these three cases:
  - 1. Zero
  - 2. One
  - 3. More than one
- It's called the “0-1-n-Rule” and it's based on the premise that if method can handle more than one of something, it can probably handle 10, 20, or 1,000.
- Sometimes n may be significant -
  - top 10 results
  - leading 100 users

Time

- Relative time (ordering in time)
- Absolute time
- Concurrency issues

Time - Relative

- Some interfaces are inherently stateful:
  - login() will be called before logout()
  - prepareStatement() is called before executeStatement()
  - connect() before read() which is before close()
- Test calling methods out of the expected order try skipping the first, last and middle of a sequence
- Relative time might also include issues of timeouts in the code: how long your method is willing to wait for some resource to become available

Time - Absolute

- The actual “wall clock” time.
- Most of the time, this makes no difference. However, occasionally, the actual time of day will matter.
- e.g: Question: every day of the year is 24 hours long? - true or false?

Time - Absolute

- Answer: It Depends!
- In UTC (Universal Coordinated Time, the modern version of Greenwich Mean Time, or GMT), the answer is YES.
- In areas of the world that do not observe Daylight Savings Time (DST), the answer is YES.
- In most of the U.S. (which does observe DST), the answer is NO.
    - In April, you'll have a day with 23 hours (spring forward) and in October you'll have a day with 25 (fall back).
    - This means that arithmetic won't always work as you expect; 1:45AM plus 30 minutes might equal 1:15, for instance.

Time - Concurrency

“Most code you write in Java will be run in a multi-threaded environment”

- Is this true?
  - Simple Console Application?
  - RMI application?
  - Swing GUI Application?
  - Spring Web Application?
- What will happen if multiple threads use this same object at the same time? Are there global or instance level data or methods that need to be synchronized?
- How about external access to files or hardware?

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit