Extracted from:

Pragmatic Unit Testing in Java 8 with JUnit

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Unit Testing in Java 8 with JUnit

Jeff Langr

with Andy Hunt
& Dave Thomas

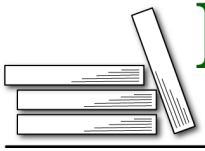
edited by
Susannah Davidson Pfalzer

Pragmatic Unit Testing in Java 8 with JUnit

Jeff Langr

with Andy Hunt
Dave Thomas

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Eileen Cohen (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-94122-259-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2015

It can be hard to look at a method or a class and anticipate all the bugs that might be lurking in there. With experience, you develop a feel for what's likely to break and learn to concentrate on testing those areas first. Until then, uncovering possible failure modes can be frustrating. End users are quite adept at finding our bugs, but that's embarrassing and damaging to our careers! What we need are guidelines to help us understand what's important to test.

Your *Right-BICEP* provides you with the strength needed to ask the right questions about what 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]-BICEP: Are the Results Right?

Your tests should first and foremost validate that the code produces expected results. The arithmetic-mean test in Chapter 1, *Building Your First JUnit Test*, on page ? demonstrates that the `ScoreCollection` class produces the correct mean of `6` given the numbers `5` and `7`. We show it again here.

iloveyouboss/13/test/iloveyouboss/ScoreCollectionTest.java
```
@Test
public void answersArithmeticMeanOfTwoNumbers() {
   ScoreCollection collection = new ScoreCollection();
   collection.add(() -> 5);
   collection.add(() -> 7);

   int actualResult = collection.arithmeticMean();

   assertThat(actualResult, equalTo(6));
}
```

You might bolster such a test by adding more numbers to ScoreCollection or by trying larger numeric values. But such tests remain in the realm of *happy-path* tests—positive cases that reflect a portion of an end-user goal for the software (it could be a tiny portion!). If your code provides the *right* answer for these cases, the end user will be happy.

A happy-path test represents one answer to the important question:

> If the code ran correctly, how would I know?

Put another way: if you don't know how to write a test around the happy path for a small bit of code, you probably don't fully understand what it is you're trying to build—and you probably should hold off until you can come up with an answer to the question.

In fact, some unit testers explicitly ask themselves that question with every unit test they write. They don't write the code until they've first written a test that demonstrates what answer the code should return for a given scenario. Read more about this more disciplined form of unit testing in the chapter on TDD (see Chapter 12, *Test-Driven Development*, on page ?).

"Wait," sez you, "Insisting that I know all the requirements might not be realistic. What if they're vague or incomplete? Does that mean I can't write code until all the requirements are firm?"

Nothing stops you from proceeding without answers to every last question. Use your best judgment to make a choice about how to code things, and later refine the code when answers do come. Most of the time, things change anyway: the customer has a change of mind, or someone learns something that demands a different answer.

The unit tests you write document your choices. When change comes, you at least know how the current code behaves.

Right-[B]ICEP: Boundary Conditions

An obvious happy path through the code might not hit any *boundary conditions* in the code—scenarios that involve the edges of the input domain. Many of the defects you'll code in your career will involve these corner cases, so you'll want to cover them with tests.

Boundary conditions you might want to think about include:

- Bogus or inconsistent input values, such as a filename of "!*W:X\&Gi/w$→>$g/h#WQ@.
- Badly formatted data, such as an email address missing a top-level domain (fred@foobar.).
- Computations that can result in numeric overflow.
- 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 150 years.
- Duplicates in lists that shouldn't have duplicates, such as a roster of students in a classroom.
- Ordered lists that aren't, and vice versa. Try handing a presorted list to a sort algorithm, for instance—or even a reverse-sorted list.
- Things that happen out of expected chronological order, such as an HTTP server that returns an OPTIONS response after a POST instead of before.

