Agile Software Development

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

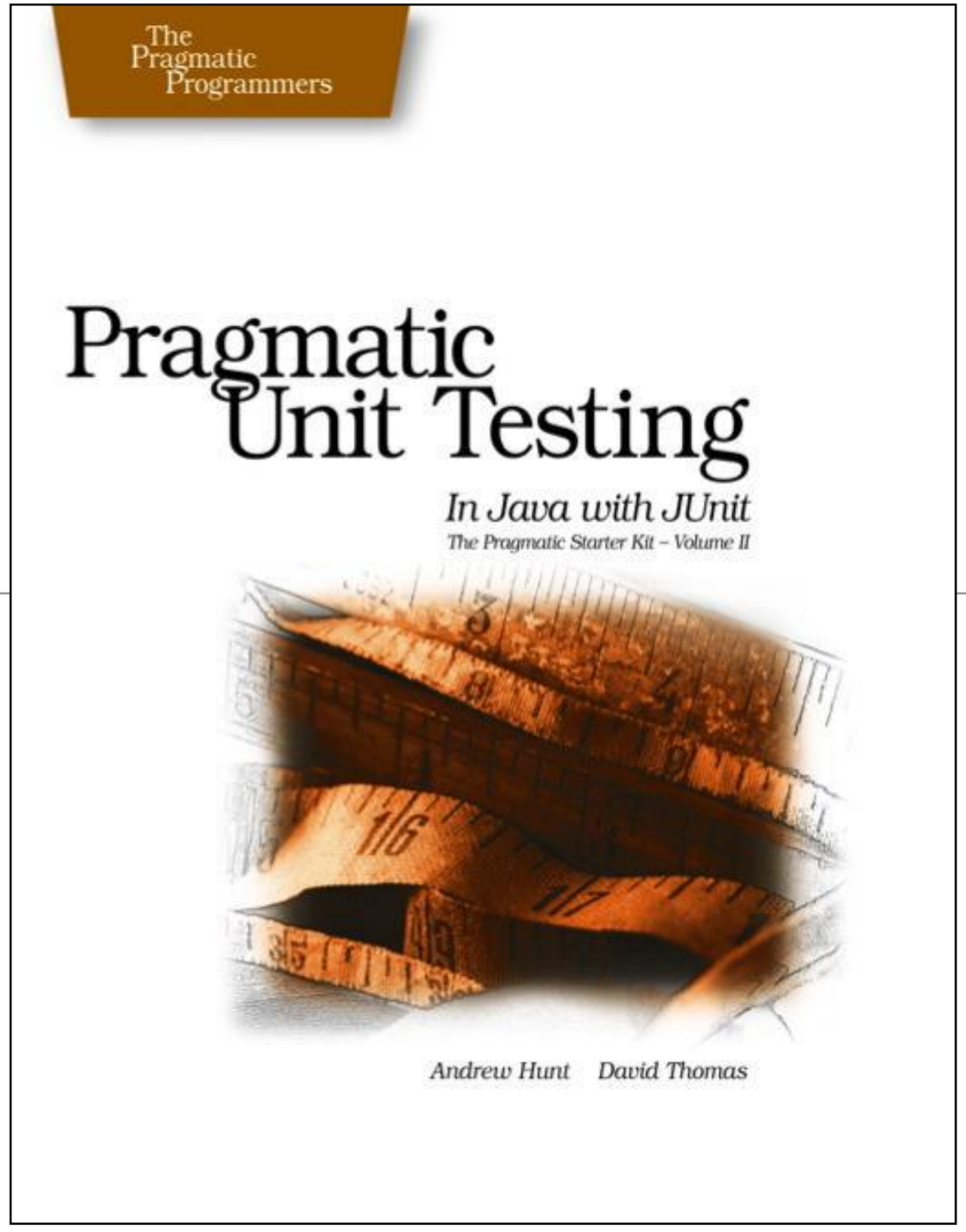
http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

“The Right-BICEP”



Right B.I.C.E.P.

- Guidelines of some areas that might be important to test:
  - **Right** - Are the results right?
  - **B** - Are all the boundary conditions CORRECT?
  - **I** - Can you check inverse relationships?
  - **C** - Can you cross-check results using other means?
  - **E** - Can you force error conditions to happen?
  - **P** - Are performance characteristics within bounds?