The ScoreCollection code from Chapter 1, *Building Your First JUnit Test*, on page ? seems innocuous enough:

iloveyouboss/13/src/iloveyouboss/ScoreCollection.java

```
package iloveyouboss;

import java.util.*;

public class ScoreCollection {
   private List<Scoreable> scores = new ArrayList<>();

   public void add(Scoreable scoreable) {
      scores.add(scoreable);
   }

   public int arithmeticMean() {
      int total = scores.stream().mapToInt(Scoreable::getScore).sum();
      return total / scores.size();
   }
}
```

Let's probe some boundary conditions. Maybe pass a null Scoreable instance:

iloveyouboss/14/test/iloveyouboss/ScoreCollectionTest.java
```
@Test(expected=IllegalArgumentException.class)
public void throwsExceptionWhenAddingNull() {
   collection.add(null);
}
```

The code generates a NullPointerException in the arithmeticMean() method, a bit too late for our tests. We'd rather let the clients know as soon as they attempt to add an invalid value. A guard clause in add() clarifies the input range:

iloveyouboss/14/src/iloveyouboss/ScoreCollection.java
```
public void add(Scoreable scoreable) {
   if (scoreable == null) throw new IllegalArgumentException();
   scores.add(scoreable);
}
```

It's possible that no Scoreable instances exist in the ScoreCollection:

iloveyouboss/14/test/iloveyouboss/ScoreCollectionTest.java
```
@Test
public void answersZeroWhenNoElementsAdded() {
   assertThat(collection.arithmeticMean(), equalTo(0));
}
```

The code generates a divide-by-zero ArithmeticException. A guard clause in add() answers the desired value of 0 when the collection is empty:

iloveyouboss/14/src/iloveyouboss/ScoreCollection.java
```
public int arithmeticMean() {
   if (scores.size() == 0) return 0;
   // ...
}
```

If we're dealing with large integer inputs, the sum of the numbers could exceed Integer.MAX_VALUE. Perhaps we'd like to allow that:

iloveyouboss/14/test/iloveyouboss/ScoreCollectionTest.java
```
@Test
public void dealsWithIntegerOverflow() {
   collection.add(() -> Integer.MAX_VALUE);
   collection.add(() -> 1);

   assertThat(collection.arithmeticMean(), equalTo(1073741824));
}
```

Here's one possible solution:

iloveyouboss/14/src/iloveyouboss/ScoreCollection.java
```
long total = scores.stream().mapToLong(Scoreable::getScore).sum();
return (int)(total / scores.size());
```

The narrowing cast from long down to int gives us pause. Should we probe again with another unit test? No. The add() method constrains the input to int values, and because division by a count always returns a smaller number, it shouldn't be possible to end up with a result larger than an int.

When you design a class, it's entirely up to you whether or not things like potential integer overflow need be a concern in the code. If your class represents an external-facing API, and you can't fully trust your clients, you want to guard against bad data.

However, if the clients are coded by members of your own team (who are also writing unit tests), then you might choose to eliminate the guard clauses and let your clients beware. This is a perfectly legitimate choice and can help minimize the clutter of redundant overchecking of arguments in your code.

If you remove guards, you could warn client programmers with code comments. Better, add a test that documents the limitations of the code:

iloveyouboss/15/test/iloveyouboss/ScoreCollectionTest.java

```
@Test
public void doesNotProperlyHandleIntegerOverflow() {
   collection.add(() -> Integer.MAX_VALUE);
   collection.add(() -> 1);

   assertTrue(collection.arithmeticMean() < 0);
}
```

(You probably don't want to allow unchecked overflow in most systems, however. Better to trap and throw an exception.)

Remembering Boundary Conditions with CORRECT

The CORRECT acronym gives you a way to remember potential boundary conditions. For each of these items, consider whether or not similar conditions can exist in the method that you want to test, and what might happen if these conditions are violated:

- *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 (is it non-null, nonzero, present in a set, and so on)?

- *C*ardinality—Are there exactly enough values?
- *T*ime (absolute and relative)—Is everything happening in order? At the right time? In time?

We'll examine all of these boundary conditions in the next chapter.