Right

- Key question : *If the code ran correctly, how would the developer know?*
  - If this question cannot be answered satisfactorily, then writing the code or the test may be a complete waste of time.
- Does that mean code cannot be written until all the requirements are in?
  - No. If the requirements are truly not yet known, or complete, you the developer will extrapolate as a stake in the ground.
    - They may not be correct from the user's point of view, but the developer now knows what he/she thinks the code should do, and so you can answer the question.
- The definition of correct may change over the lifetime of the code in question, but at any point, developer should be able to prove that it's doing what he/she thinks it should be doing.

B. Boundary Conditions

- Identifying boundary conditions is one of the most valuable parts of unit testing, because this is where most bugs generally live - at the edges.

```
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);
  }
}
```

Example Boundaries:

- Totally bogus or inconsistent input values, such as a file name of "!*W:Xn&Gi/w>g/h#WQ@".
- Badly formatted data, such as an e-mail address without a top-level domain ("fred@foobar.").
- Empty or missing values (such as 0, 0:0, "", or null).
- Values far in excess of reasonable expectations, such as a person's age of 10,000 years.
- Duplicates in lists that shouldn't have duplicates.
- Ordered lists that aren't, and vice-versa. Try handing a pre-sorted list to a sort algorithm, for instance, or even a reverse-sorted list.
- Things that arrive out of order, or happen out of expected order, such as trying to print a document before logging in.

Boundary Conditions 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?

I. Check Inverse Relationships

- Some methods can be checked by applying their logical inverse.

- e.g. check a method that calculates a square root by squaring the result, and testing that it is tolerably close to the original number:

```
public void testSquareRootUsingInverse()
{
   double x = mySquareRoot(4.0);
   assertEquals(4.0, x * x, 0.0001);
}
```

- or - check that some data was successfully inserted into a database by then searching for it.

C. Cross-check Using Other Means

- Where possible, use a different source for the inverse test (bug could be in original and in inverse).

- Usually there is more than one way to calculate some quantity;
  - pick one algorithm over the others because it performs better, or has other desirable characteristics - use that one in production.
  - use one of the other versions to cross-check our results in the test system.

- Especially helpful when there's a proven, known way of accomplishing the task that happens to be too slow or too complex to use in production code.

```
public void testSquareRootUsingStd()
{
   double number = 3880900.0;
   double root1 = mySquareRoot(number);
   double root2 = Math.sqrt(number);
   assertEquals(root2, root1, 0.0001);
}
```

C. Cross-check Using Other Means (2)

Another example - a library database system:

- The number of copies of a particular book should always balance:

  e.g. number of copies that are checked out + number of copies sitting on the shelves should always equal the total number of copies.

- These are separate pieces of data, and they may even be reported by objects of different classes, but they still have to agree, and so can be used to cross-check one another.

E. Force Error Conditions

- In the real world, errors happen: disks fill up, network lines drop, e-mail goes down, and programs crash. Developer should test that code handles many of these real world problems by forcing errors to occur.
- That's easy enough to do with invalid parameters and the like, but to simulate specific network errors without unplugging any cables takes some special techniques.
- For instance:
  - Running out of memory.
  - Running out of disk space.
  - Network availability and errors.
  - System load.
  - Limited color palette.
  - Very high or very low video resolution.

P. Performance Characteristics

- Performance characteristics - does not necessarily mean measuring performance itself - but rather performance trends as input sizes grow, as problems become more complex.

- The approach is not to objectively measure performance, but to incorporate general tests just to make sure that the performance curve remains stable.

Performance example

- A filter that identifies web sites to block.
- The code may works well with a few dozen sample sites, but will it work as well with 10,000? 100,000.
- This test may take 6-7 seconds to run, so may run only nightly.
- See JUnitPerf for tools to simplify such tests.

```
public void testURLFilter()
{
  Timer timer = new Timer();
  String naughty_url = "http://www.xxxxxxxxxxx.com";

  // First, check a bad URL against a small list
  URLFilter filter = new URLFilter(small_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 1.0);

  // Next, check a bad URL against a big list
  URLFilter f = new URLFilter(big_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 2.0);

  // Finally, check a bad URL against a huge list
  URLFilter f = new URLFilter(huge_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 3.0);
}
```